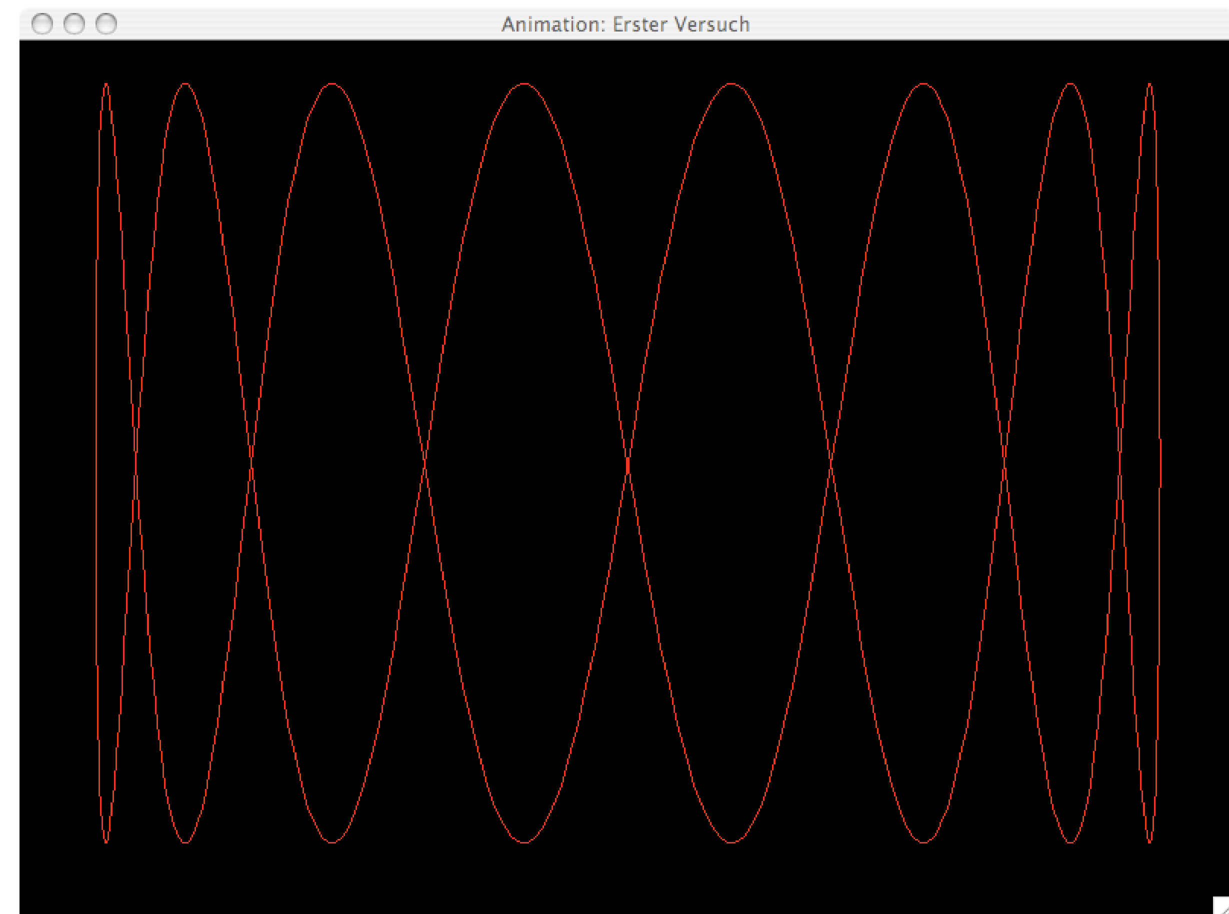
Animierte Grafik

Animation: Versuch 1

```
public static void warte(long ms) {
   try { Thread.sleep(ms); }
  catch(InterruptedException e) {}
 }
```

- wartet *ms* Millisekunden
- `repaint()` fordert Neu-Zeichnen an (z.B. in Endlos-Schleife)
- flackert normalerweise



⇒ Programm `Animation1.java`

Fenster-Aktualisierung

Klasse `Frame`

Eigene Klasse

Windowmanager

`repaint()`

`update()`

1. füllt Fenster mit Hintergrund-Farbe
2. ruft `paint()` auf

`paint()`

Fenster-Aktualisierung

Klasse `Frame`

Eigene Klasse

Windowmanager

`repaint()`

`update()`

1. füllt Fenster mit Hintergrund-Farbe
2. ruft `paint()` auf

`paint()`

Fenster-Aktualisierung

Klasse `Frame` — Eigene Klasse

Windowmanager

`repaint()`

`update()` (Klasse Frame)

`update()` (Eigene Klasse)

überschreiben

`paint()`

1. füllt Fenster mit Hintergrund-Farbe
2. ruft `paint()` auf

Offscreen-Images

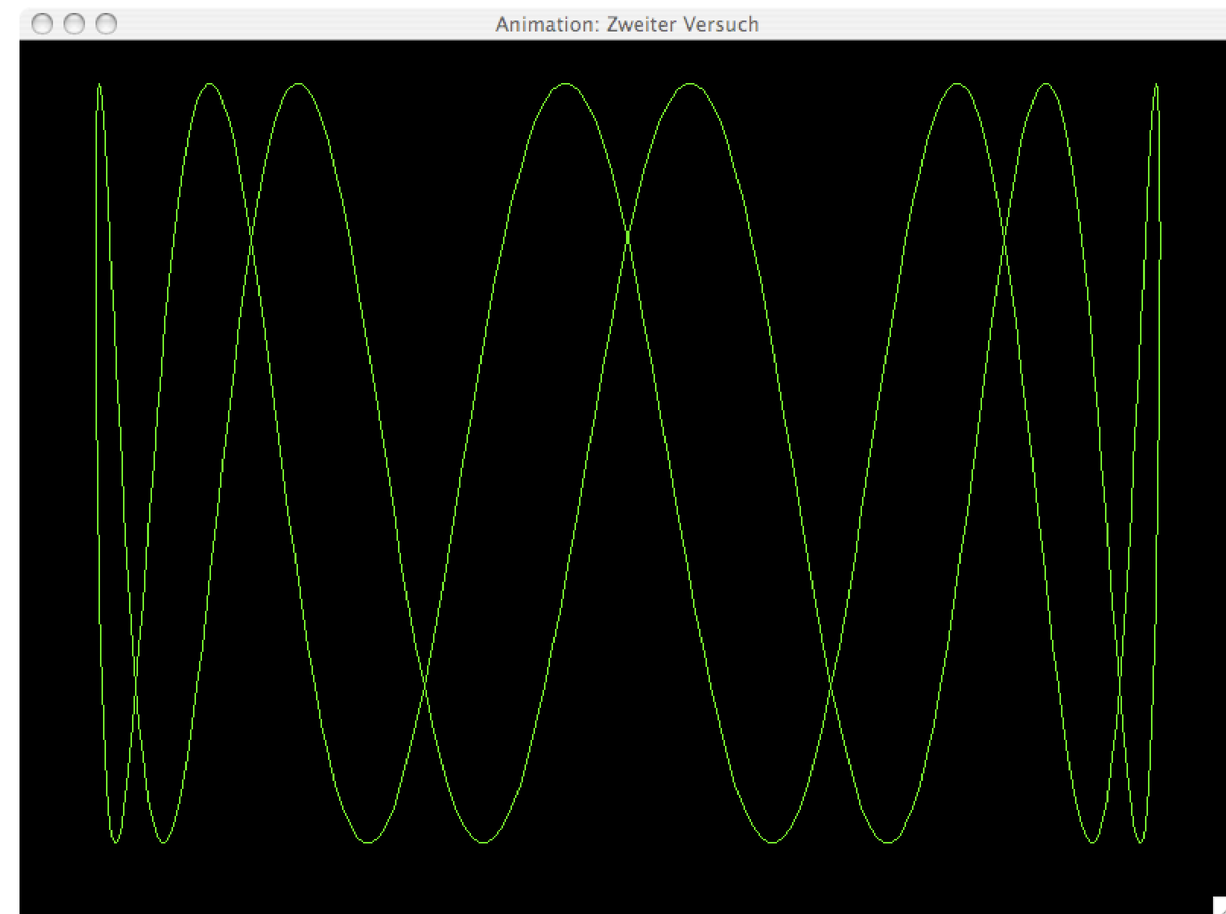
- `createImage(int` *`b`*`, int` *`h`*`)` erzeugt `Image` der Größe *b*, *h*
  Achtung: Funktioniert nicht an beliebiger Stelle – bei Bedarf z.B. in `paint()` erzeugen
- Methode `getGraphics()` der Klasse `Image` liefert `Graphics`-Kontext
- `drawImage(Image` *`img`*`, int` *`x`*`, int` *`y`*`, ImageObserver` *`o`*`)` zeichnet *img* mit linker oberer Ecke an Position `(`*`x`*`,` *`y`*`)`
  Für *o* kann „`this`" eingesetzt werden

Animation: Versuch 2

- Methode `update()` überschreiben:

```
public void update(Graphics g) {
  paint(g);
 }
```

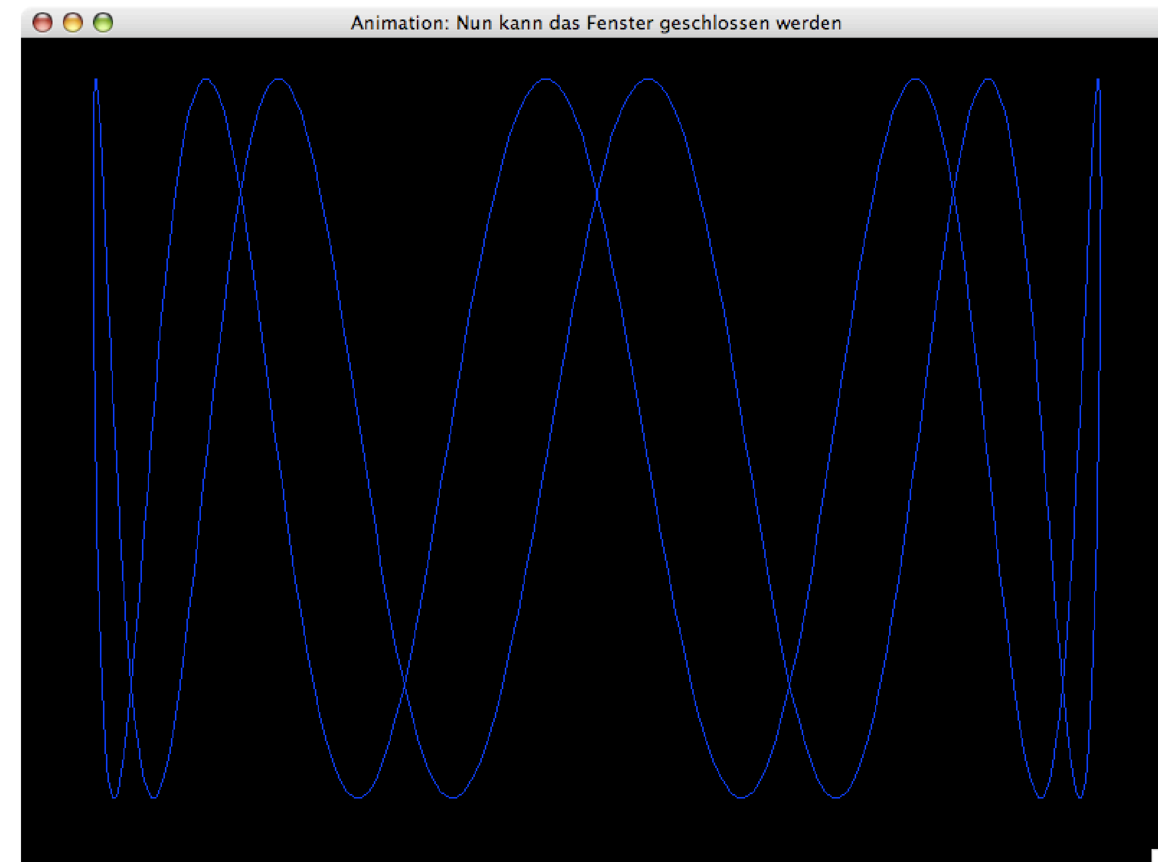
- Methode `paint()`:
  - ★ Grafik in Offscreen-Image aktualisieren
  - ★ Offscreen-Image → Bildschirm



⇒ Programm `Animation2.java`

Fenster sauber schließen

- `java.awt.event` einbinden
- Interface `WindowListener` implementieren
- Methode `windowClosing(WindowEvent e)`:
  1. `setVisible(false)`: Fenster vom Bildschirm entfernen
  2. `dispose()`: Aufräumen
  3. `System.exit(0)`: Programm verlassen
- Rest vom Interface: leer
- `addWindowListener(this)`: `Listener` beim Fenster anmelden



⇒ Programm `Animation3.java`

Knöpfe und anderer Schnickschnack

Klasse `Button` (Knöpfe)

- Konstruktor `Button(String` *`Text`*`)` erzeugt Knopf mit Inhalt „*`Text`*“
- Methode `setLabel(String` *`Text`*`)` ersetzt den aktuellen Text des Knopfs durch „*`Text`*“
- Methode `addActionListener(ActionListener` *`l`*`)` aktiviert `Listener` für den Knopf

  (wenn wir das Interface `ActionListener` in der aktuellen Klasse implementieren, können wir „`this`“ für *`l`* einsetzen)

Klasse `TextField`

- Konstruktor `TextField(String` *`Text`*`, int` *`l`*`)` erzeugt Eingabefeld, initialisiert es mit „*`Text`*“ & reserviert *`l`* Zeichen für die Eingabe
- Methode `getText()` gibt den (ggfs. geänderten) Inhalt des Eingabefeldes als `String` zurück
- Methode `addActionListener(ActionListener` *`l`*`)` aktiviert `Listener` für das Eingabefeld (vgl. Klasse `Button`)

Klasse `Label`

- Konstruktor `Label(String` *`Text`*`)` erzeugt Textfeld und füllt dieses mit „*`Text`*"
  Die Größe des Textfeldes wird dabei durch die Länge der Zeichenkette *`Text`* festgelegt!
- Methode `setText(String` *`Text`*`)` ersetzt den aktuellen Text des Textfeldes durch „*`Text`*"

- *`Objekt`*e wie Knöpfe, Textfelder & Labels werden mit der Methode `add(`*`Objekt`*`)` zum Fenster hinzugefügt

Layout-Manager

Die `Objekte` sind im Fenster zu verteilen:

- `setLayout(null)` erzeugt ein sog. Null-Layout (leider das Standard-Layout für Java-Programme). Alle Komponenten müssen dann ihre Position & Größe selbst festlegen – z.B. mit der Methode `setBounds(int x, int y, int breite, int hoehe)`
- Einfacher: verschiedene vorgefertigte Layouts – insbesondere `FlowLayout()` – verteilen `Objekte` (Knöpfe, ...) automatisch

```
setLayout(new FlowLayout());
```