Kursus 02199: Programmering
Nedarvning: afsnit 7.1-7.5

Anne Haxthausen, IMM, DTU

1. Klasse-hierarkier: superklasser og subklasser (afsnit 7.1, 7.3)
2. Object klassen: er automatisk superklasse til alle klasser (læs selv 1. del af afsnit 7.3)
3. Nedarvning: superklassens felter og metoder nedarves til subklasser (afsnit 7.1)
4. Konstruktorer nedarves ikke (afsnit 7.1)
5. Synlighedsmodifikatorer (afsnit 7.1)
6. Type konverteringer og check (afsnit 7.4)
7. Overskrivning af metoder: omdefinering af en nedarvet metode (afsnit 7.2)
8. Polymorfi: objektets klasse afgør hvilken metode, der kaldes (afsnit 7.4)
9. Enkel versus multipel nedarvning (afsnit 7.1)

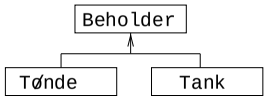
Begrebs-hierarkier

Eksempler på *begreber* er: tidspunkt, aftale, dyr, ko, person, beholder, tønde, . . .

Beslægtede begreber kan arrangeres i et *hierarki* (efter hvor generelle de er).

Eksempel 1: 'dyr' kan opdeles i 'pattedyr' (herunder 'ko'), 'fugl', 'fisk'

Eksempel 2: 'beholder' kan opdeles i 'tønde', 'tank', . . .



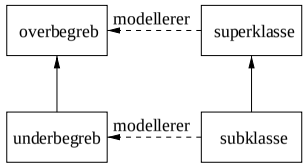
Begrebet 'beholder' er mere generelt end 'tank', da man kan sige 'en tank *er en* beholder'.

Idet hvert begreb har nogle egenskaber, kan man også forklare hierarkiet ved at sige, at et begreb B er et underbegreb af et andet begreb A, hvis underbegrebet B har (arvet) alle A's egenskaber, samt evt. har nogle flere.

Begrebshierarkier bruges ofte til at beskrive verdenen omkring os.

Modellering af begrebs-hierarkier som klasse-hierarkier

I Java og mange andre objektorienterede programmeringssprog modelleres begrebs-hierarkier vha klasse-hierarkier. (Klasser repræsenterer jo begreber.)



En superklasse modellerer et generelt begreb (f.eks. beholder), en subklasse et mere specielt begreb (f.eks. tank).

Nedarvning:

Et underbegreb har alle egenskaberne, som dets overbegreb har.

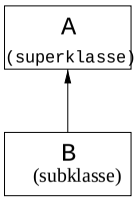
En subklasse har derfor alle felter og metoder, som dens superklasse har. Ofte gøres subklassen mere specifik ved at definere flere felter og metoder end superklassen.

Skabelse af klasse-hierarkier i Java

Java har en sprogkonstruktion til at lave klasse hierarkier:

En klasse B kan nemlig defineres som en *udvidelse* af en eksisterende klasse A, så A bliver en *superklasse* til B, og B en *subklasse* af A.

```
class B extends A {
  nye felter og metoder
  omdefinerede metoder
  konstruktorer
}
```



Eksempel:

```
class Tank extends Beholder {
  ...
}
```

Nedarvning

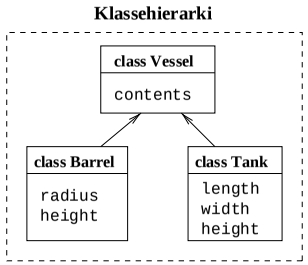
En subklasse *arver* metoder og felter, men ikke konstruktorer, fra sin superklasse.

De kan bruges i subklassen som om de var defineret i subklassen.

En subklasse kan derudover definere nye felter og metoder, eller omdefinere (overskrive) metoder som ellers ville være blevet nedarvet.

Subtilitet vedr. private felter og metoder forklares senere.

Eksempel: klasse-hierarki for beholdere



Klassen `Vessel` skal repræsentere, hvad der er fælles for alle slags beholdere.

Klassen `Barrel` skal repræsentere tøndeformede beholdere og `Tank` kasseformede beholdere.

`Barrel` og `Tank` skal arve fælles egenskaber fra `Vessel`.

`Barrel` og `Tank` skal hver for sig definere egenskaber der er specielle for netop tønder og tanke.

Implementering af beholder-hierarki i Java

```
class Vessel {
  double contents; //i liter (= kubikdecimeter, dm^3)
}

class Tank extends Vessel {
  double length, width, height; //i decimeter, 1 dm = 10 cm

  Tank(double length, double width, double height)
  { this.length = length; this.width = width; this.height = height; }
}

class Barrel extends Vessel {
  double radius, height; //i decimeter, 1 dm = 10 cm

  Barrel(double radius, double height)
  { this.radius = radius; this.height = height; }
}
```


Eksempel på brug af objekter fra beholder-hierarkiet

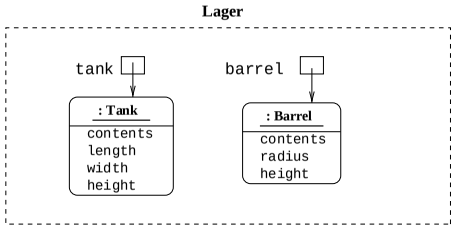
```
public class Vessel1 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    Barrel barrel = new Barrel(2.5, 8);

    tank.contents = 0; barrel.contents = 1.5;
    System.out.println("Indhold af tank = " + tank.contents);
    System.out.println("Bredde af tank = " + tank.width);

  }
}
```

Alle `Tank`- og `Barrel`-objekter har et `contents`-felt, arvet fra klassen `Vessel`.

Objekterne i `Vessel1.java`



Opgave: Ville det være lovligt at skrive `System.out.println(barrel.width)`?

`System.out.println(barrel.height)`?

`System.out.println(tank.height)`?

Konstruktorer nedarves ikke.

Men det første, en subklasses konstruktor gør, er at kalde en konstruktor for superklassen.

Dette kan gøres eksplicit med et kald af formen **super(**. . .).

Hvis ikke det gøres eksplicit, så underforstås et kald **super()** til en argumentløs konstruktor.
(Sådan en findes automatisk, hvis man ikke selv har defineret en.)

Eksempel: kald af superklassens konstruktorer

```
class Vessel {
  double contents;

  Vessel() { contents = 0.0; }
  Vessel(double contents) { this.contents = contents; }
}

class Tank extends Vessel {
  double length, width, height;

  Tank(double length, double width, double height)
  { this.length = length; this.width = width; this.height = height; }
}

class Barrel extends Vessel {
  double radius, height;

  Barrel(double contents, double radius, double height)
  { super(contents); this.radius = radius; this.height = height; }
}
```



```
public class Vessel2 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    Barrel barrel = new Barrel(1.5, 2.5, 8);
    System.out.println("Indhold␣af␣tank␣=␣" + tank.contents);
    System.out.println("Indhold␣af␣barrel␣=␣" + barrel.contents);
  }
}
```

Kørsel af dette program giver følgende udskrift på skærmen:

```
Indhold af tank = 0.0
Indhold af barrel = 1.5
```

Subtilitet vedr. private metoder og felter

Hvis et felt (eller en metode) er **private** i en superklasse A, så kan det ikke eksplicit refereres til i subklasser af A.

Feltet findes dog i subklassens objekter.

default synlighed

Synligheds-egenskaberne for et felt eller en metode, der ikke har nogen synlighedsmodifikator, er som **public** i klasser (og subklasser) i samme pakke, men som **private** for klasser (og subklasser) i andre pakker.

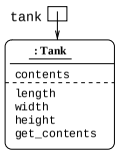
protected metoder og felter

Synligheds-egenskaberne for et felt eller en metode erklæret **protected** er som **public** i klasser (og subklasser) i samme pakke og i alle subklasser i andre pakker, men som **private** for ikke subklasser i andre pakker.

private metoder og felter: eksempel

```
class Vessel {
  private double contents; //i liter (= kubikdecimeter, dm^3)
  Vessel() { contents = 0.0; }
  double getcontents() { return contents; }
}
class Tank extends Vessel {  ...  } // contents ukendt navn i Tank

public class Vessel6 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    System.out.println("Indhold af tank = " + tank.getcontents());
    //System.out.println("Indhold af tank = " + tank.contents);  er ulovlig
} }
```



Type konverteringer

implicit widening konvertering fra subklasse til superklasse

En variabel af type T kan referere til objekter hørende til klassen T *og dens subklasser*.

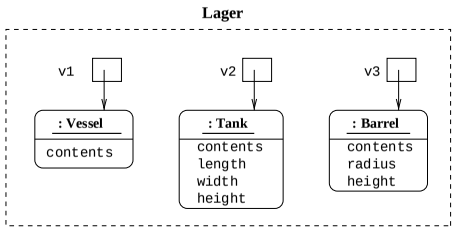
En variabel, der kan referere til objekter af forskellige klasser, kaldes en *polymorf* reference.

Eksempel 3:

```
Vessel v1 = new Vessel();
Vessel v2 = new Tank(15, 9, 12);
Vessel v3 = new Barrel(1.5, 2.5, 8);
```

En variabel (som `v1, v2, v3`) af type `Vessel` kan referere til et objekt af klasse `Vessel`, `Tank` eller `Barrel`. Derfor er f.eks. tildelingen `v2 = new Tank(15, 9, 12);`, der får `v2` til at referere til et objekt af klasse `Tank`, lovlig.

Lager ved Vessel3.java



Type konverteringer

Explicit narrowing konvertering med cast fra superklasse til subklasse

Eksempel:

```
Vessel v2 = new Tank(15, 9, 12);
Vessel v3 = new Barrel(1.5, 2.5, 8);
Tank tank1  = v2;           //ulovlig (giver compileringsfejl)
Tank tank2  = (Tank) v2;    //der skal vaere et explicit cast
Tank tank3  = (Tank) v3;    //ulovlig (giver koerselsfejl)
```

Typekonverteringen `(Tank)v3` går galt, når programmet køres, da `v3` ej henviser til et `Tank`-objekt.

En variabel har en type, et objekt har en klasse

Vigtig skelnen:

- En variabel har en erklæret *type*, f.eks. har `v2` i eksempel 3 typen `Vessel`.
- Et objekt har (dvs. hører til) en bestemt *klasse*. Objektets klasse bestemmes af hvilken konstruktor der blev brugt til at skabe objektet.

  F.eks. vil et objekt skabt med **new** `Tank(15, 9, 12)` have klassen `Tank`.

Check af tilgang til objekters felter og metoder

Regel:

Felt-tilgang `o.f` og metodekald `o.m(...)` tjekkes ud fra variablens `o`'s erklærede type `T`:

`o.f` er lovlig, hvis `T` har et felt `f` med passende synlighedsegenskaber (virker som **public** og ikke **private** på det givne sted.) Tilsvarende for `o.m(...)`.

Eksempler

Variablen `v2` har type `Vessel`, og `Vessel` har et felt kaldet `contents`.

Så udtrykket `v2.contents` accepteres af Java-oversætteren.

Men `Vessel` har ikke noget felt kaldet `width`, så udtrykket `v2.width` forkastes af Java-oversætteren, selvom `v2` faktisk refererer til et `Tank`-objekt, som har et `width`-felt.

Overskrivning af metoder

Overloading

Definerer en klasse flere metoder med samme navn, men forskellige parameter typer, kaldes det *overloading*, jf. forelæsning 6.

Overskrivning (omdefinering, 'overriding')

Definerer en subklasse en metode med samme navn `m` og parameter typer som superklassen, kaldes det *overskrivning* (eng. overriding). Subklassen arver da ikke superklassens metode `m`, men har sin egen version af `m`. Superklassens version kan dog nås via **super.**`m(...)`.

Denne fleksibilitet er god, fordi klasser, der er i "familie", kan bruge samme navne-konventioner for metoder, der gør de "samme ting".

Erklærer man en metode med **final** kan den *ikke* overskrives.

Eksempel på overskrivning af `volume` metode i beholder-hierarkiet

```
class Vessel {
  double contents;
  ...
  double volume() { return 0; }
}

class Tank extends Vessel {
  double length, width, height;
  ...
  double volume() { return length * width * height; }
}

class Barrel extends Vessel {
  double radius, height;
  ...
  double volume() { return height * Math.PI * radius * radius; }
}
```

Subklasserne *overskriver* ('overrider', omdefinerer) metoden `volume` fra superklassen.

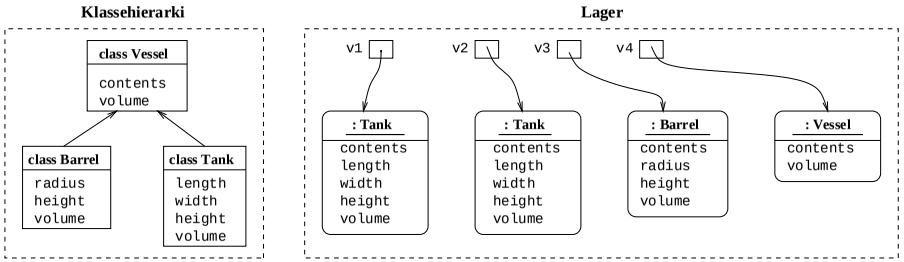

```
public class Vessel4 {
  public static void main(String[] args) {
    Vessel v1 = new Tank(15, 9, 12);
    Vessel v2 = new Tank(0.7, 0.7, 2.05);
    Vessel v3 = new Barrel(1,5, 2.5, 8);
    Vessel v4 = new Vessel();

    System.out.println("Rumfang af v1 = " + v1.volume());
    System.out.println("Rumfang af v2 = " + v2.volume());
    System.out.println("Rumfang af v3 = " + v3.volume());
    System.out.println("Rumfank af v4 = " + v4.volume());
  }
}
```

Uddata ved kørsel af Vessel4:

```
Rumfang af v1 = 1620.0
Rumfang af v2 = 1.0044999999999997
Rumfang af v3 = 157.07963267948966
Rumfank af v4 = 0.0
```


Klassehierarkiet og lager ved `Vessel4.java`



Opgave: Hvilken af de tre `volume`-metoder kaldes af `v1.volume()`, `v2.volume()`, `v3.volume()`, `v4.volume()`?

Overskrivning og polymorfi

`v1` er en polymorf reference, der kan referere til både `Vessel`, `Tank` og `Barrel` objekter. Da hver af disse har en `volume` metode, er der potentielt 3 muligheder for, hvilken metode, der kaldes med `v1.volume()`.

Hvilken, det er, afgøres af klassen af det objekt, som `v1` refererer til.

Den erklærede type for `v1` — som her er `Vessel` — bestemmer altså ikke hvilken metode, der kaldes. Men der *skal* være en metode med det angivne navn i `Vessel`, ellers forkaster oversætteren programmet (cf. reglen side 19).

Regel:

Hvilken udgave af en overskreven metode `m`, der kaldes med `o.m(...)`, afhænger af klassen af det objekt, som `o` refererer til, ikke af `o`'s type.

Eksempel på brug af super til at kalde en overskreven metode

```
class Vessel {
  ...
  public String toString() { return "indhold: " + contents +  " l"; }
}

class Tank extends Vessel {
  ...
  public String toString() {
    return "Tank med volumen: " + volume() + " l og " + super.toString();
  }
}

public class Vessel5 {
  public static void main(String[] args) {
    Vessel v1 = new Tank(15, 9, 12);
    System.out.println("v1: " + v1.toString());
}
```

Uddata: `v1: Tank med volumen: 1620.0 l og indhold: 0.0 l`

Reerklæring af felter

Hvis subklassen erklærer et felt med samme navn `f` som et felt i superklassen, så får man to felter.

Subklassens felt blot hedder `f`.
Feltet fra superklassen kan tilgås i subklassen som **super.** `f` (dog ikke, hvis det er **private**).

Det er næsten altid forkert og resulterer i forvirring og hovedpine. **Overskriv kun metoder, ikke felter!**

Opsummering vedr. super referencen

super er en reference i lighed med **this**.

1. Konstruktorer fra superklassen kan kaldes fra subklassen med **super**(...).
2. Metoder m fra superklassen kan kaldes fra subklassen med **super.** m`(...)`.
3. Felter f fra superklassen kan nås med **super.** f.

2 og 3 gælder dog ikke, hvis m og f er private.

Enkel og multipel nedarving

I nogle objektorienterede sprog kan en klasse have flere superklasser.

Multipel nedarvning er nyttig når man arbejder med to forskellige begrebshierarkier på en gang.

F.eks. beholdere (Vessel, Tank, Barrel) og farver (Plain, Colored).

- En farvet tønde er både farvet (Colored) og en beholder (Vessel).
  Så et objekt skal kunne være Colored og Vessel på en gang.
- En almindelig tønde er en Vessel, men ikke Colored.
  Så Vessel kan ikke være en subklasse af Colored.
- Et farvet papirark er Colored, men ikke en Vessel.
  Så Colored kan ikke være en subklasse af Vessel.

Enkel og multipel nedarving

Java har kun enkel nedarvning ('single inheritance'): en klasse kun have én umiddelbar superklasse.

Det er fordi multipel nedarvning fører til teoretiske og praktiske problemer.

(Eksempel: I hvilken rækkefølge skal superklassernes konstruktorer kaldes?)

(Eksempel: Hvis to metoder med samme signatur arves fra to forskellige superklasser, hvilken en skal så bruges?)

Nedarvning i Java: sammenfatning

- Klasser kan ordnes i hierarkier, der reflekterer begrebshierakier.
- En subklasse arver felter og metoder fra superklassen, dvs. de kan bruges, som var de defineret i subklassen. Undtagelser:
  - Konstruktorer nedarves ikke, men de kan bruges i subklassen som **super**(...).
  - Private felter og metoder nedarves ikke, men de eksisterer og kan tilgås indirekte.
- Subklassen kan definere nye felter og metoder.
- Subklassen kan redefinere (overskrive, 'override') eksisterende metoder `m` (der ikke er **final**). I subklassen kan superklassens `m` da nås under navnet **super**.`m` .
- Hvilken udgave af `m`, der kaldes med `o.m(...)`, afhænger af objektets klasse, ikke variablens (`o's`) type `T`.
- Subklassen kan reerklære en variabel, men det anbefales ikke.
- En variabel `o` af klassetype `T` kan referere til objekter af klasse `T` og alle dens subklasser.
- Felttilgang `o.f` og metodekald `o.m(...)` tjekkes ud fra variablens `o's` erklærede type `T`, ikke objektets klasse.
- Man kan eksplicit typekonvertere ('caste') et udtryk af klassetype `T` til en subklasse af `T`.

Fordele ved at bruge nedarvning

- klassehierarkiet afspejler explicit problemdomænets begrebshierarki
- genbrug af kode (hurtigere at skrive, nemmere at vedligeholde)
- kan bruge f.eks. en tank, hvor der forventes en beholder