Vererbung

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Wir wiederholen in diesem Abschnitt zunächst die Grundlagen des Subtyping aus Kapitel 12. Darauf aufbauend werden wir weitere Möglichkeiten und Aspekte des Subtyping in Abschnitt 2 behandeln; im Vordergrund stehen dabei Vererbung und abstrakte Klassen.

Lernziele dieses Kapitels:

- Das Konzept der Vererbung in der OO-Programmierung zu erläutern.
- Vererbung von Attributen und Methoden in Java anzuwenden.
- Information Hiding im Zusammenhang mit Vererbung sinnvoll umzusetzen.

1 Subtyping revisited

Java erlaubt es Programmierern eigene Referenztypen zu deklarieren. Dies geschieht in Form von Interface- und Klassendeklarationen. Die Deklaration eines Typs T legt dabei die direkten Supertypen von T fest.

Subtypbildung in Java: Klassen Die Syntax der Klassendeklaration (bis jetzt) ist zusammenfassend folgende:

```
<Modifikatorenliste> class <Klassenname>
    [  implements <Liste von Interface-Namen> ] {

   <Liste von Attribut-, Konstruktor-,Methodendeklaration>

}
```

Eine Klasse ***implementiert*** ggf. mehrere Interfaces. Die in der `implements`-Klausel genannten Interface-Typen müssen implementiert werden; d.h. für jede vom Interface geforderte Methode muss in der Klassendeklaration zunächst eine Implementierung angegeben werden. Alle implementierten Interface-Typen und der spezielle Typ `Object` sind Supertypen der deklarierten Klasse.

Subtypbildung in Java: Interfaces Hier die Syntax der Interface-Deklaration:

```
<Modifikatorenliste> interface <Schnittstellenname>
    [  extends <Liste von Schnittstellennamen> ] {

   <Liste von Methodensignaturen (und Konstanten)>

}
```

Bei einer Interface-Deklaration gilt:

- Gibt es keine `extends`-Klausel, ist `Object` der einzige Supertyp.
- Andernfalls sind die in der `extends`-Klausel genannten Interface-Typen die direkten Supertypen. Alle geforderten Methoden eines Supertyps werden auch vom Subtyp gefordert. Sie müssen aber nicht erneut aufgelistet werden.

Die Subtyprelation ist im übrigen

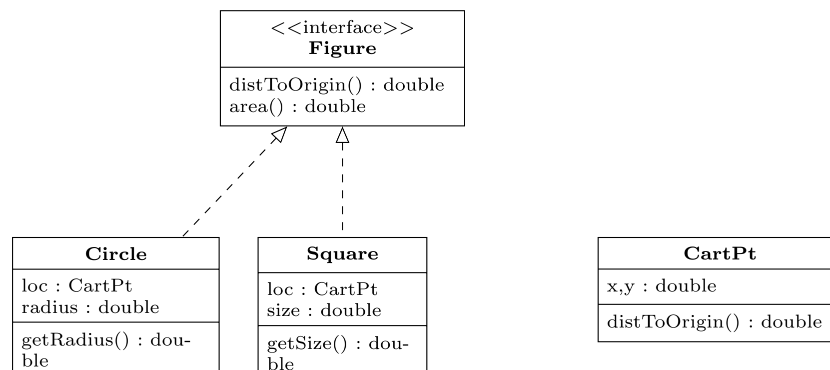
- reflexiv (d.h. T ist ein Subtyp von T) und
- transitiv (d.h. wenn T ein Subtyp von S ist und S ein Subtyp von U, dann ist T auch ein Subtyp von U).

1.1 Beispiel: Geometrische Figuren

In einem Zeichenprogramm sollen verschiedene geometrische Figuren in einem zweidimensionalen Koordinatensystem dargestellt werden (Einheit: ein Pixel). Wir betrachten zunächst zwei Arten von Figuren:

- Kreise mit dem Mittelpunkt als Referenzpunkt und gegebenem Radius
- Quadrate mit Referenzpunkt links oben und gegebener Seitenlänge

Diese Figuren können wir nun in einem objekt-orientierten Modell abbilden:



Zwischen `Figure` und `Circle`, `Square`, `Dot` besteht eine “ist ein”-Beziehung: Ein Kreis ist eine geometrische Figur. Alle Eigenschaften, die geometrische Figur haben sollen, müssen von Kreisen erfüllt werden. Beispielsweise muss man für Kreise den Abstand zum Ursprung und die Fläche ermitteln können. Kreise stellen zusätzliche weitere Methoden bereit (`getRadius()`). Desweiteren benötigen wir noch Referenzpunkte im Koordinatensystem (`CartPt`) zur Modellierung. Diese werden als Attribute für die Figuren genutzt, sind aber nicht Teil ihrer Typhierarchie.
Das objekt-orientierte Modell für Figuren können wir in Java folgendermaßen implementieren:

- `Figure` ist ein Interface.
- `Circle` und `Square` sind Klassen, die das Interface `Figure` implementieren.

```
1  // Interface fuer Geometrische Figuren
2  interface Figure {
3    // ermittelt den Abstand zum Ursprung
4    double distToOrigin();
5    // ermittelt die Flaeche
6    double area();
7  }
```

```
1  class Circle implements Figure {
2    private CartPt loc; // Referenzpunkt ist Mittelpunkt
3    private double radius;  // Radius
4
5    Circle (CartPt loc, double radius) {
6      this.loc = loc;
7      this.radius = radius;
8    }
9
10   public double area() {
11     return radius * radius * Math.PI;
12   }
13
14   public double distToOrigin() {
15     return Math.max(loc.distToOrigin() - radius,0);
16   }
17
18   public double getRadius() {
19     return radius;
20   }
21 }
```

```
1  class Square implements Figure {
2    private CartPt loc;  // Referenzpunkt ist linke untere Ecke
3    private double size;  // Seitenlaenge
4
5    Square (CartPt loc, double size) {
6      this.loc = loc;
7      this.size = size;
8    }
9
```

```
10    public double area() {
11      return size * size;
12    }
13
14    public double distToOrigin() {
15      return loc.distToOrigin();
16      // Mathematisch korrekt nur fuer Referenzpunkte mit
17      // x >= 0 und y >= 0 (siehe Uebung)
18    }
19
20    public double getSize() {
21      return size;
22    }
23  }
```

Die Referenzpunkte werden dargestellt durch Objekte der Klasse `CartPt`:

```
1   // Punkt im zweidimensionalen kartesischen Koordinatensystem
2   class CartPt {
3     private double x, y;
4
5     CartPt(double x, double y) {
6       this.x = x;
7       this.y = y;
8     }
9     double distToOrigin() {
10      return Math.sqrt(x * x + y * y);
11    }
12  }
```

Frage 1:

- `Square`-Objekte sind vom Typ ....
- `CartPt`-Objekte sind vom Typ ....

1.2 Prinzip der Substituierbarkeit

In einer objektorientierten Programmiersprache mit Subtyping wie Java gilt:

> Sei S ein Subtyp von T, $S \leq T$. Dann ist an allen Programmstellen, an denen ein Objekt vom Typ T zulässig ist, auch ein Objekt vom Typ S zulässig.

Die geometrischen Figuren lassen sich nun beispielsweise in einer Liste sammeln und verarbeiten.

Beispiel Ermitteln der Figur mit der größten Fläche:

```
// requires !l.isEmpty()
Figure getLargestFigure(List<Figure> l) {
  Figure f = l.get(0);
```

```
    for (Figure k : l) {
      if (f.area() < k.area()) {
        f = k;
      }
    }
    return f;
  }
```

Die Elemente dieser Liste müssen vom Typ `Figure` sein; sowohl Quadrate als auch Kreise sind hier zulässig:

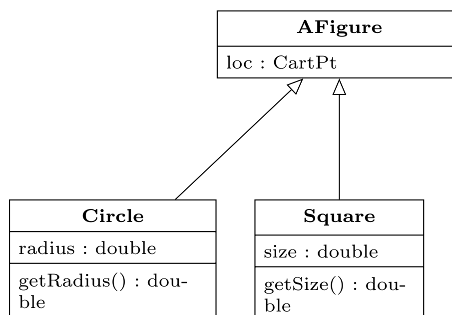
```
List<Figure> l = new ArrayList<Figure>();
l.add(new Circle(new CartPt(1.0,2.0), 5.0));
l.add(new Square(new CartPt(4.0,5.0), 2.0));
l.add(new Circle(new CartPt(1.0,0.0), 1.0));
l.add(new Circle(new CartPt(1.0,-2.0), 9.0));

Figure x = getLargestFigure(l);
```

Frage 2: Welcher der folgenden Ausdrücke mit obiger Variable `x` ist zulässig?

- `x.getRadius()`
- `x.area()`

2 Vererbung

Abstraktion bezeichnet die Verallgemeinerung des konkreten Einzelfalls, wobei generelle Strukturmerkmale betont werden. In der objekt-orientierten Software-Entwicklung ist die Extraktion der gemeinsamen Eigenschaften unterschiedlicher Objekte / Klasse ein wichtiges Abstraktionsverfahren, um Code-Duplikation zu vermeiden.
Dabei gilt es zunächst Programmfragmente (Attribute und Methoden) mit ähnlicher Bedeutung oder Struktur zu finden. In einem zweiten Schritt wird eine Klasse erstellt, die die gemeinsamen Eigenschaften zusammenfasst.
Im Beispiel mit den geometrischen Figuren sieht man, dass alle geometrischen Figuren einen Referenzpunkt besitzen (Attribut `loc`). Dieser lässt sich in eine Klasse `AFigure` abstrahieren.



Vererbung (engl. *inheritance*) im engeren Sinne bedeutet, dass eine Klasse Programmteile von einer anderen übernimmt. Die erbende Klasse heißt ***Subklasse***, die vererbende Klasse heißt ***Superklasse***.
In Java sind die ererbten Programmteile Attribute und Methoden; nicht vererbt werden Konstruktoren sowie statische (Klassen)Attribute und statische (Klassen-)Methoden.
Spezialisierung bedeutet das Hinzufügen speziellerer Eigenschaften zu einem Gegenstand oder das Verfeinern eines Begriffs durch Einführen weiterer Merkmale (z.B. berufliche Spezialisierung).
Vererbung unterstützt Spezialisierung durch:

- Hinzufügen von Attributen (***Zustandserweiterung***)
- Hinzufügen von Methoden (***Erweiterung der Funktionalität***)
- Anpassen, Erweitern bzw. Reimplementieren von Supertyp-Methoden (***Anpassen der Funktionalität***)

Vererbung von Attributen Im Beispiel können wir die Vererbung von Attributen so umsetzen:

```
class AFigure {
  protected CartPt loc;
  AFigure(CartPt loc) {
    this.loc = loc;
  }
}
class Square extends AFigure { ... }
class Circle extends AFigure { ... }
```

Der Konstruktor der `AFigure`-Klasse enthält und initialisiert nur das `loc`-Attribut. `Square`-Objekte haben zusätzlich ein `size`-Attribut. Dieses Attribute muss im Konstruktor der `Square`-Klasse initialisiert werden. Die Initialisierung von `loc` wird an die Superklasse delegiert.

```
class Square extends AFigure {
  private double size;
  Square(CartPt loc, double size) {
    super(loc);
    this.size = size;
  } ...
}

class Circle extends AFigure {
  private double radius;
  Circle(CartPt loc, double radius) {
    super(loc);
    this.radius = radius;
  } ...
}
```

- Der Aufruf `super(...)` ruft den Konstruktor der Superklasse auf.

- Er muss **zu Beginn** des Konstruktors der Subklasse verwendet werden.
- Falls kein expliziter Aufruf von `super(...)` verwendet wird, wird zu Beginn der Defaultkonstruktor der Superklasse aufgerufen.
- Konstruktoren werden in Java nicht vererbt.

Vererbung von Methoden Methodendefinitionen können genauso vererbt werden wie Attribute. Dies ist dann sinnvoll, wenn die Definition einer Methode für alle Subklassen identisch ist.
Zum Beispiel könnten wir die Methode `moveTo` für Figuren hinzufügen, welche die Position einer Figur ändert.

```
class Circle extends AFigure { ...
  private double radius;
  ...
  public void moveTo(CartPt p) { this.loc = p; }
  ...
}

class Square extends AFigure { ...
  private double size;
  ...
  public void moveTo(CartPt p) { this.loc = p; }
  ...
}
```

Da die Implementierung für Kreise und Quadrate identisch ist, können wir die Methode stattdessen in die Klasse `AFigure` auslagern. Sie wird dann von `Circle` und `Square` geerbt und muss daher in diesen Klassen nicht mehr implementiert werden.

```
class AFigure {
  protected CartPt loc;
  AFigure(CartPt loc) {
    this.loc = loc;
  }
  void moveTo(CartPt p) { this.loc = p; }
}
// Aus den Klassen Circle und Square entfernen wir die Definition
// dieser Methode!!
```

2.1 Überschreiben

In vielen Fällen ist es nötig, die Implementierung einer Methode der Superklasse in der Subklasse anzupassen, insbesondere um den erweiterten Zustand zu berücksichtigen.
Überschreiben (engl. overriding) einer ererbten Methode `m` der Superklasse bedeutet, dass man in der Subklasse eine neue, eigene Deklaration für `m` angibt. Die überschreibende Methode muss in Java eine kompatible[1] Signatur zur überschriebene haben und mindestens so zugreifbar sein. Nur zugreifbare Methoden können überschrieben werden.
Die überschriebene Methode kann durch `super` - Aufrufe benutzt werden:

```
super.<methodenName>( <AktParam1>,...)
```

[1] gleicher Name, gleiche Parameter-Typen und Subtyp als Ergebnistyp

Der aktuelle implizite Parameter (`this`) eines `super`-Aufrufs ist der aktuelle implizite Parameter der aufrufenden Methode.

```
class AFigure { ...
  private CartPt loc;
  public double distToOrigin() {
    return loc.distToOrigin();
  }
}

// Klasse Square enthaelt keine eigene Definition dieser Methode mehr!!
class Circle { ...
  public double distToOrigin() {
    return Math.max(super.distToOrigin() - radius, 0);
  }
}
```

Der Aufruf von `loc.distToOrigin()` erfordert, dass `loc` innerhalb der Klasse `Circle` direkt verwendet werden kann. Durch Rückgriff auf die Implementierung der Superklasse kann `loc` zu einem privaten Attribut von `AFigure` werden.

2.2 Beispiel: Wetterdaten

In einem Programm zur Erfassung von Wetterdaten werden Messungen von Temperatur und Luftdruck verwaltet. Für jeden Tag werden Minimal- und Maximalwerte der jeweiligen Messung gespeichert.

Temperature
high, low : int date : Date
getHigh() : int getLow() : int asString() : String

Pressure
high, low : int date : Date
getHigh() : int getLow() : int asString() : String

Frage 3: Implementieren Sie für die Klassen `Temperature` und `Pressure` eine Superklasse `Recording`, welche die Gemeinsamkeiten abstrahiert!
Passen Sie die Klassen dann so an, dass Sie Code-Duplikation vermeiden!

```
// Temperaturmessungen [in Celsius]
class Temperature {
  private Date date;
  private int high;
  private int low;

  Temperature (Date date, int high, int low) { ... }

  int getHigh() {
    return high;
  }
```

```
  int getLow() {
    return low;
  }
  String asString() {
    return date + " : " + low + "-" + high + "C");
  }
}

// Druckmessungen [in hPa]
class Pressure {
  private Date date;
  private int high;
  private int low;

  Pressure (int high, int low, Date date) { ... }

  int getHigh() {
    return high;
  }
  int getLow() {
    return low;
  }
  String asString() {
    return date + " : " + low + "-" + high + "hPA");
  }
}
```

2.3 Vererbung und Information Hiding

Durch die Vererbung gibt es nun zwei Arten, eine Klasse K zu nutzen:

- *Anwendungsnutzung*: Erzeugen und Verwenden der Objekte von K
- *Vererbungsnutzung*: Spezialisieren und Erben von K

Damit die erbende Klasse die geerbten Programmteile geeignet nutzen kann, benötigt sie meist einen intimeren Zugriff als ein Anwendungsnutzer.

Geschützter Zugriff Viele Programmiersprachen bieten einen gesonderten Zugriffsbereich für Vererbung, der alle Subklassen einer Klasse umfasst. Programmelemente, die als ***geschützt*** deklariert sind, d.h. mit dem Modifikator `protected`[2], sind in allen Subklassen zugreifbar.
Will man also Programmelemente, insbesondere Attribute, für Subklassen bereitstellen, müssen sie mindestens geschützten Zugriff gewähren. Private Attribute sind für Subklassen nicht direkt zugreifbar.

```
class C {
  public    int a = 0;
  protected int b = 1;
  private   int c = 2;
  int getC() {
    return c;
  }
}
```

```
class D extends C {
  int getB() {
    return b;
  }
}
```

[2]siehe Kapitel 10

```
public class Attributvererbung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut a: " + d.a);
    System.out.println("Attribut b: " + d.getB());
    System.out.println("Attribut c: " + d.getC());
  }
}
```

2.4 Verschattung von vererbten Attributen

Attribute können ererbte Attribute gleichen Namens verschatten.

> **Frage 4:** Was ist die Ausgabe des folgenden Programms?
> Verwenden Sie im Java-Visualizer die Option “Show overridden fields”, um alle (auch die verschatteten) Attribute zu visualisieren.

```
class C {
  public    int a = 0;
  public    int b = 2;
  private   int c = 3;
  int getC() {
    return c;
  }
}
```

```
class D extends C {
  public int e = 10;
  public int b = 12;
}
```

```
public class Zustandserweiterung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut e: " + d.e);
    System.out.println("Attribut b: " + d.b);

    System.out.println("Attribut a: " + d.a);
    System.out.println("Attribut b: " + ((C) d).b);
    System.out.println("Attribut c: " + d.getC());
  }
}
```

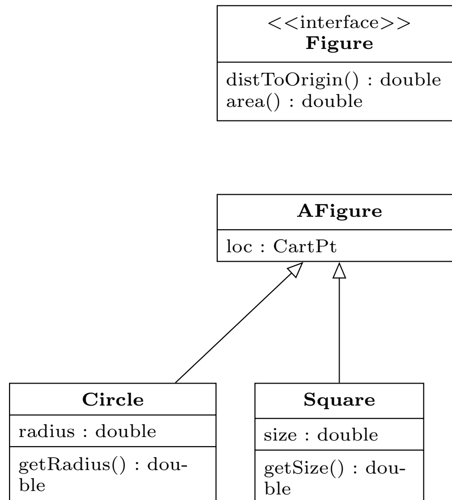
Attribute werden statisch, d.h. während des Compilierens gebunden. Maßgebend ist also der (statische) Typ des selektierten Ausdrucks und nicht der Typ des Objekts, dass sich bei Auswertung des Ausdrucks ergibt.

2.5 Abstrakte Klassen

Beispiel: Geometrische Figuren Für geometrische Figuren haben wir bisher:

```
interface Figure {
  double area();
  double distToOrigin();
}
class AFigure {
```

```
  protected CartPt loc;
  AFigure (CartPt loc)   { ... }
  public double distToOrigin() { ... }
}
class Circle extends AFigure {
  Circle (CartPt loc, double radius) { ... }
  public double area()                     { ... }
  public double distToOrigin()             { ... }
  public double getRadius()                { ... }
} ...
```

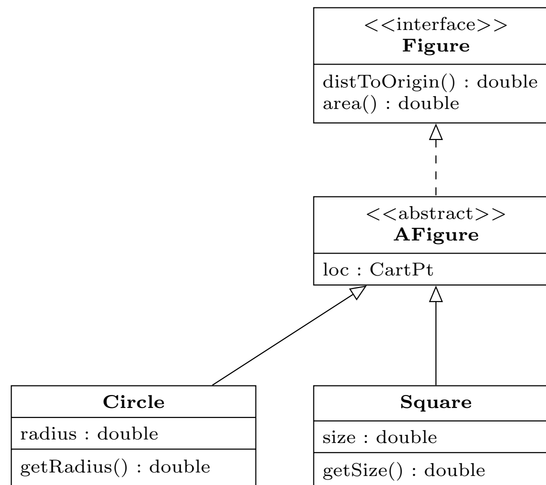


Wenn `AFigure` das Interface `Figure` implementieren soll, dann müssen alle Methoden, die `Figure` verlangt, auch in `AFigure` definiert werden.
Problem: Es ist unklar, wie z.B. `area()` allgemein definiert werden soll, da jede geometrische Figur diese Methode anders definiert.

Eine Methode heißt ***abstrakt***, wenn für sie kein Rumpf angegeben ist. So sind beispielsweise alle Methoden in Interfaces abstrakt. In Klassen können Methoden ebenfalls als abstrakt deklariert werden, indem man ihnen den Modifikator `abstract` voranstellt. Eine Klasse mit abstrakten Methoden (deklariert oder von Supertypen gefordert) muss als abstrakt deklariert werden (ebenfalls Modifikator `abstract`).
Es ist unzulässig, Instanzen abstrakter Klassen zu erzeugen (obwohl abstrakte Klassen Konstrukturen haben können).
Beispiel: Abstrakte `AFigure`-Klasse

```
abstract class AFigure implements Figure {
  protected CartPt loc;
  AFigure (CartPt loc)   { ... }
  public double distToOrigin()  { ... }
  public abstract double area();
}
```

Im Modell:



Frage 5: In der zuvor gezeigten Version der Wetterdaten kann ein Programmierer direkt Objekte von `Recording` ableiten. Wandeln Sie die Klasse `Recording` in eine abstrakte Klasse um, um dies zu verhindern! Fügen Sie ausserdem der Klasse Recording und ihren Subklassen eine Methode `asString()` hinzu, die die Messdaten geeignet als String repräsentiert.

2.6 Dynamische Methodenauswahl bei Vererbung

Der dynamische Typ eines Objekts entspricht dem Klassentyp der Klasse, die bei der Erzeugung des Objektes angegeben wurde.

```
obj.m(...);
```

Beim Methodenaufruf wird die Implementierung der Methode basierend auf dem dynamischen Typ von `obj` ausgewählt.
Ist eine solche Implementierung in der Klasse des dynamischen Typs nicht vorhanden, wird die Implementierung jener Superklasse gewählt, die in der Vererbungshierarchie am weitesten "unten" liegt (d. h. die von der Vererbung her am nächsten ist).

Beispiele: Dynamische Methodenauswahl

1. Auswahl zwischen Methode der Super- und Subklasse:

   ```
   AFigure c = new Circle (new CartPt(4.0,4.0), 1.0);
   ...
   c.distToOrigin();
   ```

   Der statische Typ der Variablen `c` ist `AFigure`. Der dynamische Typ des von `c` referenzierten Objekts ist jedoch `Circle`, weil das Objekt mit `new Circle(...)` erstellt wurde. Deshalb wird hier die Implementierung der Methode `disToOrigin` aus der Klasse `Circle` ausgeführt.

2. Auswahl zwischen Methoden verschiedener Subklassen:

```
double getMaxArea (Figure[] df) {
  double maxArea = 0.0;
   for (int i = 0; i < df.length; i++) {
      Math.max(maxArea, df[i].area());
   }
   return maxArea;
}
```

   In `df` sind Referenzen auf Objekte vom Typ `Figure` gespeichert. Diese können auf Objekte mit dynamischem Typ `Circle` oder `Square` verweisen. Beim Aufruf der Methode `area` wird auf Basis dieses Typs entschieden, welche der beiden Implementierungen der Methode ausgeführt wird.

3. Dynamische Methodenbindung auf `this`:

```
1  class A {
2    int f() {
3      return g() * 2;
4    }
5    int g() {
6      return 3;
7    }
8  }
9  class B extends A {
10   int g() {
11     return 21;
12   }
13 }
```

   Bei einem Aufruf von `new B().f()` wird zunächst die Methode `f` in `A` aufgerufen, da sie nicht in `B` implementiert ist. Die Methode `f` ruft in Zeile 3 die Methode `g` auf (zur Erinnerung: dies ist äquivalent mit `this.g()`). Da der dynamische Typ von `this` in diesem Fall `B` ist, wird die Implementierung von `g` in `B` aufgerufen. Das Ergebnis von `new B().f()` wäre somit `42`.

3 Zusammenfassung: Syntax für die Klassendeklaration

KlassenDeklaration $\rightarrow$

  ModifikatorenListe AbstraktMod `class` $\ll$ *Bezeichner* $\gg$ TypParameter
    KlassenExtendsKlausel ImplementsKlausel `{`
      DeklarationsListe
  `}`

AbstraktMod

   `abstract`
  | ϵ

TypParameter $\rightarrow$

   `<` TypParameterListe `>`
  | ϵ

```
TypParameterListe →
        ≪ Bezeichner ≫  , TypParameterListe
      | ≪ Bezeichner ≫
KlassenExtendsKlausel →
        extends Typ
      | ϵ
ImplementsKlausel →
        implements TypListe
      | ϵ
```

Eine Klasse ***erweitert*** eine direkte Superklasse und ***implementiert*** ggf. mehrere Schnittstellentypen.
Der Entwurf geeigneter Klassenhierarchien ist ein zentraler Aspekt des objektorientierten Modellierung. Dabei sind Abstraktion und Spezialisierung sinnvoll zu kombinieren. Abstraktion/Generalisierung erlaubt es, Klassen zu deklarieren, die die relevante Gemeinsamkeiten einer Gruppe von Klassen ausdrücken. Spezialisierung erlaubt es, Klassen zu deklarieren, die die Funktionalität existierender Klassen erweitern. Vererbung bietet hierzu ein mächtiges Sprachkonzept: Statt Elemente explizit von der Super- in die Subklassen zu kopieren, steht die vererbten Elemente automatisch in der Subklasse bereit. Vererbung ist transitiv.

Hinweise zu den Fragen

Hinweise zu Frage 1:

- `Square`-Objekte sind vom Typ `Square`, `Figure` und `Object`.
- `CartPt`-Objekte sind vom Typ `CartPt` und `Object`.

Hinweise zu Frage 2: Der Ausdruck `x.area()` ist zulässig, nicht aber `x.getRadius()`. Der Typ von `x` ist `Figure` .

Hinweise zu Frage 3:

```
// Messungen
class Recording {
    protected Date date;
    protected int high;
    protected int low;

    Recording (int high, int low, Date date) {
      this.high = high;
      this.low  = low;
      this.date = date;
    }
    int getHigh() {
      return high;
    }
    int getLow() {
      return low;
    }
    String asString() {
        return date + " : " + low + "-" + high);// ohne Einheit
    }
}

class Pressure extends Recording {
  Pressure(int high, int low, Date date) {
    super(high, low, date);
  }
  String asString() {
    return super.asString() + "hPa";
  }
}

class Temperature extends Recording {
  Temperature(int high, int low, Date date) {
    super(high, low, date);
  }
  String asString() {
    return super.asString() + "C";
  }
}
```

Hinweise zu Frage 4:

```
public class Zustandserweiterung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut e: " + d.e); // deklariertes e -> 10
    System.out.println("Attribut b: " + d.b); // deklariertes b -> 12

    System.out.println("Attribut a: " + d.a);       // ererbtes a -> 0
    System.out.println("Attribut b: " + ((C) d).b); // ererbtes b -> 2
    System.out.println("Attribut c: " + d.getC());  // ererbtes c -> 3
  }
}
```

Hinweise zu Frage 5:

```
abstract class Recording {
    protected Date date;
    protected int high;
    protected int low;

    Recording (int high, int low, Date date) {
      this.high = high;
      this.low  = low;
      this.date = date;
    }
    int getHigh() {
      return high;
    }
    int getLow() {
      return low;
    }
    String asString() {
        return date + " : " + low + "-" + high + unit();
    }
    abstract String unit();
}

class Pressure extends Recording {
  Pressure(int high, int low, Date date) {
    super(high, low, date);
  }
  String unit() {
    return "hPa";
  }
}

class Temperature extends Recording {
  Temperature(int high, int low, Date date) {
    super(high, low, date);
  }
  String unit() {
    return "C";
  }
}
```

Die Implementierung hier zeigt das ***Template-and-Hook Muster***: Das Template gibt die grundsätzliche Funktionalität vor und sieht Erweiterungspunkte (Hooks) vor, an denen Subklassen erforderliche Erweiterungen anbringen. Abstrakte Hook-Methoden zwingen die Subklassen die Erweiterungen durchzuführen.