TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Subtyping revisited

Subtypbildung in Java: Klassen

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

   <Liste von Attribut-, Konstruktor-,
    Methodendeklaration>
}
```

- Eine Klasse **implementiert** ggf. mehrere Interfaces.
- Für jede vom Interface geforderte Methode muss in der Klassendeklaration eine Implementierung angegeben werden.
- **Direkte** Supertypen der deklarierten Klasse: Alle implementierten Interface-Typen bzw. `Object` (falls keine `implements`-Klausel)
- Supertypen sind die direkten Supertypen sowie die reflexiv-transitive Hülle der Supertyp-Relation.

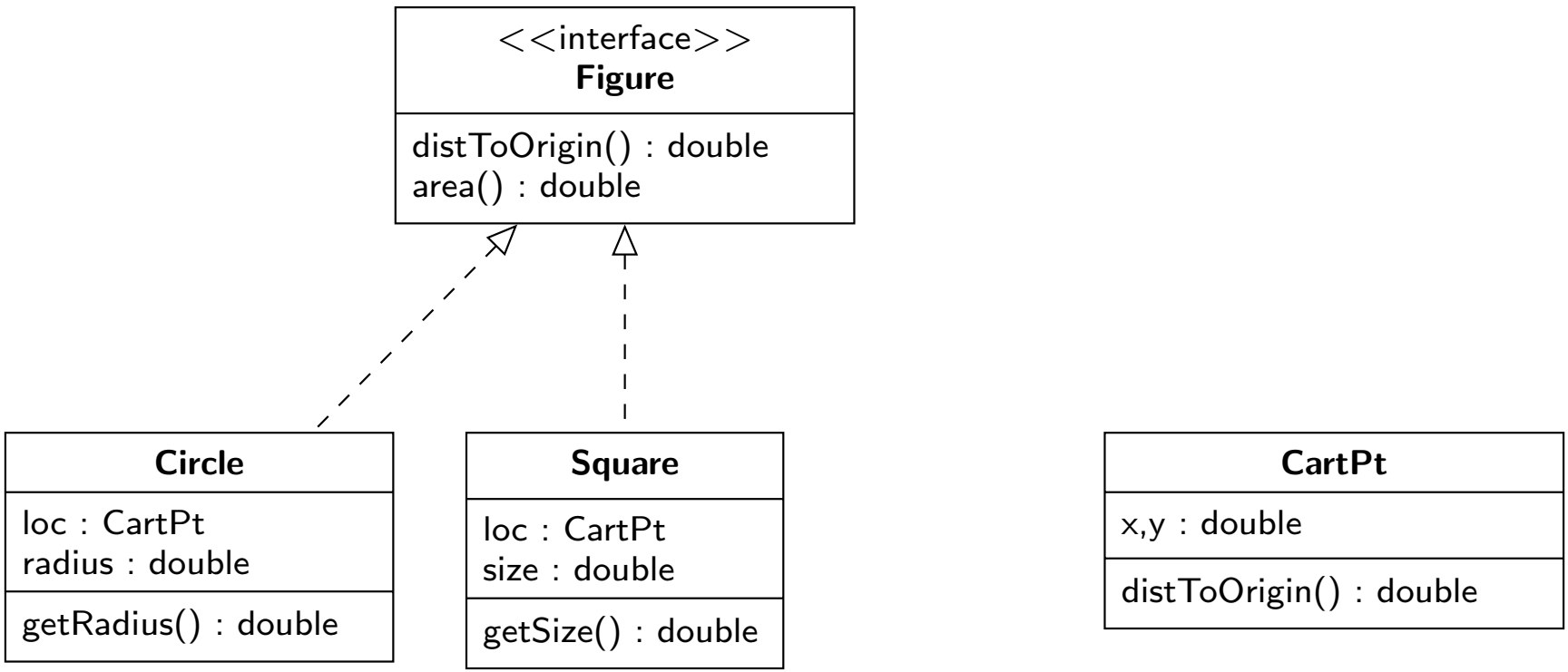
Subtypbildung in Java: Interfaces

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

   <Liste von Methodensignaturen (und Konstanten)>

}
```

- **Direkte** Supertypen: In der `extends`-Klausel genannten Interface-Typen bzw. `Object` (falls keine `extends`-Klausel)
- Auch hier: Supertypen sind die direkten Supertypen sowie die transitive Hülle der Supertyp-Relation.
- Alle geforderten Methoden eines Supertyps werden auch vom Subtyp gefordert. Sie müssen aber nicht erneut aufgelistet werden.

Beispiel: Geometrische Figuren

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



Implementierung I

```
// Interface fuer Geometrische Figuren
interface Figure {
  // ermittelt den Abstand zum Ursprung
  double distToOrigin();
  // ermittelt die Flaeche
  double area();
}
```

Implementierung II

```
class Circle implements Figure {
  private CartPt loc; // Referenzpunkt ist Mittelpunkt
  private double radius;  // Radius

  Circle (CartPt loc, double radius) {
    this.loc = loc;
    this.radius = radius;
  }
  public double area() {
    return radius * radius * Math.PI;
  }
  public double distToOrigin() {
    return Math.max(loc.distToOrigin() - radius,0);
  }
  public double getRadius() {
    return radius;
  }
}
```

Implementierung III

```
class Square implements Figure {
  private CartPt loc;  // Referenzpunkt ist linke untere Ecke
  privat double size;  // Seitenlaenge

  Square (CartPt loc, double size) {
    this.loc = loc;
    this.size = size;
  }
  public double area() {
    return size * size;
  }
  public double distToOrigin() {
    return loc.distToOrigin();
    // Mathematisch korrekt nur fuer Referenzpunkte mit
    // x >= 0 und y >= 0 (siehe Uebung)
  }
  public double getSize() {
    return size;
  }
}
```

Implementierung IV

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

```
// Punkt im zweidimensionalen kartesischen Koordinatensystem
class CartPt {
  private double x, y;

  CartPt(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double distToOrigin() {
    return Math.sqrt(x * x + y * y);
  }
}
```

Frage

Wenn $S \leq T$ gilt, dann sind alle Objekte vom Typ S auch vom Typ T.

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

Frage

Wenn $S \leq T$ gilt, dann sind alle Objekte vom Typ S auch vom Typ T.

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

Prinzip der Substituierbarkeit I

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.

Beispiel:

```
// 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;
}
```

Prinzip der Substituierbarkeit II

```
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);
```

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

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

Vererbung

Abstraktionen auf Klassenebene

Abstraktion bezeichnet die Verallgemeinerung des konkreten Einzelfalls, wobei generelle Strukturmerkmale betont werden.
Beispiele:

- Personen in der Universität (Professoren, Studierende, Verwaltungsangestellte, ...)
- Komponenten von Fenstersystemen (Menues, Schaltflächen, Textfelder, ...)
- Ein-/Ausgabeschnittstellen (Dateien, Netze, ...)

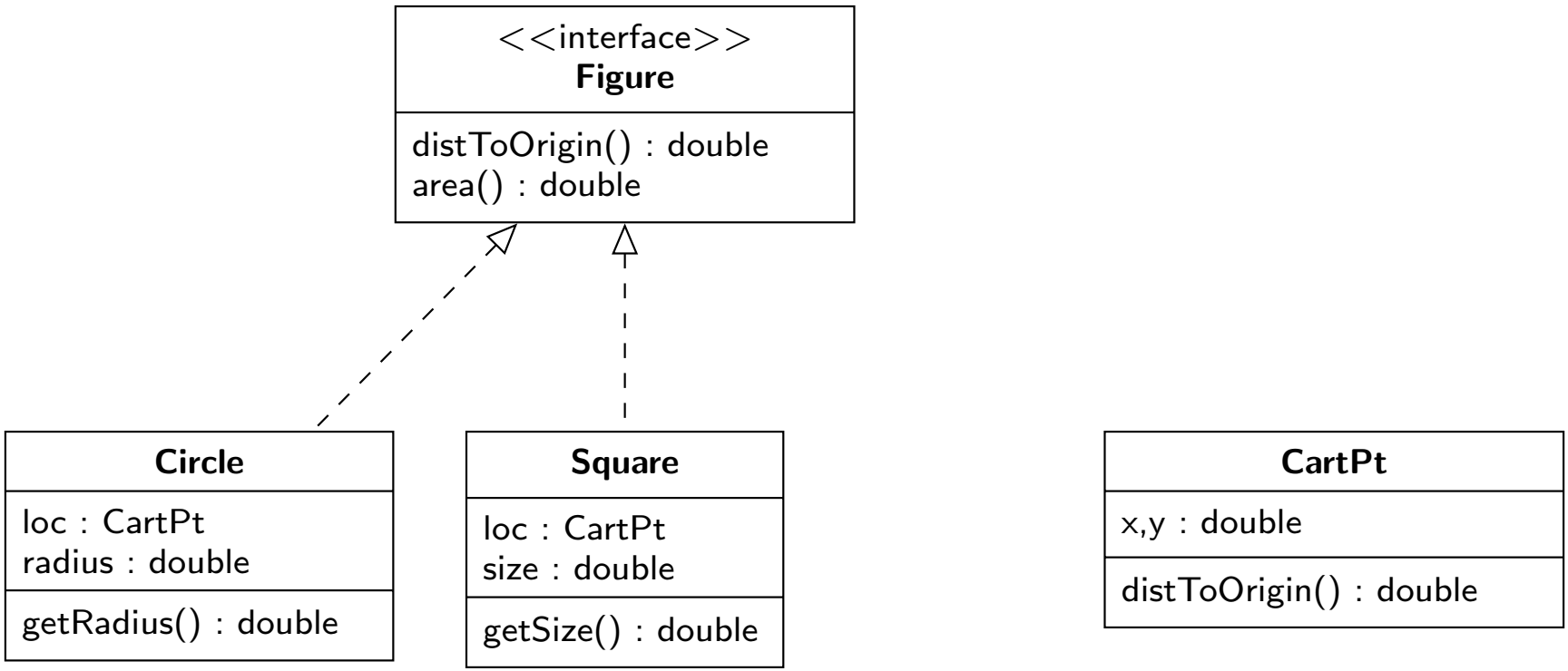
In der objekt-orientierten Software-Entwicklung:

- Extraktion der gemeinsamen Eigenschaften unterschiedlicher Objekte / Klasse
- Vermeiden von Duplikation (weniger Code $\rightarrow$ weniger Arbeit)

Vorgehen

1. Auffinden von Programmfragmenten (Attribute und Methoden) mit ähnlicher Bedeutung / Struktur
2. Erarbeiten einer abstrakteren Klasse, die die gemeinsamen Eigenschaften zusammenfasst und eine entsprechende (verkleinerte) Schnittstelle bereitstellt

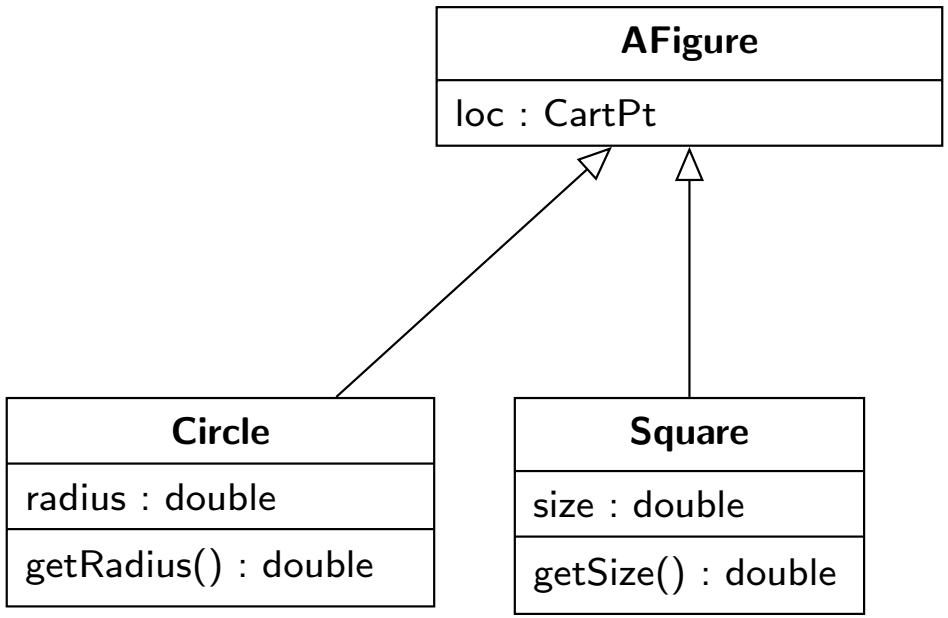
Beispiel: Geometrische Figuren



Idee

- Alle geometrischen Figuren besitzen Attribut `loc`

→ Abstraktion in Klasse `AFigure`



Vererbung (engl. *inheritance*)

- Eine Klasse übernimmt Programmteile von einer anderen Klasse.
- Die erbende Klasse heißt **Subklasse**, die vererbende Klasse heißt **Superklasse**.
- In Java: Attribute und Methoden, **nicht** vererbt werden Konstruktoren sowie statische (Klassen)Attribute und statische (Klassen-)Methoden

Vererbung unterstützt Spezialisierung durch:

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

Beispiel

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

class Square extends AFigure { ... }

class Circle extends AFigure { ... }
```

Auswirkung auf Konstruktoren

- Konstruktor von `AFigure`-Klasse initialisiert `loc`-Attribut.
- `Square`-Objekte haben zusätzlich ein `size`-Attribut.
  $\rightarrow$ muss im Konstruktor der `Square`-Klasse initialisiert werden

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

// analog bei Circle
```

- 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.

Vererbung von Methoden I

Beispiel: Methode `moveTo` zur Positionsänderung

```
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; }
  ...
}
```

Vererbung von Methoden II

Auslagern in Klasse `AFigure`

```
class AFigure {
  protected CartPt loc;
  AFigure(CartPt loc) {
    this.loc = loc;
  }
  public void moveTo(CartPt p) { this.loc = p; }
}
// Die Klassen Circle und Square enthalten keine eigene Definition
// dieser Methode mehr!!
```

Überschreiben (engl. *overriding*) I

- Oft Anpassung der Implementierung einer Methode der Superklasse in der Subklasse notwendig
- Beispiel: `equals()`, `toString()`, `hashCode()` aus Klasse `Object`
- **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 Signatur haben (gleicher Name, gleiche Parameter-Typen und Subtyp als Ergebnistyp) und mindestens so zugreifbar sein.
- Nur zugreifbare Methoden können überschrieben werden.

Überschreiben (engl. *overriding*) II

```
class AFigure { ...
  protected 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(loc.distToOrigin() - radius, 0);
  }
}
```

Überschreiben (engl. *overriding*) III

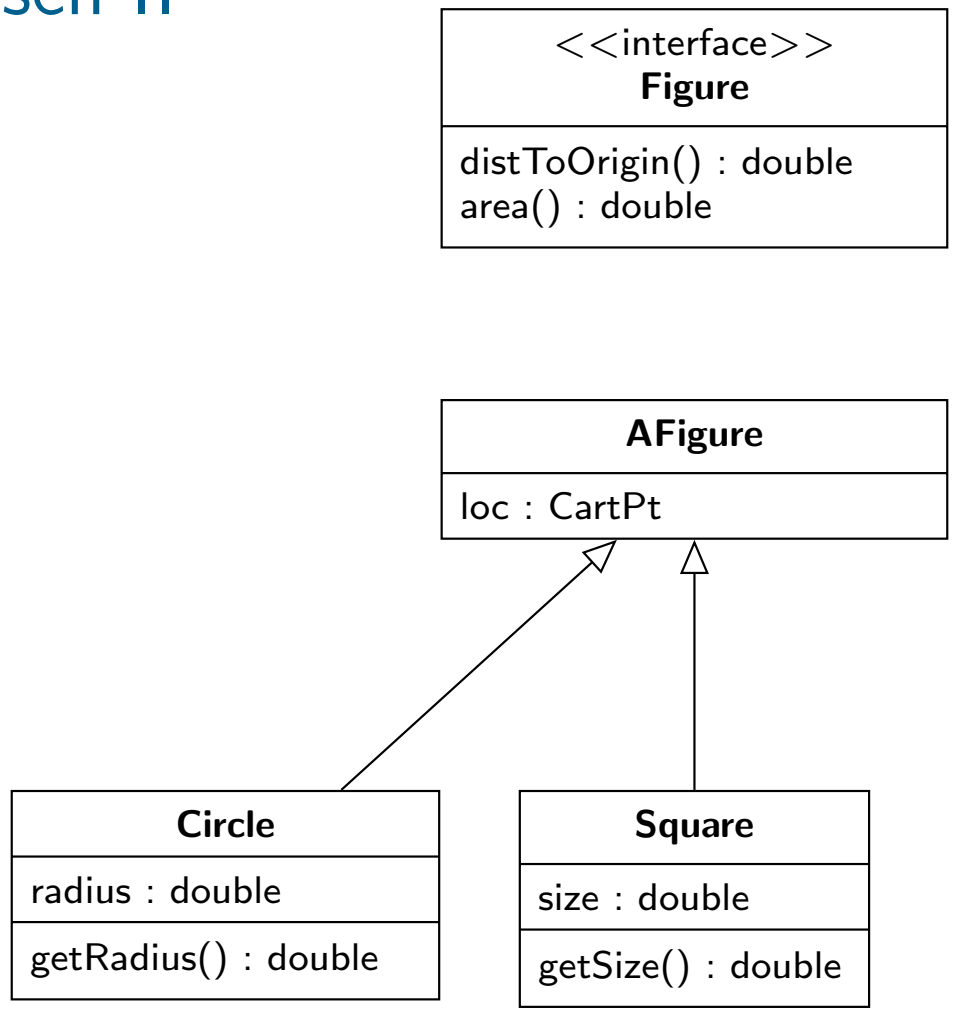
Der Aufruf von `loc.distToOrigin()` erfordert, dass `loc` innerhalb der Klasse `Circle` direkt verwendet werden kann. Durch Rückgriff auf die Implementierung der Superklasse kann `loc` zu einem privaten Attribut von `AFigure` werden.

```
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);
  }
}
```

Abstrakte Klassen I

```
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()               { ... }
} ...
```

Abstrakte Klassen II



Problem: Es ist unklar, wie z.B. `area()` allgemein definiert werden soll!

Abstrakte Klassen III

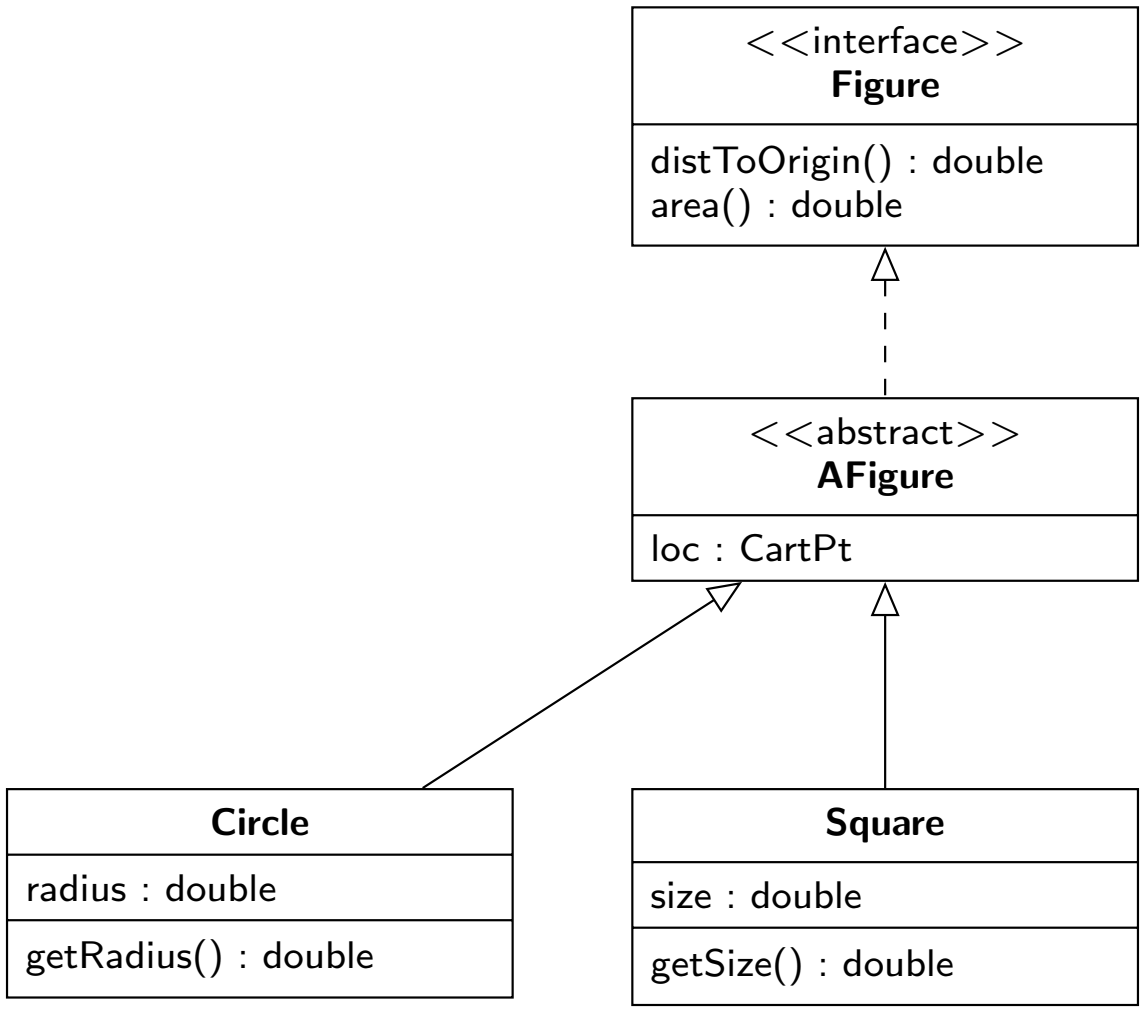
- Eine Methode heißt **abstrakt**, wenn für sie kein Rumpf angegeben ist.
- Alle Methoden in Interfaces sind abstrakt.
- In Klassen können Methoden ebenfalls als abstrakt deklariert werden durch Modifikator `abstract`
- Eine Klasse mit abstrakten Methoden (deklariert oder von Supertypen gefordert) muss als abstrakt deklariert werden (Modifikator `abstract`).
- Es ist unzulässig, Instanzen abstrakter Klassen zu erzeugen (obwohl abstrakte Klassen Konstrukturen haben können).

Beispiel I

```
abstract class AFigure implements Figure {

  protected CartPt loc;

  AFigure (CartPt loc)    { ... }

  public double distToOrigin()  { ... }
  public abstract double area();
}
```

Beispiel II

Im Modell:



Zusammenfassung: Syntax für die Klassendeklaration

KlassenDeklaration →
    ModifikatorenListe AbstraktMod `class` ≪ *Bezeichner* ≫ TypParameter
        KlassenExtendsKlausel ImplementsKlausel {
            DeklarationsListe
    }

AbstraktMod
        `abstract`
    | ϵ
TypParameter →
        < TypParameterListe >
    | ϵ
TypParameterListe →
        ≪ *Bezeichner* ≫ , TypParameterListe
    | ≪ *Bezeichner* ≫
KlassenExtendsKlausel →
        `extends` Typ
    | ϵ
ImplementsKlausel →
        `implements` TypListe
    | ϵ

Beispiel: Wetterdaten I

- Verwaltung von Messungen von Temperatur und Luftdruck
- Für jeden Tag: Minimal- und Maximalwerte der jeweiligen Messung

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

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

Aufgabe

- 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!

Beispiel: Wetterdaten II

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

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

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

Beispiel: Wetterdaten III

```
// 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");
  }
}
```

Lösungsvorschlag I

```
// 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
    }
}
```

Lösungsvorschlag II

```
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";
  }
}
```

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

- Modifikator `protected`: zugreifbar in allen Subklassen
- 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;
  }
}
```

```
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());
  }
}
```

Verschattung von vererbten Attributen

```
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.

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.

Beispiele: Dynamische Methodenauswahl I

1. Auswahl zwischen Methode der Super- und Subklasse:

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

   - Statische Typ der Variablen `c` ist `AFigure`.
   - Dynamischer Typ des von `c` referenzierten Objekts ist `Circle`.

   → Implementierung der Methode `disToOrigin` aus der Klasse `Circle` wird ausgeführt

Beispiele: Dynamische Methodenauswahl II

2. Auswahl zwischen Methoden verschiedener Subklassen:

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

- `df` hat Referenzen auf Objekte vom Typ `Figure`
- Verweist auf Objekte mit dynamischem Typ `Circle` oder `Square`
- Beim Aufruf der Methode `area` wird auf Basis dieses Typs entschieden, welche der beiden Implementierungen der Methode ausgeführt wird.

Beispiele: Dynamische Methodenauswahl III

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 }
14
```

- Ausdruck `new B().f()` ruft Methode `f` in `A` auf
- Die Methode `f` ruft in Zeile 3 die Methode `g` auf (äquivalent zu `this.g()`)
- Da der dynamische Typ von `this` in diesem Fall `B` ist, wird die Implementierung von `g` in `B` aufgerufen.
- Ergebnis von `new B().f()` ist 42

Zusammenfassung: Syntax für die Klassendeklaration

KlassenDeklaration →
    ModifikatorenListe AbstraktMod `class` ≪ *Bezeichner* ≫ TypParameter
        KlassenExtendsKlausel ImplementsKlausel {
            DeklarationsListe
    }

AbstraktMod
        `abstract`
    | ϵ
TypParameter →
        < TypParameterListe >
    | ϵ
TypParameterListe →
        ≪ *Bezeichner* ≫ , TypParameterListe
    | ≪ *Bezeichner* ≫
KlassenExtendsKlausel →
        `extends` Typ
    | ϵ
ImplementsKlausel →
        `implements` TypListe
    | ϵ