Kapitel 13

Abstrakte Methoden und Interfaces

Fachgebiet Eingebettete Systeme und ihre Anwendungen
Prof. Dr. Andreas Koch

TECHNISCHE
UNIVERSITÄT
DARMSTADT

13. Abstrakte Klassen und Interfaces

1. Abstrakte Klassen
2. Interfaces und Mehrfachvererbung

Abstrakte Methoden und Klassen

- Manchmal macht es überhaupt keinen Sinn, eine Methode für eine Basisklasse tatsächlich zu implementieren.

- Für solche Fälle gibt es in Java die Möglichkeit,
  - ◊ eine Methode in einer Klasse einzuführen
  - ◊ ohne sie zu implementieren.

- *Syntaktische Unterschiede*:
  - ◊ Direkt vor dem Rückgabetyp das Schlüsselwort `abstract`.
  - ◊ Der Methodenrumpf `{...}` wird einfach durch ein Semikolon hinter der Parameterliste der Methode ersetzt.

- Eine solche Methode nennt man *abstrakt*.

- Wenn eine Klasse mindestens eine abstrakte Methode hat, nennt man die Klasse ebenfalls *abstrakt*.

- → `abstract` muss auch vor `class` geschrieben werden.

Beispiel

```
public abstract class X {
    public void f () {
        System.out.println ( "Methode f in X" );
    }

    public abstract void g ();
}

public class Y extends X {
    public void g () {
        System.out.println ( "Methode g in Y" );
    }
}

public abstract class Z extends X {
    public void h () {
        System.out.println ( "Methode h in Z" );
    }
}
```

daher muß auch die Klasse als `abstract` deklariert werden

Methode `g` wird in der Basis-Klasse `X` nicht implementiert, daher als `abstract` deklariert

in der abgeleiteten Klasse `Y` wird `g` definiert, daher ist `Y` nicht abstrakt

in der abgeleiteten Klasse `Z` wird `g` auch nicht definiert, daher ist `Z` ebenfalls abstrakt

Abstrakte Klassen

- Von einer abstrakten Klasse `X` kann man
  - durchaus Variable definieren,
  - aber **keine** Objekte anlegen!
- Es wäre auch fatal, wenn das ginge:
  - Dann könnte `X` der dynamische Typ einer Variable vom Klassentyp werden.
  - Dann könnten die Methodenimplementationen von `X` aufgerufen werden.
  - Die existieren aber gar nicht alle!
- *Glücklicherweise*:
  - Wenn es keinen Sinn macht, alle Methoden einer Klasse zu implementieren,
  - dann macht es typischerweise auch keinen Sinn, ein Objekt dieser Klasse zu erzeugen.

Beispiel

```
abstract public class X
 {
  X () { ... }

  abstract public void f ();

  public void g () { ... }
 }
...

X x = new X();
x.f();
```

Fehlermeldung des Compilers:
`X` ist abstrakt!

Beispiel

- Ein Programm soll verschiedene geometrische Figuren verwalten: Kreise, Rechtecke, Quadrate, ...
- Jede dieser Objektarten muss natürlich anders gespeichert werden:
  - ◊ Kreis: Radius
  - ◊ Rechteck: lange und kurze Seite
  - ◊ Quadrat: Seitenlänge
  - ◊ ...
- → Sollte jeweils eine eigene Klasse werden.
- Aber alle diese Objektarten haben gemeinsame Eigenschaften
  - insbesondere kann man für jede den Umfang und die Fläche berechnen
- Weiter kann es auch günstig sein, Arrays von beliebigen geometrischen Objekten zu definieren.
- → d.h. wir brauchen eine Basisklasse für alle diese Klassen.

Beispiel (Fs.)

- Jede dieser Klassen soll Methoden `flaeche` und `umfang` bekommen, die entsprechend definiert sind

- → Auch die Basisklasse, damit man zum Beispiel in einer Schleife für alle Elemente eines Arrays mit ”gemischten“ geometrischen Objekten Fläche und Inhalt für jedes Objekt berechnen kann.

- Natürlich muss jede dieser Methodenimplementationen an die internen Daten der jeweiligen Klasse angepasst werden.

- Daher macht es überhaupt keinen Sinn, die Methoden für die Basisklasse zu implementieren.

Beispiel-Code

```
public abstract class Figur {
    public abstract double umfang ();
    public abstract double flaeche ();
  }

public class Kreis extends Figur {


    protected double r;

    public Kreis ( double radius ) {
       this.r = radius;
     }

    public double flaeche () {
       return Math.PI * r * r;
     }

    public double umfang () {
       return 2 * Math.PI * r;
     }
```

jede geometrische `Figur` muss eine `umfang` und eine `flaeche` Methode haben

ein Kreis ist eine geometrische Figur

definiert durch seinen Radius

Konstruktor für Kreise

konkrete Implementierung von `flaeche` für `Kreis`-Objekte

konkrete Implementierung von `umfang` für `Kreis`-Objekte

Beispiel-Code (Fs.)

```
public class Quadrat extends Figur {
    protected double a;

    public Quadrat ( double seite ) { a = seite; }

    public double flaeche () { return a * a; }

    public double umfang () { return 4 * a; }
 }


public class Rechteck extends Figur {
    protected double l, b;

    public Rechteck ( double laenge, double breite ) {
        l = laenge;
        b = breite;
    }

    public double flaeche () { return l * b; }

    public double umfang () { return 2 * (l + b); }
}
```

konkrete Implementierungen von `flaeche` und `umfang` für `Rechteck` und `Quadrat`

Verwendung des Beispiel-Codes

Die abstrakte Klasse kann verwendet werden, um einen Array von (unterschiedlichen) geometrischen Figuren zu definieren:

```
Figur[] f = new Figur[3];

f[0] = new Kreis    ( 2.0 );
f[1] = new Rechteck( 1.0, 3.0 );
f[2] = new Quadrat  ( 2.0);

double gesamtflaeche = 0;
double gesamtumfang = 0;
for ( int i=0; i < f.length; i++ ) {
   gesamtflaeche += f[i].flaeche();
   gesamtumfang  += f[i].umfang();
}
```

definiert ein Array von Figuren

die einzelnen Elemente des Arrays sind verschiedene Figuren

Berechnung von Gesamtfläche und Gesamtumfang aller Figuren des Arrays

Verboten ist aber:

```
f[0] = new Figur();
```

→ **Figur** ist eben abstrakt!

13. Abstrakte Klassen und Interfaces

1. Abstrakte Klassen
2. Interfaces und Mehrfachvererbung

Interfaces und Mehrfachvererbung

- Bis jetzt haben wir
  - ◊ immer nur eine einzige Klasse hinter `extends` geschrieben,
  - ◊ das heißt, immer nur von einer einzelnen Basisklasse direkt Abgeleitet.

- In vielen objektorientierten Programmiersprachen kann man eine Klasse von mehreren Basisklassen zugleich direkt ableiten.

- Aus verschiedenen (guten!) Gründen ist in Java direkte Vererbung von mehreren Klassen zugleich nicht möglich.

- → Es darf also tatsächlich auch nur der Name einer einzigen Klasse hinter `extends` stehen.

- Oft ist es aber wünschenswert, dass ein und dasselbe Objekt hinter verschiedenen ”Fassaden“ verwendet werden kann.

Lösung in Java: Interfaces

- Die Syntax von Klassen- und Interface–Deklarationen sind im Großen und Ganzen identisch.

- *Wesentliche Unterschiede*:
  - ◊ Das Schlüsselwort `interface` ersetzt das Schlüsselwort `class`.
  - ◊ Methoden werden nicht implementiert, sondern nach dem Methodenkopf steht nur noch ein Semikolon.

    → Wie bei **abstrakten** Methoden.
  - ◊ Bei der Ableitung einer Klasse von einem Interface wird das Schlüsselwort `extends` durch das Schlüsselwort `implements` ersetzt.
  - ◊ Alle Methoden müssen `public` sein
  - ◊ Klassenmethoden sind nicht erlaubt.
  - ◊ Nur konstante Datenkomponenten sind erlaubt.

Beispiel

•Wir wollen nun unsere `Figur`-Klasse erweitern, so daß wir Figuren haben, die **zeichenbar** sind

- zeichenbar heißt, daß es für jedes Objekt Methoden geben muss, die
  - die Position der Zeichnung bestimmen
  - die Farbe der Figur festlegen
  - die Figur tatsächlich zeichnen

Mögliche Lösung:

- Definieren einer abstrakten Klasse `ZeichenbareFigur`, die die entsprechenden Methoden definiert
- und konkrete Unterklassen, wie `ZeichenbarerKreis`, `ZeichenbaresQuadrat`, etc., die diese Methoden definieren

Problem:

- Man müßte dann allerdings wieder die Methoden `umfang` und `flaeche` für jedes zeichenbare Objekt neu implementieren

Syntax von Interfaces

- Man sagt dann auch nicht, die Klasse ist vom Interface abgeleitet.
- Sondern man sagt, die Klasse ***implementiert*** das Interface.
  - Beispiel:

```
public class MeineKlasse implements MeinInterface
```

- Eine Klasse kann mehrere **Interfaces** zugleich implementieren.
  - Beispiel:

```
public class MeineKlasse
       implements MeinInterface1, MeinInterface2
```

- Aber sie darf weiterhin nur von maximal einer **Klasse** abgeleitet sein.
  - Beispiel:

```
public class MeineKlasse
             extends MeineBasisKlasse
             implements MeinInterface1, MeinInterface2
```

Beispiel

- Zeichenbare Objekte haben Methoden
  - zum Festlegen der Farbe
  - zum Festlegen der Position
  - und zum Zeichnen auf einem Graphik-Objekt.

```
public interface Zeichenbar {
   public abstract void setColor (Color c);
   public abstract void setPosition (double x, double y);
   public abstract void zeichne (Graphics g);
 }
```

Anmerkung:
Das Schlüsselwort `abstract` kann man hier auch weglassen, da *alle* Methoden eines Interfaces automatisch abstrakt sind.

Beispiel-Code

```
public class ZeichenbaresQuadrat
        extends Quadrat
        implements Zeichenbar {
```

ein `ZeichenbaresQuadrat` erbt alle Methoden von `Quadrat`

und implementiert die Methoden des Interfaces `Zeichenbar`

```
    protected Color c;
    protected int x, y;
```

Die Datenkomponente `a` wird von `Quadrat` geerbt, die Komponenten `x`, `y`, und `c` werden in der Klasse definiert

```
    public ZeichenbaresQuadrat ( double seite ) { super(seite); }

    public void setColor (Color c) { this.c = c }
    public void setPosition (double x, double y) {
        this.x = (int) x;
        this.y = (int) y;
    }
    public void zeichne (Graphics g) {
        g.setColor(c);
        g.drawRect(x,y,(int) a,(int) a);
    }
```

Die Methoden `flaeche` und `umfang` werden von `Quadrat` geerbt, die Methoden des Interfaces müssen implementiert werden.

Interfaces und `instanceof`

- Interfaces werden im Sinne der Mehrfachvererbung wie **normale Klassen** behandelt
- daher kann auch mit dem Operator `instanceof` überprüft werden, ob ein Objekt ein bestimmtes Interface implementiert
- Beispiel:

```
ZeichenbaresQuadrat q = new ZeichenbaresQuadrat(4.0);

if (q instanceof Figur) {
    System.out.println("Flaeche: " + q.flaeche());
}
else {
    System.out.println("Keine Figur!");
}

if (q instanceof Zeichenbar) {
   q.setPosition(1,1);
}
else {
  System.out.println("Objekt ist nicht zeichenbar.");
}
```

Für `q` liefern die Bedingungen in beiden `if`-Statements `true`

Interfaces als abstrakte Klassen

Analog zu Klassen können auch Interfaces als Container verwendet werden

```
Zeichenbar[] f = new Zeichenbar[3];
```

definiert ein Array von zeichenbaren Objekten

Wir nehmen an, `ZeichenbaresRechteck` und `ZeichenbarerKreis` sind analog zu `ZeichenbaresQuadrat` definiert

```
f[0] = new ZeichenbarerKreis    ( 2.0 );
f[1] = new ZeichenbaresRechteck( 1.0, 3.0 );
f[2] = new ZeichenbaresQuadrat  ( 2.0);

for ( int i=0; i < f.length; i++ ) {
   f[i].setColor(Color.blue);
}
```

Setze die Farbe für alle Objekte auf blau

Verboten ist natürlich auch:

```
f[0] = new Zeichenbar();
```

Abstrakte Klassen vs. Interfaces

•Abstrakte Klassen

- können einige ihrer Methoden implementieren und weitervererben
  - die abstrakten Methoden müssen mit dem Schlüsselwort `abstract` gekennzeichnet werden
- Können auch beliebige Datenkomponenten definieren und weitergeben
- Eine Klasse kann nur von einer abstrakten Klasse **erben**

•Interfaces

- Implementierung und Vererbung von Methoden nicht möglich
  - daher sind alle Methoden automatisch als `abstract` deklariert
  - muß nicht extra hingeschrieben werden (kann aber)
- Können nur Konstanten weitervererben
- eine Klasse kann viele Interfaces **implementieren**

Konstanten in Interfaces

- Interfaces können keine Datenkomponenten enthalten
  - macht auch keinen Sinn, da Interfaces nicht erzeugt werden könnten, sondern nur abstrakte Definitionen sind
- Interfaces können aber Klassen-**Konstanten** enthalten
  - also Komponenten, die `static` und `final` sind
- diese werden dann von Klassen, die die Interfaces implementieren, geerbt.

```
interface A {
   static final char CONST = 'A';
}

class C implements A {
   void f () {
      System.out.println( CONST );
   }
}
```

Die Konstante `CONST` wird vom Interface `A` zur Klasse `C` vererbt.

Mehrfache Interfaces

•Eine Klasse kann die Methoden mehrerer Interfaces implementieren

- z.B. könnte es auch noch ein Klasse `Skalierbar` geben, die angibt, daß ein Objekt mit einem bestimmten Faktor vergrößert werden kann

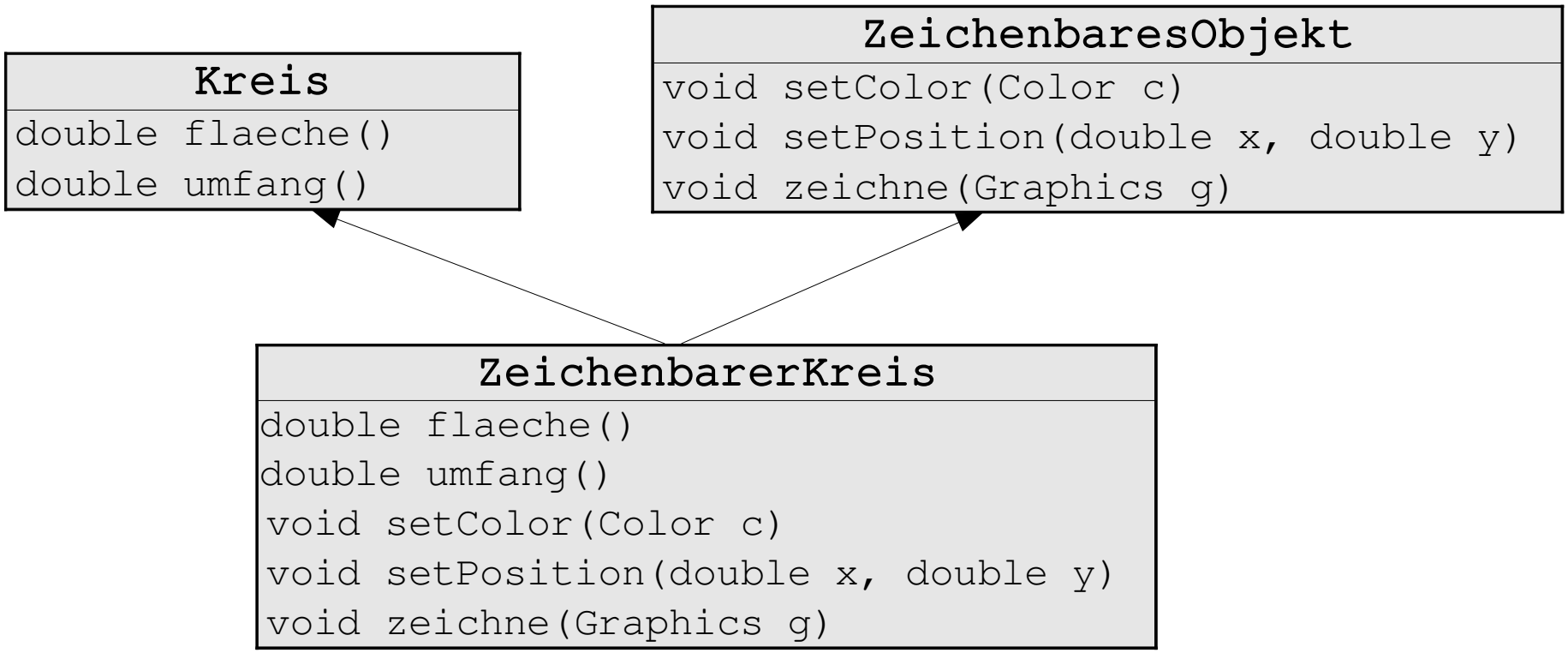
•Die Definition eines skalierbaren, zeichenbaren Quadrats könnte dann so lauten:

```
public class ZeichenUndSkalierbaresQuadrat
             extends Quadrat
             implements Zeichenbar, Skalierbar {

  // die Methoden von Zeichenbar und Skalierbar
  // muessen hier implementiert werden

}
```

Mehrfachvererbung

•in anderen Programmiersprachen (z.B., C++) ist eine Mehrfachvererbung möglich

- das heißt, dass ein Objekt von mehreren Elternklassen erben kann



Interfaces vs. Mehrfachvererbung

- Interfaces können keine internen Datenkomponenten haben
- daher können in Interfaces auch keine Methoden implementiert werden
- obwohl das unter Umständen Sinn machen würde
  - z.B. könnte die Methode `setColor` könnte für alle Unterklassen von `Zeichenbar` gleich sein.

- Bei Mehrfach-Vererbung ist das möglich
- eine neue Klasse kann Methoden und Datenkomponenten von mehreren Basis-Klassen erben
- also könnte `ZeichenbarerKreis`
  - die Methoden `laenge` und `umfang` und die Datenkomponente `r` von `Kreis` erben
  - die Methoden `setColor` und `setPosition` und die Datenkomponenten `c`, `x`, `y`, von `Zeichenbar` erben
  - und nur die Methode `zeichne` tatsächlich neu implementieren

Vererbung von Interfaces

- Interfaces müssen nicht immer neu definiert werden
- sondern können genauso wie Klassen bestehende Interfaces **erweitern**
  - Definition wie gehabt mittels `extends` in der Interface Deklaration

- Besonderheit:
  - Im Unterschied zu Klassen kann ein Interface auch von mehreren Interfaces erben!

- Eine Klasse die solch ein Interface implementiert, muß auch alle Methoden der zugrundeliegenden Basis-Interfaces definieren

Beispiel

```
public interface InterfaceF {
    public void f ();
}

public interface InterfaceG {
    public void g();
}

public interface InterfaceFGH
                  extends InterfaceF, InterfaceG {
    public void h ();
}

public class MeineKlasse implements InterfaceFGH
   {
    public void f () { ... }
    public void g () { ... }
    public void h () { ... }
   }
```

`InterfaceFGH` erbt die Anforderungen von `InterfaceF` und `InterfaceG`

und stellt eine neue

Objekte, die `InterfaceFGH` implementieren, müssen daher Methoden `f`, `g`, und `h` implementieren.