Bitte geben Sie ***MeinArrayStack**.java*, ***MeinListenStack.java*** sowie ***Snake.java*** bis zum 22.11.2020 ab!

Aufgabe 14

Stack, verkette Liste

Erzeugen Sie das neue Paket ***listen*** und darin das **Interface *MeinStack<T>*** mit den rechts gezeigten Methoden.

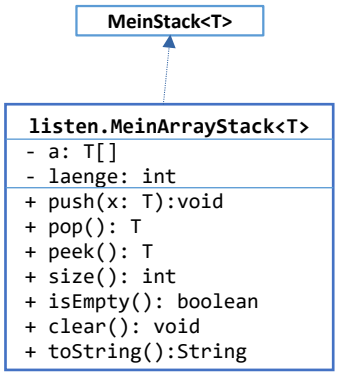
```
«interface» listen.MeinStack<T>
+ push(x: T):void
+ pop(): T
+ peek(): T
+ size(): int
+ isEmpty(): boolean
+ clear(): void
```

push legt ein neues Element auf den Stack,
pop holt das oberste Element vom Stack,
clear leert den Stack, indem *top* zu *null* gesetzt wird,
size liefert die Länge,
isEmpty liefert *true*, wenn der Stack leer ist.

Implementieren Sie dann die von *MeinStack* abgeleitete Klasse ***listen.MeinArrayStack<T>*** aus der Vorlesung, welche den Stack in einem Array speichert.

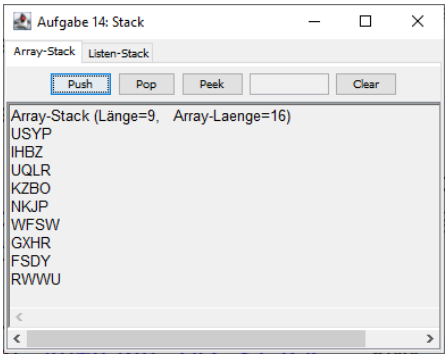
Die Variable **a** muss mit einem Array vom Typ *Object* initialisiert werden, da generische Typen nicht instantiiert werden können. Mit einem **Cast** umgehen wir dabei die strengen Java Regeln zur Typisierung:

```
T[] a = (T[]) new Object[2];
```



Die Methode ***push*** prüft zunächst, ob der Array noch ein Element aufnehmen kann, wenn nicht verdoppelt Sie das Array mit Hilfe von *Arrays.copyOf(…)*. Den Array haben wir absichtlich sehr kurz gemacht, damit dieser Mechanismus schnell getestet wird.

Die Methode ***toString*** gibt den Stack als Zeichenkette zurück, wie im Bild unten gezeigt.




```
public static String zufallString(int laenge) {
  final String alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  final Random zufall = new Random();
  StringBuffer s = new StringBuffer();
  for(int i=0;i<laenge;i++) {
     s.append(alpha.charAt(zufall.nextInt(alpha.length())));
  }
  return s.toString();
}
```

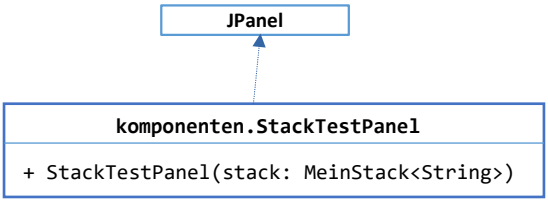
Zum Testen Ihres Stacks erweitern Sie zunächst die Klasse ***ArrayTools*** um oben stehende Methode ***zufallString***, die eine zufällige Zeichenkette der angegebenen Länge erzeugt. Aus Effizienzgründen wird dabei ein *StringBuffer* verwendet.

Dann fügen Sie die rechts gezeigte Klasse ***ObjektTextArea*** in Ihr Paket *komponenten* ein.

Diese Klasse finden Sie auch im online Kurs, wir werden sie in den nächsten Übungen immer wieder brauchen.

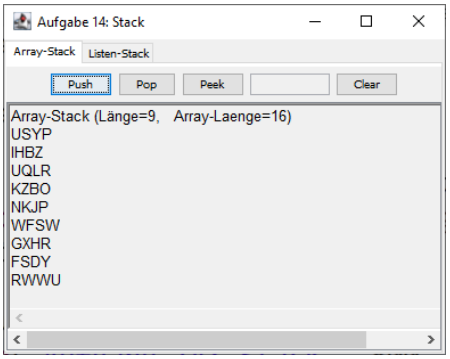
Sie ist eine Text-Area, die ein beliebiges Objekt als String darstellt.

```
public class ObjektTextArea<T> extends TextArea {
	private T objekt;
	public ObjektTextArea(T obj) {
		this.objekt = obj;
		this.setEditable(false);
		this.update();
	}
	public void update() {
		this.setText(objekt.toString());
		this.setCaretPosition(0);
	}
	public T getObjekt() {
		return objekt;
	}
}
```



Schreiben Sie die von *JPanel* abgeleitete Klasse ***StackTestPanel***, die nur aus einem Konstruktor besteht.

Ein *StackTestPanel* Objekt soll wie im Bild gezeigt aussehen. Es hat ein *BorderLayout* und besteht aus einem oberen Panel mit den vier gezeigten *JButton* Objekten und dem *JTextField*. Auf den mittleren Bereich des Panels wird eine *JscrollPane gelegt,* die eine *ObjektTextArea* enthält.



Der **Konstruktor** erzeugt die Buttons sowie das *JTextField* und legt sie auf ein *JPanel*, das auf *NORTH* platziert wird, sowie die *ObjektTextArea* welche auf die Position *CENTER* kommt.

Dann schreiben Sie für jeden Button eine anonyme Listener-Klasse:

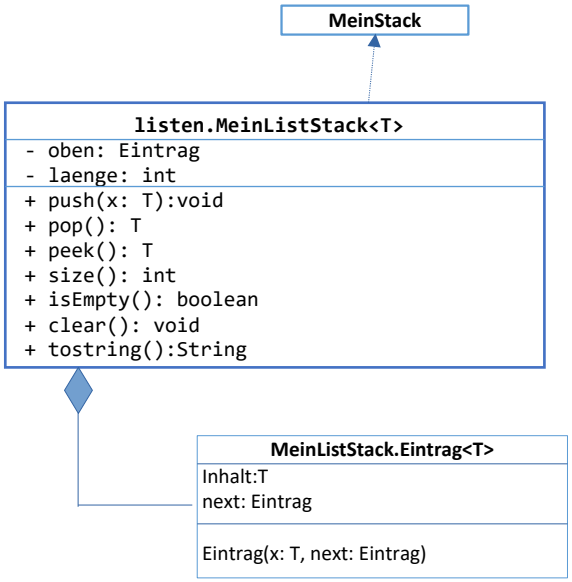
Bei ***Push*** wird eine zufällige Zeichenkette der Länge 3 auf den Stack gelegt, anschließend wird für die *ObjektTextArea* die Methode ***update*** gerufen, damit der Stack neu dargestellt wird.

Pop hold das oberste Element vom Stack und stellt es im *JTextField* dar, anschließend wird für die *ObjektTextArea* die Methode ***update*** gerufen.

Peek hold das oberste Element vom Stack und stellt es im *JTextField* Objekt dar, ohne den Stack zu ändern.

Clear leert den Stack und ruft die Methode ***update***.

Die Klasse ***Aufgabe14*** wird von *StandardAnwendung* abgeleitet und enthält ein *JTabbedPane* Objekt, auf dem ein *StackTestPanel* liegt, dessen Konstruktor ein Objekt vom Typ *MeinArrayStack<String>* bekommt.



Implementieren Sie die weitere Klasse ***listen.MeinListStack<T>*** aus der Vorlesung nach dem UML Diagramm. Die innere Klasse *Eintrag* wird *private* deklariert.

Beim Konstruktor und den Variablen der inneren Hilfsklasse *Eintrag* geben wir keine explizite Sichtbarkeit an, da die Klasse selbst private ist können ohnehin nur Methoden der äußeren Klasse darauf zugreifen.

Für die Methode *toString* benötigen wir eine Hilfsvariable vom Typ *Eintrag*. Um sie effizienter zu machen, können wir auch wieder einen *StringBuffer* verwenden:

```
public String toString() {
  StringBuffer s = new StringBuffer("Listen-Stack ");
  s.append("Länge="+laenge+" (oberes Element zuerst)");
  Eintrag nadel = oben;
  while(nadel!=null) {
       s.append("\n"+nadel.inhalt);
       nadel = nadel.next;
  }
  return s.toString();
}
```

Fügen sie zur *JTabbedPane* in *Aufgabe14* ein weiteres *StackTestPanel* hinzu, dessen Konstruktor ein *MeinListStack<String>* Objekt bekommt.

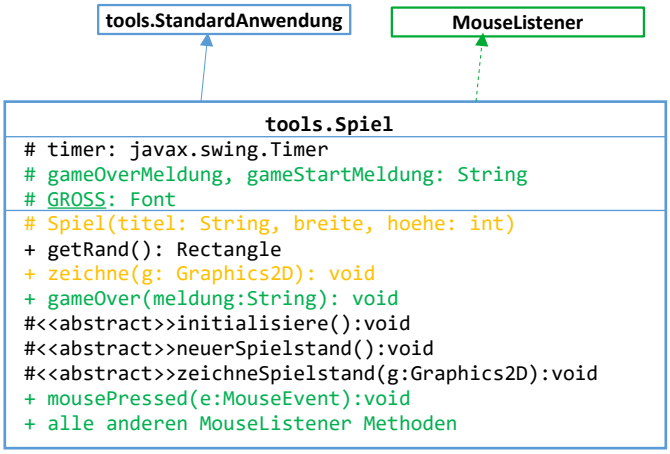
Aufgabe 15

Snake Spiel Fortsetzung

In dieser Aufgabe ergänzen und vervollständigen wir das Snake-Spiel für zwei Spieler.

Zunächst erweitern wir ***Spiel*** um zwei Dinge zu ermöglichen:

1. Vor dem Start eines Spieles kann eine Meldung gezeigt werden, in der z.B. die Regeln erklärt werden. Erst beim Mausklick beginnt dann das eigentliche Spiel.
2. Durch Aufruf der neuen Methode *gameOver* wird das Spiel beendet und eine Meldung auf dem Bildschirm gezeigt.



Zunächst fügen wir der Klasse ***Spiel*** wie im UML Diagramm gezeigt die grün markierten Elemente hinzu und erweitern die gelb markierten Methoden.

Zusätzlich lassen Sie *Spiel* das Interface *MouseListener* implementieren und **fügen alle Methoden von *MouseListener* hinzu**, damit wir diese nicht in allen abgeleiteten Klassen implementieren müssen.

gameOverMeldung und ***gameStartMeldung*** werden mit *null* initialisiert, **GROSS** mit
`Font("Helvetica",Font.BOLD,20)`

Die rechts gezeigte Methode ***gameOver*** speichert ihren Parameter in *gameOverMeldung*, stoppt den Timer und fügt den *MouseListener* hinzu.

Die rechts gezeigte Methode ***mousePressed*** startet das Spiel, wobei unterschieden wird ob es zum ersten Mal oder nach *GameOver* gestartet wird. Somit können wir ein Spiel mehrfach starten, vorausgesetzt die Methode *initialisiere* ist korrekt implementiert.

```
public void gameOver(String meldung) {
  this.gameOverMeldung = meldung+"\nKlick um weiter zu spielen.";
  this.timer.stop();
  this.addMouseListener(this);
}
@Override
public void mousePressed(MouseEvent arg0) {
        if(this.gameOverMeldung!=null) {
                this.gameOverMeldung = null;
                this.initialisiere();
                if(gameStartMeldung==null) timer.start();

        }else {
                this.gameStartMeldung = null;
                timer.start();
                this.removeMouseListener(this);
        }
}
```

Die Methode ***zeichne*** wird durch die rechts gezeigte Methode ersetzt. Sie unterscheidet zwischen drei Zuständen:

1. Es existiert eine *gameStartMeldung*: dann wird diese angezeigt, um zum Beispiel die Spielregeln zu erklären. Der Timer läuft in diesem Zustand nicht. Erst beim Klick mit der Maus wird der Timer gestartet.
2. Der Timer läuft: in diesem Zustand wird einfach nur der Spielstand gezeichnet
3. Der Timer läuft nicht und es existiert eine *gameOverMeldung*: dann wird diese Meldung angezeigt. Dieser Zustand wird durch Aufruf von *gameOver* erreicht.

```
@Override
public final void zeichne(Graphics2D g) {
  g.setFont(GROSS);
  g.setColor(Color.WHITE);
  int y=80, schriftHoehe=g.getFontMetrics().getHeight();
  if(gameStartMeldung!=null) {
   for (String line : gameStartMeldung.split("\n"))
                  g.drawString(line,50,y+=schriftHoehe);
  }else if(timer.isRunning()) zeichneSpielstand(g);
   else if(gameOverMeldung!=null) {
      zeichneSpielstand(g);
      for (String line : gameOverMeldung.split("\n"))
                   g.drawString(line,50,y+=schriftHoehe);
  }
  if(!timer.isRunning()) this.repaint();
}
```

Weitere wichtige Änderung: Im **Konstruktor** wird der Timer **nur dann** gestartet, wenn *gameStartMeldung* den Wert *null* hat, außerdem registriert sich das Spiel als sein eigener *MouseListener*.

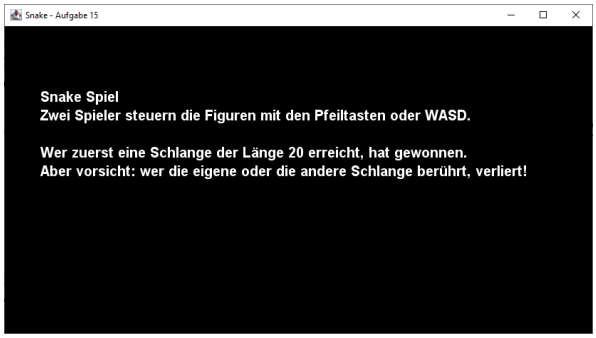
Kopieren Sie jetzt das Paket *aufgabe12* in *aufgabe15 und nennen die Klasse in Aufgabe15 um.*

Um zu Beginn des Spiels eine Erklärung zu zeigen bevor der Anwender das Spiel mit einem Klick startet, fügen Sie folgende Zeilen in die Methode ***Aufgabe15.initialisiere*** ein:

```
this.gameStartMeldung="Snake Spiel\nZwei Spieler steuern die Figuren mit den Pfeiltasten oder WASD."
                    +"\n\nWer zuerst eine Schlange der Länge 20 erreicht, hat gewonnen."
                    + "\nAber vorsicht: wer die eigene oder die andere Schlange berührt, verliert!";
```

In der Klasse ***Snake*** erweitern wir die Methode ***bewege***: fügen Sie zur *if*-Anweisung einen *else*-Zweig hinzu, der durchlaufen wird, wenn die Figur sich nicht mehr bewegt. Rufen Sie darin `this.spiel.gameOver((wasd?"Gelb":"Grün")+" hat den Rand berührt");`

Der **Bedingungsoperator** `wasd?"Gelb":"Grün"` prüft, ob *wasd true* ist. Falls ja, wird der Wert **vor** dem Doppelpunkt eingesetzt, falls nicht, der Wert **nach** dem Doppelpunkt.



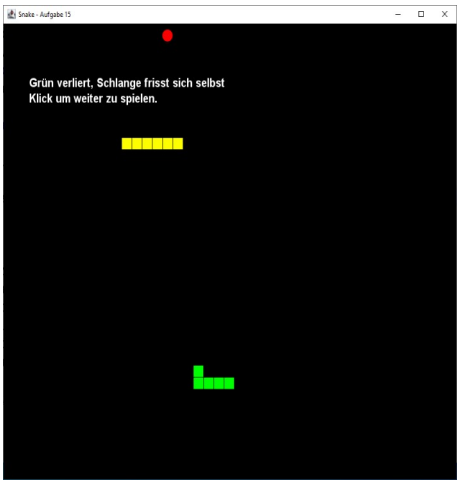
Das Spiel sollte jetzt zunächst die Start-Meldung zeigen und erst bei Klick loslaufen.

Sobald eine der Figuren den Rand berührt, sollte die entsprechende Meldung erscheinen und bei Klick beginnt ein neues Spiel.

Frisst eine Schlange sich selbst?

Wenn der Kopf einer Schlange eines ihrer Segmente berührt, ist das Spiel verloren. Dazu ersetzen Sie in ***Snake*** die leere Methode ***pruefeKollision*** durch die gezeigte Version, welche das Spiel mit einer passenden Meldung beendet, falls die Schlange sich selbst frisst.

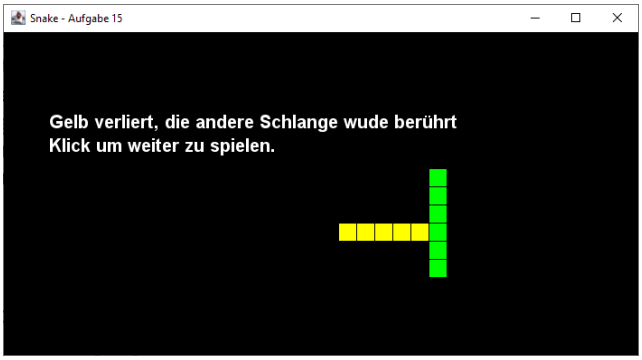
```
public void pruefeKollision(){
 Rectangle2D kopf = schlange.getFirst();
 for(Rectangle2D segment:schlange){
  if(kopf!=segment){
   if(kopf.intersects(segment)){
    String meldung=(wasd?"Gelb":"Grün");
    meldung+=" verliert, Schlange frisst sich selbst";
    spiel.gameOver(meldung);
    }
   }
  }
}
```



Am Ende der Methode ***bewege*** rufen Sie *pruefeKollission* auf. Jetzt sollte das Spiel beendet werden, falls eine der Schlange sich selbst frisst wie rechts gezeigt.

Eine Schlange frisst die andere

Falls der Kopf einer Schlange die andere Schlange berührt, ist das Spiel beendet und der Spieler, dessen Schlange die andere frisst, hat verloren. Dazu implementieren wir in der neuen überladenen Methode ***pruefeKollision(Snake andereSchlange)*** folgende Logik:

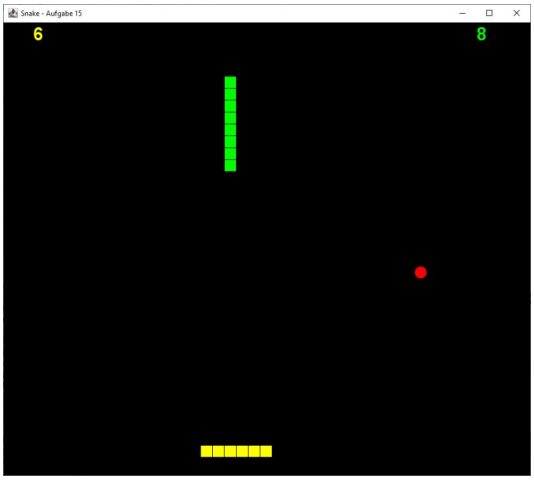


- Der Kopf der eigenen Schlange wird in einer lokalen Variable vom Typ Rectangle2D gespeichert (*schlange.getFirst*).
- In einer *for*-Schleife werden alle Segmente der anderen Schlange durchlaufen.
- Wenn eines der Segmente den eigenen Kopf berührt (*intersects*), wird *spiel.gameOver* gerufen und der Text „XXX verliert: die andere Schlange wurde berührt“ ausgegeben.

Am Ende von ***Aufgabe15.neuerSpielstand*** wird für jede Schlange die Methode *pruefeKollision* (mit der jeweils anderen Schlange) gerufen.

Ziel festlegen, Spielstand anzeigen

Um die Usability zu verbessern, zeigen wir zunächst am oberen Spielfeldrand die Länge der beiden Schlangen an. Dazu fügen Sie am Ende der Methode *Snake.zeichne* folgende Logik ein:



Berechnen Sie die horizontale Textposition: für die grüne Schlange 50 Pixel, für die gelbe Schlange 90 Pixel vom rechten Rand des Spielfelds (*spiel.getWidth()-90*) . Rufen Sie *g.setFont(font)* und schreiben die Länge der Schlange (Metheode *size*) mit der Methode *g.drawString* an die berechnete x-Position, die y-Position setzen Sie auf 35. Vergessen Sie nicht, die Farbe zu setzen, sonst wird der Text in Schwarz auf Schwarz dargestellt!

Damit das Spiel ein **definiertes Ende** hat, erweitern Sie die Methode ***wachse*** so, dass das Spiel beendet wird, wenn die Länge der Schlange größer oder gleich einer bestimmten maximalen Länge (z.B. 20) ist. Rufen Sie in diesem Fall *spiel.gameOver* und geben Sie als Text aus „XXX hat gewonnen: Länge=20“.

Sound

Bauen Sie ein paar Sounds direkt in die Klasse *Aufgabe15* ein, z.B. wenn eine Schlange den Apfel frisst oder wenn das Spiel zu Ende ist. Die Klang-Objekte sollten Klassenvariablen von *Aufgabe15* sein, um die Performance zu verbessern.

Sie können Sounds selbst bauen, sie finden aber auch Sounds im Netz, z.B.:
freesound.org, www.partnersinrhyme.com/pir/PIRsfx.shtml, www.pacdv.com/sounds, soundimage.org
Für einen Game-Over Sound überschreiben Sie in Aufgabe15 die Methode *gameOver*, rufen *super.gameOver* und spielen dann den Sound ab.