Bitte lösen Sie Aufgabe 20-21 bis zum 31.05.2020 **und geben Aufgabe 21 ab!**

Ziele des heutigen Übungstermins

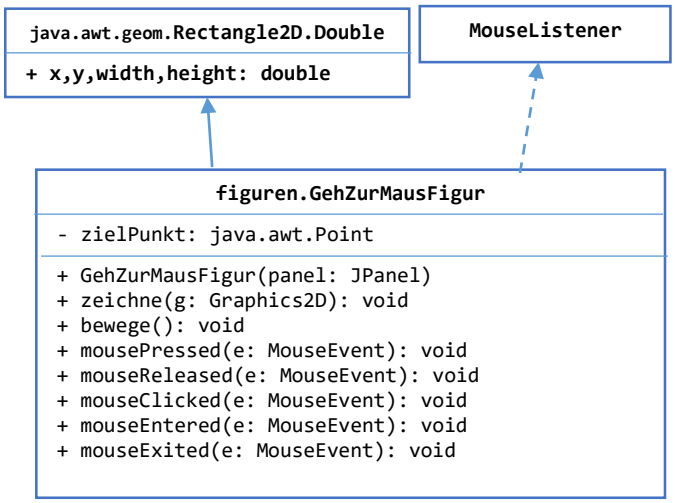
- Sie schreiben eine Basisklasse, die in Zukunft die Entwicklung von Spielen stark vereinfacht.
- Sie lernen folgende Konzepte der objektorientierten Programmierung in Java kennen:
  - Von **abstrakten Klassen** kann man keine Objekte erzeugen, sie dienen lediglich als Basisklassen.
  - **Abstrakte Methoden** sind Platzhalter, die von abgeleitete Klassen überschrieben werden müssen.
  - **final-Methode** können von abgeleiteten Klassen nicht mehr überschrieben werden.

Aufgabe 20

Bessere Grafik, erste Spielelemente, Vererbung, Struktur

Wir werden als ***Aufgabe20*** ein kleines Spiel programmieren, in dem eine Spielfigur auf Mausklick zum Mauszeiger läuft und eine andere Figur der Maus folgt.

Für die Spielfiguren schreiben wir eigene Klassen, um die Anwendung besser zu strukturieren.



Implementieren Sie zunächst das Gerüst der Klasse ***GehZurMausFigur*** anhand des UML Diagrammes. Wenn Sie beim Anlegen der Klasse in Eclipse gleich angeben, dass das Interface *MouseListener* implementiert werden soll, werden die fünf Methoden des Interface automatisch erzeugt. **Achtung:** die Klasse ***java.awt.geom.Rectangle2D.Double*** verwenden.

Dann erweitern Sie folgende Methoden:

Der **Konstruktor** speichert in *zielPunkt* ein neues *Point*-Objekt mit den Koordinaten (200,200), setzt *this.height* und *this.width* auf 40 und registriert sich als *MouseListener* von *panel* (*panel.addMouseListener(this)*).

Die Methode ***zeichne*** stellt ein grünes Rechteck dar, indem sie *g.setColor(Color.GREEN)* und `g.fill(this)` ruft. Da *GehZurMausFigur* von *Rectangle2D.Double* abgeleitet ist, ist *this* ein Rechteck mit der linken oberen Ecke (*this.x, this.y*) und der Größe *this.width* und *this.height*.

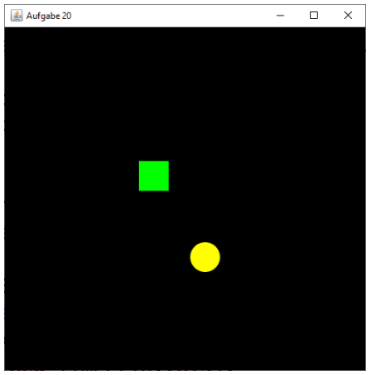
Die Methode ***bewege*** soll die Spielfigur ein kleines Stück in Richtung auf den Zielpunkt bewegen. Dazu führt sie folgende Berechnungen:

$$this.x = \text{this.x} + \frac{\text{zielPunkt.getX()} - \text{this.x}}{10}$$

$$this.y = \text{this.y} + \frac{\text{zielPunkt.getY()} - \text{this.y}}{10}$$

Der Wert 10 im Nenner steuert die Geschwindigkeit, mit der die Figur sich in Richtung Maus bewegt. Da wir die jeweils noch fehlende Strecke durch 10 teilen, bremst die Figur vor dem Zielpunkt ab.

Die Methode ***mousePressed*** speichert die Cursorposition in der Instanzvariable *zielPunkt*. Die Position des Cursors bekommen wir aus dem *MouseEvent* Objekt, das der Methode *mousePressed* übergeben wurde: this.zielPunkt = *e.getPoint()*.



Die eigentliche Anwendung ist die Klasse *Aufgabe20*:

Fangen Sie wieder damit an, dass Sie das Gerüst anhand des UML Diagramms programmieren.

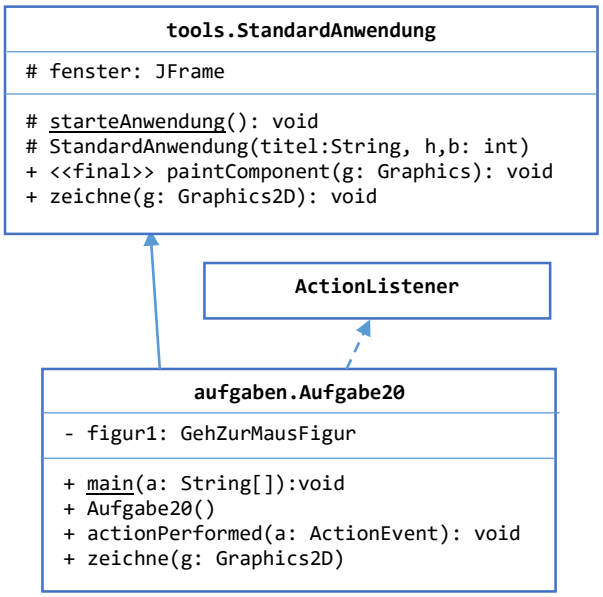
Im **Konstruktor** von ***Aufgabe20*** speichern Sie in *figur1* ein neues *GehZurMausFigur* Objekt, dessen Konstruktor übergeben Sie *this*. Dann erzeugen Sie einen *Timer* mit einem Intervall von 40 Millisekunden und

this als *ActionListener* (**wichtig: verwenden Sie javax.swing.Timer).** Danach starten Sie den Timer und färben den Hintergrund des Spiels Schwarz.

In ***actionPerformed*** rufen Sie *figur1.bewege()* und *this.repaint()*.

In ***zeichne*** rufen Sie *figur1.zeichne(g)*.

Jetzt springt bei Klick die Figur an die Position des Cursors, allerdings ist nicht die Mitte des Rechtecks, sondern die linke obere Ecke an der Cursorposition. Ändern Sie die Methode *bewege* so, dass die Mitte der Figur zum Cursor geht, verwenden Sie dabei *this.width* und *this.height.*



Die Klasse *Aufgabe20* ist eine einfache Blaupause für ein typisches Computerspiel:

Ein Timer sorgt für eine feste Bildrate, bei jedem Takt des Timers wird der neue Spielstand berechnet (in diesem Fall ist es nur die Bewegung der Spielfigur) und das Spiel neu gezeichnet.

Die Spielfigur interagiert selbständig mit dem Spiel indem Sie auf einen Mausklick innerhalb des Spielfeldes reagiert.

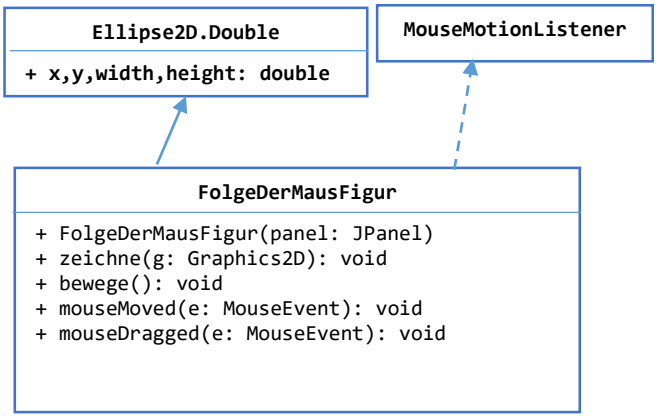
Erweitern Sie das Spiel um eine zweite Spielfigur, indem Sie die Klasse ***FolgeDerMausFigur*** schreiben. Die Basisklasse ist hier *Ellipse2D.Double*, so dass die Figur als Kreis dargestellt wird. Sie implementiert *MouseMotionListener* und reagiert auf Bewegungen der Maus.

In ***mouseMoved*** wird *this.x auf e.getX()* und analog *this.y* auf *e.getY()* gesetzt, in ***zeichne*** verwenden Sie die Farbe Gelb, die Methode ***bewege*** bleibt leer.

Fügen Sie ***Aufgabe20*** die Instanzvariable *figur2* vom Typ *FolgeDerMausFigur* hinzu, die im Konstruktor initialisiert wird, und in *bewege* bewegt und in *zeichne* dargestellt wird.

Verbessern Sie *FolgeDerMausFigur* so, dass der Cursor im Mittelpunkt der Figur liegt.

Schlagen Sie in der online Java-Doc die Klassen *Graphics2D*, *javax.swing.Timer*, *Rectangle2D.Double*, *Ellipse2D.Double* sowie die Interfaces *MouseListener* und *MouseMotionListener* nach.



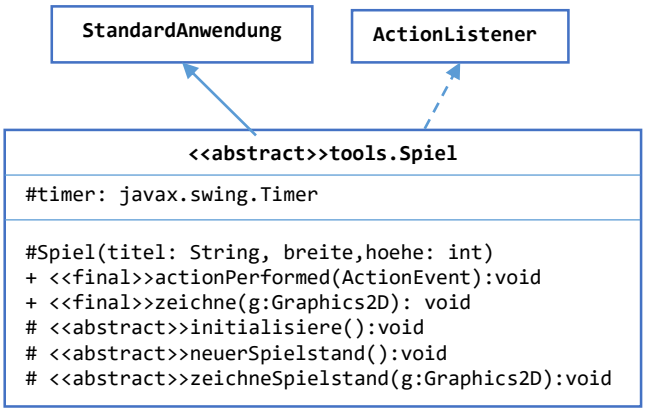
Aufgabe 21

Basisklasse für Spiel: mehr Struktur

Da wir im Laufe des Semesters mehrere kleine Spiele programmieren wollen, schreiben wir die Klasse ***Spiel*** anhand des UML Diagramms. Diese Klasse ist genau wie *StandardAnwendung* abstrakt:
`public abstract class Spiel extends …` d.h. man kann von ihr keine Objekte erzeugen, sondern sie dient nur als Basisklasse. Sie enthält drei **abstrakte** Methoden, diese haben anstatt eines Methodenrumpfs ein Semikolon:

```
protected abstract void initialisiere();
protected abstract void neuerSpielstand();
protected abstract void zeichneSpielstand(Graphics2D g);
```

Abgeleitete Klassen **müssen** diese abstrakten Methoden implementieren.



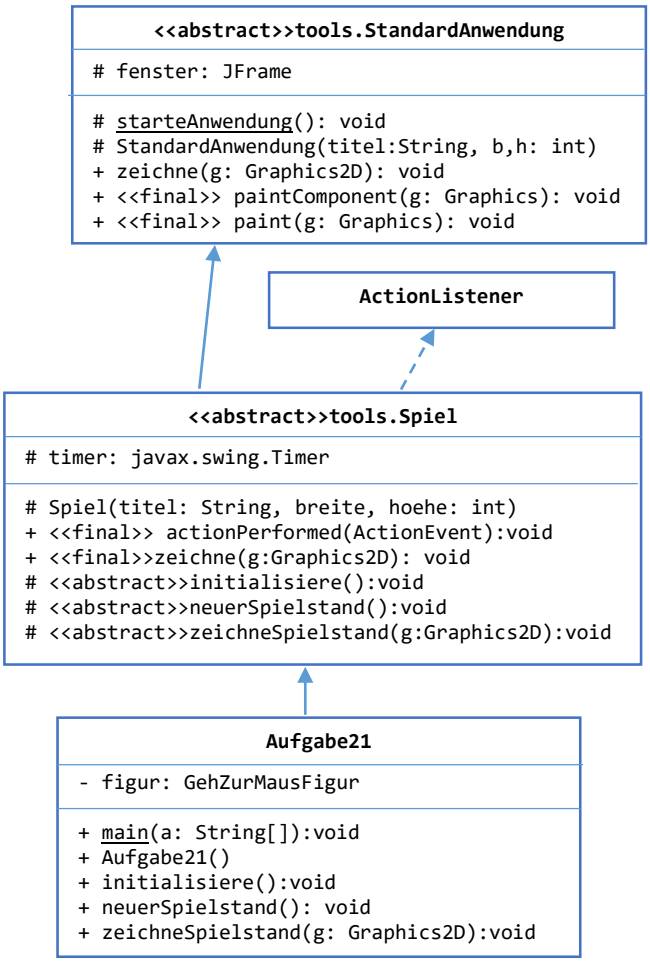
Der **Konstruktor** gibt seine Parameter an den Basisklassenkonstruktor weiter und setzt die Farbe des Hintergrunds auf Schwarz. Dann erzeugt er den Timer mit einer Periode von 40 Millisekunden und *this* als *ActionListener*, danach ruft er die Methode *initialisiere*.
Wichtig: **erst nach dem Aufruf von *initialisiere* wird der Timer gestartet**. Auf diese Weise kann die abgeleiteten Klasse erst alle Spielfiguren initialisieren, bevor das Spiel zum ersten Mal dargestellt wird.

Die Methoden ***actionPerformed*** und ***zeichne*** bekommen den Zusatz *final*:
`public final void actionPerformed(…`
damit wird verhindert, dass abgeleitete Klassen diese Methoden überschreiben können.

actionPerformed ruft *neuerSpielstand* und danach *repaint* um das Spiel neu darzustellen.

zeichne ruft *zeichneSpielstand*.

Jetzt schreiben Sie die von *Spiel* abgeleitete Klasse ***Aufgabe21***. Die vollständige Hierarchie unserer Klassen ist im UML Diagramm dargestellt.



Der **Konstruktor** ruft nur den Basisklassenkonstruktor.

Die Methode ***initialisiere*** erzeugt die Spielfigur. **Machen Sie sich klar, wann diese Methode ausgeführt wird: im Basisklassenkonstruktor vor dem Starten des Timers.**

Die Methode ***neuerSpielstand*** bewegt die Figur(en).

Die Methode ***zeichneSpielstand*** zeichnet die Figur(en).

Durch Vererbung konzentrieren wir Programmlogik, die wir mehrfach brauchen werden, in Basisklassen. Dabei haben wir folgende Konzepte verwendet:

- **Abstrakte Klassen** kann man nicht instanziieren, also keine Objekte davon erzeugen.
- **Abstrakte Methoden** müssen in abgeleiteten Klassen implementiert werden.
- Methoden, die ***final*** deklariert sind, kann man nicht mehr überschreiben.

Wichtig: Reihenfolge der Aufrufe

Machen wir uns klar, in welcher Reihenfolge die Methoden im Spiel durchlaufen werden:

main(…)	wird vom Betriebssystem gerufen, ruft *starteAnwendung*
starteAnwendung()	erzeugt ein Objekt vom Typ *Aufgabe21*, ruft also den Konstruktor
Aufgabe21()	ruft *super(…)*, also den Konstruktor von *Spiel*
Spiel(…)	ruft *super(…)*, also den Konstruktor von *StandardAnwendung*
StandardAnwendung(…)	erzeugt das Fenster, legt das Aufgabe21 Objekt auf das Fenster
tick = new Timer(…)	erzeugt den Timer, startet ihn aber nicht!
initialisiere()	**ruft initialisiere in Aufgabe21, bevor der Timer läuft**
tick.start()	erst jetzt wird der Timer gestartet
……	im Konstruktor von Aufgabe 21 nach super(), der Timer tickt jetzt schon im Hintergrund!

Die Reihenfolge ist deshalb so wichtig, weil

- die Methode *initialisiere()* von Aufgabe21 aufgerufen wird **bevor** der Timer gestartet wird,
- im Konstruktor von Aufgabe21 nach super() läuft der Timer schon, es kann also zu spät sein, dann noch Variablen zu initialisieren.

Deshalb ist wichtig: Alle Instanzvariablen (Spielfiguren, Sound, u.s.w.), die in *neuerSpielstand* oder *zeichneSpielstand* verwendet werden, müssen in initialisiere() erzeugt werden, im Konstruktor von Aufgabe21 ist es zu spät, falls der Timer schon vorher zum ersten Mal feuert.

Falls Sie in *neuerSpielstand()* oder *zeichneSpielstand()* den Fehler *NullPointerException* bekommen, liegt es wahrscheinlich daran, dass Sie eine Instanzvariable gar nicht oder zu spät initialisiert haben.

Zusammenfassung der wichtigsten Punkte

- Die Klasse *JPanel* besitzt eine Methode `public void paintComponent(Graphics g)` die vom Display Manager immer dann gerufen wird, wenn das Panel auf dem Bildschirm neu dargestellt wird. Überschreibt man diese Methode in einer abgeleiteten Klasse, kann man über das *Graphics* Objekt auf dem JPanel zeichnen.
- Der Parameter von *paintComponent* zeigt auf ein Objekt der von *Graphics* abgeleiteten Klasse ***Graphics2D***.
- Wenn eine Klasse ***abstract*** deklariert ist, kann man von ihr keine Objekte erzeugen, sondern sie nur als Basisklasse verwenden.
- Eine **abstrakte Methode**, hat anstelle des Methodenrumpfes ein Semikolon:
  ```
  public void lesen();
  ```
  Eine Klasse, die mindestens eine abstrakte Methode enthält, muss selbst abstrakt sein. Abgeleitete Klassen müssen die abstrakten Methoden implementieren oder selbst abstrakt sein.
- Ein Interface enthält ausschließlich abstrakte Methoden.
- Für das Abfangen von Ereignissen gibt es das Interface ***ActionListener***, es enthält nur eine Methode: `public void actionPerformed`.
- Will man auf einen Klick auf einen *JButton* reagieren, registriert man ein Objekt einer Klasse, die ***ActionListener*** implementiert, auf den Button. Dazu ruft man die Methode *addActionListener*.
- Um auf Maus-Klicks innerhalb eines Panels zu reagieren, registriert man auf dem Panel ein Objekt einer Klasse, welche ***MouseListener*** implementiert.
- Will man auf Maus-Bewegungen innerhalb eines Panels reagieren, registriert man auf dem Panel ein Objekt einer Klasse, welche ***MouseMotionListener*** implementiert.
- Die Methoden von *MouseListener* und *MouseMotionListener* bekommen einen Parameter vom Typ ***MouseEvent***, von ihm kann man z.B. mit `getPoint()`die Position, an dem der Klick erfolgt ist, abfragen.
- Ein Objekt vom Typ ***java.util.Random*** erzeugt eine zufällig verteilte Zahlenfolge. Die Methode **`nextInt(max:int)`** erzeugt die nächste Zahl der Folge, der Wert liegt zwischen 0 und max-1.
- Mit der *if*-**Anweisung** kann man eine **Verzweigung** realisieren.
  Die Anweisungen hinter der *if*-Anweisung werden nur ausgeführt, wenn die Bedingung *true* ist. Optional kann ein ***else*-Zwe**ig hinzugefügt werden.

- Verfahren zur Datenreduktion sind entweder **verlustfrei** oder **verlustbehaftet**.
- Ein Datenkompressionsverfahren ist **symmetrisch**, wenn es etwa den gleichen Aufwand für die Kompression wie für die Dekompression braucht, andernfalls nennt man es asymmetrisch.
- **Lauflängencodierung** (RLC) arbeiten mit Mustererkennung und ersetzen häufig vorkommende Muster durch kurze Zeichenfolgen. RLC Verfahren sind verlustfrei und asymmetrisch.
- **Quantisierung (sampling)** ist die Speicherung eines Signals in einer begrenzten digitalen Auflösung. Sie ist verlustbehaftet und symmetrisch.

Eine Form der Quantisierung ist die **analog-digital (AD) Wandlung**. Ein anschauliches Beispiel für die AD Wandlung ist die Abtastung eines Tonsignals mit einer festen **Abtastfrequenz** [Hz].
Die **Auflösung der Amplitude** wird dabei ähnlich wie die Farbtiefe von Bildern in **Bit** angegeben, dabei bedeuten 10 Bit dass 2^{10} also 1024 unterschiedliche Werte unterschieden werden können.