

Nedarvning

Sidst ændret: 07/19/2018 14:11:40

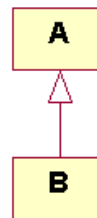
[Opgaver](#)

I forrige kapitel så vi hvordan man kan lave noget nyt ved at sætte objekter sammen - ved komposition. I dette kapitel skal vi studere hvordan man kan lave en ny klasse ud fra en allerede eksisterende klasse, og på den måde lave noget nyt.

Nedarvning er en relation

At lave en ny klasse ud fra en allerede eksisterende klasse kan gøres ved **nedarvning**. Man arver det den oprindelige klasse har, og får mulighed for at **erstatte** egenskaber eller **supplere** med nye. Man kan sammeligne det med, at man tager en fotokopi af en klasse - føjer nye ting til kopien og streger andre ud. At en klasse arver fra en anden, er en **relation**, og for at kunne beskrive denne relation anvender man ofte et diagram - et **klassediagram**. Hvis klassen **B** nedarver fra klassen **A**, kan man illustrere det med følgende diagram:

Figur 1:
B nedarver fra A



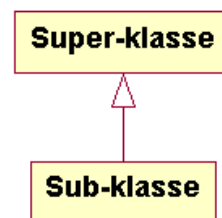
Pilen går fra klassen der arver, til klassen der nedarves fra. Som vi skal se i det følgende afspejler dette at kendskabet går fra **B** til **A** - det er **B** der kender **A** - ikke omvendt. Mao. **B** ved udemærket godt at den arver fra **A**, men **A** ved det ikke!

Kun arve fra én klasse

I Java er det kun muligt at nedarve fra én klasse. Det betyder at såkaldt **multipl nedarvning** ikke findes. Man kan til gengæld nedarve fra én klasse *til* flere subklasser.

Man anvender ofte to (relative) betegnelser, der beskriver forholdet mellem **A** og **B**. Man siger at **A** er **B's super-klasse** og **B** er **A's sub-klasse**.

Figur 2:
Super- og sub-klasse



Pga. disse betegnelser, kalder man ofte et klassediagram, som beskriver nedarvning, et **klasse-hierarki**.

Lad os skitsere hvordan de to klasser **A** og **B** kan implementeres i Java:

Source 1:
B nedarver fra A

```
public class A {
    ...
}

public class B extends A {
    ...
}
```

extends

Man bemærker, at det i erklæringen af **B** specificeres, at den nedarver fra **A** - eller *udvider* (eng.: extend) **A** som det benævnes i Java (vi bruger dog *altid* betegnelsen arver eller nedarver).

Lad os antage at **A** har en integer variabel og en dertil hørende get-metode:

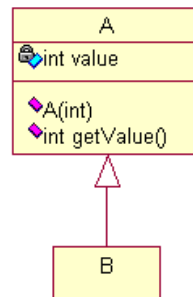
Source 2:
Klasse **A** med
variabel og
metode

```
public class A {  
    private int value;  
  
    public A( int v ) {  
        value = v;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    ...  
}
```

```
public class B extends A {  
    ...  
}
```

I et klassediagram beskriver man tilstedeværelsen af instans-variablen og metoden på følgende måde:

Figur 3:
Klassediagram
med instans-
variabel og
metode







Tre-delt kasse

For hver klasse inddeler man "kassen" i tre dele. I øverste del placerer man klassens navn. I midten, erklæringen af instans-variable. Nederst placerer man signaturerne for klassens konstruktører og metoder. I **B** er de to nederste felter fjernet, da man normalt ikke viser felter hvis de er tomme.

Foran hver instans-variabel/metode placeres ofte et lille icon der beskriver tilgængeligheden. Betydningen fremgår af følgende oversigt:

Tabel 1:
Symboler for
public og
private

	private instansvariabel
	public instansvariabel
	private instansmetode
	public instansmetode

Symbolet for **private** er en lille hængelås. Hvis det ikke er praktisk at bruge symbolerne, kan man anvende tegnet '+' i stedet for **public** og '-' for **private**. Symbolerne '+' og '-' er standard i UML, men jeg synes iconerne ser pænere ud.

Lad os vende tilbage til det konkrete eksempel. Hvad får **B** ud af at arve fra **A**?

Betinget arv

B får det hele, men på forskellige betingelser. Betingelserne kan i første omgang være komplicerede at forstå, men de muligheder vi har ved nedarvning afhænger af dem, og de er derfor væsentlige at kende.

1. Nedarvning af metoder og instansvariable

public instans-variable (som vi naturligvis *aldrig* laver!) nedarves også som **public** i subklassen. Det betyder at instans-variable, som ikke er indkapslede i instanser af **A**, heller ikke vil være indkapslede i instanser **B**.

private instans-variable nedarves, men de er ikke direkte tilgængelige. Det betyder at vi ikke kan lave følgende set-metode i **B**:

Source 3:
Kan ikke tilgå
value

```
public class B extends A {  
  
    public void setValue( int v ) {  
        value = v;  
    }  
  
    ...  
}
```

Vi kan simpelthen ikke tilgå **value**.

Der er naturligvis ikke meget ved at arve noget man ikke kan tilgå. I tilfældet med **private** instansvariable er der dog mulighed for at tilgå dem *indirekte*.

Enhver metode i **A** kan tilgå **value**. Det betyder at de metoder vi arver fra **A** stadig kan tilgå **value**, også selvom de nu er nedarvet til **B**. Dette fænomen skaber en indkapsling af instansvariablene i **A**, i forhold til metoderne i **B**. Kun via metoderne i **A** kan vi arbejde med den **private** del af datakernen vi har arvet.

Ingen af metoderne i **A** giver os mulighed for at ændre **value**. I **B** kan vi derfor ikke ændre **value** - hverken direkte eller indirekte.

De **public** metoder vi arver fra superklassen er også **public** i subklassen. Man kan derfor sende de samme requests til **B** som man kan til **A**. F.eks. kan vi bruge get-metoden fra **A** på instanser af **B**, da **B** har arvet den.

Source 4:
Kald af
nedarvet get-
metode





```
B ref = new B( 5 );  
System.out.println( ref.getValue() );
```

5

For **private** metoder gælder der det samme som for **private** instans-variable - de er ikke direkte tilgængelige i **B**. I særdeleshed kan de ikke kaldes udefra, men de kan heller ikke kaldes i **B**. På samme måde som for instans-variable har vi kun mulighed for at anvende dem indirekte, ved at vi kalder andre metoder i **A**, der er **public**, som kalder dem.

Man kan sammenfatte reglerne for nedarvning i følgende oversigt:

Tabel 2:
Betydning af
tilgængeligheds-
angivelser ved
nedarvning

	private instansvariabel	Ikke tilgængelig i subklassen.
	public instansvariabel	Som andre public instans-variable i subklassen (vi laver dog <i>aldrig</i> public instans-variable).
	private instansmetode	Ikke tilgængelige i subklassen.
	public instansmetode	Som andre public metoder i subklassen, og de kan give <i>indirekte</i> adgang til private dele af superklassen.

2. Nedarvning af konstruktører

Jeg har i virkeligheden snydt mht. udskriften ovenfor (Source 4), for programmet ville ikke kunne oversættes. Hvis man prøver, får man hele to fejl:

```

...java:..: No constructor matching A() found in class A.
public class B extends A {
    ^
...java:..: Wrong number of arguments in constructor.
    B ref = new B( 5 );
            ^
2 errors

```

Vi vil studere disse to fejl, fordi de lærer os om nedarvning af konstruktører. Vi vil dog se på dem i omvendt rækkefølge.

2.1 Den anden fejl

```

...java:..: Wrong number of arguments in constructor.
    B ref = new B( 5 );
            ^

```

Hvorfor siger den, at antallet af parametre ikke passer til nogen konstruktor? Har vi da ikke arvet en konstruktor fra **A** med netop én integer parameter? Nej, det har vi ikke - ihvertfald ikke unden videre!

Der gælder det specielle for konstruktører, at de ikke (rigtig) nedarves. Hvis vi vil have en konstruktor i **B**, der tager én integer som parameter må vi lave en ny:

```

public class B extends A {

    public B( int v ) {
        value = v;
    }

    ...
}

```

Source 5:

Kan ikke tilgå
value

Men dette viste assignment er (desværre) ikke muligt, da vi ikke har adgang til **value** fra subklassen **B**. Dette skyldes at **value** er **private**.

Hvordan skal det så kunne lade sig gøre, hvis vi ikke kan tilgå instans-variablen? Så må vi tilgå den *indirekte*. At vi kan gøre det indirekte skyldes at konstruktører nedarves, men med en klausul: *De kan kun kaldes fra konstruktører i subklassen*.

Det giver os mulighed for at få den tilsvarende konstruktor i **A** til at gøre arbejdet for os, da den har den direkte adgang til instans-variablen.

```

public class B extends A {

    public B( int v ) {
        super( v );
    }

    ...
}

```

Source 6:

Super-kald af
konstruktor

Når man skal kalde en konstruktor, der er nedarvet fra superklassen, gør man det med et **super**-kald. Kaldet udføres ved at man i super-klassen søger en konstruktor med passende formelle parametre, hvilket svarer til ethvert andet konstruktor-kald. Vi har nu to muligheder for at kalde konstruktører: **this**, der kalder en konstruktor i samme klasse, og **super** der kalder en konstruktor i super-klassen. Der er dog en ekstra betingelse vedrørende super-kaldet: *Det skal være den første sætning i konstruktøren*.

Hvis vi indfører denne konstruktor i **B** forsvinder også den første fejl, men lad os alligevel se nærmere på dens betydning.

2.2 Den første fejl

```
...java:...: No constructor matching A() found in class A.  
class B extends A {  
    ^
```

Hvorfor vil den have der skal være en default-konstruktor i **A**? Der er da ikke nogen der bruger den!

I den situation hvor vi ikke har nogen konstruktorer i **B** (hvilket var tilfældet inden vi lavede den set-konstruktor, vi netop har set ovenfor) vil Java, som bekendt, automatisk lave en default-konstruktor. Det er denne default-konstruktor i **B**, som er kilden til problemet. Jeg har tidligere nævnt at default-konstruktoren vil være tom - dvs. ikke have noget indhold. Det var ikke helt rigtigt - den har én linie:

Source 7:
*Automatisk
default-
konstruktor*

```
public B() {  
    super();  
}
```

Den kalder default-konstruktoren i super-klassen!

Hermed er vi fremme ved kilden til problemet. Compileren vil have en default-konstruktor i **A**, som default-konstruktoren i **B** kan kalde.

At jeg havde undladt at vise **super**-kaldet i default-konstruktoren var kun delvist en udeladelse. Det skyldes at *enhver* konstruktor som første linie har et sådant kald af default-konstruktoren, uanset om man selv anfører det eller ej!

Den eneste mulighed man har for at slippe af med dette **super**-kald til default-konstruktoren, er ved selv at anføre et super-kald til en anden af super-klassens konstruktorer. Det betyder at én af super-klassens konstruktorer *altid* vil blive kaldt, uanset hvad man gør.

At fejlen forsvinder, hvis vi laver en konstruktor i **B**, der tager en integer som parameter, skyldes at Java ikke længere laver default-konstruktoren i **B**, og der dermed ikke længere er nogen, der ønsker at kalde en default-konstruktor i **A**.

3. Overriding

**Overloading
supplerer**

Overriding er et begreb der komplementerer overloading. Overloading er som bekendt, at man supplerer et navns betydning med en anden betydning. Dette kunne f.eks. være en metode som man overloader ved at lave en ny metode, med samme navn, men med andre parametre.

**Overriding
erstatte**

Ved overriding laver vi også en metode med samme navn, men *med samme parametre*. På den måde *erstatte* den nye metode den gamle. Overriding kan kun forekomme ved nedarvning hvor vi erstatte en nedarvet metode med en ny. F.eks.:

Source 8:
*Overriding af
toString*

```
public class A {  
    private int value;  
  
    public A( int v ) {  
        value = v;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    public String toString() {  
        return "[" + value + "];"  
    }  
}
```

```
public class B extends A {  
  
    public B( int v ) {  
        super( v );  
    }  
}
```

```

    public String toString() {
        return "(" + getValue() + ")";
    }
}

```

```

public class Main {

    public static void main( String[] argv ) {
        B ref = new B( 5 );
        System.out.println( ref );
    }
}

```

(5)

Her overrider vi **toString** for at instanser af **B**, i modsætning til **A**, returner værdien i almindelige paranteser i stedet for kantede. Forskellen i paranteser tjener naturligvis ikke andet formål end at fastslå hvilken af de to metoder der reelt bliver kaldt.

Det er et simpelt eksempel på variation vha. nedarvning. **A** gør én ting, men **B** vil gøre noget andet.

Er den metode, der er blevet overrider endegyldigt tabt? Nej!

Man har stadig mulighed for at anvende den, men kun i subklassen. F.eks.:

```

class B extends A {

    public B( int v ) {
        super( v );
    }

    public String toString() {
        return "(" + getValue() + ")";
    }

    public String f() {
        return super.toString();
    }
}

```

```

public class Main {

    public static void main( String[] argv ) {
        B ref = new B( 5 );
        System.out.println( ref.f() );
    }
}

```

[5]

Source 9:
Kald til metode
i super-klassen

Her går metoden **f** udenom den nye **toString** og kalder i stedet den nedarverede, der ellers er overrider.

Anvendelsen af **super** i denne forbindelse, går godt i tråd med den tilsvarende anvendelse af **this** til eksplicit kald af metoder i klassen. **this** og **super** indgår begge i samme syntaktiske konstruktioner og har analog semantik ved kald af henholdsvis konstruktor og metode.

Man skal dog være opmærksom på at **super**, i modsætning til **this**, ikke er nogen rigtig reference. Man kunne f.eks. ikke have lavet metoden **f**, på følgende måde:

```

public String f() {
    return "" + super;
}

```

Source 10:
Kald til metode
i super-klassen

Her vil **super** ikke blive betragtet som en reference i forbindelse med klistre-plus. Klistre-plus vil ikke automatisk kalde **toString** på **super**, fordi **super** ikke er en rigtig reference. I stedet får man en fejlmeddelelse, hvis man prøver at oversætte det.

4. Packages

Samling af klasser

En **package** (dk.: **pakke**) er en samling af klasser. Grunden til at man vælger at samle nogle klasser i en package er deres logiske sammenhæng. Det kunne f.eks. være klasser der er stærkt koblete og bruges til at lave instanser, der arbejder sammen i et objektsystem, der gør det logisk at betragte dem som en gruppe af klasser med et særligt fællesskab.

Package scope

En anden grund til at samle klasser i en package er **package scope**. Med **private** og **public** har vi mulighed for at styre tilgængeligheden af instansvariable og metoder i en klasse, men der findes to andre tilgængeligheds-angivelser.

Manglende angivelse



Den første er helt at **undlade en angivelse**. I så fald er der tale om *rent* package scope. Det betyder at instansvariable og metoder er som **public** indenfor package, men som **private** udenfor package. Man kan også formulere det på den måde: At klasser i en package er friends, der giver hinanden adgang, mens klasser udenfor package ikke får adgang.

protected

Den anden angivelse er **protected** og er en mindre *udvidelse* af package scope. Hvis man subklasser en klasse fra en package, uden selv at være med i den pågældende package, er de dele af superklassen der er **protected**, tilgængelige i subklassen. Man kan også formulere det på den måde: At subklasser, udenfor package, til klasser i package, er friends med deres superklasser (men ikke med de andre klasser i package).

Ligesom vi har symboler for **private** og **public** har vi også to symboler for **protected** instansvariable og **protected** metoder (vi har derimod ikke symboler for *rent* package scope):

Tabel 3:
*Symboler for
protected*

	protected instansvariabel
	protected instansmetode

Symbolet for **protected** er en nøgle, der signalerer en indkapsling der ligger et sted "mellem" **private** og **public**. Man kan alternativt bruge tegnet: '#', hvis det ikke er praktisk at bruge symbolet. '#' er standard i UML, men jeg synes som nævnt at iconerne ser bedre ud.

Lad os se et eksempel med to klasser **A** og **B** i en package. Et eksempel der er uhyre enkelt, men illustrerer teknikken i packages. Lad os starte med klassen **A**:

Source 11:
*Klasse der
tilhører package
VorPakke*

```
package VorPakke;  
  
public class A {  
    protected int value;  
  
    public A( int v ) {  
        value = v;  
    }  
}
```

Man bemærker den første linie i filen:

```
package VorPakke;
```

Skal stå i alle filer i package

Denne linie skal stå *i alle filer der skal høre til vores package*, som vi i dette tilfælde har valgt at kalde: **VorPakke**. Linien skal samtidig være *den første linie* i filen (bortset fra tomme linier og kommentarer). Det skal samtidig bemærkes at filen kun må indeholde klasser der skal høre til **VorPakke**.

protected mere læsbar

Til slut bemærker man, at vi lader **value** være **protected**, i modsætning til den sædvanlige praksis med indkapsling med **private**. Vi vil normalt anvende **protected** frem for at undlade en angivelse (hvilket ville være *rent* package scope), da **protected** er lettere at læse. Hvis der ikke står noget, kan man let overse at der er tale om package scope. Man kan samtidig blive i tvivl om en manglende angivelse betyder at programmøren ønsker at det skal være package scope, eller han rent faktisk har glemt en angivelse - f.eks. at der skulle have stået **private**!. Det er sjældent, at det betyder noget hvilken af de to former for package scope vi vælger. Derfor tillader vi os denne præference for **protected**.

Lad os lave en subklasse til **A**:

Source 12:
protected giver
subklassen
adgang

```
package VorPakke;

public class B extends A {

    public B( int v ) {
        super( v );
    }

    public void setValue( int v ) {
        value = v;
    }

    public int getValue() {
        return value;
    }
}
```

B skal også være med i **VorPakke**, og vi har derfor angivet dette i første linie.

Man bemærker, at vi to steder i denne klasse tilgår instans-variablen **value**. Dette er muligt, fordi den ikke er **private**, men **protected**.

Endelig har vi en testanvendelse:

Source 13:
import af
package

```
import VorPakke.*;

public class Main {

    public static void main( String[] argv ) {
        B ref = new B( 7 );

        System.out.println( ref.getValue() );
        ref.setValue( 5 );
        System.out.println( ref.getValue() );
    }
}
```

```
7
5
```

Man bemærker at vi, som for andre packages, skal have en import-angivelse i starten af filen, der importerer klasserne fra **VorPakke**. Hvis man både har en package-angivelse, der placerer filens klasser i en bestemt pakke, og samtidig har en angivelse af diverse ting man importerer, skal package-angivelsen *altid* stå først, og import-angivelserne skal dernæst anføres på sædvanlig vis.

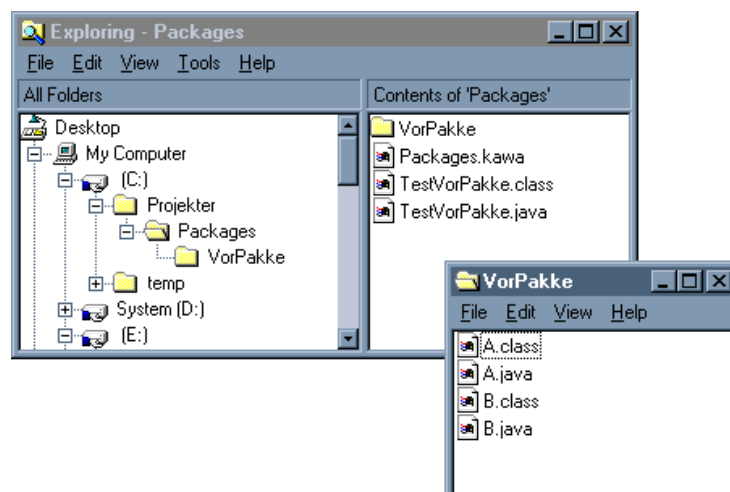
Skal alle i samme directory

I alt har vi nu tre filer, der udgør vores eksempel. Grunden til at vi her omtaler filerne er en ekstra detalje ved packages. *Alle filer der hører til en packages skal i et directory for dem selv.*

Subdirectory

Dette directory skal være et subdirectory i vores projekt. Directory-strukturen kunne konkret have flg. udseende:

Figur 4:
Placering af subdirectory til package



(der skal laves en ny figur, hvor scr og bin er delt)

Samme navn som package

Subdirectoryet skal have samme navn som package, og det er det eneste directory-navn, der har nogen betydning i eksemplet. Man bemærker, at vi har kaldt vores projekt-directory **Packages**, da eksemplet drejer sig om dette emne, men den kunne have heddet hvad som helst.

Vi kan afprøve indkapslingen af **value** i package, ved at forsøge at tilgå den udefra:

Source 14:
*Forsøg på brud
på indkapsling i
package*

```
import VorPakke.*;

public class TestVorPakke {

    public static void main( String[] argv ) {
        B ref = new B( 7 );

        System.out.println( ref.getValue() );
        ref.value = 5;
        System.out.println( ref.getValue() );
    }
}
```

```
----- Compiler Output -----
TestVorPakke.java:9: Variable value in class VorPakke.B not accessible
                    from class Main.
        ref.value = 5;
          ^
1 error
```

Som det ses, afvises dette af compileren (Linien er knækket af hensyn til bredden).

5. class Dato

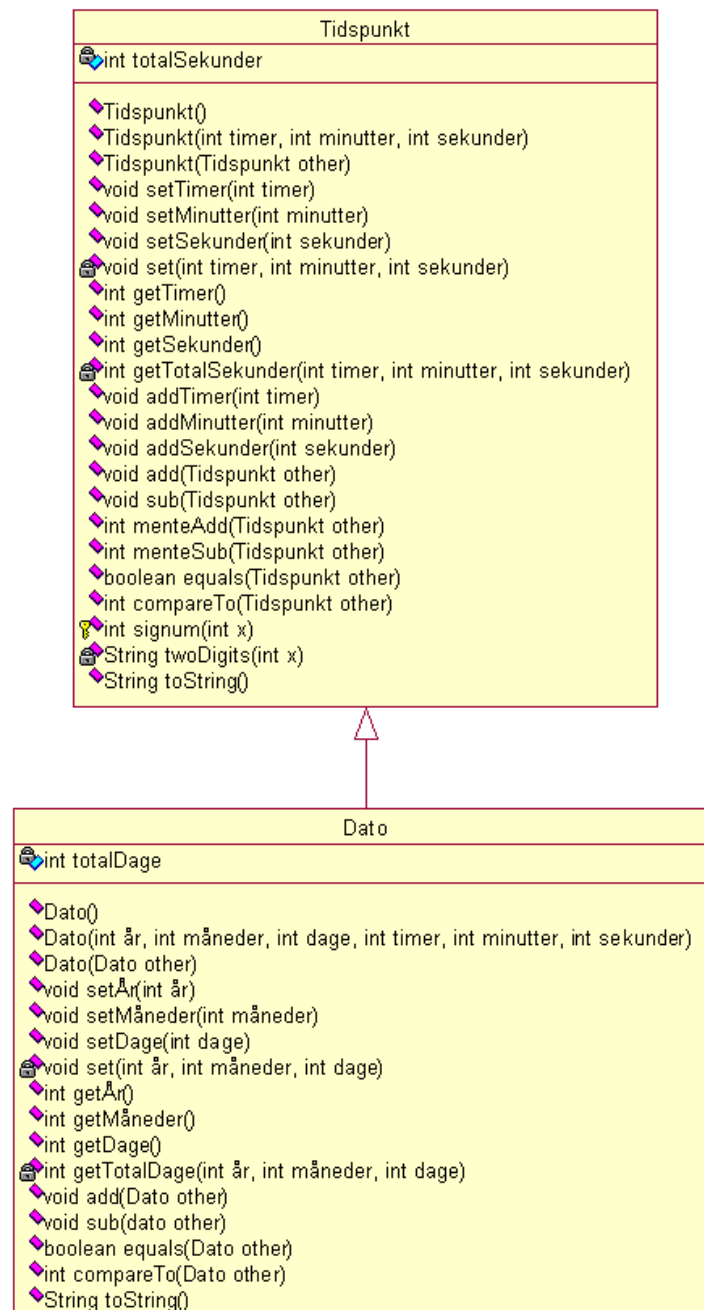
Samme eksempel

Vi vil i det følgende eksempel konstruere klassen **Dato**, svarende til klassen af samme navn i kapitlet "Komposition", men her vil vi i stedet gøre det ved *nedarvning* fra **Tidspunkt**. Den klasse **Dato** vi laver her, skal tilbyde nøjagtig de samme metoder til en klient som den tilsvarende klasse i kapitlet "Komposition".

Det anbefales at man, parallelt med læsningen af dette afsnit, løbende sammenligner med implementationen af **Dato** i kapitlet "Komposition".

Først vil vi se hvordan de to klasser ser ud i et klasse-diagram:

Figur 8:
*Klassediagram
med Dato, der
nedarver fra
Tidspunkt*



Som man ser bliver det et lidt voldsomt diagram, men det viser hvor omfattende en arv **Dato** modtager fra **Tidspunkt**, og man forstår bedre at man ikke gentager de metoder, der nedarves i subklassen. Hvis de nedarverede metoder også skulle anføres i subklassen ville billedet blive endnu mere uoverskueligt.

Sammenligne komposition og nedarvning

Den grundlæggende idé i at bruge samme eksempel, som i kapitlet "Komposition", er naturligvis at kunne lave en sammenligning af de to måder hvorpå man kan lave "noget nyt". Der er derfor kun foretaget nogle ganske få ændringer i **Tidspunkt**-klassen, så som fjernelse af metoder vi i det tidligere kapitel reelt kun overvejede at inkludere i klassen, samt at **signum**-metoden er gjort **protected**, da den også finder anvendelse i **Dato**-klassen..

5.1 Konstruktører

De tre konstruktører i **Dato**, løser alle deres opgave ved hjælp af én af **Tidspunkt**'s konstruktører, der tager sig af timer, minutter og sekunder (implementeret som total-sekunder), mens **Dato**'s egen konstruktør tager sig af den del af datakernen den bidrager med, nemlig år, måned og dag (implementeret som total-dage).

Source 15:

```
public Dato() {
```

Default-konstruktoren

```
totalDage = 0;
}
```

Default-konstruktoren anvender super-klassens default-konstruktør, da en manglende angivelse af et **super**-kald, som bekendt, automatisk afstedkommer et kald af super-klassens default-konstruktør. Selvom man ikke kan se det, er et kald af **Tidspunkt**'s default-konstruktør, den første linie i **Dato**'s default-konstruktør.

Dernæst har vi copy-konstruktoren:

Source 16: Copy-konstruktoren

```
public Dato( Dato other ) {
    super( other );

    this.totalDage = other.totalDage;
}
```

Dato er et Tidspunkt

Copy-konstruktoren anvender **Tidspunkt**'s copy-konstruktør. At vi på denne måde kan anvende en reference til en **Dato** som parameter til en konstruktør der egentlig tager et **Tidspunkt** som parameter, kræver en forklaring! Eftersom en **Dato** har alle de metoder der findes i et **Tidspunkt**, kan den fuldt ud optræde i alle sammenhænge hvor man anvender et **Tidspunkt**. De muligheder som dette forhold mellem super- og sub-klasse giver, skal vi studere indgående i det næste kapitel: "Polymorfi". Indtil videre vil vi blot konstatere at alle metoder der tager et **Tidspunkt** som parameter også kan tage en **Dato** som parameter.

Endelig har vi set-konstruktoren:

Source 17: Set-konstruktoren

```
public Dato( int år, int måneder, int dage, int timer, int minutter, int sekunder ) {
    super( timer, minutter, sekunder );

    totalDage = getTotalDage( år, måneder, dage );
}
```

Set-konstruktoren anvender **Tidspunkt**'s set-konstruktør, da den netop tager **timer**, **minutter** og **sekunder** som parameter. Bemærk, at **set**-metoden er **Dato**-klassens **set**-metode, og ikke den nedarvede, der forøvrigt er **private**.

5.2 Metoder

5.2.1 Get- og set-metoder

De fire get-metoder, og de fire set-metoder, er fuldstændig tilsvarende dem vi lavede i forbindelse med kompositions-løsningen. Det skyldes at vi i disse slet ikke forholder os til **Tidspunkt**.

Metoderne følger her, for fuldstændighedens skyld. Men eftersom vi allerede har diskuteret dem i forbindelse med kompositions-løsningen, vil vi ikke gøre det her:

Source 18: Get- og set-metoderne

```
/*
 * set metoderne
 */
public void setÅr( int år ) {
    set( år, getMåneder(), getDate() );
}

public void setMåneder( int måneder ) {
    set( getÅr(), måneder, getDate() );
}

public void setDate( int dage ) {
    set( getÅr(), getMåneder(), dage );
}

private void set( int år, int måneder, int dage ) {
    totalDage = getTotalDage( år, måneder, dage );
}
```

```

    }

    /*
     * get metoderne
     */
    public int getÅr() {
        return totalDage / ÅR;
    }

    public int getMåneder() {
        return ( totalDage % ÅR ) / MÅNED + 1;
    }

    public int getDate() {
        return totalDage % MÅNED + 1;
    }

    private int getTotalDage( int år, int måneder, int dage ) {
        return år * ÅR + ( måneder - 1 ) * MÅNED + ( dage - 1 );
    }

```

Dernæst vil vi uden tøven lave en **toString**-metode, da den er nyttig til testformål:

Source 19:
toString

```

    public String toString() {
        return "[" + getDate() + "/" + getMåneder() + "-" +
            getÅr() + " " + super.toString() + "]";
    }

```

Man bemærker kaldet af **Tidspunkt**'s **toString**. Da vi i **Dato** overrider **toString** er det nødvendigt med et eksplisit **super**-kald. Kun på den måde er super-klassens **toString** tilgængelig.

5.2.1 Ændringer

Ændringer foretages med metoderne **add** og **sub**. Metoderne ligner indholdsmæssigt meget dem fra kapitlet "Komposition", men kaldet af **menteAdd** henholdsvis **menteSub** er direkte, og ikke via en reference, da metoderne her er nedarvede.

Source 20:
*add og sub
ændrer Dato*

```

    public void add( Dato other ) {
        int mente = menteAdd( other );
        this.totalDage += other.totalDage + mente;
    }

    public void sub( Dato other ) {
        int mente = menteSub( other );
        this.totalDage -= other.totalDage + mente;
    }

```

Her ser vi ligesom i konstruktorerne, at en del af opgaven løses ved at bruge det nedarvede, mens resten løses direkte i subclassesen.

5.2.2 Sammenligninger

Først er der metoden, der tester for lighed.:

Source 21:
Test for lighed

```

    public boolean equals( Dato other ) {
        return this.totalDage == other.totalDage &&
            super.equals( other );
    }

```

Her er super-kaldet igen en erstatning for det delegerende kald fra implementationen med komposition.

Endelig er der metoden, der tester for ulighed:

Source 22:
Test for ulighed

```

    public int compareTo( Dato other ) {
        int sign = signum( this.totalDage - other.totalDage );
    }

```

```

if ( sign == 0 )
    return super.compareTo( other );
else
    return sign;
}

```

Service-metoden **signum** er nedarvet fra **Tidspunkt** og anvendes her med et efterfølgende **super**-kald af **Tidspunkt**'s **compareTo**, der (igen) erstatter det delegerende kald fra implementationen med komposition.

5.3 Sammenligning med **class Dato** fra kapitlet "Komposition"

Flere eller færre objekter

Antallet af objekter er mindre ved nedarvning end ved komposition. Vi har her ét objekt for hver dato - der er en instans af **Dato**. Med komposition har vi to objekter - en instans af **Dato** og en af **Tidspunkt**.

Objekter bliver større ved nedarvning

Ved nedarvning bliver **Dato** "større" end ved komposition. Det skyldes at Dato nu også har de metoder der nedarves fra **Tidspunkt**. Dette gør alt andet lige Dato mere uoverskuelig. Derfor: *Ved nedarvning bliver objekterne "større"*.

Hvad er bedst?

I næste afsnit vil vi generelt se på spørgsmålet: "Hvad er bedst: Komposition eller nedarvning?", men hvad er bedst: At lave klassen **Dato** med komposition eller nedarvning?

Dette eksempel er for simpelt

Her er situationen så simpel, at det ikke gør den store forskel. Det kræver større eksempler end det er praktisk at gennemgå her, for at gøre kvalitets-forskellene tydelige. Det vil dog ikke afholde os fra, i næste afsnit, at se på disse forskelle.

6. Komposition kontra nedarvning

Nedarvning er ikke alt

I en stor del af den objektorienterede litteratur får man det indtryk, at det hele står og falder med nedarvning. I sidste halvdel af 90'erne blev det dog klart at nedarvning nok er en vigtig mekanisme, men det er ikke den eneste måde hvorpå man kan opnå variation. Komposition er, specielt med fremkomsten af design patterns, blevet et betydeligt alternativ. Det betyder at man i dag skal foretage et valg når man designer - før var der kun nedarvning. Eftersom komposition er den senest ankomne af de to, er der i visse kredse en tendens til at betragte den som nedarvningens afløser, men det er for enøjlet. Nedarvning og komposition har fordele og ulemper, og dem må man have for øje når man i hvert enkelt tilfælde træffer sine valg.

6.1 Dynamisk kontra statisk

Komposition er dynamisk

Komposition har en stor fordel frem for nedarvning: den er dynamisk. Under program-udførelsen kan man ændre dynamisk på objekt-systemerne - sætte dem sammen på en ny måde så de fungerer anderledes. Med nedarvning skal man lave en ny klasse og oversætte programmet igen. Nedarvning er statisk!

Komposition er dog afhængig af at de klasser, som man kombinerer instanser af, eksisterer. Selvom man kan ændre adfæren af et program ved at kombinere objekterne anderledes skal disse muligheder være designet før programmet kompiles. Derfor er designet og dermed mulighederne i sidste ende statisk fastlagt.

Som vi skal se i det følgende kapitel om polymorfi, hvor vi vil studere mulighederne for at kombinere objekter mere indgående, afhænger komposition også af nedarvning. Mulighederne for at sammensætte objekter forkrøbles uden nedarvning, og kompositionens styrke bygger derfor delvis på nedarvning.

6.2 Indirection

Nedsat effektivitet

Med komposition kan graden af indirection blive betydelig. Indirection koster altid effektivitet, selvom den kan levere designmæssige kvaliteter. Nedarvning har ikke denne forøgelse af indirection, fordi den direkte adgang til datakernen nedarves, og man derfor går direkte til kernen fra de nye metoder.

Komposition kan overdrives, og prisen i faldende effektivitet kan blive for stor. Alt har sin pris og nedarvning er grundlæggende mere tids-efficient end komposition.

6.3 Komplexitet

Objektsystemer er mere komplicerede end ét stort objekt der klarer den samme opgave. Denne kompleksitet stiller krav til designere og programmører når de skal lave systemer. Moderne EDB-systemer er i forvejen blevet mere og mere komplekse, og det koster. Prisen er, at det bliver lettere at lave fejl, og vanskeligere at finde/rette dem.

Grafiske bruger-grænseflader

Vi bliver hele tiden bedre til at lave systemer, men det er vanskeligt at følge med. Brugere stiller stadig større krav om mere og mere komplekse systemer. 90'ernes betydeligste bidrag på denne front har været den grafiske brugergrænseflade. Alene det har skabt nye fagområder: design af brugergrænseflader og ikke mindst implementation af dem. Design og implementation af brugergrænseflader hviler i høj grad på løsninger der anvender komposition. Derfor har komposition afhjulpet noget af kompleksiteten, men den har selv bidraget med sin egen, i form af flere objekter. Forskellen i objekter mellem et system der laves udelukkende med nedarvning, og et, hvor der anvendes en høj grad af komposition, kan nemt være en faktor 5-10. Denne forøgelse af objektmængden stiller større krav til designerne.

6.4 Hvad så?

Høj grad af komposition

Hvad skal man så gøre? Man må foretage et valg - et valg baseret på en indsigt i hvad man laver og hvordan man kan gøre det. Antallet af udviklere der mestrer komposition og en bred vifte af design patterns, er relativ beskeden. Derfor er svaret ikke kun, at man skal anvende en relativ høj grad af komposition suppleret den med nedarvning. Realiteten er, at man skal gøre det man kan bedst. Hvis man prøver at lave komposition uden den nødvendige uddannelse og en vis erfaring, vil man stå i endnu større problemer, end hvis man kun holdt sig til nedarvning.

Repetitionsspørgsmål

- 1 Hvad er nedarvning?
- 2 Hvordan beskriver man nedarvning i et klassediagram?
- 3 Hvad betyder super- og subklasse?
- 4 Hvordan angiver man nedarvning i Java?
- 5 Hvordan angiver man instansvariable og metoder i et klassediagram?
- 6 Hvad er symbolet for **private**?
- 7 Hvad er "erstatnings-symbolerne" for **public** og **private**?
- 8 Hvad får man ud af at arve en **public** instansvariabel?
- 9 Hvad får man ud af at arve en **private** instansvariabel?
- 10 Hvad får man ud af at arve en **public** metode?
- 11 Hvad får man ud af at arve en **private** metode?
- 12 Hvad vil det sige at konstruktører ikke (rigtig) nedarves?
- 13 Hvordan kan man kalde en konstruktor i superklassen fra subklassen?
- 14 Den automatiske default-konstruktor indeholder én linie - Hvilken?

- 15 Hvad er overriding?
- 16 Er en overridden metode tabt?
- 17 Hvad er en package?
- 18 Hvorfor laver man packages?
- 19 Hvad er *rent* package scope?
- 20 Hvad er **protected**?
- 21 Hvad er symbolet for **protected**?
- 22 Hvad skal der stå i alle filer, med klasser der tilhører en package?
- 23 Hvorfor vælger vi normalt at bruge **protected** i stedet for *rent* package scope?
- 24 Hvor skal man placere alle filer, med klasser der hører til en package?
- 25 Når man bruger nedarvning i stedet for komposition kalder man ikke metoder fra superklassen via en reference. Hvad gør man i stedet?
- 26 Hvorfor er der færre objekter i nedarvning end komposition?
- 27 Hvorfor bliver objekterne "større" ved nedarvning end ved komposition?
- 28 Hvorfor er der ikke tydelige kvalitetsforskelle mellem komposition og nedarvning i eksemplet med **Dato**?
- 29 Hvad mente man tidligere var det klart vigtigste begreb i objektorienteret programmering?
- 30 Hvorfor er komposition dynamisk og nedarvning statisk?
- 31 Hvad er problemet med indirection?
- 32 Hvorfor er komposition mere komplekst end nedarvning?
- 33 Hvad bør man vælge?

Svar på repetitionsspørgsmål

- 1 At man laver en ny klasse ud fra en allerede eksisterende klasse.
- 2 Klasserne beskrives med hver sin kasse, og nedarvning beskrives som en pil fra klassen der arver til den klasse den arver fra.
- 3 Super-klassen er den klasse som en klasse arver fra. Subklasser er klasser, der nedarver fra klassen.
- 4 I subklassen, efter angivelsen af klassens navn, skrives **extends** efterfulgt af navnet på den klasse der nedarves fra.
- 5 I det midterste felt i kassen angives erklæringen af instansvariable. Nederst i kassen angives signaturerne for metoderne.
- 6 En hængelås.
- 7 '+' for **public** og '-' for **private**?
- 8 De blive også **public** i subklassen.

- 9 Ikke ret meget. De er kun indirekte tilgængelige via metoder, der er arvet fra superklassen.
- 10 Det samme som svaret på nr. 8.
- 11 Det samme som svaret på nr. 9.
- 12 De er kun tilgængelige via super-kald fra subclasses konstruktører.
- 13 Ved **super**-kald, der syntaktisk ligner **this**-kald af konstruktører i selve klassen, blot skriver man **super** i stedet for **this**.
- 14 Et kald af superklassens default-konstruktør.
- 15 At man erstatter noget med noget andet.
- 16 Nej, den kan stadig kaldes via et direkte super-kald, analogt til et **this**-kald.
- 17 En samling af klasser
- 18 Hvis en gruppe af klasser logisk hører sammen og/eller hvis man ønsker, at de skal have et fælles scope: package scope.
- 19 Når man ikke anfører nogen tilgængelighedsangivelse. Package scope betyder, at det er **public** i package, men **private** udenfor.
- 20 Det samme som package scope, dog med den ekstra mulighed, at subclasser udenfor package til klasser i package har adgang til **protected** dele af deres superklasse.
- 21 '#'
- 22 Der skal stå **package** efterfulgt af navnet på den package de skal tilhøre. Dette skal være den første rigtige linie i filen.
- 23 Fordi **protected** gør kildeteksten mere læsbar, og vi sjældent har brug for den nuance-forskel der er mellem rent package scope og **protected**.
- 24 De skal placeres i det samme directory, som skal hedde det samme som package.
- 25 Man kan kalde dem direkte, da de er nedarvet. Hvis man har overridet dem, er det nødvendigt at gøre det via et **super**-kald.
- 26 Fordi nedarvning bygger videre på en klasse og laver ét samlet objekt, mens komposition deler det op i flere klasser og dermed flere objekter.
- 27 Fordi subclasser supplerer den arv de modtager fra superklassen, men nye ting. Derved blive subclassen "større".
- 28 Fordi eksemplet er for simpelt.
- 29 Nedarvning.
- 30 Fordi man med komposition har mulighed for at udskifte objekter og kombinere dem på nye måder mens programmet kører.
- 31 At det koster tid. Med nedarvning undgår man indirection og implementationen bliver derfor mere tids-efficient.
- 32 Fordi man skal holde styr på flere objekter ved komposition, end ved nedarvning.
- 33 Man bør vælge en relativ høj grad af komposition suppleret med nedarvning.