Workshop

Das Spiel des Lebens

Neue Version vom Februar 2020

Schritt 1

Eine ganz einfache Java-Anwendung

Hier der komplette und lauffähige Quelltext:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Anwendung extends JFrame implements ActionListener
{
    Button ok;

    public Anwendung()
    {
        setSize(500,500);
        setTitle("Game of Life");
        setResizable(false);
        setVisible(true);
        initComponents();
    }

    public void initComponents()
    {
        ok = new Button("OK");
        setLayout(null);
        ok.setBounds(20,430,460,40);
        ok.setFont(new Font("Arial",1,18));
        ok.addActionListener(this);
        add(ok);
    }

    public void actionPerformed(ActionEvent event)
    {
       if (event.getSource() == ok)
          ok.setLabel("geklickt");
    }

    public static void main(String[] args)
    {
        new Anwendung();
    }
}
```

Die Klasse heißt `Anwendung`. Bei dieser Klasse handelt es sich um eine Tochterklasse der Java-Klasse `JFrame`. Diese Klasse ist sozusagen die "Mutter" aller Java-Anwendungen, und unsere Tochterklasse `Anwendung` erbt wichtige Attribute und Methoden von dieser Mutterklasse.

Stichwort Vererbung

Eine Klasse X kann Attribute und Methoden an eine oder mehrere Tochterklassen Y1, Y2 etc. vererben. Diese Tochterklassen verhalten sich dann genau so wie die Mutterklasse, können aber neue Attribute und Methoden bekommen, so dass die Tochterklassen noch "mächtiger" sind als die Mutterklasse. Das Schlüsselwort für diese Vererbungs-Beziehung ist "extends".

Neben dieser "klassischen" Art der Vererbung gibt es in Java eine weitere Vererbungsmethode, die durch das Schlüsselwort "implements" gekennzeichnet ist. Unsere Klasse `Anwendung` ist nicht nur eine Tochterklasse von `JFrame`, sondern implementiert auch Methoden der Klasse `ActionListener`. Im Grunde handelt es sich hierbei um eine Art Mehrfachvererbung.

Die Anwendung wird im Konstruktor durch mehrere Befehle initialisiert.

```
public Anwendung()
{
    setSize(500,500);
    setTitle("Game of Life");
    setResizable(false);
    setVisible(true);
    initComponents();
}
```

Die Fenstergröße wird auf 500 x 500 Pixel gesetzt, der Titel des Fensters auf "Game of Life", mit `setResizable(false)` wird dafür gesorgt, dass der Benutzer das Fenster nicht mit der Maus vergrößern oder verkleinern kann, mit `setVisible(true)` wird das Fenster überhaupt erst mal sichtbar gemacht, und mit `initComponents( )` wird eine Methode aufgerufen, die für weitere Initialisierungen sorgt.

```
public void initComponents()
{
    ok = new Button("OK");
    setLayout(null);
    ok.setBounds(20,430,460,40);
    ok.setFont(new Font("Arial",1,18));
    ok.addActionListener(this);
    add(ok);
}
```

Zunächst wir ein Button-Objekt namens "ok" erstellt. Der Button erhält die Beschriftung "OK".

Der folgende Befehl `setLayout(null)` ist ganz wichtig, damit man die Komponenten des Fensters pixelgenau positionieren kann. Würde dieser Befehl fehlen, so würden die Komponenten ähnlich wie auf einer Webseite nacheinander angeordnet.

Der `setBounds()`-Befehl teilt dem Button-Objekt mit, wo es sich genau befinden soll und wie groß es sein soll. Die beiden ersten Zahlen geben die Koordinaten der linken oberen Ecke des Buttons an, die beiden letzten Zahlen bestimmen die Breite und die Höhe des Buttons. Der Button ist also 460 Pixel breit und 40 Pixel hoch und befindet sich ganz unten in dem Fenster (y-Wert der Oberkante = 460).

Mit dem Befehl `setFont()` kann man bestimmen, in welcher Schriftart und -größe der Button beschriftet sein soll.

Der Befehl `addActionListener(this)` sorgt dafür, dass diese (this) Anwendung auf einen Buttonklick reagiert. Ohne diesen Befehl könnte man den Button zwar sehen und auch anklicken, es würde aber nichts passieren.

Mit dem letzten Befehl `add(ok)` teilt man der Anwendung mit, dass da überhaupt ein Button vorhanden ist. Würde man diesen Befehl weglassen, wäre der Button zwar da, die Anwendung wüsste aber nichts davon.

```
public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == ok)
       ok.setLabel("geklickt");
}
```

Das ist eine wichtige Schlüsselmethode der Anwendung. Diese Methode überprüft ständig, ob der Benutzer des Programms irgendetwas macht, zum Beispiel einen Button drückt.

Mit `event.getSource()` wird die Quelle der Benutzeraktion festgestellt. Eine Anwendung kann ja viele Buttons, Listboxen, Scrollbalken und andere Komponenten besitzen, die der Benutzer betätigen kann.

Wenn also der Button angeklickt wurde, ist die Bedingung

`event.getSource() == ok`

erfüllt, und der folgende Quelltext wird ausgeführt. Noch ist das nichts Besonderes, es wird lediglich die Beschriftung des Buttons geändert.

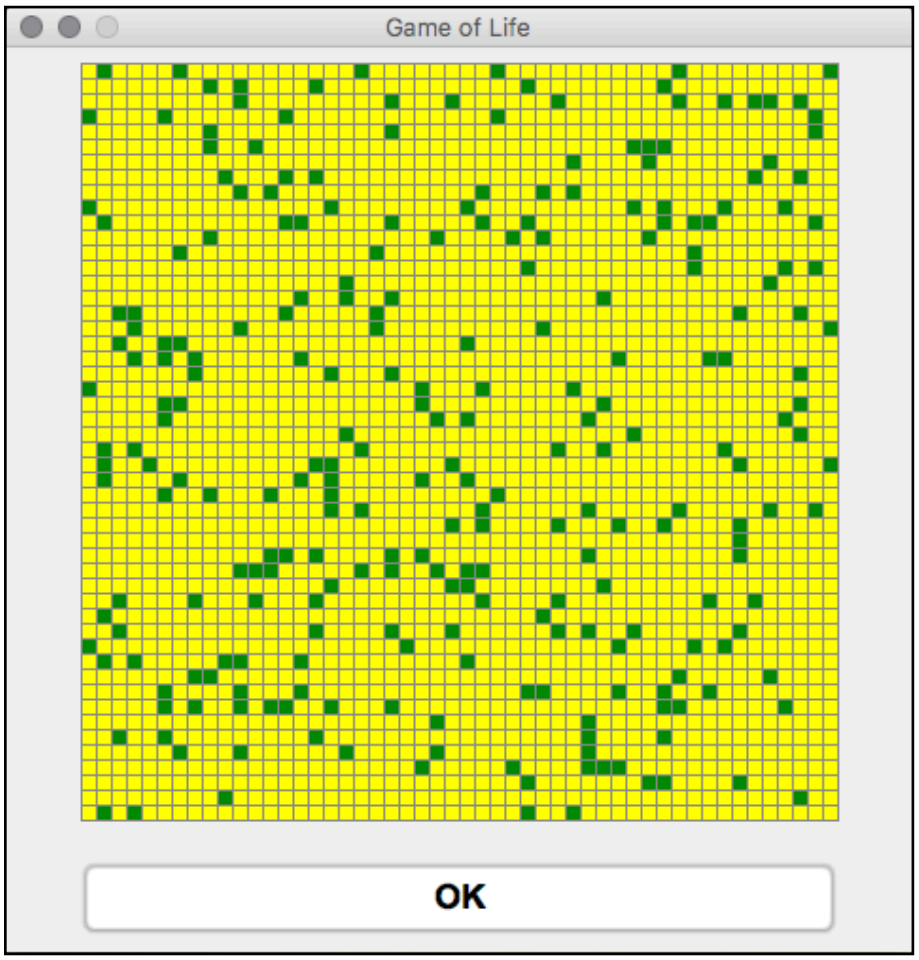
```
public static void main(String[] args)
{
    new Anwendung();
}
```

Das ist nun die Hauptmethode der Anwendung. Immer dann, wenn eine Java-Anwendung gestartet wird, wird als erstes die `main()`-Methode ausgeführt. Über den oder die Parameter dieser Methode müssen wir uns jetzt noch keine Gedanken machen.

Aufgabe dieser `main()`-Methode ist es, die eigentliche Anwendung zu erstellen. Das geschieht über den Befehl `new Anwendung()`.

Schritt 2
Ein Spielfeld erzeugen

Betrachten wir nun folgendes Java-Programm:



Die Anwendung zeigt das Spielfeld von unserem "Game of Life". Wir sehen genau 50 x 50 einzelne Felder, von denen einige zufällig ausgewählte den Zustand "lebend" haben. Die als "lebendig" geltenden Felder sind hier grün (dunkel) gezeichnet. Die anderen (gelben) Felder gelten als "tot".

Verantwortlich für das Erzeugen und Zeichnen des Spielfeldes ist die neue Klasse `Spielfeld`, deren Quelltext wir uns auf der folgenden Seite anschauen.

```
import java.util.Random;
import java.awt.*;

public class Spielfeld
{
     int[][] feld;
     Random rand = new Random();

    public Spielfeld()
     {
        feld = new int [50][50];
        for (int y = 0; y < 50; y++)
           for (int x = 0; x < 50; x++)
           {
              int zufall = rand.nextInt(10);
              if (zufall < 9)
                 feld[x][y] = 0;
              else
                 feld[x][y] = 1;
           }
    }

    public void anzeigen(Graphics g)
    {
       for (int y = 0; y < 50; y++)
           for (int x = 0; x < 50; x++)
           {
              if (feld[x][y] == 1)
                 g.setColor(new Color(0,127,0));
              else if (feld[x][y] == 0)
                 g.setColor(Color.YELLOW);

              g.fillRect(40+x*8, 30+y*8,8,8);
              g.setColor(new Color(127,127,127));
              g.drawRect(40+x*8, 30+y*8,8,8);
           }
    }
}
```

Das Schlüsselelement von `Spielfeld` ist der zweidimensionale int-Array `feld`. Ferner benötigen wir einen Generator für Zufallszahlen, dazu wird die Java-Bibliothek Random mit der Klasse `Random` importiert. Mit der Zeile

```
Random rand = new Random();
```

erzeugen wir ein Objekt der Klasse `Random`, das uns dann als Zufallszahlen-Generator dient.

Im Konstruktur wird der zweidimensionale Array initialisiert, und zwar auf die Größe 50 x 50.

```
        for (int y = 0; y < 50; y++)
           for (int x = 0; x < 50; x++)
           {
              int zufall = rand.nextInt(10);
              if (zufall < 9)
                 feld[x][y] = 0;
              else
                 feld[x][y] = 1;
           }
```

Diese zweifach geschachtelte for-Schleife erzeugt dann die Inhalte der 2500 Felder. Um den Zustand eines Feldes zu bestimmen, wird eine Zufallszahl zwischen 0 und 9 gewürfelt. Das geschieht mittels

```
int zufall = rand.nextInt(10);
```

Die folgende if-Anweisung sorgt dann dafür, dass die meisten Felder den Zustand 0 (tot) erhalten. Nur wenn tatsächlich die Zahl 9 "gezogen" wird, erhält das Feld den Zustand 1 (lebendig).

```
              if (feld[x][y] == 1)
                 g.setColor(new Color(0,127,0));
              else if (feld[x][y] == 0)
                 g.setColor(Color.YELLOW);
```

In der `anzeigen()`-Methode werden alle 2500 Felder in der richtigen Farbe dargestellt. Je nach Zustand des Arrayelements wird entweder die Farbe grün gewählt oder die Farbe gelb.

```
              g.fillRect(40+x*8, 30+y*8,8,8);
              g.setColor(new Color(127,127,127));
              g.drawRect(40+x*8, 30+y*8,8,8);
```

Die erste Zeile malt ein gelb oder grün gefülltes Rechteck in die Java-Anwendung. Die erste Zahl des `fillRect()`-Befehls ist der x-Wert der linken oberen Ecke des Rechtecks. Die zweite Zahl steht für den y-Wert der linken oberen Ecke. Die dritte und die vierte Zahl geben die Breite und die Höhe des Rechtecks an.

Die zweite Zeile setzt die Farbe der Graphikausgabe auf einen mittleren Grauwert. In diesem Grauton wird dann noch einmal der Umriss des Rechtecks nachgezeichnet.

Betrachten wir nun den leicht veränderten Quelltext der eigentlichen Java-Anwendung:

```
{
    Spielfeld welt;
    Button    ok;

    public Anwendung()
    {
        welt    = new Spielfeld();
        setSize(480,500);
        setTitle("Game of Life");
        ...
    }

    public void initComponents()
    {
        ...
    }

    public void actionPerformed(ActionEvent event)
    {
        ....
    }

     public void paint(Graphics g)
     {
         super.paint(g);
         welt.anzeigen(g);
     }

    public static void main(String[] args)
    {
        new Anwendung();
    }
}
```

Viel hat sich hier nicht geändert, daher wurde der Großteil des unveränderten Quelltextes weggelassen (...).

Als Erstes wird ein Objekt der Klasse `Spielfeld` deklariert und im Konstruktor initialisiert.

Die Größe des Anwendungsfensters wurde leicht verändert, damit das Programmfenster symmetrischer aussieht. Neu ist die `paint()`-Methode. Diese Methode ist für das Zeichnen in der Anwendung zuständig, deshalb heißt sie ja auch "paint". Zunächst wird die `paint()`-Methode der übergeordneten Mutterklasse `JFrame` aufgerufen. Dann folgt der Aufruf der `anzeigen()`-Methode des Spieltfeld-Objektes `welt`.

Schritt 3

Es kommt Bewegung in das Spiel

Bisher tut sich noch nichts, wenn man auf den Button klickt. Nun soll Leben in das Spiel kommen. Dazu erst mal ein paar grundsätzliche Gedanken zu dem "Spiel des Lebens".

Die einzelnen Felder repräsentieren lebende oder tote Zellen. Welchen Status eine einzelne Zelle nun hat, hängt von dem Zustand der vier Nachbarzellen ab.

Fall 1: Alle vier Nachbarzellen sind tot

> In diesem Fall soll die betrachtete Zelle absterben, falls sie lebt. Vier tote Zellen sind kein gutes Milieu für eine lebende Zelle. Sie steckt sich an der Krankheit an und stirbt ebenfalls.

Fall 2: Ein, zwei oder drei Nachbarn leben

> In diesem Fall soll die zentrale Zelle leben. Auch dann, wenn sie vorher tot war. Dann vermehrt sich eben eine der benachbarten lebenden Zellen, und es gibt eine lebende Zelle mehr.

Fall 4: Vier Nachbarn leben

> In diesem Fall stirbt die zentrale Zelle. Zu viele lebende Nachbarn sind nicht gut, sie konkurrieren um Wasser und Nährstoffe und nehmen quasi "die Luft zum Atmen weg".

Das sind sozusagen die "Spielregeln" des "Spiel des Lebens". Natürlich kann man diese Spielregeln leicht abändern. Das ist sogar eine der Hauptaufgaben bei der Anwendung des Spiels, die Regeln so abzuändern, dass sich im Laufe der Zeit interessante Muster herausbilden.

Die Klasse `Spielfeld` muss nun recht großzügig erweitert werden. Das ist hauptsächliche Ihre Aufgabe. Aber ein paar kleine Hilfen bekommen Sie noch.

Betrachten wir beispielsweise einmal folgenden Quelltext-Ausschnitt:

```
    public void neuerZyklus()
    {
       for (int y = 0; y < 50; y++)
          for (int x = 0; x < 50; x++)
          {
              int s = nachbarSumme(x,y);
              if (s == 0)
                temp[x][y] = 0;
              else if (s < 3)
                temp[x][y] = 1;
              else
                temp[x][y] = 0;
          }

       for (int y = 0; y < 50; y++)
          for (int x = 0; x < 50; x++)
             feld[x][y] = temp[x][y];
    }
```

Die Methode `neuerZyklus()` berechnet aus den alten Zuständen der 2500 Felder die jeweils neuen Zustände. Somit ist diese Methode eine Schlüsselmethode der ganzen Anwendung.

In der ersten zweifach geschachtelten for-Schleife wird für jede Zelle die Anzahl lebender Nachbarn berechnet. Das übernimmt die sondierende Methode

```
public int nachbarSumme(int x, int y)
```

die Sie selbst noch implementieren müssen. Die Methode liefert eine int-Zahl zwischen 0 (kein Nachbar) und 4 (vier Nachbarn) zurück.

Dann folgt eine geschachtelte if-Anweisung, welche die oben erläuterten Spielregeln umsetzt. Wenn `nachbarSumme()` zum Beispiel den Wert 0 zurückliefert, dann muss die zentrale Zelle sterben. **Allerdings erst in der nächsten Generation, noch nicht in dem aktuellen Zyklus.**

Aus diesem Grund muss der neu berechnete Zustand zwischengespeichert werden. Dazu dient der Hilfsarray `temp`. Dieser Hilfsarray ist - wie der Hauptarray `feld` - ein zweidimensionaler Array aus 50 x 50 Elementen. Der Hilfsarray kann also die künftigen Zustände aller 2500 Zellen speichern.

Erst wenn von jeder Zelle der Folgezustand berechnet ist, kann die nächste Generation starten. Dazu dient die zweite geschachtelte for-Schleife. Die in dem Hilfsarray `temp` zwischengelagerten neuen Zustände werden jetzt in den Hauptarray `feld` übertragen.

Eine Torus-Welt

Wenn Sie die Zahl der lebenden Nachbarn berechnen, müssen Sie darauf achten, dass es sich bei der Spielwelt um eine sogenannte Toruswelt handelt.

Stellen Sie sich die Welt zunächst als großes Quadrat aus 50 x 50 Zellen vor. Nun nehmen wir das Quadrat und machen einen Zylinder daraus. Das heißt, wir verbiegen es so, dass das rechte Ende des Quadrats mit dem linken Ende verbunden wird. Das ist genau so, wie wenn Sie ein Blatt Papier zu einer Röhre verkleben.

Das Ergebnis dieser Operation: Der rechte Nachbar ein Zelle, die am rechten Rand liegt, ist eine Zelle am linken Rand, die in gleicher Höhe liegt.

Konkret: Der rechte Nachbar der Zelle `feld[49][12]` ist die Zelle `feld[0][12]`, und der linke Nachbar der Zelle `feld[0][12]` ist die Zelle `feld[49][12]`.

Diese Nachbarzellen müssen also mitgezählt werden, wenn wir die Anzahl der lebenden Nachbarzellen berechnen wollen.

Aber wir sind noch nicht am Ende. Die Welt ist ja keine Zylinder-Welt, sondern eine Torus-Welt. Wir müssen unseren Zylinder nun so verbiegen, dass das obere Ende des Zylinders mit dem unteren Ende verbunden wird. Es entsteht dann so etwas wie ein Fahrradschlauch oder ein Autoreifen.

Konkret: Der obere Nachbar der Zelle `feld[12][0]` ist die Zelle `feld[12][49]` und umgekehrt.

Wenn wir also wissen wollen, ob der obere Nachbar einer Zelle lebt, müssen wir schreiben:

```
    private int nachbarOben(int x, int y)
    {
       if (y == 0)
          return feld[x][49];
       else
          return feld[x][y-1];
    }
```

Mit der if-Bedingung überprüfen wir, ob wir eine obere Zelle untersuchen `(y == 0)`. Ist dies der Fall, müssen wir die Zelle `feld[x][49]` untersuchen. Trifft die Bedingung nicht zu, untersuchen wir einfach die unterhalb der Zelle liegende Zelle. Da wir nur zwei Zustände haben, die durch die Werte 0 (tot) und 1 (lebend) repräsentiert werden, ist die

Ermittlung der Anzahl der lebenden Nachbarzellen recht simpel. Man muss nur die Zustände der Nachbarzellen addieren.

Schritt 4

Der Button wird aktiv

Bisher tut sich immer noch nichts, wenn man auf den Button klickt. Das wollen wir jetzt mit folgender Ergänzung der Klasse Anwendung ändern:

```
    public void actionPerformed(ActionEvent event)
    {
       if (event.getSource() == ok)
       {
          welt.neuerZyklus();
          repaint();
       }
    }
```

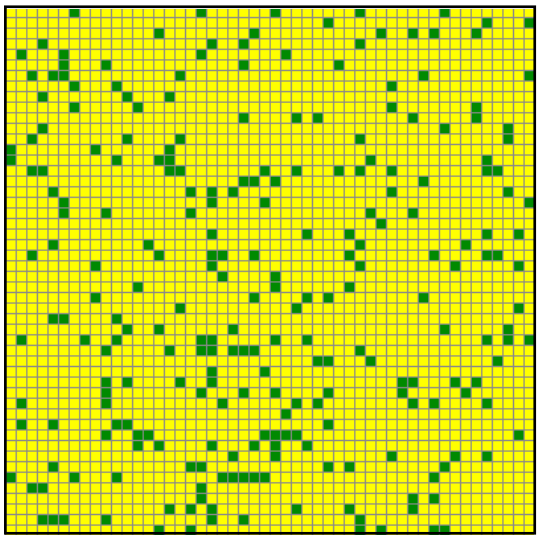
Wie Sie sehen, muss man nur die Methode actionPerformed() etwas ändern, und schon funktioniert das Programm.

Wenn der OK-Button geklickt wurde, wir die Methode `neuerZyklus()` der Klasse `Spielfeld` aufgerufen. Die eigentlichen Berechnungen finden dann in den Methoden dieser Klasse statt.

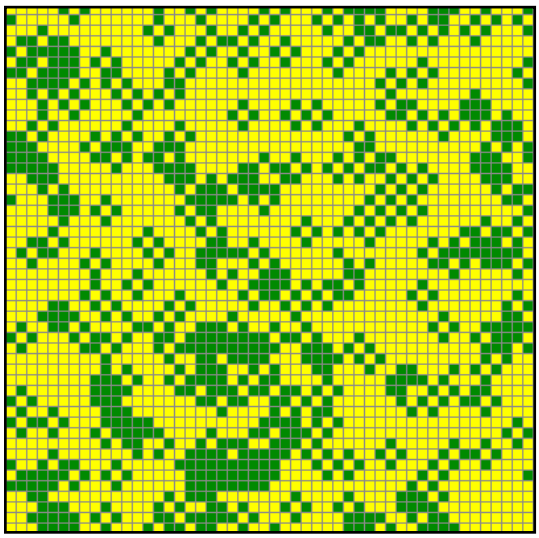
Wenn das `Spielfeld`-Objekt welt mit den Berechnungen fertig ist, muss die Anwendung neu gezeichnet werden. Dafür sorgt die wichtige Methode `repaint()`, die auf gar keinen Fall vergessen werden darf. Die Methode `repaint()` macht eigentlich nichts anderes, als die Methode `paint()` erneut aufzurufen.

Als Nächstes wollen wir testen, ob alles so funktioniert, wie wir es uns vorgestellt haben.

Hier ein Screenshot vom Ausgangszustand unserer Welt:



Jetzt klicken wir den OK-Button an und erhalten:



Mit weiteren Buttonklicks verändert sich die Spielwelt immer weiter.

Aufgaben

1. Bringen Sie das Spiel zum Laufen, indem Sie sich die vorgegebenen Quelltexte auf Ihr Laufwerk kopieren und dann die Klasse Spielfeld vervollständigen
2. Spielen Sie mit den Spielregeln herum. Die vorgegebenen Regeln waren ja recht einfach, es wurden nur drei Fälle unterschieden.
3. Bisher wurden nur vier Nachbarn berücksichtigt. Erweitern Sie das Spiel so, dass auch die diagonalen Nachbarn ausgewertet werden.