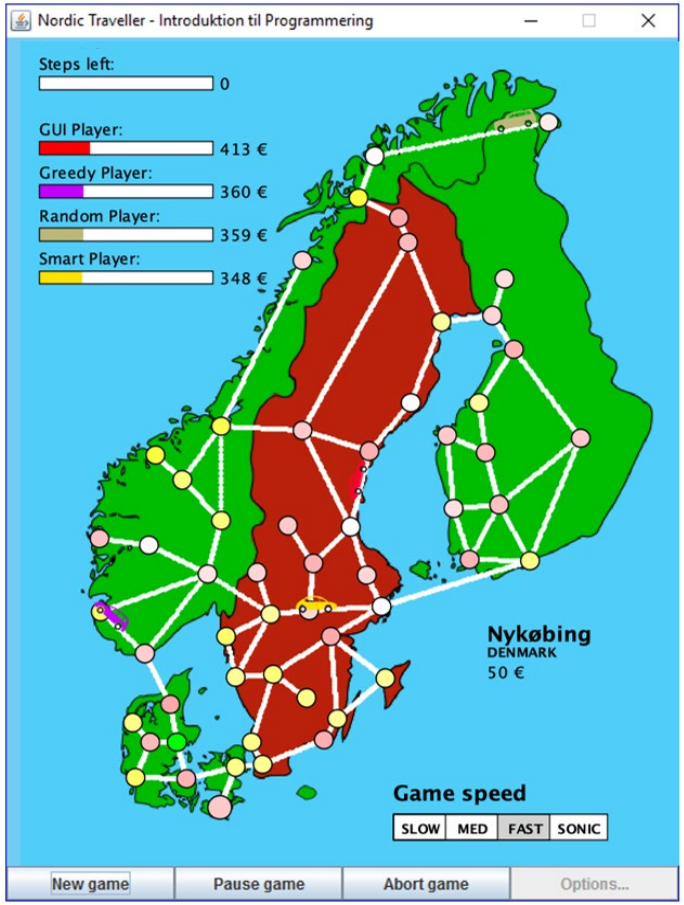
Forelæsning Uge 11

- **Nedarvning**
  - En klasse kan være en subklasse af en anden
  - Det betyder at subklassen arver feltvariabler og metoder fra klassen
- **Object klassen**
  - Indeholder en række nyttige metoder, bl.a. toString, equals, hashCode og getClass
  - Alle klasser er subklasser af Object klassen, hvorfor de arver ovenstående metoder
- **Protected access**
  - Alternativ til public og private
- **Projektopgave om computerspil**
  - I kursets sidste fem uger skal I (sammen med jeres makker) programmere et computerspil

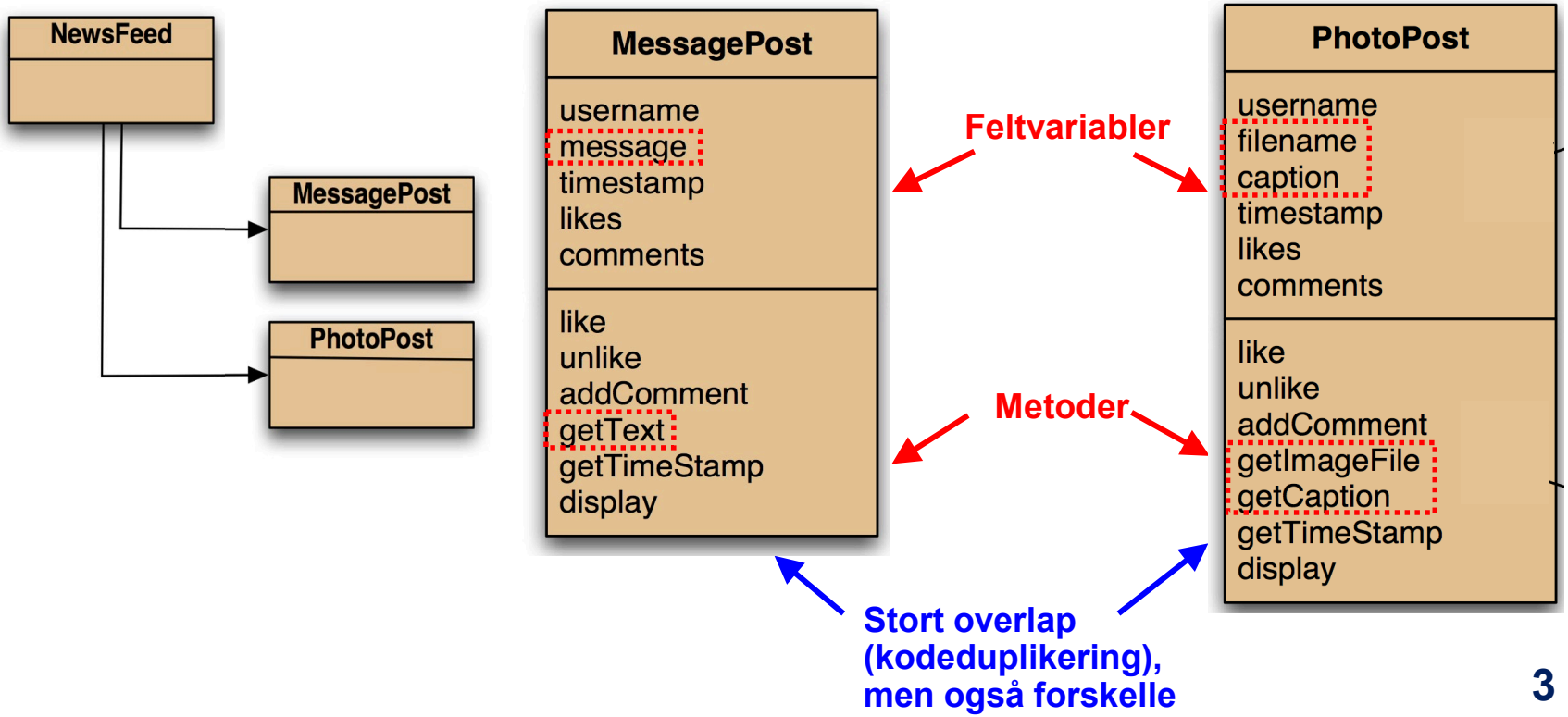


Træning i mundtlig præsentation

- **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
- **Evnen til at lave gode mundtlige præsentationer kan forbedres kraftigt ved intensiv træning**
  - Vi bruger derfor den anden af de to ugentlige øvelsesgange på dette
  - Det er **obligatorisk** at lave mindst 2 præsentationer – som skal **godkendes** af instruktoren
- **Tag træningen alvorligt**
  - Det er den eneste gang under jeres studier, hvor I får **omfattende konstruktiv feedback** på, hvordan I kan **forbedre** jeres mundtlige præsentationer – og dermed jeres karakterer ved mundtlig eksamen
  - Træning gør mester – de timer I bruger på det, er virkelig godt givet ud
  - Se videoerne om den "perfekte" eksamenspræstation og hør jeres medstuderendes præsentationer – det lærer I også meget af

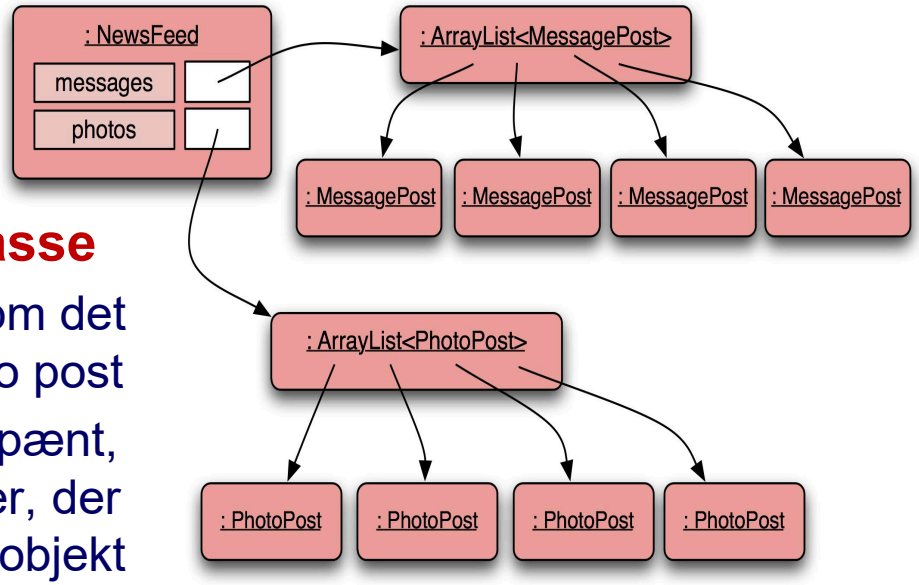
Nedarvning (subklasser)

- **Vi vil gerne modellere et simpelt nyhedssystem med to slags meddelelser**
  - Almindelige tekstmeddelelse (MessagePost)
  - Fotomeddelelse (PhotoPost)
- **Uden brug af nedarvning ser det sådan ud**



Dublering af kode

- **MessagePost og PhotoPost ligner hinanden**
  - Store dele af de to klassers Java kode er identiske
  - Det gør koden svær at overskue og vanskelig at vedligeholde
  - Der opstår let fejl og inkonsistens, f.eks. når man ændrer noget i den ene klasse, men glemmer at rette i den anden
- **Også dublering i NewsFeed klassen**
  - Den har to arraylister
  - Mange metoder gennemløber begge lister (dubleret kode)

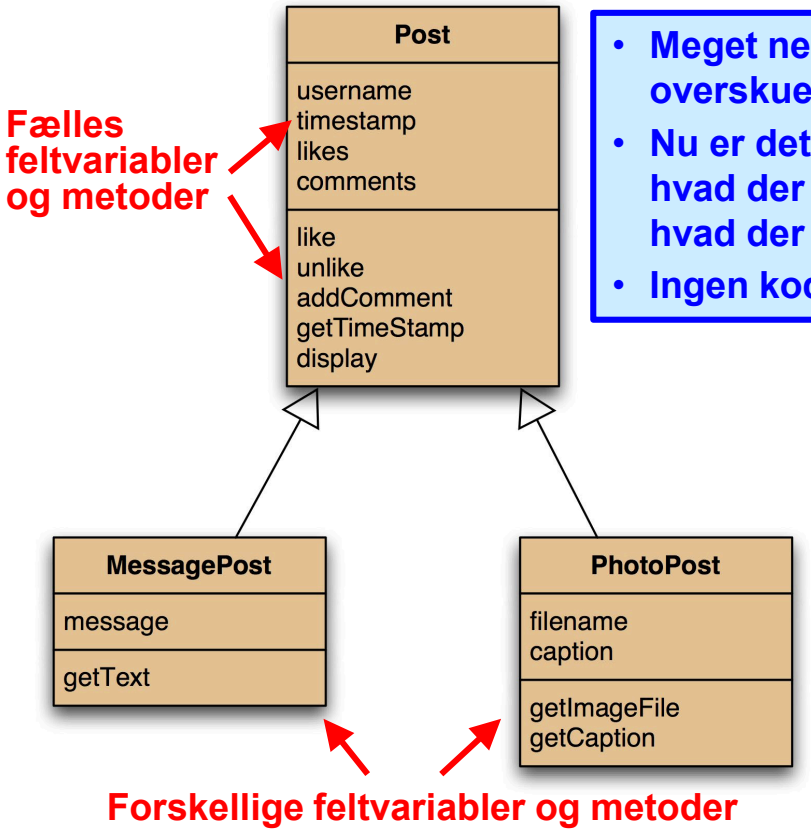


- **Vi kunne nøjes med en Post klasse**
  - Med en feltvariabel, der angiver om det er en message post eller en photo post
  - Men det bliver heller ikke særligt pænt, idet der så er feltvariabler/metoder, der ikke giver mening for det enkelte objekt

Det er selvfølgelig endnu værre, hvis vi i stedet for to slags postings har mange forskellige slags

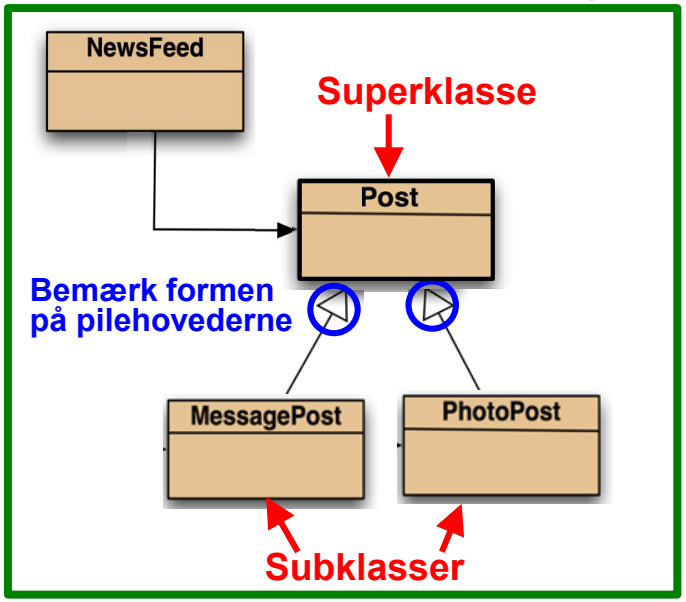
Brug af nedarvning (subklasser)

- **Vi kan i stedet lave en Post klasse, som har MessagePost og PhotoPost som subklasser**
  - En subklasse **arver** alle feltvariabler og metoder fra den oprindelig klasse, som kaldes en superklasse
  - Alt det der er fælles for subklasserne placeres i superklassen



- **Meget nemmere at overskue**
- **Nu er det helt klart, hvad der er fælles, og hvad der er forskelligt**
- **Ingen kodedublering**

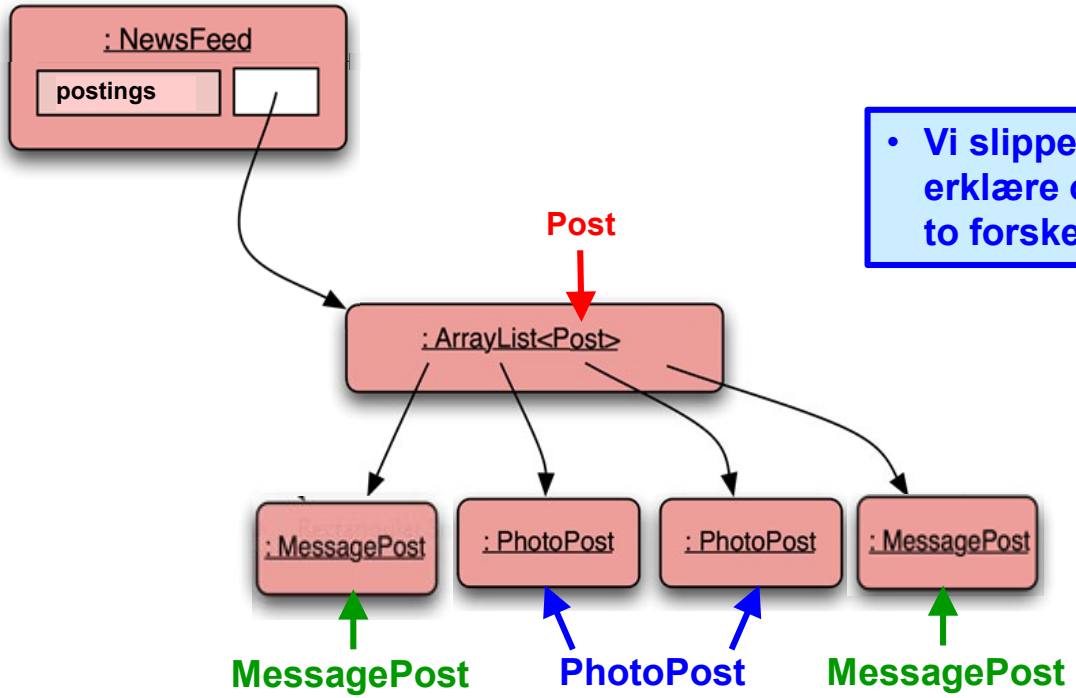
Klassediagram



- **Newsfeed klassen bruger Post klassen, men har ikke behov for at kende noget til underklasserne**

NewsFeed klassen

- **Vi undgår også kodedublering i NewsFeed klassen**
  - Nu har vi kun **én** arrayliste med elementtypen **Post**
  - Alle de steder, hvor man skal bruge et **Post** objekt, kan man i stedet bruge et objekt fra en **subklasse**
  - Dvs. at elementerne i arraylisten kan være af typen **Post**, **MessagePost** eller **PhotoPost**



- **Vi slipper for at skulle erklære og gennemløbe to forskellige arraylister**

Nedarvning i Java

Superklasse

```
public class Post {
  private String username;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;
  public Post(String author) {
     username = author;
     timestamp = System.currentTimeMillis();
     likes = 0;
     comments = new ArrayList<>();
  }
  // Methods omitted
}
```

4 feltvariabler

Konstruktør
- Initialiserer de 4 feltvariabler

Angiver at klassen er en subklasse af Post

Subklasse

```
public class MessagePost extends Post {
  private String message;
  public MessagePost(String author, String text) {
     super(author);
     message = text;
  }
  // Methods omitted
}
```

Feltvariabel
- I tilgift til de 4, der nedarves

Konstruktør
- Kalder superklassens konstruktør, så de fire nedarvede feltvariabler bliver initialiseret
- Initialiserer egen feltvariabel

Kald af superklassens konstruktør (med de rette parametre)

Access rettigheder

Superklasse

```
public class Post {
  private String username;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;
  public Post(String author) {
    username = author;
    timestamp = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<>();
  }
  // Methods omitted
}
```

Subklassen arver de fire feltvariabler i superklassen

- Men de er private, og derfor kan subklassen (som alle andre klasser) kun tilgå dem via public accessor og mutator metoder defineret i superklassen

public metoder fra superklassen kan kaldes i subklassen som om de var lokale

- Uden brug af dot notation, f.eks. getTimestamp()

Subklasse

```
public class MessagePost extends Post {
  private String message;
  public MessagePost(String author, String text) {
    super(author);
    message = text;
  }
  // Methods omitted
}
```

- Java kræver, at første sætning i en subklasses konstruktør er et kald til superklassens konstruktør
- Hvis dette ikke er tilfældet, indsætter compileren et sådant kald (uden parametre)
- Det virker kun, hvis der er en konstruktør uden parametre

Subtyper

- **Alle objekt klasser bestemmer en type**
  - Når en klasse B er en subklasse af klassen A, siger vi at typen B er en subtype af typen A (som er en supertype af typen B)
- **De steder, hvor der skal bruges et objekt af typen A, kan man i stedet bruge et udtryk b, der evaluerer til et objekt af en subtype B**
  - Assignments
  - Parameterværdier
  - Returværdier
  - Elementer i objektsamlinger

```
a = b;
```

```
public void pip(A param) {...};
```

```
pip(b);
```

```
public A pop() {
   ...
   return b;
   ...
}
```

```
ArrayList<A> list;
list.add(b);
```

- **Ovenstående er kendt som Liskovs substitutionsprincip**
  - Vi har tidligere sagt at typerne skal **matche** (for assignments, parametre, returværdier og objektsamlinger)
  - Mere præcist betyder det, at typen af udtrykket **b** skal være en subtype af den forventede type **A** (eller A selv)

NewsFeed klassen

```
public class NewsFeed {
    private ArrayList<Post> posts;
    public NewsFeed() {
        posts = new ArrayList<>();
    }
    public void addPost(Post post) {
        posts.add(post);
    }
    public void show() {
        for(Post post : posts) {
            post.display();
            System.out.println();
        }
    }
}
```

Feltvariabel
- Arrayliste med elementtypen Post
- Elementerne i listen vil være af typen MessagePost eller PhotoPost (eller Post)

Konstruktør
- Initialiserer feltvariablen

Metode til indsættelse af postings
- Bemærk at parameteren er af typen Post
- Argumenter af typen MessagePost og PhotoPost er også lovlige

Metode til udskrift af alle postings
- For-each løkken løber arraylisten igennem og udskriver de enkelte elementer ved hjælp af display metoden fra Post klassen

Statisk og dynamisk type

- **Lad os se lidt nærmere på show metoden fra foregående slide**

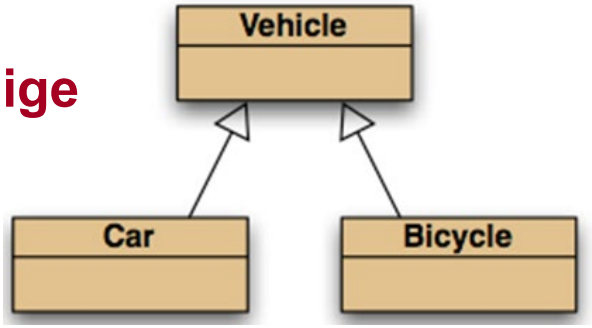
```
public void show() {
   for(Post post : posts) {
     post.display();
     System.out.println();
   }
}
```

- **Den lokale variabel post er erklæret til at have typen Post**
  - Vi siger derfor, at den **statiske type** for variablen post er Post
- **Javas variabler er polymorfe (≈ kan antage forskellige former)**
  - De kan pege på objekter af forskellig type
  - Når post peger på et MessagePost objekt er den **dynamiske type** MessagePost
  - Når post peger på et PhotoPost objekt er den **dynamiske type** PhotoPost
  - Når post peger på et Post objekt er den **dynamiske type** Post
- **Opsummering**
  - Den **statiske type** bestemmes af variablens erklæring
  - Den **dynamiske type** bestemmes af det objekt, som variablen pt peger på
  - Den dynamiske type er altid en **subtype** af den statiske type (eller identisk med denne)

Typecheck

- **Liskovs substitutions princip betyder at nedenstående tre assignments alle er lovlige**

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```



- **Oversætterens typecheck anvender de statiske typer**
  - De dynamiske typer kendes først under programudførelsen, og kan derfor ikke bruges af oversætteren (som ikke aner, hvad de er)
  - I nedenstående erklæringer, har v Vehicle som statisk type, mens c har Car som statisk type – begge har Car som dynamisk type

```
Vehicle v = new Car();
Car c = new Car();
```

  - Ifølge Liskovs substitutionsprincip er det **lovligt** at assigne c til v

```
v = c;
```

← • c's statiske type, Car, er en subtype af v's statiske type, Vehicle

  - Det modsatte er **ulovligt** – og giver en **oversætterfejl**

```
c = v;
```

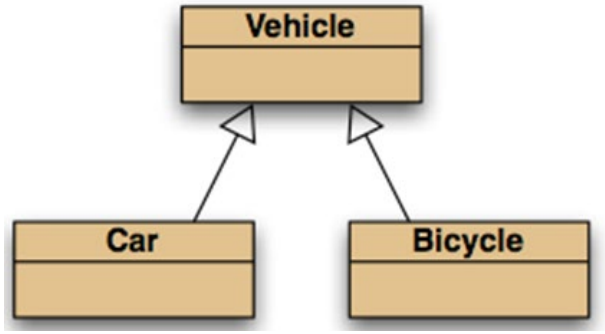
← • v's statiske type, Vehicle, er ikke en subtype af c's statiske type, Car
• Det er lige meget at v's dynamiske type er Car

Typeskift (type cast)

Samme erklæringer som før

```
Vehicle v = new Car();
Car c = new Car();
```

```
c = v;
```
← Ulovligt



- **Ved at bruge et type cast kan vi ændre den statiske type af et udtryk**

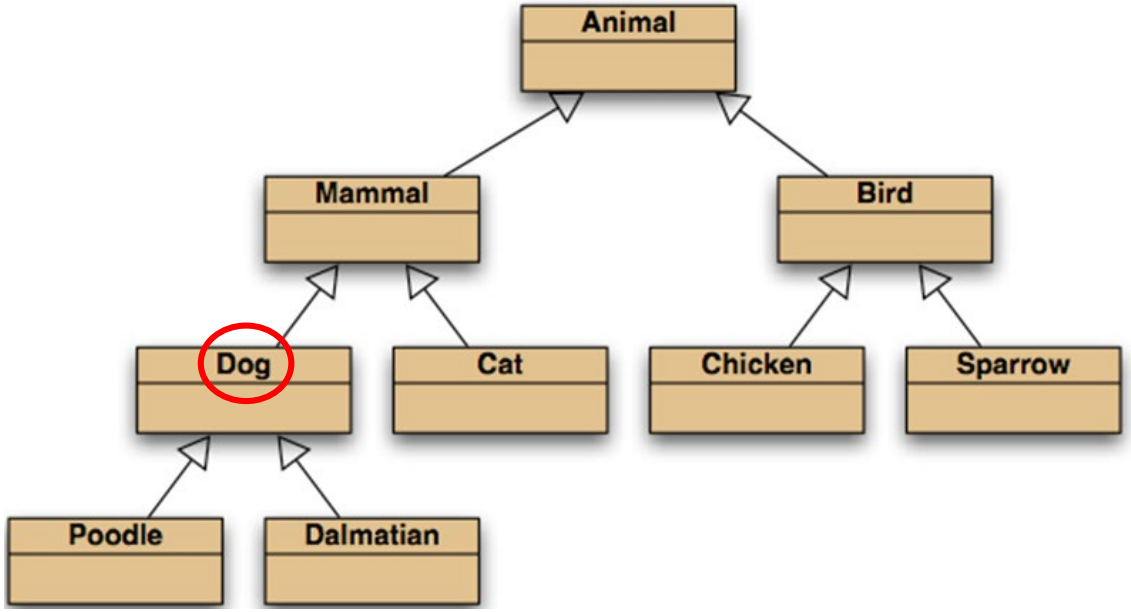
```
c = (Car) v;
```

  - Højresiden af assignment har nu den statiske type Car, og oversætteren godkender derfor assignmentet
  - Under **udførslen** af programmet tjekkes det, at den **dynamiske type** af v er Car (eller en subtype af Car)
  - Ellers får man en run-time fejl (ClassCastException)

- **Objektet, som v peger på, ændres ikke ved et type cast**
  - Det eneste, der ændres, er **oversætterens opfattelse** af udtrykkets type

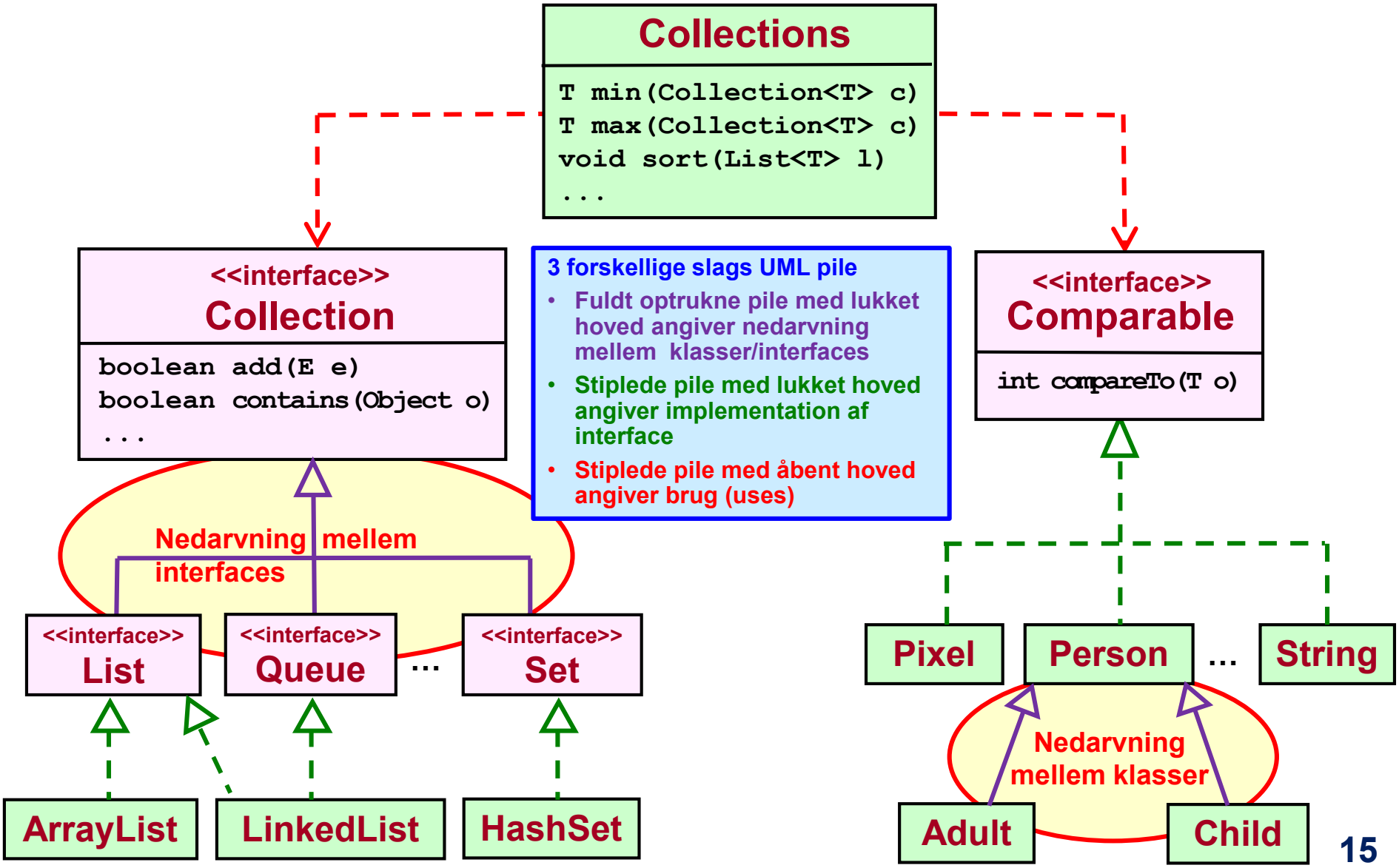
Nedarvning i flere niveauer

- **En subklasse kan igen have subklasser, osv.**



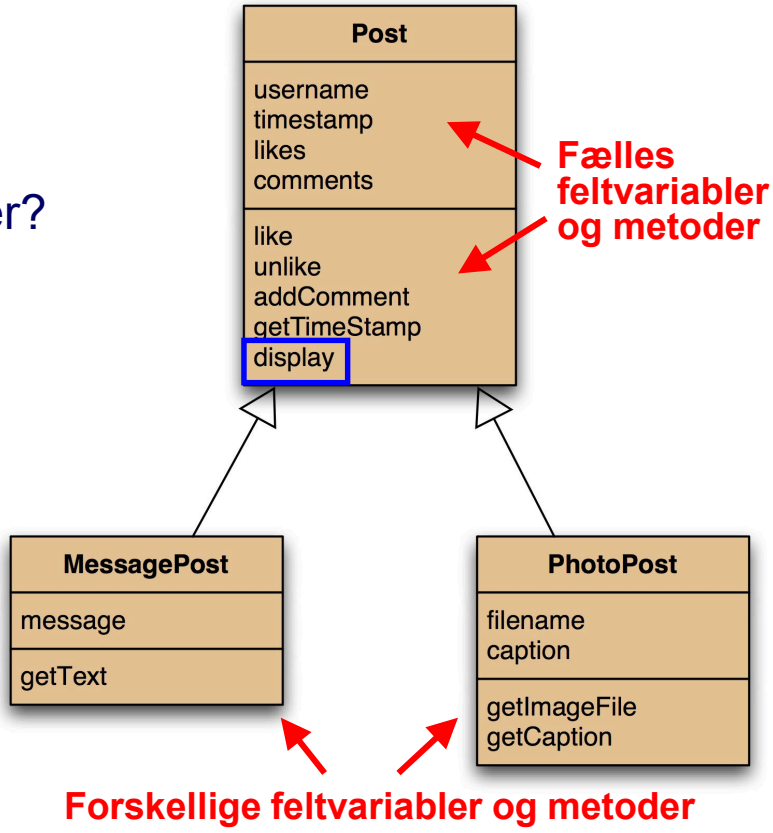
- **Dog klassen er en**
  - direkte subklasse af Mammal klassen
  - subklasse af Animal klassen (subklasse relationen er transitiv)
  - subklasse af sig selv (subklasse relationen er refleksiv)
  - direkte superklasse for Poodle og Dalmatian klasserne
  - superklasse af sig selv (superklasse relationen er refleksiv og transitiv)

Brug af Collection og Comparable



Metoder i subklasser

- **Lad os nu kigge lidt nærmere på metoder i super- og subklasser**
  - Post klassen har en display metode, der udskriver information om klassens tilstand
  - display metoden har kun adgang til feltvariabler i Post klassen
  - Hvad gør vi, hvis vi gerne vil udskrive information om subklassernes feltvariabler?



display metoden udskriver klassens tilstand på terminalen

- **Det ville give bedre cohesion at have en toString metode, der returnerer en tekststreng**
- **Så kan man på kaldsstedet selv bestemme, hvad man vil gøre ved tekststrengen**

Situationen er som vist nedenfor

Post klassen

```
public void display() {
  System.out.println(username);
  System.out.print(timeString(timestamp));
  if(likes > 0) {
    System.out.println(likes + " people like this.");
  }
  else {
    System.out.println();
  }
  if(comments.isEmpty()) {
    System.out.println("No comments.");
  }
  else {
    System.out.println(comments.size() + " comment(s).");
  }
}
```

- Post klassens display metode udskriver information om de fire feltvariabler, der ligger i Post klassen
- Men den kender ikke de feltvariabler, der ligger i subklasserne, og kan derfor ikke udskrive information om disse

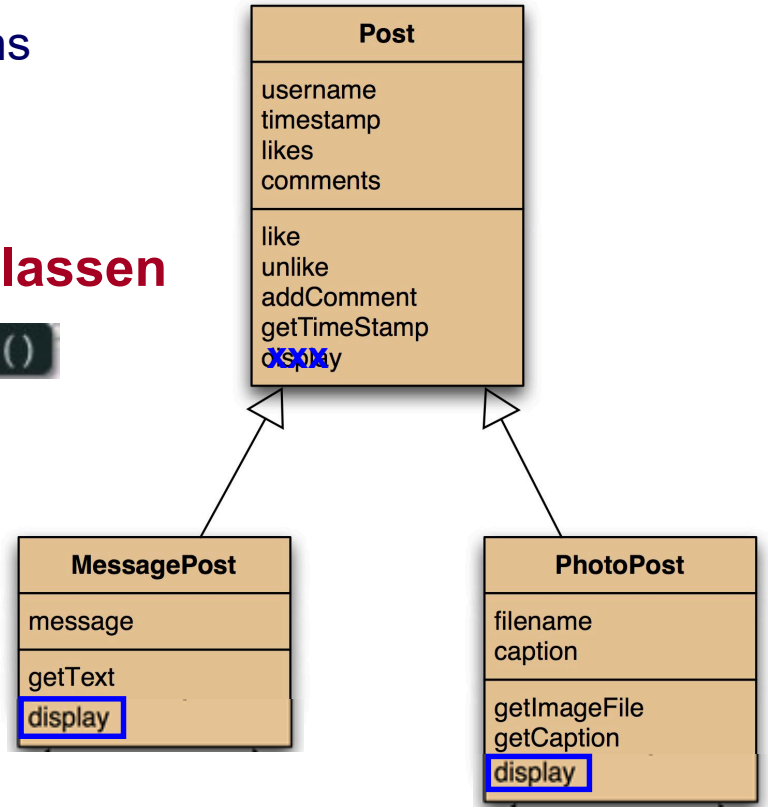
```
public void show() {
  for(Post post : posts) {
    post.display();
    System.out.println();
  }
}
```

Newsfeed

Første løsningsforslag (virker ikke)

- **Vi kan flytte display metoden til de to subklasser**
  - Metoderne har kun adgang til superklassens feltvariabler via acces metoder
  - Det giver massiv kodedublering
- **Vi får en compile-time fejl i NewsFeed klassen**

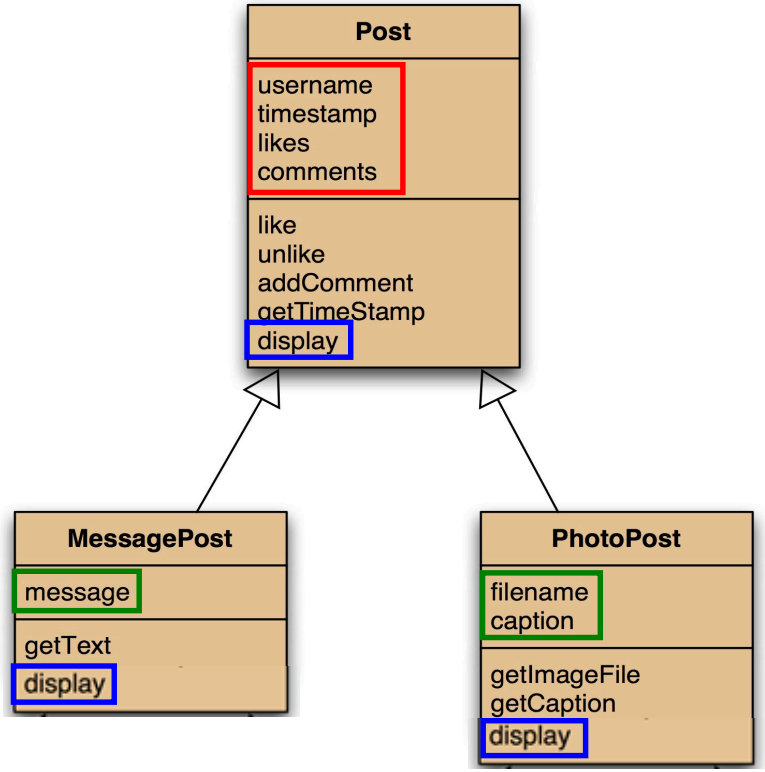
```
cannot find symbol -   method display()
public void show() {
   for( Post post : posts) {
     post.display();
     System.out.println();
   }
}
                                  Newsfeed
```



- **Oversætteren bruger de statiske typer**
  - Den statiske type af post er Post og oversætteren kigger derfor (uden held) i Post klassen (og dens superklasser) for at finde display metoden
  - Det hjælper **ikke**, at alle subklasserne har en display metode

Korrekt løsning

- **Vi har både en display metode i superklassen og i de to subklasser**
  - display metoderne i subklasserne kalder display metoden i superklassen
  - Via den er der adgang til superklassens feltvariabler uden brug af acces metoder (og uden kodedublikering)




```
public void display() {
   super.display();
   System.out.println(message);
}
```
MessagePost

```
public void display() {
   super.display();
   System.out.println(filename);
   System.out.println(caption);
}
```
PhotoPost

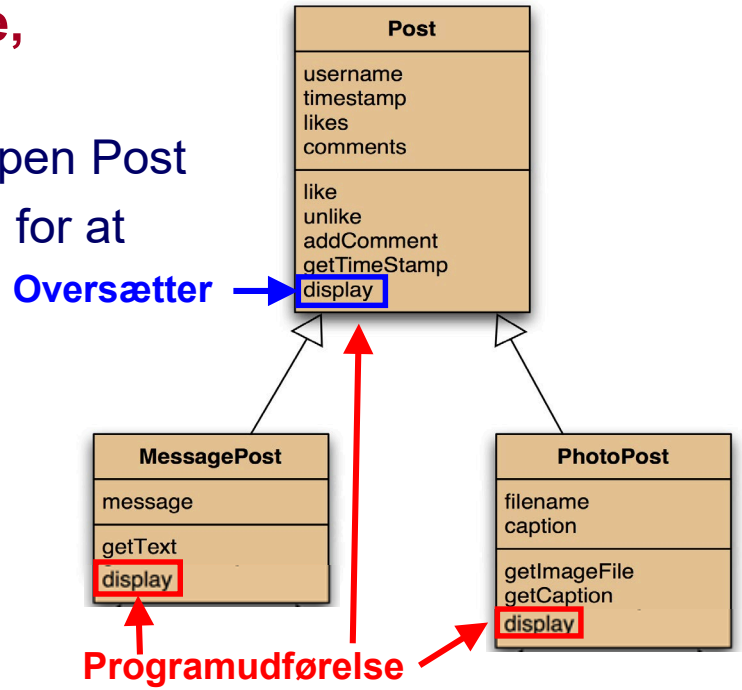
Kald af superklassens display metode

Kaldet af superklassens display metode behøver ikke at være i første linje, og det kan helt mangle (eller der kan være flere kald)

Statiske og dynamiske typer

- **Oversætteren bruger den statiske type, dvs. typen fra variablens erklæring**
  - Variablen post er erklæret til at være af typen Post
  - Oversætteren kigger derfor i Post klassen for at finde display metoden (og finder den)

```
public void show() {
   for(Post post : posts) {
     post.display();
     System.out.println();
   }
}
```
Newsfeed

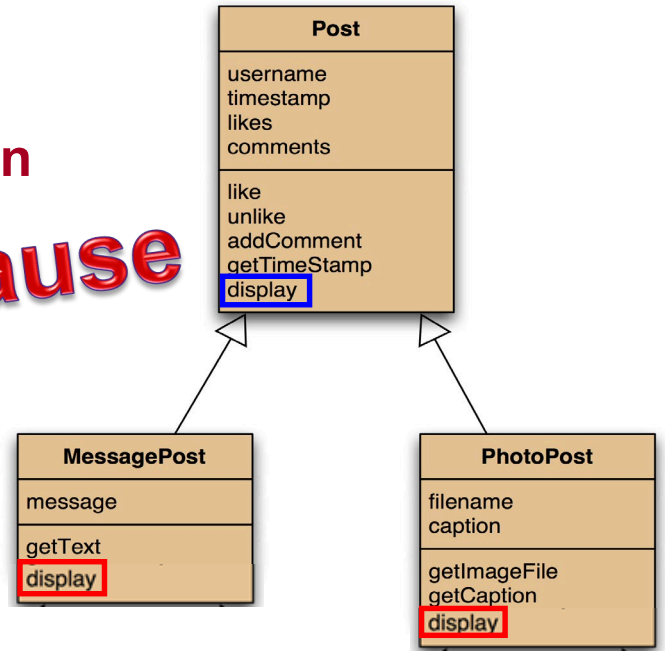


- **Under programudførelsen bruges den dynamiske type, dvs. typen af variablens aktuelle værdi**
  - Når den dynamiske type er en MessagePost, kaldes display metoden fra MessagePost
  - Når den dynamiske type er en PhotoPost, kaldes display metoden fra PhotoPost
  - Når den dynamiske type er en Post, kaldes display metoden fra Post
  - Dette princip kaldes **dynamisk method lookup**

Overskrivning

- **De to subklasser overskriver superklassens display metode med deres egne udgaver af metoden**
  - På engelsk: override ≈ tilsidesætte / underkende
  - De nye metoder vil ofte have **samme** signatur, returtype og access modifier, som den de overskriver
  - Men det er nok, at de **matcher** (se Link)
- **Når man overskriver en metode, bør man angive dette via et @Override tag**
  - Gør det lettere at forstå koden
  - Oversætteren protesterer, hvis signatur, returtype eller access modifier ikke matcher

Pause



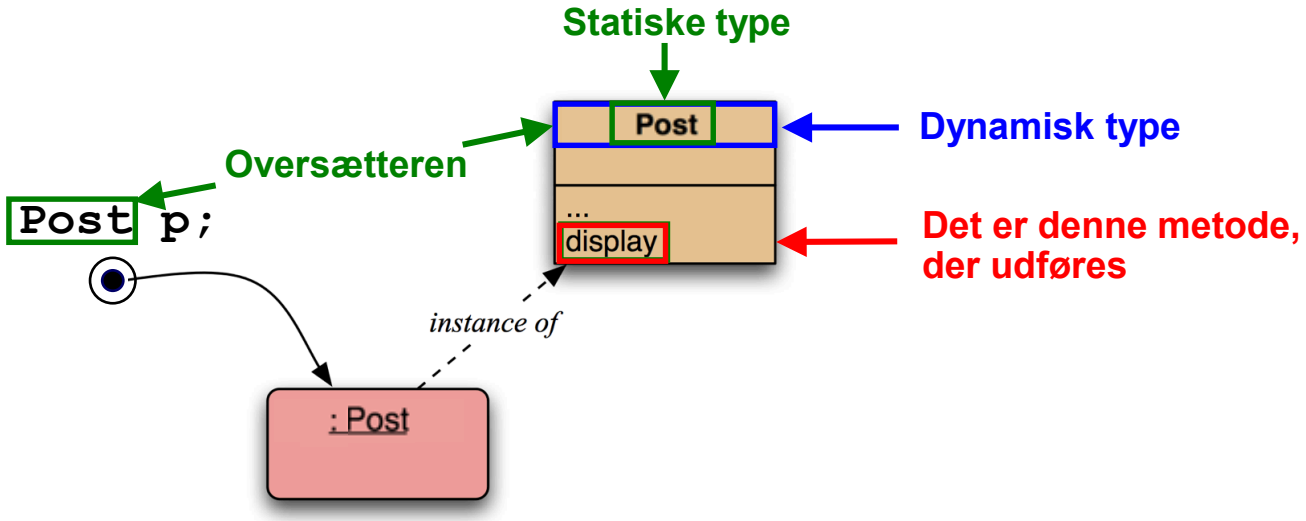
De to røde display metoder overskriver den blå

```
@Override
public void display() {
   super.display();
   System.out.println(message);
}
```

MessagePost

Metodekald (simpelt eksempel, én klasse)

- **Hvilken metode udføres ved metodekaldet p.display()?**



Oversætteren

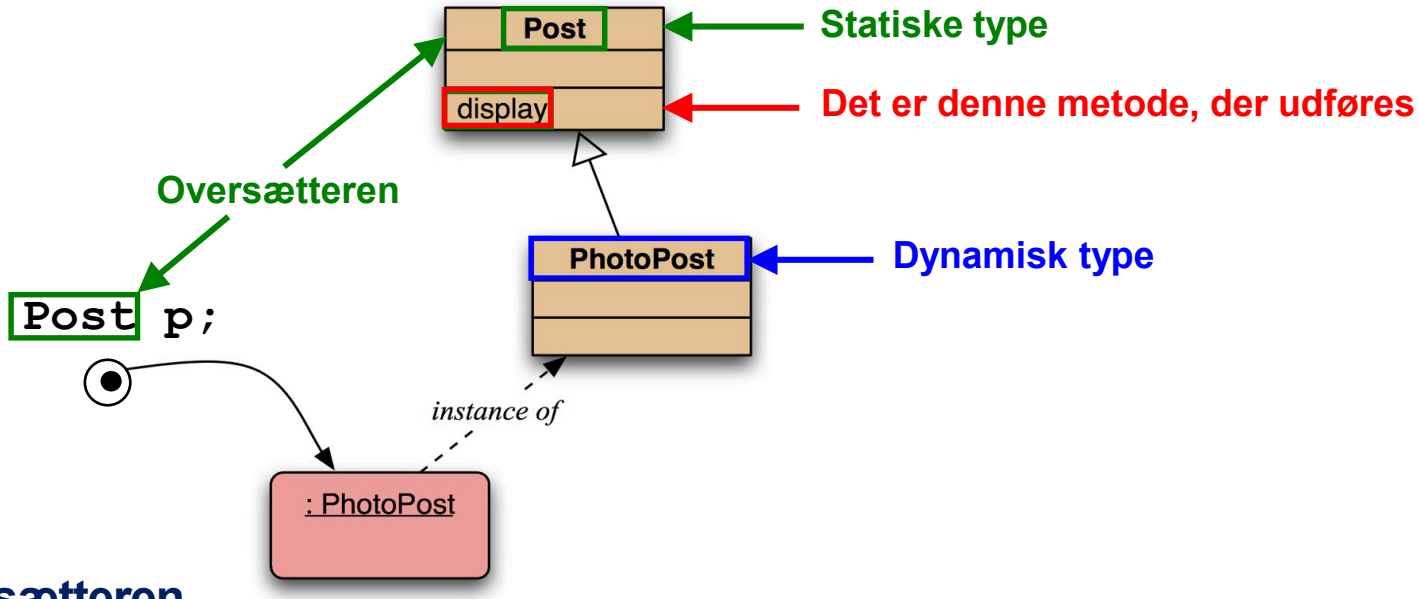
- Variablen p har den statiske type Post ,
  og her har oversætteren fundet en display metode

Programudførelsen

- Variablen p har den dynamiske type Post
- display metoden søges i Post og findes der

Metodekald (nedarvning)

- **Hvilken metode udføres ved metodekaldet p.display()?**



Oversætteren

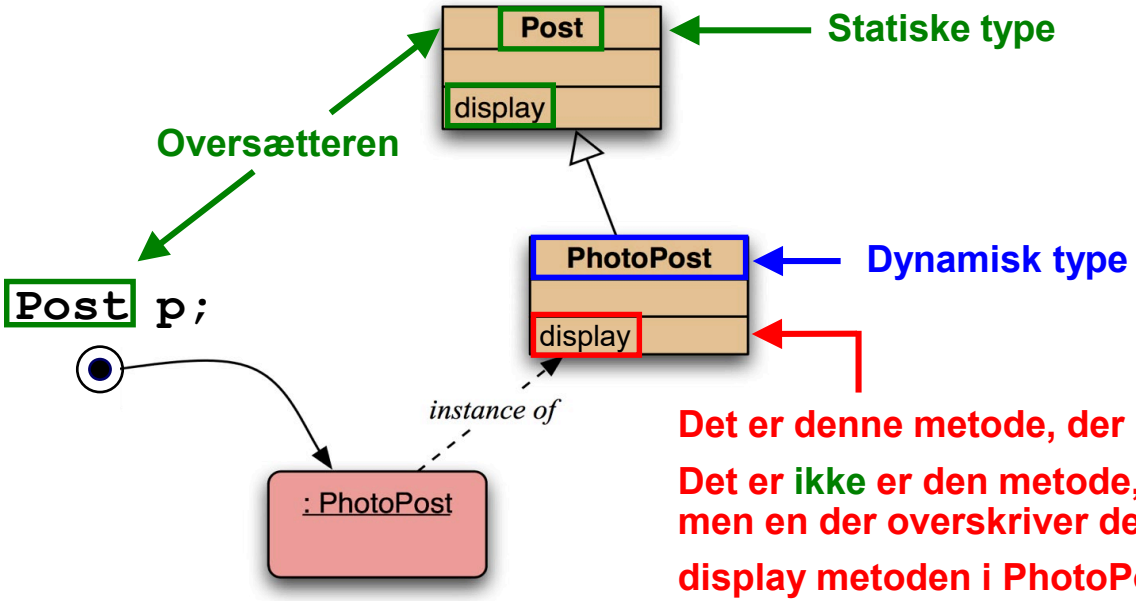
- Variablen p har den statiske type Post,
  og her har oversætteren fundet en display metode

Programudførelsen

- Variablen p har den dynamiske type PhotoPost
- display metoden søges i PhotoPost (uden held)
- display metoden søges i superklasserne (efter tur) og findes i Post

Metodekald (nedarvning og overskrivning)

- **Hvilken metode udføres ved metodekaldet p.display()?**



Oversætteren

- Variablen p har den statiske type Post, og her har oversætteren fundet en display metode

Programudførelsen

- Variablen p har den dynamiske type PhotoPost
- display metoden søges i PhotoPost og findes der

Dynamic method lookup

- **Under udførslen findes den rigtige metode ved at**
  - bestemme variablens dynamiske type
  - søge metoden i denne klasse eller i en superklasse af den
- **Oversætteren har tjekket, at metoden findes i**
  - variablens statiske type (eller en af dennes superklasser)
- **Den dynamiske type er en subtype af den statiske type**
  - Vi er derfor sikre på at finde metoden (under udførelsen)
  - Den er i den statiske type eller en af dennes superklasser
  - Kan være overskrevet i en subklasse heraf
- **Ovenstående er kendt som dynamic method lookup**
  - Spiller en essentiel rolle i anvendelsen af subklasser
  - Tillader at de enkelte subklasser kan lave deres egne specialtilpassede versioner af en metode
  - Eksempel på responsibility-driven design

Polymorfi

- **Betyder at noget kan antage forskellige former**
- **Vi har tidligere set, at Javas variabler er polymorfe – dvs. kan pege på objekter af forskellig type**
  - Nedenstående variabel kan pege på objekter af typen MessagePost og PhotoPost (samt Post)

```
public void show() {
   for(Post post : posts) {
     post.display();
     System.out.println();
   }
}
```

Newsfeed

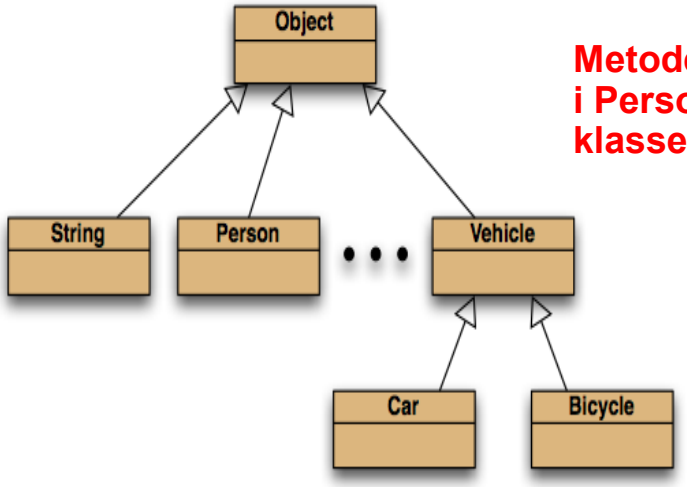
Polymorf variabel

Polymorft metode kald

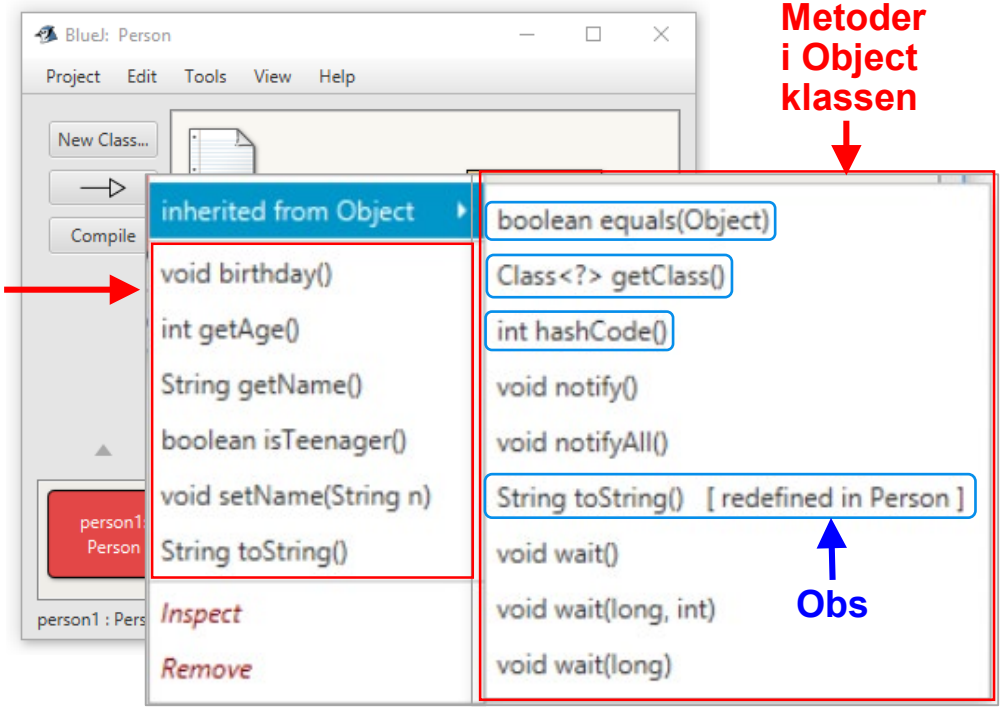
- **Vi har nu set, at også Javas metodekald er polymorfe – dvs. kan aktivere metoder i forskellige typer (klasser)**
  - Ovenstående metodekald kan aktivere display metoden i MessagePost og display metoden i PhotoPost (samt display metoden i Post)

Object klassen

- **Alle klasser i Java er subklasser af Object klassen**



Metoder i Person klassen



Metoder i Object klassen

- **Object klassen indeholder en række nyttige metoder, som de øvrige klasser nedarver (og eventuelt overskriver)**
  - **toString** metoden returnerer en tekstrepræsentation af objektet
  - **equals** metoden tjekker om to objekter "ligner hinanden"
  - **hashCode** metoden beregner en hashkode
  - **getClass** metoden returnerer objektets dynamiske type

hashCode metoden

- **Metoden returnerer en hashkode for objektet**
  - Hashkoder bruges som nøgler i såkaldte hashtabeller, f.eks. i klasserne HashSet og HashMap
  - Det kræves, at hashcode metoden er **konsistent** med equals metoden
    - **x**.equals(**y**) $\Rightarrow$ hashcode(**x**) == hashCode(**y**)
  - hashCode metoden bør konstrueres således, at der for to umage objekter er **lille sandsynlighed** for, at deres hashkoder er identiske
- **I Object klassen returnerer hashCode metoden objektets memory adresse (det sted, hvor objektet ligger i computerens lager)**
  - Object klassens equals metoden er defineret ved hjælp af == operatoren (hvilket betyder, at equals kun evaluerer til sand, hvis objekterne er identiske)
  - Dermed er det nemt at se, at konsistenskravet er opfyldt
- **Hvis jeres egne klasser overskriver equals metoden, vil det normalt også være nødvendigt at overskrive hashCode metoden**
  - Hvis I tillader to forskellige objekter at være mage til hinanden, skal de ifølge konsistenskravet også have samme hashkode

Hashkoder for tekststrenge

- **Det er en hel videnskab at definere hashkoder, hvor der er så få kollisioner som muligt**
  - Lad os starte med at se, hvordan hashCode kan defineres i String klassen
  - Et første forsøg kunne være nedenstående, hvor s er en feltvariabel, der indeholder værdien af tekststrengen

```
public int hashCode() {
   int hc = 0;
   for(int i=0; i<s.length(); i++) {
     hc = hc + s.charAt(i);
   }
   return hc;
}
```

```
hashCode("eat") = 'e' + 'a' + 't' = 101 + 97 + 116 = 314
```

- **Det fungerer ikke ret godt**
  - Hvis vi ombytter tegnene i tekststrengen får vi samme hashkode
  - Det giver rigtigt mange kollisioner (tekststrenge med samme hashkode)

```
hashCode("tea") = 't' + 'e' + 'a' = 116 + 101 + 97 = 314
```

Hashkoder for tekststrenge version 2

- **Ved at gange et primtal på før summeringen, får vi langt færre kollisioner (tekststrenge med samme hashkode)**

```
public int hashCode() {
   int hc = 0;
   for( int i = 0; i < s.length(); i++) {
     hc = 31 * hc + s.charAt(i);
   }
   return hc;
}
```

```
hashCode("eat") = 31² * 'e' + 31 * 'a' + 't' = 100184
```

```
hashCode("tea") = 31² * 't' + 31 * 'e' + 'a' = 114704
```

Overskrivning af hashCode metoden

- **For Person klassen kan man overskrive hashCode som følger**

```
public int hashCode() {
   return 11 * name.hashCode() +
          13 * new Integer(age).hashCode();
   }
}
```

Kald af hashCode metoden i String klassen

Kald af hashCode metoden i Integer klassen

  - Hashkoderne for feltvariablerne multipliceres med **forskellige** primtal
  - Primitive værdier konverteres først til deres wrapper type

- **Vi bruger kun feltvariabler, som sammenlignes i equals metoden**
  - På den måde vil konsistenskravet automatisk være opfyldt

- **I subklassen Child kan man definere hashCode som følger**

```
public int hashCode() {
   return super.hashCode() +
          17 * new Integer(noOfToys).hashCode();
}
```

- Kald af hashCode metoden i superklassen
- Hashkode baseret på superklassens feltvariabler

Hashkode baseret på feltvariabler i subklassen

toString metoden

- **Metoden bruges bl.a. i print og prinln metoderne**
  - Hvis argumentet til metoderne ikke allerede er en tekststreng, bruges toString til at konvertere argumentet til en tekststreng
  - Det er også toString metoden, der bruges, når den ene side af en + operation ved konkatenering skal konverteres til en tekststreng
- **I Object klassen er toString metoden defineret til at returnere**
  - **Person@19bb25a**, hvor Person er klassens navn og 19bb25a er objektets hashkode i det hexadecimale system (16-tals systemet)
  - Ovenstående giver ikke ret meget interessant information, og mange klasser overskriver derfor toString metoden
  - Ex: I String klassen returnerer toString metoden værdien af den tekststreng som String objektet repræsenterer
- **Jeres egne klasser kan også overskrive toString metoden**
  - For Person klassen kan man f.eks. returnere tekststrengen **"Cecilie:18"**, hvor Cecilie er personens navn og 18 er personens alder
  - I køreprøvesættene overskrev i toString metoden (for den simple af klasserne)

getClass metoden

- **I tilgift til Object klassen er der også en klasse, der hedder Class**
  - I et kørende Java program har hver type (også de primitive) ét (og kun et) tilhørende objekt fra Class klassen
  - Class er en parametriseret type og objektet for klassen A tilhører Class<A>
- **getClass metoden returnerer objektets dynamiske type**
  - Mere præcist det Class object, som svarer til den dynamiske type

```
v1.getClass() == v2.getClass()
```

Tjekker om de dynamiske typer for variablerne v1 og v2 er er identiske

```
v.getClass()== Person.class
```

Tjekker om den dynamiske type for variablen v er Person

Giver os det Class objekt, der svarer til Person klassen

- **I begge tilfælde skal de dynamiske typer være identiske**
  - Det er ikke nok, at den ene er en subklasse af den anden

Det reserverede ord instanceof

- Man kan også undersøge en variabels dynamiske type ved hjælp af det reserverede ord instanceof

```
v instanceof Car
```

Tjekker om den dynamiske type for variablen v er Car eller en subklasse af Car (her kræves ikke at typerne er identiske)

- Bruges ofte i forbindelse med type cast

```
if( v instanceof Car ) {
  c = (Car) v;
}
```

Når vi har tjekket, at variablens dynamiske type er Car (eller en subklasse af Car), kan vi uden fare for run-time exceptions lave et type cast til Car

equals metoden

- **Metoden bruges bl.a. i forbindelse med mængder (Set)**
  - For en mængde vil et kald **add(elem)** kun tilføje elem, hvis der ikke er et element e i mængden, der ligner (elem.equals(e))
- **I Object klassen er equals metoden defineret ved hjælp af ==**
  - Det betyder at e1.equals(e2) kun evaluerer til sand, hvis de to objekter er identiske (e1 == e2)
  - Dette er ofte for restriktivt, og mange klasser overskriver derfor equals metoden
  - Ex: I String klassen tjekker equals metoden om de to tekststrenge er mage til hinanden, dvs. indeholder de samme tegn (i samme rækkefølge)
- **Det kræves at equals metoden opfylder to betingelser**
  - Den er en ækvivalensrelation
    - **Refleksiv**: **x**.equals(**x**) for alle x
    - **Symmetrisk**: **x**.equals(**y**) $\Leftrightarrow$ **y**.equals(**x**) for alle x,y
    - **Transitiv**: **x**.equals(**y**) $\wedge$ **y**.equals(**z**) $\Rightarrow$ **x**.equals(**z**) for alle x,y,z
  - Ingen objekter er mage til null: x != null $\Rightarrow$! x.equals(null) for alle x
  - Hvis I **overskriver** en equals metode, **skal** ovenstående være opfyldt

Overskrivning af equals metoden

- For Person klassen kan man overskrive equals som følger

```
public boolean equals(Object otherObject) {
  if( this == otherObject ) {return true;}
  if( otherObject == null ) {return false;}
  if( getClass() != otherObject.getClass() ) {return false;}
  Person other = (Person) otherObject;
  return name.equals(other.name) && age == other.age;
}
```

Optimalisering

null?

Dynamiske type

Type cast

Sammenlign (udvalgte) feltvariabler

Bemærk at parameteren har typen Object

equals metoden i String

For primitive typer er det ok at bruge ==

- I subklassen Child kan man overskrive equals som følger

- Kald af equals metoden i superklassen
- Tjekker den dynamiske type og superklassens feltvariabler

```
public boolean equals(Object otherObject) {
  if( ! super.equals(otherObject) ) {return false;}
  return noOfToys == otherObject.noOfToys;
}
```

Tjek af subklassens egne feltvariabler (eller nogle af dem)

BlueJ bogen overskriver equals på en lidt anden måde og kommer ikke ind på overskrivning i subklasser

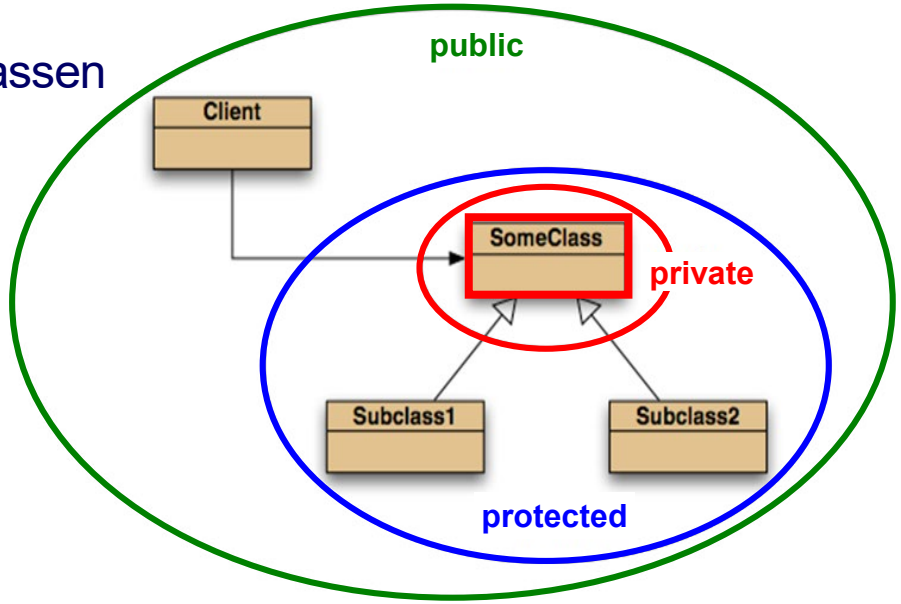
Hvis I overskriver equals, skal I også overskrive hashCode

I behøver ikke at huske detaljerne i overskrivning af equals og hashCode metoderne udenad

I kan slå op i mine slides, når I får bruge for det, f.eks. i Computerspil 1

Protected access

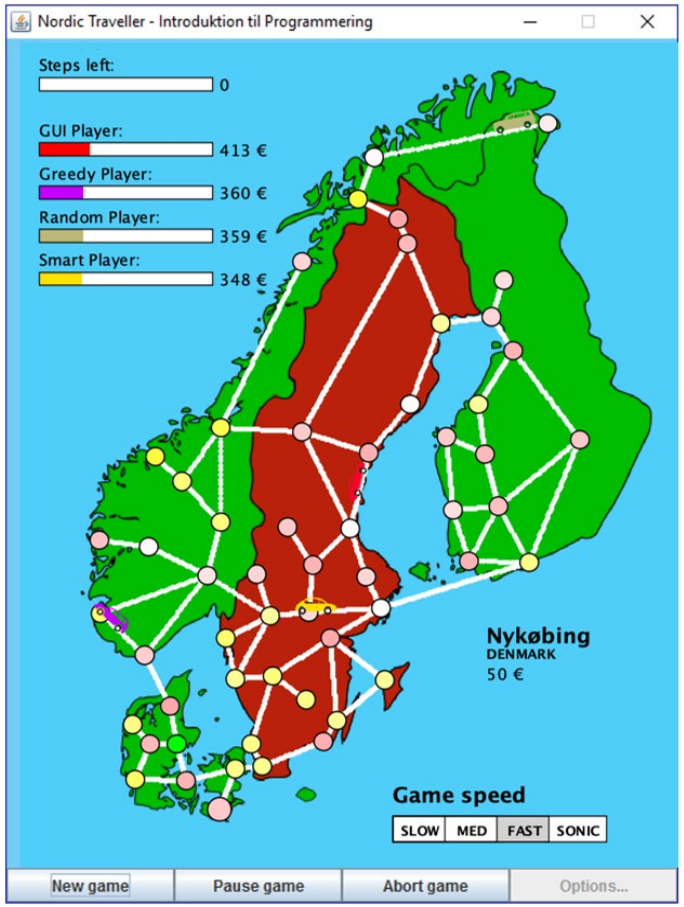
- **Vi har tidligere set, at feltvariabler og metoder kan være private eller public**
  - De kan også være **protected**
  - Det betyder, at de kan tilgås fra klassen selv og alle dens subklasser
- **protected feltvariabler**
  - Samme ulemper som public
  - Bør **aldrig** bruges
- **protected metoder**
  - Kan i nogle situationer give god mening



- **I Java kan feltvariabler og metoder, der er protected, også tilgås fra klasser i samme programpakke (package)**
  - Det gør det mere tvivlsomt at benytte protected
  - Man kan også helt **udelade** access modifier'en, hvilket betyder, at feltvariabler/metoder kan tilgås fra klasser i samme programpakke

Projektopgave om computerspil

- **I kursets sidste fem uger skal I, sammen med jeres makker, programmere et computerspil**
  - Den første af ugens øvelsesgange bruges på projektopgaven
  - Hver uge tilføjer I ting, som I har lært om i de nærmest foregående uger
- **Et antal spillere rejser rundt mellem byer i forskellige lande og indsamler point**
  - En af spillerne (GUI player) styres af brugeren ved hjælp af musen
  - De øvrige tre spillere styres af spillet (dvs. computeren)



Pointgivning

- **Hver delaflevering giver op til 3 point, som (sammen med pointene fra køreprøven) indgår ved fastlæggelsen af den endelige karakter for kurset**
  - 3 = fremragende (få og uvæsentlige mangler)
  - 2 = fortrinlig (nogle mindre mangler)
  - 1 = acceptabel (adskillige mangler)
  - 0 = genaflevering
  - Der kan gives halve point

- **≥ 1 er godkendt**
- **De påpegede fejl rettes i næste delaflevering**

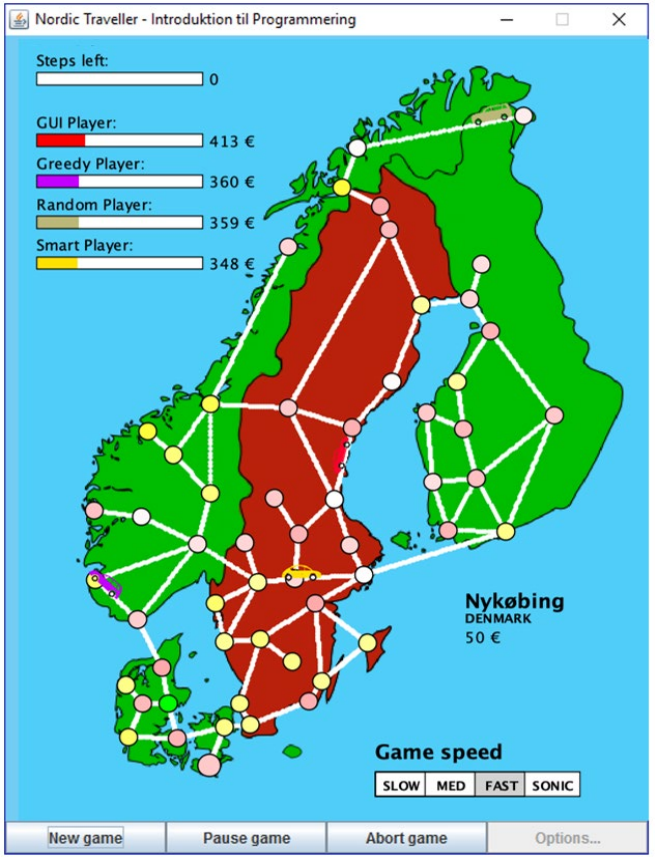
- **0 betyder genaflevering, hvor man højst kan få 1**

- **Fradrag**
  - Hvis man afleverer for sent (uden gyldig grund) trækkes 1 point pr påbegyndt døgn
  - Hvis man ikke retter fejl fra de foregående delafleveringer, trækker de ned en gang til
  - Genaflevering skal ske senest 1 uge efter den oprindelige afleveringsfrist
  - En aflevering tæller først som værende afleveret, når eventuelle genafleveringer af alle foregående opgaver (inklusiv Dronninger og Raflebæger 4) også er afleveret

- **Opgaverne rettes først, når afleveringsfristen er udløbet**
  - Hvis I (inden afleveringsfristens udløb) afleverer en forbedret udgave, er det den, der rettes
  - Hvis man har afleveret (en eller flere gange) inden fristen, må man **ikke** aflevere efter fristen (det betragtes som forsøg på snyd og kan give fradrag)

Computerspil 1 (første delaflevering)

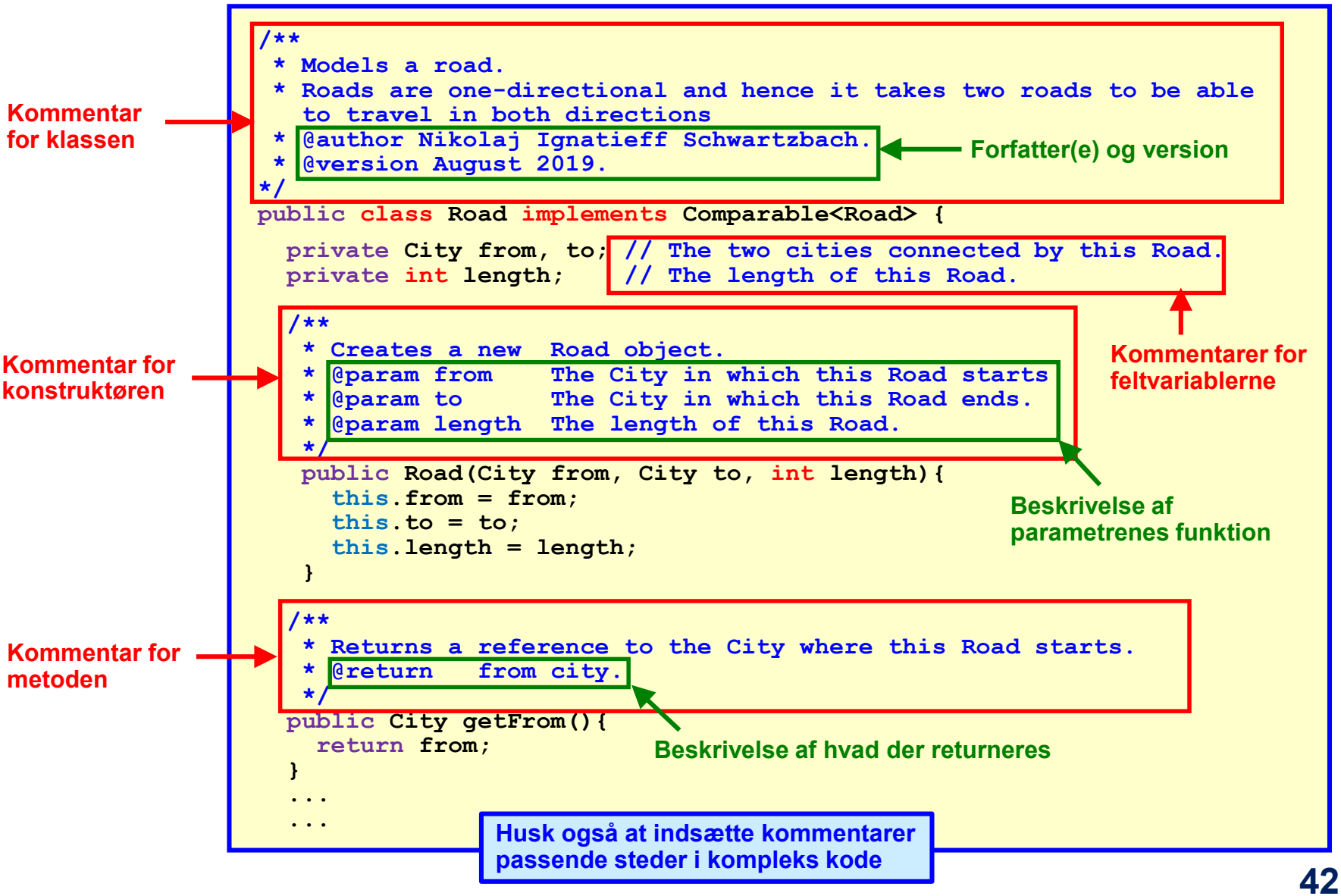
- **I første delaflevering skal I bl.a. modellere byerne og vejene imellem dem, samt landene hvori byerne ligger i**
  - Når I er færdige, vil I kunne spille
  - Der er forholdsvis meget at lave, men langt de fleste af tingene er lette at implementere
  - En af instruktorerne lavede en løsning på under en time



Krav til jeres dokumentation

- **Dokumentation skal følge de retningslinjer, der gives i BlueJ bogen**
  - Herunder specielt afsnit 6.11, afsnit 9.7 og Appendix I
- **Alle klasser, konstruktører og metoder skal have en passende kommentar (der begynder med /** og slutter med */)**
  - Første sætning kopieres automatisk til **summary-delen** i Documentation View
  - Hele kommentaren kopieres automatisk til **detail-delen** i Documentation View
  - Der skal indsættes @author og @version tags i alle klasser samt @param og @return tags i alle metoder/konstruktører
  - Indsæt også forklarende kommentarer i komplekse kodestumper
  - For at undgå for mange sprogskift, anbefales det, at alle kommentarer skrives på Engelsk/Amerikansk
- **Brug Documentation view til at kontrollere, at resultatet er fornuftigt og giver relevant og letlæselig information til brugere, der ikke kender implementationen af klasserne**

Dokumentation af Road klassen



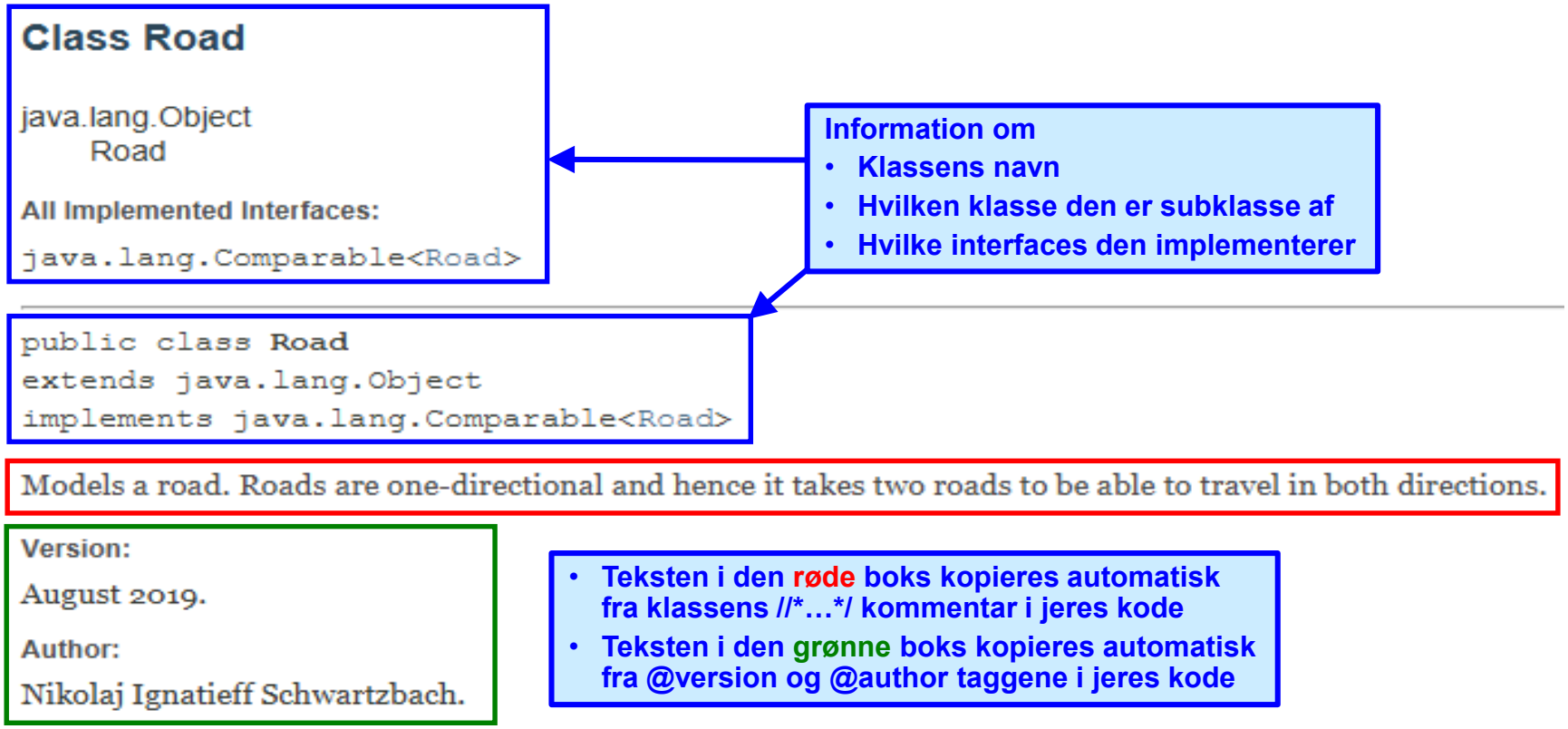

```
/**
 * Models a road.
 * Roads are one-directional and hence it takes two roads to be able
   to travel in both directions
 * @author Nikolaj Ignatieff Schwartzbach.
 * @version August 2019.
*/
public class Road implements Comparable<Road> {
  private City from, to; // The two cities connected by this Road.
  private int length;    // The length of this Road.

  /**
   * Creates a new  Road object.
   * @param from    The City in which this Road starts
   * @param to      The City in which this Road ends.
   * @param length  The length of this Road.
   */
   public Road(City from, City to, int length){
     this.from = from;
     this.to = to;
     this.length = length;
   }

  /**
   * Returns a reference to the City where this Road starts.
   * @return   from city.
   */
  public City getFrom(){
    return from;
  }
  ...
  ...
```

Documentation view

- **Husk at kontrollere, at resultatet er fornuftigt**
  - Skal give relevant og letlæselig information – også for brugere, der ikke kender implementationen af jeres klasser



Documentation view (summaries)

Constructor Summary

Constructors

Constructor and Description
`Road(City from, City to, int length)` Creates a new Road object.

- **Teksten i de røde bokse kopieres automatisk fra //*…*/ kommentarerne i jeres kode**
- **I summary-delene medtages kun første sætning i kommentaren**

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`int`	`compareTo(Road r)`
`boolean`	`equals(java.lang.Object otherObject)`
`City`	`getFrom()` Returns a reference to the City where this Road starts.
`int`	`getLength()` Returns the length of this Road.
`City`	`getTo()` Returns a reference to the City where this Road ends.
`int`	`hashCode()`
`java.lang.String`	`toString()`

Documentation view (details)

Constructor Detail

Road

```
public Road(City from,
            City to,
            int length)
```

Creates a new Road object.

Parameters:

`from` - The City in which this Road starts.

`to` - The City in which this Road ends.

`length` - The length of this Road.

Method Detail

getFrom

```
public City getFrom()
```

Returns a reference to the City where this Road starts.

Returns:

The from city of this Road.

- Teksten i de røde bokse kopieres automatisk fra //*…*/ kommentarerne i jeres kode
- I detail-delene medtages hele kommentaren
- Teksten i de grønne bokse kopieres automatisk fra @param og @return taggene i jeres kode

Test af computerspilsprojektet

- **I forhold til jeres tidligere afleveringsopgaver er computerspillet et stort og komplekst Java program**
  - Husk at teste jeres kode via testserveren
  - Hvis testserveren finder fejl, skal I gennemgå jeres kode og rette dem
- **Hvis jeres program fejler, kan det være vanskeligt at lokalisere fejlen**
  - Derfor afprøver testserveren hver enkelt af de klasser, som I selv har skrevet, sammen med vores løsning – således at man kan se, hvilke klasser, der er fejl i
  - Afprøvning foretages ved at udføre en række **regression tests** for konstruktørerne og metoderne i klassen

Brug af testserveren

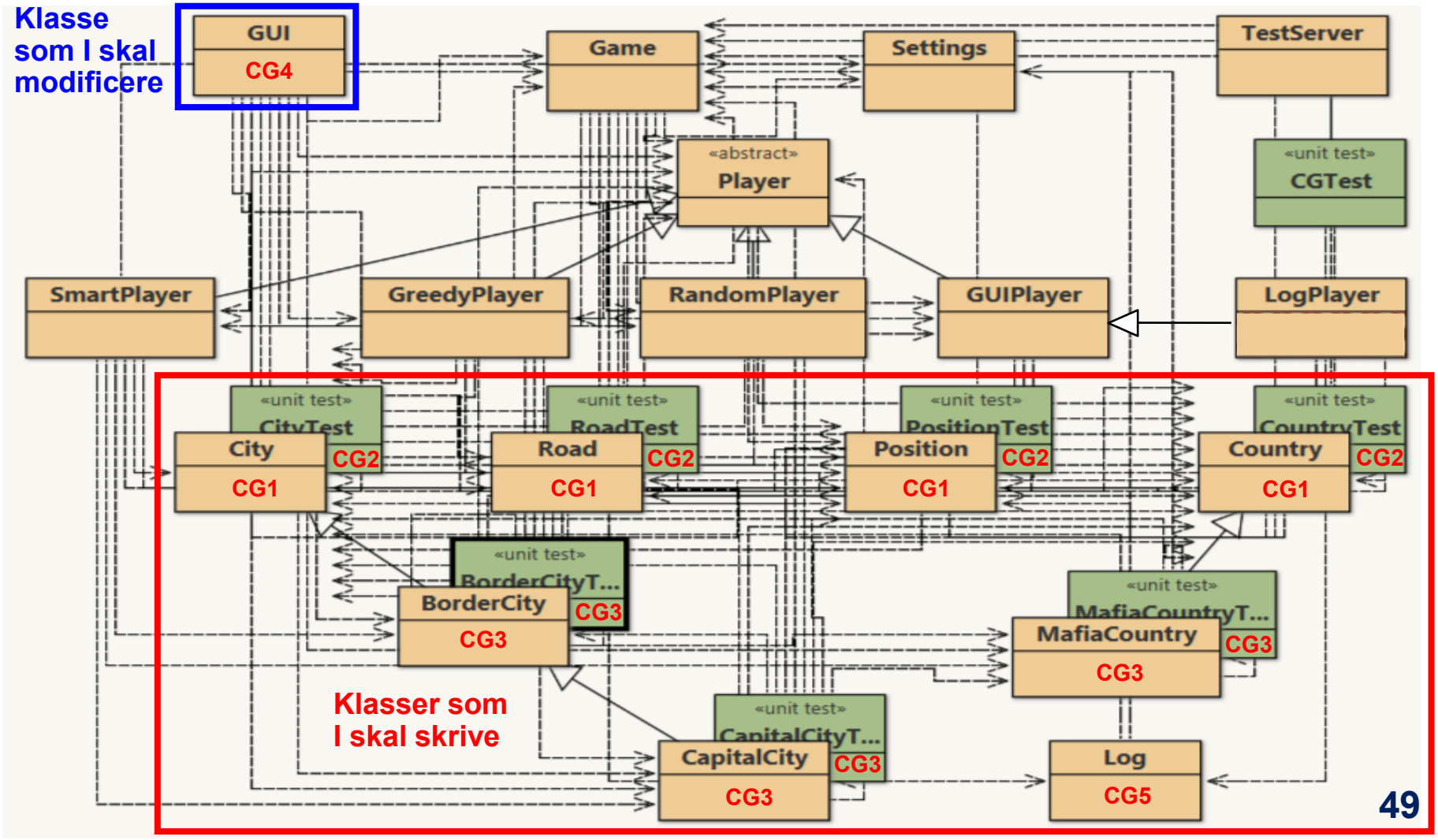
- **Man kan teste hvert enkelt opgave i Computerspil 1**
  - Man kan bruge testserveren på de **enkelte opgaver**, således at man efter hver opgave, straks kan teste, at det man har lavet i opgaven er korrekt
  - Men det er ikke altid muligt at gentage gamle tests
  - Når man f.eks. i opgave 5 ændrer City klassens konstruktør, så den tager en ekstra parameter, kan man ikke længere køre den gamle test fra opgave 1, uden at denne fejler
- **Brug testserveren med omtanke**
  - Når I får en fejlrapport, bør I rette **alle** de fejl, der rapporteres og teste, at rettelserne er korrekte, før I atter forsøger at køre testserveren
  - Hvis I blot retter en enkelt fejl ad gangen (uden selv at teste om rettelsen fungerer) kommer I let til at bruge alt for megen tid på at vente på, at testserveren genererer rapporter til jer (specielt hvis der er kø)

Computerspil 2-5

- **I de næste fire delafleveringer skal I**
  - Lave **regression tests** for alle konstruktører og metoder (Computerspil 2)
  - Bruge **nedarvning** (subklasser) og **dynamisk method lookup** (med overskrivning af metoder) til at strukturere jeres kode, således at der er flere forskellige slags lande og flere forskellige slags byer (Computerspil 3)
  - Udvide den **grafiske brugergrænseflade** med nogle ekstra knapper, labels og tekstfelter samt en menubar (Computerspil 4)
  - Optage spil ved hjælp af en **Log klasse**, hvis objekter kan gemmes i filsystemet, for sidenhen at blive genindlæst og afspillet (Computerspil 5)

Klassediagram

- **Det færdig projekt indeholder 25 klasser**
  - I skal skrive 8 almindelige klasser og 7 testklaser samt modificere 1 klasse



Opsummering

- **Nedarvning**
  - Subklasser
  - Forskellen på en variabels statiske og dynamiske type
  - Subtyper og Liskovs substitutionsprincip
  - Typeskift (type cast)
- **Metoder i subklasser**
  - Dynamic method lookup i forbindelse med nedarvning og overskrivning
- **Object klassen**
  - Superklasse for alle klasser
  - Indeholder en række nyttige metoder, bl.a. toString, equals, hashCode og getClass
- **Protected access**
  - Alternativ til public og private
- **Projektopgave om computerspil**

Det var alt for nu….. … spørgsmål