1. Abstrakte Klassen

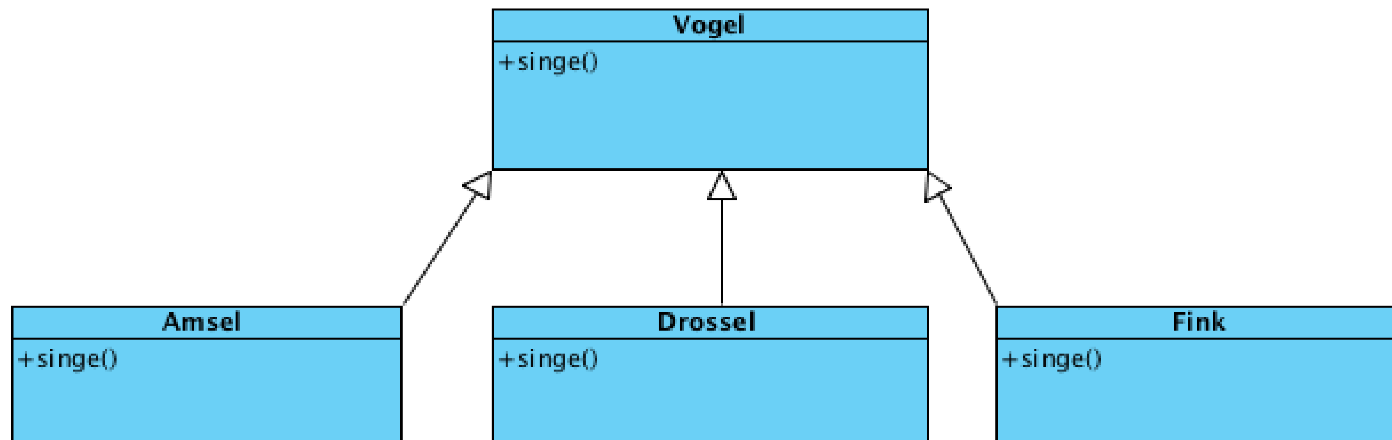
Lernziele:

- Das Konzept abstrakter Klassen und abstrakter Methoden kennen und verstehen,
- in der Lage sein, abstrakte Klassen und Methoden in Java zu formulieren,
- die UML-Notation im Zusammenhang mit abstrakten Klassen beherrschen und
- abstrakte Klassen adäquat bei der Programmentwicklung einsetzen können.

Beispiel: “Alle Vögel sind schon da ...

... Amsel, Drossel, Fink und Star und die ganze Vogelschar.”

Angenommen, wir wollen die folgende Klassenhierarchie implementieren:



UML-Klassendiagramm: *Einführung in die Programmierung (EidP)*, S. 382 ff.

Vorschlag zur Implementierung

```
public class Vogel {
    public void singe() { System.out.println( "Vogel singt: ???"); }
}

public class Amsel extends Vogel {
    public void singe() { System.out.println( "Amsel singt: uiuiii..."); }
}

public class Drossel extends Vogel {
    public void singe() { System.out.println( "Drossel singt: zwawaaa..."); }
}

public class Fink extends Vogel {
    public void singe() { System.out.println( "Fink singt: zrzrrr..."); }
}
```

Bemerkung: Natürlich jede Klasse in einer eigenen Datei *Klassenname*.java

Probleme des Implementierungsvorschlags

(1) In der Realität gibt es keinen Vogel, der "nur" ein Vogel ist. Jeder instanziierte Vogel sollte `Amsel`, `Drossel` oder `Fink` sein.

☞ Nur für die Unterklassen von `Vogel` sollte eine Instanziierung möglich sein.

(2) Ein Programmierer erweitert die Klassenhierarchie um die noch fehlende Klasse `Star` und vergisst dabei, die Methode `singe()` zu überschreiben. Dann haben Stare keine Stimme.

☞ Man sollte eine Möglichkeit haben, bei der Erweiterung der Klassenhierarchie den Entwickler zum Überschreiben zu zwingen.

Probleme des Implementierungsvorschlags (2)

(3) Wenn (1) erfüllt ist und der Programmierer den Fehler von (2) nicht gemacht hat, dann wird die Methode `singe()` in der Klasse Vogel nie aufgerufen.

Wir können die Methode `singe()` in der Klasse `Vogel` aber auch nicht einfach weglassen. Warum nicht?

☞ Auf Methodenimplementierungen, die nicht genutzt werden, sollte man verzichten können.

Abstrakte Klasse

- Problem (1) können wir vermeiden, indem wir eine Klasse als *abstrakt* kennzeichnen.
- Zugehöriges Schlüsselwort in Java: `abstract`

```
[public] abstract class Klassenname ... { ... }
```

- Von abstrakten Klassen können keine Instanzen erzeugt werden.
- D. h.: Wenn `K` eine abstrakte Klasse ist, dann liefert die Anweisung

  ```
  new K();
  ```

  bei der Compilierung einen Fehler.

Konsequenzen

Für unser Beispiel:

- Wenn wir die Klasse `Vogel` als abstrakt kennzeichnen, können wir nur noch Amseln, Drosseln oder Finken instanziieren.
- Problem (1) ist gelöst.

Allgemein für die Definition einer abstrakten Klasse:

- Um Instanzen erzeugen zu können, muss mindestens eine nicht abstrakte Unterklasse zu einer abstrakten Klasse definiert werden.

Abstrakte Klassen in der Klassenhierarchie

- Eine abstrakte Klasse kann Unterklasse einer nicht abstrakten Klasse sein.
- Eine abstrakte Klasse kann weitere abstrakte Klassen als Unterklassen haben.
- Auch für abstrakte Klassen sollten Konstruktoren definiert werden.

  Grund: Die Initialisierung von Objekten erfolgt entlang der Klassenhierarchie (siehe späteres Beispiel zu geometrischen Objekten).
- Eine Klasse kann nicht gleichzeitig final und abstrakt sein.

  Konsequenz: Der Compiler verbietet die Kombination von `abstract` und `final` im Klassenkopf.

Finale Klassen

- Der Modifier `final` schützt Klassen vor Vererbung.
- `[public] final class` *`Klasse`* `{ ... }`
- Wirkung: Von *`Klasse`* kann keine Unterklasse abgeleitet werden.
- Typischerweise sind Klassen, die nur `static` Definitionen enthalten, `final`.
- Beispiel: `java.lang.Math`

Finale Methoden

- `final` vor einer Methode schützt diese Methode vor dem Überschreiben.
- `public final void meineMethode()`
- Finale Methoden verwendet man, wenn man sicherstellen möchte, dass eine Methode nicht verändert wird, z.B. weil Sie kritisch für den Zustand eines Objektes ist.
-

```
class ChessAlgorithm {
    static final int WHITE = 0;
    static final int BLACK = 1;
        ...
    final int getFirstPlayer() {
        return ChessAlgorithm.WHITE;
    }
     ...
}
```

Vereinbarung von Konstanten

- Die Vereinbarung von *Konstanten* erfolgt innerhalb einer Klassendefinition.
- Konstanten sind Klassenvariablen.
- Durch das Schlüsselwort `final` wird die nachträgliche Änderung des Variablenwerts verboten.

Syntax für die Deklaration öffentlicher Konstanten:

```
public static final Datentyp Variablenname = Initialwert;
```

Beispiel: Definition von Konstanten für Studienfächer:

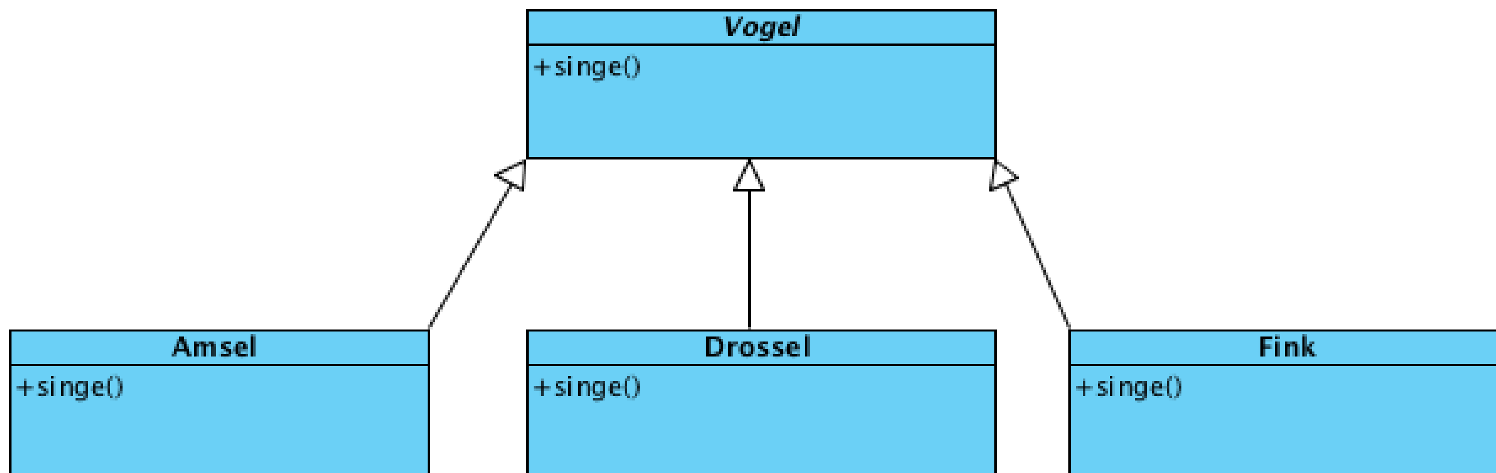
```
public static final int MATHEMATIK_STUDIUM = 1;
public static final int INFORMATIK_STUDIUM = 2;
public static final int BIOLOGIE_STUDIUM = 3;
...
```

Finale Variablen

- finale Klassenvariablen
  entsprechen Konstanten, siehe vorangegangene Folie
- finale Instanzvariablen
  - Variablenwert für eine Instanz muss einmalig durch Initialisierungsausdruck oder in allen Konstruktoren festgelegt werden
  - nicht änderbar
- finale Methodenparameter
  - keine Änderung des Parameterwerts in der Methode zulässig
  - wegen Call-by-Value Parameterübergabe in Java prinzipiell ohne Bedeutung

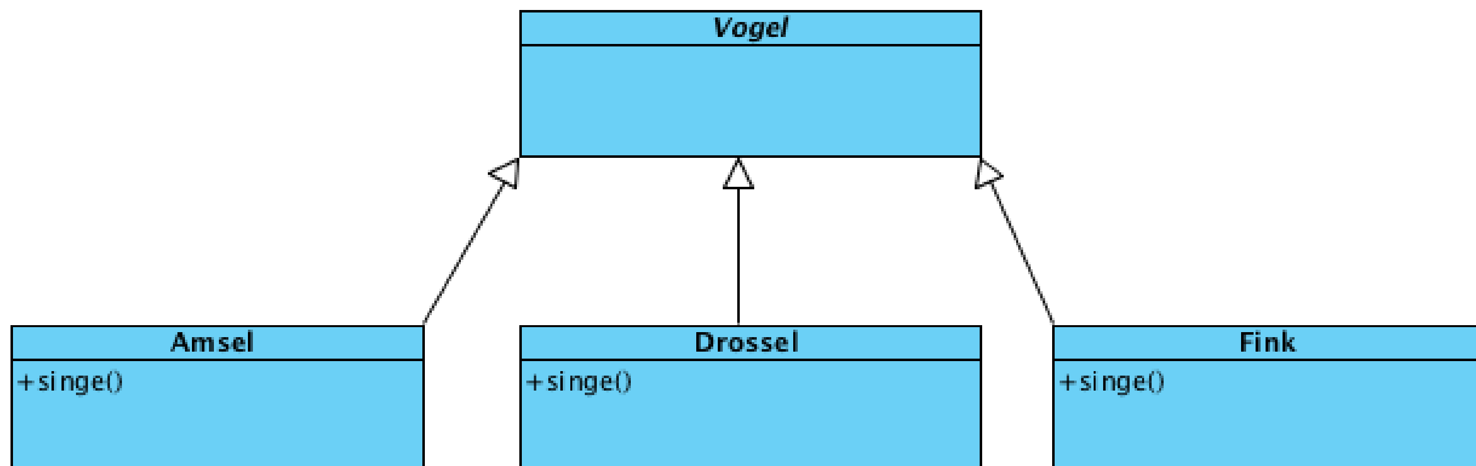
Abstrakte Klasse in UML

Im UML-Klassendiagramm werden abstrakte Klassen durch einen kursiven Klassennamen gekennzeichnet.



Die folgende Modellierung ist keine Lösung, ...

... der Probleme (2) und (3), sondern würde nur zu weiteren Problemen führen, wie die folgenden Folien zeigen.



Vererbung und Methodenausführung

Es sei die Modellierung/Implementierung von Folie 17/18 gegeben:

```
Vogel vogel = new Amsel();
vogel.singe();
```

- Welche Methode `singe()` wird hier ausgeführt?

  Die der Klasse `Vogel`, weil die Variable `vogel` von diesem Typ ist?
  ☞ *statischer Typ* der Variablen `vogel`

  Die der Klasse `Amsel`, weil das Objekt, auf dem `singe()` aufgerufen wird, eine Instanz von `Amsel` ist?
  ☞ *dynamischer Typ* der Variablen `vogel`

☞ Entscheidend für die Methodenauswahl ist der dynamische Typ einer Variablen. (EidP S. 370–374)

Methoden-Polymorphie

- Methodenaufrufe in Java sind polymorph.
- Derselbe Methodenaufruf kann zu unterschiedlichen Zeitpunkten verschiedene Methoden aufrufen.
- Welche Methode tatsächlich aufgerufen wird, ist abhängig vom dynamischen Typ der Variablen, mit der der Aufruf durchgeführt wird.

Weiteres Beispiel:

- Szenario Folie 17/18
- Für `i==0` wird die Methode `singe()` von `Amsel` aufgerufen und
- für `i==1` die Methode `singe()` von `Drossel`.

```
Vogel v;
Vogel v1 = new Amsel();
Vogel v2 = new Drossel();

for (int i=0 ; i<2 ; i++) {
   if (i==0)
      v = v1;
   else
      v = v2;
   v.singe();
}
```

Methodensuche

Wir wollen die Frage, welche Instanzmethode bei einem Methodenaufruf aufgerufen wird, noch etwas genauer betrachten.

Methodensuche:

- Beim Aufruf einer Instanzmethode findet eine sogenannte Methodensuche statt.
- Ausgehend vom dynamischen Typ der Variablen wird in der Vererbungshierarchie nach einer Methode gesucht, die auf die Signatur des Aufrufs passt.
- Wenn die Klasse des dynamischen Typs keine solche Methode aufweist, wird in der direkten Oberklasse gesucht, usw.
- Die Suche endet spätestens in der Klasse `Object`.

Problem der Sichtbarkeit

Die Modellierung auf Folie 29 garantiert zwar (prinzipiell) Polymorphie aber keine Sichtbarkeit der Methode `singe()`.

```
Amsel a = new Amsel();
Vogel v = a;         // kein Problem, Substituierbarkeit
a.singe();           // kein Problem, Methode singe() für Typ Amsel bekannt
v.singe();           // Fehler (Compiler), Methode singe für Typ Vogel unbekannt
                     // --> statische Typisierung
```

Man beachte: `a.singe()` und `v.singe()` würde zum Aufruf der selben Methode führen!

☞ Problem hier: Die Sichtbarkeit von Methoden basiert auf dem statischen Typ einer Variablen.

Abstrakte Methoden

Zur Lösung der Probleme (2) und (3) erklären wir die Methode `singe()` in der Klasse Vogel zu einer *abstrakten Methode*.

- Die Definition einer *abstrakten Methode* besteht aus einer Methodensignatur ohne einen Rumpf.
- Statt eines Blocks als Rumpf folgt dem Methodenkopf ein Semikolon.
- Solch eine Methode wird mit dem Schlüsselwort `abstract` markiert.

Damit haben wir Problem (3) gelöst: Wir brauchen keine Dummy-Implementierung mehr für die Methode `singe()`.

Abstrakte Methoden (2)

Syntax:

Modifikator `abstract` *Datentyp* *Methodenname*`(`*Parameterliste*`);`

Beispiel: Abstrakte Klasse mit einer öffentlichen abstrakten Methode:

```
public abstract class Vogel {
    public abstract void singe();
}
```

Abstrakte Methoden (3)

Die Definition einer abstrakten Methode m in einer Klasse K bedeutet:

- Für die Klasse K und somit auch für alle Unterklassen U von K ist die Methode m bekannt.
- K möchte/kann die Methode nicht implementieren. Dies muss in einer Unterklassen U von K erfolgen.
- Sei U eine (nicht abstrakte) Unterklasse von K. Dann ist folgendes möglich.

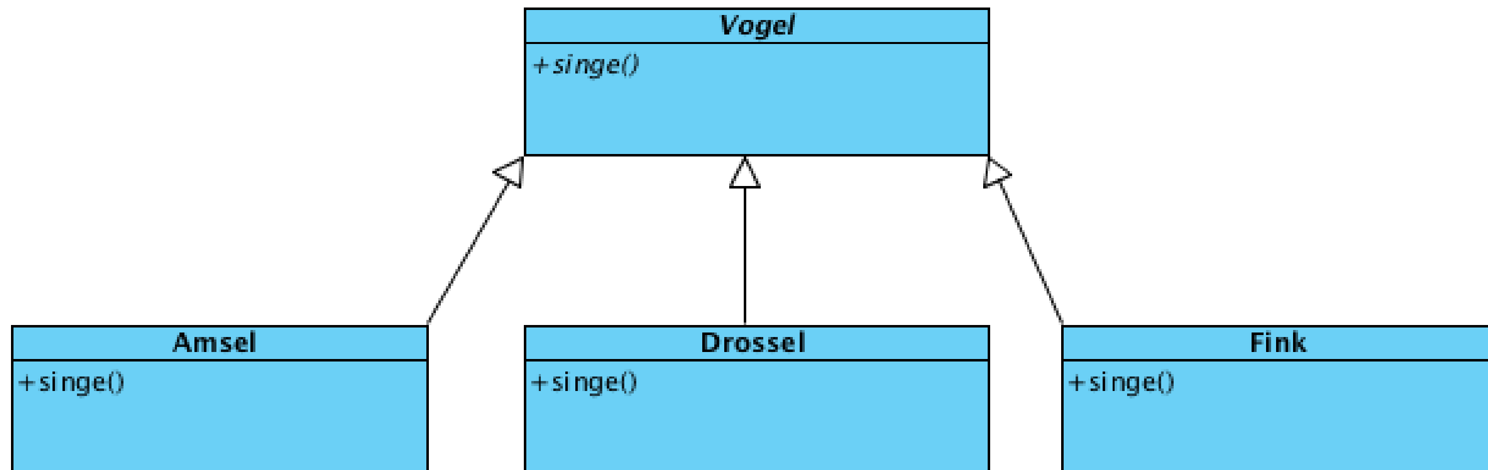
```
K k = new U();

k.m();
```

- Die Methodenauswahl basiert auf dem dynamischen Typ von k.

Abstrakte Methoden in UML

Im UML-Klassendiagramm werden abstrakte Methoden durch eine kursive Schreibweise gekennzeichnet.



Zusammenspiel: Abstrakte Klassen und Methoden

- Abstrakte Klassen dürfen abstrakte Methoden anbieten.
- Jede Klasse, die mindestens eine (evtl. geerbte) abstrakte Methode hat, ist selbst abstrakt und muss entsprechend deklariert werden.
- Damit eine Unterklasse einer abstrakten Klasse eine konkrete Klasse werden kann, muss sie Implementierungen für alle geerbten abstrakten Methoden anbieten.
- Andernfalls ist die Unterklasse selbst abstrakt und muss als solche gekennzeichnet werden.

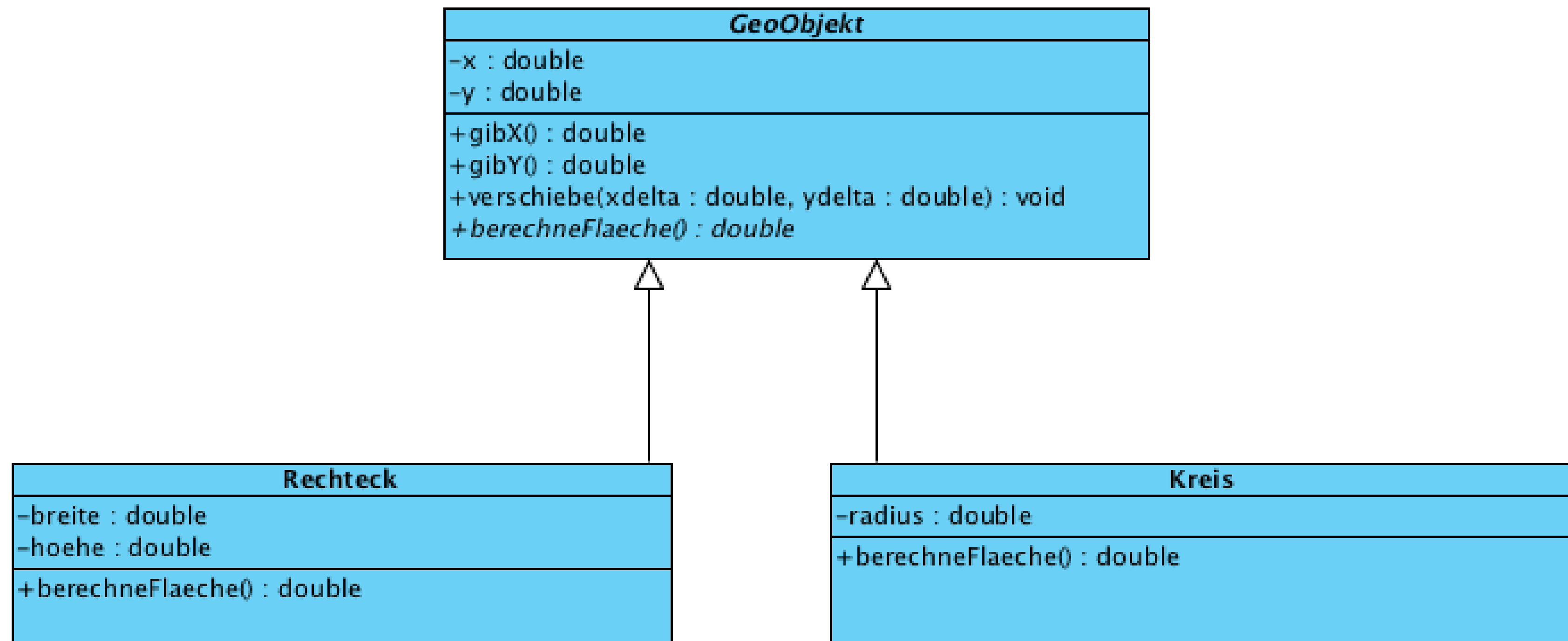
Damit haben wir Problem (2) gelöst:

- Wenn der Programmierer der Klasse `Star` vergessen sollte, die Methode `singe()` zu überschreiben, meldet der Compiler einen Fehler.

Wann setzt man abstrakte Klassen ein?

- Zur Repräsentation einer Generalisierung verschiedener Klassen,
- die Eigenschaften der gleichen Art haben,
- die in den Unterklassen aber unterschiedlich berechnet werden müssen.
- Für die Berechnung der Eigenschaften sehen wir in der Generalisierung jeweils eine abstrakte Methode vor, um deutlich zu machen, dass Objekte dieser Klasse (und der Unterklassen) diese Eigenschaften haben.
- Die jeweilige Art und Weise der Berechnung wird aber erst in den Unterklassen festgelegt.

Beispiel: Geometrische Objekte mit Eigenschaft Flächeninhalt



Beispiel: Geometrische Objekte und deren Eigenschaft Fläche (2)

```
public abstract class GeoObjekt {

    private double x;
    private double y;

    public GeoObjekt(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double gibX() { return this.x; }
    public double gibY() { return this.y; }

    public void verschiebe(double xdelta, double ydelta) {
        this.x += xdelta;
        this.y += ydelta;
    }

    public abstract double berechneFlaeche();
}
```

Beispiel: Geometrische Objekte und deren Eigenschaft Fläche (3)

```
public class Rechteck extends GeoObjekt {

    private double hoehe;
    private double breite;

    public Rechteck(double x, double y, double hoehe, double breite) {
        super(x,y);
        this.hoehe = hoehe;
        this.breite = breite;
    }

    public double berechneFlaeche() {
        return this.hoehe*this.breite;
    }
}
```

Beispiel: Geometrische Objekte und deren Eigenschaft Fläche (4)

```
public class Kreis extends GeoObjekt {

    private double radius;

    public Kreis(double x, double y, double radius) {
        super(x,y);
        this.radius = radius;
    }

    public double berechneFlaeche() {
        return Math.PI * this.radius * this.radius;
    }
}
```

Typumwandlungen

```
Vogel v = new Amsel();  // Upcast
```

Kann man jetzt aus dem Vogel-Objekt `v` wieder ein Amsel-Objekt machen? Ja, mit einer expliziten Typumwandlung.

```
Amsel a = v;           // Fehler (Compiler)
Amsel b = (Amsel) v;   // OK, Downcast
```

- Eine (typischerweise implizite) Typumwandlung von einem Untertyp zu einem Obertyp heißt *Upcast*.
- Eine (nur explizit mögliche) Typumwandlung von einem Obertyp zu einem Untertyp heißt *Downcast*.

Typumwandlungen (2)

```
public class OK { ... }
public class UK1 extends OK { ... }
public class UK2 extends OK { ... }

OK a = new UK1();        // ok, Upcast
OK b = new UK2();        // ok, Upcast
UK1 c = (UK1) a;         // ok, Downcast
UK2 d = (UK2) b;         // ok, Downcast
UK2 e = (UK2) c;         // Fehler, erkennt Compiler
UK2 f = (UK2) a;         // Fehler zur Laufzeit, ClassCastException
```

Typumwandlung bei Feldern

- Die Typhierarchie setzt sich auf Felder fort.
- Wenn `OK` eine Oberklasse von `UK` ist, dann ist `OK[]` ein Obertyp von `UK[]`.
- Dieses Verhalten nennt man bei Programmiersprachen *Kovarianz* (in Kapitel 4 mehr dazu).
- `[]` fungiert hier als sogenannter *Typkonstruktor*.
- Typkonstruktoren ermöglichen die Konstruktion neuer Datentypen (hier: Feld von `OK` bzw. Feld von `UK`) aus bereits vorhandenen Basisdatentypen (hier `OK` bzw. `UK`).

```
Vogel[] vogelschar = new Amsel[20];     // ok, Upcast
vogelschar[0] = new Amsel();            // ok, Upcast
Amsel a = (Amsel) vogelschar[0];        // ok, Downcast
vogelschar[1] = new Drossel();          // Fehler, ClassCastException
```

Der `instanceof`-Operator

- zweistelliger Operator, der einen boolschen Wert liefert
- Syntax:

  `Ausdruck` `instanceof` *`Referenztyp`*

- Liefert true, wenn ein Cast von *`Ausdruck`* zu *`Referenztyp`* möglich ist.
- Hierdurch kann eine ClassCastException vermieden werden.
- Auch als Test möglich, um zu prüfen, ob ein Objekt aus einer bestimmten Unterklasse stammt.

Der `instanceof`-Operator (2)

Beispiel:

```
Vogel[] vogelschar = Vogelschar.erzeugeVogelschar(10);
int     amselzaehler = 0;

for (int i=0 ; i<vogelschar.length ; i++)
    if ( vogelschar[i] instanceof Amsel ) {
        amselzaehler++;
    }
}
System.out.println("Wir haben " + amselzaehler + " Amseln in der Vogelschar.");
```

Zusammenfassung

- Abstrakte Klassen für die Generalisierung, abstrakte Klassen müssen spezialisiert werden
- Abstrakte Methoden für Operationen in abstrakten Klassen, deren Verhalten erst in den Spezialisierungen genau definiert werden kann.
- Für eine konkrete Spezialisierung müssen alle abstrakten Methoden überschrieben werden.
- Typumwandlungen: Up- und Downcast
- Typüberprüfung mit Hilfe des Operators `instanceof`