Litt om *Arv* I OO

Mer om dette i slutten av semesteret

Arv (eng: inheritance)

- En grunnleggende OO-mekanisme
- Definerer nye klasser fra eksisterende
  - *spesialisere* eksisterende klasser, og/eller
  - *utvide* eksisterende klasser
- *Subklassen* arver alle egenskapene til *superklassen*
  - *instanser* av subklassen vil inneholde alle felt og metoder (inkl. konstruktører) deklarert i både superklassen og subklassen
  - alle grensesnitt arves, dvs. garanti for implementerte metoder
  - En subklasse kan referere til alle **public**- og **protected** felt og metoder deklarert i superklassen

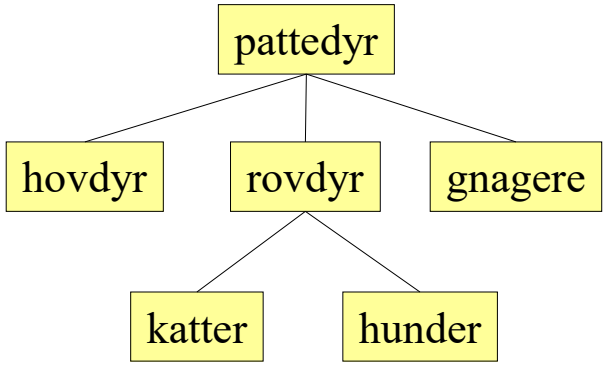
Arv (eng: inheritance)

- *Subklassen* kan
  - Definere egne felt og metoder
  - Redefinere eksisterende metoder (override)
- Subklasser kan brukes i koden der superklassen kan brukes, men ikke motsatt:
- Hvis BarneBok er *subklasse* til Bok, så gir den siste av disse kompileringsfeil:

  - Bok bok1 = new Bok("Subnaiv", "El Moe", 314); //OK
  - Bok bok2 = new BarneBok("ABC", "Tris Burti", 3); //OK
  - BarneBok bok3 = new Bok("Nay", "No Good", 0); //NOT

Hvorfor arv?

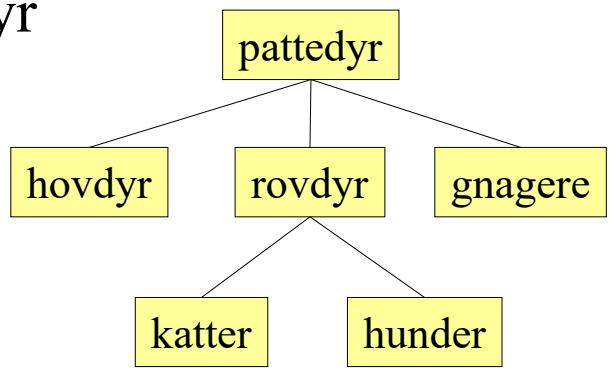
- På enkleste nivå, så gir arv mulighet for gjenbruk av kode, men viktigere:
- Klassifisering i klassehierarkier er en intuitiv organisering av fenomener i verden
  - hovdyr er et pattedyr
  - katt og hund er et rovdyr er et pattedyr
  - gnager er et pattedyr



Vi kan bruke arv til å gjenspeile dette i programstrukturen

Hvorfor arv?

- I et OO-system, kan f.eks. det som er felles for alle pattedyr, håndteres i en *superklasse* **Pattedyr**
- Subklassen Hovdyr, Gnagere kan både utvide I hver sin retning og modifisere Patterdyr
- Det som er felles for alle pattedyr, av fenomener i verden
  - hovdyr er et pattedyr
  - katt og hund er et rovdyr er et pattedyr
  - gnager er et pattedyr



Arv, er det bra?

- Noen mener at arv må støttes for at et programmerings-språk skal være “ekte” objekt-orientert
- Noen mener at arv bør I stor grad bør unngås, og at en heller bruker andre mekanismer for å oppnå det samme

Ukritisk bruk av arv kan gi utfordringer

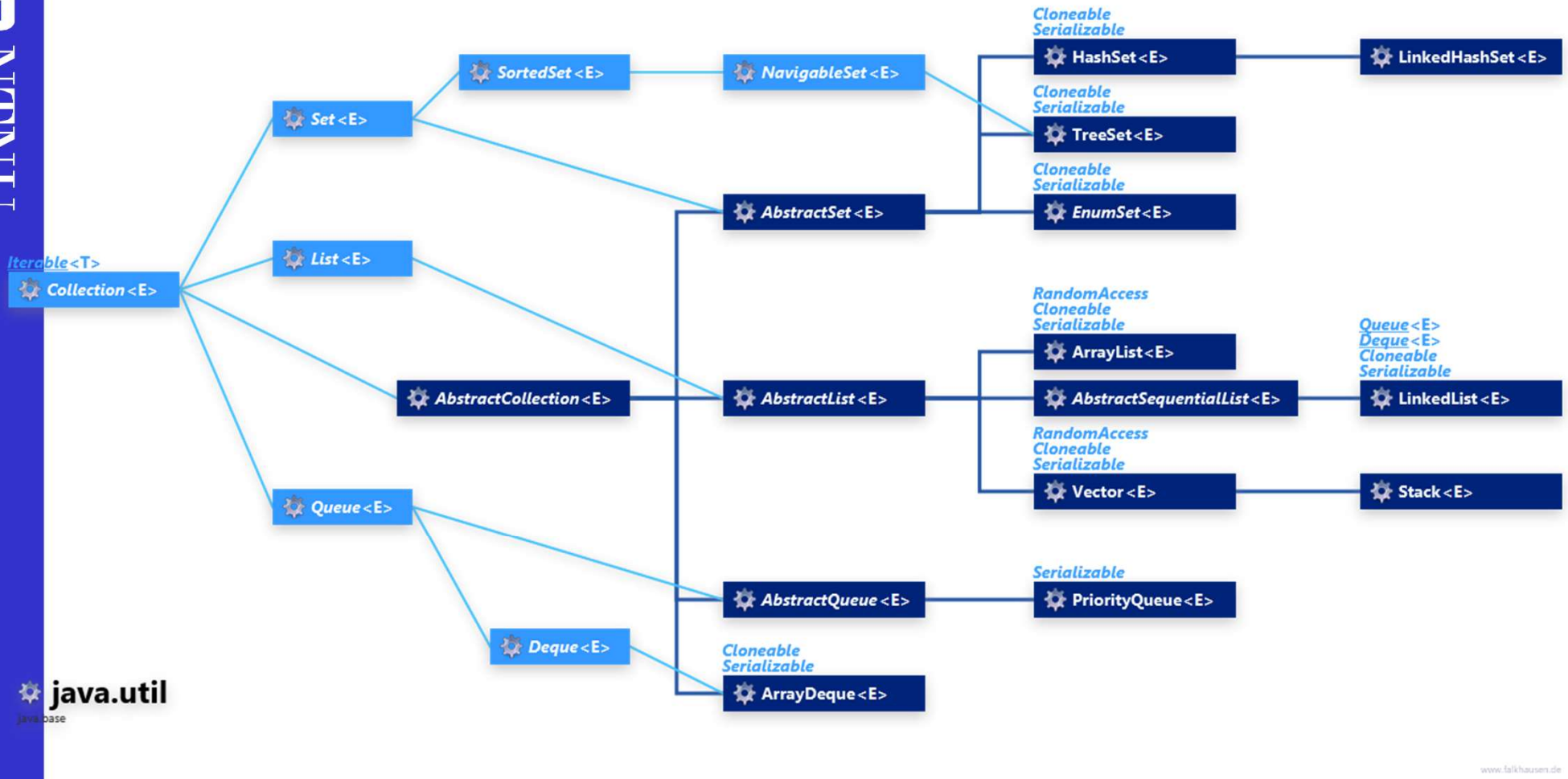
  - F.eks: Overdreven bruk kan gi veldig store klasse-hiarkier, og hvor en trenger å gjøre endringer I mange forskjellige klasser når en skal legge til ny funksjonalitet.

Uansett, arv brukes i mye kode dere må forholde dere til

F.eks.

- Alle klasser i Java arver fra klassen **Object**
- Når vi skal lage apper I JavaFX, så må den arve fra en gitt klasse "Application".

Collectionsrammeverket er forsøkt tegnet på så mange måter at jeg ikke gidder selv.



https://falkhausen.de/Java-10/java.util/Collection-Hierarchy-simple.html

NTNU

Se Samlinger.java

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**

Arv av metoder

- En subklasse arver alle egenskaper til superklassen
  - felter og (vanlige) metoder
  - Konstruktører arves, men blir ikke tilgjengelig utenifra.
    I stedet kan de *brukes* av subklassens konstruktør.
- 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

NTNU

Se Bok.java og BarneBok.java

Polymorfisme

- Vi ser i eksemplet om Bok og BarneBok, at det ikke er hva variablen er deklarert som, som bestemmer hvilken toString-metode som kalles.
- Det er den faktiske typen til objektet som bestemmer det.