Læringsmål for forelesningen

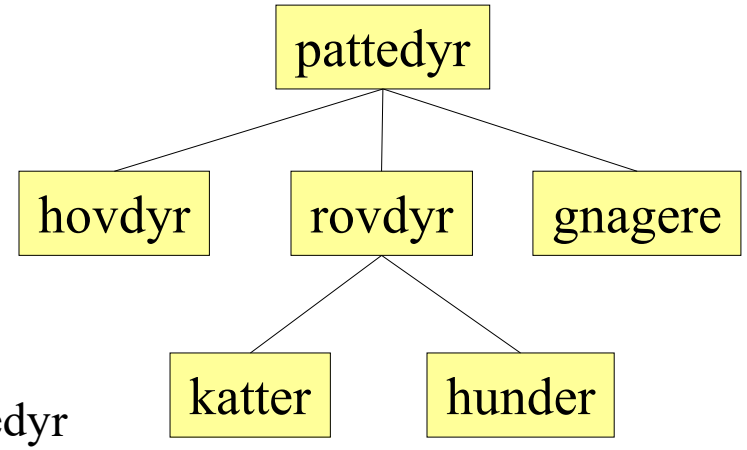
- Objektorientering
  - Arv
- Java-programmering
  - Arv i Java, hvilke regler
- VS Code
  - Undersøke klassehierarki i editoren

Arv (eng: inheritance)

- Arv gir muligheten til å definere nye klasser, basert på eksisterende
  - *spesialisere*
  - *utvide*
- Subklasser arver alle egenskapene til superklassen
  - Subklassen inneholder alle felt og metoder (inkl. konstruktører) deklarert i både superklassen
  - alle grensesnitt arves, dvs. garanti for implementerte metoder
  - alle **public**-, **protected**- og (**default-**) felt og metoder i superklassen, kan refereres til i subklassens ***kode***
  - Dvs. **private** felt og metoder kan ikke refereres til eller kalles i subklassens kode, selv om de finnes.

Hvorfor arv?

- Klassifisering i klassehierarkier er en intuitiv organisering av fenomener i verden
  - hovdyr er et pattedyr
  - katt og hund er et rovdyr er et pattedyr
  - gnager er et pattedyr



- Arv gir mulighet for gjenbruk av kode
- Forenkler strukturering av komplekse problemer og programmer

Arv: «…er en…»

- Arv (er ment å) modellerer «er en» relasjon, som er ment å være sterkere enn at man oppfyller en kontrakt.
- Fordelen fremfor at klasser implementerer et felles grensesnitt, er at vi gjenbruker felt og metoder.
- Ulempen er at koblingen blir sterk, og en må passe mer på, f.eks. når en endrer koden.

Hvorfor arv?

Selv om vi har et grensesnitt vi kunne implementere, så er det eksempler hvor det er hensiktsmessig å utvide eksisterende klasser som implementerer grensesnittet.

Pass da på:

- Liskovs substitusjonsprinsipp, dvs. at det man redefinerer ikke gir ødelegger logikken i superklasser, eller eksponerer implementasjonsdetaljer som er innkapslet i superklassen.
- Dette kan virke unødvendig hvis en f.eks. bare skal bruke den nye klassen i en begrenset sammenheng, men …

En må da også vurdere om en vil i stedet bruke komposisjon: Da må en skrive mer kode, men har svakere kobling, og

Hvorfor arv?

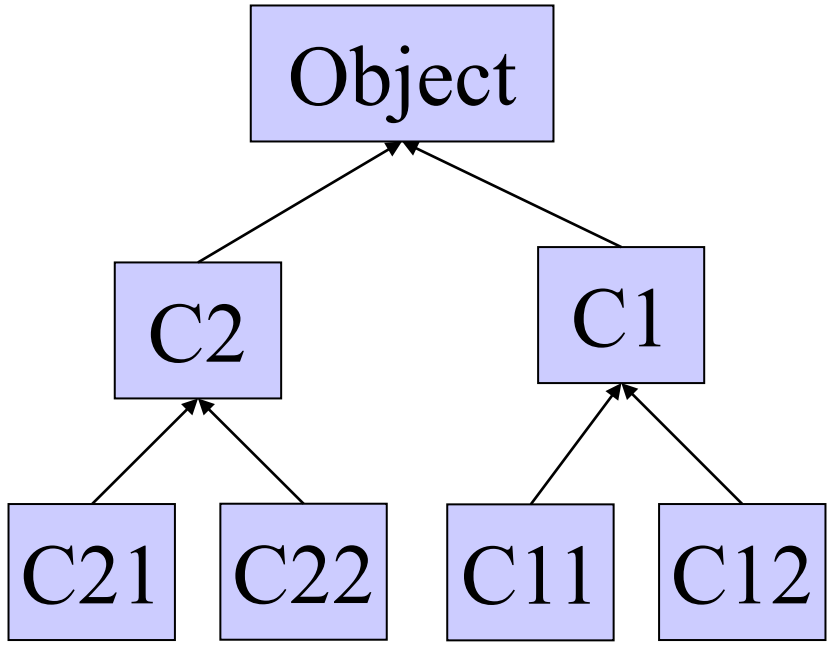
Noen ganger er vi «nødt» til å bruke arv: Hvis vi vil ha et modifisert objekt som skal fungere i eksisterende kode

Eksempler:

- I Java arver vi jo alltid fra Object, eller en subklasse av Object, så vi har metodene toString, equals, …
- Egendefinert Exception-klasse
- Generelt, hvor vi ønsker å plugge inn egen klasse et sted hvor variabelen/parameteren er spesifisert som en eksisterende klasse, og ikke et grensesnitt (interface)
- Enhetstester i Junit

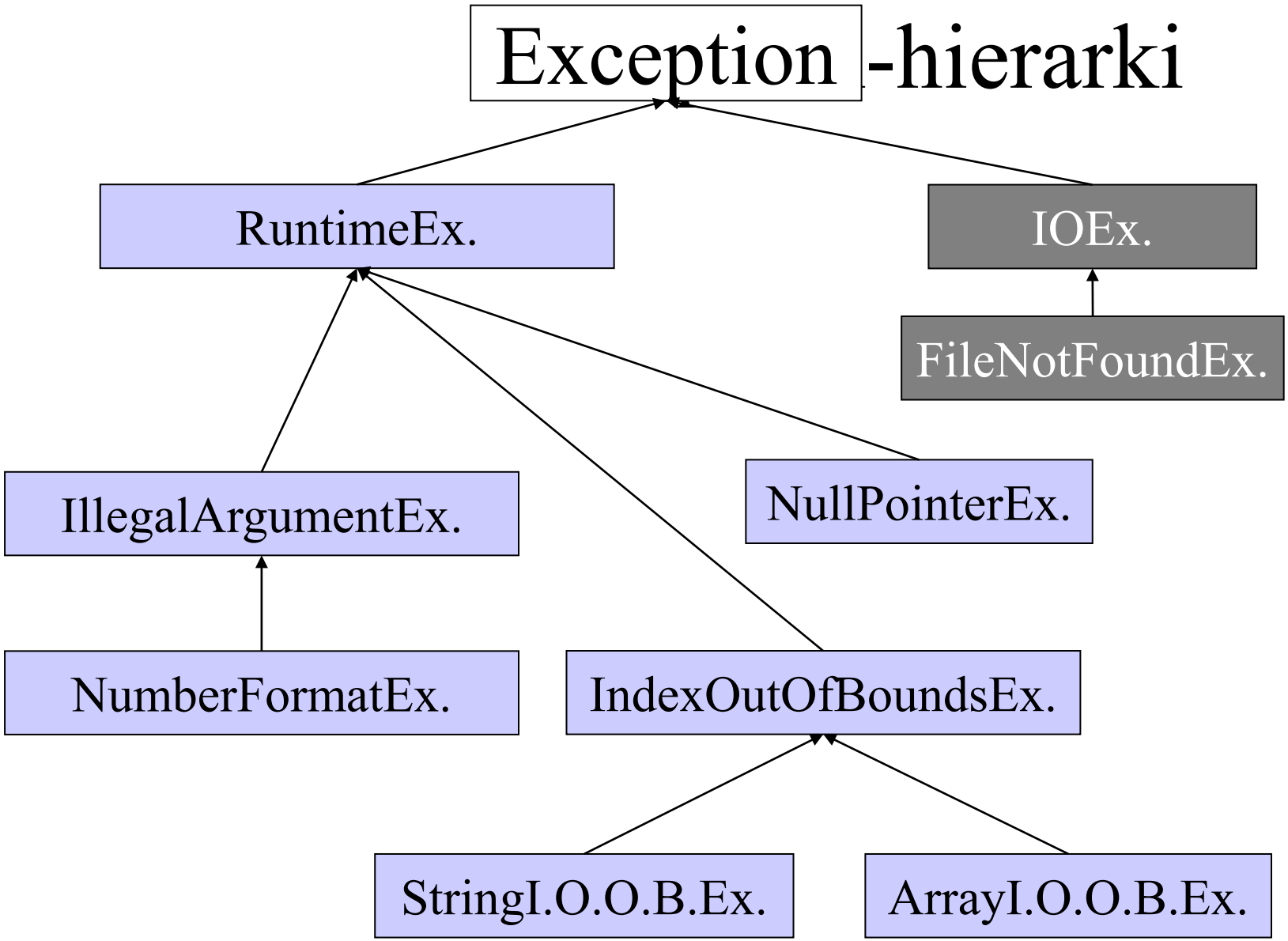
Arv skjematisk

- Klasser struktureres i et hierarki, f.eks. C1, C11, C12, C2, C21, C22
- Et objekt laget som en instans av en klasse C, er `instanceof` C *og alle C sine superklasser*



⟶ peker på superklassen

```
C21 c21 = new C21();

c21 instanceof C2 = = true
```


Exception -hierarki
RuntimeEx.
IOEx.
FileNotFoundEx.
IllegalArgumentEx.
NullPointerEx.
NumberFormatEx.
IndexOutOfBoundsEx.
StringI.O.O.B.Ex.
ArrayI.O.O.B.Ex.

Typisk to typer bruk av arv

- Spesialisering av eksisterende klasse(r)
  - noen har allerede laget en eller flere klasser i et hierarki, og vår subklasse skal spesialisere den som passer best
  - eksempel:
    - Exception/RuntimeException
- Vi lager vårt eget klassehierarki, med tett koblede klasser
  - superklasse Bok, subklasser Ordbok og Tegneseriealbum

Spesialisering/subklassing av eksisterende klasse(r)

- Utgangspunkt:
  - Eksisterende klassehierarki
  - Andre klasser som bruker klassene i hierarkiet på en veldefinert måte
- Eksempler:
  - Object-klassen og toString (som brukes av PrintStream/PrintWriter-klassene)
  - Exception/RuntimeException og unntakssystemet

Object – øverst i klassehierarkiet

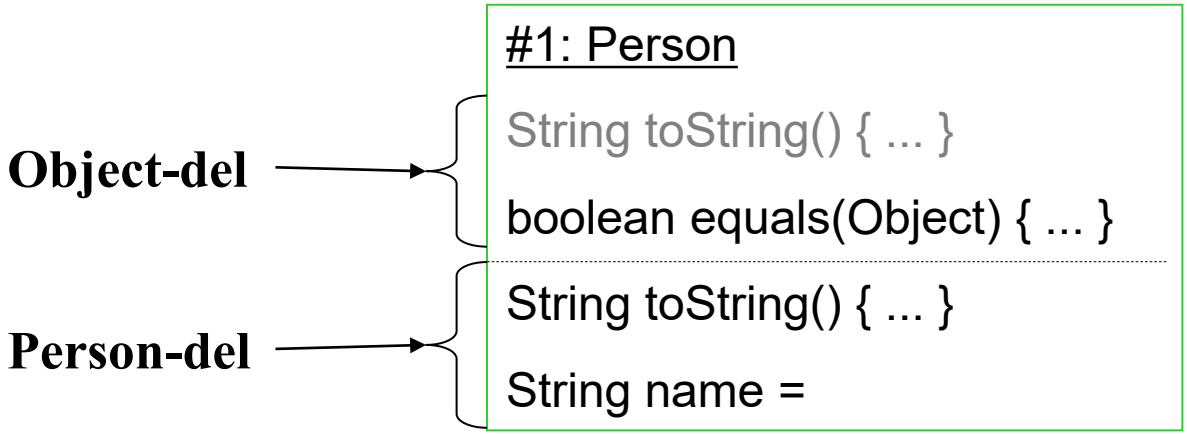
- Object
  - superklassen til alle klasser som ikke eksplitt angir en superklasse
  - definerer en del metoder, som all kode kan forvente finnes, og som kan være nyttige å redefinere
- toString()
  - brukes når en trenger en tekstlig representasjon av et objekt
- equals(Object)
  - angir om argumentet i praksis er lik dette objektet (this)
  - symmetrisk, **o1.equals(o2) == o2.equals(o1)**
- hashCode()
  - beregner en **int** som skal være mest mulig unikt for den delen av objekt-innholdet som **equals**-metoden ser på
  - **o1.equals(o2)** betyr/krever at **o1.hashCode() == o2.hashCode(),** men ikke nødvendigvis omvendt
  - brukes ifm. ordning av objekter, f.eks. av **HashMap**

Object.toString(), +-operatoren og PrintStream/PrintWriter-klassene

- +-operatoren baserer seg på at **toString()**-metoden kan brukes for å lage en tekstlig representasjon av et objekt
  - `"hei" + obj` betyr omtrent `"hei".concat(obj.toString())`
- PrintStream/PrintWriter-klassene baserer seg på **toString()**-metoden
  - når en skriver ut objekter, så vil **toString()**-metoden bli brukt til å konvertere objektet til tekst
- Ved riktig redefinering av **toString()** så vil vår egen (**Object**-sub)klasse gli pent inn i Java sin objekt-til-tekst-konverteringsmekanisme

Redefinering av toString()

```
public class Person {
        private String name;
        public String toString() {
            return "[Person " + name + "]";
        }
}
```



- Person har nå to **toString()**-metoder, men kun den i subklassen er synlig *utenifra*
  - en metode i en subklasse *skygger for* en metode med samme navn og parametre i superklassen
  - subklassen kan referere til superklassens ved bruk av **`super.toString()`**
  - **`Se også MyObject.java`**

Exception/RuntimeException og unntakssystemet

- **throw**, **catch** og **throws** forventer en **Exception**(-instans)
  - **throw** new IllegalArgumentException(...)
  - **catch** (IllegalArgumentException iae)
  - **throws** IllegalArgumentException
- Viktig distinksjon mellom checked og unchecked exceptions:
  - **RuntimeException** og subklassene er unchecked
  - **Exception** og andre subklasser er checked
- Ved riktig bruk av subklassing (visse regler må overholdes) så vil vår egen (**Runtime**)**Exception**-subklasse gli pent inn i Java sitt unntakssystem

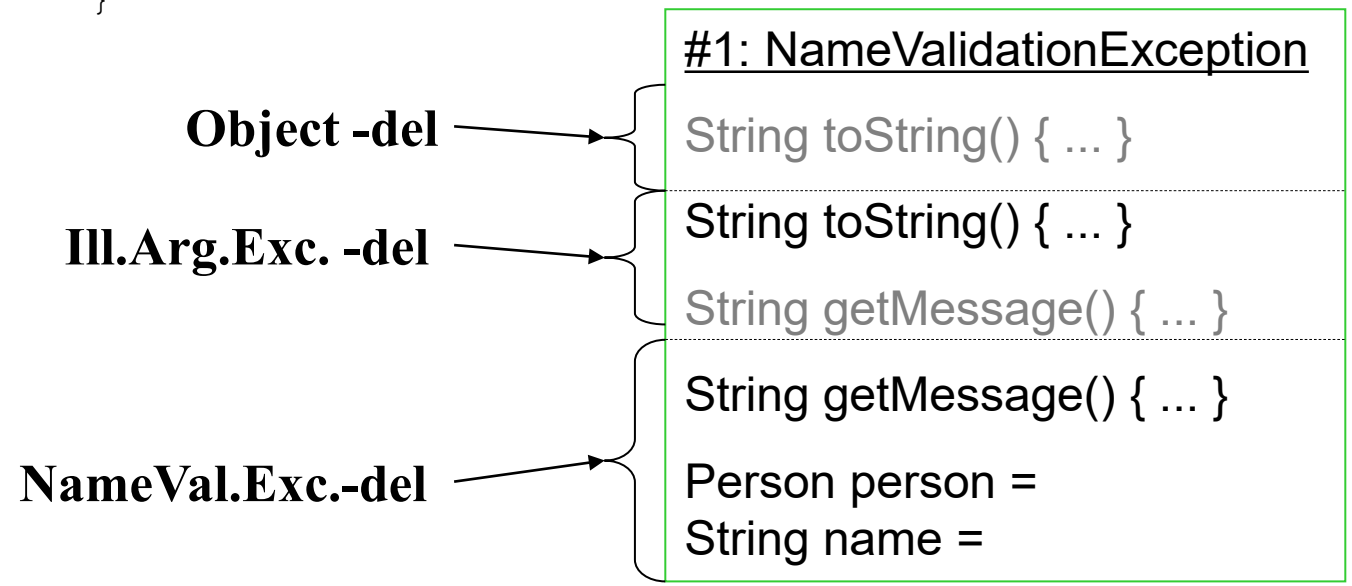
NameValidationException

- Person har en setName-metode og regler for hvilke tegn som tillates i et navn
- Hvis reglene ikke overholdes, så skal det utløses en ny type unntak: **NameValidationException**
- **NameValidationException** lagrer informasjon om det nye (ulovlige navnet) og **Person**-objektet

- Den nye klassen arver fra den mest spesifikke eksisterende klassen som den har et «is-a» forhold til. Her arver vi fra IllegalArgumentException.
- Se Person.java

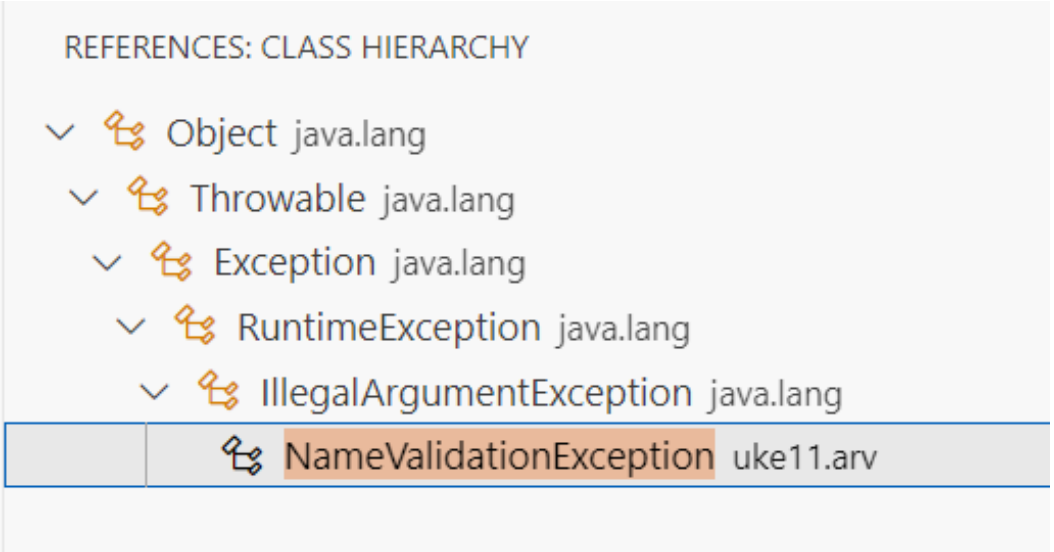
NameValidationException

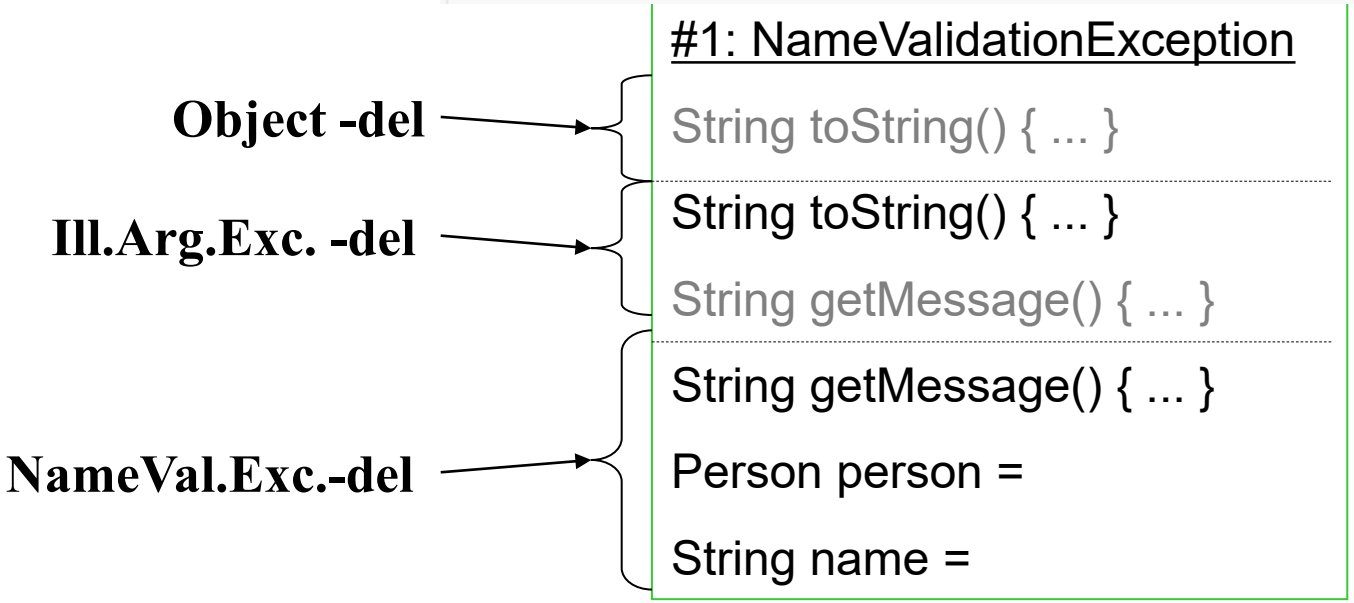
```
public class NameValidationException extends IllegalArgumentException {

        private Person person;
        private String illegalName;

        public NameValidationException(Person person, String illegalName) {
                this.person = person;
                this.illegalName = illegalName;
        }
        public String getMessage() {
                return illegalName + " is an illegal name for " + person;
        }
}
```



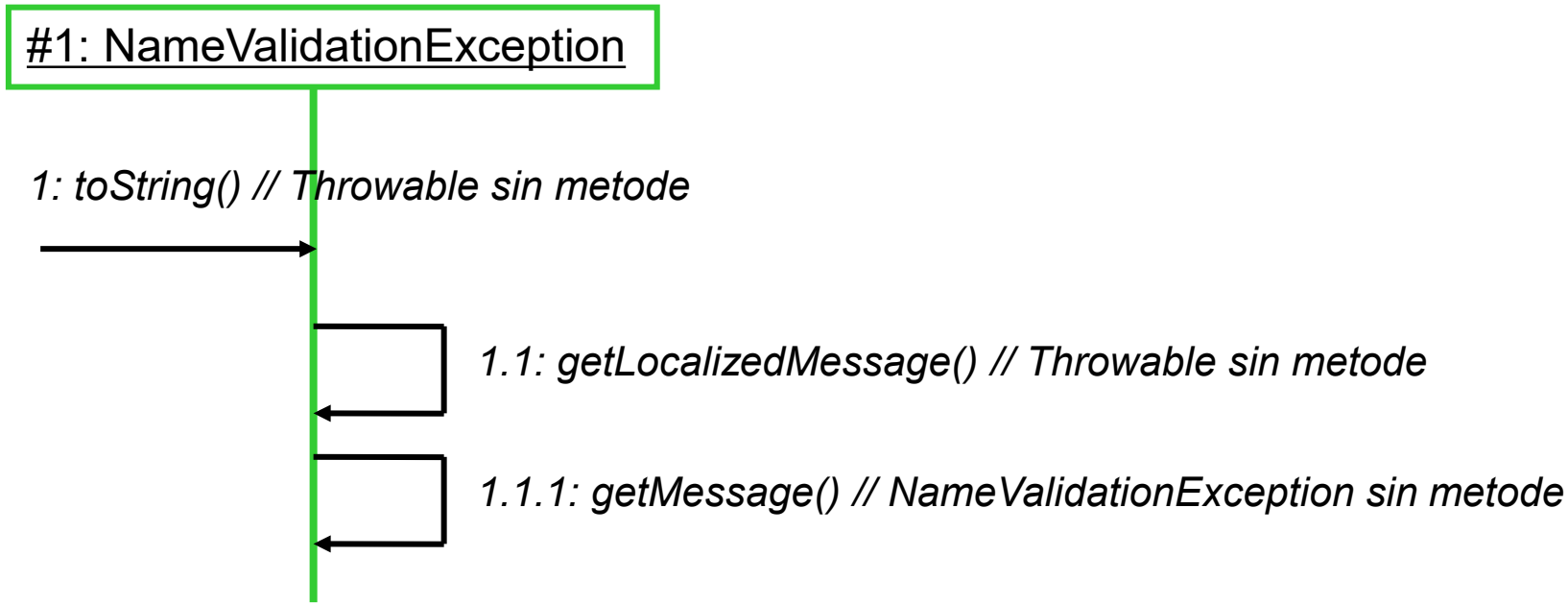
Hierarchy-view

- Hvordan ser objektet vårt ut?
- Høyreklikk på klassenavnet i kildekoden, og velg «Show Type Hierarchy»





Sekvensdiagram



- Ved å redefinere riktig metode kan vi skyte vår egen kode inn i en kallsekvens på strategisk riktig sted
- Dette krever at vi forstår hvordan superklassen bruker sine egne metoder

Substitusjonsprinsippet

- Viktig prinsipp som gjør det enklere å bruke arv riktig:
  - Der en bruker en instans av superklassen, skal en kunne bruke en instans av en subklassen
  - Dvs. subklassen skal følge de samme *regler for oppførsel* som en superklasse,
- Prinsipper går på å ivareta korrekt/forventet oppførsel, når arv/subklasser introduseres.
- Dette er ikke en feil som oppdages av Java (kompileringsfeil)
- Se https://en.wikipedia.org/wiki/Liskov_substitution_principle

Eksempel

Eget klassehierarki – utnytter likehetstrekk mellom klasser

- bok-eksempel

Eget klassehierarki, med tett koblede klasser

- En bok har en tittel

```
public class Bok {
    String tittel;
}
```

- *Ordbøker* og *tegneseriealbum* er bøker
  - *Bok* er et generelt begrep
    - kun ett felt, **tittel**, med visse begrensninger på bokstavene som kan brukes
  - En *ordbok* er en spesiell type bok
    - lar **Ordbok**-klassen *arve* egenskaper fra **Bok**-klassen
    - definerer i tillegg feltet **antallOrd**
  - Et *tegneseriealbum* er også en *bok*
    - lar Tegneseriealbum-klassen *arve* egenskaper fra **Bok**-klassen
    - definerer i tillegg feltet **antallStriper**

Ordbok og Tegneseriealbum *arver* fra Bok

- Ordbok
  - **`extends`** `Bok`, arver dermed **tittel**-feltet
  - definerer også eget **antallOrd**-felt

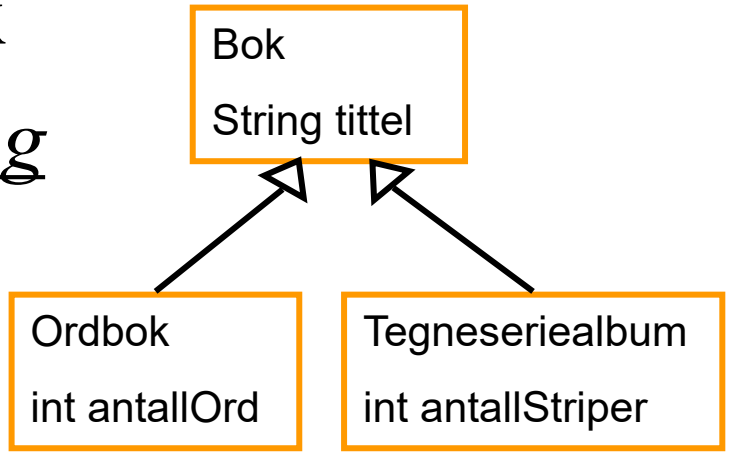
```
public class Ordbok extends Bok {

   private int antallOrd;
```

- Tegneseriealbum
  - **`extends`** `Bok`, arver dermed **tittel**-feltet
  - definerer også eget **antallStriper**-felt

```
public class Tegneseriealbum extends Bok {

   private int antallStriper;
```

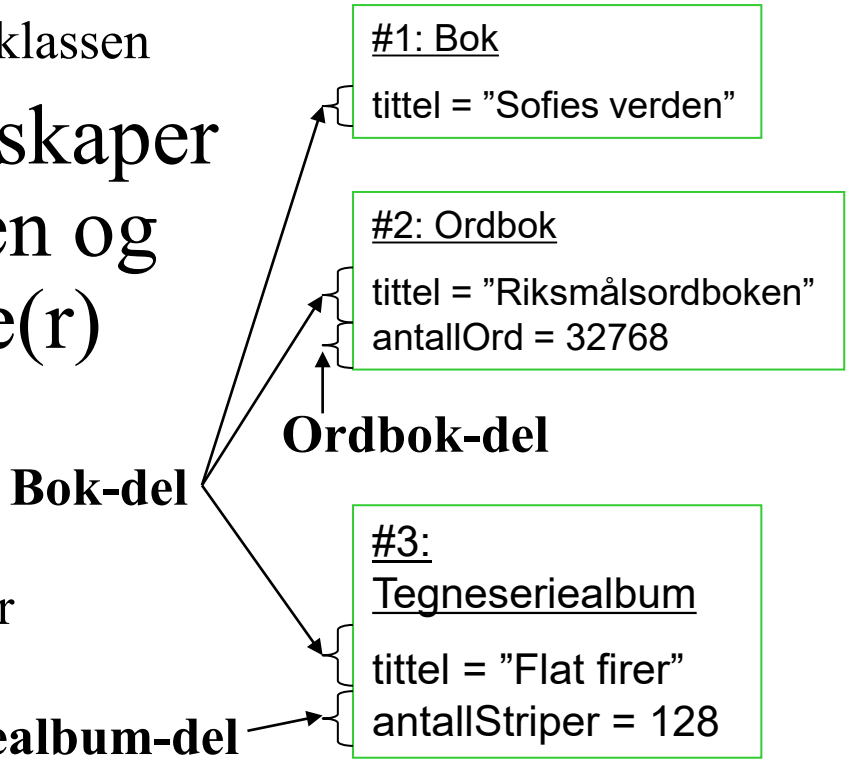
Begreper knyttet til arv

- Vi sier at **Bok** er *superklassen* til **Ordbok** og **Tegneseriealbum**, mens **Ordbok** og **Tegneseriealbum** er *subklasser* av **Bok**
- **Ordbok** og **Tegneseriealbum** er *spesialiseringer* av **Bok**
- **Bok** er en *generalisering* av **Ordbok** og **Tegneseriealbum**



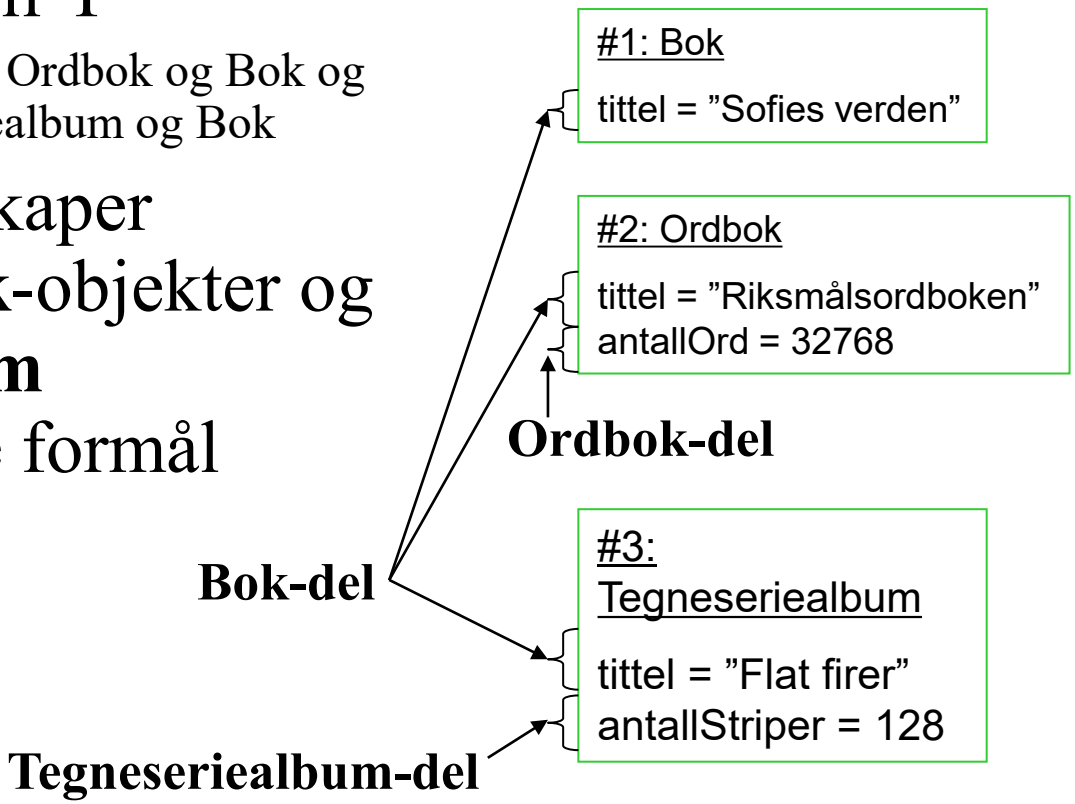
Instanser og klassetilhørighet

- Et objekt *instansieres* av én bestemt klasse med **new**
  - object.getClass() gir oss denne klassen
- Objektet har alle egenskaper definert i denne klassen og (alle) dens superklasse(r)
  - #1 er en Bok og har feltet tittel
  - #2 er en Ordbok og har feltene tittel og antallOrd
  - #3 er et Tegneseriealbum og har feltene tittel og antallStriper



Instanser og klassetilhørighet

- Vi kan si at en Y-instans også er en X, dersom X er (en av) superklassen(e) til Y
  - #1 er Bok, #2 er både Ordbok og Bok og #3 er både Tegneseriealbum og Bok
- Arving av egenskaper sikrer at **Ordbok**-objekter og **Tegneseriealbum** for alle praktiske formål *er* **Bok**-objekter
- *Substitusjons-prinsippet?*
- *Person.java*



Arv av metoder

- En subklasse arver alle egenskaper til superklassen
  - felter og (vanlige) metoder
  - Konstruktører arves IKKE, men en blir alltid kalt først fra subklassens konstruktører.
- Metodene som arves fra en superklasse, kan (naturlig nok) kun referere til felt og metoder som er deklarert i superklassen
  - f.eks. vil get- og set-metoder i **Bok** virke som før, de leser og setter felter i **Bok**-delen av en **Ordbok**

Arv og redefinering av metoder

- Flere metoder vi har brukt er egentlig arvet
  - `toString`- og `equals`-metodene, som alle objekter har, er egentlig arvet fra java.lang.Object
- Dersom en definerer samme metode i en subklasse, vil denne brukes istedenfor den i superklassen, jfr. egendefinerte **toString**-metoder.
- Vi sier at en metode som er redefinert i en subklasse, *skygger for* (eng: overrides) den i superklassen, ved at den gjelder i stedet for
- Skygging er uavhengig av hvem som kaller og kalles og fungerer også internt i en klasse

Arving og redefinering av toString-metoden

```
public static void main(String args[]) {
    Bok bok = new Bok();
    Ordbok ordbok = new Ordbok();

    System.out.println(bok);
    System.out.println(ordbok);
}
```

Run

```
Problems Javadoc Declaration Console
<terminated> Bibliotekprogram [Java Application] C:\Program Files\Java\jre
forelesinger.uke06.Bok@45a877
forelesinger.uke06.Ordbok@1372a1a
```

- Redefinerer toString i Bok-klassen:

```
public String toString() {
    return "[Bok]";
}
```

Run

```
Problems Javadoc Declaration Console
<terminated> Bibliotekprogram [Java Application] C:\Prog
Bok
Bok
```

- Redefinerer toString i Ordbok-klassen:

```
public String toString() {
    return "[Ordbok]";
}
```

Run

```
Problems Javadoc Declaration Console
<terminated> Bibliotekprogram [Java Application] C:\Prog
Bok
Ordbok
```

Redefinering av metoder

- Redefinerte metoder *skygger for* arvede, slik at metoden definert i subklassen kalles istedenfor den i superklassen
  - **Bok** sin **toString()**-metode blir brukt istedenfor den som arves fra **Object**

- Hvilken metode som kalles er avhengig av klassen som objektet er laget av, *ikke* typen til referansen

```
Ordbok ordbok = new Ordbok();
// kaller toString-metoden definert i Ordbok
System.out.println(ordbok);

Bok bok = ordbok;
// kaller fortsatt toString-metoden definert i Ordbok
System.out.println(bok);

Object objekt = bok;
// kaller fortsatt toString-metoden definert i Ordbok
System.out.println(objekt);
```

Mer om redefinering av metoder (eng: overriding)

- Ved subklassing arves både felter og metoder.
- Arv av metoder gjør det mulig å bruke superklassens metoder på instanser av en subklasse
  - **toString()** definert i **Object** kan brukes på instanser av alle klasser
  - **getTittel()** definert i **Bok** kan brukes på alle **Ordbok**-instanser
- Dersom en ønsker å erstatte en arvet metode med en egen implementasjon, defineres bare en metode med samme synlighet, returverdi og parametre i subklassen
  - **Bok** kan definere en alternativ **toString()**-metode med *signaturen* `public String toString()`, som setter sammen relevante **Bok**-felter

Hva med innkapsling?

- Innkapslingsprinsippet gjelder fortsatt.
- En ny synlighetsmodifikator:
  `protected`
  innføres for å definere hvilke egenskaper i superklassen som blir synlige i subklassene

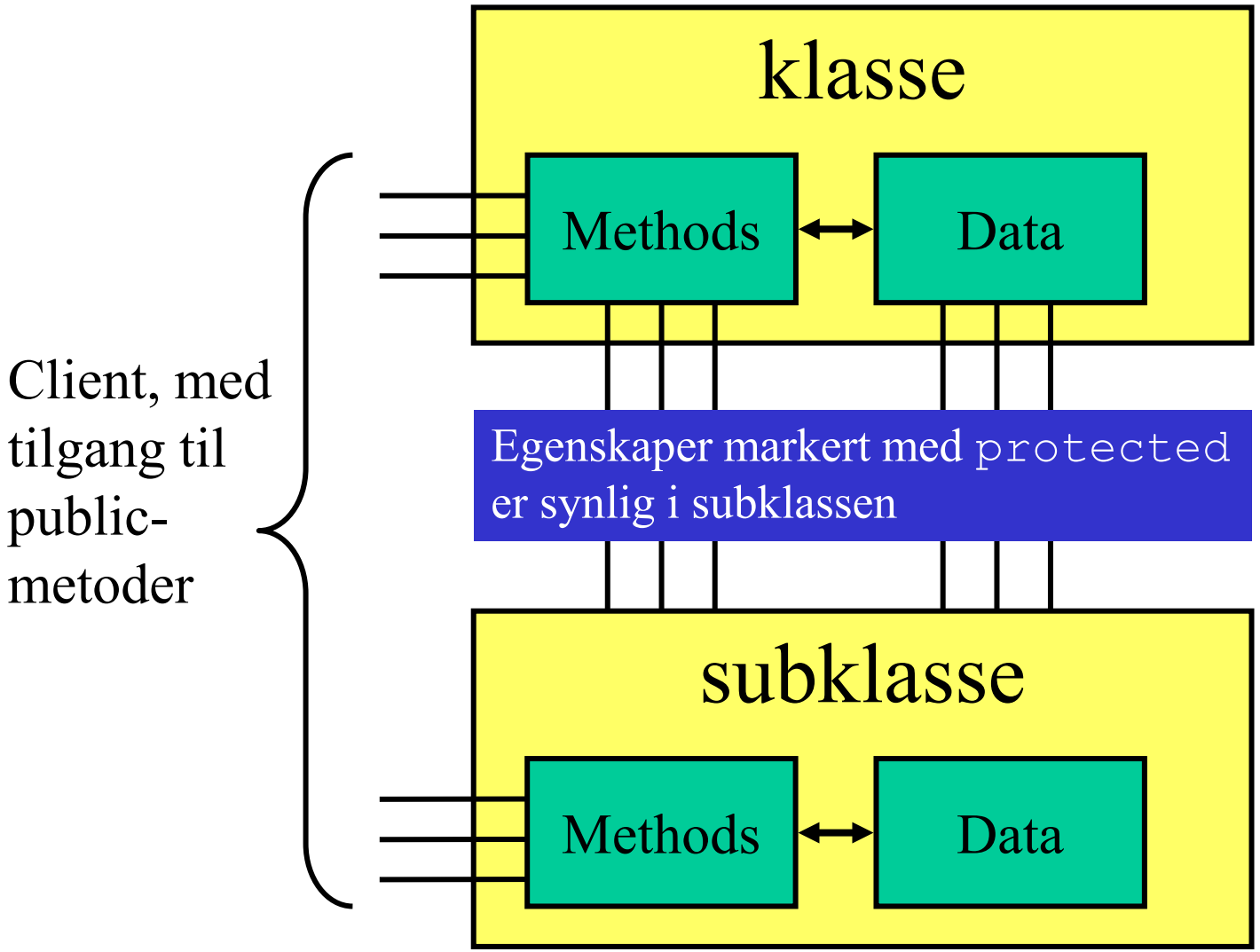
Synlighet av arvede egenskaper

- Arv krever introduksjon av en ny grad av synlighet, mellom **`public`** og **`private`**
- Egenskaper markert som **`protected`** i en superklasse er synlige kun i subklassene
  - felter markert med **`protected`** kan både leses og endres i subklassen
  - metoder markert med **`protected`** kan både kalles og redefineres i subklassen
- Merk forskjellen mellom *synlighet* og *eksistens*:
  - et private-felt er ikke *synlig* i subklassens, men *finnes* fortsatt
  - en private-metode er ikke synlig i subklassens kode, men finnes fortsatt og kan kalles indirekte via synlige metoder

Synlighet av arvede egenskaper

- **NB!** **`protected`**-egenskaper skaper en tettere og ofte uønsket binding mellom super- og subklasser
  - **`protected`** skal brukes i tilfeller hvor direkte tilgang til superklassens egenskaper er nødvendig
  - metoder som det er meningen/naturlig at skal redefineres i subklasser bør markeres som **`protected`**
  - felt bør kun i nødsfall markeres som **`protected`**
  - **`Merk:`** `package og protected`
  - **`Vite mer?`** `Det blir komplisert`...

Innkapsling og arv



Legger til valideringskode i Bok sin setTittel-metode

- Sjekker om tittelen inneholder ulovlige bokstaver

```
private static String ulovligeBokstaver = "#¤{[]}";

public void setTittel(String tittel) {
    for (int i = 0; i < ulovligeBokstaver.length(); i++) {
        if (tittel.indexOf(ulovligeBokstaver.charAt(i)) >= 0) {
            return;
        }
    }
    this.tittel = tittel;
}
```

- Det er viktig at subklasser ikke omgår denne sjekken, men er med på å sikre at reglene for **Bok** sin oppførsel følges

Oppgave: Definer en alternativ **setTittel**-metode for **Ordbok**, som sikrer at **tittel**en til en **Ordbok** slutter på ”ordbok”

Sjekker om **tittel** slutter på ”ordbok” og i tilfelle ikke, legger til ”ordbok”

Alternativ setTittel-metode

- Redefinerer setTittel i Ordbok-klassen:
  - *samme signatur* som **setTittel** i **Bok**-klassen
  - sjekker endelsen og legger evt. til ”ordbok”-endelsen

```
public void setTittel(String tittel) {
    if (! tittel.endsWith("ordbok")) {
        tittel += "ordbok";
    }
    this.tittel = tittel;
}
```

- To problemer:
  - tittel-feltet må være `protected` og ikke `private`, for at Ordbok sin **setTittel**-metode skal kunne endre tittel-feltet
  - Bok sin **setTittel**-metode inneholder egen valideringskode, som burde vært duplisert i **Ordbok** sin **setTittel**-metode
- Forslag til løsning?

Ved redefinering av en metode i en subklasse, hadde det vært fint å kunne bruke superklassen sin metode, slik at en slipper å duplisere kode (og logikk)

Java gir oss muligheten til å kalle metoder i superklassen fra en subklasse, selv om disse er redefinert i subklassen

super.<metode>(...)-konstruksjonen

- Med `super` som prefiks kan en kalle en metode i superklassen, selv om denne er redefinert i subklassen!

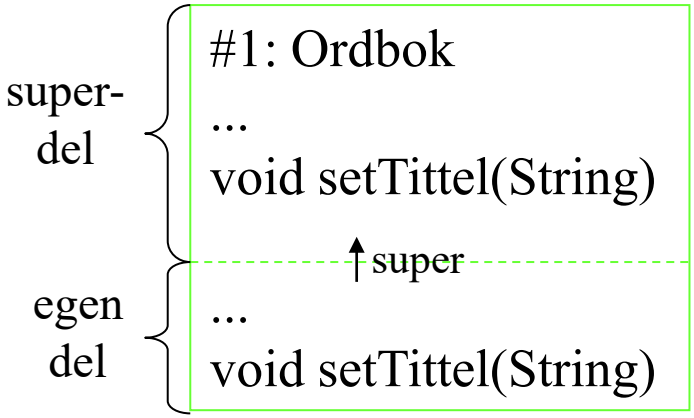
```
public void setTittel(String tittel) {
    if (! tittel.endsWith("ordbok")) {
        tittel += "ordbok";
    }
    super.setTittel(tittel);
}
```

- Dette løser begge våre problemer
  - vi kan deklarere tittel-feltet som `private`, siden vi kan sette den vha. Bok (altså superklassen) sin setTittel-metode
  - vi kan kalle Bok sin setTittel-metode med en evt. *endret* tittel, og være sikker på at denne valideres iht. Bok sine egne regler
- Merk at super.<metode>(...)
  - kan kalles fra *alle* metoder i subklassen, ikke bare fra <metode>
  - kun kan brukes for å kalle metoder definert i den direkte superklassen, og ikke metoder i super-superklassen
- this-referansen vil være den samme

bok_avansert

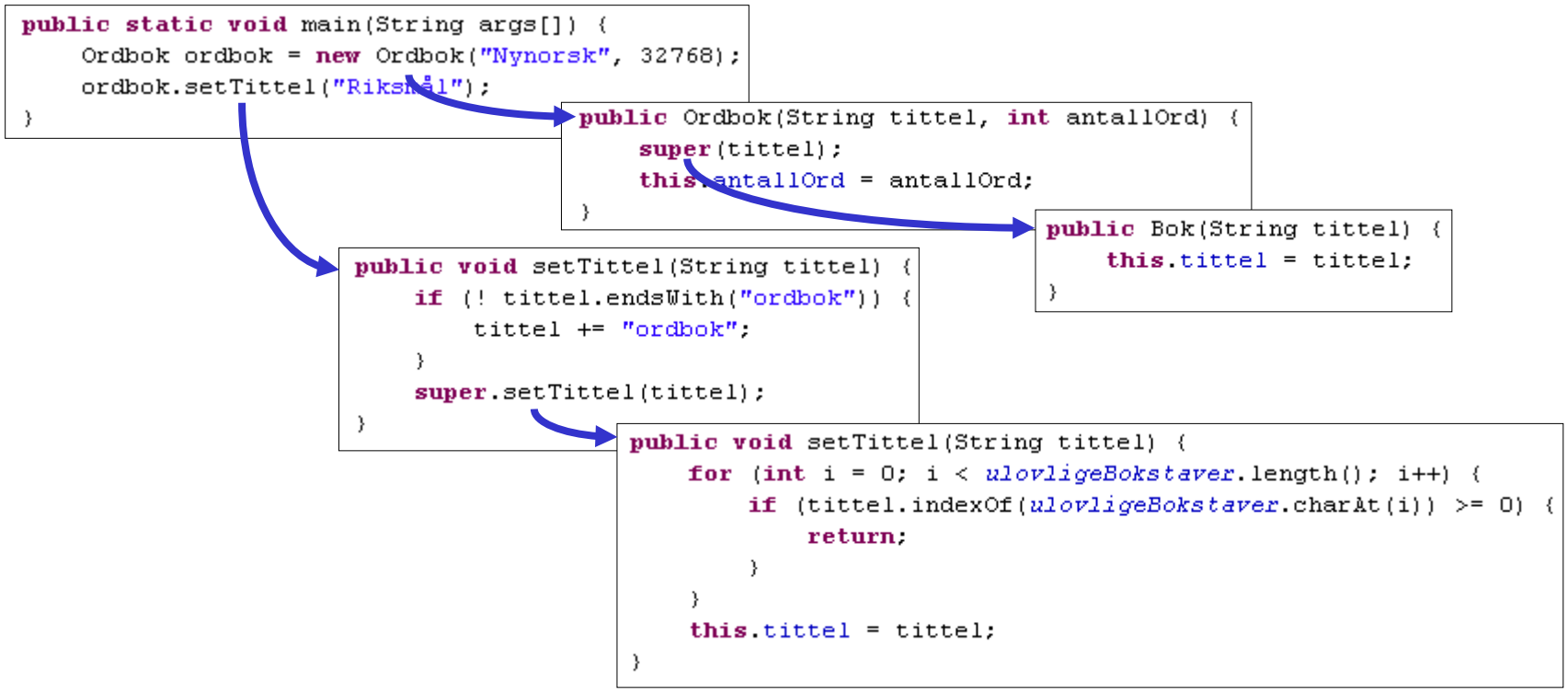
`super` gjør det mulig å (gjen)bruke superklassens metoder, selv om de redefineres i subklassen

Det er som om Ordbok består av én Bok- eller super-del og én egen Ordbok-del, og super gir mulighet til å referere til metoder i super-delen



Kallsekvens

- main-metode
  - lager en Ordbok-instans og endrer tittelen



Redefinering vs. overlasting (overriding vs. overloading)

- Redefinering
  - Metoder med like navn og parameterlister i super- og subklasser
  - Hvilken *faktisk instans* metoden kalles på, avgjør hvilken klasse sin metode som faktisk kalles
  - Metoden som kalles avgjøres dynamisk ved utførelsen
  - Dette kalles **polymorfi**!
- Overlasting
  - Metoder med like navn, men ulike parameterlister (antall og type), som ikke har annet felles enn navnet.
  - Hvilken metode(variant) som kalles, avgjøres av den *deklarerte typen* til parametrene
  - Metoden som kalles avgjøres statisk ved kompilering, den velger alltid den mest spesifikke av de mulige metodene
- Kombinasjonen kan være forvirrende...

Konstruktører kan også overlastes

- Flere konstruktører med ulike parameterlister kan defineres
- En konstruktør kan kalle en annen definert i samme klasse med **this(...)**

```
public Ordbok(String tittel) {
    super(tittel);
}

public Ordbok(String tittel, int antallOrd) {
    this(tittel);
    this.antallOrd = antallOrd;
}
```

- Bruk av **this(...)** er analogt med bruk av **super(...)**: Den navnløse konstruktørmetoden i egen (**this**) eller superklassen (**super**) kalles

- NB! Dersom en konstruktør må kalle en annen, må dette skje i første linje!!!

Arv og konstruktører

- En konstruktør er en spesiell metode som kalles ifm. initialisering av en nyopprettet instans (ved new …)
- `super(...)` MÅ brukes for å kalle superklassens konstruktør, slik at en sikrer at også super-delen blir initialisert.
- `super(...)` MÅ stå *først* i konstruktøren, for å sikre at superklassens felter initialiseres før subklassens!
- Hvis superklassen har konstruktør uten parametre, så blir denne kalt om det ikke spesifiseres i subklassens konstruktør

```
public Bok(String tittel) {
    this.tittel = tittel;
}
```

```
public Ordbok(String tittel, int antallOrd) {
    super(tittel);
    this.antallOrd = antallOrd;
}
```

- I Bok-klassen:
  - initialiserer tittel-feltet
- I Ordbok-klassen:
  - kaller superklassen sin konstruktør
  - initialiserer antallOrd-feltet
  - alternativt kunne en fjernet antallOrd-parametret og initialiseringen av det, men konstruktøren MÅ defineres

Initialisering litt komplisert?

- Ved bruk av **new** utføres følgende:
  - initialiseringskode (utenfor konstruktører) utføres først
    - **List<String> strings = new ArrayList<>();**
    - **Map<Person, String> nicknames = new HashMap<>();**

```
{
  // initialiseringsblokk for å fylle nicknames-map
  nicknames.put(hallvard, “hal”);
}
```

  - så utføres selve konstruktøren
  - Hvis et felt settes her, så er det den som blir gjeldende.

Initialisering litt komplisert?

Initialiseringskode og konstruktører i superklassen er garantert å kjøre før tilsvarende i subklassen

- Hvis ikke super() angis eksplisitt, blir super() uten argumenter likevel kalt som første linje i subklassens konstruktør. Så hvis en slik ikke finnes, MÅ vi eksplisitt kalle en med super(…)

DETTE KAN SKJE:

super-konstruktøren kan kalle en overlagret metode i subklassen FØR initialiseringskode og konstruktør i subklassen er utført.

KONSEKVENS?

Eksempel

- Bok-superklasse
  - **setTitle(String)** kaller **isValidTitle(String)**
  - konstruktør kaller **setTitle**
- OrdBok-subklasse
  - overlagrer **isValidTitle(String)**
  - implisitt eller eksplisitt **super(...)**-kall vil altså kalle **isValidTitle(...)**, før resten av konstruktøren er utført
  - hva skjer hvis **isValidTitle** bruker et felt som ennå ikke er initialisert...
  - Se pakken ordbok_feil (eksempelet er forenklet fra andre Bok)

Læringsmål for forelesningen

- Objektorientering
  - Arv
- Java-programmering
  - Arv i Java
- VS Code
  - Undersøke instanser

Neste uke

- Objektorientering
  - Arv vs komposisjon
- Java-programmering
  - Mer om arv, Abstrakte klasser, arv og grensesnitt