

Forelæsning Uge 3 – Torsdag

- **Billedredigering**

- Gråtonebilleder (som er lidt simplere end farvebilleder)
- I uge 4 er der en afleveringsopgave, hvor I selv skal lave billedredigering

- **Rekursive metoder**

- Metoder der kalder sig selv
- Giver ofte meget elegante og simple løsninger på komplekse problemer

- **Refaktorering**

- Vi vil omstrukturere MusicOrganizer
- Et musiknummer repræsenteres nu ved hjælp af en Track klasse (i stedet for en tekststreng)
- Det gør det muligt at lave mere præcise søgninger

- **Iterator typen**

- Endnu en måde at gennemløbe en objektsamling



Om programmering

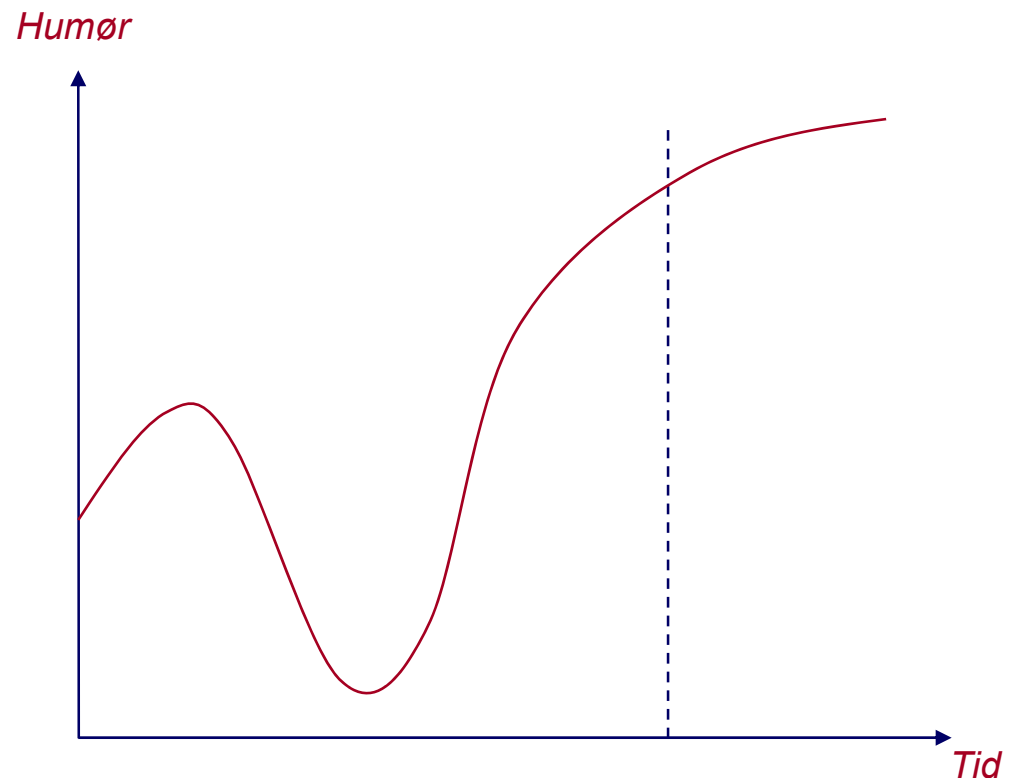
- **Programmering**

- Anderledes
- Ny tankegang

Fortvivl ikke – Tingene ændrer sig hurtigt

- **Faser**

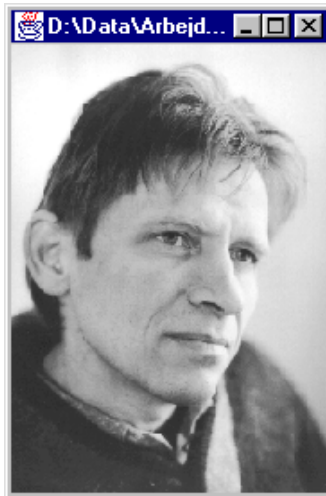
- Motivation
- Begejstring
- **Tvivl?**
- Frustration
- **Eksistentiel krise**
- Heureka!
- Fascination
- Indsigt
- **Magt over teknologien**



- **Hurra!**

- Programmering er **sjovt** og **stærkt vanedannende**
- Når man først kommer godt i gang, kan det være svært at stoppe igen

● Billedredigering



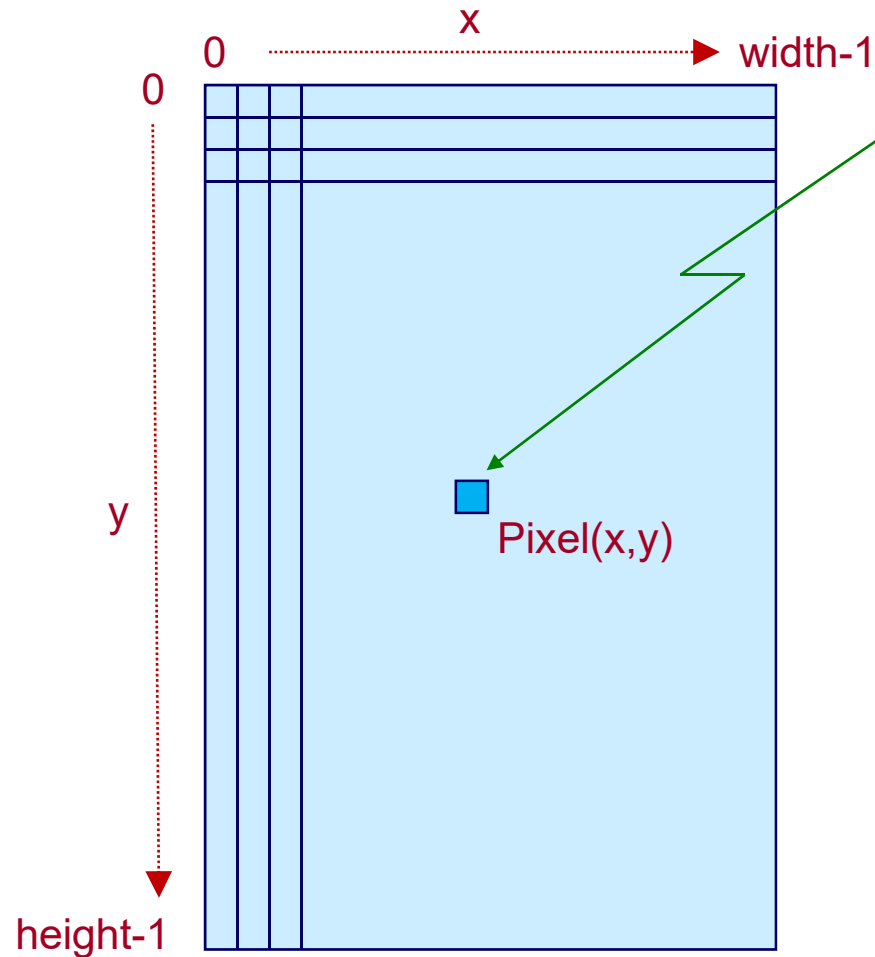
Lysere

Mørkere

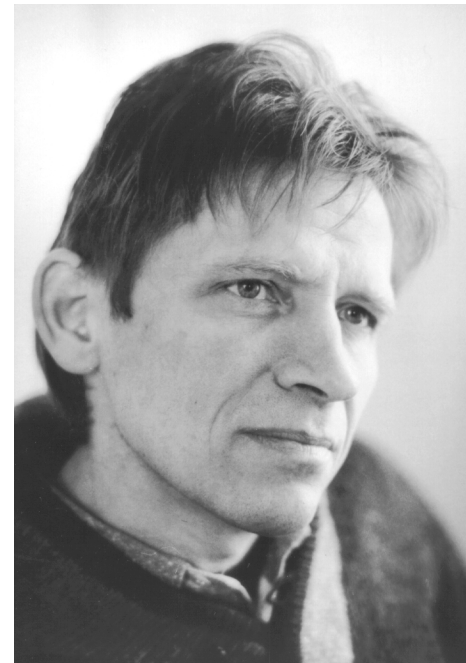
Inverteret

Uskarpt

Repræsentation af billede



Hver pixel har en gråtoneværdi i intervallet $[0..255]$, hvor 0 ~ sort og 255 ~ hvid

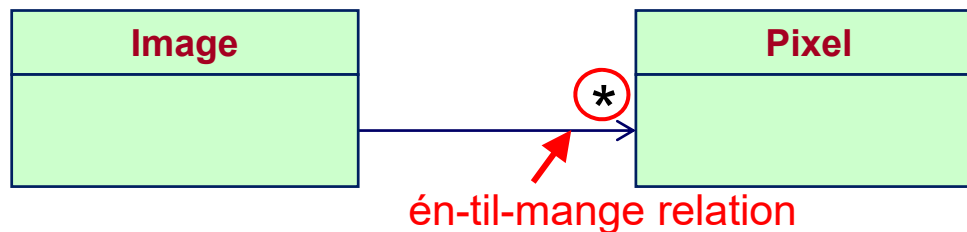


Intervallet $[0..255]$ har 256 værdier og kan derfor repræsenteres ved hjælp af en byte (8 bits): $2^8 = 256$

Image og Pixel klasserne

- **Vi bruger to klasser**

- Image repræsenterer et billede og har metoder, som arbejder på billedet, bl.a. brighten, darken, invert og blur
- Pixel repræsenterer en enkelt pixel og har metoder til at aflæse og sætte pixelens gråtoneværdi



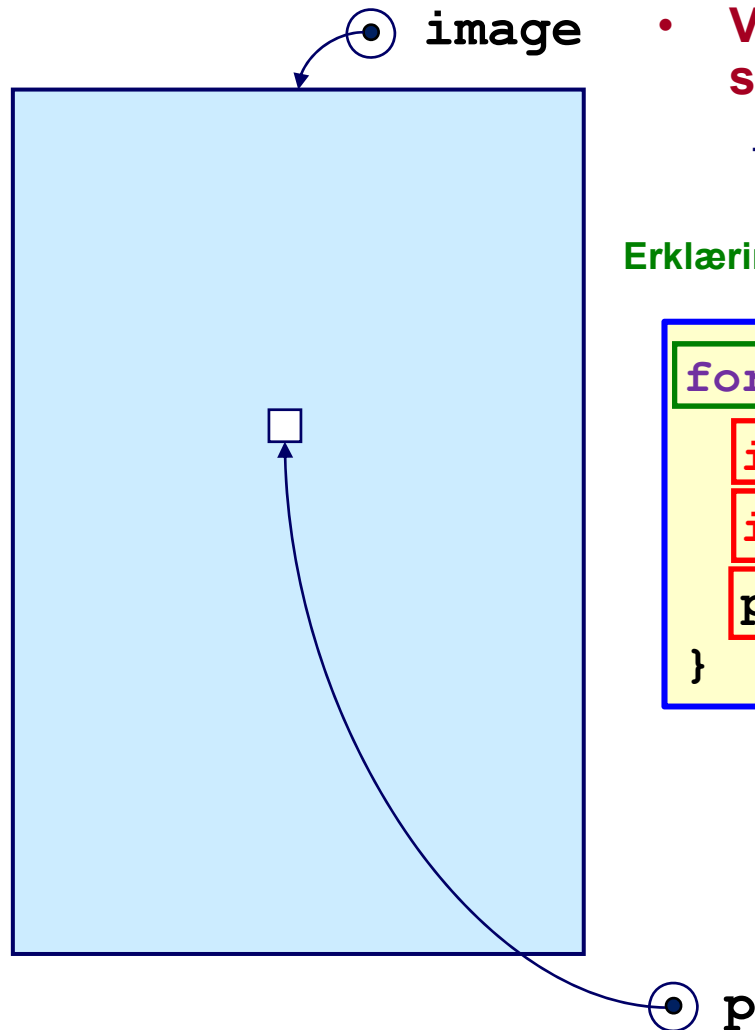
```
public class Pixel {  
    private int value; // Pixelens gråtoneværdi [0,255].  
    // Konstruktøren initialiserer gråtoneværdien.  
    public Pixel(int value) {...}  
    // Returnerer gråtoneværdien for denne pixel.  
    public int getValue() {...}  
    // Opdaterer gråtoneværdien for denne pixel.  
    public void setValue(int value) {...}  
}
```

Interface for Image klassen (signaturer)

```
public class Image {  
    ...  
    // Returnerer billedets bredde.  
    public int getWidth() {...}  
  
    // Returnerer billedets højde.  
    public int getHeight() {...}  
  
    // Returnerer pixlen på position (x,y).  
    public Pixel getPixel(int x, int y) {...}  
  
    // Returnerer en arrayliste med samtlige pixels i billedet.  
    public ArrayList<Pixel> getPixels() {...}  
  
    // Returnerer de op til ni naboer til (x,y) (inklusive (x,y)).  
    public ArrayList<Pixel> getNeighbours(int x, int y) {...}  
  
    // Gentegner billedet.  
    public void updateCanvas() {...}  
    ...  
}
```

Udvalgte metoder

Skabelon for simpel billedoperation



- Vi bruger en for-each løkke til at gennemløbe samtlige pixels og opdatere dem en efter en
 - Rækkefølgen er ligegyldig for os

Erklæring af lokal variabel Den arrayliste der skal gennemløbes

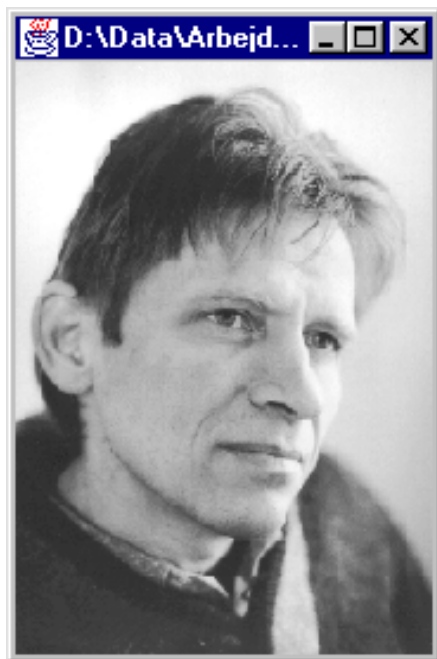
```
for( Pixel p : image.getPixels() ) {  
    int oldValue = p.getValue();  
    int newValue = .....oldValue.....;  
    p.setValue(newValue);  
}
```

find p's
værdi

opdater p's værdi

beregn en ny
værdi ud fra
den gamle

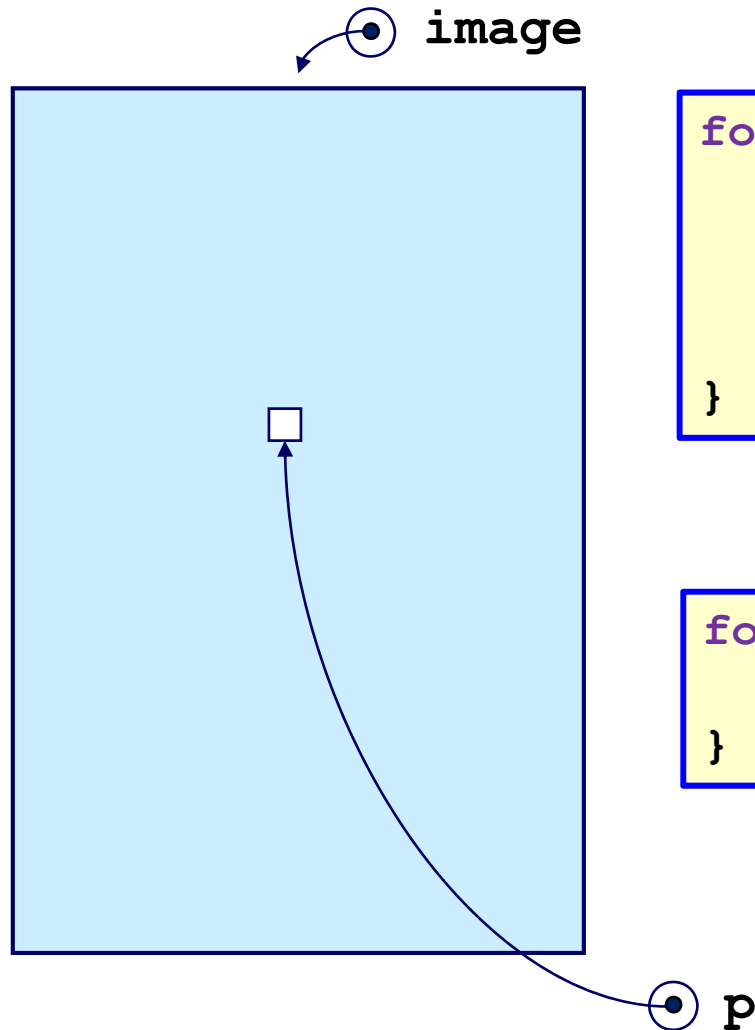
Brighten (lysere)



`newValue = oldValue + 30`

0 ~ sort,
255 ~ hvid

Brighten, Javakode



```
for( Pixel p : image.getPixels() ) {  
    int oldValue = p.getValue();  
    int newValue = oldValue + 30;  
    p.setValue(newValue);  
}
```



```
for( Pixel p : image.getPixels() ) {  
    p.setValue(p.getValue() + 30);  
}
```

Kan I se et potentielt problem?

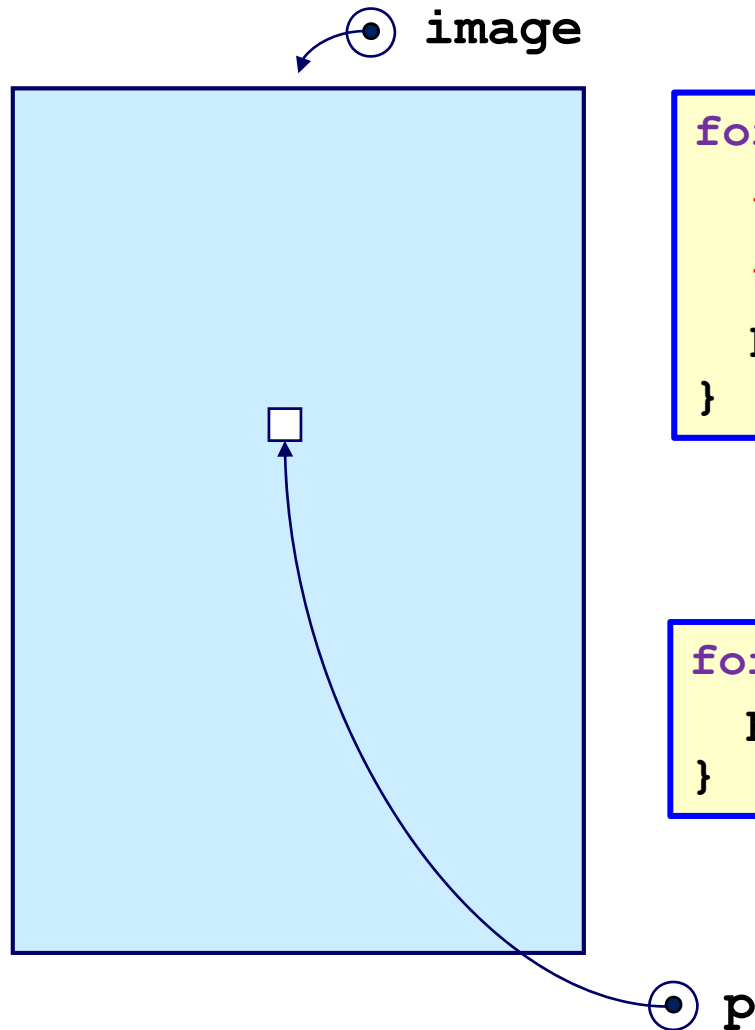
**setValue metoden sørger for at
værdien ligger i intervallet [0,255]**

Invert (byt om på sort og hvid)



`newValue = 255 - oldValue`

Invert, Javakode



```
for( Pixel p : image.getPixels() ) {  
    int oldValue = p.getValue();  
    int newValue = 255 - oldValue;  
    p.setValue(newValue);  
}
```



```
for( Pixel p : image.getPixels() ) {  
    p.setValue(255 - p.getValue());  
}
```

Andre billedoperationer

- **I den anden afleveringsopgave i uge 4 skal I implementere nedenstående billedoperationer**
 - **brighten** Gør billedet lidt lysere
 - **darken** Gør billedet lidt mørkere
 - **invert** Inverterer hver gråtone
 - **blur** Erstatte hver pixel med gennemsnittet af naboerne
 - **mirror** Spejler billedet om den lodrette midterakse
 - **flip** Spejler billedet om den vandrette midterakse
 - **rotate** Roterer billedet 90 grader med uret
 - **resize** Skalerer billedet, så størrelsen ændres

Quiz

● Rekursive metoder

- Fakultets funktionen $n!$ er defineret ved

$$n! = 1 * 2 * \dots * (n-1) * n \quad \text{for } n \geq 1$$

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

$$1! = 1$$

- Beregning ved hjælp af en for løkke

```
public int faculty(int n) {  
    int result = 1;  
    for(int i=2; i<=n; i++) {  
        result *= i;  
    }  
    return result;  
}
```

Fakultets funktionen (rekursiv)

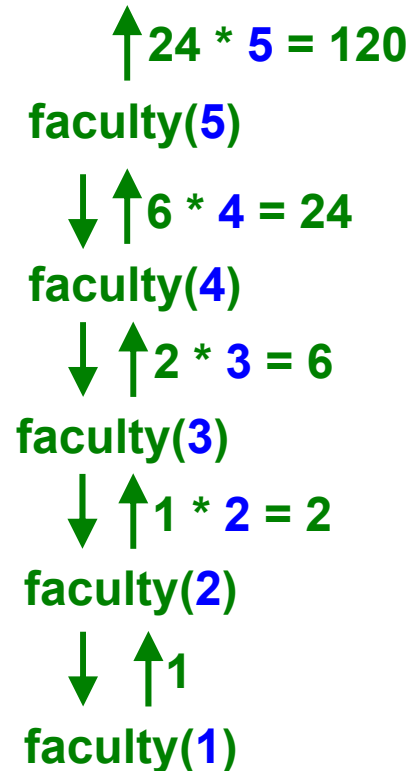
- Fakultets funktionen kan også defineres rekursivt, dvs. ved hjælp af sig selv

```
1! = 1
n! = (n-1)! * n    for n > 1
```

- Rekursiv metode til beregning af n!

```
public int faculty(int n) {
    if (n == 1) { return 1; }
    return faculty(n-1) * n;
}
```

- Hvad sker der, hvis metoden kaldes med en negativ parameter værdi?
 - Vi laver en "uendelig" sekvens af rekursive kald
 - Det kan datamaskinen ikke klare, idet den jo har begrænset lagerplads)



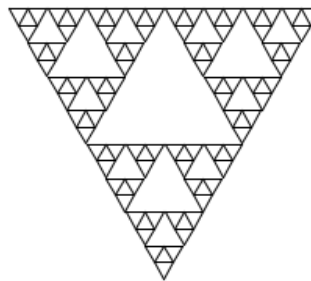
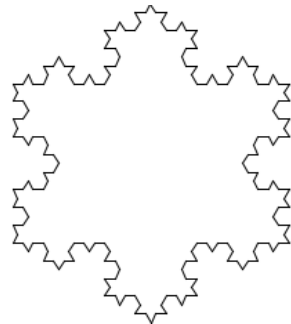
```
java.lang.StackOverflowError
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
at RecursiveMethods.faculty(RecursiveMethods.java:16)
```

Idéen bag rekursion

- **Vi har en række problemer, der "ligner hinanden", men har forskellig "størrelse"**
 - F.eks. ligner beregningen af $5!$, $4!$, $3!$, $2!$ og $1!$ hinanden, men har forskellige størrelse (nemlig 5, 4, 3, 2 og 1)
- **Vi løser problemet for en given størrelse ved at bruge løsningen af et mindre problem**
 - Typisk løses **problem(n)** ved hjælp af løsningen for **problem(n-1)**
 - I vores eksempel ved vi, at $n! = (n-1)! * n$
 - Det betyder at $n!$ kan løses/beregnes ved hjælp af $(n-1)!$
- **Sudoku løseren (fra første forelæsning) bruger også rekursion**
 - Hvert rekursivt kald placerer et ciffer (i første tomme felt), hvorefter det laver et nyt rekursivt kald
 - I dette tilfælde er problemets størrelse **antallet af tomme** felter (der endnu ikke har fået et ciffer)
 - Når der ikke er flere tomme felter **stopper** vi rekursionen og udskriver den fundne løsning

Rekursion ligner induktionsbeviser

- I induktionsbeviser har vi også en række problemer, der ligner hinanden, men har forskellig "størrelse"
 - Idéen bag **induktionsbeviser** er, at vi **beviser** $\text{problem}(n)$ ud fra $\text{problem}(n-1)$
 - Idéen bag **rekursion** er, at vi **løser/beregner** $\text{problem}(n)$ ud fra $\text{problem}(n-1)$
 - Man kan bruge induktionsbeviser til at **bevise** at en rekursiv beregning er korrekt
- Vi vil nu kigge på et par andre eksempler på rekursive beregninger
 - I næste forelæsning vil vi se på nogle rekursive metoder til at tegne komplekse figurer



Fibonacci tallene

- Nedenstående talfølge, hvor hvert tal er lig summen af de to foregående

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

- Beregning ved hjælp af en for løkke

```
public int fibonacciLoop(int n) {  
    if(n == 1) { return 0; }  
  
    int first = 0;  
    int second = 1;  
    int temp;  
  
    for(int i = 3; i <= n; i++) {  
        temp = second;  
        second += first;  
        first = temp;  
    }  
  
    return second;  
}
```

Lidt svært at gennemskue,
hvad der sker i løkken

Beregning af 233
ud fra 89 og 144
first = ~~89~~ 144
second = ~~144~~ 233
temp = 144

Fibonacci funktionen (rekursiv)

- **Rekursiv definition**

```
fib(1) = 0
fib(2) = 1
fib(n) = fib(n-2) + fib(n-1)    for  $n \geq 3$ 
```

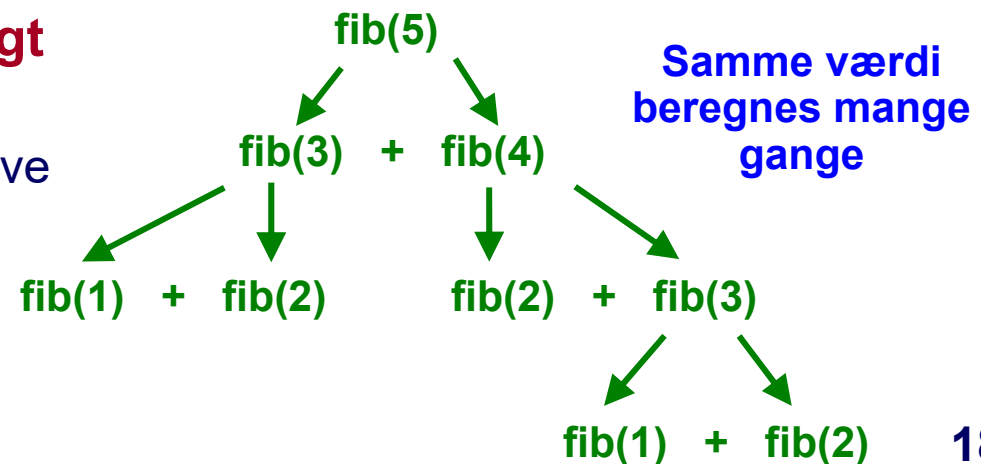
- **Rekursiv beregning**

```
public int fibonacci(int n) {
    if (n == 1) { return 0; }
    if (n == 2) { return 1; }
    return fibonacci(n-2) + fibonacci(n-1);
}
```

I dette tilfælde løses problem(n) ved hjælp af problem(n-2) og problem(n-1)

- **Rekursion er utroligt nyttigt og anvendes meget**

- Sommetider kan det dog give ineffektive løsninger



Palindrom

- Nedenstående metode tjekker om parameteren er et palindrom, dvs. ens forfra og bagfra (fx "kik", "anna", "!" og "")

Java
API

- Den bruger tre metoder fra String klassen

- **length()** returnerer længden af strengen

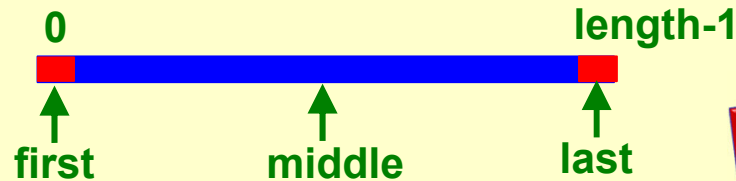
- **charAt(int i)** returnerer det tegn (af typen char), der er på pladsen index i

- **substring(int i1, int i2)** returnerer den delstreng, der starter i i1 og slutter i i2-1

Et String objekt består af et antal tegn (char) indexeret fra 0 til length()-1

```
public boolean palindrome(String s) {  
    int length = s.length();  
    if(length <= 1) { return true; }  
    // Divide string.  
    char first = s.charAt(0);  
    char last = s.charAt(length-1);  
    String middle = s.substring(1, length-1);  
  
    if(first != last) {  
        return false;  
    }  
    else {  
        return palindrome(middle); // Recursive call.  
    }  
}
```

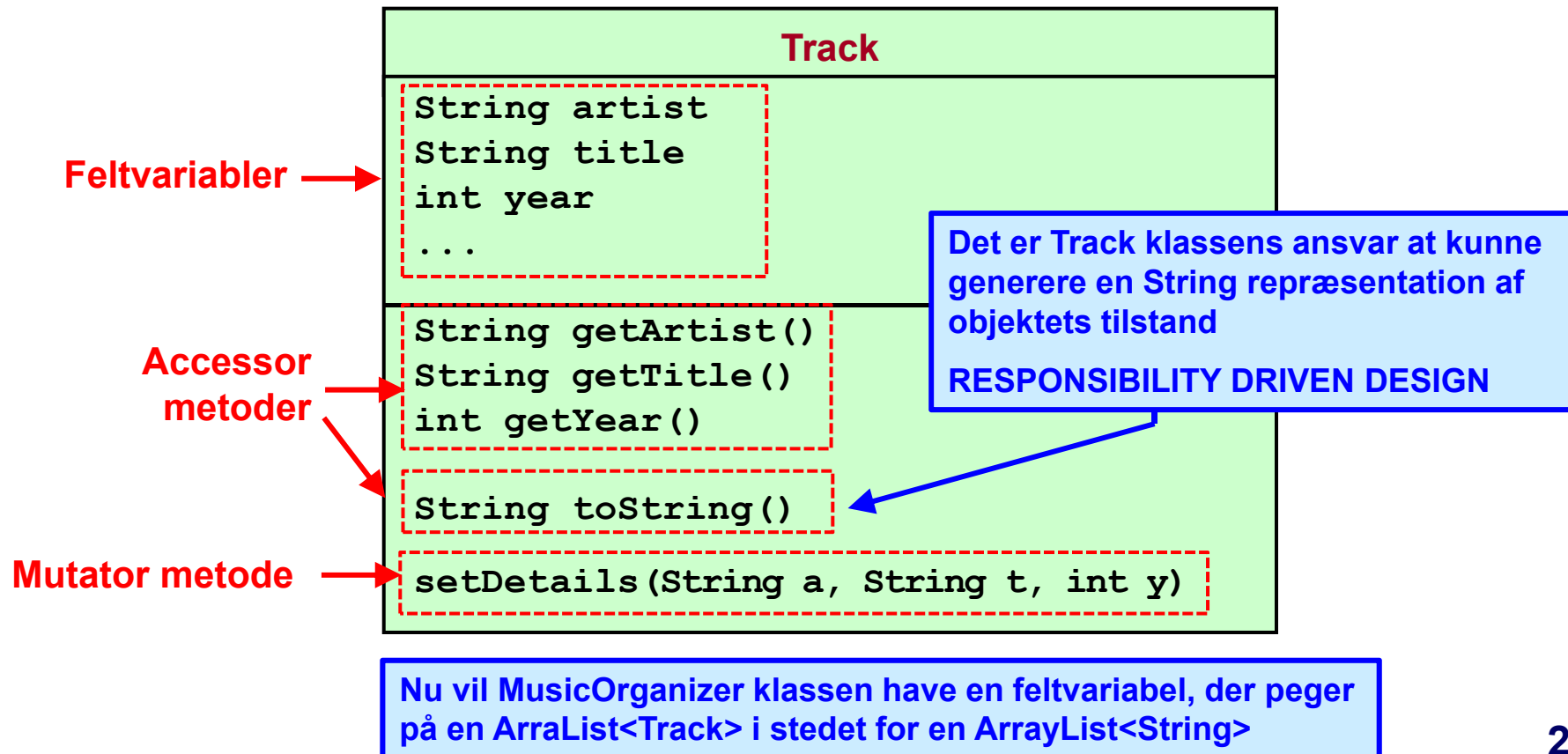
I dette tilfælde løses problem(n) ved hjælp af problem(n-2), hvor n er teststrengens længde



Pause

● Refaktoring af MusicOrganizer

- I første version af MusicOrganizer har vi repræsenteret et musiknummer ved hjælp af en tekststreng
 - Nu vil vi indføre en klasse Track som modellerer musiknumre
 - På den måde kan vi bedre skelne mellem de enkelte elementer, f.eks. navnet på artisten og navnet på musiknummeret



Vi kan nu lave mere præcise søgninger

- Find **et** musiknummer, hvor titlen indeholder en bestemt tekststreng

Hvis vi finder et track, der opfylder betingelsen, returneres det, ellers returneres null

```
public Track findByTitle(String q) {  
    for (Track t : tracks) {  
        if (t.getTitle().contains(q)) {  
            return t;  
        }  
    }  
    return null;  
}
```

- Ifølge BlueJ bogen bør man kun bruge for-each løkker, når man vil gennemløbe hele arraylisten
- Jeg har intet problem med, at man afbryder gennemløbet undervejs, når man har fundet det, man søger
- Bogen er ikke konsistent: På side 301 afbrydes gennemløbet af en for-each løkke

- Find **alle** musiknumre, hvor kunstnernavnet indeholder en bestemt tekststreng

```
public ArrayList<Track> findByArtist(String q) {  
    ArrayList<Track> result = new ArrayList<>();  
    for (Track t : tracks) {  
        if (t.getArtist().contains(q)) {  
            result.add(t);  
        }  
    }  
    return result;  
}
```

De tracks, der opfylder betingelsen, returneres i en arrayliste (som kan være tom)

Refaktoring (omstrukturering)

- **På de foregående slides har vi foretaget en refaktoring af arkitekturen for MusicOrgnizer**
 - Vi har erstattet brugen af tekststrengene til repræsentation af musiknumre med Track klassen – som giver en bedre og mere detaljeret beskrivelse af musiknumres egenskaber
 - Andre eksempler på refaktoring vil være opdeling af en klasse, der er blevet meget stor eller indeholder metoder, der ikke naturligt hører sammen
- **Under udviklingen af et system er der ofte behov for at lave refaktoring**
 - Når man refaktorerer ændrer man systemets arkitektur uden at ændre dets opførsel
 - Efter refaktoringen tester man, at det nye program virker på samme måde som det gamle
 - Først derefter tilføjer man ny funktionalitet

Forskellige former for gennemløb

- Når vi har en arrayliste kan vi gennemløb elementerne på forskellige vis

for-each
løkke

```
for( String s : list) {  
    print(s);  
}
```

Hjælpe metode

```
private void print(String s) {  
    System.out.println(s);  
}
```

for
løkke

```
for( int i = 0; i < list.size(); i++ ) {  
    print(list.get(i));  
}
```

while
løkke

```
int i = 0;  
while( i < list.size() ) {  
    print(list.get(i));  
    i++;  
}
```

Quiz

Iterator +
while løkke

```
import java.util.Iterator;  
.....  
Iterator<String> it = list.iterator();  
while( it.hasNext() ) {  
    print(it.next());  
}
```

- Brug af Iterator er mere kompliceret end de andre løkker
- Men har nogle fordele, som vi skal se på om lidt

● Iteratorer typen

- **Iterator typen indeholder faciliteter til gennemløb af en objektsamling (f.eks. en arrayliste)**
 - Metoden **hasNext()** returnerer en boolsk værdi, som indikerer, om der er flere elementer at besøge
 - Metoden **next()** returnerer det næste element i objektsamlingen
 - "Flytter" samtidig iteratoren – således at næste kald af **next()** returnerer det efterfølgende element i objektsamlingen
 - Hvis **hasNext()** returnerer false, vil et kald af **next()** generere en runtime fejl
- **Alle collection typer har en iterator**
 - Man får fat i et iterator objekt ved at kalde metoden **iterator()**

```
Importér Iterator → import java.util.Iterator;
...
Erklær lokal variabel it (reference til en iterator for den objektsamling, som list peger på) → Iterator<String> it = list.iterator();
...
Brug iteratorens metoder → while (it.hasNext()) {
    print(it.next());
}
...
```

Parametriseret type

24

Hvorfor bruge en iterator?

- **Nogle collection typer mangler et index begreb (det gælder f.eks. mængder og træer)**
 - For disse kan man **ikke** bruge en løkke, der referer til indices
 - Men man kan bruge en **for-each** løkke til at gennemløbe alle elementer
- **Man kan have behov for at fjerne elementer i den objektsamling, som man er i færd med at gennemløbe**
 - Hvis man kalder objektsamlingens **remove** metode under et gennemløb af en **for-each** løkke får man en runtime fejl (exception)
 - Hvis man gør det inde i en **for**, **while** eller **do-while** løkke, går der let "koks" i iterationen (fordi indices forskydes)
- **Iterator typen har en remove metode, som tillader, at man fjerner det element, som sidste kald af next returnerede**
 - Ved at bruge **iterator typens remove metode** (sammen med hasNext og next) kan man i en while eller do-while løkke fjerne elementer, uden at der går "koks" i iterationen

● Opsummering

- **Billedredigering**

- Gråtonebilleder (som er lidt simplere end farvebilleder)

- **Rekursive metoder**

- Metoder der kalder sig selv
- Giver ofte meget elegante og simple løsninger på komplekse problemer

- **Refaktoring**

- Vi omstrukturerede MusicOrganizer
- Et musiknummer repræsenteres nu ved hjælp af en Track klasse (i stedet for en tekststreng)

- **Iterator typen**

- Ny måde at gennemløbe en objektsamling
- Bruges når objektsamlingen ikke har indices eller man har behov for at fjerne elementer under gennemløbet

Husk at aflevere

- Quiz 3 (alene)
- Raflebæger 3 (par)
- Skildpadde 1 (par)
- Eventuelle genafleveringer fra tidligere uger

Det var alt for nu.....

... spørgsmål

