Læringsmål for forelesningen

- Objektorientering
  - Abstrakte klasser og grensesnitt, redefinering av metoder.

- Java-programmering
  - Arv og bruk av abstrakte klasser
- Eclipse
  - Undersøke instanser i Eclipse

Dagens forelesning

- Motivasjon:
  gjenbruk av kode vha. arv
- Bok-eksempel:
  - samle felles kode i samleklassen Bok
  - stor grad av gjenbruk for subklasser (Roman…)
- Abstrakte klasser og grensesnitt
  - Abstrakte klasser er ufullstendige klasser, som spesifiserer hva som trengs for å komplettere dem
  - Grensesnitt er tomme klasser, som spesifiserer…

Arv og gjenbruk av kode

- Gjennom arv kan en subklasse få egenskaper som er definert i en annen klasse
  - `Ordbok extends Bok` gjør at Ordbok får alle egenskapene som er definert i Bok, både attributter og metoder
- Arving gjør at en sparer kode, siden flere klasser kan nyttiggjøre seg den samme koden
  - både Bok (superklassen), Ordbok og Tegneseriealbum (subklassene) vil dra nytte av koden skrevet for Bok-klassen (i Bok.java).

- Dersom flere klasser har likhetstrekk, f.eks. like attributter og/eller metoder, kan det være praktisk å samle det som er felles, i en superklasse og definere resten i subklassene
- En slik ”samleklasse” er ofte spesiell, fordi det ikke er noe poeng å lage instanser av den, men kun av subklassene

Bok som ”samleklasse”

- Vi lager oss et klassehierarki hvor Bok er en samlebetegnelse/-klasse
  - Vi ønsker ikke instanser av typen Bok, men kun instanser av subtypene
    - Ordbok
    - Roman
    - Biografi
    - Tegneseriealbum
    - etc.
- En kan knytte mange egenskaper til samleklassen
  - Alle bøker har en tittel og et antall sider

Bok som ”samleklasse”

- Vi definerer Bok-klassen
  - to felter: `String tittel`, `int antallSider` og
  - en konstruktør som setter de to feltene
  - toString-metode som bruker disse feltene

- Vi definerer to subklasser, Roman og TegneserieAlbum
  - begge har konstruktører som kaller Bok-konstruktøren

Bok-klassen

- Attributter og konstruktør

```
private String tittel;
private int antallSider;

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

- toString()-metode
  - setter sammen en tekst, basert på attributtene

```
public String toString(){
    return tittel + ", " + antallSider + "s.";
}
```

Roman- og TegneserieAlbum-klassene

- Konstruktør
  - kaller Bok sin konstruktør med spesifikke verdier

```
public class Roman extends Bok {

public Roman(String tittel, int antallSider){
    super(tittel, antallSider);
}
```

```
public class TegneserieAlbum extends Bok {

public TegneserieAlbum(String tittel, int antallSider){
    super(tittel, antallSider);
}
```

```
Roman roman = new Roman("Bakgård", 100);
TegneserieAlbum tegn = new TegneserieAlbum("Enhjørninger og avsagde hagler", 50);
Bok bok = new Bok ("Bakgård", 100);
```

Abstrakte klasser

- Hvis vi alle bøker har en bestemt sjanger (roman, ordbok, tegneseriealbum) så er det ingen poeng i å kunne instansiere Bok-klassen, snarere tvert imot
- Bok-klassen kalles ”abstrakt”, fordi den ikke gir mening å instansiere med **new**-operatoren
- Dersom Bok markeres med `abstract`, vil ikke Java la oss bruke `new Bok(...)`
  - `public abstract class Bok { ... }`

Det er to grunner til å markere en klasse som `abstract` og dermed hindre instansiering:

1. Det gir ikke mening å ha instanser som ikke (samtidig) er instans av en subklasse (som pattedyr)

2. Klassen er *ufullstendig*, ved at en eller flere metoder ikke er implementert.

Vi skal nå se nærmere på mulighet nr. 2

Endring av Bok-klassen

- Vi ønsker å lage en bedre toString()-metode, hvor en subklasse kan ”skyte inn” egen sjangertekst
- Definerer en ny metode **getSjanger()** som det er meningen at subklassene skal redefinere

```
protected String getSjanger() {
    return "?";
}
```

- **getSjanger()**-metoden brukes av **toString()**:

```
public String toString(){
    return getSjanger() + ": " + tittel + ", " + antallSider + "s.";
}
```

- Roman redefinerer getSjanger, slik at utskriften blir bedre:

```
protected String getSjanger(){
    return "Roman";
}
```

Vi bør ikke kunne la være å redefinere getSjanger!

```
Roman: Bakgård, 100s.
Tegneserie: Enhjørninger og avsagde hagler, 50s.
```

Abstrakte metoder

- Når en definerer en abstrakt klasse, er det ofte nyttig å kunne *kreve* av subklasser at de (re)definerer spesifikke metoder
  - Bok-klassen bør f.eks. kreve at subklasser inkl. Roman-klassen redefinerer getSjanger-metoden

- En metode i en klasse kan markeres som `abstract` for å spesifisere at den *må* (re)defineres av en subklasse
  - `protected abstract String getSjanger();`
  - Merk at *metodekroppen må utelates* når den er markert som `abstract`.

- Roman-klassen *må* (re)definere getSjanger-metoden

```
protected String getSjanger(){
    return "Roman";
}
```

```
Roman: Bakgård, 100s.
Tegneserie: Enhjørninger og avsagde hagler, 50s.
```

Abstrakte klasser

- Klasser som har abstrakte metoder kalles *abstrakte* klasser
  - *Må* defineres som **abstract** class

```
public abstract class Bok {
```

- Dersom en ser på en abstrakt klasse som en *ufullstendig* klasse, er poenget å spesifisiere hva som skal til for at en subklasse skal være fullstendig, altså de *abstrakte metodene*.
- Alle Bok-subklasser, som ikke selv er deklarert som `abstract`, må (re)definere/implementere **getSjanger()**-metoden
- En *abstrakt* klasse er altså en klasse med *deklarerte*, men *uimplementerte* metoder

Abstrakt klasse og subklasse: puslespillbiter som passer sammen

- Den abstrakte klassen tilbyr mye kode å gjenbruke, men har noen hull (abstrakte metoder) som subklassen må fylle
- Subklassen må fylle alle superklassens hull for å selv være komplett
- Dersom noen hull ikke fylles, må subklassen selv være abstrakt

Samspill mellom metoder i super- og subklasser

- En superklasse definerer det som er felles for subklassene.
  - Tittel og antall sider
- En abstrakt superklasse stiller krav til metoder som må (re)defineres i subklassene.
  - **getSjanger()**-metoden
- Generelle metoder i den abstrakte superklassen kan kalle de abstrakte metodene og være trygg på at alle faktiske instanser har (re)definert dem.
  - **toString()**-metoden kaller **getSjanger()**-metoden
- Subklassene arver de generelle metodene fra superklassen og gir dem en spesifikk oppførsel ved å implementerer de påkrevde spesifikke metodene
  - **Roman**- og **TegneserieAlbum**-instansene får forskjellige toString-tekster, basert på samme generelle kode i **Bok**, ved å returnere forskjellige verdier fra **getSjanger()**.

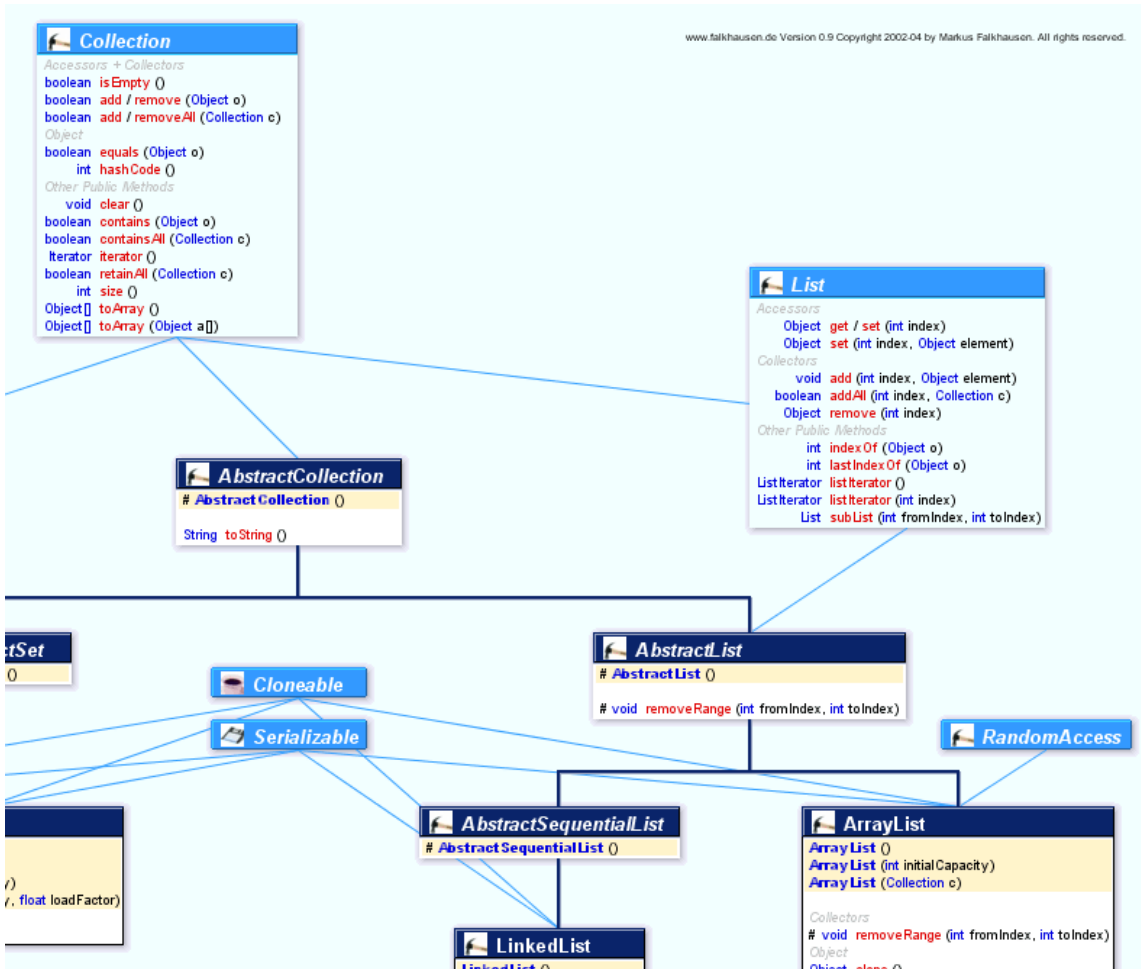
Abstrakte klasser vs. interface

- Abstrakte klasser og interface har fellestrekk
  - Begge spesifiserer metoder som en (annen) klasse må implementere
  - Selv om begge teknisk sett er klasser, kan de ikke instansieres
- Forskjellen er at en abstrakt klasse samler felles felter og metoder og selv implementerer (så mye som mulig av) felles logikk, som subklassene kan utnytte gjennom arv.[1]
- Abstrakte klasser brukes ofte sammen med interface, som nesten ferdige implementasjoner…

1) Java-interface har nå såkalte **default**-metoder, som i praksis arves av implementerende klasser

Abstrakte klasser vs. interface

- AbstractCollection implementerer (delvis) Collection
- AbstractList implementerer (delvis) List
- ArrayList og LinkedList fyller hullene



Abstrakt klasse vs. interface og puslespill-analogien

- Et interface er en puslespillbrikk med bare hull, dvs. ingen gjenbrukbar kode, kun krav til utfyllende metoder
- (bortsett fra default-metoder, som vi ikke går inn på i dette kurset)
- Subklassen må fylle alle interfacenes hull for å være komplett
- Dersom noen hull ikke fylles, må subklassen selv være abstrakt

implements vs. extends

- Arv spesifiseres vha. **`extends`**
  - Kun mulig å arve **én** klasse
    (gjelder også abstrakte klasser)
- Interface som implementeres spesifiseres vha. **`implements`**
  - Mulig å implementere flere interface
- Et interface kan arve (utvide) et annet interface vha. **`extends`**

```
public interface EtGrensesnitt extends EtAnnetGrensesnitt
```

Læringsmål for forelesningen

- Objektorientering
  - Abstrakte klasser og grensesnitt, redefinering av metoder
- Java-programmering
  - Arv og bruk av abstrakte klasser