

# Forelæsning Uge 9

- **Arrays**

- Objektsamlinger med et fast (på forhånd kendt) antal elementer
- Velkendt fra mange andre programmeringssprog

- **Brug af Java uden BlueJ**

- Start af Java fra konsolvindue

- **Principper for design af klasser**

- Undgå f.eks. at have den samme kode stående flere steder

- **Mundtlig præsentation**

- Kan som alt andet trænes
- De næste uger vil vi gøre det systematisk
- Vi vil fokusere på mundtlig eksamen, men det, som I lærer, vil også være nyttigt i mange andre situationer

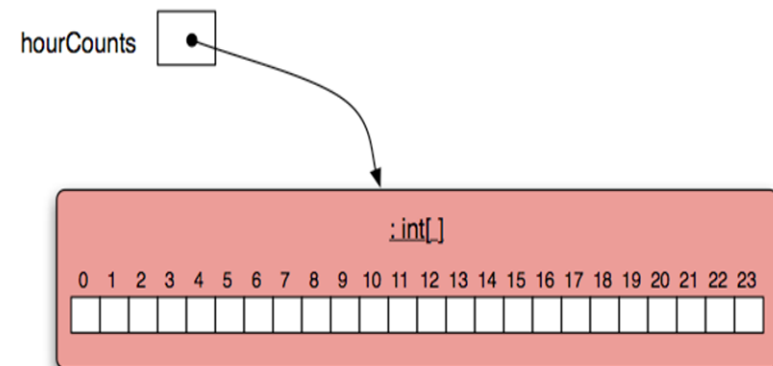
- **Afleveringsopgave: Dronninger (Queens)**

Tillykke – køreprøven blev klaret flot

- 83% afleverede fuld besvarelse
- 94% klarede mindst 4 tjekpunkter

Brug diskussionsforummet til at rapportere de fejl, som I finder – så de kan blive rettet

- Forelæsningslides
- Opgaveformuleringer
- Testserveren (mere end 50.000 linjers kode)



# ● Arrays

- **Arrays har et fast (på forhånd kendt) antal elementer**
  - Ligner Collections, men er bygget direkte ind i Java sproget med egen specialtilpasset syntax

Skal ikke importeres

## 1. Erklæring (som feltvariabler)

```
private String[] texts;  
private int[] hourCounts;
```

De firkantede parenteser angiver, at det er et array

Kan også bruges på primitive typer

## 2. Initialisering (ofte i konstruktør)

```
hourCounts = new int[24];
```

Udtrykket i [...] skal evaluere til et ikke-negativt heltal, som angiver størrelsen

Arrayets index'er nummereres fra 0 til 23

## Eksempler på brug

```
count = hourCounts[hour];  
hourCounts[hour] = value;  
hourCounts[hour]++;
```

Udtrykkene i [...] skal evaluere til et heltal i intervallet [0,23] (ellers runtime fejl)

Antal elementer i {...} bestemmer størrelsen

- **Oprettelse og initialisering med værdier kan slås sammen**

```
private int[] monthLength = new int[] {31,28,31,30,...,30,31};
```

# Løkker brugt på arrays

- Vi kan gennemløbe et array ved hjælp af en for løkke

Index i arrayet

- Længden af arrayet
- Variabel (ingen parenteser bagefter)
- Arrayets index'er løber fra 0 til length-1

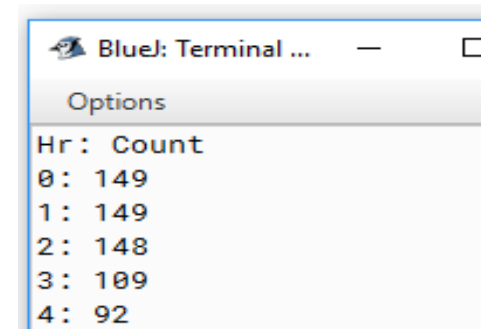
```
for(int hour = 0; hour < hourCounts.length; hour++) {  
    System.out.println( hour + ": " + hourCounts[hour] );  
}
```

Opslag i arrayet

- Vi kan også bruge en for-each løkke

```
for(int count : hourCounts) {  
    System.out.println(count);  
}
```

- Men så har vi ikke et index og kan ikke udskrive timetallene (med mindre vi laver vores egen tæller inde i for-each løkken)



```
BlueJ: Terminal ...  
Options  
Hr: Count  
0: 149  
1: 149  
2: 148  
3: 109  
4: 92
```

# Forskelle på arrays og arraylister

---

- Arrays har et fast (på forhånd kendt) antal elementer
- Kan anvendes på primitive typer (uden brug af wrapper klasse)
- Simplere syntax

Erklæring

```
private int[] hourCounts;  
private ArrayList<Integer> hourCounts;
```

Initialisering

```
hourCounts = new int[24];  
hourCounts = new ArrayList<>();
```

Kald

```
hourCounts[13];  
hourCounts.get(13);
```

- Velegnet til at håndtere flerdimensionele strukturer

```
minuteCounts = new int[24][60];  
secondCounts = new int[24][60][60];
```

- Indbygget i Java sproget

- Giver simplere syntax som til gengæld afviger fra den, vi kender fra andre objektsamlinger
- Kendt fra mange andre programmeringssprog

- Ikke en del af Collection frameworket

- Men man kan alligevel bruge for-each løkker på dem
- Array objekter har ikke metoder (i stedet kan man bruge klassemetoder fra forskellige klasser bl.a. java.util.Arrays)

# Eksempler på brug af arrays

---

- **En arrayliste er en liste af objekter implementeret ved hjælp af et array**
  - Der er ubrugte elementer i arrayet, således at man kan indsætte nye elementer i arraylisten
  - Når alle elementer er brugt, udskiftes arrayet med et nyt (og større)
- **En tekststreng (objekt af typen String) er en liste af tegn implementeret ved hjælp af et array af bytes**
  - Hver byte angiver et tegn
  - F.eks. er 'a'  $\approx$  97, 'b'  $\approx$  98, 'c'  $\approx$  99, 'd'  $\approx$  100, 'e'  $\approx$  101 osv.
  - Her er der ikke ubrugte elementer
  - Man har ikke behov for at kunne tilføje flere tegn, idet String objekter er immutable (ikke kan ændres, når de først er skabt)

# Adressebog med personer

addressB1:  
AddressBook

addressB1 : AddressBook

private ArrayList<Person> persons

Inspect

Get

Show static fields

Close

[0] : Person

private String name "Cecilie" Inspect

private int age 18 Get

Show static fields

- String objektet er implementeret via et **byte** array
- Lidt sært, at det ikke er et **char** array

name : String

private byte[] value Inspect

private byte coder 0 Get

private int hash 0

Show static fields

Close

persons : ArrayList<Person>

Object[] elementData

private int size 5

protected int modCount 5

Show static fields

Arraylisten er implementeret via et Object array

Array'et har 10 elementer, hvoraf de første fem pt er i brug

elementData : Object[]

int length 10

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null

Show static fields

Close

value : byte[]

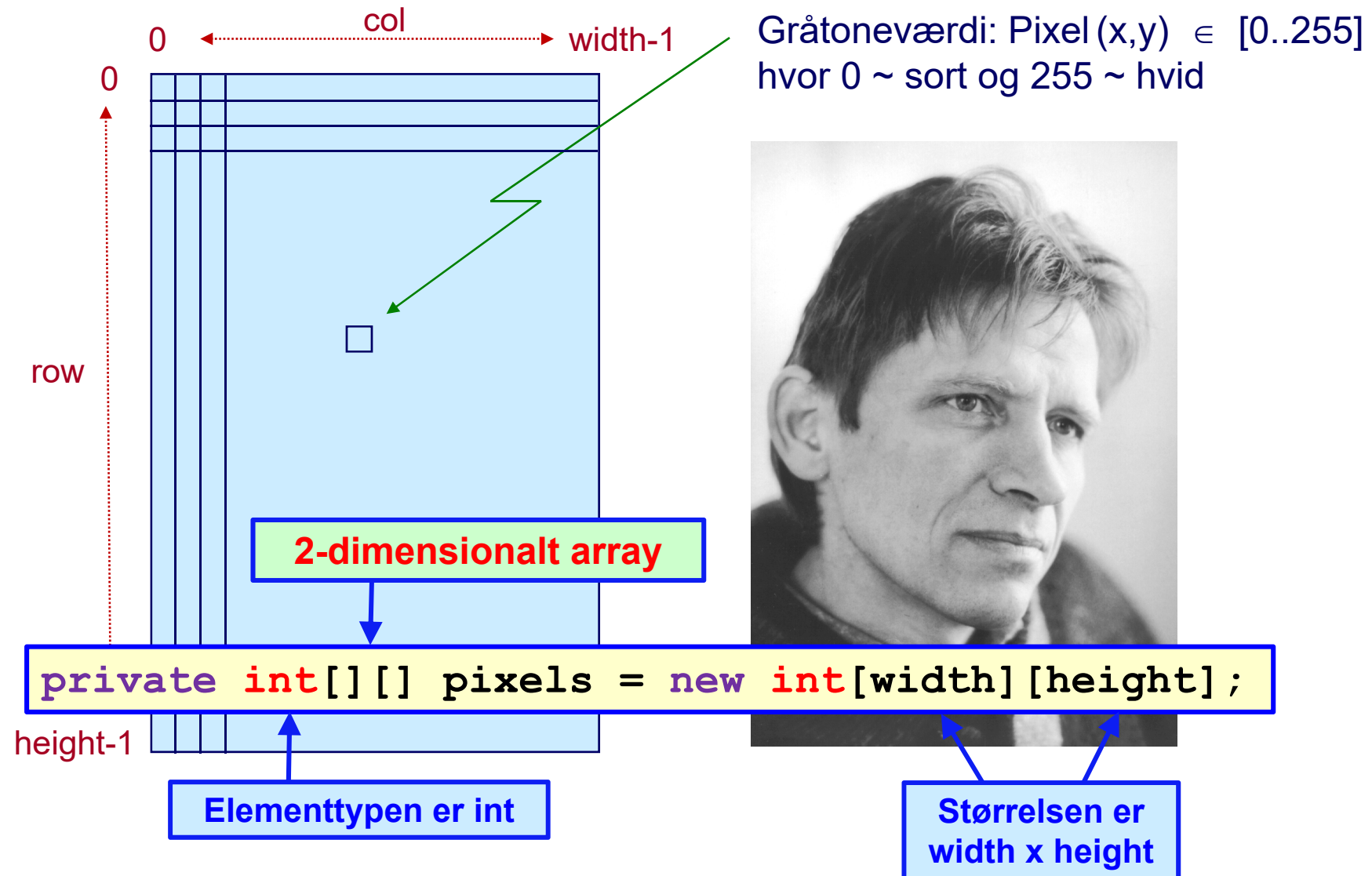
int length 7

[0]	67	C
[1]	101	e
[2]	99	c
[3]	105	i
[4]	108	l
[5]	105	i
[6]	101	e

Show static fields

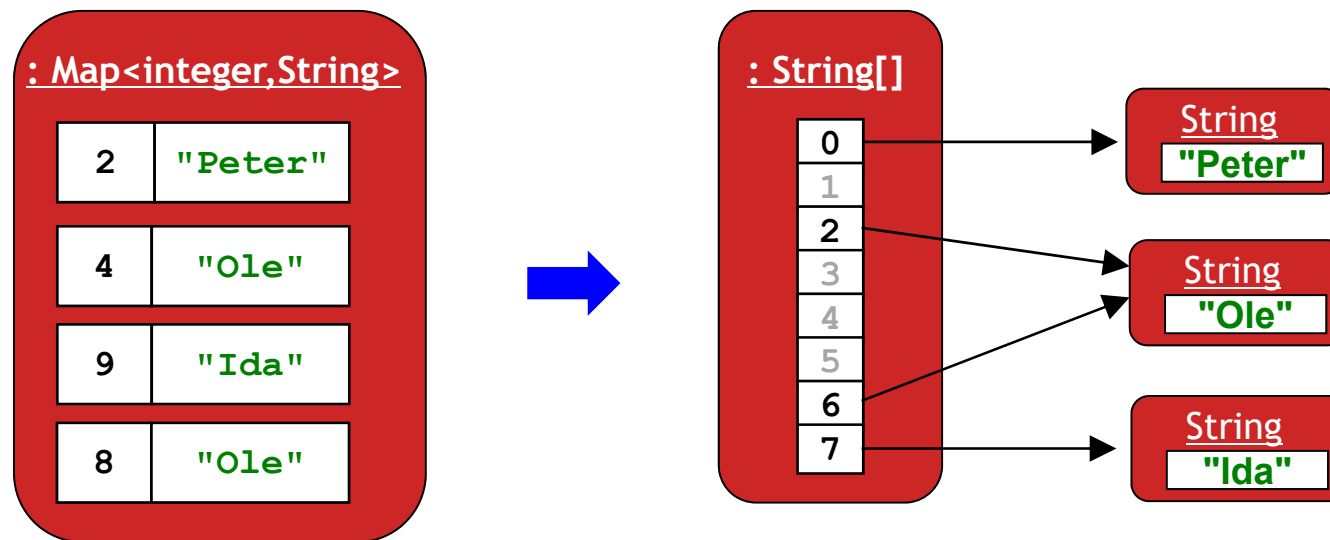
Close

# Billedrepræsentation via arrays



# Afbildning via arrays

- En afbildning `Map<Integer, V>` kan erstattes af et array `V[ ]`
  - Forudsætter at vi på forhånd ved, hvilket interval af heltal nøglerne befinder sig i
  - At der ikke er for store "huller" imellem nøglerne
  - Hvis de brugte nøgler ligger i intervallet `[min, max]` repræsenteres afbildningen ved et array `V[max-min+1]`, og nøgler konverteres til indices ved at subtrahere min



De fire grå array indgange er ubrugte, dvs. lig null



# Nyttige metoder

---

- **Konvertering fra lister til arrays**
  - ArrayList og andre List klasser indeholder metoden **toArray**, der konverterer listen til et array
- **Klassen java.util.Arrays indeholder nogle nyttige klassesmetoder til manipulation af arrays, heriblandt**
  - **stream** returnerer en Stream med elementerne i et array
  - **equals** tester om to arrays er ens (samme elementer i samme rækkefølge)
  - **toString** tekstrepræsentation af et array og dets elementer  $[e_0, e_1, \dots, e_{\text{last}}]$
  - **fill** opdaterer alle elementer i et array til en specificeret værdi
  - **copyOf** kopierer arrayet og ændrer længden
  - **sort** sortering af elementerne i et array
  - **binarySearch** søgning i et array
  - **asList** returnerer en (fixed-size) List implementeret via et array

```
List<T> asList(T... a)
```

Metoden har et **variabelt** antal parametre, som alle er af typen T

```
List<String> names = Arrays.asList("Peter", "Anna", "Sofus", "Ida");
```

# Polymorfi og mangel på samme

- **Arrays klassen har næsten 200 metoder**
  - Det store antal skyldes, at de fleste metoder findes i **9 versioner** (en for hver af de 8 primitive typer og en der dækker alle objekt typer)

<code>static String</code>	Dækker arrays, hvor elementtypen er <b>boolean</b>	<code>toString(boolean[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>byte</b>	<code>toString(byte[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>char</b>	<code>toString(char[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>double</b>	<code>toString(double[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>float</b>	<code>toString(float[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>int</b>	<code>toString(int[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>long</b>	<code>toString(long[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker alle objekt typer, idet parameteren <b>a</b> er <b>polymorf</b>	<code>toString(Object[] a)</code> Returns a string representation of the contents of the specified array.
<code>static String</code>	Dækker arrays, hvor elementtypen er <b>short</b>	<code>toString(short[] a)</code> Returns a string representation of the contents of the specified array.

- Hvis man havde krævet, at arrays kun kunne bruges på objekt typer (som Collections), havde vi kunnet nøjes med én metode

# Mini-quiz om arraylister og arrays

---

Hvor mange af nedenstående erklæringer er ulovlige?



`private ArrayList<int> hourCounts;` ArrayList kan kun bruges på objekt typer



`private ArrayList<String> texts;`



`private ArrayList<Integer> hourCounts;`



`private ArrayList<Integer><Integer> minCounts;`



`private boolean[] bits;`

Skal i stedet skrives

ArrayList<ArrayList<Integer>>



`private String[] texts;`



`private Pixel[][][] pixels;`



`private Integer[][] hourCounts;` Parenteserne kan ikke være inde i hinanden

# ● Brug af Java uden BlueJ

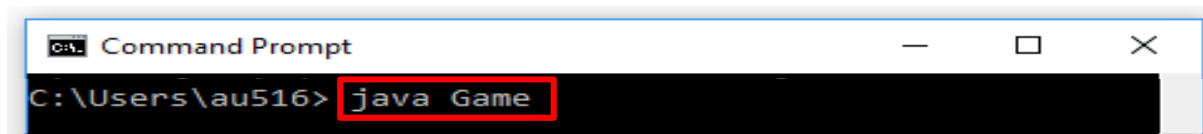
---

- Java projekter gemmes som et antal filer i en separat folder (directory)
- For hver klasse, f.eks. **Game**, vil der være følgende filer
  - **Game.java** indeholder koden for klassen (som I skriver den og ser den i editoren)
  - **Game.class** indeholder den oversatte binær kode (som kan udføres af Java's virtuelle maskine)
  - **Game.html** indeholder dokumentationen (som vises, når I vælger Documentaion view i editoren; filen ligger i en subfolder **doc**)
  - **Game.ctxt** fil indeholder noget ekstra information om klassens kommentarer og dokumentation (findes kun i BlueJ og nogle få andre udviklingsomgivelser)
- I **BlueJ** generes de tre sidste filer **automatisk** ud fra den første
  - De opdateres automatisk, når klassen genoversættes (eller man skifter til Documentation view)
  - Så det er ikke noget, som I behøver at tænke på

# Start og oversættelse af Java kode

---

- Det er muligt at **starte Java projekter fra et konsolvindue (uden brug af BlueJ)**
  - Når man gør det, skal man angive den klasse, hvori programudførelsen skal starte, f.eks. Game klassen



- **To ting være opfyldt**
  - Der skal være lavet en sti (path) der angiver placeringen af Java's virtuelle maskine (hvordan dette gøres afhænger af operativsystemet)
  - Konsollens aktive folder skal indeholde en fil **Game.class** med den **oversatte kode** for en Java klasse, hvori der findes en metode ved navn **main** (som opfylder nogle ting, der beskrives på næste slide)
- **Oversættelse af Game klassen (og de klasser, som den bruger) sker ved at skrive javac Game.java i konsollen**
  - Se [www.dummies.com/programming/java/how-to-use-the-javac-command/](http://www.dummies.com/programming/java/how-to-use-the-javac-command/) [Link](#)

# main metoden

---

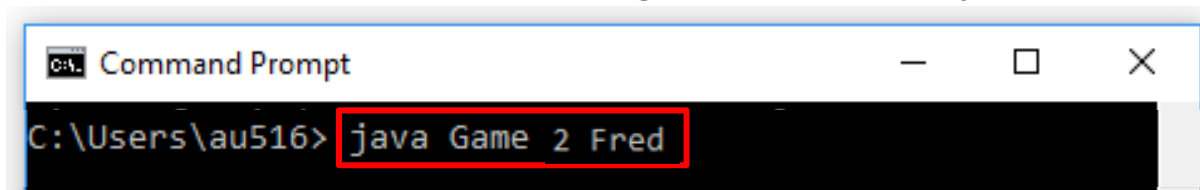
- **main metoden skal have nedentående udseende**

```
public static void main(String[] args) {...}
```

- Den skal være **public**, således den kan kaldes uden for klassen
- Den skal være en **klassemetode**, da der på kaldstidspunktet ikke eksisterer nogen objekter
- Den skal have et **String array** som parameter (denne konvention er, sammen med en masse af Java's syntax, arvet fra programmeringssproget C)

- **Argumentet til main metoden skrives efter klassens navn**

- I nedenstående eksempel er argumentet et array med elementerne "2" og "Fred"



- Hvis man ikke skriver noget efter klassens navn, er argumentet et tomt String array

# BlueJ kontra andre Java editorer

---

- **BlueJ gør det let at håndtere Java projekter uden brug af konsol og kendskab til diverse fil extensions (og main metoden)**
  - Den er derfor særdeles velegnet for nye programmører
  - Man kan let "lege" med programmer og udføre deres metoder uden at kende deres detaljerede indhold (Java kode)
- **Andre Java editorer er mere komplekse at bruge, men stiller til gengæld flere faciliteter til rådighed**
  - De kan f.eks. automatisk generere kode for konstruktører, accessormetoder og import statements (det I ikke måtte bruge under køreprøverne)
- **Mange af vores studerende bruger IntelliJ udviklingsmiljøet**
  - Det er frivilligt om I vil skifte til IntelliJ
  - Hvis I er tilfredse med BlueJ, kan I sagtens fortsætte med den kurset ud
- **Mere information om start af Java fra konsol samt brug af IntelliJ**
  - To videoer under uge 9: **Java uden BlueJ** og **Brug af IntelliJ**
  - Brightspace siden **Brug af IntelliJ** under **Afleveringsopgaver**
  - Appendix E i BlueJ bogen

# ● Principper for design af klasser

---

- **Software er ikke noget, der laves på kort tid for derefter at forblive uændret i al sin levetid**
  - Software vedligeholdes (rettes, udvides, tilpasses, porteres, ...)
  - Mange forskellige mennesker er involveret med en tidsmæssig udstrækning på flere årtier
  - Software overlever kun, hvis det kan vedligeholdes – ellers har det kort levetid (og dårlig økonomi)
- **Seks principper for design af klasser – så de bliver lette(re) at læse, vedligeholde og genbruge**
  - Undgå kodedublering (code duplication)
  - Løs kobling mellem klasserne (loose coupling)
  - Sammenhængende klasser og metoder (cohesion)
  - Responsibility-driven design
  - Tænk fremad (think ahead)
  - Regelmæssig omstrukturering (refactoring)



# Princip 1: Undgå dublering af kode

---

- **Vi vil gerne undgå at samme kode (sekvens af sætninger) forekommer flere steder**
  - Sparer tid både for dem, der skriver koden, og for dem, der skal læse og forstå koden
  - Gør koden meget lettere at vedligeholde, idet man kun skal rette ét sted
  - Man undgår inkonsistente versioner, hvor en rettelse er foretaget nogle steder, men glemt andre steder
- **Fire vigtige midler til at undgå kodedublering**
  - Indpakning i **løkke** (hvor kroppen udføres mange gange)
  - Indpakning i **hjælpemetode** (private or public)
  - God **parametrisering** af metoder, så man kan nøjes med én metode, i stedet for at have flere, der ligner hinanden
  - I kapitel 10 skal vi se, at **subklasser** (med nedarvning) er fortrinlige til at undgå dublering af feltvariabler og deres tilhørende accessor og mutator metoder

# Princip 2: Løs kobling mellem klasserne

---

- **Klasser interagerer med hinanden via deres metoder og konstruktører**
  - Vi vil gerne kunne ændre implementationen af en klasse, uden at skulle ændre implementationen af alle de klasser, der bruger den
  - Klasser der opfylder dette siges at være **løst koblede**
- **Feltvariabler skal være private, således at de kan ændres uden at genere andre klasser**
  - Når feltvariabler er private, kan andre klasser kun tilgå dem via deres accessor og mutator metoder
  - Disse metoder er lette at lokalisere og modificere, når implementationen ændres, idet de befinder sig i samme klasse som feltvariablerne
  - Det er vigtigt, at signaturene for metoder og konstruktører er så velvalgte og generelle, at man ikke behøver at ændre disse undervejs
- **På et BlueJ klassediagram er mængden af pile en god indikation for koblingsgraden mellem klasserne**
  - Jo færre pile jo bedre

# Princip 3: Sammenhængende (cohesion)

---

- **Hver klasse skal være en sammenhængende enhed**
  - Det betyder, at klassen skal håndtere ét problemkompleks, og alle de data og metoder, der hører til denne
  - Hvis en klasse håndterer to eller flere forskellige problemkomplekser, kan klassen med fordel opdeles i to eller flere klasser
  - Opdelingen gør det nemmere for brugeren at finde de rigtige metoder, og nemmere for programmøren at lokalisere de steder, som skal ændres, når der laves en opdatering
- **Metoder skal også være sammenhængende**
  - En metode, der konstruerer en tekststreng, og udskriver den på terminalen, kan med fordel ændres til at returnere den konstruerede tekststreng
  - Det er så op til klassen, der kalder metoden, at beslutte, hvad tekststrengen skal bruges til. Den kan udskrives, lægges ind i en objektsamling (collection), analyseres eller noget helt fjerde
  - Metoder, der kun gør én ting, har langt større sandsynlighed for at kunne bruges i mange forskellige sammenhænge

# Princip 4: Responsibility-driven design

---

- **Hver klasse bør være ansvarlig for håndtering af egne data**
  - Klassen bør f.eks. have en (eller flere) metoder der returnerer en tekststreng, som beskriver objektets tilstand (eller dele heraf)
  - Andre klasser kunne selv gøre det – ved at få den nødvendige information via accessor metoder og stykke den sammen til en tekststreng
  - Men det ville betyde, at disse klasser skal tilpasses, hver gang der tilføjes/fjernes feltvariabler
  - Når de metoder, der skal ændres, er i samme klasse som feltvariablerne, er det langt mere sandsynligt, at man husker at ændre dem, og det er langt hurtigere at lokalisere de steder, hvor ændringerne skal foretages

# Princip 5: Tænk fremad

---

- **Prøv at forudse fremtidige ændringer/tilføjelser**
  - Eksempel: Når man designer de første versioner af et computerspil, vil det være nærliggende, at interaktionen med brugere på et tidspunkt skal ske via et grafisk interface i stedet for tekstbaseret input/output via terminalen
  - Hvis man fra start forsøger at samle de ting, der har med brugerinteraktionen at gøre, i en enkelt klasse, bliver det sidenhen lettere at ændre interaktion fra at være tekstbaseret til at være grafisk
  - Hvis man bruger lidt tid på at forudse mulige fremtidige ændringer/tilføjelser – og forberede disse – kan man senere spare enorme mængder af tid og kræfter

# Princip 6: Regelmæssig omstrukturering (refactoring)

---

- **Når man er i gang med et større (eller mindre) projekt får man fra tid til anden behov for at omstrukturere sin kode**
  - Man kan f.eks. ønske at opdele en klasse, der er blevet stor og gør mange forskellige ting, i to eller flere klasser, som er nemmere at overskue
  - Begreber, der indtil nu er repræsenteret ved hjælp af en simpel tekststreng, vil man måske fremover modellere via en dedikeret klasse med feltvariabler, der kan give en mere fyldestgørende beskrivelse af objekternes tilstand (jvf. **Track** klassen fra afsnit 4.11 i BlueJ bogen)
  - Omstrukturering af software kaldes også **refaktoring** (refactoring)
- **Principper for refaktoring**
  - Start med at lave den ønskede omstrukturering **uden** at tilføje ny funktionalitet
  - Test at det omstrukturerede program opfører sig som det gamle (vi vender tilbage til testteknikker i næste forelæsning)
  - Først når man er overbevist om, at det omstrukturerede program er korrekt (dvs. opfører sig som det gamle) tilføjes ny funktionalitet
  - Ved at iagttage denne tidsmæssige opdeling i refaktoring og tilføjelse af ny funktionalitet kan man spare masser af tid og kræfter

# De fem C'er – for godt design af klasser

---

- **Cohesion (sammenhængende)**
  - Klassen håndterer **ét problemkompleks**
  - **Turtle** klassen kan producere 2D stregtegninger (og intet andet)
  - move og turn er separâte metoder – i stedet for **moveAndTurn**
- **Completeness (komplet)**
  - Der er metoder til **alt** det, der er nødvendigt indenfor problemkomplekset
  - Når pennen har en farve, bør der være metoder til at ændre/aflæse denne
- **Convenience (bekvem)**
  - Almindelige ting kan gøres **let** og **hurtigt**
  - Turtle klassen bliver mere bekvem efter tilføjelsen af **jump** og **jumpTo**
  - Når man skal tegne cirkler, er **circle** mere bekvem end **polygon**
- **Clarity (transparens)**
  - Metoderne skal gøre det, man **forventer** af dem – og kun det
  - Det vil være forvirrende, hvis **jump** og **jumpTo** også drejer skildpadden eller skifter pennens farve
- **Consistency (konsistent)**
  - Metoderne skal fungere på **"samme måde"**
  - Turtle klassen angiver alle afstande og vinkler som en **double**

**Pause**

# ● Mundtlig præsentation

---

- **Det er vigtigt for it-folk at kunne præsentere tekniske problemstillinger for fagfæller og lægfolk**
  - Det er en essentiel del af vores faglige kompetencer, og I kommer alle til at gøre det i jeres daglige arbejde
  - Mundtlig eksamen tester, at I kan jeres stof, og at I er i stand til at fremlægge det for andre (eksaminator og censor)
  - Det sidste er slet ikke så let, som det lyder
  - Mange eksaminander kan stoffet, men er dårlige til at præsentere det, hvilket medfører en for lav karakter i forhold til deres reelle viden



# Mundtlig eksamen

---

- **Ved den mundtlige eksamen forventer vi, at du demonstrerer**
  - Kendskab til de **vigtigste begreber** inden for det trukne emneområde
  - Evne til at **programmere i Java** ved at præsentere og forklare små velvalgte programstumper indenfor emneområdet
  - Evne til at **svare på spørgsmål** inden for emneområdet, herunder relatere kursets afleveringsopgaver til emneområdet
- **Der gives karakter efter 12-skalaen**
  - Pointene fra køreprøven og computerspilsopgaven tæller med tilsammen 25% i fastlæggelsen af den endelige karakter for kurset
  - I praksis betyder det, at høje point kan trække en karakter op, mens lave point kan trække en karakter ned
  - Uanset pointtal kan man dumpe, hvis den mundtlige præstation er uacceptabel

# Forløbet af eksamen

---

- **Ved eksaminationens start trækkes et spørgsmål (ud af 9 mulige)**
  - Mens den foregående eksaminand er oppe, har du ca. 15 minutter til at **genopfriske** detaljerne i det trukne spørgsmål
  - Du kan **ikke** nå at lære tingene, hvis du ikke kan dem i forvejen
  - Under forberedelsen må du gerne kigge i noter, slides, lærebogen og andet materiale
- **Præsentationen (ca. 15 min)**
  - De første 2-3 minutter får du lov til at skrive din disposition på tavlen og snakke uforstyrret (indtil den værste nervøsitet har lagt sig)
  - Derefter vil eksaminator/censor afbryde med forskellige spørgsmål for at hjælpe dig med
    - at rette eventuelle småfejl / uklarheder
    - at få dækket de vigtigste ting indenfor emneområdet
  - Ved at stille spørgsmål tjekker vi også, om du har forstået stoffet eller blot lært det udenad
  - Jo bedre du har forberedt dig og jo mere initiativ du udviser – jo bedre har du styr på, hvor du "kommer hen" under eksamen (f.eks. hvilke programmeringseksempler, du skal gennemgå)
- **Votering mv**
  - De næste 3-5 minutter bruges til votering, meddelelse og forklaring af din karakter samt skift til næste eksaminand

# Træning gør mester

---

- **Evnen til at lave gode mundtlige præsentationer kan forbedres kraftigt ved intensiv træning**
  - Vi vil derfor i kursets anden halvdel bruge den anden af de to ugentlige øvelsesgange på systematisk træning i mundtlig præsentation
  - Det er **obligatorisk** at lave mindst 2 præsentationer af eksamensspørgsmål (som **godkendes** af instruktoren)
- **I begyndelsen er det svært, men efterhånden bliver det lettere**
  - Husk på hvor god, hurtig og sikker du blev til at programmere, da du trænede i ugerne op til køreprøven
  - Det samme vil ske med din evne til at lave en god mundtlig præsentation
  - Det vil hjælpe dig til eksamen – i dette og efterfølgende kurser
- **Nervøsitet**
  - Man kan træne sig op til at kunne håndtere nervøsitet – og det er en stor hjælp at vide, at man godt kan
  - Læs mere på disse tre websider:

# Organisering af træningen

---

- **I uge 10-15 bliver holdet ved ugens sidste øvelsesgang delt i to**
  - Den ene halvdel har øvelser på det normale øvelsestidspunkt
  - Den anden halvdel har øvelser på et andet tidspunkt (se "Vigtig meddelelse")
- **Ved hver øvelsesgang trænes 1-2 af de 9 eksamensspørgsmål**
  - Hvert spørgsmål præsenteres af 2-4 studenter (efter hinanden)
  - Instruktoren fungerer som eksaminator
  - Efter hver præsentation diskuteres, hvordan den kan forbedres
  - Der er 6 uger med 4-6 præsentationer på hvert af de 2 delhold, dvs. ca. 60 præsentationer og dermed 2-3 til hver student
- **En af de mest effektive måder at træne til eksamen, er at høre andre studerende (og lære af deres gode og dårlige ting)**
  - Vi opfordrer derfor **kraftigt** til, at I deltager i alle øvelsesgangene – også de gange, hvor I ikke selv skal præsentere
  - Vi opfordrer også til, at I under den rigtige eksamen går ind og hører nogle af jeres medstuderende
  - Nogle synes, at det er "upassende" – men faktisk vil det for langt de fleste eksaminander være betryggende, at der er "neutrale" tilhørere tilstede under eksaminationen
  - Man bliver mindre nervøs, når man har set, hvordan eksamen foregår

# Forberedelse til mundtlig eksamen

---

- **Disposition**

- For hvert af de 9 spørgsmål laves en kort velgennemtænkt disposition
- A4-ark med 10-20 ord (ingen figurer, formler, programstumper, eller lignende)
  - Opremser de begreber og eksempler, som du vil præsentere
- Til eksamen starter du med at skrive dispositionen op i et hjørne af tavlen
  - Dulmer ofte den værste nervøsitet
  - Herefter lægges dispositionen **helt væk** (eller med bagsiden opad)
- Du får ikke point for at kunne læse op af dispositionen
  - Derfor skal den være kort og præcis
  - Den er en **huskeliste** over, hvad du vil præsentere og i hvilken rækkefølge
  - Du bør kunne præsentere de ting, som du har udvalgt i dispositionen på ca. 10 min
  - Resten af tiden bruges til at svare på diverse spørgsmål fra eksaminator og censor

# Java kode eksempler

---

- **Valg af eksempler**

- Det er vigtigt, at du **på forhånd**, for hvert af de 9 spørgsmål, har valgt nogle gode eksempler på Java kode, som du vil præsentere
- Eksemplerne kan være "stjålet" fra lærebogen, mine slides eller nogle af de opgaver, som du har lavet på kurset
- Brug tid på at finde gode eksempler, og tid på at træne i at præsentere dem
- Eksemplerne skal være korte og tydeligt vise de ting, som er centrale for din præsentation

- **Hvis du ikke selv forbereder små velvalgte Java eksempler, finder vi nogle, som du skal præsentere**

- Det gør ikke opgaven lettere

- **Det er tilladt at lave sine egne eksempler eller finde dem på nettet**

- Dette anbefales dog **ikke**.
- Dels skal du bruge længere tid på at forklare dem (da eksaminator og censor ikke kender dem i forvejen)
- Dels kan du ikke så godt få hjælp (fra eksaminator og censor), hvis der opstår problemer under præsentationen

# Træning gør mester

---

- **Hvert spørgsmål bør trænes mindst 5 gange**
  - Træningen skal være så realistisk som overhovedet muligt
  - Det er ikke nok at tænke på, hvad du vil sige og skrive
    - Du skal formulere sætningerne og sige dem højt
    - Du skal skrive tingene på et whiteboard eller et stykke papir
- **Eksamen er ikke en test i skønskrift – men det er oplagt en fordel at eksaminator og censor kan læse det, som du skriver**
  - Det er forbavsende svært at skrive læseligt på et whiteboard (øv dig i det)
  - Hold fornuftig tavleorden
  - Du må gerne forkorte lange navne og lignende og bruge gentagelsestegn/ streger (øv dig i at gøre det på en god måde)
  - Visk ikke noget ud (bortset fra smårettelser)
  - Eksaminator og censor tæller alt det med, som du har skrevet og sagt
- **Hold dig til dispositionen**
  - Lad være med at improvisere undervejs
  - Opfind ikke nye eksempler, som du ikke har gennemtænkt
- **Under de sidste træninger bør du ved hjælp af et ur tjekke, at du har stof nok til 10 minutter – hverken mere eller mindre**
  - De resterende 5 minutter af eksaminationen bruges til at besvare spørgsmål fra eksaminator og censor

# Gode råd omkring eksamen (fortsat)

---

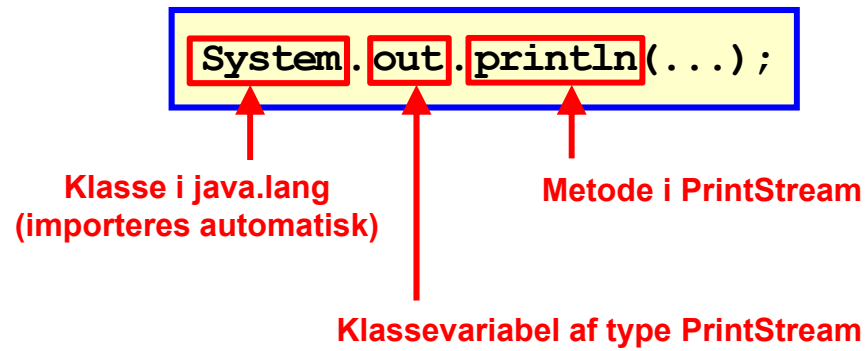
- **Husk at præsentere de overordnede begreber – inden du kaster dig ud i detaljeret Java kode**
  - Oprids f.eks. de vigtigste forskelle på imperativ og funktionel programmering samt fordelene ved funktionel programmering før du kaster dig ud i en masse detaljer
  - Forklar forskellen på **test** og **debugging**
- **Vis at du forstår, hvad der ligger bag de navne, som vi bruger i bogen / på kurset**
  - Forklar f.eks. at **regression** tests er automatiske tests, der let kan gentages når man har ændret sin kode – for at undersøge om der skulle være sket **regression** (tilbageslag/forringelse) i form af, at man har introduceret nye fejl under ændringerne
- **Du må gerne inddrage relevante ting fra andre spørgsmål**
  - F.eks. vil det være helt relevant at nævne, at brug af subklasser er fortrinlig til at undgå **kodeduplikering**
  - Brug dog ikke for megen tid på ting, der primært hører til andre spørgsmål



# Du skal kunne forklare din kode

---

- **Det er ikke nok at kunne skrive noget Java kode op**
  - Du skal også vise, at du forstår koden



# Videoer om mundtlig eksamen

---

- **Vi har produceret to videoer, som viser eksemplariske eksamenspræsentationer**
  - Det er en rigtig god ide at gennemse disse videoer – **gerne flere gange**
  - Videoerne giver en masse gode råd om, hvad man bør gøre, og hvad man bør undgå
- **Video 1: Arrays**
  - Videoen findes under Uge 9, og kan ses, så snart I har læst kapitel 7 i BlueJ bogen
- **Video 2: Graphical User Interfaces**
  - Videoen findes under Uge 13, og kan ses, så snart I har læst kapitel 13 i BlueJ bogen
- **I begge videoer varer præsentationen længere end de 10 minutter, som I har til jeres rådighed**

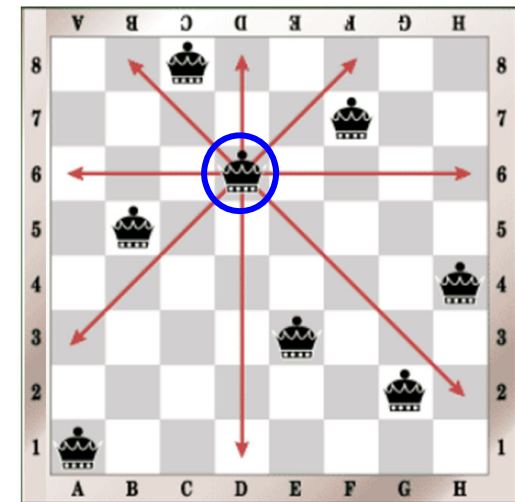
# ● Afleveringsopgave: Dronninger (Queens)

- I skal skrive et **rekursivt** program, der ved hjælp af **backtracking** finder alle løsninger til det såkaldte 8-dronningeproblem

- I skal placere 8 dronninger på et 8x8 skakbræt, således at ingen af dronningerne kan slå hinanden
- Den enkelte dronning må ikke have andre dronninger i den række, søjle og de to diagonaler, som går igennem dens position

- Mere generelt skal I løse n-dronningeproblemet

- I skal placere n dronninger på et n x n skakbræt, således at ingen af dronningerne kan slå hinanden (for  $n \geq 1$ ).



- **Repræsentation af dronningernes position på skakbrættet**
  - En oplagt mulighed er at bruge et 2-dimensionelt array **boolean[n][n]**, hvor den boolske værdi angiver, om der står en dronning på feltet eller ej
- **Men vi ved, at der højst kan være én dronning i hver række**
  - Derfor kan vi nøjes med et 1-dimensionelt array **int[n]**, hvor heltallet angiver, den position som dronningen i den pågældende række har
  - `queens[5] == 3` betyder, at dronningen i række 5 står i søjle 3
  - Som sædvanlig starter vi nummereringen med 0, dvs. række 0 og søjle 0

# positionQueens metoden (rekursiv metode)

---

- **Den centrale metode i programmet hedder positionQueens og fungerer som følger**
  - Et kald med parameterværdien 0 placerer en dronning i række 0, for derefter at kalde metoden med parameterværdien 1, hvilket placerer en dronning i række 1,
  - for derefter at kalde metoden med parameterværdien 2, hvilket placerer en dronning i række 2, for derefter at kalde metoden med parameterværdien 3,
  - osv.
- **Undervejs kan det være nødvendigt at gå tilbage (back-track'e) fordi man ikke kan placere den næste dronning**
- **Et kald, hvor parameterværdien er lig med antallet af dronninger, betyder, at vi har fået alle dronninger placeret**
  - Vi udskriver den fundne løsning
  - Derefter returnerer kaldet, og vi fortsætter med at finde de øvrige løsninger
- **Når vi har afprøvet alle muligheder stopper vi**

# Hvordan fungerer positionQueens?

Kald (7) → ← Række 7

Kald (6) →

Kald (5) →

Kald (4) →

Kald (3) →

Kald (2) → ← Række 2

Kald (1) →

Kald (0) →

↑ Søjle 0      ↑ Søjle 2

Og så videre.....

For 8 dronninger findes første løsning efter

- 876 forsøg på positioneringer
- 114 rekursive kald

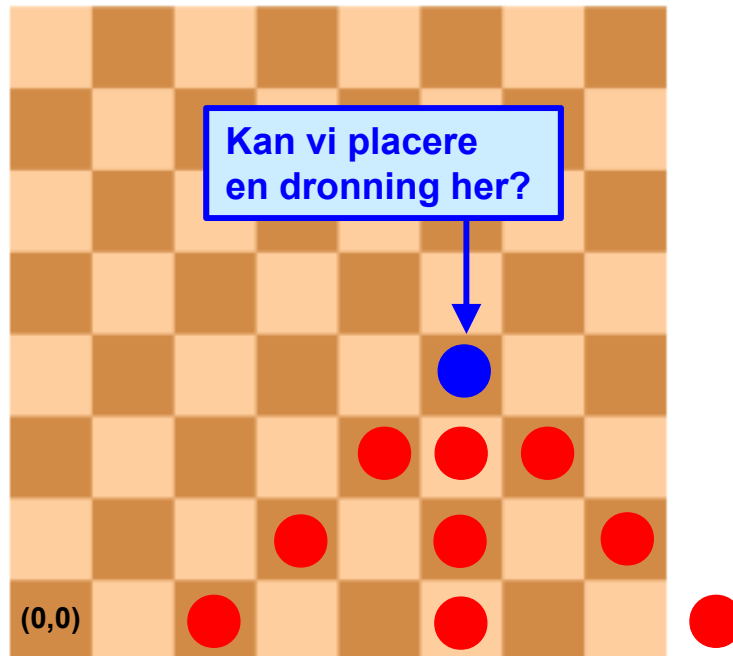
Godt vi ikke skal gøre det manuelt

Problemet vokser **meget hurtigt**, når vi øger antallet af dronninger

Dronninger	Løsninger	Rekursive kald	Tid
8	92	Godt 2 tusinde	1 millisekund
12	14.200	Knap 1 million	Under 1 sekund
16	Knap 15 millioner	Godt 1 milliard	Nogle få minutter

# legal metoden

- Skal tjekke om det er legalt at placere en dronning på et givet felt



Vi skal tjekke søjlen og de to diagonaler

- Vi behøver ikke at tjekke opad (der har vi endnu ikke placeret dronninger)
- Det gør ikke noget, at vi kommer uden for brættet (der er vi helt sikre på ikke at finde en dronning)

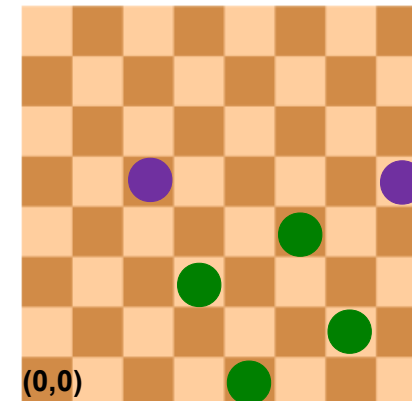
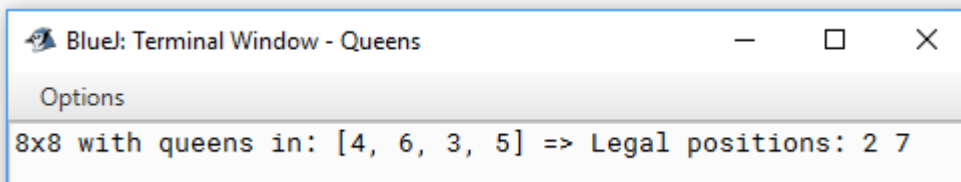
- **Det er vigtigt at denne metoder er effektiv**
  - For 8 dronninger kaldes den godt 15 tusinde gange
  - For 12 dronninger kaldes den godt 10 millioner gange
  - For 16 dronninger kaldes den godt 18 milliarder gange
  - Metoden bruger godt halvdelen af den samlede beregningstid

# Test af legal metoden

- Det er selvfølgelig også vigtigt at legal metode er korrekt
  - Lav en grundig afestning, før I forsøger at bruge den i positionQueens
  - Det kan f.eks. gøres som vist nedenfor

Størrelsen af brættet      Dronninger, der allerede er placeret

```
testLegal(8, 4, 6, 3, 5);
```



```
public void testLegal(int n, int... pos) {  
    noOfQueens = n;  
    queens = Arrays.copyOf(pos, n);  
    System.out.print(n + "x" + n + " with queens in: " +  
        Arrays.toString(pos) + " => Legal positions: ");  
    for(int i=0; i<n; i++) {  
        if(legal(pos.length, i)) { System.out.print(i + " ") };  
    }  
    System.out.println();  
}
```

Variabelt antal parametre  
Konverteres til et int array

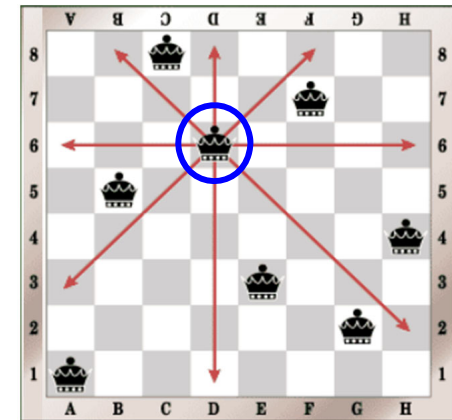
Initialiser feltvariable

Udskrift

Kald af legal

# convert metoden

- **Skal konvertere fra (row,col) notation til den sædvanlige skaknotation, hvor f.eks. (5,3) skrives som d6**
  - For at kunne gøre dette, har vi behov for at kunne mappe fra tal til bogstaver
  - I ovenstående eksempel skal heltallet **3** mappes i char værdien **'d'** (som er det 4. bogstav)
- **Det kan gøres på mindst fire måder, idet man kan bruge**
  - et map **Map<Integer, Char>**
  - et char array **char[ ]**
  - en tekststreng **"abcdefghijklmnopqrstuvwxyz"** og **charAt** metoden i String klassen
  - udtrykket **(char)('a' + i)**, der evaluerer til det i'te bogstav efter 'a' (virker kun hvis  $i \leq 27$ ; antallet af bogstaver i det engelske alfabet)



```
(char) ('a' + 3)
(char) (97 + 3)
(char) (100)
'd'
```

← 'a' forfremmes til heltallet 97

← 100 begrænses til char værdien 'd'

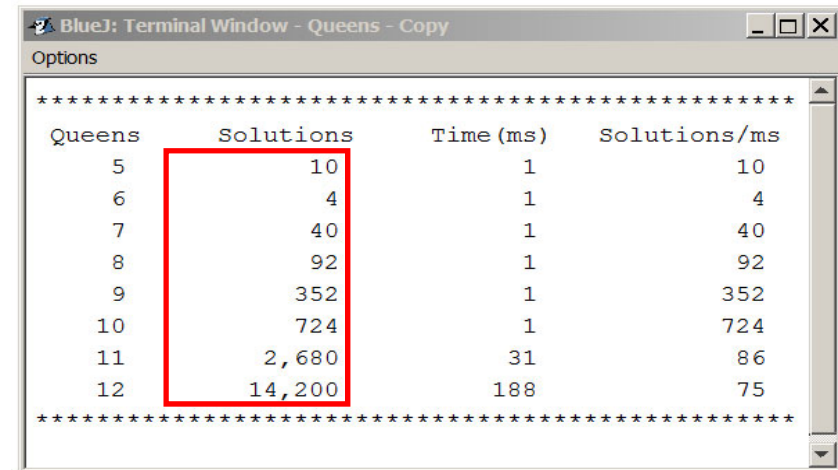


# Udskrift af tabeller

- Når man skal udskrive tabeller som nedenstående, kan det være hensigtsmæssigt at bruge **format** metoden (i stedet for println)

```
System.out.format("%6d %,12d %,10d %,12d %n",  
    noOfQueens, noOfSolutions, duration, noOfSolutions/duration);
```

- **Format metoden tager et variabelt antal parametre**
  - Første parameter angiver formatet af den tekststreng, der skal udskrives, mens de øvrige parametre angiver de værdier, der skal indsættes i tekststrengen (konverteres til et Object array)
  - Format parameteren indeholder nogle **format specifiers**
  - F.eks. angiver **%,12d**, at der på dette sted, skal udskrives et heltal, der skal fylde 12 tegn med et komma indsat for hvert tredje
  - **%n** angiver linjeskift (new line)
  - Der er tilsvarende format specifiers for reelle tal, datoer og tidspunkter
  - Detaljer kan ses på:  
<https://docs.oracle.com/javase/tutorial/java/data/numberformat.html> [Link](#)



Queens	Solutions	Time (ms)	Solutions/ms
5	10	1	10
6	4	1	4
7	40	1	40
8	92	1	92
9	352	1	352
10	724	1	724
11	2,680	31	86
12	14,200	188	75

# Hjælp og dokumentation

---

- **Dronningeopgaven er større og mere kompleks end de opgaver, som I hidtil har løst**
- **Via diskussionsforummet kan I få hurtig hjælp**
  - I må meget gerne svare på hinandens postings
  - Man kan poste anonymt, hvis man ønsker det (så kan andre studerende ikke se, hvem der poster, men forelæser og instruktører kan godt)
- **Studiecaféen er bemandet med en instruktør fra kurset på nedenstående tidspunkter**
  - Mandag kl. 12-14
  - Tirsdag kl. 8-10
  - Onsdag kl. 10-12
  - Torsdag kl. 10-12
  - Fredag kl. 13-15
- **Husk at dokumentere jeres program**
  - Sådan som det er vist i BlueJ bogen og ved en forelæsning
  - Ellers får I automatisk genaflevering

# ● Opsummering

---

- **Arrays**
  - Objektsamlinger med et fast antal elementer
  - Velkendt fra mange andre programmeringssprog
- **Brug af Java uden BlueJ**
  - Start af Java fra konsolvindue via main metode
- **Principper for design af klasser**
  - Undgå dublering af kode (code duplication)
  - Løs kobling mellem klasserne (loose coupling)
  - Sammenhængende klasser og metoder (cohesion)
  - Ansvarsfuldt design (responsibility-driven design)
  - Tænk fremad (think ahead)
  - Regelmæssig omstrukturering (refaktorering)
- **Mundtlig præsentation**
  - Kan som alt andet trænes – de næste uger vil vi træne systematisk
  - Vi vil fokusere på mundtlig eksamen, men det, som I lærer, vil også være nyttigt i mange andre situationer
- **Afleveringsopgave: Dronninger (Queens)**

**Det var alt for nu.....**

**... spørgsmål**

---

