Klassedesign:
Kobling og sammenhæng

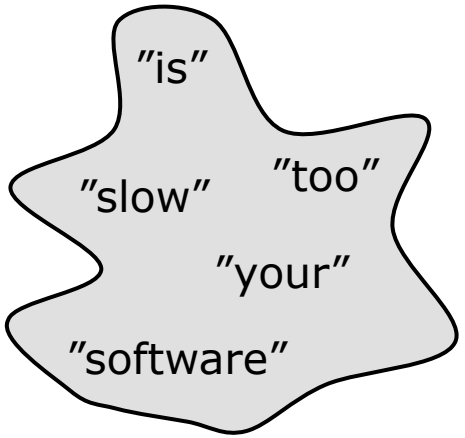
GRPRO: "Grundlæggende Programmering"

Forskellen på List, Set og Map

- **List:** Nummereret, dubletter tilladt

”your”	”software”	”is”	”too”	..

- **Set:** Uordnet, ingen dubletter



- **Map:** Afbildning nøgle→værdi

Nøgle	Værdi
”slow”	”I think this has to do with your hardware...”
”bug”	”Well, you know, all software has some bugs...”
...	...

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Implementation versus dokumentation/interface

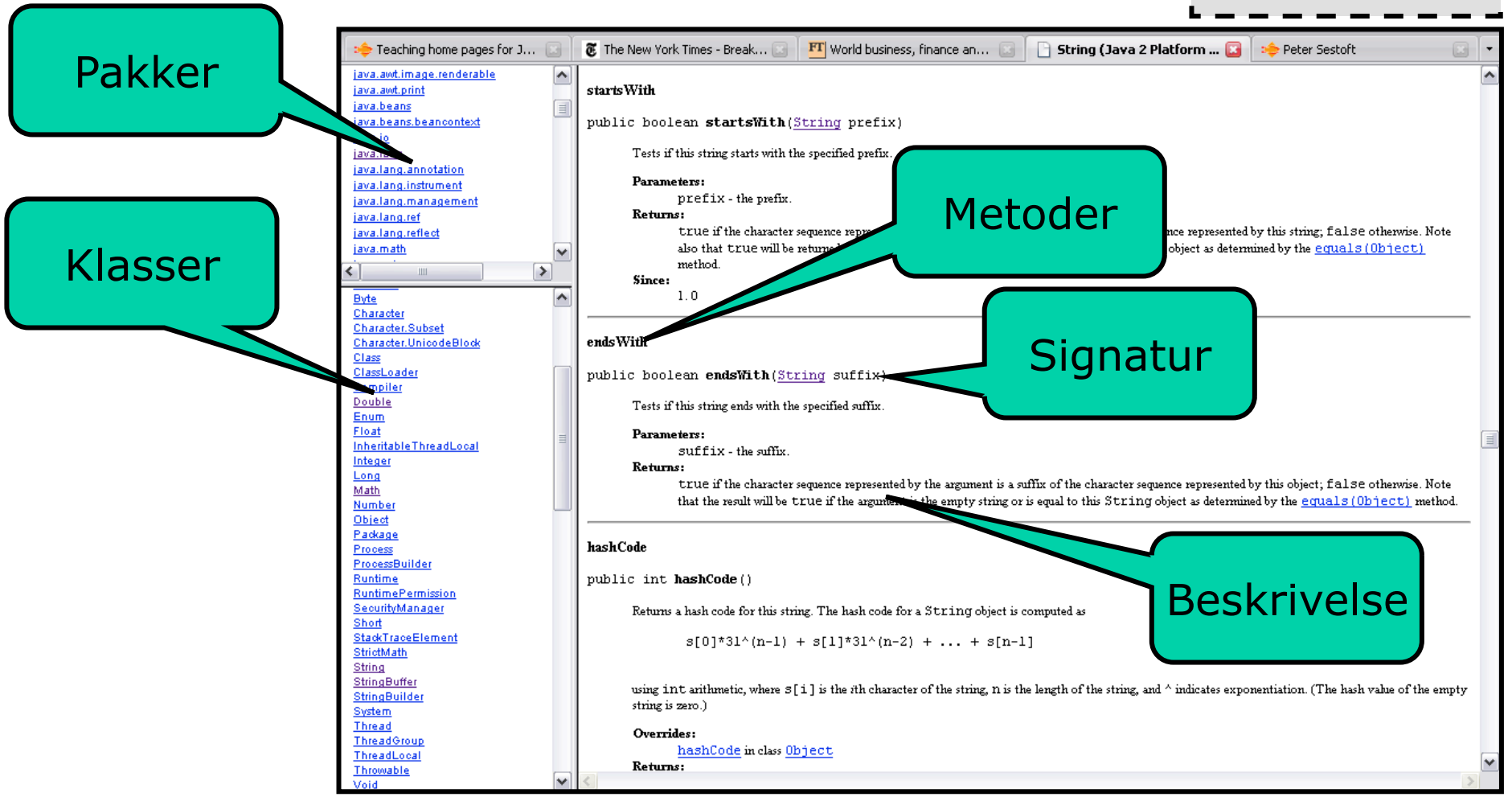
- Indtil nu har vi mest fokuseret på ***implementation*** (dvs. Java-kode)
- Og vi har set lidt ***dokumentation*** ('`JavaDoc`' for Java's klassebiblioteker)

- **Q:** Hvordan kan vi ***selv lave*** dokumentation (til vores Java-kode)?
- **A:** '`JavaDoc`' er en standard for at ***dokumentere*** Java-kode
  - (BlueJ har indbygget HTML-generator til JavaDoc)

Dokumentation til Javas klassebiblioteker

- **BlueJ:** Help -> Java Class Libraries

...or Google: **"Java API"**



JavaDoc for en klasse

- '**JavaDoc**':
  - har specielle ***kommentarer*** af form '**/****' og '***/**'
  - samt *"tags"* så som '**@author**' og '**@version**'

```
/**
 * Class BallDemo - provides two short demonstrations
 * showing how to use the Canvas class.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2006.03.30
 */

public class BallDemo {
    ...
}
```

Resulterende dokumentation for klasse BallDemo

Class BallDemo

```
java.lang.Object
  └ BallDemo
```

```
public class BallDemoextends java.lang.Object
```

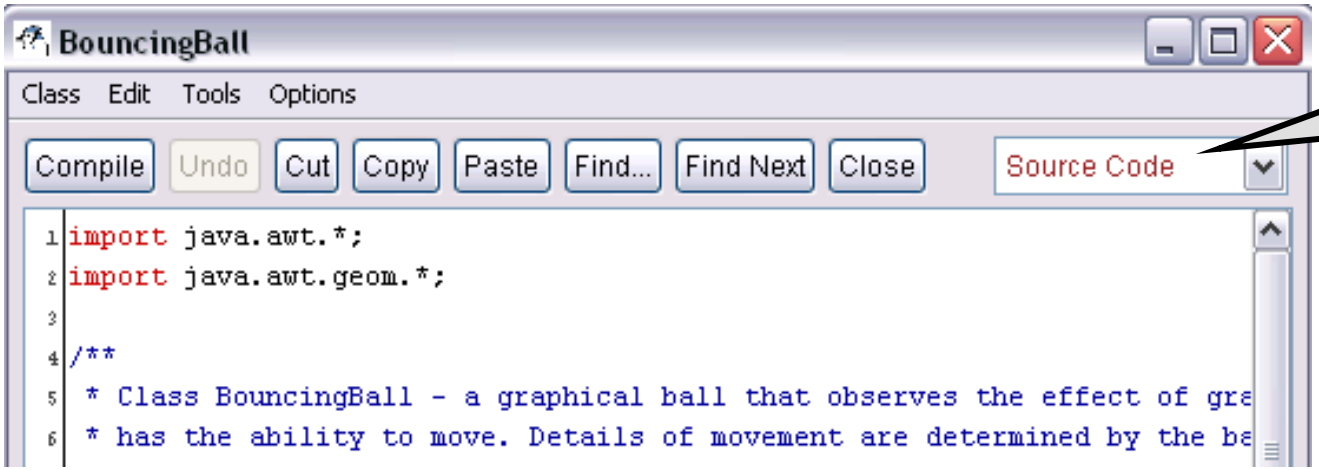
Class BallDemo - a short demonstration showing animation with the Canvas class.

Version:
2011.07.31

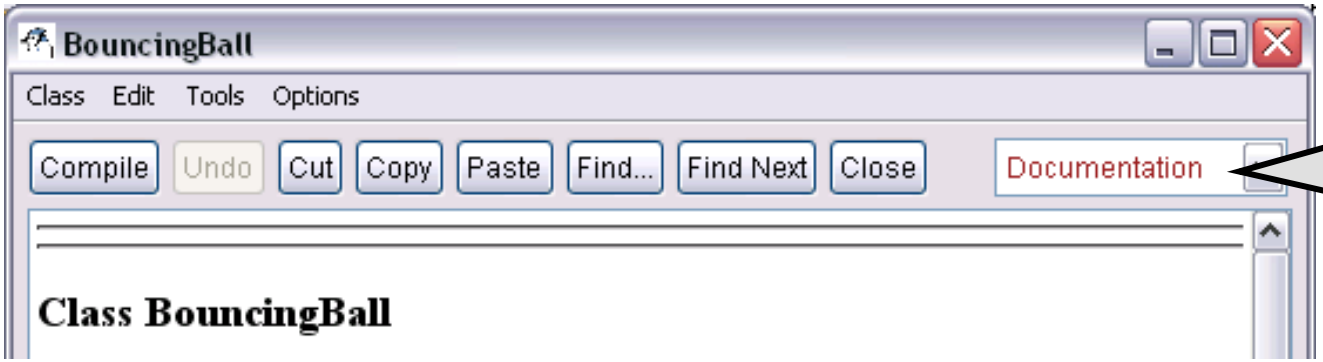
Author:
Michael Kölling and David J. Barnes

Generering af JavaDoc i BlueJ editor

- Skift fra ***Source Code*** til ***Documentation:***



Her skiftes syn (view)



Laves første gang man ser den samt efter ændringer !

JavaDoc for en metode

- Samme stil, andre tags: '@param' og '@return'

```
/**
  * The number of days from this date to the given end date.
  *
  * @param endDate The end date
  * @return      The number of days from this date to the
  *              given end date.  Zero if the dates are
  *              identical.  Negative if endDate precedes
  *              this date.
  */
public int daysTill(Date endDate) {
   return endDate.dayNumber() - this.dayNumber();
}
```

- Dokumentation bruger JavaDoc-kommentarer og metode-signatur men ikke metodekroppen

Resulterende dokumentation for metode daysTill (i Date)

daysTill

`public int` **`daysTill`**`(Date endDate)`

The number of days from this date to the given end date.

Parameters:

`endDate` - The end date

Returns:

The number of days from this date to the given end date. Zero if the dates are identical. Negative if endDate precedes this date.

```
/**
  * The number of days from this date to the given end date.
  *
  * @param endDate The end date
  * @return      The number of days from this date to the
  *              given end date.  Zero if the dates are
  *              identical.  Negative if endDate precedes
  *              this date.
  */
public int daysTill(Date endDate) {
   return endDate.dayNumber() - this.dayNumber();
}
```

Hvad er god dokumentation?

- **Målgruppe** for metodedokumentation:
  - Andre softwareudviklere
  - En selv (efter X måneder :-)

- **Succeskriterier:**
  - Klassen og dens metoder kan ***bruges*** uden at man ser på implementationen
  - Klassen og dens metoder kan ***testes*** uden at man ser på implementationen (se [B&K], Kap 7)
  - Implementationen kan ***forbedres*** og ***ændres*** uden at dokumentationen skal ændres
  - Dokumentationen afslører ***hvad***, men ***ikke hvordan*** (unødigt) implementationen ser ud**:**

- **Hvad** (dokumentation) -vs- **Hvordan** (implementation)

Gælder det for balls-projektet?

- For klasse '**`BouncingBall`**' burde vi kunne forstå implementationen (source code)
- Men klasse '**`Canvas`**' er mere kompliceret
- Her kan vi forhåbentlig nøjes med at se på dokumentationen

Indkapsling: `private` og `public`

- **Generelle designprincipper:**
  - Et objekt skal udelukkende vise omverdenen (andre objekter) hvad der er absolut nødvendigt for dem
  - Vis hellere metoder (adfærd) end felter (tilstand)

- **Fordi:**
  - så bliver andre ikke afhængige af objektets indre detaljer
  - dermed lettere at lave forbedringer
  - og lettere at rette fejl

- **Bemærk:**
  - Der kan være flere implementationer svarende til en og samme dokumentation (interface)!

Felt `private`, metode `public`

- Ofte laver man et felt *private*, og så laver man en *public* ***accessor*** (aka, "get-metode") til feltet:

```
public class BouncingBall {
  private int xPosition;
  ...
  public int getXPosition() {
    return xPosition;
  }
```

- **Q:** Når vi har en accessor (`getXPosition()`), kan feltet (`xPosition`) så ikke lige så godt være public?

- **Nej!, ...:**
  - Nu kan feltet ***læses udefra***, men ***ikke ændres udefra***
  - Internt i klassen kan man så ***skifte repræsentation***; fx fra antal pixels (int) til et tal mellem 0 og 1 (double)

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Et `static` felt hører til klassen

- Et normalt felt hører til et bestemt ***objekt***
- Et **`static`** felt hører til ***klassen***
  (dvs. er fælles for alle klassens objekter):

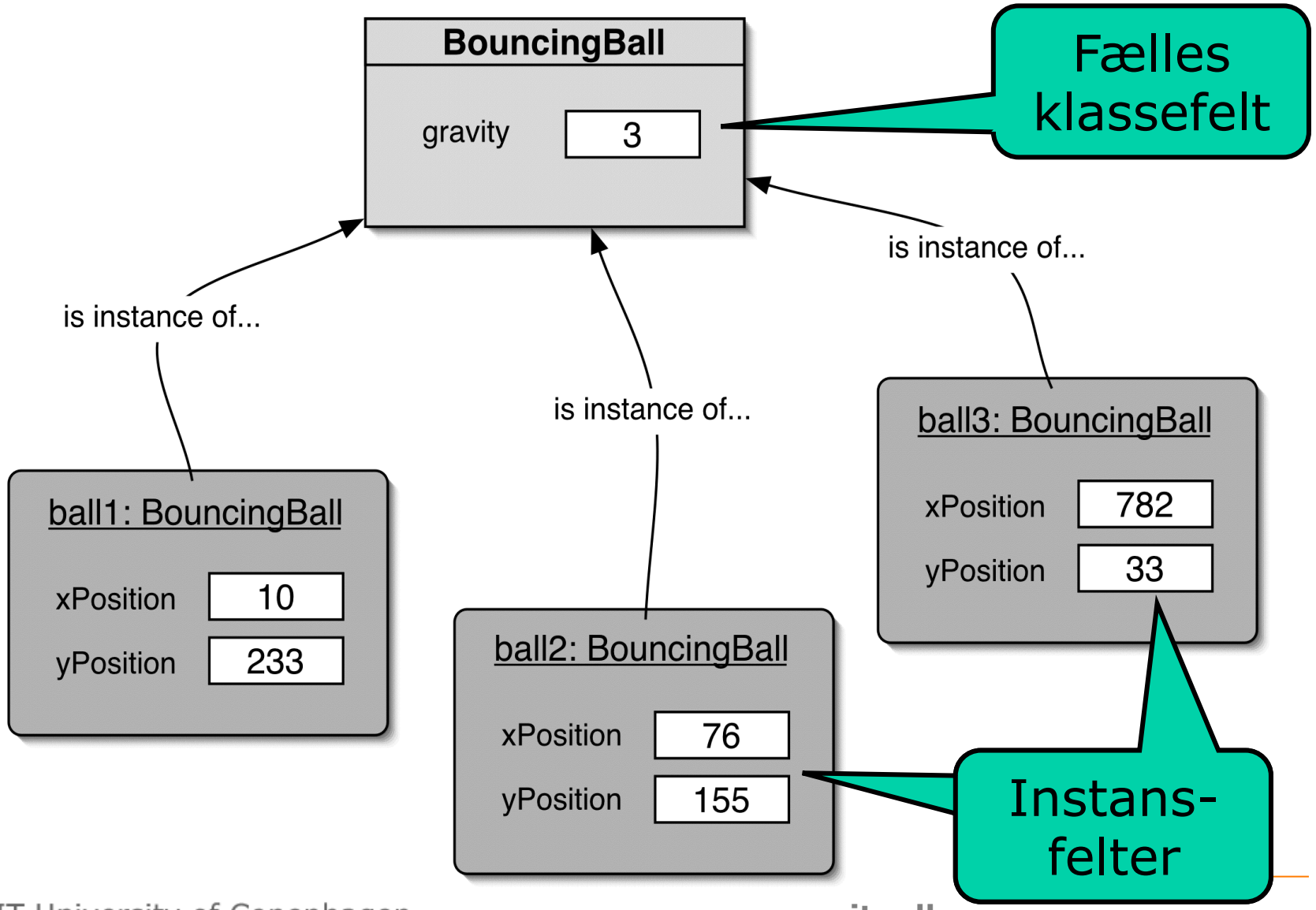
```
public class BouncingBall {
   static int GRAVITY = 3;
   int xPosition;
   int yPosition;
   ...
}
```

- Et **`static`** felt tilgås ved at skrive:
  klassens navn dot feltets navn:

```
int gravity = BouncingBall.GRAVITY;
System.out.println(gravity);
...
BouncingBall b = new BouncingBall(...);
...
```

- **NB:** Et instansfelt kræver et objekt - et **`static`** felt gør ikke!

Static felter = klassefelter, versus instansfelter



Klassemetoder

- Nu har vi set ***statiske felter***
  (aka, "klasse-konstanter"):

```
private static final int GRAVITY = 3; // effect of gravity!
```

- Der findes også ***statiske metoder***
  (aka, "klasse-metoder"):

- *Klassemetoder* er metoder:
  knyttet til klasser i stedet for objekter

Klassemetoder (definition og brug)

- Klassemetoder erklæres vha. ordet **`static`**:

DEF

```
class Calendar {
    ...

    public static int getNumberOfDaysThisMonth() {
        ...
    }
}
```

- Og kaldes ved reference til klassenavnet:

USE

```
...
int x = Calendar.getNumberOfDaysThisMonth();
...
```

- **Bemærk:** Definitioner af klassemetoder kan ikke bruge alm felter og metoder, kun andre statiske felter og metoder! **Hvorfor?**

Statiske felter i Java API

- Fx har '**`System`**' det statiske felt **`out`**:

```
class System {
    static PrintStream out;

    ...
}
```

- (Objektet vi tidligere har kaldet '**`println(..)`**' på.)

- Klassen '**`Math`**' har også statiske felter:

```
class Math {
    static double E;
    static double PI;

    ...
}
```

Statiske metoder i Java API

- Klassen '**Math**' har også statiske metoder:

```
class Math {
   static double E;
   static double PI;

   static double abs(double a) { ... }
   static double cos(double a) { ... }
   static double log(double a) { ... }
   static double pow(double a) { ... }
   static double sin(double a) { ... }
   static double sqrt(double a) { ... }

   ...
}
```

- (Bogen benytter ikke så mange statiske metoder)

Afvikling udenfor BlueJ

- Opret statisk metode '**main**' med signatur:

```
public static void main(String[] args) {
    // this code gets executed when the program is run!
    Game game = new Game();
    game.play();
}
```

- Gå til projektets directory og skriv:

```
%> java Game
```

- ...på en kommandolinie efter programmet er oversat (compilet) hvilket gøres ved:

```
%> javac Game.java
```

- Mere information:
  - **[B&K]** Appendix E
  - ...samt Google :-)

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

'final': Uforanderlige felter

- Et **final** felt kan ikke ændres!
- Det skal initialiseres ved **erklæringen** eller i **konstruktoren**

```
public class BouncingBall {
   private final int diameter;
   private int xPosition, yPosition;
   ...

   public BouncingBall(int diameter, ...) {
      this.diameter = diameter;
      ...
   }
}
```

- (Lav et felt **final** hvis det er en uforanderlig egenskab ved objektet)

Konstanter: static og final

- Tyngdekraften er både fælles for alle bolde (så **`static`**), og den er konstant (så **`final`**):

```
public class BouncingBall {
  private static final int GRAVITY = 3;
  private final int diameter;
  ...
}
```

diameteren er konstant derfor final

men ikke fælles for alle bolde, derfor *ikke* static

public final, er det OK?

- Et `public final` felt kan læses, ikke ændres
- Det giver god mening fx for et dato-objekt, hvor år, måned og dag er de oplagte felter:

```
public class Date {
  public final int year;
  public final int month;
  public final int day;

  public Date(int year, int month, int day) {
    this.year = year; …
  }
```

- Men selv her argumenterer mange for:
  private felter og public get-accessor metoder

- **Q:** Har de en pointe?

NB: 'final' på array betyder muligvis ikke helt hvad man skulle tro

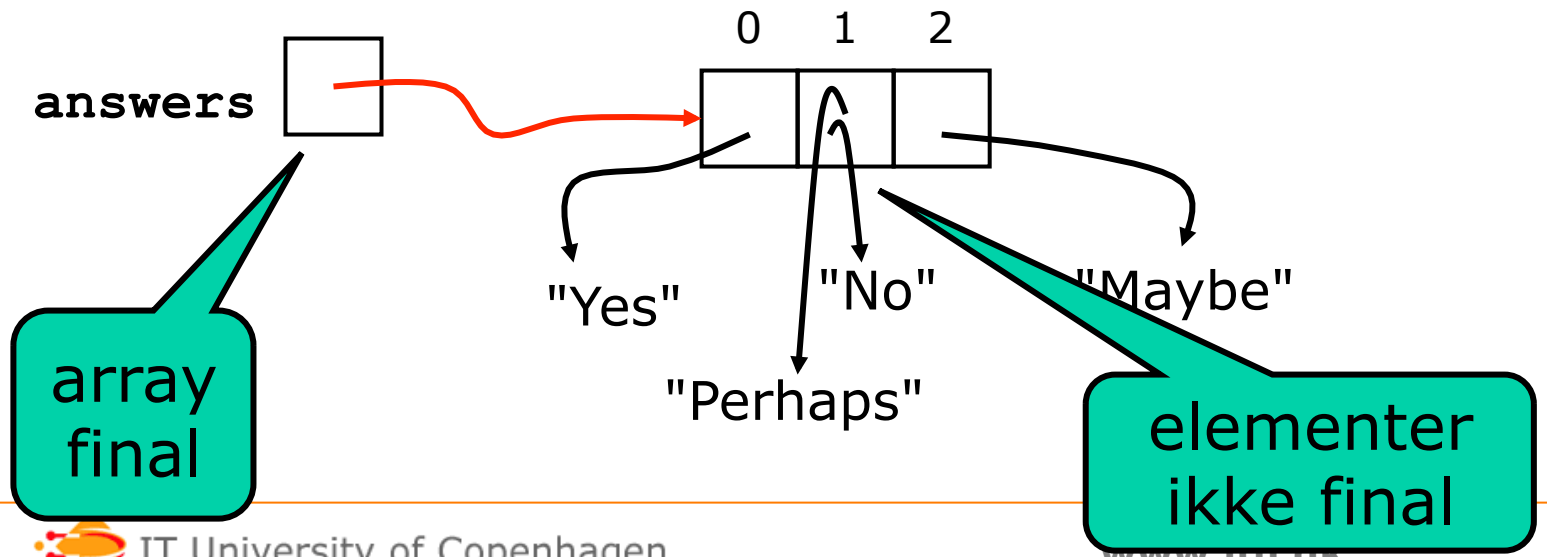
- Et `final` felt af array-type kan ikke ændres:

```
private final static String[] answers
   = { "Yes", "No", "Maybe" };
```

```
answers ✗= { "Ja", "Nej", "Tjaaa" };
```

- ...men arrayets ***indhold kan ændres:***

```
Answer.answers[1] ✓= "Perhaps";
```



A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Vigtigste læringsmål for denne uge

Efter denne uge skal du kunne...:

- ***Identificere*** og ***forbedre*** flg. eksempler på dårlig kode:
  - ***Kode-duplikering***
  - ***Høj kobling***
  - ***Lav sammenhæng***
- ***Refaktorisere*** et programdesign mhp udvidelse

Motivation

- ***Målsætninger:***
  - Programmer skal kunne læses af andre
  - Programmer skal let kunne opdateres og udvides
  - Det skal være let at rette fejl

- ***Konsekvenser ved dårligt design:***
  - Svært at vide hvor man skal starte hvis man vil tilføje funktionaliteter
  - Svært at vide om en fejlretning er komplet

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Opdatering

- ***Software der ikke opdateres uddør!***
  - Skal tilpasses ny teknologi
  - Funktionaliteter skal tilføjes
  - Nye regler og krav
  - Nyt design

- ***Tænk fremad***

Kodeduplikering (duplication)

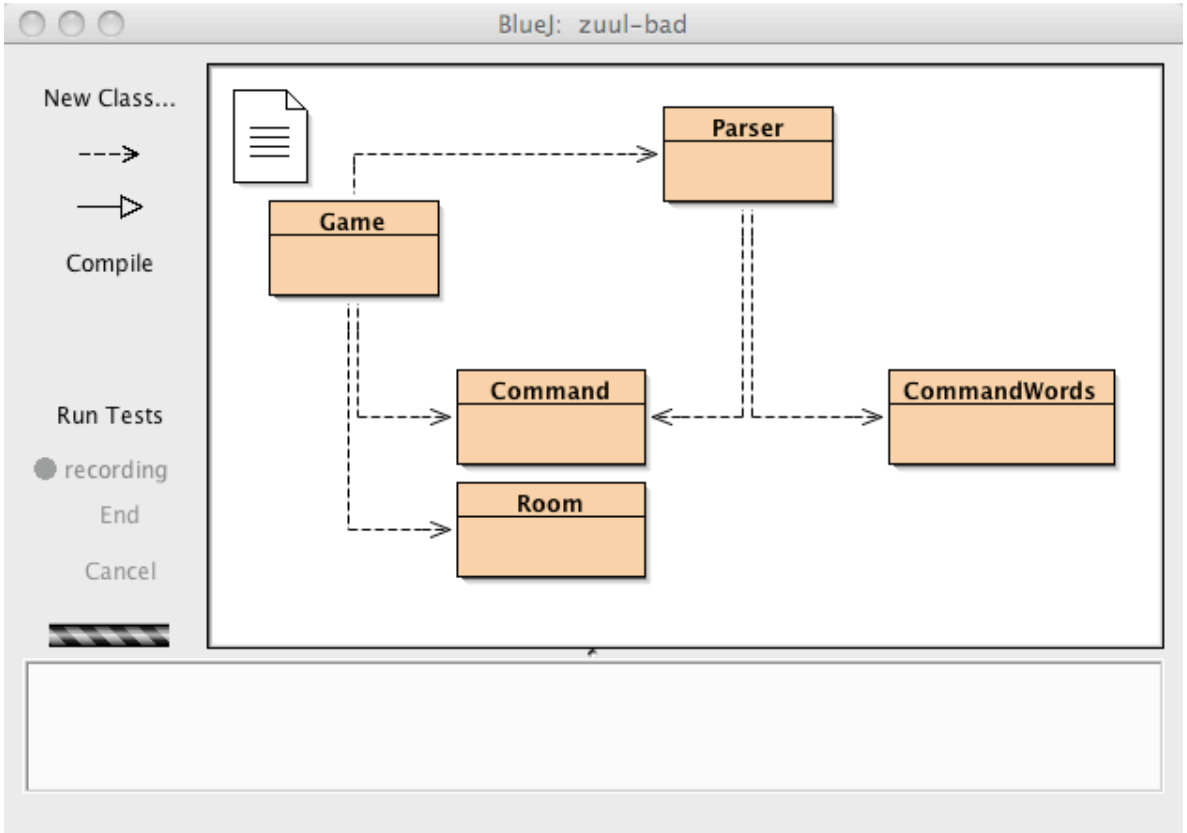
- ***Redundans:*** samme kode i flere metoder!

- ***Problemer:***
  - Dobbelt arbejde
  - Svært at fejlrette (2 steder)
  - Svært at opdatere (2 steder)
  - Inkonsistens?

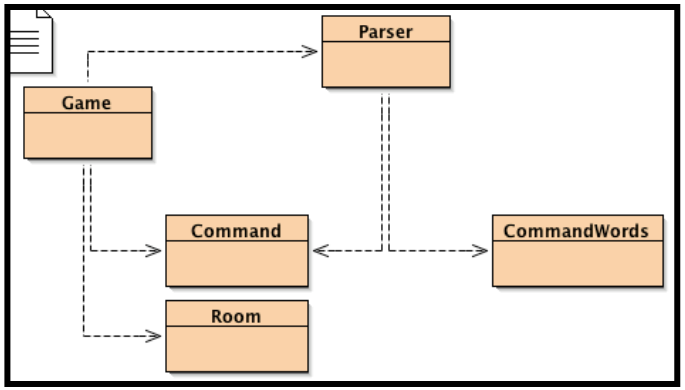
- **Almindelig løsning:**
  - Erklær ny metode ("Sharing")

World-of-zuul eksempel

- **World-of-zuul:**



Klasser i zuul-bad :-(



- **Game**: Opretter spilverden, holder spil kørende, processerer ordrer, ved hvor spilleren er
- **Room**: Repræsenterer et værelse. (Kender udgange.)
- **Parser**: Tager ordrer fra kommandolinien.
- **Command**: Repræsenterer en kommando. (En kommando består af et eller to ord.)
- **CommandWords**: Kender kommandonavne.

Kodeduplikering

- Redundans i **`printWelcome()`** og **`goRoom()`**:

```
System.out.println("You are " + currentRoom.getDescription());
System.out.print("Exits: ");
if (currentRoom.northExit != null) {
   System.out.print("north ");
}
if (currentRoom.eastExit != null) {
   System.out.print("east ");
}
if (currentRoom.southExit != null) {
   System.out.print("south ");
}
if (currentRoom.westExit != null) {
   System.out.print("west ");
}
System.out.println();
```

- **Q1:** Hvorfor er det et problem?
- **Q2:** Hvad kan vi gøre?

Løsning

- Løsning:

```
private void printLocationInfo() {
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if (currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if (currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if (currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if (currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

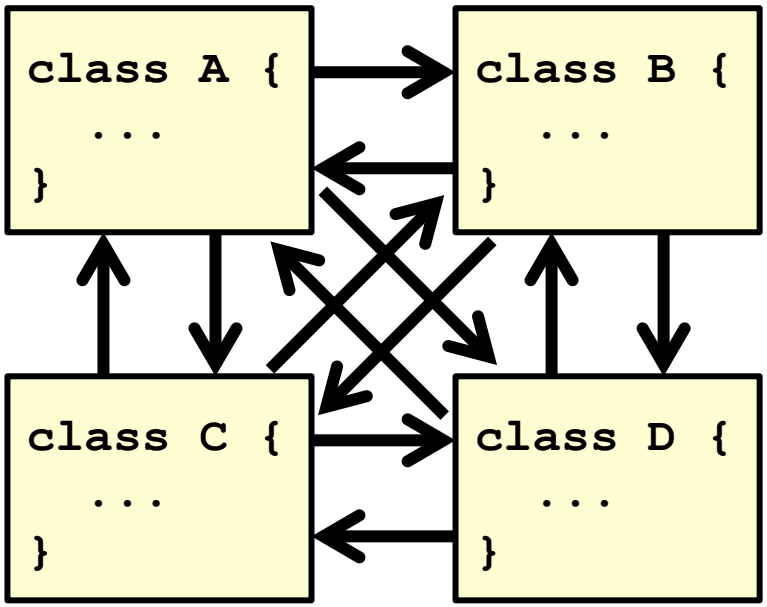
- Kaldes fra **`printWelcome()`** og **`goRoom()`**

A G E N D A

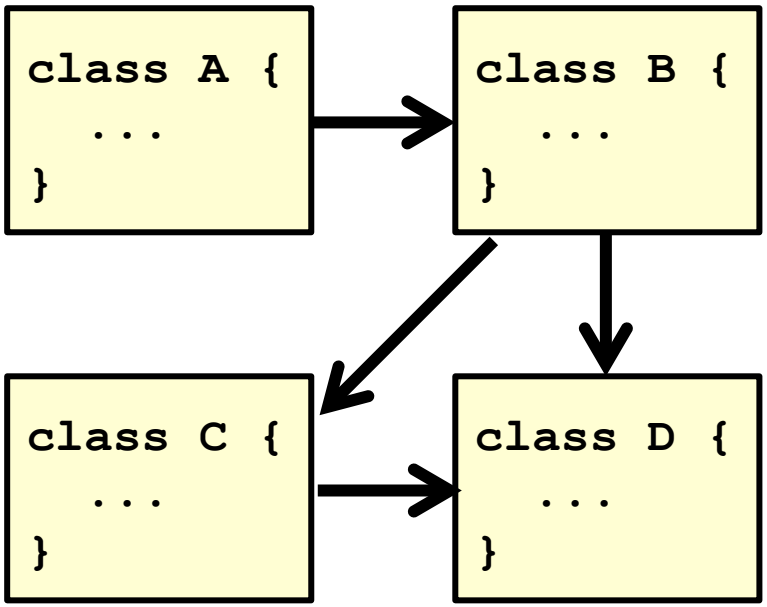
- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Lav Kobling (Coupling)!

Høj kobling:

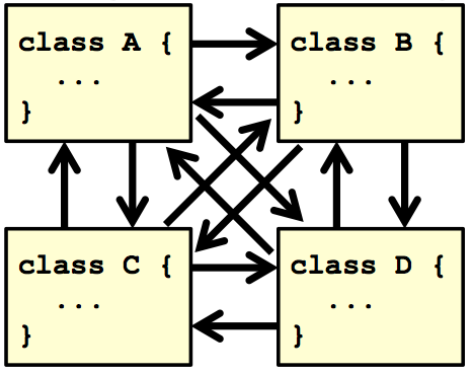


Lav kobling:

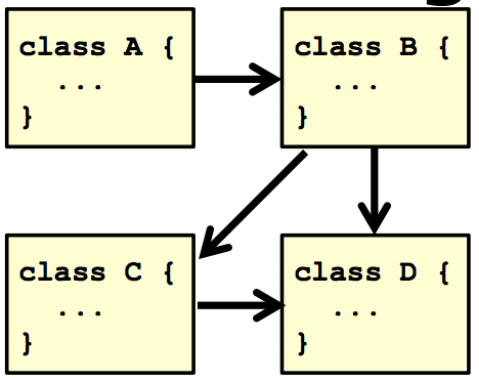


Kobling

Høj kobling:



Lav kobling:



- **Mål for hvor tæt forbundne klasser er!**
- En vis grad af kobling er nødvendig
- Lav kobling er ønskværdig
- Problemer med høj kobling:
  - Ændringer og fejlretning er svære at lokalisere
  - Programmer bliver uoverskuelige
  - Alt involverer hurtigt mange forskellige klasser
  - Mange afhængigheder

Høj kobling i eksemplet

- **`Game`** bruger ***felter i*** **`Room`** klassen:

```
if (direction.equals("north")) {
    nextRoom = currentRoom.northExit; // field access :-(
}
```

- **Problem?**
  - **`Game`** ***afhænger af implementationen af*** **`Room`!**
- **Konsekvenser:**
  - Hvordan kan vi ***tilføje andre retninger*****?**
  - Hvad skal vi gøre hvis vi vil holde udgange i ***anden datastruktur*** i `Room` klassen (f.eks. **`HashMap`**)**?**

Gem implementationen!

Løsning: ***"Gem implementationen":***

- **1)** Felter bør altid være private **!**
- **2)** Tilgang af felter bør kun ske gennem (public) metodekald **!**

Indkapsling i `Room`

- Tilføj metode **`getExit()`** til **`Room`** klassen
  - Ny implementation af **`Game`** bruger:

```
Room nextRoom = null;
if (direction.equals("north")) {
    nextRoom = currentRoom.northExit;
}
if (direction.equals("east")) {
    nextRoom = currentRoom.eastExit;
}
if (direction.equals("south")) {
    nextRoom = currentRoom.southExit;
}
if (direction.equals("west")) {
    nextRoom = currentRoom.westExit;
}
```


```
Room nextRoom =
    currentRoom.getExit(direction);
```

- (**Tilsvarende:** Tilføj også metode **`getExitString()`** til **`Room`** klassen)

Forbered på nye udgange

- **`setExits()`** metoden i **`Room`** tillader ikke nemt at tilføje nye udgange:

```
// initialize room exits:
outside.setExits(null, theater, lab, pub);
theater.setExits(null, null, null, outside);
pub.setExits(null, outside, null, null);
lab.setExits(outside, office, null, null);
office.setExits(null, null, null, lab);
```

- Bedre med kald af formen:

```
// initialize room exits:
lab.setExits("north", outside);
lab.setExits("east", office);
```

- **`Game`** er nu ikke nær så afhængig af **`Room`**!

Ny implementation af `Room`

- Brug **`HashMap`** til at holde styr på udgange:

```
public class Room {
   private String description;
   private HashMap<String,Room> exits;
   ...
}
```

- **(NB:** Vi kunne ikke have gjort dette så længe **`Game`** bruger felter fra **`Room`)**

- Nu er det også nemt at tilføje nye udgange (fx. "up", "down", "northeast", "southwest", ...)

```
// initialise room exits
lab.setExits("northeast", fredagsbar); // :-)
lab.setExits("up", balcony);
```

Opgaver

[M&K], 6.6:

- **i)** Tilføj get-metode `getExit()` til **Room** klassen og lav alle felterne private.
- **ii)** Opret `getExitString()` i **Room** klassen som lister alle mulige udgange fra et rum

[M&K], 6.7:

- Kald `getExitString()` i `printLocationInfo()` i **Game** klassen

[M&K], 6.8:

- Brug `HashMap` til at holde styr på udgange i **Room** klassen

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Ansvars-drevet design

- **Hver klasse har et ansvar**
- Ansvar kan handle om:
  - at vide ting
  - at gøre ting

- ***"Each class should be responsible for handling its own data"***

- **Fx:** I eksemplet havde **`Room`** ansvar for at kende sine udgange og bør derfor også have ansvaret for fx at liste dem

Ansvar, eksempel

- I eksemplet udskriver **`Game`** en beskrivelse af rummet:

```
currentRoom = nextRoom;
System.out.println("You are " + currentRoom.getDescription());
System.out.println(currentRoom.getExitString());
```

- Således:

```
> go north
You are outside the main entrance of the university
Exits: east south west
```

essentially a description of a room

- **Q:** Er det hensigtsmæssigt?
  - Hvad hvis vi nu ville tilføje andre ting til et rum?: (items?, monsters?, andre spillere?, ...?)

Implicit kobling

- Kobling er ikke altid åbenlys som i tilfældet med offentlige felter
  - (I eksemplet ville man altid opdage koblingen hvis man prøvede at ændre implementationen af `Room`)

- ***Implicit kobling*** vil sige at en klasse afhænger af en anden (men ikke i form af metodekald eller andet som let kan opdages – deraf navnet "implicit")
- Implicit kobling er lumsk og kan give anledning til fejl, der ofte ikke opdages med det samme.

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Sammenhæng (cohesion)

- ***Høj sammenhæng for klasser*** betyder at hver klasse har ***et velafgrænset og sammenhængende ansvarsområde***
- En klasse bør svare til netop én type entitet

- ***Høj sammenhæng for metoder*** betyder at hver metode gør netop én ting

- ***Konsekvenser af høj sammenhæng:***
  - Øget læselighed
  - Bedre mulighed for kode-genbrug
  - Det metoden gør burde kunne afspejles i navnet
  - Højere design-stabilitet

Sammenhæng af metoder

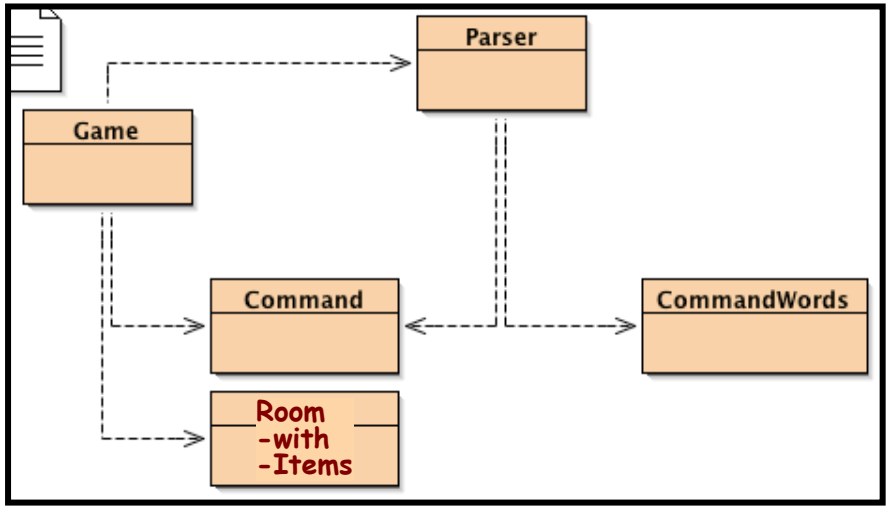
- **Eksempel:** Metoden **`play()`** kunne indeholde koden for **`printWelcome()`**

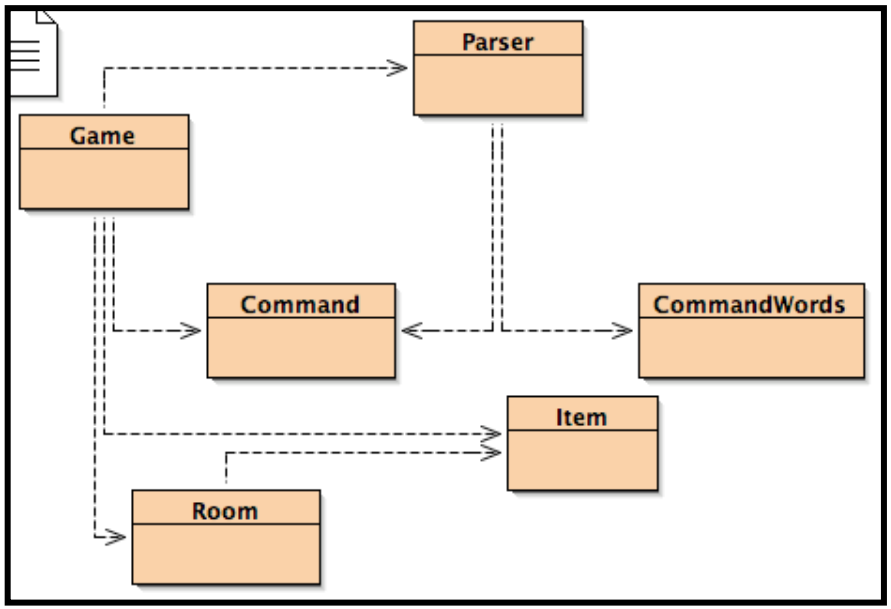
- **Q:** Hvad ville konsekvenen af det være?

Eksempel

- **Eksempel:** Tilføj ting (items) til spillet:
  - En ting har en beskrivelse og en vægt
  - Et rum skal kunne indeholde en ting
  - Hvad skal ændres i klassedesignet?

- **Tænk fremad...!**





A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Opsummering

- Godt klassedesign er:
  - Når program er let for andre at læse
  - Når program let kan opdateres og udvides
  - Når fejl let kan rettes

- Følgende begreber karakteriserer godt design
  - **Ingen kode-duplikering**
  - **Lav kobling**
  - **Høj sammenhæng**

A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Refaktorisering

- Refaktorisering betyder at omstrukturere kode (inkl klasse-design)
- Kan være nødvendigt ved implementering af ny funktionalitet
- Sker ofte hvis klasser eller metoder er blevet for store/tunge/besværlige/...
- Ofte erklæres nye klasser og ansvar flyttes til disse fra gamle klasser

Processen

- Refaktorering bør altid ske i ***to skridt:***
  - **1)** Først ***ændres*** til nyt klassedesign og gammel funktionalitet ***genoprettes***
  - **2)** Kun derefter ***tilføjes ny funktionalitet***

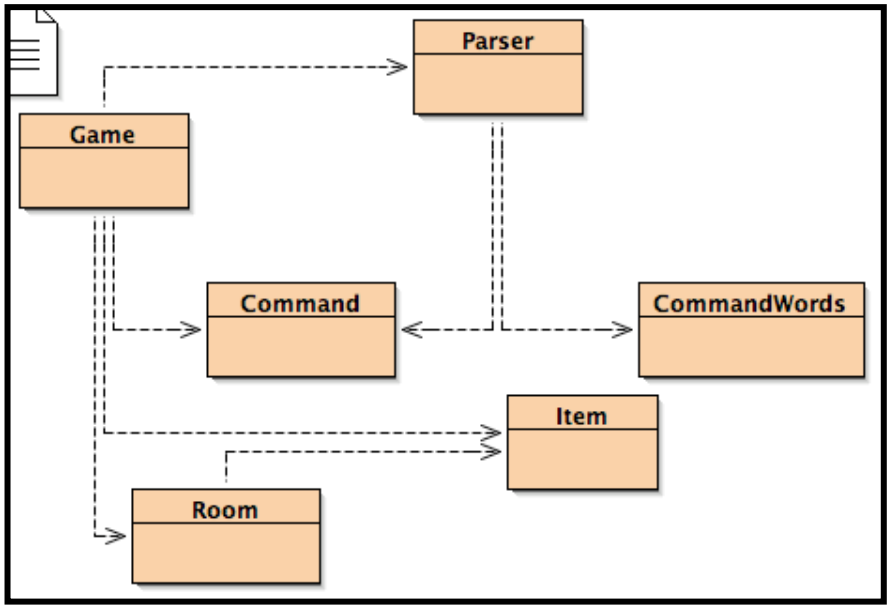
- **Q:** Hvad bør man ***huske*** efter man har lavet en ændring?

- Man bør udføre ***test*** efter første skridt
- (Ofte skal der skrives nye test-cases)

Udvidelser af zuul

- Vi ønsker at spillere skal kunne samle genstande (items) op og lægge dem fra sig

- **Q:** Kan vi gøre dette med det nuværende design?



A G E N D A

- **Impl -vs- Dokumentation (og JavaDoc)**
- **Statiske felter og metoder**
- **Final felter**
- **Kodeduplikering** (code duplication)
- **Kobling** (coupling)
- **Ansvars-drevet design**
- **Sammenhæng** (cohesion)
- **Refaktorisering** (refactoring)
- **Enum klasser**

Optællingstyper (enum types)

- Enum typer er typer (klasser) med en lille samling foruddefinerede elementer

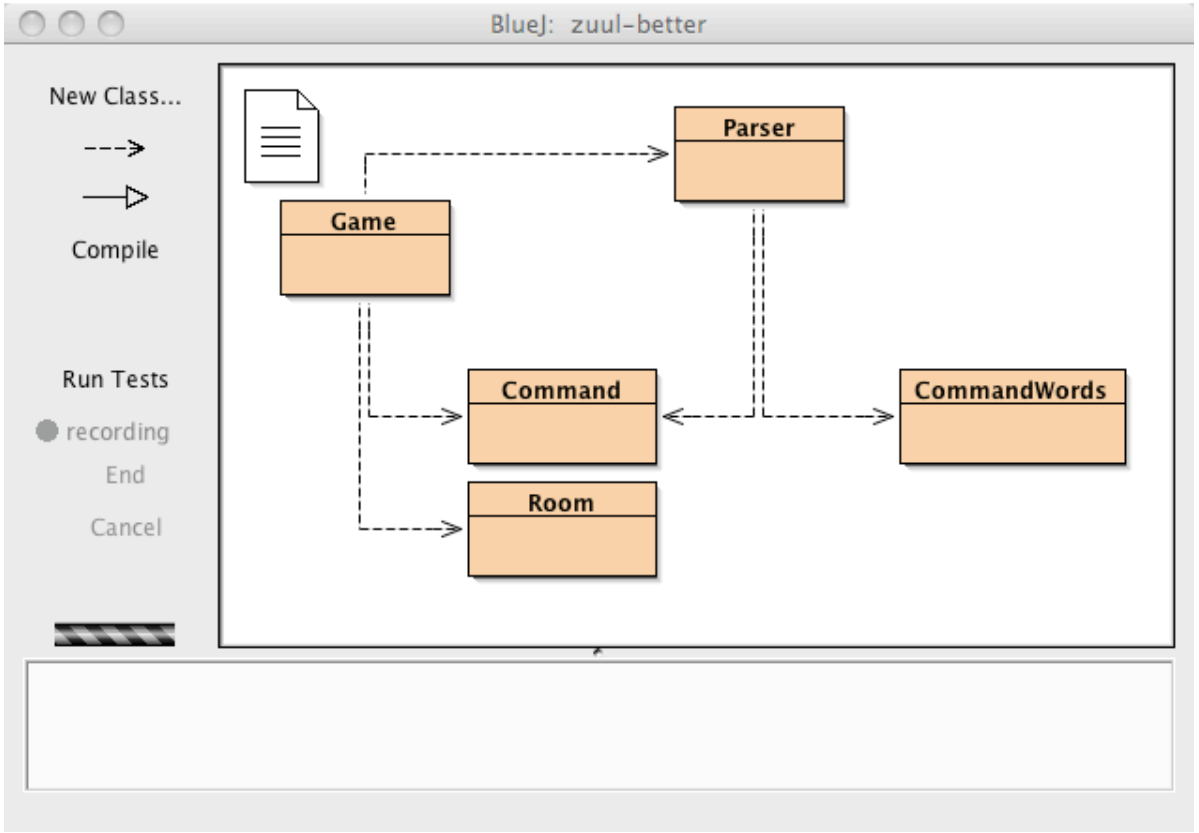
- **Defintion:**

DEF
```
public enum Month {
  JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC;
}
```

- **Brug:**

USE
```
Month month;
...
if (month == Month.JAN) System.out.println("It's January.");
```

Kommandoer i zuul



Kommando-navne i zuul

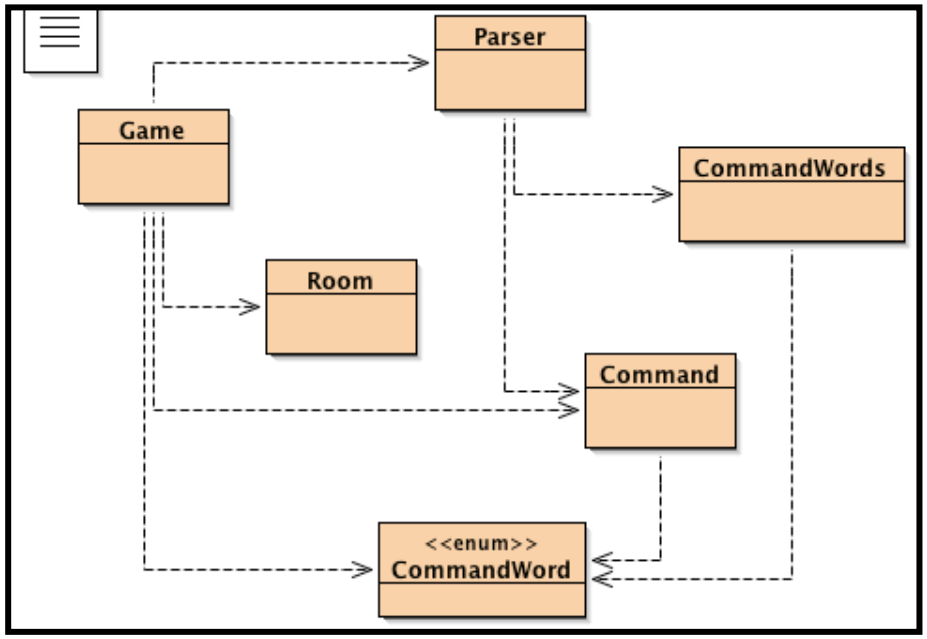
- Navne på kommandoer optræder eksplicit (som strenge) i både **`Game`**...:

```
if (commandWord.equals("help")) {
    printHelp();
} else if (commandWord.equals("go")) {
    goRoom(command);
} else if (commandWord.equals("quit")) {
    wantToQuit = quit(command);
}
```

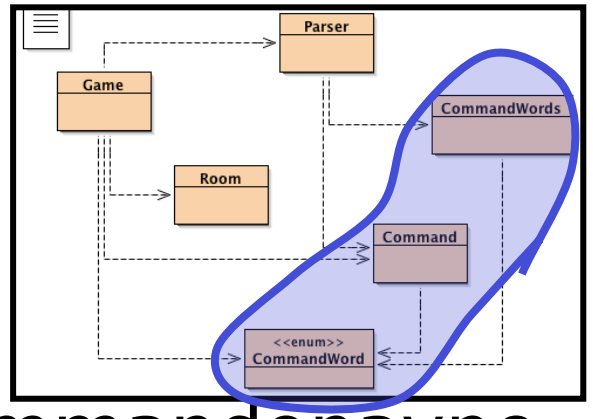
- ...og **`CommandWords`**:

```
private static final String[] validCommands = {
    "go", "quit", "help"
};
```

Enum typer i zuul



Zuul-with-enums-v1



- **`CommandWord`**: Enum-type. Repræsenterer programmets ***interne*** repræsentation af kommandonavne

- **`Command`**: Repræsenterer kommandoer. En kommando består af et commandWord og et andet ord i kommandoen

- **`CommandWords`**: Kender relationen mellem intern repræsentation af kommandonavne og den spilleren/brugeren kender

Enum typer

- Enum typer er klasser hvor der kun findes en foruddefineret (og endelig) mængde af mulige værdier

- Ofte kan man bruge strenge i stedet for enum typer:
  - Men oversætteren vil ikke opdage fejlen:

```
if (commandWord.equals("halp")) {
    printHelp();
}
```

  - Hvorimod den vil opdage:

```
if (commandWord == CommandWord.HALP)) {
    printHelp();
}
```

Opgave

- **Q:** Hvor let/svært ville det være at skifte sprog for kommandoer til dansk?
  - Uden enums?
  - Med enums?

Tak

Spørgsmål?