Bitte lösen Sie Aufgaben 31-32 bis zum 12.07.2020 **und geben Aufgabe 31 ab!**

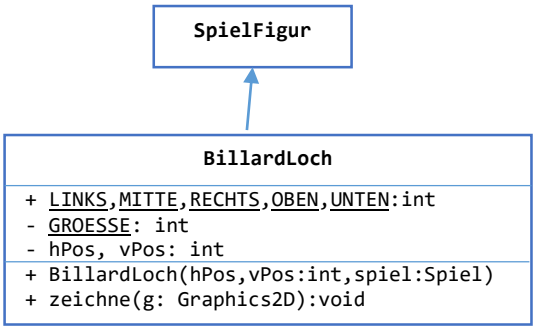
Aufgabe 31

Billard-Spiel

Im nächsten Schritt bekommt unser Billard-Tisch Löcher, in denen die Kugeln verschwinden. Erstellen Sie zunächst das Paket ***aufgabe31*** als Kopie von *aufgabe30*.

Dann sorgen Sie für eine realistischere Aufstellung der Kugeln wie unten gezeigt: die weiße Kugel ist links auf dem Feld, rechts sind 15 farbige Kugeln. Den Billard-Tisch habe ich auf 1400*900 Pixel vergrößert, das ist ein realistischeres Seitenverhältnis.

Löcher am Rand des Spielfeldes



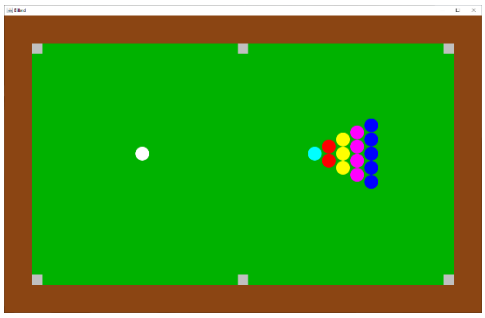
Für die Löcher schreiben wir die Klasse ***BillardLoch***. GROESSE setzen wir auf 30, die anderen Klassenvariablen auf beliebige, aber **unterschiedliche** Werte.

Der **Konstruktor** ruft den Basisklassenkonstruktor mit (0,0,GROESSE,GROESSE, spiel) und speichert die beiden anderen Parameter in den Instanzvariablen.

In ***zeichne*** berechnen wir die Position (x,y) des Lochs, dabei prüfen wir *hPos* gegen *LINKS*, *RECHTS* und *MITTE*, sowie *vPos* gegen *OBEN* und *UNTEN*:

$$x = \begin{cases} R_x^{min} & \text{für } hPos = LINKS \\ R_x^{min} + \dfrac{R_x^{max} - R_x^{min} - GROESSE}{2} & \text{für } hPos = MITTE \\ R_x^{max} - GROESSE & \text{für } hPos = RECHTS \end{cases}$$

$$y = \begin{cases} R_y^{min} & \text{für } vPos = OBEN \\ R_y^{max} - GROESSE & \text{für } vPos = UNTEN \end{cases}$$



Dabei ist R_x^{min} der linke Rand des Spielfeldes, den wir mit `spiel.getRand().getMinX()` bekommen - analog gibt es die Funktionen `getMinY()` und `getMaxX()` sowie `getMAxY()`.
Am Ende von ***zeichne*** wird die Spielfigur in der Farbe LIGHT_GRAY dargestellt.

BillardSpiel bekommt die Instanzvariable ***loecher*** vom Typ *LinkedList<BillardLoch>*. In ***initialisiere*** wird die Liste erzeugt und es werden 6 *BillardLoch*- Objekte mit den horizontalen Position *BillardLoch.LINKS*, *BillardLoch.MITTE* sowie *BillardLoch.RECHTS* und den vertikalen Positionen *BillardLoch.OBEN*, *BillardLoch.UNTEN* hinzugefügt.

In ***zeichneSpielstand*** stellen Sie die Löcher dar.

Das Billard-Spiel sollte jetzt wie dargestellt aussehen. Um eine schönere grafische Darstellung kümmern wir uns später.

Die Kugeln verschwinden in den Löchern

Fügen Sie bitte in der Klasse ***SpielFigur*** noch die Methode ***+getSpiel():spiel*** ein welche *spiel* zurückgibt sowie in der Klasse ***BillardKugel*** die Methode ***+getFarbe():Color***, welche die Farbe der Kugel liefert.

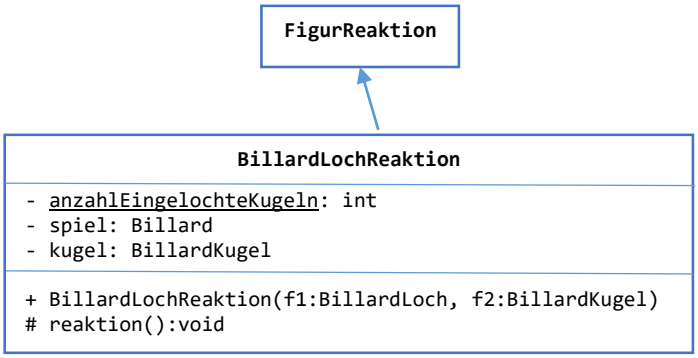
Für die Interaktion der Kugeln mit den Löchern schreiben wir die Klasse ***BillardLochReaktion***.

Der **Konstruktor** ruft den Basisklassenkonstruktor, die Sound-Datei *billard_kugel_loch.mp3* kann man für den Klang verwenden. Dann speichert er die *f2* in der Instanzvariable ***kugel*** und das Spiel (f2.getSpiel()) in ***spiel***.

Die Methode ***reaktion*** setzt die Bewegung der Kugel auf 0.

Handelt es sich um die weiße Kugel (*.equals(…)* verwenden!) , wird die Position auf (200,400) gesetzt, d.h. wenn die weiße Kugel in ein Loch gerät, wird sie wieder auf das Spielfeld gesetzt:

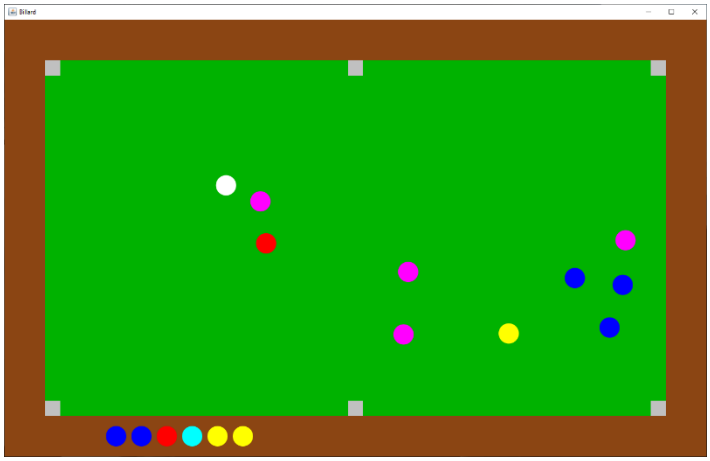
kugel.x=200; kugel.y=400;



Alle anderen Kugeln bekommen zunächst die Randreaktion RAND_IGNORIEREN und werden dann auf den unteren Rand des Billardtisches gesetzt, d.h. z.B. x wird zu *this.x = 200+50*anzahlEingelochteKugeln* und this.y = *spiel.getHeight()-60*. Abschließend zählen Sie *anzahlEingelochteKugeln* hoch.

In der Klasse Billard ändern Sie bitte zunächst den Typ der Liste ***reaktionen*** von *LinkedList<ElastischerStoss>* auf *LinkedList<FigurRealtion>*, dann können wir auch die Reaktion zwischen Kugel und Loch mit in diese Liste aufnehmen (Zuweisungskompatibilität).

Dann fügen Sie in ***initialisiere*** zu dieser Liste für jedes Loch und jede Kugel eine *BillardLochReaktion* hinzu, dazu programmieren Sie zwei ineinander geschachtelte *for*-Schleifen über die Löcher und die Kugeln.

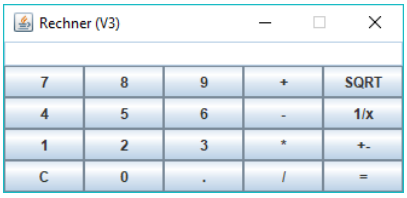


Jetzt sollten alle bis auf die weiße Kugel in den Löchern verschwinden und am unteren Rand des Spielfeldes aufgereiht werden wie im Bild gezeigt.

Aufgabe 32

Taschenrechner voll funktionsfähig

Bitte erzeugen Sie zunächst das Paket *aufgabe32* als Kopie von *aufgabe24*.



Es fehlen noch die Tasten für die mathematischen Operationen +, -, /, * sowie die = Taste. Bevor wir diese programmieren, führen wir folgende Vorüberlegung zur Arbeitsweise des Taschenrechners durch:

Wenn Sie nacheinander die Tastenfolge 4, +, 5, * eingeben, wird bei Drücken von + nicht sofort die Addition ausgeführt, sondern sie wird als ausstehende Operation gespeichert und der Taschenrechner sorgt dafür, dass bei der nächsten Eingabe einer Ziffer zunächst die Anzeige gelöscht wird. Erst beim Drücken von * wird die Addition von 4 und 5 durchgeführt.

Sobald eine Taste für eine Operation gedrückt wird, passiert also folgendes:

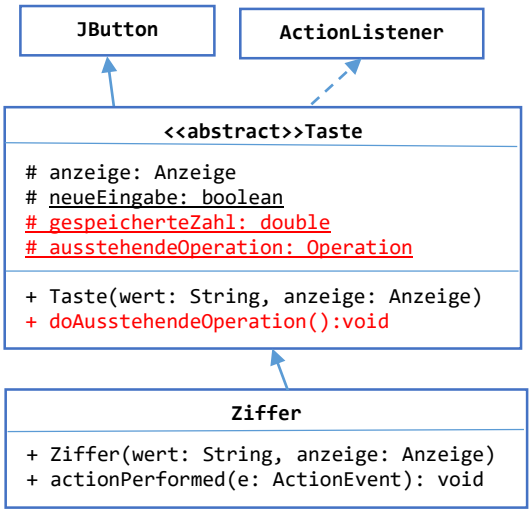
- Eine ggf. ausstehende Operation wird ausgeführt.
- Der Taschenrechner merkt sich die gerade im Display angezeigte Zahl sowie die ausstehende Operation für später.
- Der Taschenrechner geht in den Modus ‚neue Eingabe', d.h. beim Drücken einer Zifferntaste wird die Anzeige gelöscht.

Um dieses Verhalten zu programmieren, schreiben wir zunächst das öffentliche Interface *Operation*, das folgende Methode fordert: **`+ doOperation(x,y: double): double.`**

Ergänzen Sie die Klasse ***Taste*** um die beiden **Klassenvariablen** *gespeicherteZahl* sowie *#ausstehendeOperation*.

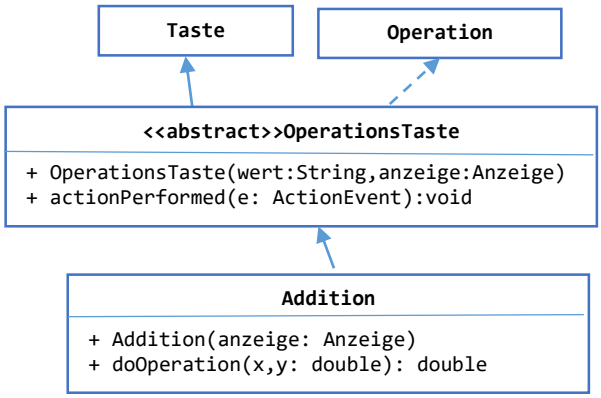
Die neue Instanzmethode ***doAusstehendeOperation*** setzt zunächst *neueEingabe* auf *true*.
Falls *ausstehendeOperation* nicht *null* ist, ruft sie für *ausstehendeOperation* die Methode *doOperation* auf und übergibt als ersten Parameter *gespeicherteZahl* sowie als zweiten Parameter den aktuellen Wert aus der Anzeige. Das Ergebnis schreibt sie wieder auf die Anzeige (hängen Sie die Zahl an einen leeren String). Am Ende wird *ausstehendeOperation* zu *null* gesetzt.

Zu Beginn der Methode ***actionPerformed*** der Klassen ***Wurzel***, ***PlusMinus*** und ***Kehrwert*** rufen Sie *doAusstehendeOperation*. ***Clear.actionPerformed*** setzt *ausstehendeOperation* auf *null*.



Dann schreiben Sie die von *Taste* abgeleitete **abstrakte** Klasse ***OperationsTaste***, welche das Interface *Operation* implementiert und als Basisklasse für die Tasten +, -, /, * sowie = dient.

Der **Konstruktor** gibt seine Parameter an den Konstruktor der Basisklasse weiter.
In der Methode ***actionPerformed*** rufen wir zunächst *doAusstehendeOperation* auf, dann setzen wir *ausstehendeOperation* auf *this* und *gespeicherteZahl* auf den Wert aus der Anzeige. Die Methode ***doOperation*** implementieren wir nicht, das machen die abgeleiteten Klassen.



Die von *OperationsTaste* abgeleiteten neuen Klassen ***Addition***, ***Subtraktion***, ***Multiplikation***, ***Division*** und ***IstGleich*** bestehen jeweils nur aus dem Konstruktor mit einem Parameter vom Typ *Anzeige* sowie der Methode *doOperation*.
Der **Konstruktor** ruft nur den Basisklassenkonstruktor.
Die Methode ***doOperation*** führt die jeweilige mathematische Operation durch, bei *IstGleich* gibt sie einfach *x* zurück.

Wenn Sie die JButton Objekte in Taschenrechner durch Objekte der neuen Klassen ersetzen, sollte der Taschenrechner funktionieren.

Und zum Schluss sorgen Sie für mehr Sicherheit:

- Falls beim Drücken der Taste *sqrt* eine negative Zahl oder beim Kehrwert die 0 im Anzeigefeld steht, berechnen Sie kein Ergebnis, sondern geben den Text ERROR aus.
- In der Methode *getZahl* der Klasse *Anzeige* prüfen Sie, ob die Anzeige leer ist oder "ERROR" enthält und geben in diesen Fällen den Wert 0.0 zurück.

Schauen Sie sich bitte nochmals die Struktur der Anwendung genau an und versuchen alle Methoden zu verstehen!