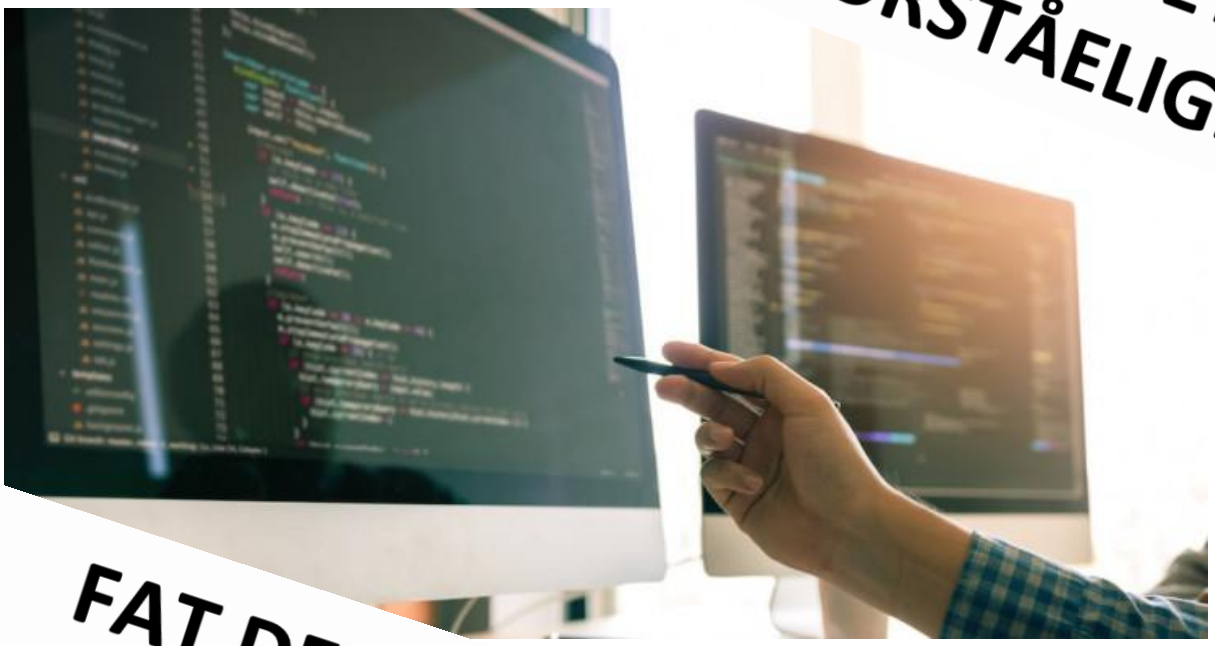
DEN STORE
FAT ALT-MANUAL



TAG EN KOP
KAFFE FØRST!

Indholdsfortegnelse

1. Intro

Jeg har hygget lidt med et forslag til en grundlæggende kode, som måske kan bruges som udgangspunkt til 7-kabale-projektet.
Det var egentligt slet ikke meningen, at jeg skulle skrive en hel masse kode, men jeg kom til at købe det her amazing tastatur:

Og så fik jeg lyst for at trykke en masse på alle knapperne.
Dette dokument er et forsøg på at forklare den struktur, der ligger bag koden, som den ser ud lige nu.

2. Funktionalitet lige nu

Koden kan følgende i skrivende stund:

- Indlæse fil fra harddisken, som er genereret af openCV.
- Opdatere kabalen i programmets datastruktur
- Finde en liste over samtlige mulige træk
- Vælge et træk (Lige nu: Det første træk. Så ingen AI implementeret)
- Holde styr på hvor mange kort, der endnu ikke er vendt om i hver bunke
- Holde styr på hvor mange kort, der er tilbage i trækbunken
- Holder styr på hvor mange kort, der er forsvundet (udplaceret) fra trækbunken
- Holde styr på byggebunkerne
- Holde styr på at man kun må blande bunken tre gange
- Undgå at foreslå det samme træk to gange i træk (Feks. at klør 7 bliver flyttet frem og tilbage mellem spar 8 og klør 8 uendeligt mange gange
- Undgå at spillet bliver hængende i at flytte kort mellem søjlerne i et uendeligt loop, men på et tidspunkt foreslår at trække et nyt kort
- Standser spillet, hvis der ikke er flere træk
- Standser spillet, hvis der ikke er flere kort som mangler at blive vendt (win-condition)

- Udskriver det foreslåede træk til spilleren

Koden kan IKKE følgende:

- Vælge optimalt træk blandt flere træk i samme kategori. Eksempelvis er det normalt bedst at flytte et kort, som eksponerer et kort med bagsiden opad. Lige nu flytter den bare et hvilket som helst kort. Med andre ord: AI mangler.

3. Overordnet filosofi

Koden kører cyklisk som følger:

1. Indlæs fil og opdatér kabalestruktur
2. Sammensæt ArrayListe over mulige træk
3. Vælg første træk på listen, skriv tekst til spilleren, og opdater kabalestruktur med det valgte træk

Derefter startes forfra med 1. punkt indtil spillet er tabt eller vundet

4. Syvkabalens regelsæt

Rimeligt standard regelsæt. Følgende fremhæves:

- Kun ét kort tilbydes fra trækbunken ad gangen (modsat visse regelsæt, som tilbyder tre kort ad gangen)
- Bunken må kun blandes 3 gange. Løber bunken tør for 4. gang, og der ikke er flere mulige træk, tabes spillet.
- Man må ikke tage kort tilbage fra byggebunkerne, når de først er placeret dér. (Dette simplificerer koden væsentligt. Som i virkeligt meget.

5. Main-klassen

Main-klassen ser således ud:

```
3  public class Main {
4      public static void main(String[] args) {
5
6          Table.debugText = true;      // Sets if you want various debug print-outs to be displayed
7
8          Table.createTable();         // Instantiates the Table. This is the primary data structure of the game board. Only one instance can exist.
9          ArrayList<Move> legalMoves;  // This ArrayList contains all legal moves found in the form of Move-objects
10         FindLegalMoves findLegalMoves = new FindLegalMoves();   // New findLegalMoves object.
11         AI ai = new AI();                                       // New AI object
12         FileHandler fileHandler = new FileHandler();            // New FileHandler object
13
14         while (Table.gameOn) {                          // Game loops forever while 'gameOn' is true
15             fileHandler.updateTable();                  // This method reads file created by OpenCV and updates Table accordingly
16             legalMoves = findLegalMoves.findMoves();    // Call method findMoves(). Returns an ArrayList of legal Moves
17             ai.thinkHard(legalMoves);                   // Call method thinkHard. Argument is the ArrayList of legal Moves
18         }
19     }
20 }
```

Bemærk variablen *Table.debugText*. Når denne variabel er TRUE, udprintes ufattelige mængder tekst på skærmen, der bruges/er blevet brugt til debugging. Hvis man sætter variablen til FALSE, udskrives kun teksten til spilleren. Alle debugging-udskrifter er pakket ind i en if-statement, som tjekker denne variabel.

eksempel:

```
if (Table.debugText) {  // Call method to print file contents if 'debugText' is TRUE
    printCards(this.cards);
}
```

Her ses, at metoden *printCards(this.cards)* kun kaldes, hvis debugText er sat til TRUE. Koden vrimler med de her. Jeg vil foreslå, at vi fortsat pakker alle debug-prints ind i sådan en if-statement, så det er let at slå til/fra.

Linie 8:

```
8       Table.createTable();        // Instantiates the Table. This is the primary data structure of the game board. Only one instance can exist.
```

Denne her er lidt speciel, for nu instantieres den klasse, Table, der indeholder den primære kabaledatastruktur. Normalt instantieres en klasse med NEW, men her gøres det via en metode. Når metoden *Table.createTable()* kaldes, bliver klassen instantieret med en NEW inde i selve metoden. Det er smart, for så kan man styre, at klassen kun kan instantieres én gang.
Det bliver tydeligere i kapitlet om Table-klassen.

Linie 9:

```
9       ArrayList<Move> legalMoves;  // This ArrayList contains all legal moves found in the form of Move-objects
```

Her deklarerer vi en ArrayList, *legalMoves,* som kommer til at indeholde alle de lovlige træk, som programmet senere finder frem til. Den kan indeholde objekter af typen *Move*.

Linie 10-12:

```
10      FindLegalMoves findLegalMoves = new FindLegalMoves();   // New findLegalMoves object.
11      AI ai = new AI();                                       // New AI object
12      FileHandler fileHandler = new FileHandler();            // New FileHandler object
```

I de her tre linier instantieres de tre vigtige klasser *FindLegalMoves, AI* og *FileHandler.* De tre instanser får – vældigt uoriginalt – variabelnavnene *findLegalmoves*, *ai* og *fileHandler*.

Klassenavnene afslører lidt selv, hvad de enkelte klasser har af ansvarsområde.

Allright, nu kommer selve programkørslen fra linje 14 til 18:

```
14      while (Table.gameOn) {                      // Game loops forever while 'gameOn' is true
15          fileHandler.updateTable();              // This method reads file created by OpenCV and updates Table accordingly
16          legalMoves = findLegalMoves.findMoves();// Call method findMoves(). Returns an ArrayList of legal Moves
17          ai.thinkHard(legalMoves);               // Call method thinkHard. Argument is the ArrayList of legal Moves
18      }
```

Det er en while-løkke, som kører uendeligt indtil *Table.gameOn* bliver sat til FALSE. Selve spillet kører nu med disse tre begivenheder, som looper derudaf:

1. Indlæs fil og opdatér kabalestruktur
2. Sammensæt ArrayListe over mulige træk
3. Vælg første træk på listen, skriv til spilleren, og opdater kabalestruktur med det valgte træk

Vi kan se, at metoden *findLegalMoves.findMoves()* returnerer en ArrayList af *move*-objekter. Vi definerede netop tidligere, at variablen *legalMoves* netop blev deklareret til at kunne indeholde en sådan ArrayList af *move*-objekter.

legalMoves bliver herefter fodret til metoden *ai.thinkhard* som et parameter. Denne metode gør alt arbejdet med at skrive til spilleren og opdatere kabalestrukturen, men det ser vi på senere.

Bom. Det var Main.

6. Table-klassen

Okay. Nu går det løs. Nu kommer vi til den klasse, der indeholder den primære kabaledatastruktur, samt et par andre features.

Her er de første par liner:

```
4   public class Table {
5
6       // This variable is only null until the class is instantiated for the first time
7       private static Table singleInstance = null;
```

Det her første felt er lidt specielt. Vi kan se, at variablen *singleInstance* faktisk er deklareret til at indeholde en instans af selve *Table*-klassen. Til at begynde med, er den defineret som en null-pointer. Men det bliver der lavet om på længere nede i metoden *createTable()*, som vi netop kaldte ude i Main. Det hele handler om, at *Table*-klassen er en såkaldt SINGLETON. Det er sådan en klasse, hvor man sikrer sig, at der kun kan oprettes én instans og at alle felter er PUBLIC STATIC. Så kan vi nemlig tilgå klassen og dens variable (blandt andet selve kabalestrukturen) fra alle andre klasser og metoder. Så behøver vi ikke at instantiere klassen lokalt, hvilket ville være noget rigtigt rod, da vi jo så vil havne med forskellige instanser af kabalen. Vi vil kun have én global kabale kørende. Jeg vil varmt anbefale at google ”singleton java” og læse lidt om konceptet.

Dette her link giver en god idé om konceptet: https://www.geeksforgeeks.org/singleton-class-java/

Så kommer selve kabalen, hvor kortene bliver gemt:

```
9       // The twelve positions of the solitaire table. This variable is STATIC and can be accessed from all other classes
10      public static List<List<String>> position;
11      /*
12          Positions 0-6:    Columns
13          Positions 7-10:   Foundation piles
14          Positions 11:     Draw pile
15      */
```

Der er simpelthen en ArrayList af ArrayLists af Strings. Selve variablen hedder *position.* Når kabalen bliver oprettet, kommer der til at være en overordnet ArrayList, som indeholder 12 Arraylists, som hver især indeholder et antal Strings (selve kortene). De første 7 ArrayLists indeholder søjlerne (indekseret 0 til 6), de næste fire ArrayLists indeholder byggebunkerne (indekseret 7-10) og den sidste ArrayList indeholder trækbunken.

Skal man feks. have fat i det 5. kort i søjle syv, skal man altså skrive:

position.get(6).get(4)

Når man bruger ArrayLists skal vi altså bruge get(), hvilket umiddelbart kan virke lidt besværligt. Men ArrayList-formen tilbyder også metoder som remove() og add(), hvilket almindelige arrays ikke gør. Så på den måde fungerer ArrayLists fint.

Så kommer der en ordentlig røvfuld felter:

```
17      public static boolean debugText;                // Boolean to switch debug printouts on/off
18      public static boolean gameOn = true;            // When true, game continues indefinitely.
19      public static int cardsLeftInDrawPile = 23;     // When game starts, draw pile has 23 cards
20      public static int cardsRemovedFromDrawPile = 0; // Track how many cards we permanently remove from draw pile
21      public static int shuffles = 3;                 // Track how many shuffles we may still do
22      public static int[] unseen = {0, 1, 2, 3, 4, 5, 6, 0, 0, 0, 0, 0};       // Track how many unseen cards remain in each column
23      public static String justMoved = "XX";                                   // Track last move to avoid moving same card two times in a row
24      public static int columnToColumn = 0;                                 // Track number of column moves to avoid looping
25      public static final int MAX_COLUMN_TO_COLUMN = 2;
26      // Maximum allowed column-to-column moves before game is forced to try another type of move
```

debugText: Det er den variabel, som vi kunne sætte TRUE eller FALSE i Main. Den bliver brugt megameget i alle klasser til at bestemme, om der skal printes diverse hjælpetekster.

gameOn: Den har vi også nævnt før. Den bliver født TRUE. Så snart vi sætter den til FALSE, stopper while-løkken ude i Main, og spillet stopper.

cardsLeftInDrawPile: Nå spillet starter, er der 23 kort i trækbunken. Hvorfor det? Jo, der er 52 kort i et sæt spillekort, vi lægger 28 kort ud i kabalen, og så vender vi det øverste kort i trækbunken. Tilbage har vi 23 kort i trækbunken med bagsiden opad.

cardsRemovedFromDrawPile: Hver gang vi fjerner et kort fra trækbunken, feks. fordi vi lægger et es ud i en byggebunke eller flytter et kort ned på en søjle, forsvinder et kort fra trækbunken. Det skal vi holde regnskab med, for når vi blander bunken, bliver trækbunken ikke bare tryllet tilbage til 23 kort. Den bliver kun tryllet tilbage til 23, MINUS de kort, som er forsvundet. Derfor tracker vi forsvundne kort her.

shuffles: Her sætter vi hvor mange gange vi må blande bunken. Jeg satte den bare til 3, fordi det er sådan et pænt tal. Vi kan bare sætte noget andet, hvis det er bedre.

unseen: Det her er et integer array. Den holder øje med hvor mange USETE kort, altså kort som endnu ikke er blevet vendt, der findes i hver søjle. De syv søjler initialiseres til 0,1,2,3,4,5 og 6, hvilket er situationen, når kabalen lige er lagt.
De øvrige bunker bliver initialiseret til 0 usete kort, hvilket forbliver sådan resten af spillet. Hvorfor så overhovedet have dem? Fordi nogle af algoritmerne tjekker alle bunker igennem på alle mulige måder, og så var det pludselig nødvendigt at alle bunker havde denne variabel tilknyttet. Lidt fjollet, men koden er vokset ret organisk, så ikke alle facetter er lige amazing.

justMoved: Denne her string gemmer det kort, der sidst er blevet flyttet. På den måde kan man tjekke, om spillet forsøger at flytte det samme kort to gange i træk. Det er vildt fjollet, hvis spillet forslår at flytte feks. hjerter 5 frem og tilbage mellem klør 6 og spar 6 i et uendeligt loop. Vi bruger *justMoved*-variablen senere hen i *FindLegalMoves*-klassen. Det fungerer sådan, at et ellers lovligt træk, som gerne vil flytte det samme kort to gange i træk, simpelthen ikke bliver tilføjet til *legalMoves*-ArrayListen.

columnToColumn: Her tracker vi hvor mange gange i træk, programmet flytter rundt på kort mellem søjlerne. Når vi flytter kort rundt mellem søjlerne, har det to formål:

- Enten ønsker vi at eksponere et uset kort, så det kan blive vendt om.
- Eller også ønsker vi at eksponere et kort, som kan flyttes op i byggebunkerne.

Problemet opstår, når programmet uhæmmet flytter rundt på kort mellem søjlerne, uden vi opnår en af de to nævnte positive effekter. Det kunne feks være:

1) Spar 3 til ruder 4.
2) Ruder 7 til klør 8.
3) Klør 9 til ruder 10
4) Spar 3 til ruder 4….. (det samme som i 1.)
5) Ruder 7 til klør 8…. (det samme som i 2.)
6) Klør 9 til ruder 10…. (det samme som i 3.)
7) Hvorefter disse tre træk gentages uendeligt uden der sker noget interessant…

Dette løses her ved kun at tillade spillet X antal søjle-til-søjle-træk, hvorefter programmet tvinges til at gøre noget ANDET end søjle-til-søjle, typisk flytte et kort ned fra trækbunken eller vende et nyt kort.

MAX_COLUMN_TO_COLUMN: Her sætter vi netop hvor mange søjle-til-søjle-træk vi vil acceptere inden programmet skal prøve noget andet. Lige nu er den 2. Det er måske lidt for lavt sat, men super fint i testfasen.

Så kommer konstruktøren:

```
28      // Constructor. It is private, so the Table class can only be instantiated from inside this class.
29      private Table() {
30          position = new ArrayList<>();              // The 'outer' ArrayList is initialized
31
32          for (int i = 0; i < 12; i++) {
33              position.add(new ArrayList<>());       // The 12 'inner' ArrayLists are added
34          }
35
36          for (int i = 0; i < 7; i++) {
37              position.get(i).add("XX");             // An "XX" is added to each column, denoting no cards
38          }
39
40          position.get(7).add("XC");                 // Foundation pile 1 is marked with clubs. X means no card.
41          position.get(8).add("XD");                 // Foundation pile 2 is marked with diamonds. X means no card.
42          position.get(9).add("XH");                 // Foundation pile 3 is marked with hearts. X means no card.
43          position.get(10).add("XS");                // Foundation pile 4 is marked with spades. X means no card.
44
45          position.get(11).add("XX");                // Draw pile is also initialized with "XX", denoting no card
46      }
```

Den er også lidt speciel. Den er nemlig PRIVATE, så klassen kan slet ikke instantieres udefra. Det gør vi i metoden *createTable,* som blev kaldt i Main. Vi kigger på den lige om lidt.

Her i konstruktøren befolker vi Table-instansen med værdier til felterne. For det første skal *position*-variablen (vores ArrayList af ArrayLists af Strings) først oprettes. Det sker i linie 30 med NEW.

Derefter skal de tolv ArrayLists oprettes til de tolv forskellige bunker: 7 søjler, 4 byggebunker og 1 trækbunke. Det sker med for-loopet i linjerne 32 til 34.

I linjerne 36 til 38 lægger vi et ”XX” ind i hver af de 7 søjler. Hvorfor nu det? Jo, når spillet skal undersøge om en bunke er tom, tjekker den bare om øverste kort er ”XX”. Desuden er det vigtigt, at der altid ligger et eller andet i en ArrayList. Hvis man forsøger at hente en oplysning fra en tom ArrayList, feks size(), får man en superirriterende null-pointer error, som stopper programmet. Så her sikrer vi, at der som minimum ligger et ”XX” i hver søjle.

I linjerne 40-43 lægge vi henholdsvis ”XC”, ”XD”, ”XH” og ”XS” ind i de 4 byggebunker. Det gør vi for at reservere hver bunke til henholdsvis Clubs, Diamonds, Hearts og Spades. Det har tre formål:

- Vi sikrer, at der altid ligger et eller andet i bunken, ligesom vi gjorde med søjlerne ovenover.
- Ved at reservere hver bunke til en bestemt farve, behøver vi ikke at ”lede” efter en ledig bunke, når der skal flyttes f.eks.. et es over i byggebunkerne. Vi véd bare fra starten, at klør altid skal over i bunke 7, ruder skal over i bunke 8, osv. Så skal vi bare tjekke om øverste kort i bunken har en værdi én lavere end det kort, vi kigger på.
- X bliver tolket af programmet som værdien 0. Så véd spillet, at et es (som har værdien 1) vil passe perfekt over i den tilhørende byggebunke.

I linje 45 får trækbunken også en ”XX” som første kort. Det har samme to formål som med søjlerne:

- Bunken er aldrig helt tom
- Vi kan let tjekke, om der skal vendes et nyt kort fra trækbunken.

Det var konstruktøren. Det var fedt.

Nu kommer *createTable*-metoden, som vi kalder én gang fra Main.

```
48      // The following method is the essence of a so called "singleton" class.
49      // When this public method is called for the first time, the Table Class is instantiated in
50      // the variable "singleInstance". If this method is called a second time, nothing happens.
51      public static void createTable() {
52          if (singleInstance == null) {          // This is only true the first time method is called
53              singleInstance = new Table();      // The table structure is created and initialized as seen in the constructor
54          }
55      }
```

Her tjekkes om *singleInstance* er NULL. Det var det første felt i klassen. *singleInstance* er NULL til at starte med. Det vil medføre, at programmet går ind i den viste IF-statement. Så instantieres klassen med NEW Table(). Det medfører, at konstruktøren bliver kaldt og hele vores klasseinstans får plads i hukommelsen og udfyldt med de ønskede startværdier. *singleInstance* er herefter ikke længere NULL. Kaldes denne metode ved en fejl igen, vil programmet IKKE længere gå ind i denne IF-statement. Vi sikrer os på denne måde, at der aldrig kan eksistere mere end én kabale.

I linjerne 57 til 87 møder vi følgende metode:

```
57      public static void printTable() {          // Method to print the entire Table for debugging
58          System.out.println();
59          System.out.println("******* CURRENT TABLE *******");
```

Kun de tre første linjer er vist her. Denne metode printer hele kabalen ud i konsollen, altså *position*-variablen. Den bruges kun til debugging og bliver kaldt et andet sted i koden hvis *debugText* er sat til TRUE i main.
Det kan måske være en god ide at kigge metoden igennem for at få en forestilling om hvordan man kan iterere igennem kabalen.

Vi kigger lige på en detalje fra *printTable,* nemlig når trækbunken bliver printet ud:

```
61          int drawpileSize = Table.position.get(11).size();
62          System.out.print("Draw pile: ");
63          for (int i = 0; i < drawpileSize; i++) {
64              System.out.print(Table.position.get(11).get(i) + " ");      // Print drawpile.
65          }
```

Først henter vi størrelsen af trækbunken. Der 3 elementer i at finde den størrelse:

- *Table.position:* Fordi det hedder vores kabalestruktur, når vi kalder den fra en anden klasse
- *get(11):* Fordi trækbunken har indeks 11 i *Table.position*-ArrayListen
- *size():* Fordi size() returnerer antallet af elementer (strings) i den ArrayListe vi fik fat på med indeks 11.

Derefter kører vi et for-loop et antal gange svarende til den længde, vi fandt. Resultatet kunne feks. se sådan her ud:

XX KH 5D 2H

Dette vil betyde:
XX (Som altid er nederste kort)
Hjerter konge
Ruder 5
Hjerter 2

Til allersidst i klassen fra linjerne 89 til 155 finder vi metoden *convertToText:*

```
89  @   public static String convertToText(int value, String card) {    // Receives raw card such as '12' and "H" and returns plaintext version,
90
91          String plainText = "";
92
93          switch (card) {
94              case "C":
95                  plainText = "klør ";
96                  break;
```

Her er bare vist de første par linjer.
Den bliver brugt, når kortenes værdi og kulør skal oversættes til klar tekst, når spilleren skal have præsenteret trækforslag. ”C” bliver til ”klør” og ”5” bliver til ”fem”, osv. Bemærk at metoden modtager to parametre, en integer og en string.

Det var Table-klassen. Fedt fedt fedt.

7. FileHandler-klassen

FileHandler-klassen bliver instantieret i Main. Den indeholder 3 felter:

```
5   public class FileHandler {
6
7       private String fileName = "C:/Users/janmu/IdeaProjects/Solitaire/src/tableFile";
8       private String splitter = ",";
9       private String[] cards;
```

fileName er stien til det sted, hvor OpenCV-filen ligger. Der findes uden tvivl en smartere måde at gøre det på, for man skal være helt sikker på at filen altid ligger fuldstændigt samme sted. Det må være en plan at fikse det snarest.

splitter er den karakter, som dataene bliver splittet op med i data-filen fra OpenCV. Og det er altså et komma.

cards[] er det array af Strings, som kommer til at indeholde de tolv kort fra filen, efter den er blevet læst og klippet i 12 små bidder.

Selve filen ser sådan her ud:

```
1    KS,UU,UU,UU,UU,5S,5H,QC,7H,8D,9D,9C
```

Første plads er trækbunken, her KS.

De næste 4 pladser er byggebunkerne. Programmet bruger slet ikke oplysninger om byggebunkerne, så der kan stå hvad som helst, feks. UU. Grunden til, at programmet ikke skal bruge oplysningerne om byggebunkerne er, at spillet altid selv bestemmer hvad og hvornår der lægges kort op i byggebunkerne. Spillet kan derfor selv holde styr på dem.

Derimod bliver der jævnligt præsenteret overraskelser i trækbunken og i søjlerne, når vi vender kort om. Derfor skal vi have et snapshot af trækbunken og søjlerne hver runde, men ikke byggebunkerne.

De sidste 7 pladser er søjlerne. Vi behøver kun det øverste kort. Bagvedliggende kort er gemt i *Table.position*-strukturen.

Her kommer den vigtigste metode i *FileHandler*-klassen:

```
14    public void updateTable() { // This is the only public method in this class. Amazing encapsulation.
15        readFile();             // Method to read the file created by OpenCV.
16        populateTable();        // Method that fills in the Table with the cards from the file
17        if (Table.debugText) Table.printTable();    // Print the Table after it is updated with new cards from file
18    }
```

updateTable() bliver kaldt fra Main hver eneste runde.

Her sker 2 ting:

- *readFile*-metoden kaldes. Metoden læser filen fra OpenCV og udfylder *card*-arrayet med de aktuelle kort på bordet.
- *populateTable*-metoden kaldes. Denne metode adderer kortene til vores Table.position-struktur – men kun hvis kortet er anderledes end det kort, der allerede lå der i forvejen. På den måde sikres, at kun nyligt afslørede kort bliver tilføjet vores søjler eller vores trækbunke. Alle andre ændringer, som når man flytter kort rundt mellem søjler eller flytter et kort op til en byggebunke, bliver udført af AI-klassen, når et givent træk er blevet valgt for spilleren. Derfor behøver OpenCV ikke tage billeder af alle kort, men kun de øverste kort.

Vi behøver ikke at kigge nærmere på *readFile*-metoden. Det er en ret standardiseret opbygning, som henter filen med en filerReader()-metode og klipper den i stykker med en split()-metode. Den kan man hyggelæse, hvis man elsker mystiske Java-konstruktioner som bufferedReader, readLine og IOexeptions.

Her er den fulde *populateTable*-metode:

```
40      private void populateTable() {
41          // All cards from the file is compared to the current top card on the Table.
42          // If they are different, it can only be because a card has been flipped. If so, that
43          // card is then inserted into the column or draw pile in Table
44
45          if (Table.debugText) System.out.println("\n******* NOW UPDATING TABLE WITH CARDS FROM FILE *******");
46
47          int sizeDrawPile = Table.position.get(11).size();                          // Get size of draw pile
48          if (!cards[0].equals(Table.position.get(11).get(sizeDrawPile - 1))) {      // Compare top card of draw pile with card from file
49              Table.position.get(11).add(cards[0]);                                  // If new, add it
50              if (Table.debugText) System.out.println("Card added to draw pile: " + cards[0]);
51          }
52
53          for (int i = 0; i < 7; i++) {
54              // only add card from file to column if it is DIFFERENT (newly flipped)
55              if (!Table.position.get(i).get(Table.position.get(i).size() - 1).equals(cards[5 + i])) {
56                  Table.position.get(i).add(cards[5 + i]);
57                  if (Table.debugText) System.out.println("Card added to column " + i + ": " + cards[5 + i]);
58              }
59          }
60          // Note: Foundation piles are not updated using the file. They are updated by AI.
61          if (Table.debugText) System.out.println("******* TABLE UPDATE COMPLETE *******");
62      }
```

Linie 45:
Metoden er fyldt med *Table.debugText*-checks, som printer tekst ud i konsollen, hvis *debugText* er TRUE. Det er som nævnt til generel debugging, så vi kan se om vores datastruktur bliver udfyldt som vi forventer det.

Linjerne 47-51:
Først finder vi størrelsen på Table.position.get(11) – med andre ord, så finder vi antallet af kort i byggebunken inklusiv ”XX”. Dette antal gemmes i variablen *sizeDrawPile.*
Så kigger vi på det allerøverste kort i byggebunken. Det vil have indeks *sizeDrawPile minus én* (da ArrayLister er nul-indekseret).
Så sammenligner vi dette øverste kort fra byggebunken med det kort, som vi netop har fået ind fra filen. Dette friske kort fra filen er gemt i *cards*-arrayet på indeks [0].
Vi sammenligner de to kort med *equals*-metoden i linie 55.
Hvis kortene er FORSKELLIGE, gemmes det nye kort i vores *Table.position*-struktur på indeksplads 11, som er indekset for byggebunken.

Det samme finder sted for søjlerne i linjerne 53 til 59. Her bruges en forløkke, så alle 7 søjler køres igennem.
Bemærk noget lidt fjollet: Søjlernes indeksplacering er forskellig i *Table.position* og i *cards.* Derfor bliver der lagt værdien 5 til ”i” i linjerne 55, 56 og 57. Lidt tosset, men på dette tidspunkt var opbygningen af både OpenCV-filen og *Table.position*-strukturen for længst på plads og det ville være supernederen at skulle ændre på det, i forhold til at skrive ”+5” tre steder.

Bummelum. Så er kabalen blevet opdateret.

Sidste metode i *fileHandler*-klassen hedder *printCards.* Den modtager som argument det *cards*-array med strings, som vi trak ud af filen fra OpenCV:

```
65  @    private void printCards(String[] cards) {       // Print current file contents for debugging
66           System.out.println("******* FILE CHECK *******");
67           System.out.println("Number cards found in file: " + cards.length);
68           System.out.println("Draw pile: " + cards[0]);
69           System.out.println("Foundation pile 0: " + cards[1]);
70           System.out.println("Foundation pile 1: " + cards[2]);
71           System.out.println("Foundation pile 2: " + cards[3]);
72           System.out.println("Foundation pile 3: " + cards[4]);
73           System.out.println("Column 0: " + cards[5]);
74           System.out.println("Column 1: " + cards[6]);
75           System.out.println("Column 2: " + cards[7]);
76           System.out.println("Column 3: " + cards[8]);
77           System.out.println("Column 4: " + cards[9]);
78           System.out.println("Column 5: " + cards[10]);
79           System.out.println("Column 6: " + cards[11]);
80           System.out.println("******* FILE CHECK END *******");
81       }
```

Metodens eneste formål er at printe indholdet fra filen ud. Det bruges kun til debugging. Metoden bliver kaldt i *readFile*-metoden, hvis *debugText* er TRUE.

Allright, det var supersjovt.

Nu skal vi til at kigge på træk.

8. Move-klassen

Move-klassen blive brugt til at opbevare et gyldigt træk. For hvert gyldigt træk, bliver der instantieret et *Move*-objekt. *Move*-klassen indeholder udelukkende felter og getter-metoder. Det er altså en ren dataklasse. Der er ingen mystiske metoder eller lignende. (På nær en toString-metode)

Vi tager et kig på felterne:

```
1   public class Move {
2
3       private int fromPosition;   // Which column (or draw pile) can we move something from
4       private int toPosition;     // Which column or foundation pile can we move something to
5       private int cut;            // Where to cut the column when moving a group of cards. 1 = top card
6       private int type;           // What type of move we are doing (see below)
7       // Type 1 : move card from a column or draw pile to a foundation pile
8       // Type 2 : move a king from a column or the draw pile to an empty column
9       // Type 3 : move one or more cards from a column to a column
10      // Type 4 : move the card from the draw pile to a column
11      private String card;        // The 'raw' card directly from the file. (3D, 6S, etc)
12      private String plainText;   // The 'raw' card converted to nice plaintext such as "spar seks".
```

fromPosition: Dette felt indeholder indeksnummeret på den søjle eller trækbunke, som et givent kort kan flyttes FRA. Hvis vi eksempelvis kan flytte spar 5 fra første søjle til sidste søjle, vil dette felt have værdien '0', da første søjle har indeks 0.
Er der tale om et træk fra trækbunken, vil der stå '11' i dette felt, da trækbunken har index 11 i *Table.position.*

toPosition: Dette felt indeholder indexnummeret på den søjle eller byggebunke, som kortet kan flyttes TIL. Vi husker i øvrigt for en god orden skyld, at søjler har indeksnumre 0 til 6, byggebunker har indeksnumre 7 til 10 og trækbunken har indeksnummer 11.

cut: Tallet her angiver HVOR i en søjle, at en søjle "skæres over", når der flyttes dele af en søjle. Det nederste kort, altså kortet "tættest på bordpladen", har cut 1.
Hvis det øverste kort flyttes (det fuldt synlige kort, som ikke dækkes af andre kort), vil cut være lig med antallet af kort i bunken.

Eksempel:

Vi har følgende kortstak i søjle 4:

XX
Hjerter 6 <- cut 1
Kløv 5 <- cut 2
Ruder 4 <- cut 3
Spar 3 <- cut 4

Følgende kort ligger i søjle 5:

XX
Spar 5

Vi kan altså flytte de to kort ruder 4 og spar 3 over i søjle 5. Dette udløser følgende værdier:
fromPosition: 3 toPosition: 4 cut: 3

type: Helt grundlæggende findes der fire slags træk:
- Flyt et kort fra en søjle eller fra trækbunken til en byggebunke
- Flyt en konge til en tom søjle
- Flyt et eller flere kort fra søjle til søjle
- Flyt et kort fra fra trækbunken til en søjle

Disse fire typer træk er navngivet tallene 1, 2, 3 eller 4 og gemmes i feltet *type.*

card: Kortets navn i rå format direkte fra *Table.position* såsom KD for Ruder Konge (King of Diamods) eller 0H for Hjerter 10.

plainText: Kortets navn i klar tekst, såsom "kløv seks". Den tidligere nævnte metode *convertToText* fra Table-klassen bliver brugt til at danne klarteksten.

Filosofien bag dette move-objekt er først og fremmest, at de øvrige klasser (specielt AI-klassen) har adgang til alle nødvendige oplysninger om alle de mulige træk uden at skulle rode rundt i *Table.position*-strukturen. Move-objekterne indeholder alt need-to-know. (Næsten i hvert fald.)

Derudover indeholder Move-klassen getter-metoder, så man kan få fat i dataene.

Her er et eksempel på et udfyldt move-objekt:

```
Move{fromPosition=4, toPosition=6, cut=1, type=3, card='8D', plainText='ruder otte'}
```

Move-objektet fortæller her, at man kan flytte et kort fra søjle 4 til søjle 6, der er tale om det nederste kort (cut 1), det er type 3 (søjle-til-søjle), kortet er 8D, hvilket i klartekst er ”ruder otte”.

Endelig indeholder *Move*-klassen en overrided toString-metode:

```
47      @Override
48      public String toString() {      // OMG the toString-method is overrided. Amazing Java skills.
49          return "Move{" +
50                  "fromPosition=" + fromPosition +
51                  ", toPosition=" + toPosition +
52                  ", cut=" + cut +
53                  ", type=" + type +
54                  ", card='" + card + '\'' +
55                  ", plainText='" + plainText + '\'' +
56                  '}';
57      }
```

Man er nødt til at override toString-metoden, hvis man vil bruge den til noget. Hvis man bare kalder toString() direkte på et objekt, får man bare returneret adressen på objektet. Derfor er det en god idé at skrive sin egen metode som explicit returnerer indholdet af alle felterne i klassen. Vi bruger det til debugging senere, hvor vi får programmet til at udskrive hele listen med fundne lovlige træk. (Linien med Move-eksemplet er et screenshot fra en move.toString)

Jamen cool. Det var *Move*-klassen.

9. FindLegalMoves-klassen.

Nu skal vi til at gennemtrevle kabalen for mulige træk.

Konceptet er som følger:

Først undersøges for mulige type 1-træk. For hvert muligt type 1-træk instantieres et Move-objekt, som beskrevet i foregående kapitel. Dette move-objekt føjes til en ArrayList med move-objekter.

Derefter undersøges for type 2-træk, derefter type 3-træk og til sidst type 4-træk. Hver gang et træk bliver fundet, instantieres et move-objekt med oplysningerne og add'es til ArrayListen *allMoves* med Move-objekter.

Derudover er der indbygget nogle små features i klassen, som egentlig burde være lagt ud til AI'en, men det var bare vildt meget nemmere at lægge det ind her. Feks. er det dumt, hvis programmet vælger at flytte en konge fra en tom søjle til en tom søjle sådan her:

```
XX  XX          ->          XX XX
KD                              KD
```

Det har ingen funktion overhovedet. Sådan en træk bliver frasorteret allerede her, så trækket slet ikke kommer med i listen over lovlige træk, selvom det sådan set er fuldt lovligt. Der er bare ingen grund til at AI'en skal ligge og rode med sådan noget, når det kan frasorteres på et tidligere tidspunkt.

Følgende træk bliver frasorteret allerede på dette stadie:

- Kongeflytning fra tom søjle til tom søjle
- Det samme kort flyttes to gange i træk
- For mange træk i streg mellem søjlerne (Fastsat med variablen MAX_COLUMN_TO_COLUMN som nævnt på side 7)

Tag en kop kaffe ekstra. Det er en ret gusten klasse og også mere kompleks end først forudset. Den kan uden tvivl shines op og gøres pænere.

Klassen har ét felt: En ArrayList af *Move*-objekter, med variabelnavnet *allMoves.* Her gemmer vi alle de fundne *Move*-objekter, inden listen returneres til Main.

```
3   public class FindLegalMoves {
4
5       ArrayList<Move> allMoves;
6
7       public FindLegalMoves() {
8           this.allMoves = new ArrayList<>();
9       }
```

Konstruktøren er simpel, den instantierer bare ArrayListen, så den er klar til udfyldning. (Hvis man glemmer at instantiere en ArrayList, får man en NULL-pointer exception. Sir' det bare.)

Her kommer den primære metode i klassen, *findMoves*. Det er den, vi kalder fra Main.

```
11      public ArrayList<Move> findMoves() {
12          allMoves.clear();           // We re-use the same FindLegalMoves object again and again, so we need to clear the allMoves variable each time
13
14          checkFoundations();         // Check if can move card from a column or from the draw pile to a foundation pile
15          checkKingToFreeSpot();      // Check if we can move a king from a column or the draw pile to an empty column
16          checkColumns();             // Check if we can move one card or a group of cards from a column to a column
17          checkDrawPile();            // Check if we can move the card from the draw pile to a column
18
19          if (Table.debugText) {      // For debugging: Print out all move-objects from the allMove ArrayList.
20              System.out.println("******* POSSIBLE MOVES FOUND *******");
21              for (int i = 0; i < allMoves.size(); i++) {
22                  System.out.println(allMoves.get(i).toString());
23              }
24          }
25          return allMoves;
26      }
```

Linie 12:
Vi starter med at cleare *allMoves*-ArrayListen. Det er den samme instans af klassen, som vi bruger igen og igen, så vi kan ikke have gamle moves liggende fra tidligere runder.

Linjerne 14 til 17:
Her kalder vi de fire metoder, der tjekker for de fire typer af træk. Hver metode fylder selvstændigt moveobjekter ind i *allMoves.* Det har den rigtigt fine fordel, at *allMoves* kommer til at indeholde alle type 1 moves først, derefter alle type 2 moves, derefter alle type 3 moves og til sidst alle type 4 moves.

Det viser sig, at denne rækkefølge faktisk er et fint udgangspunkt for en simpel AI. Der bliver ikke taget nogen fine beslutninger eller gjort store overvejelser. Men det sikrer, at programmet feks. prøver at finde kort til byggebunkerne inden det begynder at bladre uhæmmet ned gennem trækbunken. Mit gæt er, at mange kabaler vil kunne gå op blot ved at følge denne rækkefølge.

Linjerne 19 til 24:
Her bliver vores *Move*-objekter printet ud efter indsamling med den fine toString-metode som tidligere beskrevet.

Linie 25:
Hele *allMoves*-ArrayListen bliver returneret til kalderen, som var fra Main.

Nu kigger vi på type 1-undersøgelsen, om der kan flyttes et kort fra en søjle/trækbunke til en byggebunke.
Det er metoden ***checkFoundations().***

Jeg gennemgår ikke metoden i detaljer. Men overordnet set, indeholder den to for-løkker, en ydre og en indre.

Den ydre for-løkke tager fat i det øverste kort i hver søjle. Det ser vi i linie 32:

```
32      for (int i = 0; i < 7; i++) {                                   // Iterate columns
```

Den indre for-lykke kigger på byggebunkerne. Det ser vi i linie 40:

```
40          for (int j = 7; j < 11; j++) {                              // Iterate foundation piles
```

De variable, der hentes fra *Table.position* skal oversættes en smule. Vi kan ikke sammenlige feks. QH og KH direkte. Vi skal have oversat den første karakter til en værdi mellem 1 og 13. Dette gør vi med metoden *extractValue* som vi finder i linjerne 204 til 237. Den fungerer som en simple switch/case, der kigger på første karakter i inputtet og returnerer et tal mellem 1 og 13.

I linie 49 sammenligner vi om værdien af kortet er 1 højere end værdien af det kort, der allerede ligger i byggebunken SAMT om kuløren er den samme:

```
48              // Check if top card of a column can be placed on a foundation pile by comparing values and suits.
49              if (valueColumnCard == valueFoundationCard + 1 && suitColumnCard.equals(suitFoundationCard)) {
```

Hvis det er tilfældet, kan vi konvertere kortet til klar tekst (med *convertToText*-metoden beskrevet på side 10) og lave et nyt moveobjekt:

```
51              String text = Table.convertToText(valueColumnCard, suitColumnCard);     // convert raw card to plain text with amazing convertToText method
52              Move move = new Move( fromPosition: i,  toPosition: j,  cut: topColumnIndex,  type: 1,  card: topColumn,  plainText: text);    // Create Move object
53              allMoves.add(move);     // Add the new legal move to ArrayList of legal moves
```

I linie 53 tilføjes *Move*-objektet til samlingen af lovlige moves.

I linjerne 59 til 77 gentages succesen, nu er det bare det øverste kort i trækbunken, der tjekkes for match med en byggebunke.

Så kaldes metoden **checkKingToFreeSpot()** (Type 2). Konceptet er det samme, nu tjekker vi bare om der findes en konge et sted, der kan placeres i en tom søjle.

Bemærk linje 92!

```
92              if (Table.position.get(j).get(k).substring(0, 1).equals("K") && Table.unseen[j] != 0) {     // Is the card a king? And sits ontop unseen cards?
```

Her sikrer vi os, som tidligere nævnt, at en konge IKKE befinder sig som eneste kort i en ellers tom søjle. Så er der nemlig ingen grund til at flytte den. Det var det ekstra bonus-tjek, som jeg nævnte på side 16.

Så kaldes metoden **checkColumns()** som med to for-loops løber alle søjler igennem for at finde matches for søjle-til-søjle træk. (Type 3)

Bemærk allerførste begivenhed i metoden:

```
126     private void checkColumns() {
127         if(Table.columnToColumn == Table.MAX_COLUMN_TO_COLUMN){     // If limit on column-to-column moves is reached, skip this check
128             if(Table.debugText) System.out.println("Max column-to-column reached, no type 3 moves allowed.");
129             return;
130         }
```

Her tjekker vi om vi har ramt loftet for antal søjle-til-søjle moves i streg. Er dette tilfældet, returneres uden at der bliver undersøgt for mulige type-3 træk. (columnToColumn-variablen bliver nulstillet hver gang der foretages et ANDET træk end søjle-til-søjle)

I forbindelse med sammenligningerne af kort for mulige søjle-til-søjle-træk, er det nødvendigt at kende farven på kortene, altså sort/rød. Denne farve bliver fundet med et metodekald i stil med den tidligere nævnte *extractValue.* Nu bruger vi bare metoden *extractColor,* som vi finder i linjerne 239 til 245.

Så er vi klar til at tilføje søjle-til-søjle-move-objektet til vores ArrayList med tilladte træk. Men først tjekker vi, om det givne kort er identisk med det kort, der blev flyttet i forrige runde. Så er det nemlig spild af tid at pille ved det igen, som nævnt på side 16. Det foregår her i linie 161:

```
161                 if (!move.getCard().equals(Table.justMoved)) {          // Only accept move if card was not recently moved
162                     allMoves.add(move);         // Add the new legal move to ArrayList of legal moves
```

Til sidst kaldes metoden **checkDrawpile,** som undersøger, om vi kan flytte et kort fra trækbunken til en søjle (type 4) . (Vi undersøger ikke om vi kan flytte et kort fra trækbunken til en byggebunke – det gjorde vi nemlig allerede oppe i type 1-tjekket.

PYHA så kom vi igennem *findLegalMoves*-klassen. Jeg har puttet en del kommentarer ind i klassen, så jeg håber den er forståelig (Jævnfør forsiden – forstå det uforståelige).

10. AI-klassen

Det var egentlig slet ikke meningen, at jeg ville skrive ret meget kode til AI-klassen. Men så kom jeg til at købe denne her vildt lækre mus og utroligt dejlige håndledsstøtte til mit amazing tastatur:

Og så jeg kom til at skrive lidt mere.

I AI-klassen sker følgende:

Der er ingen felter eller konstruktør. Kun metoder.

Den eneste public metode vi møder, er *thinkHard.* Den bliver som parameter fodret med listen af mulige træk, *legalMoves.* Metoden bliver kaldt hver runde fra Main. Det husker vi tydeligt fra side 6.

Det første der sker i *thinkHard* er et tjek, om listen med mulige træk er tom! Det kan nemlig have flere forklaringer:

- Vi har tabt
- Vi har vundet

Vi tjekker først om vi har vundet, ved at kalde hjælpemetoden *checkIfWon.* Den ser sådan her ud:

```
192     private boolean checkIfWon() {
193         for (int i = 0; i < 6; i++) {        // Loop through the columns
194             if (Table.unseen[i] > 0) {       // Check if there are any columns, that still have unseen cards.
195                 return false;                // If so, the game is not yet finished.
196             }
197         }
198         return true;                         // But if all unseen cards have been exposed, the game is solved.
199     }
```

Metoden tjekker bare, om der er flere usete kort på bordet med vores array *unseen*[]. Hvis for-løkken finder et uset kort, er spillet ikke vundet endnu, og returnerer falsk med det samme. Ellers returneres true.

Hvis metoden returnerer true, får vi en lille vinder-besked, *Table.gameOn* sættes til FALSE for at standse spillet, og der returneres til Main. Det ser sådan her ud:

```
19              if (checkIfWon()) {                          // Check if win-condition is true (All unseen cards exposed)
20                  System.out.println("Du har vundet! KABALEN GÅR OP!");
21                  Table.gameOn = false;
22                  return;
23              }
```

Ellers går vi videre til næste tjek. Det er straks værre. Hvis der ikke er flere lovlige træk, og der ikke er flere kort i trækbunken, og vi ikke må blande bunken igen, har vi tabt. Det ser sådan her ud:

```
25          if (Table.cardsLeftInDrawPile == 0 && Table.shuffles == 0) {        // If draw pile is empty and no more shuffles allowed, the game must end
26              System.out.println("Der er ikke flere mulige træk og du må ikke blande bunken igen.\nSpillet er slut.\nTak for denne gang!");
27              Table.gameOn = false;
28              return;
29          }
```

Men det kan også være at vi GERNE må blande bunken igen:

```
31          if (Table.cardsLeftInDrawPile == 0) {        // If draw pile is empty, but player may still shuffle
32              System.out.println("Ikke flere træk! Bland trækbunken og vend et nyt kort");
33              Table.shuffles--;                        // Draw pile is shuffled. Deduct one from allowed shuffles
34              Table.position.get(11).clear();          // Empty draw pile
35              Table.position.get(11).add("XX");        // Add an XX to indicate empty
36              Table.cardsLeftInDrawPile = 24 - Table.cardsRemovedFromDrawPile;       // Reinitialize drawpile to original amount (24) minus cards removed
37              if (Table.debugText) {
38                  System.out.println("Cards left in draw pile: " + Table.cardsLeftInDrawPile + " Cards removed from draw pile:" + Table.cardsRemovedFromDrawPile);
39              }
40              promptUser();
41              return;
42          }
```

Så bliver der trukket 1 fra tilbageværende shuffles (linie 33), Vi tømmer trækbunken (indeks 11) og husker at genindsætte en ”XX”. (Linie 34 og 35)

Så genopfylder vi *cardsLeftInDrawPile* og husker at trække det antal fra, der var fjernet fra bunken i løbet af spillet indtil nu med *cardsRemovedFromDrawpile.* (linie 36).
Og så returnerer vi til Main.

Vi bemærker lige kaldet til *promptUser.* Det er den funktion, som pauser spillet indtil:

- Spilleren har læst på skærmen hvad han[1] skal.
- Spilleren har foretaget trækket
- Spilleren skal på eget initiativ vende kort om, der kan vendes
- OpenCV har taget et frisk billede af kabalens nye udseende

Når dette er sket, kan spilleren trykke sig videre. (Lige nu foregår det med 'f' og 'enter'.) Der er også en mulighed for at stoppe spillet med 'o'. (Faktisk fortsætter spillet bare spilleren trykker alt andet end 'o', men det var nemmere bare at bede spilleren om at trykke 'f'.)

Funktionen ser således ud:

```
183     private void promptUser() {            // After AI has selected a move, the game pauses until the player has made the move.
184         System.out.println("\nTast 'f' og tryk enter, når du har foretaget trækket.\nTast 'o' for at opgive spillet.");
185         Scanner myScanner = new Scanner(System.in);
186         String input = myScanner.nextLine();
187         if (input.equals("o")) {
188             Table.gameOn = false;
189         }
190     }
```

I linie 54 sker der nu noget vigtigt:

```
54              Move selectedMove = pickBestMove(legalMoves);   // This method is where AI-team implempents stuff
```

Metoden *pickBestMove* kaldes. Som parameter får den hele listen med lovlige træk.
Metoden returnerer et enkelt *move* som bliver gemt i variablen *selectedMove.*

Det er meningen, at denne metode på sigt skal prøve at finde nogle bedre træk end blot grovsortering i forhold til type 1, 2, 3 eller 4. Lige nu ser metoden således ud:

```
201 @       private Move pickBestMove(ArrayList<Move> legalMoves) {
202             return legalMoves.get(0);           // AI uncritically selects first move from legalMoves
203         }
```

Lige nu vælger metoden altså bare det første move i listen over tilladte træk med get(0). Herefter returneres dette træk. Egentlig tror jeg som nævnt, at mange kabaler vil gå op med denne grovsortering. Men det er ikke så sexet.

Allright, nu begynder det at blive lidt langhåret igen.

Nu tjekker vi, om det valgte move var et type 1 træk. Det gør vi i linie 57:

```
57          if (selectedMove.getType() == 1) {          // If move is type 1: Move top card from column or drawpile to a foundation
```

[1] Der er ingen grund til at skrive "han/hun". Vi kan nøjes med "han". Vi går på IT/elektronik.

Det betyder at spilleren nu tager et kort og lægger det over i en byggebunke.

Vi resetter først den counter, der holder øje med om vi har lavet for mange søjle-til-søjle-træk:

```
58                  Table.columnToColumn = 0;       // Reset counter
```

Så skriver vi til spilleren:

```
59                  System.out.println("Du kan flytte et kort til en byggebunke!");
```

Så undersøger vi, om kortet bliver taget fra trækbunken (og ikke fra en søjle):

```
60              if (selectedMove.getFromPosition() == 11) {         // If card comes from draw pile
61                  System.out.println("Flyt " + selectedMove.getPlainText() + " fra trækbunken til en byggebunke.");
```

Bemærk at vi her dykker ned i *move*-objektet og trækker klartekst-udgaven frem med getteren *selectedMove.getPlainText()* som er en metode fra *Move.*klassen. Det er også i move-objektet, at vi ser, at kortet flyttes fra trækbunken, med kald til *getFromPosition().*

Så skal vi huske, at nu fjerner spilleren permanent et kort fra trækbunken:

```
62                  Table.cardsRemovedFromDrawPile++;               // A card is now permanently removed from draw pile
```

Vi skal også undersøge, om kortet var det sidste synlige kort i trækbunken. Så vil spilleren nemlig på eget initiativ vende et nyt kort:

```
63              if (Table.position.get(11).size() == 1) {       // If no visible cards in draw pile, one is turned over by player
64                  Table.cardsLeftInDrawPile--;
65              }
```

Hvis ikke der var tale om et træk fra trækbunken. jamen så var det et træk fra en søjle. Så får spilleren denne her besked:

```
69          } else {
70              System.out.println("Flyt " + selectedMove.getPlainText() + " fra søjle " + (selectedMove.getFromPosition() + 1) + " til en byggebunke.");
```

Uanset hvor kortet kom fra, skal det fjernes fra bordpladen. Det kræver, at vi først konstaterer hvor mange kort, der er i søjlen/trækbunken og derefter fjerner det:

```
72          int sizeColumn = Table.position.get(selectedMove.getFromPosition()).size();             // Get size of column that the card is moved from
73          Table.position.get(selectedMove.getFromPosition()).remove( index: sizeColumn - 1);      // Remove card from Table
```

Nu skal vi undersøge, om kortet var det sidste kort i en given søjle og om der er usete kort, der nu dukker frem. Hvis det er tilfældet, fjerner vi et kort i "regnskabet" over usete kort i søjlen:

```
75          if (Table.position.get(selectedMove.getFromPosition()).size() == 1 && Table.unseen[selectedMove.getFromPosition()] > 0) { // If column now empty
76              Table.unseen[selectedMove.getFromPosition()]--; // Remove an unseen card from column, if any unseen cards remain.
77          }
```

Nu skal vi så have tilføjet det fjernede kort til den rigtige byggebunke:

```
79              Table.position.get(selectedMove.getToPosition()).add(selectedMove.getCard());          // Add card to foundation
```

Og så skal vi have opdateret den String, der husker på det senest flyttede kort:

```
80              Table.justMoved = selectedMove.getCard();                                           // Remember last moved card
```

Til sidst prompter vi spilleren og returnerer til Main efter han har foretaget sit træk, OpenCV har taget nyt billede, osv:

```
81              promptUser();
82              return;
```

De samme procedurer og overvejelser gør sig i store træk også gældende ved type 2, 3 og type 4-træk. Det tager nok lidt tid at tygge sig igennem koden.

Det bliver en lille smule mere skummelt ved type 3-træk, hvor vi ofte skal flytte en hel stak kort ad gangen. Her skal vi altså både slette en stak kort og tilføje den sammen stak kort til en anden søjle. Det ser sådan her ud når vi tilføjer en stak kort til en søjle:

```
137         // Add the card group to target column
138         for (int i = selectedMove.getCut(); i < sizeColumnFrom; i++) {                      // Iterate subgroup of cards to be moved
139             Table.position.get(selectedMove.getToPosition()).add(Table.position.get(selectedMove.getFromPosition()).get(i)); // Copy cards into new spot
```

Bemærk, at vi looper igennem kortene fra **cuttet** til **size af søjlen.** For hvert enkelt kort, bliver kortet tilføjet modtagersøjlen med add().

Og så skal korterne også slettes fra from-søjlen:

```
145         // Delete card group from source column
146         for (int i = selectedMove.getCut(); i < sizeColumnFrom; i++) {                      // Iterate subgroup of cards and remove them from Table
147             Table.position.get(selectedMove.getFromPosition()).remove(selectedMove.getCut());
```

Bemærk, at vi sletter samme position gentagne gange med *remove(selectedMove.getCut()).* Hvis vi eksempelvis skal slette 5 kort, og cut-variablen er 3, bliver indeks 3 slettet 5 gange i træk. Det er fordi, at når vi sletter et kort med *remove,* vil de resterende kort rykke en tand op. Næste *remove* vil derfor ramme et nyt kort.

Eksempel:

Udgangspunkt:

XX
0D
9C
8H <- cut 3
7C
6H
5S
4H

Vi fjerner hjerter 8:

XX
0D
9C
7C <- cut 3
6H
5S
4H

Nu fjerner vi klør 7:

XX
0D
9C
6H <- cut 3
5S
4H

Og så videre. Efter 5 remove-kommandoer, på samme cut-index, vil søjlen være fjernet fra cuttet og nedefter.

Og det var så det. Det vil nok tage lidt tid at få en fuldstændig lækker forståelse af koden. Det er altid superirriterende at skulle sætte sig ind i andres spaghetti.

11. Test

Jeg har prøvet at køre del tests af programmet. Jeg har ikke fået testet helt til bunds endnu, for det er superbesværligt. Så længe OpenCV ikke er oppe at køre, foregår test af programmet således:

- En 7-kabale startes op ved siden af programmet, enten med rigtige kort på et bord eller på computeren i et nyt vindue
- tableFile opdateres med de 7 søjler og første kort i trækbunken
- Programmet startes
- Det præsenterede træk foretages på bordet/skærmen
- tableFile opdateres igen. Man skal huske at opdatere både søjlen, som man har taget kort fra, og søjlen, som man har lagt over på. Fuldstændigt som OpenCV ville gøre det.
- Programmet tager en runde mere ved at trykke 'f'+enter.
- osv
- osv
- osv

Laver man den mindste fejl i tableFile er man fucked. Det kan ikke reddes.

Jeg har testet alle mulige træktyper og det virker fint indtil videre. Jeg er endnu ikke kommet så langt som til at se om win/loose check virker.

Der er sikkert flere bugs (blinkesmiley) som mangler at dukke op til overfladen.

12. Overvejelser

Det var slet ikke meningen at jeg ville skrive så meget kode, som det endte med. Jeg er helt med på, at det er fuldstændig for egen regning og hvis nogen i gruppen ikke er tilfredse, skal det selvfølgelig skrottes eller ændres.

Jeg håber at manualen her giver et godt udgangspunkt for den enkelte til at vurdere, om koden er OK, skal ændres, dræbes, eller andet.

Det bliver selvfølgelig ubetinget sådan, at vælger vi at bruge koden, fordeler vi den ligeligt imellem os, så vi er ligeligt repræsenteret i rapporten. Der er 6 klasser at vælge imellem, så der er vist nok til alle :-D