Bitte lösen Sie Aufgabe 26 bis zum 21.06.2020 **und geben Aufgabe 26 ab!**

Ziele des heutigen Übungstermins

- Sie erweitern die Basisklasse zur Vereinfachung der Entwicklung von Spielen.
- Sie üben die Verzweigung mit der if-Anweisung .
- Sie üben die *for* Schleife über alle Elemente eines Linked-List Objekts.
- Sie vertiefen Ihre Kenntnisse im Umgang mit den Klassen Rectangle und Graphics2D.
- Sie verwenden *final* und *static*.

Aufgabe 26

viele Spielfiguren, Reaktion mit Spielfeldrand

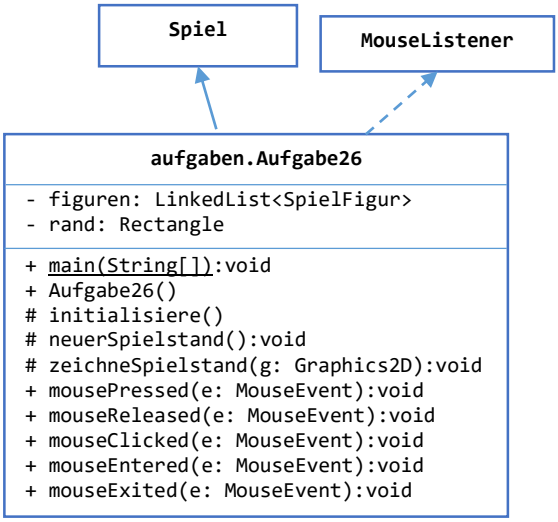
Wir programmieren ein Spiel, bei dem jeder Klick auf das Spielfeld eine neue Spielfigur erzeugt.

Der **Konstruktor** ruft den Basisklassenkonstruktor und registriert sich als sein eigener MouseListener.

In ***initialisiere*** werden *figuren* und *rand* mit dem Standardkonstruktor initialisiert.

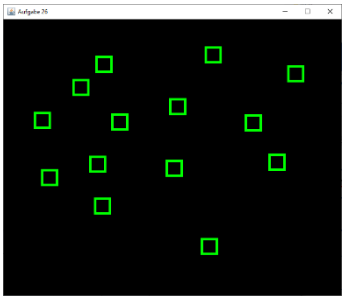
Die Methoden ***neuerSpielstand*** und ***zeichneSpielstand*** führen eine *for*-Schleife über alle Spielfiguren durch und bewegen bzw. zeichnen jede Figur:

```
for(SpielFigur f:figuren) f.bewege();
```



In ***mousePressed*** wird eine neue Spielfigur an der Cursorposition (e.getX()/getY()) erzeugt und mit der Methode *add* zu *figuren* hinzugefügt: figuren.add(new SpielFigur(…))

Jetzt erscheint bei jedem Klick eine neue Spielfigur an der Cursorposition.



Spielfeld begrenzen

Die Begrenzung des Spielfeldes muss nicht unbedingt mit dem Rand des Panels zusammenfallen, deshalb erweitern wir die Klasse ***Spiel*** um die folgende Methode:

```
/**
 * Gibt den Spielfeldrand zurück, wird von abgeleiteten Klassen überschrieben,
 * falls die Begrenzung des Spielfeldes nicht mit dem Rand des Panels übereinstimmt.
 *
 * @return Rand des Spielfeldes
 */
public Rectangle getRand() {
        return this.getBounds();
}
```

In ***Aufgabe26*** überschreiben wir ***getRand*** so, dass ein Rechteck zurückgegeben wird, welches auf allen Seiten um 80 Pixel innerhalb des Spielfeldes liegt. Aus Effizienzgründen ist es wichtig, dass dabei nicht ein neues Objekt erzeugt wird, sondern das bestehende Objekt *rand* verwendet wird:

```
@Override
public Rectangle getRand() {
   int delta=80;
   rand.setBounds(getX()+delta, getY()+delta, getWidth()-2*delta, getHeight()-2*delta);
   return rand;
}
```

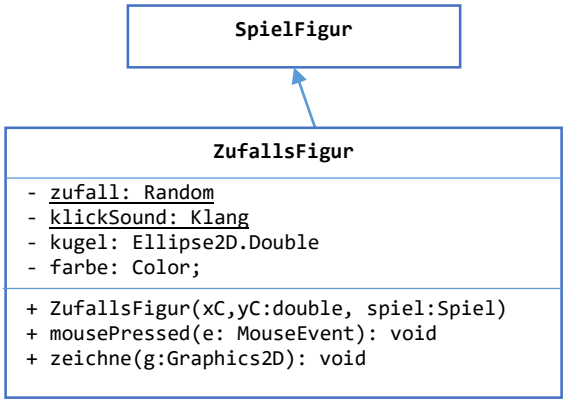
Am Anfang von ***zeichneSpielstand*** färben wir das Spielfeld ein:

```
g.setColor(Color.WHITE);
g.fill(this.getRand());
```

Jetzt erzeugen Sie in ***mousePressed*** nur dann eine neue Figur, wenn der Cursor innerhalb des Randes ist. **Wichtig**: verwenden Sie *getRand()* und nicht direkt die Variable *rand*:

```
if(this.getRad().contains(…
```

Farbige Spielfiguren



Als nächstes sollen die neu erzeugten Spielfiguren als farbige Kreise dargestellt werden. Dazu schreiben wir die Klasse ***ZufallsFigur***. Die **Klassenvariablen *zufall*** und ***klickSound*** werden gleich bei der Definition initialisiert. Für *klickSound* können Sie z.B. die Datei *blip.mp3* aus dem Online-Kurs verwenden. Auch die Instanzvariable ***kugel*** wird mit dem Standardkonstruktor initialisiert.

Der **Konstruktor** legt seine Größe auf 30*30 Pixel fest, spielt den Sound ab und speichert in *farbe* eine zufällige Farbe (siehe Aufgabe 15), wobei der Maximalwert der Farbkanäle 200 statt 255 sein sollte, damit sich die Figuren gut vor dem weißen Hintergrund abheben.

In ***zeichne*** rufen wir `kugel.setFrame(this)` damit *kugel* an der aktuellen Position der Spielfigur liegt, anschließend füllen wir *kugel* mit *farbe* aus.

Bewegte Spielfiguren

Jetzt sollen sich die neu erzeugten Spielfiguren in einer zufälligen Richtung bewegen.

Im **Konstruktor von *ZufallsFigur*** berechnen wir einen zufälligen Geschwindigkeitsvektor, dessen Komponenten *x* und *y* maximal den Betrag 5 haben, dazu rufen wir die Methode *nextDouble*, welche eine zufällige Zahl zwischen 0 und 1 liefert. Für die x-Komponente sieht die Anweisung wie folgt aus:

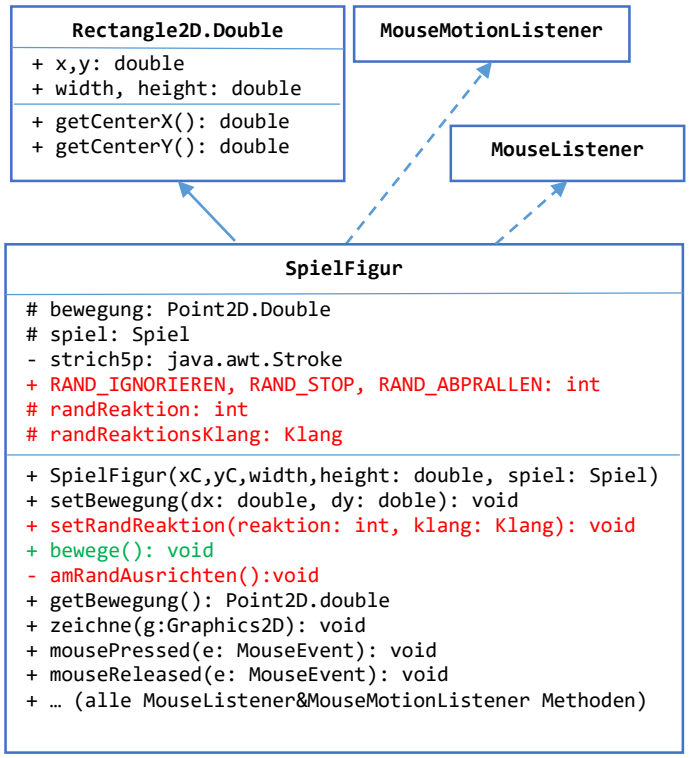
```
int max_speed=5;
bewegung.x = zufall.nextDouble()*2*max_speed - max_speed;
```

Unsere Spielfiguren fliegen jetzt in eine zufällige Richtung davon und verschwinden aus dem Spielfeld. Im nächsten Schritt werden wir dafür sorgen, dass die Figuren am Spielfeldrand liegen bleiben.

Interaktion mit dem Spielfeldrand

Wir erweitern ***SpielFigur*** um die rot markierten Komponenten und erweitern die Methode *bewege* so, dass die Figuren selbständig mit dem Rand des Spiels interagieren können. Zunächst definieren wir drei *int*-Konstanten (*final* bedeutet, dass der Wert nicht mehr änderbar ist):

```
public static final int
    RAND_IGNORIEREN=0, RAND_STOP=1,
    RAND_ABPRALLEN=2;
```



Sie stehen für drei mögliche Aktionen beim Berühren des Spielfeldes:

- Keine Reaktion,
- die Figur stoppt,
- die Figur prallt am Rand ab.

Die Variable ***randReaktion*** wird mit RAND_IGNORIEREN initialisiert.

Die Methode ***setRandReaktion*** speichert ihre Parameter in den Instanzvariablen.

Am Ende der Methode ***bewege*** (also nachdem sich die Figur bewegt hat) prüfen wir zunächst, ob die Figur mit dem Rand reagieren soll:

```
  if(randReaktion != RAND_IGNORIEREN){
```

Jetzt stellen wir fest, ob die Spielfigur den Rand überhaupt berührt, das ist der Fall, wenn die Figur (*this*) nicht mehr komplett im Rand des Spiels enthalten ist:

```
Rectangle rand = spiel.getRand();
if(!rand.contains(this)){
```

Das Ausrufungszeichen ist der logische Operator NOT, er wandelt *true* in *false* und umgekehrt.

Jetzt spielen Sie den Klang für die Reaktion mit dem Rand ab, falls er definiert ist:

```
if(randReaktionsKlang!=null) randReaktionsKlang.play();
```

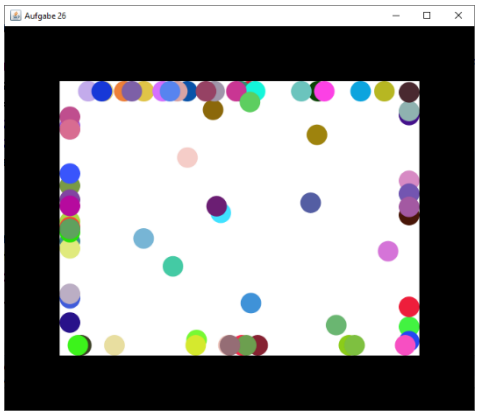
Dann kommt die eigentliche Logik zur Reaktion mit dem Rand:

```
if(randReaktion==RAND_STOP){
         this.bewegung.x = this.bewegung.y = 0;
}
if(randReaktion==RAND_ABPRALLEN){
        //hier programmieren wir später das Abprallen
}
this.amRandAusrichten();
```

Die Methode ***amRandAusrichten*** sorgt dafür, dass Spielfiguren, die sich über den Rand hinausbewegt haben, wieder genau am Rand positioniert werden:

```
/**
 * Positioniert Spielfiguren, die ueber den Rand hinausragen, genau an den Rand.
 */
 private void amRandAusrichten(){
        Rectangle spielRand = spiel.getRand();
        double deltaX = 0.0, deltaY = 0.0;
        if(spielRand.getMinX() > this.getMinX()) deltaX = spielRand.getMinX() - this.getMinX();
        if(spielRand.getMaxX() < this.getMaxX()) deltaX = spielRand.getMaxX() - this.getMaxX();
        if(spielRand.getMinY() > this.getMinY()) deltaY = spielRand.getMinY() - this.getMinY();
        if(spielRand.getMaxY() < this.getMaxY()) deltaY = spielRand.getMaxY() - this.getMaxY();
        this.x = this.x + deltaX;
        this.y = this.y + deltaY;
}
```

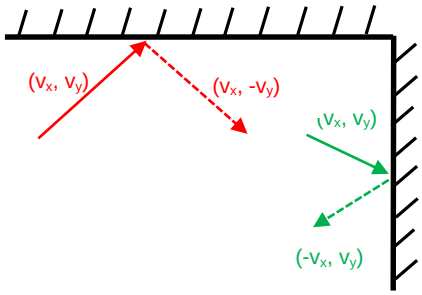
Ohne diese Methode kann es vorkommen, dass Figuren am Rand hängenbleiben, z.B. wenn man die Größe des Spielfeldes mit der Maus verändert.



Im **Konstruktor von *ZufallsFigur*** rufen Sie `this.setRandReaktion(RAND_STOP, null)`.

Die Kugeln sollten jetzt am Rand liegen bleiben.

Abprallen am Rand



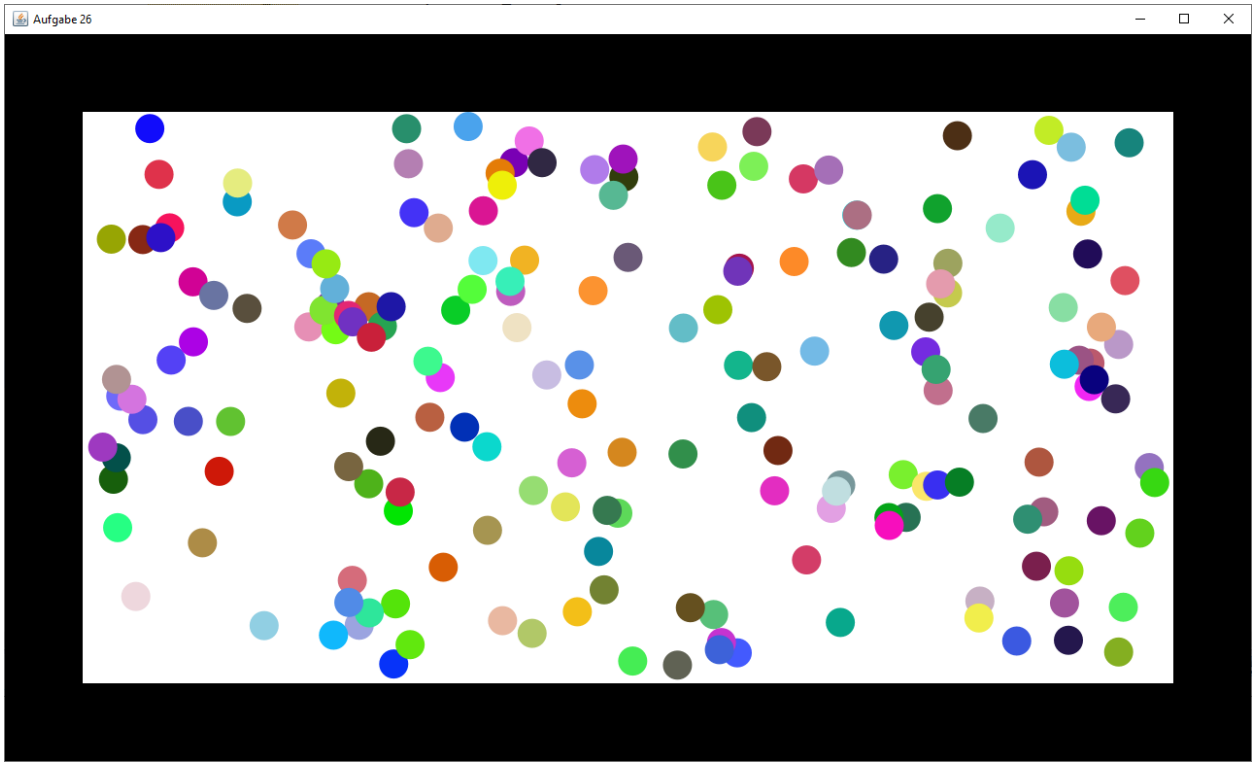
Die Logik für das Abprallen ist einfacher als Sie vermuten. Da die Ränder des Spielfeldes genau senkrecht bzw. waagerecht verlaufen, müssen wir jeweils nur das Vorzeichen einer Geschwindigkeitskomponente wechseln. Im Bild ist dies dargestellt: wenn die Kugel auf eine waagerechte Wand prallt, muss das Vorzeichen der Y-Komponente wechseln, bei einer senkrechten Wand das der X-Komponente.

In der Methode ***bewege*** der Klasse ***SpielFigur*** fügen Sie innerhalb der Abfrage `if(randReaktion==RAND_ABPRALLEN)` folgendesweisungen ein:

```
if(this.x<rand.x || this.x+this.width>rand.width+rand.x) bewegung.x = -bewegung.x;
if(this.y<rand.y || this.y+this.height>rand.height+rand.y) bewegung.y = -bewegung.y;
```

(der Operator || ist ein logisches ODER, für das logische UND gibt es den Operator &&).

Setzen Sie im Konstruktor von *ZufallsFigur* die Randreaktion auf RAND_ABPRALLEN um die Figuren am Rand abprallen zu lassen. Sie verhalten sich jetzt ein wenig so wie die Moleküle in einem Idealen Gas und nehmen immer den ganzen Raum ein, wenn man die Größe des Fensters verändert.



Die Klasse *SpielFigur* enthält jetzt Funktionalität, die wir bei verschiedenen Spielen wiederverwenden können. Wir werden beispielsweise später ein Billardspiel programmieren, dessen Kugeln vom Rand abprallen müssen – dieses Verhalten beherrschen unsere Spielfiguren jetzt schon.