

# Praktische Informatik 1

## Vererbung

Thomas Röfer

- Pakete
- Vererbung
- Frühes / spätes Binden
- Lokale Klassen
- Zugriffsschutz
- Abstrakte Klassen und Schnittstellen
- Anonyme Klassen
- Mehrfaches Erben
- Musterlösung, Übungsblatt

# Schnittstellen (Interfaces)

- Schnittstellen definieren ausschließlich Konstanten und abstrakte Methoden
- Wenn eine Klasse Schnittstellen implementiert, muss sie die darin vereinbarten Methoden definieren
- Zudem erbt sie die vereinbarten Konstanten
- Eine Klasse kann mehrere Schnittstellen implementieren
- Anmerkungen
  - Alle Methoden sind *abstract*
  - Alle Methoden und Konstanten in Schnittstellen sind *public*

```
interface Printable {  
    void print();  
}  
  
interface Drawable {  
    void draw();  
}  
  
class A implements Printable, Drawable {  
    public void print() {...}  
    public void draw() {...}  
}  
  
:  
Printable p = new A();  
p.print();  
Drawable d = (Drawable) p;  
d.draw();
```

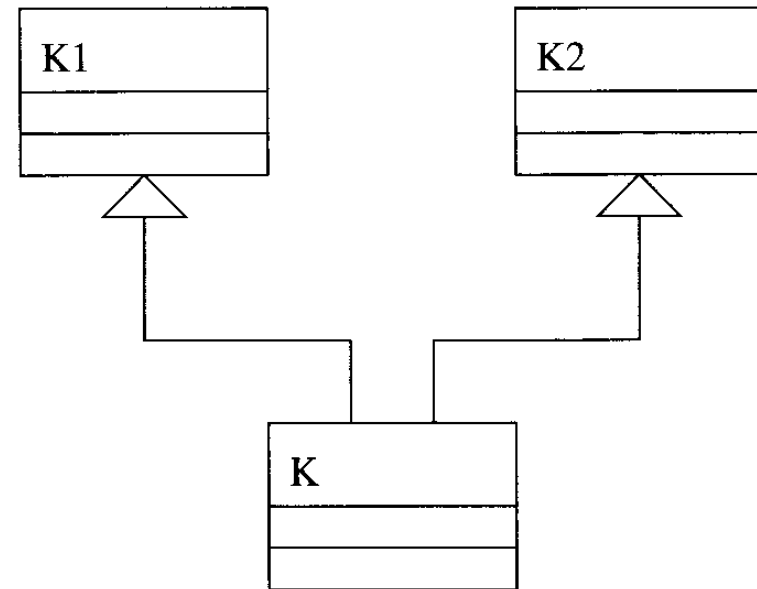
# Anonyme Klassen

- Eine anonyme Klasse hat keinen Namen
- Sie wird *am Ort* erzeugt, d.h. dort, wo eine Instanz von ihr erzeugt wird
- Statt eines eigenen Namens wird der Name der Oberklasse bzw. eines Interfaces angegeben
- Anonyme Klassen sind lokale Klassen
  - In Objekt-Methoden: Objekt-lokal
  - In Klassen-Methoden: Klassen-lokal (*static*)
- Anwendung
  - Werden oft mit Interfaces verwendet, die als Parameter verlangt werden
  - Die anonyme Klasse implementiert das Interface

```
interface Printable {  
    void print();  
}  
  
:  
Printable p = new Printable() {  
    void print() {  
        System.out.print(  
            "Nonsense");  
    }  
};  
p.print();
```

# Mehrfaches Erben

- Eine Klasse erbt von mehreren anderen Klassen
- *Multiple Inheritance* oft falsch übersetzt als *Mehrfachvererbung*
- Richtiger: *mehrfaches (Er-)Erben*
- In Java
  - Erben von mehreren Klassen oder abstrakten Klassen wird nicht unterstützt
  - Erben von mehreren Schnittstellen (Interfaces) ist möglich





# Mehrfaches Erben

## Vorteile

- Ist manchmal dem Problem angemessen
- Oft ist es aber auch bei Programmen in Sprachen, die mehrfaches Erben vollständig unterstützen (z.B. C++), so, dass alle Basisklassen bis auf eine eigentlich nur Schnittstellen sind

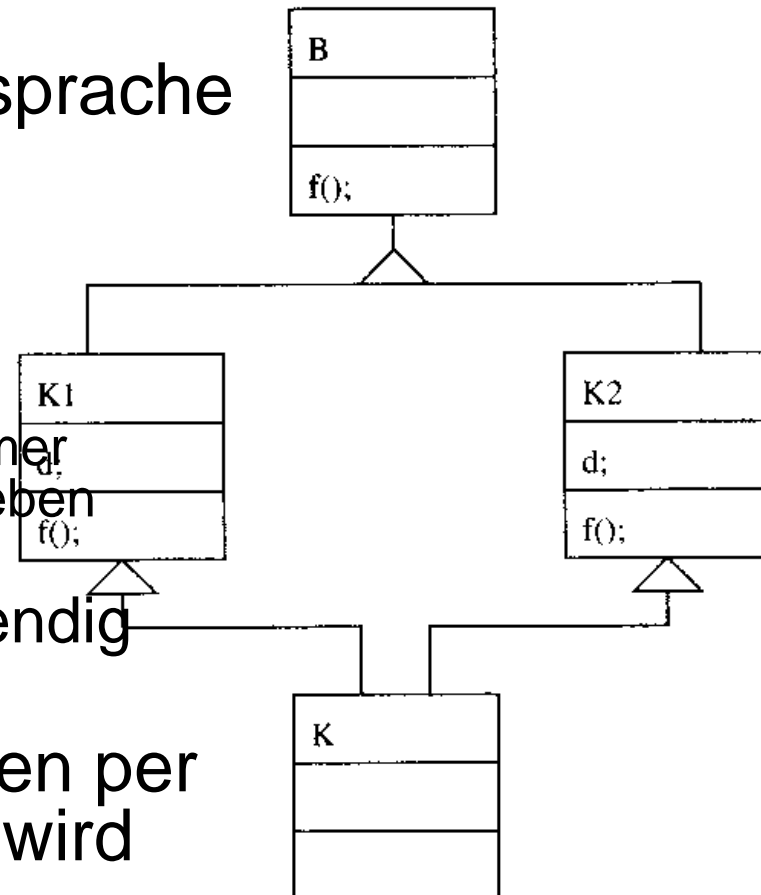
```
class EierLegendeWollMilchSau implements  
    Huhn, Schaf, Kuh, Schwein { ...
```

```
EierLegendeWollMilchSau e =  
    new EierLegendeWollMilchSau();  
Huhn h = e;  
Schaf s = e;  
Kuh k = e;  
Schwein w = e;
```

# Mehrfaches Erben

## Nachteile

- Komplexität der Programmiersprache steigt
  - Ableitungshierarchien sind azyklische, gerichtete Graphen
  - *super* nicht mehr eindeutig
    - in C++ muss daher statt *super* immer der Name der Basisklasse angegeben werden
  - Komplexes *Crosscasting* notwendig
    - *Huhn h = k;*
- Unterscheidung zwischen Erben per Wert und Erben per Referenz wird notwendig

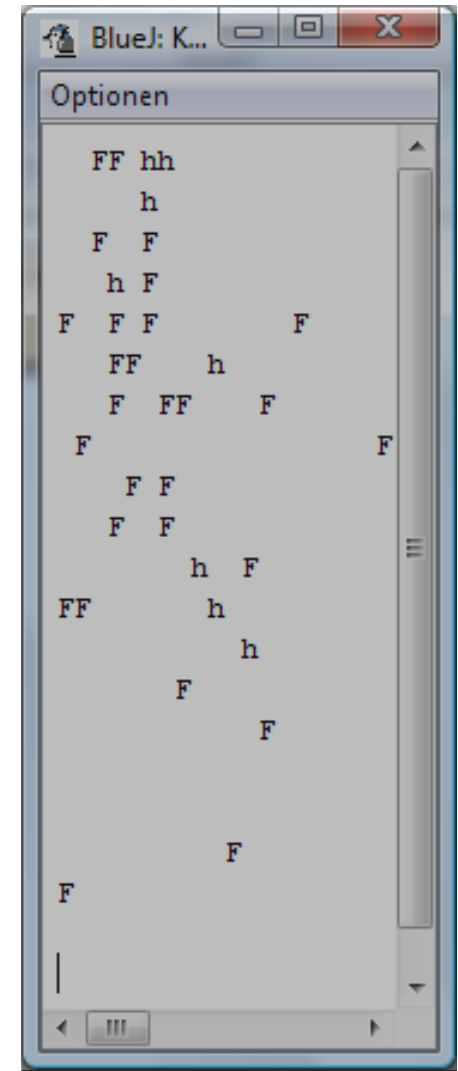


# Die Klasse Object

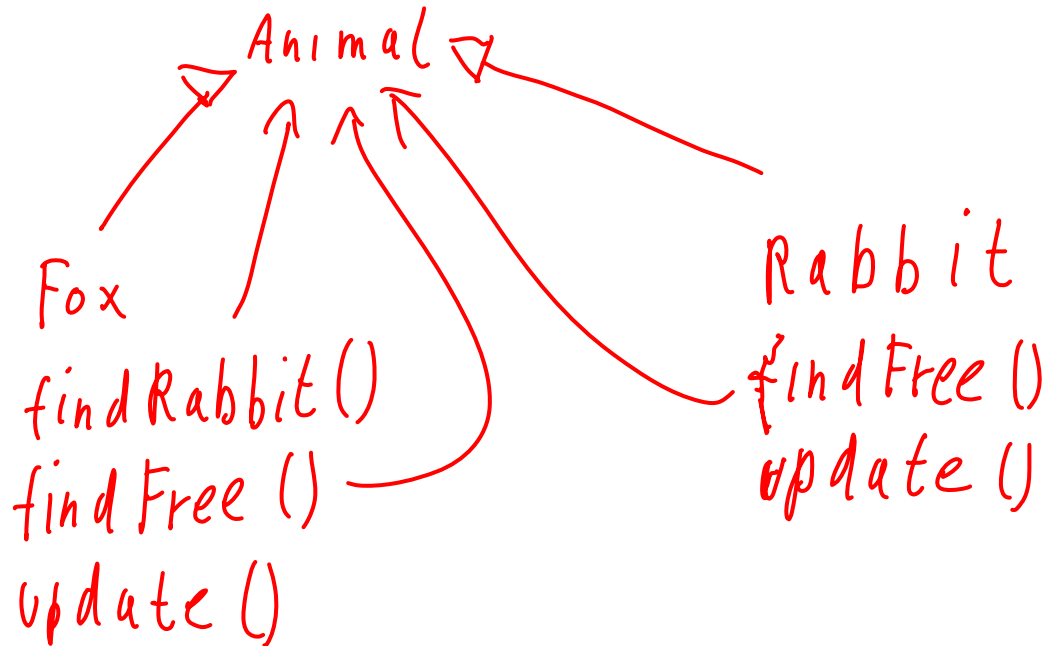
- Wenn bei einer Klasse keine Basisklasse angegeben wird, erbt sie direkt von der Klasse *Object*
  - Java fügt sozusagen *extends Object* ein
- Ansonsten wird indirekt von *Object* geerbt, da die Basisklasse entweder direkt oder wiederum indirekt von *Object* erbt
- *Object* enthält Methoden, die somit in jeder Klasse zur Verfügung stehen
  - *equals()*, *clone()*, *toString()*, ...
- Diese müssen aber in abgeleiteten Klassen passend überschrieben werden

```
class Num implements Cloneable {  
    int number;  
  
    public boolean equals(Object other) {  
        return other instanceof Num &&  
            number ==  
            ((Num) other).number;  
    }  
  
    public Object clone() {  
        Num copy = new Num();  
        copy.number = number;  
        return copy;  
    }  
  
    public String toString() {  
        return "" + number;  
    }  
}
```

- Terrain
  - 20 x 20 Zellen
  - In jeder Zelle kann maximal ein Tier leben
- Ein Hase
  - Läuft herum, wenn eine Nachbarzelle frei ist
  - Kann nach 5 Monaten erste Nachkommen bekommen
  - Bekommt in einem Monat Nachkommen mit einer Wahrscheinlichkeit von 15%
  - Lebt 50 Monate
- Ein Fuchs
  - Wenn er Hunger hat, jagt er Hasen in den Nachbarzellen
  - Ansonsten läuft herum, wenn eine Nachbarzelle frei ist
  - Kann nach 10 Monaten erste Nachkommen bekommen
  - Bekommt in einem Monat Nachkommen mit einer Wahrscheinlichkeit von 9%
  - Lebt 150 Monate oder bis er verhungert ist



# Fuchs und Hase

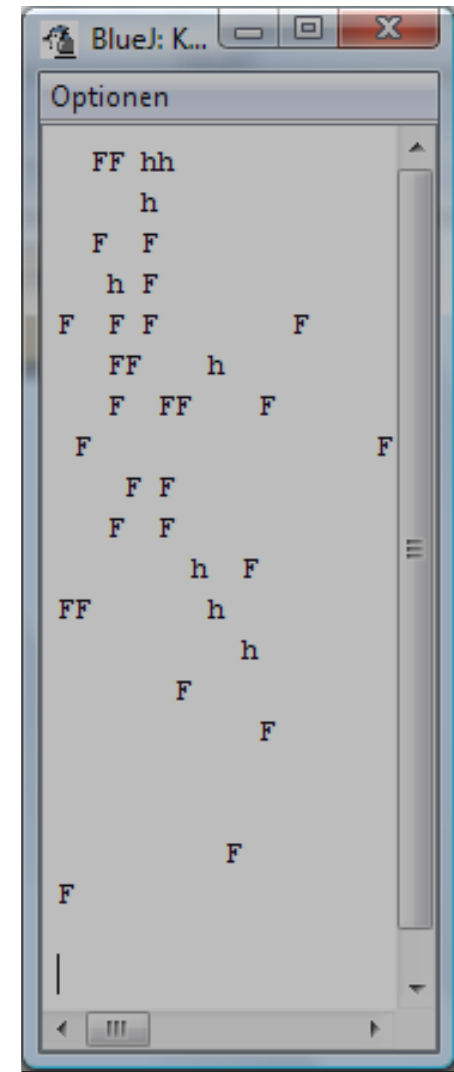


Fox And Rabbit
Tertain

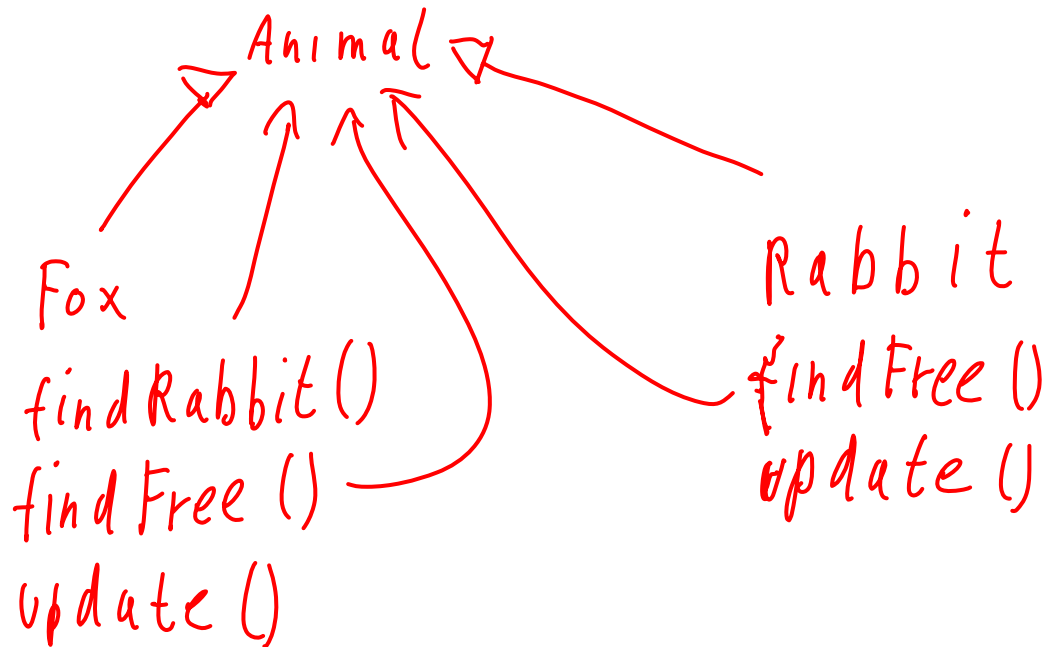
Pos
x
y

# Fuchs und Hase

- Terrain
  - 20 x 20 Zellen
  - In jeder Zelle kann maximal ein Tier leben
- Ein Hase
  - Läuft herum, wenn eine Nachbarzelle frei ist
  - Kann nach 5 Monaten erste Nachkommen bekommen
  - Bekommt in einem Monat Nachkommen mit einer Wahrscheinlichkeit von 15%
  - Lebt 50 Monate
- Ein Fuchs
  - Wenn er Hunger hat, jagt er Hasen in den Nachbarzellen
  - Ansonsten läuft herum, wenn eine Nachbarzelle frei ist
  - Kann nach 10 Monaten erste Nachkommen bekommen
  - Bekommt in einem Monat Nachkommen mit einer Wahrscheinlichkeit von 9%
  - Lebt 150 Monate oder bis er verhungert ist



# Fuchs und Hase



Fox And Rabbit
Tertain

Pos
x
y



# Fuchs und Hase – Lokale Klasse *Pos*, Schnittstelle *Criterion*

```
public class FoxAndRabbit {  
    private Animal[][] terrain =  
        new Animal[20][20];  
  
    private Pos randomPos() {  
        return new Pos(  
            (int) (Math.random() * terrain.length),  
            (int) (Math.random() * terrain[0].length));  
    }  
}
```

```
private interface Criterion {  
    boolean matches(Animal a);  
}
```

```
private static class Pos {  
    public int x, y;  
  
    public Pos(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Pos getNeighbor(int dir) {  
        switch (dir) {  
            case 0:  
                return new Pos(x, y - 1);  
            case 1:  
                return new Pos(x - 1, y);  
            case 2:  
                return new Pos(x, y + 1);  
            case 3:  
                return new Pos(x + 1, y);  
            default:  
                assert false;  
                return null;  
        }  
    }  
}
```



# Fuchs und Hase – Lokale Klasse *Animal*

```
private abstract class Animal {  
    protected Pos pos;  
    protected int age;  
  
    public Animal(Pos pos) {  
        this.pos = pos;  
        terrain[pos.y][pos.x] = this;  
        age = 0;  
    }  
  
    protected void die() {  
        terrain[pos.y][pos.x] = null;  
    }  
  
    protected void move(Pos to) {  
        terrain[pos.y][pos.x] = null;  
        pos = to;  
        terrain[pos.y][pos.x] = this;  
    }  
}
```

```
protected Pos findNeighbor(Criterion c) {  
    for (int i = 0; i < 4; ++i) {  
        int dir = (int) (Math.random() * 4);  
        Pos p = pos.getNeighbor(dir);  
        if (p.y >= 0 && p.y < terrain.length &&  
            p.x >= 0 &&  
            p.x < terrain[p.y].length &&  
            c.matches(terrain[p.y][p.x])) {  
            return p;  
        }  
    }  
    return null;  
}  
  
abstract public char symbol();  
  
abstract public void update();  
}
```

## Fuchs und Hase – Lokale Klasse *Rabbit*

```
private class Rabbit
    extends Animal {

    public Rabbit(Pos pos) {
        super(pos);
    }

    public char symbol() {
        return 'h';
    }
}
```

```
public void update() {
    if (++age == 50) {
        die();
    } else {
        Pos free = findNeighbor(new Criterion() {
            public boolean matches(Animal a) {
                return a == null;
            }
        });
        if (free != null) {
            if (age >= 5 && Math.random() < 0.15) {
                new Rabbit(free);
            } else {
                move(free);
            }
        }
    }
}
```

## Fuchs und Hase – Lokale Klasse *Fox*

```
private class Fox extends Animal {
    int repleteness;

    Fox(Pos pos) {
        super(pos);
        repleteness = 15;
    }

    public char symbol() {
        return 'F';
    }

    public void update() {
        if (++age == 150 ||
            --repleteness == 0) {
            die();
        } else {
            if (repleteness < 5) {
```

```
        Pos rabbit = findNeighbor(new Criterion() {
            public boolean matches(Animal a) {
                return a instanceof Rabbit;
            }
        });
        if (rabbit != null) {
            move (rabbit);
            repleteness = 15;
            return;
        }
        Pos free = findNeighbor(new Criterion() {
            public boolean matches(Animal a) {
                return !(a instanceof Fox);
            }
        });
        if (free != null) {
            if (age >= 10 && Math.random() < 0.09) {
                new Fox(free);
            } else {
                move (free);
            }
        }
    }
}
```



## Fuchs und Hase

```
public FoxAndRabbit() {  
    for (int i = 0; i < 100; ++i) {  
        new Rabbit(randomPos());  
        new Fox(randomPos());  
    }  
}  
  
private void print() {  
    System.out.print('\f');  
    for (Animal[] row : terrain) {  
        for (Animal a : row) {  
            if (a == null) {  
                System.out.print(' ');  
            } else {  
                System.out.print(a.symbol());  
            }  
        }  
    }  
    System.out.println();  
}  
}
```

```
public void run() {  
    while (true) {  
        for (Animal[] row : terrain) {  
            for (Animal a : row) {  
                if (a != null) {  
                    a.update();  
                }  
            }  
        }  
        print();  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {}  
    }  
}
```

# Musterlösung zu Übungsblatt 9

## • Unklarheiten

- Unterschied zwischen Klassen- und Objektmethoden
- Unterschied zwischen Klassen- und Objektattributen

## • Fehler

- Implementierung ohne Klassen
- Kein JavaDoc

Praktische Informatik I WS 2007/08

### Übungsblatt 9

Musterlösung

#### Aufgabe 1 PI-Tunes (100%)

a) Eine Klasse, die Informationen über eine Spur auf einer CD repräsentiert. Sie soll eine Methode bereitstellen, die die laufende Nummer der Spur auf der CD, den Titel und die Länge der Spur in Minuten und Sekunden auf der Konsole ausgibt.

```
1  /**
2   * Die Klasse repräsentiert eine Spur auf einem Album, d.h. ihre laufende
3   * Nummer, ihren Titel und ihre Dauer. Mit {@link print() print} steht eine
4   * Methode zur Verfügung, die alle Informationen über die Spur ausgeben kann.
5   *
6   * @author <a href="mailto:Thomas.Roefer@dfki.de">Thomas Röfer</a>
7   * @version 1.0
8   */
9  class Track {
10     /** Die laufende Nummer dieser Spur auf dem Album. */
11     int number;
12
13     /** Der Titel dieser Spur. */
14     String title;
15
16     /** Der Minutenanteil der Dauer dieser Spur. */
17     int minutes;
18
19     /** Der Sekundenanteil der Dauer dieser Spur. */
20     int seconds;
21
22     /**
23     * Konstruktor.
24     * @param number Die laufende Nummer dieser Spur auf dem Album.
25     * @param title Der Titel dieser Spur.
26     * @param minutes Der Minutenanteil der Dauer dieser Spur.
27     * @param seconds Der Sekundenanteil der Dauer dieser Spur.
28     */
29     Track(int number, String title, int minutes, int seconds) {
30         this.number = number;
31         this.title = title;
32         this.minutes = minutes;
33         this.seconds = seconds;
34     }
35
36     /**
```

# Klassen- und Objektmethoden

## • Klassenmethode

- Es steht *static* davor
- Bezieht sich nicht auf ein Objekt
- Man kann kein *this* darin verwenden
- Wird von außerhalb der Klasse über *Klassenname.methode(...)* aufgerufen
- Innerhalb der Klasse reicht *methode(...)*

## • Objektmethode

- Es steht kein *static* davor
- Bezieht sich immer auf ein Objekt
- Auf das Objekt verweist die Referenz *this*
- Wird immer über die Referenz auf ein Objekt aufgerufen, also *referenz.methode(...)*
- Von anderen Objektmethoden aus reicht *methode(...)*, weil Java ein *this.* voranstellt

# Klassen- und Objektattribute

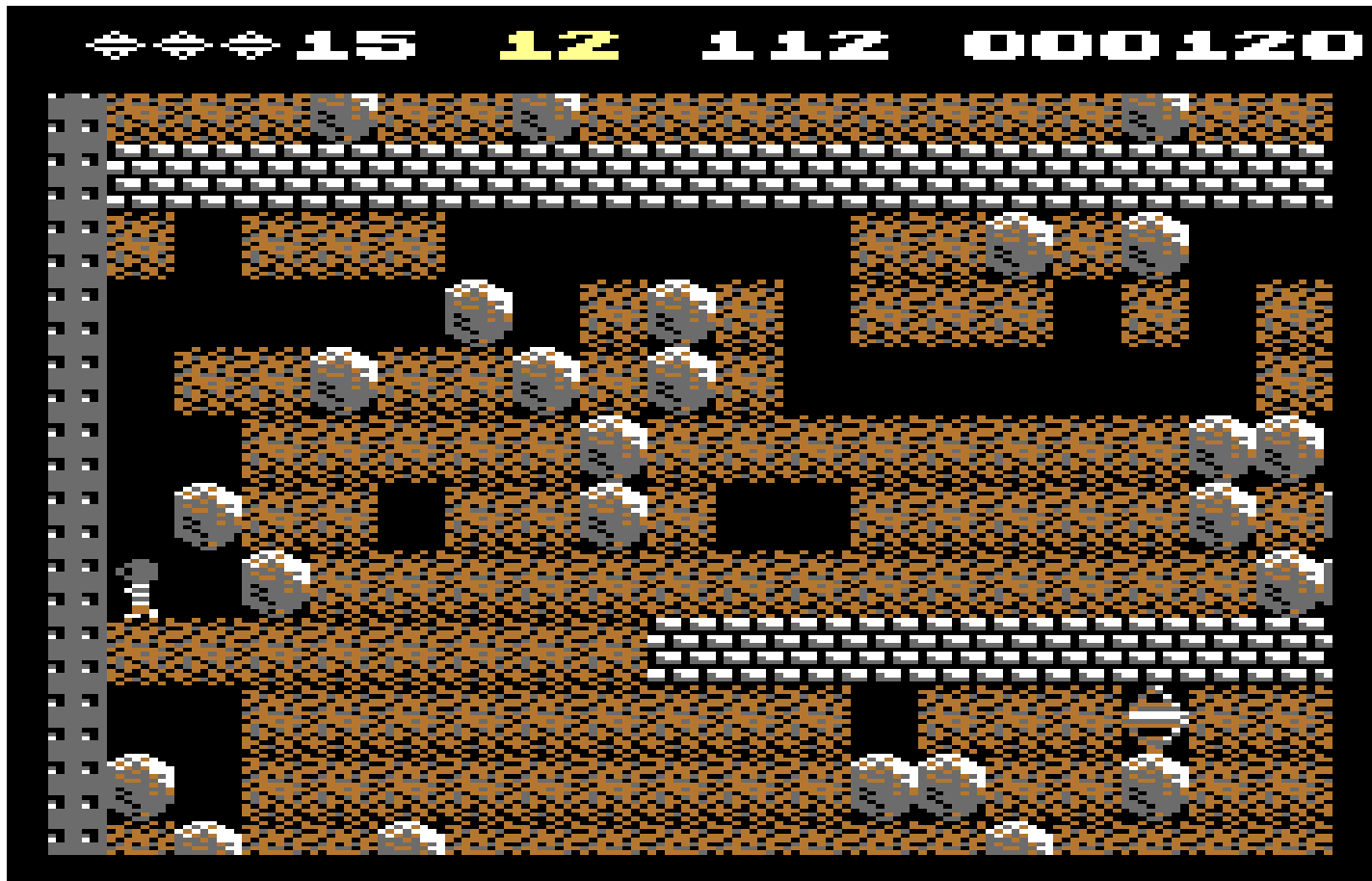
## • Klassenattribut

- Es steht *static* davor
- Gibt es nur einmal
- Darauf wird von außerhalb der Klasse per *Klassenname.attribut* zugegriffen
- Innerhalb der Klasse reicht *attribut*

## • Objektattribut

- Es steht kein *static* davor
- Gibt es einmal pro Objekt, d.h. pro *new Klassenname()*
- Darauf wird immer über die Referenz auf ein Objekt auf zugegriffen, also *referenz.attribut*
- Von Objektmethoden desselben Objekts aus reicht *attribut* weil

# Übungsblatt 11 – Boulder Dash





# Übungsblatt 11

- Implementierung
  - Eine Klasse für das Spiel
  - Eine (abstrakte) Basisklasse oder Schnittstelle für die Spielobjekte
  - Drei Klassen für Spielobjekte
- Tests
  - Vordefinierte Bewegungen des Spielers ablaufen lassen und damit die möglichen Spielenden erzeugen
  - Den dazu nötigen Aufruf und den jeweiligen Endzustand abdrucken
- Tipps
  - Wenn Spielobjekte Objekt-lokale Klassen innerhalb der Spiel-Klasse sind, können sie auf das Spielfeld zugreifen
  - Den Spieler, dem man die Richtung übergeben muss, kann man mit *instanceof* herausfinden

Praktische Informatik I WS 2007/08

## Übungsblatt 11

Abgabe: 30.01.08

### Aufgabe 1 Boulder Dash (100%)

*Boulder Dash* ist ein Spiel, bei dem ein Spieler in einem Steinbruch den Ausgang erreichen muss, ohne dass ihm ein Stein auf den Kopf fällt oder er von einem Ungeheuer gefressen wird. Es gibt folgende Spielelemente:

**Spieler.** Der Spieler („S“) lässt sich in jedem Spielzug in eine von vier Richtungen bewegen (oben, unten, links, rechts), wenn der Weg frei ist. Für den Spieler ist ein Feld frei, wenn es entweder leer („“) ist, nur Erde enthält („E“) oder der Ausgang ist („A“). Wenn er ein Feld verlässt, ist es danach immer leer („“), d.h. er kann Erde wegräumen.

**Stein.** Steine („O“) unterliegen den Gesetzen der Schwerkraft, d.h. sie fallen herunter, wenn sich unter ihnen nichts befindet („“). Befindet sich unter ihnen etwas, aber links neben und links unter ihnen nichts, rutschen sie nach links. Entsprechendes gilt für die rechte Seite. Befindet sich der Spieler unter dem Stein, wird dieser erschlagen, wenn der Stein bereits am Fallen ist, d.h. sich im Spielzug zuvor bereits bewegt hat. Ansonsten bleibt der Stein einfach auf dem Spieler liegen.

**Ungeheuer.** Ungeheuer („U“) bewegen sich immer so lange in eine Richtung, bis sie auf ein Hindernis stoßen und drehen sich dann gegen den Uhrzeigersinn, laufen also insgesamt immer im Kreis. Enthält die Zelle, in die sie sich bewegen wollen, den Spieler, wird dieser gefressen.

**Erde.** Erde („E“) ist für Steine und Ungeheuer ein Hindernis, kann aber vom Spieler wegeräumt werden.

**Mauern.** Mauern („M“) sind permanente Hindernisse. Es darf davon ausgegangen werden, dass die gesamte Spielszene immer von Mauern eingeschlossen ist, d.h. die beweglichen Elemente die Szene nicht verlassen können.

**Ausgang.** Erreicht der Spieler den Ausgang („A“), hat er gewonnen. Für Steine und Ungeheuer ist der Ausgang aber ein Hindernis (wie eine Mauer).

Alle beweglichen Elemente bewegen sich maximal eine Zelle pro Spielzug.

Eine Beispielszene könnte so aussehen:

```
1 String[] scene = {
2     "#####",
3     "A U O M",
4     "M O M",
5     "#####",
6     "M E M",
7     "M S M",
8     "#####";
```