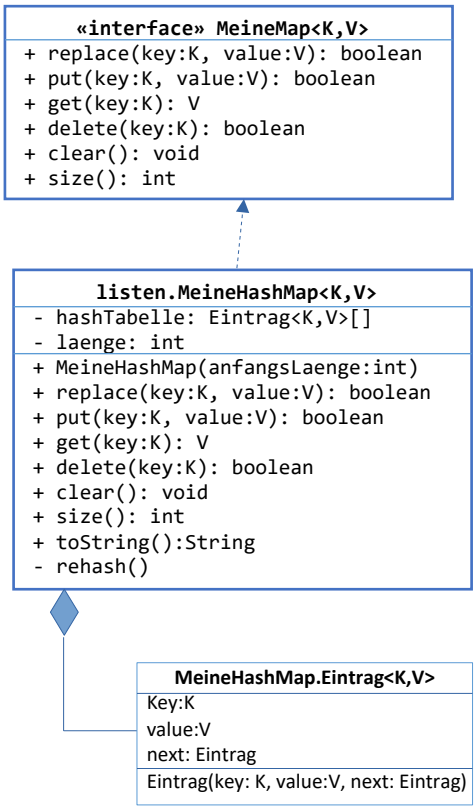
Bitte geben Sie ***MeineHashMap.java, KartenStapel**.java und **KartenVorrat.java*** bis zum 06.12.2020 ab!

Aufgabe 18

Hash-Liste



Wir beginnen mit dem Interface ***MeineMap<K,V>*** im Paket ***listen***, davon abgeleitet schreiben wir die Klasse MeineHashMap<K,V>. Beginnen Sie wie immer mit dem Gerüst anhand des UML Diagramms.

Der **Konstruktor** erzeugt das Array mit der Hashtabelle in der angegebenen Anfangslänge.

Die Methode ***replace*** wirft eine *IllegalArgumentException*, falls *key* oder *value null* sind. Dann berechnet sie den Hashwert von *key* als

`key=hashCode%hashTabelle.length`

und speichert *hashTabelle[index]* in der lokalen Variable *nadel*. Die variable *nadel* zeigt jetzt auf den ersten Eintrag der Überlauftabelle, oder sie ist *null* falls es keinen Eintrag mit dem Schlüssel *key* gibt. In einer *while*-Schleife, die solange läuft wie *nadel* nicht *null* ist, wird die Überlauf-Liste durchlaufen. Falls ein Eintrag mit *key* in der Überlauf-Liste gefunden wird, ersetzt man den Wert des entsprechenden Eintrags mit *value* und gibt *true* an das rufende Programm zurück. Falls die Schleife erfolglos durchlaufen wurde, geben wir *false* zurück.

Die Methode ***put*** ruft zunächst *replace*. Falls *replace false* zurück gibt, berechnet sie den Hashwert, trägt einen neuen Eintrag an den Anfang der entsprechenden Überlauftabelle ein und zählt *laenge* um eins hoch. Dann prüft sie, ob *laenge>3*hashTable.length/2* ist und vergrößert die Hashtabelle durch Aufruf der unten gezeigte Methode ***rehash***, die Sie auch online finden.

```
private void rehash(){   // neue Hashtabelle mit ca. doppelter Länge wie bisher
      Eintrag<K,V>[] alteTabelle = hashTabelle;
      hashTabelle = new Eintrag[2*(hashTabelle.length+1)-1]; // ungerade Länge
      for(Eintrag<K,V> nadel:alteTabelle){
             // alle Knoten in die neue Tabelle eintragen
             while(nadel!=null){
                    Eintrag<K,V> next = nadel.next;
                    int hash = nadel.key.hashCode()%hashTabelle.length;
                    nadel.next = hashTabelle[hash];
                    hashTabelle[hash] = nadel;
                    nadel = next;
             }
      }
}
```

In ***get*** wird eine ***IllegalArgumentException*** geworfen, wenn *key null* ist. Dann wird in der Überlauf-Liste zum Hashwert in einer *while*-Schleife nach *key* gesucht und der passende Wert zurückgegeben. Falls die Schleife erfolglos durchlaufen wurde, geben wir *null* zurück.

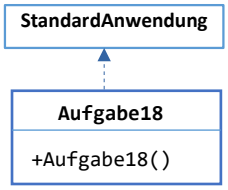
Die Methode ***delete*** wirft ebenfalls einen Fehler, falls *key* null ist. Dann sucht sie den Eintrag zu *key,* dabei merkt sie sich in der *while*-Schleife die Referenz auf den Vorgänger des aktuellen Eintrags. Wurde der Schlüssel gefunden, wird der Vorgänger mit dem Nachfolger des gefundenen Eintrags verknüpft oder, falls der Vorgänger nicht existiert, der Eintrag in der Hashtabelle auf *null* gesetzt. Dann wird *laenge* um 1 dekrementiert und *true* zurückgegeben. Wurde die Schleife erfolglos durchlaufen, geben wir false zurück.

In ***clear*** werden alle Einträge in der Hashtabelle zu *null* und *laenge* zu *0* gesetzt.

Die Methode ***toString*** finden Sie auf der nächsten Seite und auch im online Kurs:

```
@Override
public String toString(){
  StringBuilder s = new StringBuilder("Hash l="+laenge+", Länge Hashtabelle="+hashTabelle.length+":");
  int belegt = 0, laengsteKette=0;
  if(laenge>1000) s.append("\nListe ist zu lang, der Inhalt wird nicht ausgegeben (limit=1000)");
  for(int i=0;i<hashTabelle.length;i++){
     int kettenLaenge=0;
     if(laenge<=PRINT_LIMIT) s.append("\n"+i+": ");
     Eintrag<K,V> nadel = hashTabelle[i];
     if(nadel!=null) belegt++;
     while(nadel!=null){
       kettenLaenge++;
       if(laenge<=PRINT_LIMIT) s.append(" ["+nadel.key+":"+nadel.value+"]");
       nadel = nadel.next;
     }
     if(kettenLaenge>laengsteKette) laengsteKette = kettenLaenge;
  }
  int voll = Math.round(100f*belegt/hashTabelle.length);
  s.append("\n\nLängste Kette: "+laengsteKette+",  belegte Plätze in der Hashtabelle: "+belegt+" ("+voll+"%)");
  return s.toString();
}
```

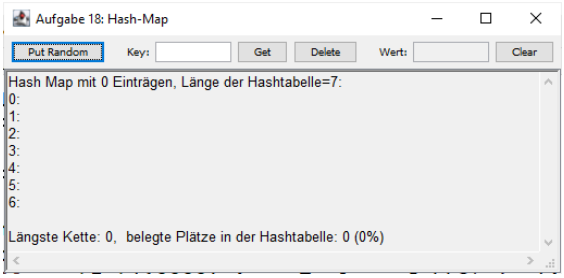
Zum testen der Anwendung schreiben wir die Klasse ***Aufgabe18***.



Der **Konstruktor** ist ähnlich aufgebaut ist wie der von *QueueTestPanel. Er* erzeugt zunächst das Objekt ***map*** vom Typ *MeineHashMap<Integer, String>* mit einer kurzen Anfangslänge, z.B. 7, um möglichst bald einen Re-Hash zu erzwingen.

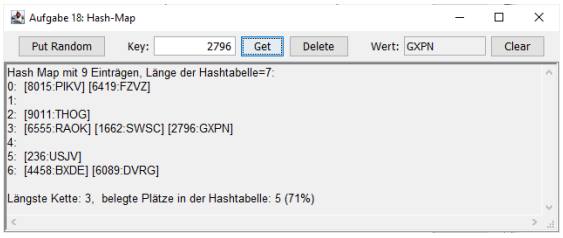
Geben Sie der Anwendung ein *BorderLayout*, erzeugen Sie das *JPanel* ***oben***, das auf *NORTH* platziert wird sowie die *ObjektTextArea* ***ausgabe***, deren Konstruktor Sie ***map*** übergeben. Auf *CENTER* platzieren Sie eine *JScrollPane* mit *ausgabe* als Viewport.

Auf das Panel *oben* legen Sie wie im Bild gezeigt die folgenden Elemente:



- Den JButton ***put*** mit dem Titel „Put Random“,
- einen *Jlabel* mit dem Text „ Key:“,
- Das *ZahlenFeld* ***eingabe***,
- den JButton **get** mit dem Titel „Get“
- den JButton **delete** mit dem Titel „Delete“
- einen *Jlabel* mit dem Text „ Wert:“,
- das JTextField ***wert***,
- den JButton ***clear*** mit dem Titel „Clear“

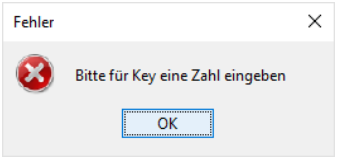
Das Layout sollte jetzt schon wie im Bild aussehen. Erzeugen Sie noch ein lokales Objekt vom Typ *Random* und dann definieren Sie für jede der Schaltflächen eine anonyme *ActionListener*-Klasse:



- ***clear*** ruft *map.clear() und ausgabe.update().*
- ***put*** fügt zu *map* einen neuer Eintrag mit einer zufälligen vierstelligen Zahl als Schlüssel sowie einer zufällige Zeichenkette als Wert hinzu (ArrayTools.zufallsString) und ruft *ausgabe.update*.
- **get** sucht den Wert zu dem Schlüssel aus *eingabe*, schreibt ihn in das Feld *wert* und ruft *ausgabe.update*.
- **delete** löscht den Eintrag mit dem Schlüssel aus *eingabe* und ruft *update*.

Beobachten Sie, wie sich die Has-Map aufbaut und die Hashtabelle füllt und bei Erreichen einer gewissen Länge ein Re-Hash stattfindet.

Sie können die Anwendung noch absichern, indem Sie bei *get* und *delete* in einem try-catch Block den Fehler *NumberFormatException* abfangen und mit einem *JOptionPane.showMessageDialog* einen Fehler melden, wenn der Benutzer keine Zahl eingegeben hat.

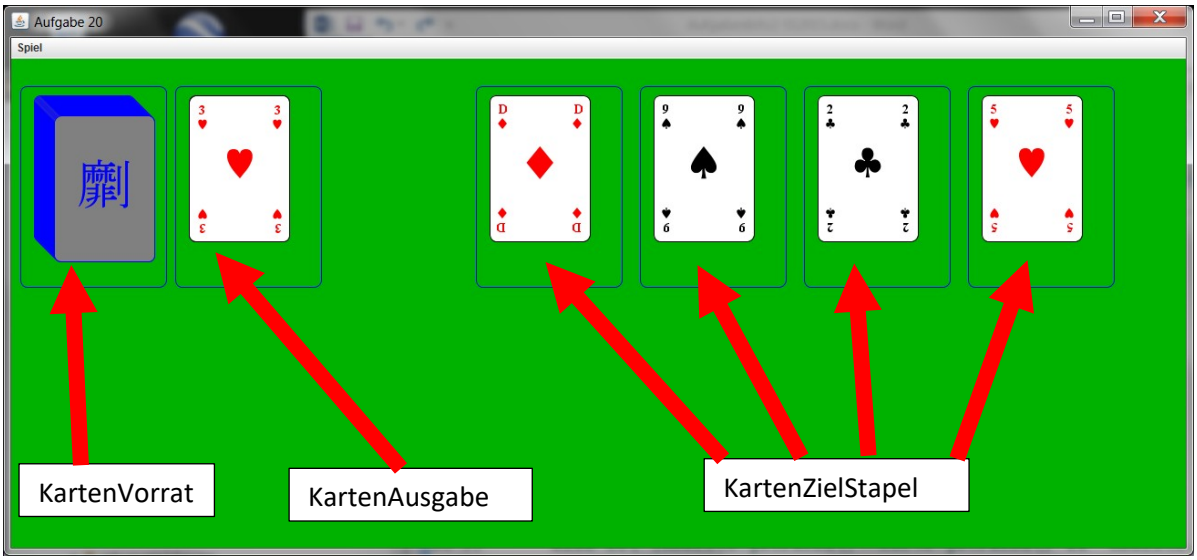


Aufgabe 19 Solitär-Kartenspiel: der Anfang

In dieser Aufgabe beginnen wir damit, ein Solitär Kartenspiel zu implementieren. Mit der Klasse *SpielKarte* haben wir ja schon ein wichtiges Element in Aufgabe 17 programmiert.

Das Solitär Spiel besteht im Wesentlichen aus mehreren Kartenstapeln, mit unterschiedlicher Funktionalität:

- Ganz links liegt der **Kartenvorrat**, seine Karten liegen verdeckt auf dem Stapel. Klickt man auf den Kartenvorrat, werden die drei obersten Karten auf die Kartenausgabe gelegt. Für den Kartenvorrat schreiben wir die Klasse ***KartenVorrat***.
- Die **Kartenausgabe** ist der Stapel rechts neben dem Kartenvorrat, die Karten liegen offen auf diesem Stapel. Klickt man auf die Kartenausgabe, wird die oberste Karte aufgenommen. Für die Kartenausgabe schreiben wir die Klasse ***KartenAusgabe***.

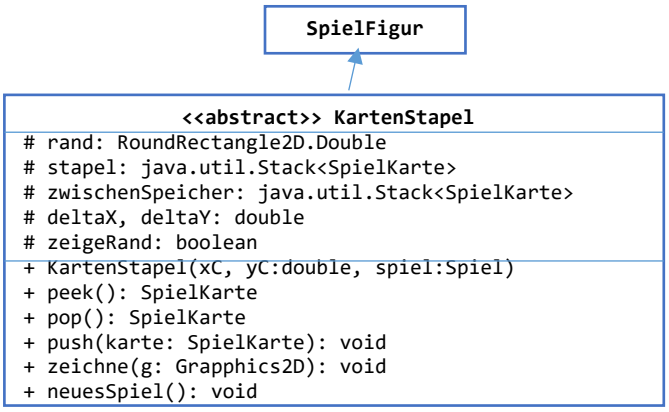


- Die vier Kartenstapel auf der rechten Seite sind die Zielstapel. Man kann Karten, die man von der Kartenausgabe aufgenommen hat, auf diesen Stapeln ablegen. Für die Zielstapel werden wir später die Klasse ***KartenZielStapel*** schreiben. In dieser Aufgabe programmieren wir nur den Kartenvorrat und die Kartenausgabe.

Kopieren Sie das Paket *aufgabe17* als ***aufgabe19*** und nennen Sie dort *Aufgabe17* in ***Aufgabe19*** um.

In diesem Paket schreiben wir zunächst die von *SpielFigur* abgeleitete **abstrakte** Klasse ***KartenStapel***, sie wird als Basisklasse für alle Kartenstapel dienen.

Die Instanzvariablen *rand*, *stapel* und *zwischenSpeicher* werden mit dem Standardkonstruktor initialisiert und *zeigeRand* wird auf *true* gesetzt. Die anderen Instanzvariablen werden nicht initialisiert.



Der ***Konstruktor*** ruft den Konstruktor der Basisklasse und übergibt ihm *xc*, *yc* sowie 160*220 Pixel als Größe. Dann wird der Radius der Ecken von *rand* (*archeight* und *arcwidth*) auf 15 gesetzt.

Die Methode ***peek*** prüft ob *stapel* leer ist und gibt entweder *stapel.peek()* oder *null* zurück.

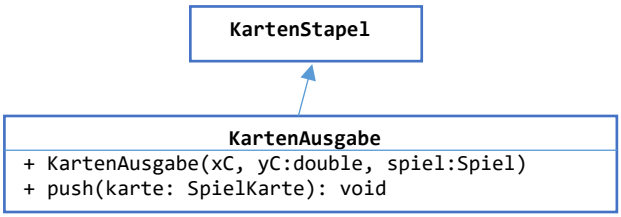
Die Methode ***pop*** prüft ob *stapel* leer ist und gibt entweder *stapel.pop()* oder *null* zurück.

Die Methode ***push*** ruft *stapel.push(karte).*

Die Methode ***zeichne*** prüft, ob *zeigeRand true* ist, wenn ja wird mit *rand.setFrame(this)* der Rand auf die aktuelle Position gesetzt und dann in Grau gezeichnet.
Dann werden die Karten in ***stapel*** mit einer vereinfachten *for*-Schleife gezeichnet, und zwar immer um *(deltaX,deltaY)* versetzt. Dazu werden zunächst die lokalen Variablen *xd* und *yd* definiert und mit *this.x* und *this.y* initialisiert. In der Schleife wird (x,y) der aktuellen Karte auf (*xd, yd*) gesetzt, die Karte gezeichnet und dann werden zu *xd* und *yd* zu *deltaX* bzw. *deltaY* hinzugezählt.

Die Methode ***neuesSpiel*** ruft stapel.clear.

Kartenausgabe



Die Klasse ***KartenAusgabe***, erweitert *KartenStapel*. Sie ist der zweite Stapel von links im Solitär-Spiel, die Karten auf diesem Stapel liegen offen.

Der ***Konstruktor*** ruft super(…) , setzt *zeigeRand* auf *true sowie deltaX* und *deltaY* jeweils auf den Wert 0.9, damit die Spielkarten versetzt dargestellt werden.

Die Methode ***push*** ruft *super.push()* und *karte.setZeigeVorderseite(true)* damit alle Karten auf diesem Stapel offen gezeigt werden.

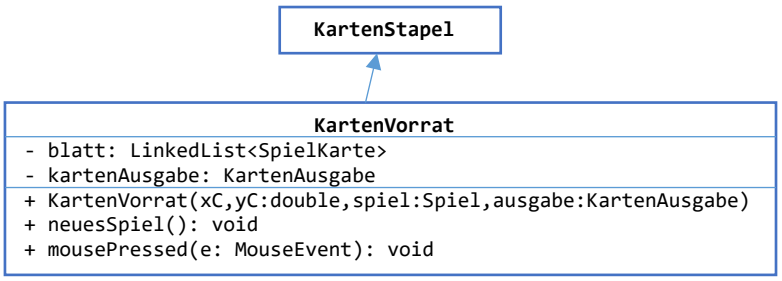
Die meiste Funktionalität ist schon in *SpielKartenStapel* enthalten, deshalb ist diese Klasse sehr einfach.

Kartenvorrat

Zunächst erweitern wir die Klasse *SpielKarte* um folgende Klassenmethode:

```
public static LinkedList<SpielKarte> erzeugeBlatt(Spiel spiel)
```

welche eine neues *LinkedList*<*SpielKarte*> Objekt erzeugt und in zwei ineinandergeschachtelten for-Schleifen über die Anzahl der Farben und die Anzahl der Werte die Liste mit Spielkarten füllt. Die Position aller Spielkarten setzen Sie dabei auf (0,0).



Die nächste Klasse ***KartenVorrat*** ist der Stapel ganz links im Solitär-Spiel, die Karten auf diesem Stapel liegen verdeckt und bei Klick auf den Stapel sollen die obersten drei Karten auf die Ausgabe gelegt werden.

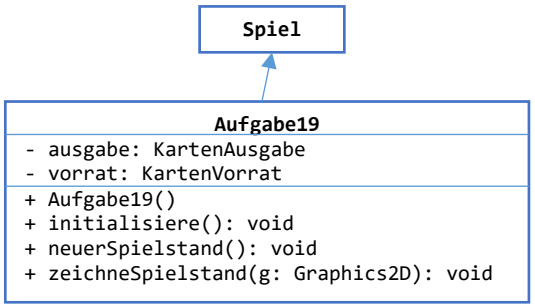
Ihr **Konstruktor** ruft *super*, speichert *ausgabe* in *kartenAusgabe* und setzt *deltaX* sowie *deltaY* auf 0.5 sowie *zeigeRand* auf *true*. Dann ruft er *SpielKarte.erzeugeBlatt* um in *blatt* ein komplettes Spielkartenblatt abzulegen und ruft *this.neuesSpiel*.

Die überschriebene Methode ***neuesSpiel*** legt ein neues gemischtes Blatt auf den Stapel. Dazu ruft sie zunächst *stapel.clear*, um den Stapel zu leeren. Dann mischt sie das Blatt, indem sie *Collections.shuffle* ruft. Am Ende werden die gemischten Karten in einer Schleife durch Aufruf von *this.push* zum Stapel hinzugefügt und *zeigeVorderseite* für jede Karte auf *false* gesetzt..

In *mousePressed* stellen wir fest, ob der Mausklick innerhalb des Stapels stattfand *this.contains(e.getPoint())*. Wenn ja, werden die obersten drei Karten vom eigenen Stapel genommen und auf die Kartenausgabe gelegt. Prüfen Sie jedesmal, ob der Stapel leer ist:

```
if(!stapel.empty()) …
```

Klasse *Aufgabe19*: Erster Schritt zum Solitär-Spiel



Der **Konstruktor** der von *Spiel* abgeleitete Anwendung ***Aufgabe19*** ruft *super* und setzt seinen Hintergrund auf ein dunkles Grün.

Die Methode ***initialisiere*** initialisiert die beiden Instanzvariablen.

Die Methode ***neuerSpielstand*** mach NICHTS.

Die Methode ***zeichneSpielstand*** ruft *zeichne* für die beiden Instanzvariablen.

Jetzt sollte beim Klick auf den Kartenvorrat die jeweils oberste Karte auf der Ausgabe abgelegt werden.

Wenn alles so funktioniert wie geplant, setzen Sie im Konstruktor von *KartenAusgabe* und *KartenVorrat* den Wert von *zeigeRand* auf *false*, den Rand wollen wir nur bei den späteren Ablagestapeln zeichnen.