Bitte geben Sie ***Aufgabe12**.java*, ***Snake.java*** sowie ***Farbkreis.java*** bis zum 15.11.2020 ab!

Aufgabe 12

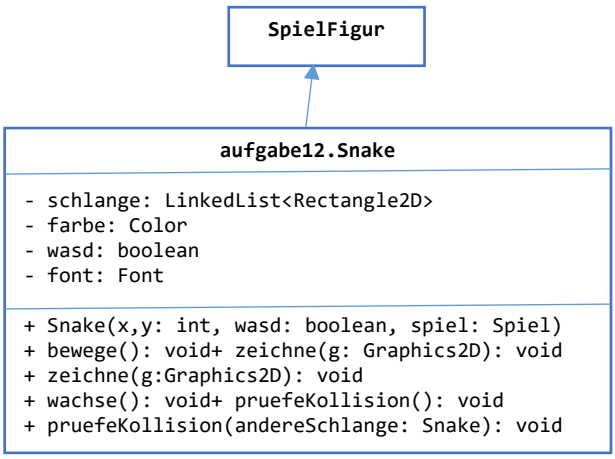
Spiel: **Grafik, LinkedList, Schleife**

In dieser Aufgabe beginnen wir mit der Programmierung des Spiels **Snake** für zwei Spieler. Wir verwenden dabei unsere Klassen *Spiel* und *Spielfigur*, die sie als UML Diagramme im ersten Übungsblatt finden.

Eine erste Version der Klasse Snake

Erzeugen Sie das Paket ***aufgabe12*** und darin die **Klasse *Snake***, einer Spielfigur, die mit der Tastatur gesteuert wird.

Implementieren Sie zunächst das Gerüst der Klasse anhand des nebenstehenden UML Diagramms.



Die Variable *schlange* wird mit dem Standardkonstruktor initialisiert, ***farbe*** mit Color.GREEN und ***font*** als `new Font("Helvetica",Font.BOLD,30)`.

Der **Konstruktor**

- ruft den Konstruktor der Basisklasse, mit der Position (x,y) und der Größe 20*20,
- speichert *wasd* in der gleichnamigen Instanzvariable,
- aktiviert die Tastensteuerung mit *(wasd,20)*
- ruft *this.setBewegung(-20,0)*, damit die Figur sich am Anfang in jedem Frame um 20 Pixel nach links bewegt,
- ruft *this.setRandReaktion(RAND_STOP)* damit die Figur stoppt, wenn sie den Rand berührt,
- setzt *farbe* auf Gelb, falls *wasd true* ist.

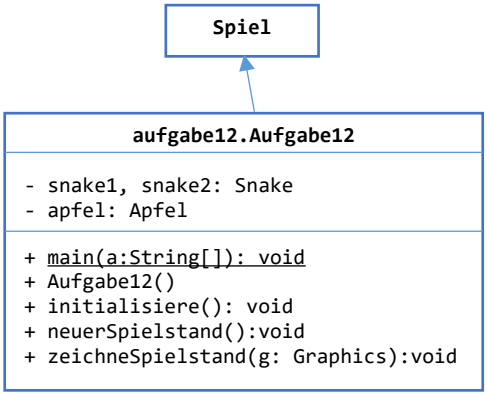
Die überschriebene Methode ***bewege*** ruft zunächst nur *super.bewege()*.

Im ebenfalls überschriebenen ***zeichne*** wird in der Farbe *farbe* das Rechteck *this* gezeichnet.

Die Methoden ***pruefeKollision*** und ***wachse*** bleiben zunächst leer.

Eine erste Version des Spiels

Schreiben Sie zunächst eine **komplett leere** öffentliche Klasse ***aufgabe12.Apfel*** und dann das Gerüst der Klasse ***Aufgabe12*** anhand des nebenstehenden UML Diagramms.



In ***main*** wird wie gewohnt `starteAnwendung` gerufen.

Der ***Konstruktor*** ruft *super* mit der Größe 1000*800 Pixel.

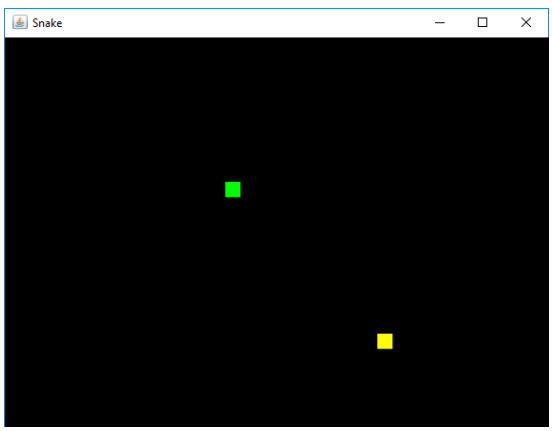
Die Methode ***initialisiere*** initialisiert die Spielfiguren *snake1* und *snake2* an verschiedenen Positionen und übergibt einmal *true* und einmal *false* als Wert von *wasd*. Die x- und y-Positionen der Figuren sollten durch 20 teilbar sein!
Dann ruft sie *this.timer.setDelay(250)* .

Die Methode ***neuerSpielstand*** bewegt beide Spielfiguren.

Die Methode ***zeichne*** stellt die beiden Spielfigur dar.

Wir haben jetzt ein Spiel, in dem zwei Quadrate mit den Pfeiltasten bzw. den Tasten *WASD* gesteuert werden. Die Figuren laufen beim Start des Spiels sofort nach links los. Durch den Takt von 250 Millisekunden bewegen sich die Figuren ruckartig über das Spielfeld.

Die Variable *apfel* wird bisher noch nicht verwendet.



Eine Schlange aus 6 Segmenten

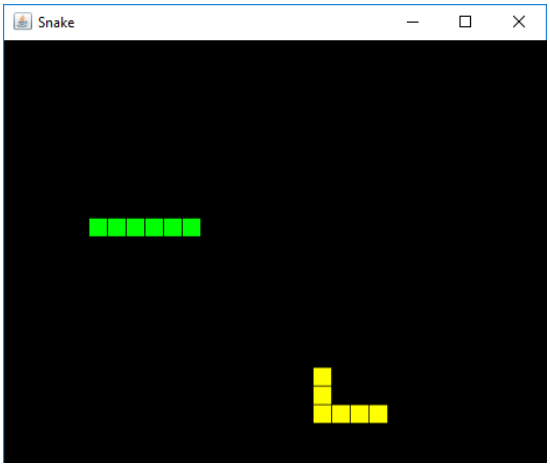
Damit aus den Quadraten eine Schlange wird, erweitern wir die Klasse **Snake** wie folgt:

Der **Konstruktor** erzeugt 6 Objekte vom Typ *Rectangle2D.Double* und fügt sie zu der Liste *schlange* hinzu (Methode *add*). Es sollen Quadrate mit einer Kantenlänge von 20 Pixeln sein. Die Position des ersten Quadrats ist *(this.x, this.y)*, jedes weitere Quadrat ist um 20 Pixel nach rechts verschoben. Sie können 6 mal die Methode *add* rufen oder eine f*or*-Schleife programmieren.

Die Methode ***bewege*** ruft zunächst *super.bewege*. Die folgenden Schritte werden nur durchgeführt, wenn sich die Schlange tatsächlich bewegt, das prüfen wir so:

```
if(this.bewegung.distance(0,0)>0){…  :
```

- Hole das letzte Element aus der Liste (*removeLast*) und speichert es in der lokalen Variablen *kopf* vom Typ *Rectangle2D*. Dieses Rechteck ist bisher das Ende der Schlange, es wird gleich zum neuen Kopf der Schlange werden.
- Setze seine Position mit *kopf.setFrame(this)* auf die aktuelle Position der Spielfigur (die sich ja gerade durch *super.bewege* bewegt hat).
- Füge es mit *addFirst* wieder in die Schlange ein.



In ***zeichne*** führen wir eine Schleife über alle Segmente der Schlange durch:

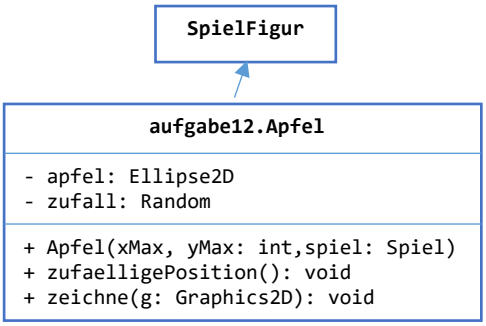
```
for(Rectangle2D segment:schlange){ …
```

Jedes Segment füllen wir zunächst mit *g.fill* in der Farbe *farbe* und zeichnen dann seinen Umriss mit *draw* nochmals in Schwarz um die Segmente optisch voneinander abzugrenzen.

Die Spielfiguren bewegen sich jetzt schon so wie im endgültigen Snake Spiel.

Ein roter Apfel

Als nächstes implementieren wir die von *SpielFigur* abgeleitete öffentliche Klasse ***Apfel*** (die bisher leer ist). Die Instanzvariable ***apfel*** initialisieren wir mit dem Standardkonstruktor von *Ellipse2D.Double*, die Instanzvariable *zufall* ebenfalls mit dem Standardkonstruktor.



Der **Konstruktor** ruft den Konstruktor der Basisklasse mit der Position (0,0) sowie der Größe 20*20 Pixel. Dann setzt er seine Position auf eine zufällige Position im Spiel, die zwischen 0 und (xMax/yMax) begrenzt is und die nicht zu nahe am Rand liegt:

```
this.x = 10+20*zufall.nextInt(xMax/20-3);
this.y = 10+20*zufall.nextInt(yMax/20-3);
```

In ***zufaelligePosition*** setzen wir *x* und *y* auf ein Vielfaches von 20, allerdings so, dass der Apfel nie direkt am Rand des Spielfeldes liegt. Für x geht das so:

```
this.x = 10 + 20*zufall.nextInt(spiel.getWidth()/20 - 2);
```

und analog für y, wobei statt der Breite die Höhe des Spielfeldes abgefragt wird. Die Methode *zufälligePosition* wird gerufen, wenn das Spiel schon läuft, deshalb können wir im Gegensatz zum Konstruktor die Breite und Höhe des Spielfeldes abfragen.

In ***zeichne*** rufen wir *apfel.setFrame(this)* und stellen den Apfel dann als roten Kreis dar (`g.fill(apfel)`)

In ***Aufgabe12.initialisiere*** wird die Variable *apfel* intialisiert, dabei übergeben wir dem Konstruktor für xMax und yMax die Breite/Höhe des Spiels, also (900,800).

In ***zeichne*** stellen Sie den Apfel dar.

Jetzt wird bei jedem neuen Spiel ein roter Punkt an einer zufälligen Position dargestellt.

Die Schlange frisst den Apfel

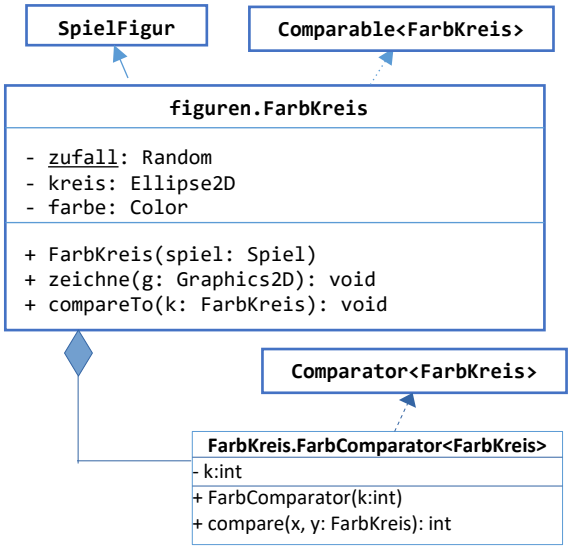
Zunächst implementieren wir in der Klasse ***Snake*** die Instanzmethode ***wachse***. Sie erzeugt ein neues *Rectangle2D.Double*-Objekt mit dem Standardkonstruktor und fügt es mit *addFirst* in die Liste *schlange* ein.

Für die Reaktion zwischen den Schlangen und dem Apfel prüfen wir in ***Aufgabe12.neuerSpielstand*** für jede Schlange, ob sich Apfel und Schlange überschneiden (Methode *intersects*). Wenn ja, wird für den Apfel *zufaelligePosition* und für die jeweilige Schlange *wachsen* gerufen.

Wenn eine Schlange den Apfel frisst, wird sie jetzt jedes mal ein Segment länger und der Apfel springt an eine neue zufällige Position. Natürlich kann man auch mehrere Äpfel ins Spiel bringen.

Aufgabe 13

Comparable, Comparator, innere und anonyme Klassen



Implementieren Sie das Beispiel aus den Vorlesungen 4.5 und 4.6, in dem wir Objekte sortierbar machen.

In diesem Beispiel verwenden wir innere und anonyme Klassen.

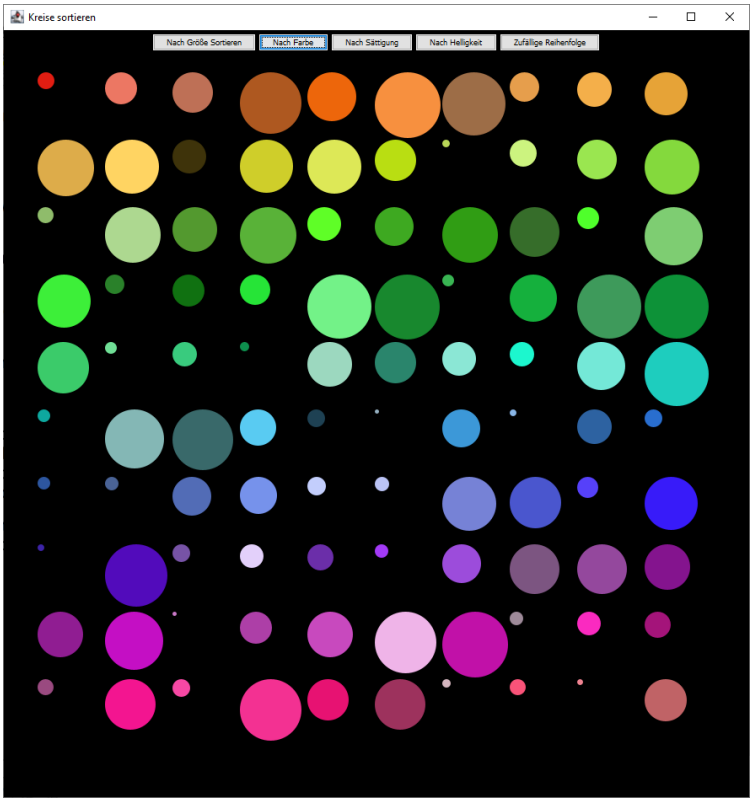
Beginnen Sie mit der Klasse ***FarbKreis***. Ein Objekt dieser Klasse ist eine Spielfigur mit zufälliger Farbe und einer zufälligen Größe, die Position ist irrelevant. Zufall und kreis werden mit dem Standardkonstruktor initialisiert. Die Methode ***compareTo*** vergleicht die Größe der beiden Figuren.

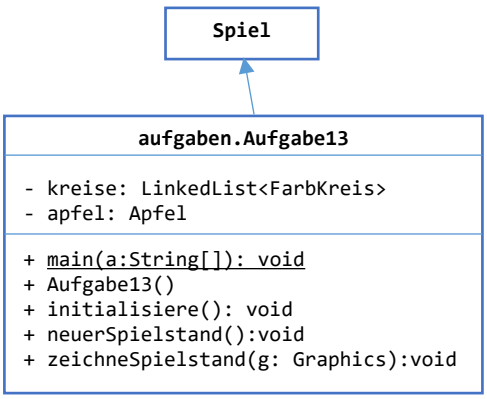
Die Anwendung ***Aufgabe13*** entspricht der Klasse *FarbKreisBeispiel* aus der Vorlesung:

Im **Konstruktor** erzeugen wir ein *JButton* Objekt mit der Aufschrift „Nach Größe Sortieren“- Bei Klick auf diesen Button soll die *LinkedList* ‚kreise' nach ihrer natürlichen Sortierung sortiert werden (*Collections.sort*). Verwenden Sie eine anonyme Klasse für den *ActionListener* dieses Buttons.

Ein zweiter Button soll sie in eine zufällige Reihenfolge anordnen (*Collections.shuffle*).

Für die Sortierung nach Farbeigenschaften schreiben wir die statisch innere Klasse ***FarbKomparator*** aus der Vorlesung, die *Comparator* implementiert und in Aufgabe13 fügen sie weitere Buttons hinzu. Die Kreise sind jetzt nach Farbton, Sättigung, Helligkeit und Größe sortierbar.





Typische Fragen für die mündliche Prüfung

Schreiben Sie folgende Methoden und vermeiden Sie eine mögliche *NullPointerException*:

`static public int anzahlGeraderZahlen(int[] a)`
liefert die Anzahl der Elemente von a, die durch 2 teilbar sind.

`static public double mittelwert(double[] x)`
berechnet den Mittelwert der Zahlen in x.

`static public boolean enthaeltNegativeZahlen(int[] b)`
gibt true zurück, wenn b mindestens eine negative Zahl enthält

`static public void fuelle(int[] a, int w)`
weist allen Elementen von a den Wert w zu.

`static public void begrenze(double[] z, double max)`
ändert alle Zahlen in *z*, die größer als *max* sind, zu max.

`static public int[] geradeZahlen(int[] x)`
gibt einen Array mit den in x enthaltenen geraden Zahlen zurück.
Hinweis: ein Array in Java darf die Länge 0 haben,

`static public void rotiereUm1NachLinks(int[] c)`
rotiert die Werte in c zirkulär um eins nach links, d.h. in c[0] steht anschließend der Wert aus c[1], in c[1] der Wert aus c[2] u.s.w. Die Methode soll in-situ arbeiten.

Schreiben Sie die Methode `selectionSort(int[ ] a)`,
welche den Selection-Sort Algorithmus implementiert.
Gegeben sei die Methode **`ixMinimum(int[] a, int startIndex)`**
welche den Index des kleinsten Elements in `a[startIndex … a.length-1]` zurückgibt.

Schreiben Sie die Methode `bubbleSort(int[ ] a)`,
welche den Bubble-Sort Algorithmus implementiert.
Gegeben sei die Methode **`paarweitseTauschen(int[] a, int maxIndex)`**
welche in `a[0 … maxIndex]` benachbarte Elemente vergleicht und bei Bedarf vertauscht.