Fallstudie: Game of Life

- Erfunden 1970 von John Horton Conway.
- Spielplan: unendliches 2D-Gitter (für uns: $\infty = 100 \times 100$ o.ä.)
- Eine Zelle ist entweder lebendig oder tot.
- Zu Beginn sind manche Zellen lebendig andere nicht.
- In jeder Runde ändern sich die Zustände wie folgt:
  - Hat eine lebendige Zelle nur einen oder gar keine lebendigen Nachbarn, so "stirbt" sie in der nächsten Runde.
  - Hat eine lebendige Zelle vier oder mehr lebendige Nachbarn so "stribt" sie ebenso in der nächsten Runde.
  - Hat eine tote Zelle genau drei lebendige Nachbarn, so wird sie in der nächsten Runde lebendig.

Aufgabe: Simulation des Spiels.

Benutzerschnittstelle

- Das Spielfeld wird als $m \times n$ Gitter im Grafikfenster dargestellt.
- Lebendige Zellen werden rot ausgefüllt, tote Zellen weiß.
- Zu Beginn werden die lebendigen Zellen mit der Maus angewählt.
- Es werden 1000 Runden simuliert; der zeitliche Abstand der Runden kann eingestellt werden.

Software Architektur

- Eine Klasse, die die Geometrie definiert. Enthält nur statische Instanzvariablen.
- Eine Klasse für die Zellen:
  - Instanzvariablen: Position, Zustand (tot/lebendig), Folgezustand (stirbt, wird lebendig in der nächsten Runde), Grafikfenster
  - Methoden: Zustand ändern, Zustand abfragen, zeichnen.
- Eine Klasse für das Spielfeld:
  - Instanzvariablen: 2D Array von Zellen, Grafikfenster
  - Methoden: zeichnen, Folgezustand berechnen
  - Initialisierung durch Maus im Konstruktor (alternativ: Initialisierungsmethode).

Umgang mit Zellen am Rand des Spielfelds

- zur Ermittlung des Folgezustands einer Zelle müssen alle acht Nachbarn (N, NO, O, SO, S, SW, W, NW) besucht werden.
- Zellen am Rand des Spielfelds haben aber nur 3 Nachbarn, noch dazu jeweils unterschiedliche.
- Vergisst man das, so greift man auf nicht existierende Arrayfächer zu. Programmabsturz.
- Explizite Behandlung der Randfälle mit Fallunterscheidungen ist aufwendig.

  Aus der Spieleprogrammierungen sind zwei Standardlösungen des Problems bekannt.

Randbehandlung durch fiktive Nachbarn

Die Randzellen bleiben immer tot; sie werden auch nicht angezeigt und ihr Folgezustand wird nicht neu berechnet. Sie fungieren nur als fiktive Nachbarn des Randes des sichtbaren Felds.

- Man verwendet ein Array S von $(m+2) \times (n+2)$ Zellen.
- Nur die Zellen $Z[i][j]$ mit $1 \leq i \leq m$ und $1 \leq j \leq n$ werden am Bildschirm angezeigt und jeweils neu aktualisiert.
- Für solche Werte i, j sind dann die 8 Nachbarzellen $Z[i+u][j+v]$ für $u, v \in \{-1, 0, 1\}(u, v) \neq (0, 0)$ alle definiert.
- Der linke Nachbar einer Zelle am linken Rand ist per Definition der rechts gegenüberliegende. Mit anderen Worten werden linker und rechter Rand verklebt, sowie auch der obere und untere. Man spielt sozusagen auf einem Autoreifen (Torus).

Der %-Operator und negative Zahlen

Ist `x` ≥ 0 und `y` > 0, so ist `x % y` der Rest der ganzzahligen Division von `x` durch `y`:

`11 % 7` $= 4$

Der Wert `y`=0 ist verboten.

Ist entweder `x` oder `y` negativ, so gilt

`x % y` $= -|$`x`$|$ `%` $|$`y`$|$

Das hat historische Gründe.

Vermutliche Motivation: `x * (x / y) + (x % y)` $=$ `x`.

Schnelles Potenzieren

Folgendes Programmstück berechnet die Potenz $\mathtt{a}^n$ (Ergebnis ist in `r`).

```
double r = 1;
double b = a;
int i = n;

while (i > 0) {
    if (i % 2 == 1) {
        r = r * b;
    }
    b = b * b;
    i = i / 2;
}
```


Beispiel 3^5

```
double r = 1; double b = a; int i = n;
while (i > 0) {
    if (i % 2 == 1)
        r = r * b;
    b = b * b; i = i / 2; }
```

b	r	i
3	1	5
9	3	2
81	3	1
81^2	243	0

Ordentliche Begründung

Wie funktioniert das ?

Es basiert auf den Gleichungen $x^{2i+1} = x^{2i}x$ und $x^{2i} = (x^i)^2$.

Wie formalisiert man diese Idee ?

Invariante

Wir bezeichnen mit I die Eigenschaft, dass $\mathtt{r} \cdot \mathtt{b}^{\mathtt{i}} = \mathtt{a}^{\mathtt{n}}$.

Die Eigenschaft I gilt zu Beginn der Schleife.

Gilt sie vor Ausführung des Schleifenrumpfes, so gilt sie auch danach. (Begründung folgt.)

Somit gilt sie bei Verlassen der Schleife.

Da dann `i`=0 ist, folgt $\mathtt{r} = \mathtt{a}^{\mathtt{n}}$.

Begründung

Es bezeichne $\mathtt{r}_{\mathrm{alt}}, \mathtt{r}_{\mathrm{neu}}$ den Wert von `r` vor und nach dem Schleifenrumpf. Ebenso für `b`, `i`.

Falls $\mathtt{i}_{\mathrm{alt}}$ gerade ist, so gilt:

$$\begin{aligned}
\mathtt{b}_{\mathrm{neu}} &= \mathtt{b}_{\mathrm{alt}}^2 \\
\mathtt{i}_{\mathrm{neu}} &= \mathtt{i}_{\mathrm{alt}}/2 \\
\mathtt{r}_{\mathrm{neu}} &= \mathtt{r}_{\mathrm{alt}}
\end{aligned}$$

Somit $\mathtt{r}_{\mathrm{neu}}\mathtt{b}_{\mathrm{neu}}^{\mathtt{i}_{\mathrm{neu}}} = \mathtt{r}_{\mathrm{alt}}\mathtt{b}_{\mathrm{alt}}^{\mathtt{i}_{\mathrm{alt}}}$.

Begründung

Falls $\mathtt{i}_{\mathrm{alt}}$ ungerade ist, so gilt:

$$\begin{aligned}
\mathtt{b}_{\mathrm{neu}} &= \mathtt{b}_{\mathrm{alt}}^2 \\
\mathtt{i}_{\mathrm{neu}} &= (\mathtt{i}_{\mathrm{alt}} - 1)/2 \\
\mathtt{r}_{\mathrm{neu}} &= \mathtt{r}_{\mathrm{alt}} \cdot \mathtt{b}_{\mathrm{alt}}
\end{aligned}$$

Wir rechnen:
$\mathtt{r}_{\mathrm{neu}}\mathtt{b}_{\mathrm{neu}}^{\mathtt{i}_{\mathrm{neu}}} = \mathtt{r}_{\mathrm{alt}} \cdot \mathtt{b}_{\mathrm{alt}} \cdot \mathtt{b}_{\mathrm{alt}}^{2\mathtt{i}_{\mathrm{neu}}} = \mathtt{r}_{\mathrm{alt}} \cdot \mathtt{b}_{\mathrm{alt}} \cdot \mathtt{b}_{\mathrm{alt}}^{\mathtt{i}_{\mathrm{alt}}-1} = \mathtt{r}_{\mathrm{alt}}\mathtt{b}_{\mathrm{alt}}^{\mathtt{i}_{\mathrm{alt}}}$.

Die Eigenschaft I heißt Invariante. nnn Es bietet sich oft an, bei Schleifen nach geeigneten Invarianten zu suchen.

Andere Beispiele

Umdrehen eines Strings:

```
String t = "";
for (int i = 0 ; i < s.length() ; i++) {
    t = s.charAt(i) + t;
```

Invariante: $\mathtt{t} = (\mathtt{s}[0..\mathtt{i}-1])^{\mathrm{rev}}$.

Auffüllen mit Leerzeichen:

```
pstr = s;
while(pstr.length() < COLUMN_WIDTH)
   pstr = " " + pstr;
```

Invariante: `pstr.length() <= COLUMN_WIDTH` und `pstr` besteht aus Leerzeichen gefolgt von `s`.

Andere Arten von Invarianten

- Invarianten eines ganzen Programms:

  Beispiel: Die Datenstruktur `Fifo` enthält nur Kästchen.

  Man muss sicherstellen, dass jeder Befehl die Invariante erhält.

- Invariante einer Klasse: Beispiel: Wert einer Instanzvariablen nichtnegativ.

  Man muss sicherstellen, dass der Konstruktor die Invariante garantiert und jede modifizierende Methode sie erhält

Vererbung

```
public class Sparkonto extends Bankkonto {
/* Neue Instanzvariablen, z.B. Zinssatz */
/* Neue Methoden, z.B. Zinsen berechnen */
/* Neue Konstruktoren, z.B. mit Angabe des Zinssatzes */
```

Die bisherigen Methoden bleiben verfügbar, als hätte man sie in die Definition von `Sparkonto()` hineinkopiert.

Bisherige private Instanzvariablen sind nicht sichtbar.

Für Konstruktoren gilt eine besondere Regel.

Sparkonten

```
public class Sparkonto extends Bankkonto {
    private double zinssatz;

    public void zinsenAnrechnen() {
        double zinsen = this.getKontostand() * zinssatz / 100.0;
        this.einzahlen(zinsen);
    }

    public Sparkonto(double satz) {
         zinssatz = satz;
    }
}
```

Man könnte “`this.`” hier auch weglassen.

Verwendung

```
Sparkonto johannasSpar = new Sparkonto(1.25);
Bankkonto matthiasGiro = new Bankkonto(10);

johannasSpar.einzahlen(200.0);
johannasSpar.zinsenAnrechnen();
System.out.println("Johannas Sparkonto: " +
                   johannasSpar.getKontostand());
```

Ausgabe:

```
Johannas Sparkonto: 207.5625
```

Erinnerung: Der Konstruktor `Bankkonto()` setzt den Kontostand auf 5,– (Geschenk der Sparkasse).

Funktionsweise

Beim Aufruf

```
johannasSpar.zinsenAnrechnen();
```

wird folgender Code ausgeführt:

```
double zinsen = johannasSpar.getKontostand() *
                                 zinssatz / 100.0;
johannasSpar.einzahlen(zinsen);
```


Terminologie

`Bankkonto` ist <u>die</u> Oberklasse (*superclass*) von `Sparkonto`.

`Sparkonto` erbt von (*inherits from*) `Bankkonto`.

`Sparkonto` ist <u>eine</u> Unterklasse (*subclass*) von `Bankkonto`.

In Java hat jede Klasse nur eine Oberklasse.

Eine Klasse kann aber mehrere Unterklassen haben.

Z.B. bei Bankkonto: Girokonto, Tagesgeldkonto, etc.

Auch von Unterklassen kann man weiter ererben: Prämiensparkonto als Unterklasse von Sparkonto, etc.

Subtyping

Ein Objekt der Unterklasse kann immer dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird.

Beispiel:

```
Bankkonto matthiasGiro = new Bankkonto();
matthiasGiro.ueberweisen(200, johannasSpar);
```

Erinnerung:

```
public void ueberweisen(double betrag, Bankkonto empfaenger)
```

Zwischen der Unterklasse und der Oberklasse besteht eine "ist-ein"-Beziehung (*"is a" relationship*).

In den Anwendungen ergibt sich oft eine komplexe Klassenhierarchie.

Klassenhierarchien

- – Thecodonten, Pterosaurier, Dinosaurier, Crocodilier sind "Unterklassen" der Archosaurier.
  – Saurischier und Ornithischier sind Unterklassen der Dinosaurier.
- – JPanel, JTextcomponent, JLabel, AbstractButton sind Unterklassen von JComponent
  – JTextField, JTextArea sind Unterklassen von JTextComponent
  – JToggleButton und JButton sind Unterklassen von AbstractButton
  – JCheckBox und JRadioButton sind Unterklassen von JToggleButton

UML-Notation

UML = Unified Modelling Language.

- Klassen als dreigeteilte Rechtecke (Name, *public* Instanzvariablen, Methoden).
- Pfeil mit hohler Dreieckspitze (—▷) von Unterklasse zu Oberklasse.
- Objekte als zweigeteilte Rechtecke: Unterstrichener Klassenname, Instanzvariablen mit Werten.
- Pfeile mit ausgefüllter Dreiecksspitze (—▶) für Verweise.

Vererbung von Instanzvariablen

Alle Instanzvariablen der Unterklasse werden vererbt,

Private Instanzvariablen sind aber in der Unterklasse nicht sichtbar.

Auf sie kann allerdings mit ererbten Methoden der Oberklasse zugegriffen werden.

Ein Sparkonto hat also die Instanzvariablen `kontostand` und `zinssatz`. Die neugeschriebenen Methoden können aber nur `zinssatz` beeinflussen.

Konstruktoren der Unterklasse

Konstruktoren der Oberklasse werden mit `super(...)` bezeichnet.

Man sollte einen Konstruktor der Oberklasse zu Beginn des Konstruktors der Unterklasse aufrufen um die ererbten privaten Instanzvariablen geeignet zu initialisieren.

Tut man das nicht (wie wir im Beispiel), so wird der Defaultkonstruktor (ohne Parameter) der Oberklasse automatisch eingefügt.

Wir könnten z.B. schreiben:

```
public Sparkonto(double betrag, double satz) {
   super(betrag);
   zinssatz = satz;
}
```

Überschreiben

Definiert man in der Unterklasse eine Methode, die es in der Oberklasse schon gibt (Methodenname und Parameterzahl und -typen stimmen überein), so wird die ererbte Methode überschrieben (engl. *overriding* = "sich hinwegsetzen über").

Die neudefinierte Methode ersetzt dann die alte überall, d.h.

auch in ererbten Methoden, die die überschriebene Methode verwenden

Beispiel Girokonto

```
public class Girokonto extends Bankkonto {
    private int anzahlTransaktionen;
    private double gebuehrensatz;

    public Girokonto(double satz) {
        super(0.0);
        anzahlTransaktionen = 0;
        gebuehrensatz = satz;
    }

    public void einzahlen(double betrag) {
        super.einzahlen(betrag);
        anzahlTransaktionen++;
    }
```


Beispiel Girokonto

```
    public void abheben(double betrag) {
        super.abheben(betrag);
        anzahlTransaktionen++;
    }

    public void gebuehrenAbziehen() {
        double gebuehren = anzahlTransaktionen * gebuehrensatz;
        super.abheben(gebuehren);
        anzahlTransaktionen = 0;
    }
}
```


Bemerkungen

- Die Pseudovariable `super` bezeichnet das aktuelle Objekt aufgefasst als Objekt der Oberklasse.
- Hätten wir geschrieben

  `this.abheben(gebuehren);`

  so wären für das Abziehen der Gebühren gleich wieder welche fällig geworden. (Entspricht wahrscheinlich sogar der Realität!).
- Wir können in `Girokonto` nicht schreiben

  `kontostand = kontostand - gebuehren;`

  da `kontostand` nicht sichtbar ist.
- Wir könnten natürlich eine neue Instanzvariable mit Namen `kontostand` deklarieren, dann hätte jedes Objekt aber zwei verschiedende Instanzvariablen des Namens `kontostand`.

Geschachtelte Vererbung

Ist A Unterklasse von B, so kann man ohne weiteres eine Unterklasse C von B definieren.

Jedes Objekt der Klasse C ist dann automatisch ein Objekt der Klasse B und als solches auch ein Objekt der Klasse A.

Die Oberklasse von C ist aber nur B.

Festgeldkonto

```
public class Festgeldkonto extends Sparkonto {
    int restJahre;
    double strafgebuehr;

    public void zinsenBerechnen() {
        restJahre = restJahre--;
        super.zinsenAnrechnen();
    }

    public void abheben(double betrag) {
        if (restJahre > 0)
            super.abheben(strafgebuehr);
        super.abheben(betrag);
    }
```


Festgeldkonto

```
    Festgeldkonto(int jahre, int gebuehr, double zinssatz) {
        super(zinssatz);
        strafgebuehr = gebuehr;
        restJahre = jahre;
    }
}
```


Polymorphie

Jedes Objekt gehört zu einer festen Klasse.

Welche Methode bei einem Objekt aufgerufen wird, richtet sich immer nach seiner Klasse, auch dann, wenn es per Subtyping als Objekt der Oberklasse benutzt wird.

Das bezeichnet man als Polymorphie. Aber Vorsicht: dieser Begriff wird auch in anderen Zusammenhängen benutzt.

Beispiel: in

```
Girokonto matthiasGiro = new Girokonto(0.25);
Sparkonto johannasSpar = new Sparkonto(200.0);
johannasSpar.ueberweisen(100.0, matthiasGiro);
```

fallen bei `matthiasGiro` Gebühren an, obwohl die Methode `abheben` in `Bankkonto` keine Gebühren beaufschlagt.

Typecast und `instanceof`

Hat ein Ausdruck den Typ einer Oberklasse, so kann man ihn explizit auf die Unterklasse konvertieren:

```
Girokonto matthiasGiro = new Girokonto(0.25);
Bankkonto x = matthiasGiro;
Girokonto y = (Girokonto)x;
```

Wird so eine Typkonversion (*type cast* der Form `(A)e` ausgewertet, so wird überprüft, ob die tatsächliche Klasse von `e` Unterklasse (evtl. über mehrere Ecken) von `A` ist.

Falls ja, so wird in der Auswertung fortgefahren, falls nein, bricht das Programm ab.

Der Ausdruck `e instanceof A` hat den Wert `true` genau dann, wenn die tatsächlich Klasse von `e` unterhalb von `A` liegt.

Schnittstellen

Eine Schnittstelle (*interface*) besteht aus Methodensignaturen, d.h. Methodendefinitionen ohne den `{...}` Block.

Syntax:

```
public interface NameDesInterface {
   MethodenSignatur1;
   MethodenSignatur2;
   ...
}
```

Beispiel aus `java.lang`:

```
public interface Comparable {
   int compareTo(Object other);
}
```

Dadurch wird `Comparable` zum Typ all derer Objekte, die eine Vergleichsoperation implementieren.

Implementierung

Man kann jetzt z.B. `Sparkonto` auch so definieren:

```
public class Sparkonto extends Bankkonto
     implements Comparable {
     ...
     public int compareTo(Object other) {
        Bankkonto x = (Bankkonto)other;
        if (this.getKontostand() < x.getKontostand())
            return -1;
        else if (this.getKontostand() > x.getKontostand())
            return 1;
        else return 0;
    }
    ...
}
```


Verwendung von Schnittstellen

Die Schnittstelle ist ein Typ wie jeder andere auch.

Hat ein Ausdruck als Typ eine Klasse, die eine Schnittstelle implementiert, so hat der Ausdruck als Typ auch diese Schnittstelle.

Zum Beispiel hat jedes `Sparkonto` den Typ `Comparable`.

In der Klasse `Arrays` gibt es die statische Methode `sort`, die ein Array von Objekten sortiert, vorausgesetzt, die Einträge implementieren die Schnittstelle `Comparable`:

```
Arrays.sort(Comparable[] a, int fromIndex, int toIndex)
```

Nochmal `Object`

Die Klasse `Object` steht an der Spitze jeder Klassenhierarchie.

Sie definiert die Methoden

```
String toString() /* in einen String konvertieren */
boolean equals(Object other) /* auf Gleichheit testen */
Object clone() /* Kopie erzeugen */
```

Alle diese Methoden kann man in einer Klasse überschreiben.

Übungen

Was ist `b.getKontostand()` am Ende ?

```
Sparkonto b = new Sparkonto(10);
b.einzahlen(4995);
b.abheben(b.getKontostand() / 2);
b.zinsenAnrechnen();
```

Was sollte hier Unterklasse sein, was Oberklasse:

- Angestellter-Vorgesetzter
- Polygon- Dreieck
- Doktorstudent-Student
- Person-Student
- Angestellter-Doktorstudent

Übungen

- Bankkonto-Girokonto
- Fahrzeug-Pkw
- Fahrzeug-Van
- Pkw-Van
- Lkw-Fahrzeug

Die Klasse `Sub` sei Unterklasse von Sandwich. Welche der folgenden Zuweisungen sind erlaubt?

```
Sandwich x = new Sandwich();
Sub y = new Sub();
x = y;
y = x;
y = new Sandwich();
```


Übungen

```
x = new Sub();
```

Man überschreibe die Methode `toString` bei `Girokonto`.

Man implementiere eine Unterklasse `Quadrat` von `Rectangle`.