Bitte geben Sie ***KartenAmCursor.java, KartenAusgabe.java***, ***KartenZielStapel.java*** sowie ***Aufgabe12.java*** und ***Baum.java*** bis zum 13.12.2020 ab!

Aufgabe 20

Kartenspiel: Drag and Drop

Erstellen Sie eine Kopie des Pakets *aufgabe19* als ***aufgabe20*** und nennen *Aufgabe19* in ***Aufgabe20*** um.

Drag und Drop vorbereiten

In der nächsten Variante unseres Solitär-Spiels wird bei Anklicken der Kartenausgabe die oberste Karte aufgenommen und an den Cursor geheftet. Beim Loslassen der Maustaste soll die Karte entweder auf einen der Zielstapel gelegt werden oder wieder auf der Kartenausgabe landen.

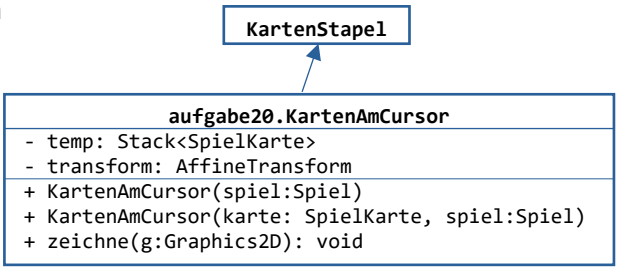
Übernehmen Sie die beiden Interfaces ***SpielFigurGeber*** und ***SpielFigurNehmer*** in das Paket *figuren.*

```
public interface SpielFigurGeber<T extends SpielFigur> extends Shape{
   public T gibFigur();
   public void nimmFigurZurueck(T  figur);
}
```

```
public interface SpielFigurNehmer<T extends SpielFigur> extends Shape{
   public boolean nimmFigur(T figzu);
}
```

Diese beiden Schnittstellen sind die Grundlage für die Implementierung des Drag- and Drop-Verhaltens. Ein *SpielFigurGeber* gibt eine Spielfigur an einen *SpielFigurNehmer* und er nimmt sie eventuell wieder zurück, falls der Nehmer sie nicht akzeptiert. Beide erweitern das Interface *java.awt.Shape*, damit werden wir feststellen können, ob sich der Cursor über dem Geber bzw. Nehmer befindet.

Ein *SpielFigurNehmer*, dem eine Spielfigur angeboten wird, kann eine Figur also entweder annehmen oder ablehnen, im letzteren Fall gibt *nimmFigur false* zurück. In einem Schachspiel wäre z.B. ein Feld auf dem Brett sowohl ein *SpielFigurNehmer* als auch ein *SpielFigurGeber*. Die Interfaces sind so typisiert, dass jede Art von Spielfigur aufgenommen und abgelegt werden kann (*T extends SpielFigur*). Verschiedene Stapel im Solitär-Spiel werden diese Schnittstellen implementieren.



Die neue Klasse ***KartenAmCursor***, enthält die aufgenommenen Karten, die an den Cursor geheftet werden, während die Maus gezogen wird.

Der **Konstruktor mit einem Parameter** ruft *super(0, 0, spiel)*, setzt *deltaX=4* und *deltaY=30* und *zeigeRand* auf *false*. AM Schluss setzt er *transform* auf eine Translation um (-40, -100) Pixel.

Der überladene **Konstruktor mit zwei Parametern** ruft den anderen Konstruktor mit *this(spiel)* und fügt *karte* zu *this.stapel* hinzu.

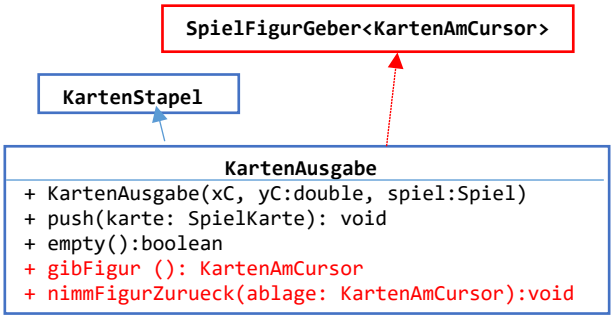
Die rechts gezeigte Methode ***zeichne*** ist etwas aufwändiger. Sie stellt in einer Schleife alle Karten im Stapel dar. Dabei werden die Karten an die aktuelle Mausposition geheftet. Damit die Karten in der richtigen Reihenfolge gezeigt werden, speichern wir sie temporär in einem Stack.

```
@Override
public void zeichne(Graphics2D g) {
   AffineTransform merk = g.getTransform();
   g.setTransform(transform);
   double xt = this.x, yt = this.y;
   while(!stapel.empty()) {
        temp.push(stapel.pop());
        temp.peek().x = xt+=deltaX;
        temp.peek().y = yt+=deltaY;
        temp.peek().zeichne(g);
   }
   while(!temp.isEmpty()) stapel.push(temp.pop());
   g.setTransform(merk);
}
```

Im nächsten Schritt implementiert ***KartenAusgabe*** das Interface *SpielFigurGeber<KartenAmCursor>*.

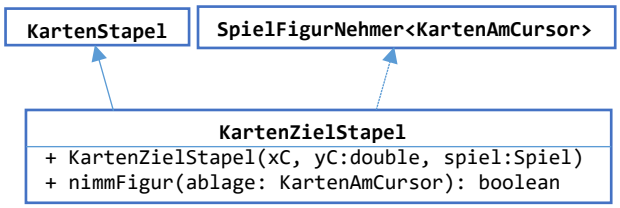
gibFigur prüft mit *empty()*, ob der eigene Stapel leer ist - wenn ja, wird *null* zurückgegeben. Ansonsten wird ein neues Objekt vom Typ *KartenAmCursor* zurückgegeben, auf das die oberste Karte vom eigenen Stapel (*this.pop*) gelegt wird.

nimmFigurZurueck holt die oberste Karte aus *ablage* und legt sie auf den eigenen Stapel zurück (this.push).



Die Zielstapel

Für die vier Stapel rechts im Solitär-Spiel schreiben wir die von *KartenStapel* abgeleitete Klasse ***KartenZielStapel***, welche *SpielFigurNehmer<KartenAmCursor>* implementiert.

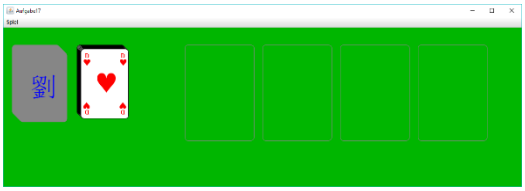


Der **Konstruktor** ruft *super* und setzt *deltaX* auf 3 und *deltaY* auf 8.

Die Methode ***nimmFigur*** holt die oberste Karte aus *ablage*, fügt sie zu sich hinzu (*this.push)* und gibt *true* zurück, d.h. der Stapel nimmt jede Karte an. Später werden wir in dieser Methode die Regel implementieren, dass die Karten alle von der gleichen Farbe sein müssen und nur in in aufsteigender Reihenfolge abgelegt werden dürfen.

Fügen Sie in ***Aufgabe20*** eine Instanzvariable vom Typ `KartenZielStapel[]` hinzu.

Die Methode ***initialisiere*** initialisiert diesen Array mit der Länge 4 und erzeugt dann die vier Zielstapel in einer for-Schleife. Die x-Position der Zielstapel berechnen Sie als `500+180*i`.



In ***zeichne*** stellen Sie sie die vier Zielstapel dar. Der Umriss der vier Stapel sollte jetzt wie rechts gezeigt sichtbar sein.

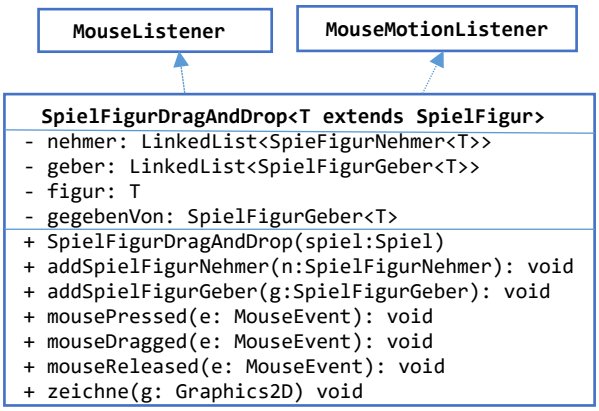
Drag und Drop implementieren

Um das Drag- und Drop-Verhalten zu implementieren schreiben wir im Paket *figuren* die Klasse ***SpielFigurDragAndDrop*** nach dem untenstehenden UML Diagramm.

Bei einem Drag-and-Drop Vorgang sind drei Objekte beteiligt: der Stapel, von dem die Spielkarte genommen wird als **Geber**, der Stapel, auf dem sie abgelegt wird als **Nehmer** und das Objekt vom Typ *SpielFigurDragAndDrop*, welches den Vorgang wie folgt steuert:

- Beim Drücken der Maus-Taste stellt es fest, ob sich unter dem Cursor ein registriertes *SpielFigurGeber* Objekt befindet. Wenn ja, nimmt es von ihm ein Objekt auf.
- Beim Ziehen der Maus (*mouseDragged*) wird die Position der aufgenommenen Figur auf die aktuelle Mausposition gesetzt, d.h. die Figur folgt dem Cursor.
- Beim Loslassen der Maus-Taste wird geprüft, ob sich unter dem Cursor ein *SpielFigurNehmer* Objekt befindet. Wenn ja, wird ihm die Figur angeboten. Falls nein, oder wenn der *SpielFigurNehmer* die Karte ablehnt, wird sie dem *SpielFigurGeber*, von dem sie aufgenommen wurde, zurückgegeben.

Die Instanzvariablen ***nehmer*** und ***geber*** werden mit dem Standardkonstruktor initialisiert, ***figur*** und ***gegebenVon*** mit *null*.



Im **Konstruktor** registriert sich das Objekt als *MouseListener* und *MouseMotionListener* des Spiels.

addSpieFigurNehmer fügt *n* zu *nehmer* hinzu.

addSpielFigurGeber fügt *g* zu *geber* hinzu

In ***mousePressed*** wird eine Schleife über *geber* durchgeführt und geprüft, ob sich der Cursor über dem jeweiligen *SpielFigurGeber* Objekt befindet. Wenn ja, rufen Sie für diesen Geber die Methode *gibFigur* und speichern das Ergebnis in der Instanzvariable *figur*. Falls *figur* nicht *null* ist, wird der Geber in der Instanzvariablen *gegebenVon* gespeichert und die Position (x,y) der Figur auf die aktuelle Mausposition gesetzt (e.getX() bzw. e.getY()).

In ***mouseDragged*** wird geprüft, ab *figur null* ist. Wenn nicht, wird die Position (x,y) der Figur auf die aktuelle Position des Cursors gesetzt (e.getX() und e.getY()).

In ***mouseReleased*** wird geprüft, ob *figur null* ist. Wenn nicht, wird in einer Schleife über *nehmer* und, ob sich der Cursor über dem jeweiligen Nehmer befindet. Wenn ja, wird dem Nehmer die Figur angeboten (*nimmFigur*). Nimmt er die Figur, werden *figur* und *gegebenVon* auf *null* gesetzt.
Nach der Schleife wird geprüft, ob *figur null* ist. Wenn nicht, hat kein Nehmer die Figur angenommen, dann geben wir sie an *gegebenVon* zurück (*nimmFigurZurueck*) und setzen dann *figur* und *gegebenVon* auf *null*.

In der Methode ***zeichne*** wird geprüft, ob *figur null* ist. Wenn nicht, wird *figur.zeichne* gerufen.

Alle nicht erwähnten Methoden von *MouseListener* und *MouseMotionListener* bleiben leer.

Das Spiel selbst

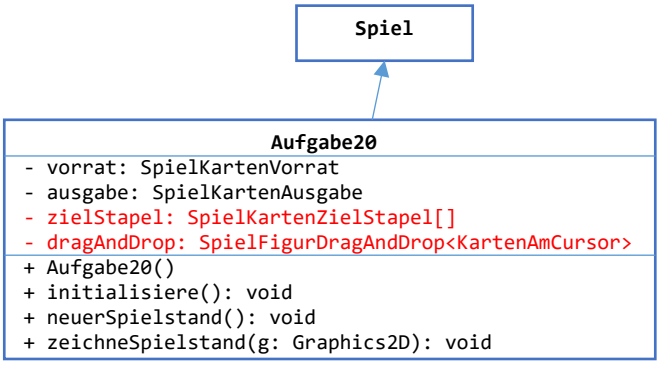
In ***Aufgabe20*** definieren Sie die weitere Instanzvariablen ***dragAndDrop***.

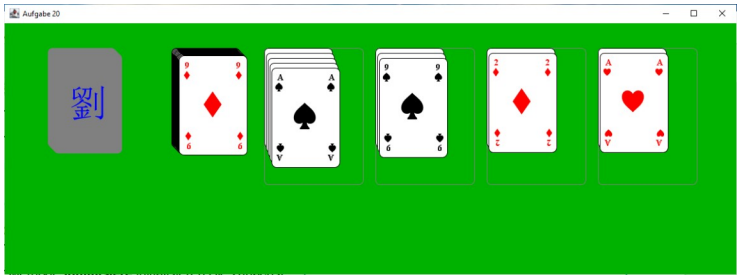
In der Methode **initialisiere** initialisieren Sie zunächst *dragAndDrop*. Dann rufen Sie dragAndDrop.*addSpielFigurGeber(ausgabe)*, d.h. sie registrieren die Kartenausgabe als Kartengeber.

In der Schleife, in der die Zielstapel erzeugt werden, registrieren Sie jeden der vier Zielstapel mit *addSpielFigurNehmer* bei *dragAndDrop*.

In ***zeichne*** zeichnen Sie auch *dragAndDrop*.

Jetzt sollte beim Klick auf die Kartenausgabe die oberste Karte aufgenommen werden und auf einer der Zielstapel ablegbar sein.

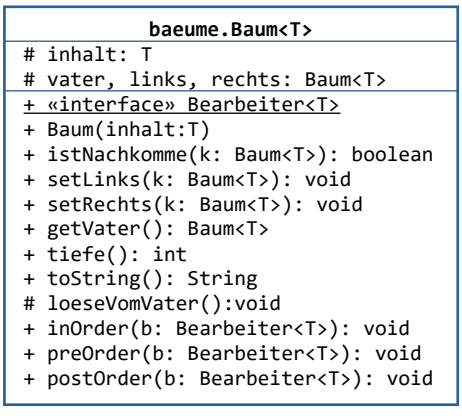




Aufgabe 21

Binärbaum

Erzeugen Sie das Paket ***baeume*** und schreiben Sie darin die Klasse ***Baum*** anhand des UML Diagramms, die wir später als Basisklasse verwenden. Sie realisiert einen Binärbaum.



Das innere Interface Bearbeiter sieht wie folgt aus:

```
public static interface Bearbeiter<T>{
     public void bearbeite(Baum<T> baum);
 }
```

Der **Konstruktor** speichert seinen Parameter in der entsprechenden Instanzvariable.

Die rekursive Methode ***istNachkomme*** stellt fest, ob *this* ein Nachkomme von *k* ist:

- Falls *vater null* ist, wird *false* zurückgegeben,
- falls *k* gleich *this,vater* ist, wird *true* zurückgegeben,
- ansonsten ruft sich die Methode rekursiv für *vater*.

Die Methode **loeseVomVater** führt folgende Schritte durch, wenn *vater* nicht *null* ist:

- setze den Teilbaum (links oder rechts) vom Vater zu null, der gleich this ist.
- Setzte this.vater zu null.

Die Methode ***tiefe*** berechnet rekursiv die Tiefe des Knotens. Die Tiefe der Wurzel ist 1, die Tiefe aller anderen Knoten ist um eins größer als die Tiefe des Vaters.

GetVater gibt die Instanzvariable *vater* zurück.

Die Methoden ***setLinks*** führt folgende Schritte durch:

- Falls *this* ein Nachkomme von k ist, wird eine *IllegalArgumentException* geworfen,
- Falls *links* nicht *null* ist, wird für *links loeseVomVater* gerufen.
- *this.links* wird zu *k* gesetzt.
- Wenn *k* nicht *null* ist, wird für *k loeseVomVater* gerufen und *k.vater* zu *this* gesetzt.

Die Methode ***setRechts*** arbeitet analog für das rechte Kind.

Die Methode ***toString*** *gibt inhalt.toString()* zurück.

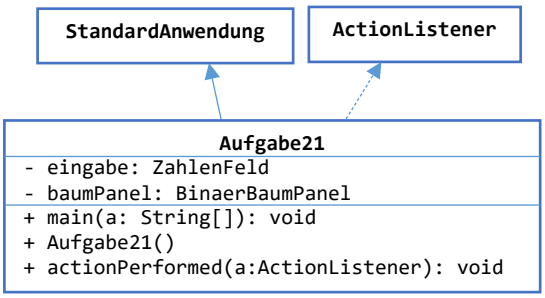
Die Methoden ***in- pre-*** und ***postOrder*** gehen in der jeweiligen Reihenfolge rekursiv durch den Baum und ruft für jeden Knoten *b.bearbeite*.

Jetzt holen Sie sich die Klasse ***BinaerBaumPanel*** aus dem Online-Kurs in das Paket *komponenten*. Es ist ein *JPanel*, das einen Binärbaum darstellen kann, der dem Konstruktor übergeben wird. Falls sich der Baum ändert, weil Knoten hinzugekommen oder gelöscht wurden, muss ***update*** gerufen werden. Mit ***ersetzeBaum*** kann der Baum durch einen anderen ersetzt werden.

Versuchen Sie die Klasse grob zu verstehen, sie verwendet intern eine *HashMap* zur Optimierung. Außerdem gibt es in *zeichneBaum* eine anonyme Klasse sowie in *getKnotenFarbe* eine *try…catch* Anweisung.

Die von *StandardAnwendung* abgeleitete Klasse ***Aufgabe21***, soll einen vollständigen Binärbaum mit zufälligen Zeichenkette erstellen.

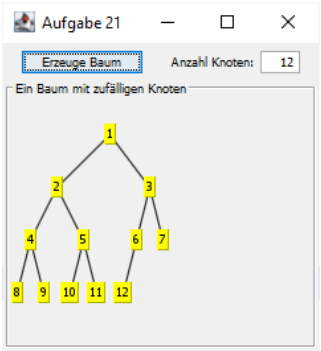
Der **Konstruktor** gibt sich ein Border-Layout, erzeugt das *JPanel oben* auf das ein *JButton* mit der Aufschrift „Erzeuge Baum“ und das *ZahlenFeld eingabe* gelegt werden, für beide registriert er sich als *ActionListener*. Dieses Panel wird mit NORTH auf der Oberfläche platziert.
Dann speichern Sie in der Instanzvariable *baumPanel* ein neues *BinaerBaumPanel* mit dem Titel „Ein Baum mit zufälligen Knoten“ und *null* als Baum. Dieses Baum-Panel wird auf CENTER platziert.



Üben Sie den Algorithmus zum Erzeugen eines Binärbaumes aus sseiner Level Order für verschiedene Bäume auf einem Blatt Papier, bis sie sicher sind, dass sie ihn verstanden haben.

Jetzt erzeugen wir einen Binärbaum mit n Knoten mit dem Algorithmus aus der Vorlesung, der eine Queue als Hilfsdatenstruktur verwendet. Dazu, führt die Methode ***actionPerformed*** folgende Schritte durch:

- Speichern der Zahl aus dem Eingabefeld in der Variable ***max***,
- Definition der lokalen Variable *naechsteZahl* mit dem Wert 1,
- speichern eines neuen *Baum<Integer>* Objekts dem Wert1 in der Variable ***wurzel***.
- Erzeugen einer Variable vom Typ *Queue<Baum<Integer>>*, die auf ein *LinkedList* Objekt zeigt.
- Einfügen von *wurzel* in die Queue (Methode *offer*).
- In einer while-Schleife, solange *naechsteZahl<=max* ist werden folgende Schritte durchgeführt:
  1. Speichern des nächste Elements aus der Queue (*poll*) in der Variable *aktuellerKnoten.*
  2. Erzeugen eines neuen Baum Objekts mit *naechsteZahl* als Wert, inkrementieren von naechsteZahl
  3. Einfügen des neuen Baum Objekts als linkes Kind in *aktuellerKnoten* (Methode *setLinks*).
  4. Einfügen *des neuen Baum Objekts* in die Queue.
  5. Falls *naechsteZahl<=max* ist, werden die Schritte 2-4 wiederholt, wobei der neue Knoten zum **rechten** Kind wird.



Am Ende von *actionPerformed* wird `baumPanel.ersetzeBaum(wurzel)` gerufen.

Um die Anwendung sicherer zu machen, fügen Sie um die Logik von *actionPerformed* einen *try-catch* Block ein, fangen die *NumberFormatException* ab und zeigen im Fehlerfall den rechts gezeigten Dialog.