

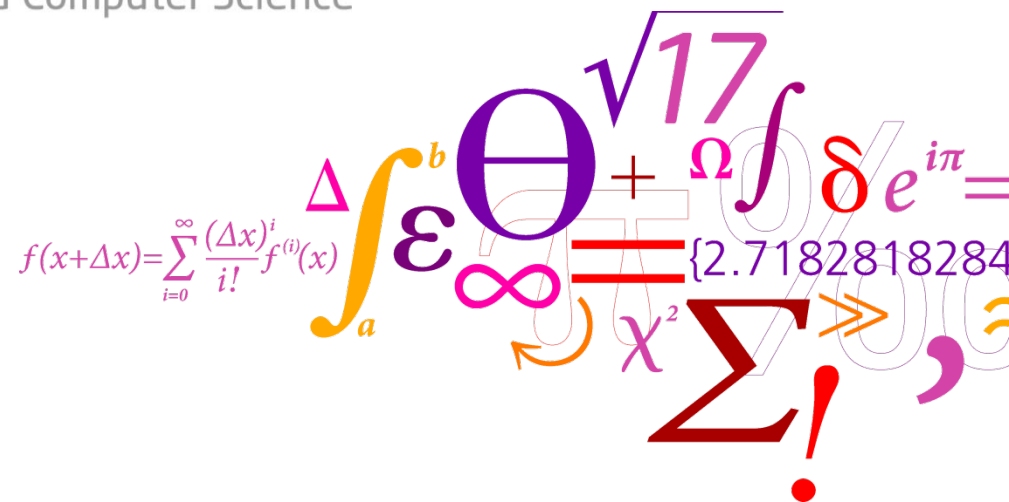
Projekt i software-udvikling (02362)

Forår 2022

Ekkart Kindler

DTU Compute

Department of Applied Mathematics and Computer Science



V. Rapportskrivning (Softwaredokumenter)

DTU Compute

Department of Applied Mathematics and Computer Science

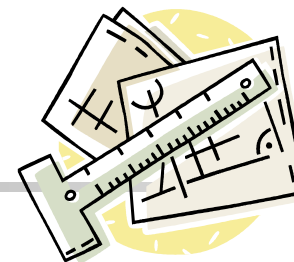
$$f(x+\Delta x)=\sum_{i=0}^{\infty}\frac{(\Delta x)^i}{i!}f^{(i)}(x)$$
$$\int_a^b \varepsilon \Theta + \Omega \int \delta e^{i\pi} = \{2.7182818284\}$$
$$\chi^2 \Sigma !$$

- Projektdefinition
- Håndbog
- Kravspecifikation
 - grov
 - detaljeret
- Fuldstændige softwaremodeller
- Systemspecifikation
- Implementeringen + dokumentation



hvorfor /
why

hvad /
what

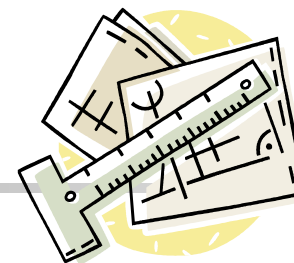


hvordan /
how

- Projektdefinition
- Håndbog
- Kravspecifikation
 - grov
 - detaljeret
- Systemspecifikation
- Fuldstændige softwaremodeller
- Implementeringen + dokumentation



grov



detaljeret

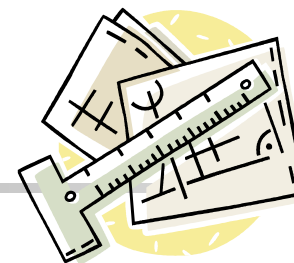
- Projektdefinition
- Håndbog
- Kravspecifikation
 - grov
 - detaljeret
- Systemspecifikation
- Fuldstændige softwaremodeller
- Implementeringen + dokumentation



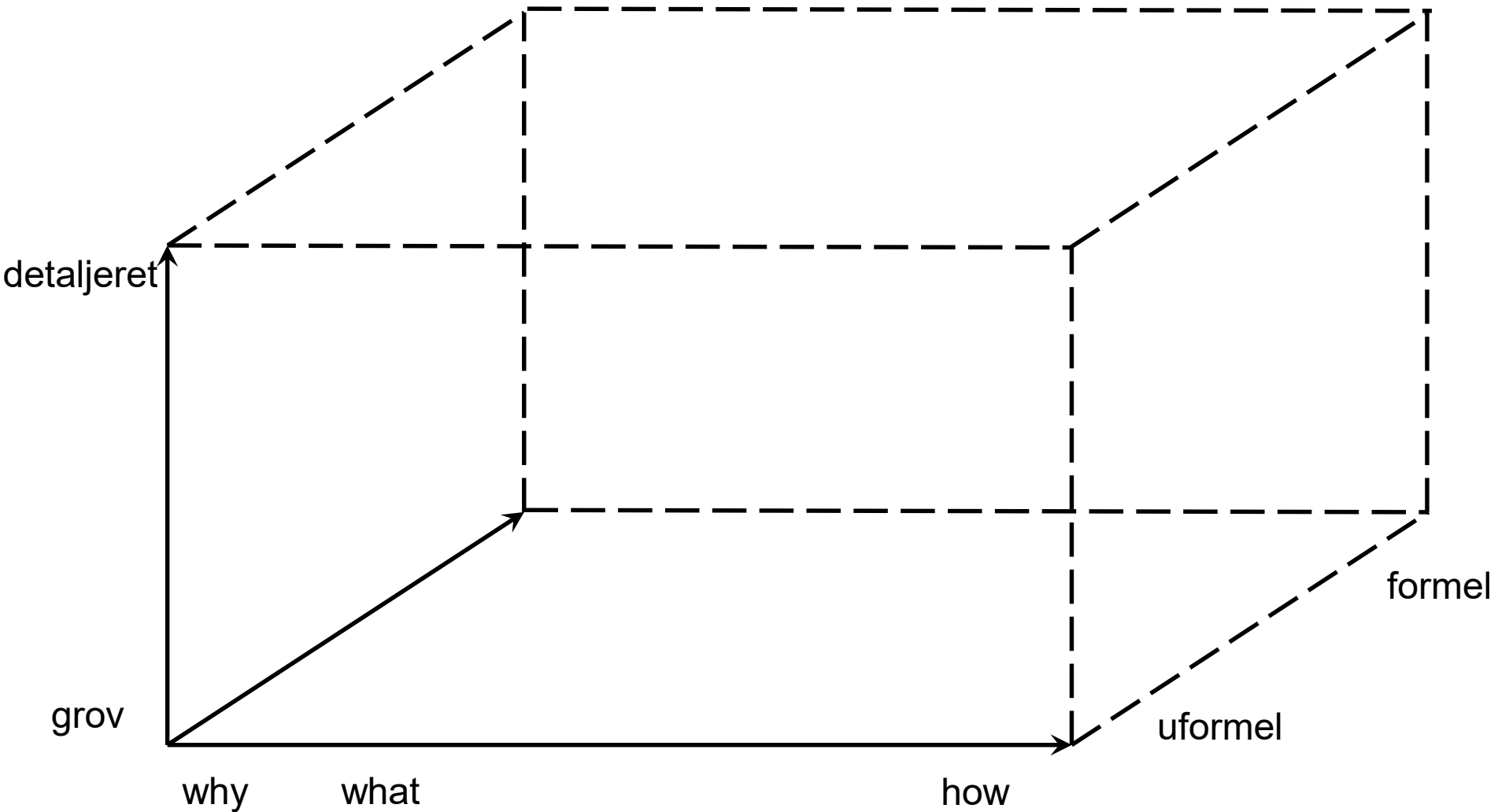
uformel



formel



- Why $\leftrightarrow$ what $\leftrightarrow$ how
- Detaljeringsgrad (grov $\leftrightarrow$ detaljered)
- Formalitetsgrad (uformel $\leftrightarrow$ formel/)



- **Introduktion**
 - Kort overblik over projektet og hovedpunkter, som opgaven indebærer
 - Overblik over selve rapporten og hvordan den skal læses
- **Problembeskrivelse**
 - Kontekst og baggrund
 - Overblik over hovedkoncepter og -funktioner (i grupper) med prioritering og argumentation for denne (kan fx bruge MoSCoW-kategorier: Must, Should, Could, Won't)
 - Husk også at nævne, hvilke data, der skal gemmes permanent (persistence)
 - Hvad kan I bygge videre på (fx GUI og kode i fik vedr. opgaver 1-3 mv.)

■ Kravspecifikation (Analyse)

Beskrivelse af krav fra brugerens perspektiv

- Beskriv begreber, funktioner og use-cases (som brugeren opfatter dem)
- Beskriv de relevante informationer som domænemodel
 - Klassediagrammer
 - Aktivitetsdiagrammer
 - evt. tilstandsdiagrammer
- Beskriv ikke-funktionelle krav: fx brugbarhed (usability), vedligeholdbarhed (maintainability), ...
- Evtl. UC-diagram med beskrivelse af de vigtigste use-cases

Alle diagrammer skal forklares i teksten!

I må gerne genbruge jeres eget materiale fra de andre afleveringsopgaver igen til jeres egen rapport! Vi arbejder inkrementel; og det gælder software og rapporten!

■ Kravspecifikation (Analyse)

Beskrivelse af krav fra brugerens perspektiv

- Beskriv begreber, funktioner og use-cases (som brugeren opfatter dem)
- **Beskriv de relevante informationer som domænenemodell**
 - Klassediagrammer
 - Aktivitetsdiagrammer
 - evt. tilstandsdiagrammer
- Beskriv ikke-funktionelle krav: fx brugbarhed (usability), vedligeholdbarhed (maintainability), ...
- Evtl. UC-diagram med beskrivelse af de vigtigste use-cases

Var afleveringsopgave A1
til i dag!

■ Kravspecifikation (Analyse)

Beskrivelse af krav fra brugerens perspektiv

- Beskriv begreber, funktioner og use-cases (som brugeren opfatter dem)
- Beskriv de relevante informationer som domænemodel
 - Klassediagrammer
 - Aktivitetsdiagrammer
 - evt. tilstandsdiagrammer
- Beskriv ikke-funktionelle krav: fx brugbarhed (usability), vedligeholdbarhed (maintainability), ...

■ Evtl. UC-diagram med beskrivelse af de vigtigste use-cases

Alt sammen op til her

(Intro, kort problembeskrivelse, kort kravspecifikation):

Aflevering A2 (til den 8. 3.): Projektbeskrivelse med kravanalyse

→ Der ligger et eksempel på kursets materiale side (dog uden analysedelen)!

- Database- og softwaredesign
Overblik over de vigtigste komponenter, som inkluderer, også tilkobling a databasen: arkitektur
 - Software design: hovedkomponenter af software og deres sammenspil
 - Klassediagrammer (indeholder især de vigtigste model-, kontroller- og view-klasser)
 - Sekvensdiagrammer som viser samspil mellem vigtige klasser/komponenter
 - Database design → se DB kursus

Husk at alle diagrammer skal forklares i teksten!

- Implementering
 - Hvordan er designet bliver implementeret (men kun nogle interessante udtræk, ikke hele koden)
 - Dette omfatter software og database
 - Herunder kan I også diskutere evt. mangler af jeres implementering
- Udviklingsproces og test
 - Udviklingsproces og brugte værktøjer
 - Diskussion af JUnit-test (automatisk)
 - Diskussion af acceptance-test (manuelle test, fx use-cases)

Det skal være muligt
at bruge softwaren
uden anden
information!

- Håndbog
 - Hvordan installerer man selve software
 - Hvordan starter man softwaren
 - Hvordan bruger man software (den eksisterende GUI er i ikke nødt til at diskutere med alle detaljer; **men giv et øeblik og diskuter jeres tilføjelser**); evt. med nogle screenshots
- Konklusion
 - Opsamling af hele resultatet
 - Nogle afsluttende bemærkninger

- Referencer
 - Litteratur og
 - Websider I har brugt
- Appendiks (evt.)
 - Nogle relevante diagrammer som ikke kunne være med i hoveddelen
 - Alle use-cases, måske med aktivitetsdiagrammer til de vigtigste regler/spille-faser
 - Glossar/taksonomi (som I ville have i forvejen gennem jeres domæneanalyse)

Alt sammen er
nogenlunde en
systemspekifikation!

- I skriver rapporten ikke for jer selv eller Ekkart!
- I skriver rapporten til en som står udenfor projektet (især censor)!
- Der er brug for at introducere konteksten og hvad RoboRally og dens begreber og regler er!

- Spiralformskrivning ($\rightarrow$ tavle)
- Undgå indforståede afsnit
(fortæl alt – ingen implicitte antagelser)
- Diskuter en idé (et argument) ad gangen
(og lav pointen eksplicit)
- Brug enkle begreber og korte sætninger
- Et begreb = et ord (det samme ord hele vejen igennem)
- Brug entalsform (singularis) hvis muligt

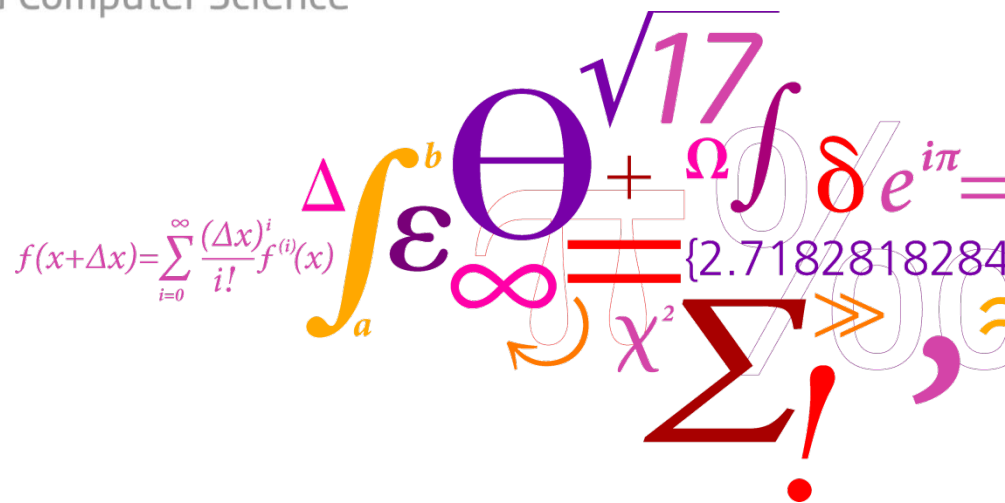
- @author tags i koden og rapporten
(helst studienummer s123456 ellers fuld navn)
- I koden:
 - på klasseniveau (for nye klasser)
 - på metode niveau (hvis eksisterende klasser ændres)
 - rækkefølgen af @author tags betyder noget
- I rapporten:
 - på afsnit eller underafsnitsniveau
 - nogle afsnit kan have flere autorer

IV. Java Praksis

Fortsætter fra tidligere

DTU Compute

Department of Applied Mathematics and Computer Science



A collage of colorful mathematical symbols and expressions. It includes a large purple Θ , a yellow integral $\int_a^b$, a pink infinity ∞ , a red summation $\sum$, a blue square root $\sqrt{17}$, a red delta δ , a blue $e^{i\pi}$, a yellow χ^2 , a red $!$, a yellow $>$, a blue $\{2.7182818284\}$, and a yellow ε . The background is white with faint, light blue mathematical symbols like $\int$, $\sum$, and ∞ .

3. Samlinger (collections)

- Motivation: Indtil videre har I mest brugt arrays for håndtere "samlinger" af objekter.
- Men koden bliver ofte svært at læse og "det er meget nemt" at lave fejl med arrays.
- Desuden er de ufleksible (kan fx ikke udvides dynamisk)
- Derfor har Java en masse datatyper til at håndtere "samlinger" (collections) af objekter af forskellig slags (lister, mængder, mappings).

Det er dem man helst skal bruge i større software – ikke arrays!

Senere i kurset lærer I også hvordan man kan

- Definere, implementere og dokumentere datastrukturer og APler på en elegant måde, og
- nogle nye programmerekoncepter som understøtter dette:
 - Interfaces
 - Algoritmer
 - Rekursion
 - Generiske datatyper
 - Exceptions
 - Javadocs

Men: efter I har lært det, skal I helst ikke implementere jeres egne datatyper, hvis der eksisterer nogle indbyggede i Java allerede:

- "Javaholdet" har meget mere erfaring,
- mere tid for at realisere og teste dem og
- flere burger som har brugt datatyper (og fundet mulige fejl)

I skal bruge jeres tid på det væsentlige i et projekt – ikke på standardstrukturer!

Derfor starter vi med at bruge
Java's indbyggede datatyper!

Derfor kigger vi nu kort på Javas indbyggede datastrukturer, især

- Collection (Set, List, Queue, Deque),
- Iterator,
- Comparable og
- Map

Siden Java 8 eksisterer der også streams med streams aggregate funktioner i Java, som er et meget elegant koncept (som man kender fra den funktionelle programmering). Det kommer vi desværre ikke ind på.

...

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class Board extends Subject {
```

...

```
private List<Player> players = new ArrayList<Player>();  
private Player current;
```



```
public void addPlayer(@NotNull Player player) {
    if (player.board == this && !players.contains(player)) {
        players.add(player);
        notifyChange();
    }
}

public Player getPlayer(int i) {
    if (i >= 0 && i < players.size()) {
        return players.get(i);
    } else {
        return null;
    }
}
```

Eksempel (fra Board)

...

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class Board extends Subject {
```

...

```
private List<Player> players = new ArrayList<Player>();  
private Player current;
```

List er faktisk et interface (og kan derfor ikke instantieres)!

ArrayList er en klasse som implementerer interface **List**.

ArrayList og **List** kommer fra **java.util**-pakken og skal importeres derfra.

Board er en modelklasse og skal derfor være et **Subject** som kan observeres.

...

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class Board extends Subject {
```

...

```
private List<Player> players = new ArrayList<Player>();  
private Player current;
```

ArrayList og List er **generiske**, dvs. de har en typeparameter (fx. **Player**) som siger hvilken type alle listens elementer skal have.

```
public void addPlayer(@NotNull Player player) {  
    if (player.board == this && !players.contains(player)) {  
        players.add(player);  
        notifyChange();  
    }  
}
```

```
public Player getPlayer(int i) {  
    if (i >= 0 && i < players.size()) {  
        return players.get(i);  
    } else {  
        return null;  
    }  
}
```

List (og dermed **ArrayList**) har en masse nyttige operationer:

- contains
- size
- add
- get
- remove
- ...

Se <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

```
for (int i = 0; i < players.size(); i++) {  
    System.out.println(players.get(i).getName());  
}
```

```
for (Player player: players) {  
    System.out.println(player.getName());  
}
```

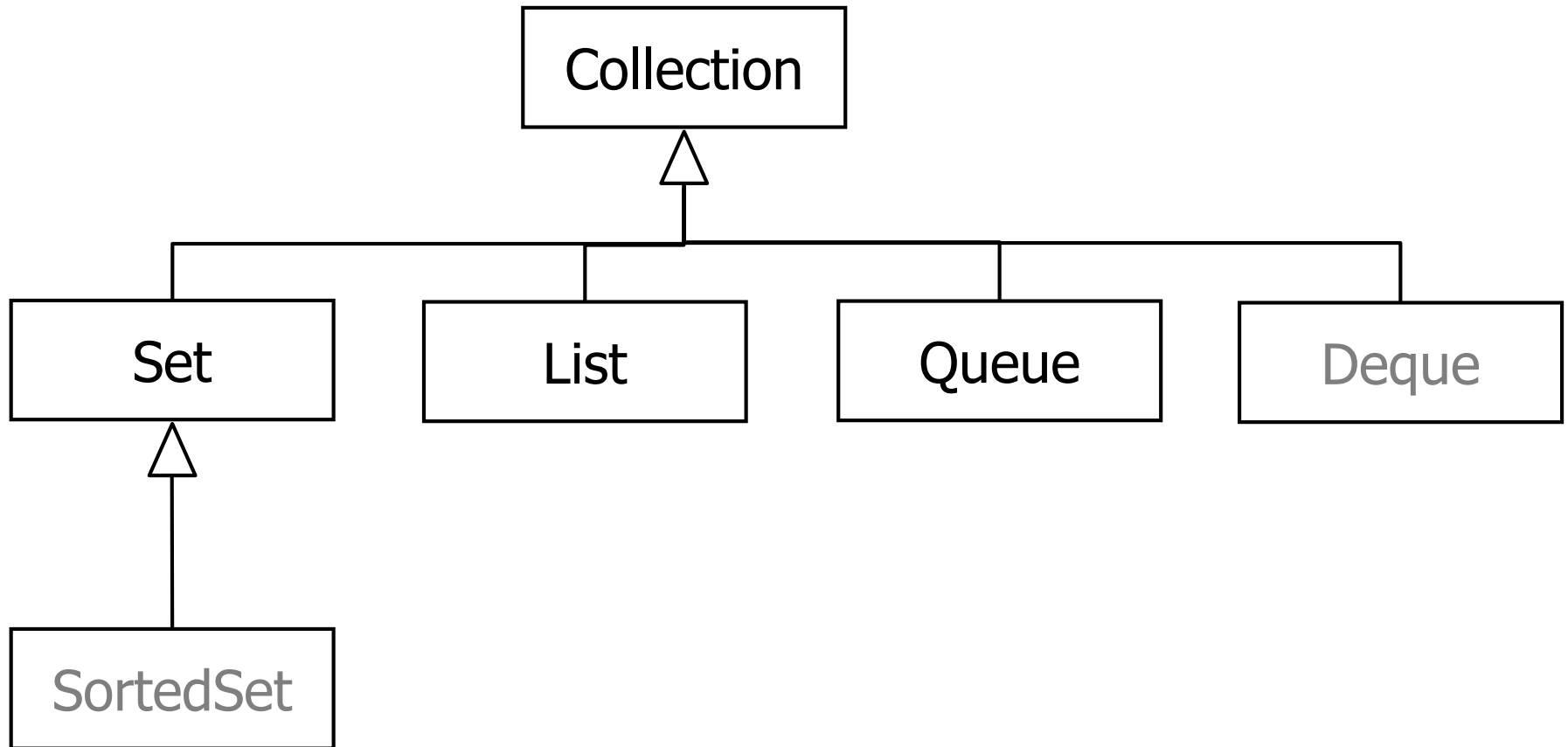
Denne form af løkken er mere elegant og sikkert. Men man må ikke direkte eller indirekte ændre listen imens løkken er i gang!

Ellers får man en fejl:
ConcurrentModificationException

Nogle gange bruges der også lister (eller andre samlinger) uden dens type parameter! Fx

```
List players = new ArrayList();
```

- Der var ingen generiske datatyper inden Java 1.5, så at man måtte programmere på en sådan måde.
- Alle elementer er "bare" objekter (**Object**).
- Det er meget besværligt og tilbøjeligt til at lave fejl!
- En generisk datatype, som bliver brugt uden dens typeparameter, kaldes også rå type (**raw type**)
- **Raw types skal I ikke bruge i jeres software! Ellers vil jeg høre et rigtigt godt argument!**



er en samling af objekter af en bestemt type **E** med følgende (hoved-)metoder:

```
void clear();  
boolean add(E e);  
boolean remove(Object o);  
int size();  
boolean isEmpty();  
contains(Object o);
```

Bemærk at for nogle implementering kan det være, at nogle metoder ikke er implementeret, fordi de ikke giver mening. Så kan de kaste en **UnsupportedOperationException**

Og mange flere:

Se <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

```
boolean addAll(Collection<? extends E> c)
```

```
boolean removeAll(Collection<?> c)
```

```
Iterator<E> iterator()
```

...

Det kommer vi tilbage til, når vi diskuterer `Iterator`.

indeholder det same element (ift. **equals()**) maximalt en gang (indeholder ikke duplikater)

- **add(e)** ændrer kun noget, hvis **e** ikke er allerede et element; men man må gerne kalde **add(e)** når der er allerede et element **e** (**contains(e)**)

Der er mange implementeringer af **Set**. En typisk implementering er **HashSet**.

Elementer i listen har en rækkefølge og kan tilføjes på en bestemt position. Nye (ift. **Collection**) metoder (udvalg):

```
void add(int index, E e);
```

```
E remove(int index);
```

```
E get(int index);
```

```
E set(int index, E e);
```

...

Der er mange implementeringer af **List**. Typiske implementeringer er **ArrayList**. og **Vector**.

elementer bliver tilføjet på den ene ende (*tail*) og fjernet på den anden ende (*head*). Det realiserer en kø.

Nogle nye metoder (udvalg):

E `element()` ;

E `peek()` ;

...

Der er mange implementeringer af `Queue`. En typisk implementering er `LinkedList`.

Begge returnerer elementet på head-siden af køen uden at fjerne det, hvis det eksisterer. Forskellen er, hvad der sker, hvis der ikke er et element i køen: `element()` kaster en exception, `peek()` returnerer `null`

Elementer kan bliver tilføjet og fjernet på begge ender af køen (*head* og *tail*).

"Deque" udtales som engelsk "deck".

Deque diskuterer vi ikke nærmere (se Java's dokumentation af collection).

Collections har en metode, som producer en iterator; iteratoren tillader så at iterere over alle elementer af selve kollektionen.

Iterator er en interface med to hovedmetoder:

boolean hasNext() ;

Returner **true** hvis iteratoren har stadig elementer tilbage.

E next() ;

Returner det næste element og så skifter et element videre. Hvis den kaldes, når **hasNext()** returner **false**, kastes en exception.

Med en iterator kan man iterere over alle dens elementer fx:

```
Collection<E> c = ...
```

```
Iterator<E> i = c.iterator();
```

```
while (i.hasNext()) {
```

```
    E e = i.next();
```

```
    // do something with e, e.g.
```

```
    System.out.println("> " + e);
```

```
}
```

Faktisk er det endnu nemmere at iterere over alle elementer af en Collection.

```
Collection<E> collection = ...
```

```
for (E element : collection) {  
    // do something with e, e.g.  
    System.out.println("> " + element);  
}
```


Konkret eksempel

```
Collection<Integer> collection = ...
```

```
int sum = 0;
```

```
for (Integer value : collection) {
```

```
    sum = sum + value;
```

```
}
```

Hvis I itererer over alle elementer af en collection på denne måde, så undgår I mange muligheder for fejl.

... men man må ikke ændre samlingen under selve løkken (se side 29)!

Det går endnu mere elegant med streams, som kom ind med Java 8:

<http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html>

... men det diskuterer vi ikke nærmere her.

- er et struktur som repræsenterer en partiel funktion (afbildning) fra værdier af klassen **K** (for key/nøgle) til værdier af klassen **V** (for value/værdi)
- **map.get(k)** returnerer den aktuelle værdi af **map** for nøglen **k**
- kan nemt opdateres med operation **m.put(k,v)**
- Siden Java 8 har maps også en *convenience method*: **map.getOrDefault(k,d)**

Returner **null**, hvis **map** ingen værdi har for nøglen **k**

Returner **d** (default), hvis **map** ingen værdi har for nøglen **k**

- Vi bruger en map af type **Map<String, Integer>** til at beregne hvor ofte et navn forekommer i en liste af navne (Strings)

- Vi antager:

```
List<String> names = new ArrayList<String>();  
names.add("Emilie");  
names.add("Jens");  
names.add("Tim");  
names.add("Tom");  
names.add("Kurt");  
names.add("Jens");  
names.add("Anna");  
names.add("Jens");  
names.add("Emilie");  
names.add("Jens"); ...
```

Der er mere elegante måder at danne en konstant liste som denne her. Fx

```
List<String> names =  
    Arrays.asList("Emilie", "Jens", "Tim", ... );
```

Map<K,V>: Eksempel

```
Map<String,Integer> nameCount =  
    new HashMap<String,Integer>();
```

Her bruger vi (meget typisk) **HashMap** implementeringen af **Map**

```
for (String name: names) {  
    nameCount.put(  
        name,  
        nameCount.get(name)+1 );  
}
```

Itererer over alle navne; virker da liste er en collection.

```
for (String name: nameCount.keySet()) {  
    System.out.println(name + ": " + nameCount.get(name));  
}
```

Adder 1 på den nuværende værdi for navnet og gem det under samme navn

Virker bare ikke!!

```
Map<String,Integer> nameCount =  
    new HashMap<String,Integer>();
```

```
for (String name: names) {  
    nameCount.put(  
        name,  
        nameCount.getDefault(name,0)+1 );  
}
```

Adder 1 på den
nuværende værdi for
navnet og gem det
under samme navn;
hvis der ingen værdi
er, så vælg 0 som
default.

```
for (String name: nameCount.keySet()) {  
    System.out.println(name + ": " + nameCount.get(name));  
}
```

HashMap<K, V> og nogle andre af Javas datastrukturer virker kun, hvis klassen **K** til nøgler (keys) er implementeret rigtigt: dvs.

- metoden **equals()** er implementeret så at den er en ækvivalentsrelation (se næste slides)
- **hashCode()** metoden er kompatibel med identitet (se næste slides)

Man siger at der er kontrakter for disse metoder. Og nogle af Javas datatyper (som fx **HashMap<K, V>**) virker kun hvis de der kontrakter er overholdt.

For alle objekter `o1`, `o2` og `o3` skal altid gælde:

- *Refleksivitet*:
`o1.equals(o1)`
- *Symmetri*:
hvis `o1.equals(o2)` så gælder også
`o2.equals(o1)`
- *Transitivitet*:
hvis `o1.equals(o2)` og `o2.equals(o3)`
så gælder også `o1.equals(o3)`

Der er nogle flere krav, men dem diskuterer vi ikke her (se Javadoc for `Object`).

For alle objekter `o1` og `o2` skal altid gælde:

- *Kompatibilitet:*

hvis `o1.equals(o2)` så gælder også
`o1.hashCode() == o2.hashCode()`

Når man ændrer metoden `equals()` i sine egne klasser, så skal man også tilpasse `hashCode()` metoden – især når man bruger dem som nøgler i strukturer som baserer på hashing.

Hvis HashMaps eller HashSets forholder sig "lidt mærkeligt" (fx. hvis de "glemmer" elementer man havde tilføjet før) i jeres programmer, så er en forkert implementering af `equals()` og `hashCode()` en meget sandsynlig årsag!

- Indtil videre har vi kun anvendt nogle indbyggede datatyper fra Java!
- Hvordan man selv programmere nogle datatyper (eller udvidelser deraf) selv vender vi tilbage til