MENU

Klasser

Opgaver

En klasse er en skabelon

Når man i java skal lave objekter, gør man det ved instantiering, dvs. ud fra en skabelon. En skabelon i java kaldes en **`class`**, eller i daglig tale en **klasse**.

En klasse beskriver hvordan et objekt skal opbygges når det instantieres.

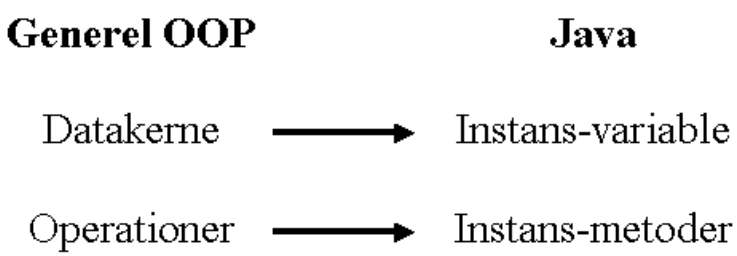
Datakernen er variable

Klassen beskriver de variable der skal udgøre datakernen. Disse kaldes **instans-variable** fordi de vil befinde sig i de instanser der laves ud fra klassen. Variablene er grundlæggende som andre variable vi tidligere har set. Den eneste forskel fra variable i almindelighed, ligger i indkapslingen i det objekt der instantieres.

Operationer er metoder

Klassen beskriver også de operationer der skal være på objekterne. Disse laves med **metoder**. Det betyder at requests er metodekald og returnering fra en request ligeledes er returnering fra en metode. Metoderne kaldes **instans-metoder**.

Termerne "instans-variable" og "instans-metoder" bruges primært når betegnelserne "variable" og "metoder" er upræcise ud fra sammenhængen. Der findes andre variable og metoder, der kaldes "klasse-variable" og "klasse-metoder", men indtil vi skal se på dem, er der sjældent mulighed for forveksling.



Figur 1: *Sammenhæng mellem begreber*

Lad os se en simpel klasse:

Source 1: Simpel klasse

Heltal.java

```
1  public class Heltal {
2    int tal;
3
4    void set( int t ) {
5      tal = t;
6    }
7
8    int get() {
9      return tal;
10   }
11 }
```

Instanser af denne klasse vil have en datakerne bestående af variablen **`tal`**. Instanser vil ligeledes have to operationer, metoderne **`set`** og **`get`**.

Virkefelt

Virkefeltet for disse tre, er området mellem de ydre tuborg paranteser. Det betyder at metoderne kan tilgå instansvariablene: dvs. ændre dem eller aflæse dem, og på den måde fungere som operationer på den datakerne som instans-variablene udgør. Metoder i en klasse kan kalde hinanden, samtidig med at de deles om instansvariablene. I sådanne kald vil man derfor aldrig sende dele af datakernen med som parametre, da metoderne allerede har adgang til disse data.

`public` og `private` styrer indkapsling

I eksemplet har vi ikke implementeret nogen indkapsling. Indkapslingen realiseres med **synligheds angivelser**. Der findes en række af disse, men vi vil i første omgang holde os til to, nemlig **`public`** og **`private`**. Angivelserne placeres umiddelbart foran dét, hvis synlighed, de regulerer. Dette gælder både for instansvariable og metoder.

`public` er tilgængeligt udefra

`public` betyder at noget er tilgængeligt udefra (dvs. udenfor objektet). Som udgangspunkt er det kun metoder vi gør **`public`**. Det skyldes at vi bruger metoder til at realisere operationerne og deres formål netop er at blive kaldt udefra.

`private` er indkapslet

`private` betyder at noget ikke er tilgængeligt udefra. Som udgangspunkt er alle instansvariable **`private`**. Det skyldes at datakernen netop ikke skal være tilgængelig udefra, idet den skal være indkapslet. Instansvariable vil derfor *aldrig* være **`public`**.

Hvis vi anvender disse angivelser i vores eksempel får vi:

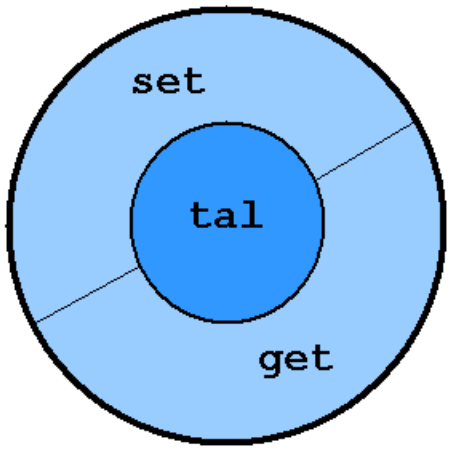
Source 2: Klasse med public og private

Heltal.java

```
1  public class Heltal {
2    private int tal;
3
4    public void set( int t ) {
5      tal = t;
6    }
7
8    public int get() {
9      return tal;
10   }
11 }
```

Man kunne illustrere en instans af denne klasse med følgende figur:

Figur 2:
Heltals-objekt



Her ligger de to metoder beskyttende omkring instansvariablen, og kun vha. disse metoder kan man udefra påvirke data-kernen.

Definition: Interface

Et objekts interface er samlingen af signaturer for de metoder som er tilgængelige udefra.

De to metoders signaturer er objektets interface, idet de udgør den mulighed andre objekter har for at kommunikere med instanser af **`Heltal`**.

1. set- og get-metoder

Omgår indkapsling

`tal` er umiddelbart indkapslet, men er den *reelt* beskyttet?

Selve instansvariablen er `private`, men metoderne `set` og `get` giver den samme adgang som hvis `tal` havde været `public`. `tal` er derfor i realiteten ikke indkapslet.

Undergraver objektorientering

set- og get-metoderne er meget simple, men de undergraver det objektorienterede, da de indirekte åbner ind til datakernen.

Kan ikke altid undgåes

I virkelighedens verden kan vi ikke undgå set- og get-metoder. De er en enkel og praktisk løsning i mange situationer, men sunde er de ikke. Man kunne alternativt til set- og get-metoder vælge at gøre den tilhørende instansvariabel `public`, som konsekvens af at brudet på indkapslingen alligevel realiseres af set- og get-metoderne.

Stort behov skyldes manglende objektorientering

Man bør dog vælge set-/get-metoderne frem for `public`, da man ellers sammenblander forskellige principper (paradigmer) for programmering; hvilket er en kilde til andre problemer. Nogle gange hører man argumentet: "Jamen, hvis jeg ikke laver dem `public`, skal jeg lave så mange set- og get-metoder". Det er meget sjældent, at dette er en berettiget klagesang. Behovet for mange set- og get-metoder skyldes næsten altid at man grundlæggende ikke har tænkt objektorienteret, da man designede klasser og objektsystemer.

`public` er endnu mindre objektorienteret

set- og get-metoder undergraver det objektorienterede, men `public` gør det totalt! Med `public` er man ovre i record-perspektivet på objekter; hvor de er døde entiteter som opbevarer data uden nogen tilhørende funktionalitet.

Når vi har valgt at starte med set-/get-metoder, er det fordi de er meget simple, samtidig med at vi tidligt berører det væsentlige begreb: **indkapsling**.

2. Instantiering

Lad os lave en instans af klassen `Heltal`, og anvende den:

Source 3: Instans af Heltal

Heltal.java

```java
public class Heltal {
  private int tal;

  public void set( int t ) {
    tal = t;
  }

  public int get() {
    return tal;
  }
}
```

Main.java

```java
public class Main {

  public static void main( String[] argv ) {
    Heltal vorTal = new Heltal();

    vorTal.set( 5 );
    System.out.println( vorTal.get() );
  }
}
```

```
5
```

Selvom dette eksempel stadig er simpelt, indeholder det flere nye ting.

Klassen med `main` har kun til

Der er to klasser. Først er der klassen `Heltal` fra før, dernæst er der klassen `Main`. Som bekendt starter enhver programudførelse med et kald af `main`-metoden. Vi laver derfor en klasse udelukkende med dette formål (en klasse vi aldrig laver instanser af!). De to klasser kan enten placeres i den samme file, eller i to forskellige. Man bør dog altid placere dem i forskellige filer, da det er mest hensigtsmæssigt i det lange løb.

formål at starte programmet

`vorTal` er en reference-variabel

I **`main`**-metoden har vi en testanvendelse af vores klasse **`Heltal`**. Vi erklærer først en variabel **`vorTal`**. **`vorTal`** er en såkaldt reference-variabel. Vi har tidligere lavet reference-variable i forbindelse med arrays; hvor de kunne referere til et array. F.eks.:

Source 4: Reference-variabel og array

```
int[] vorTabel = new int[10];
```

Instantierer objekt med `new`

Ligheden med første linie i **`main`**-metoden er slående. I vores eksempel står **`Heltal`** på typens plads og den optræder ligeledes efter **`new`**. Ligheden er ikke tilfældig, for betydningen er den samme! Vi laver en reference der kan referere til en instans af **`Heltal`**; hvilket fremgår af typeangivelsen for variabelnavnet. Ligeledes betyder **`new`** at vi allokerer noget i lagret, i dette tilfælde et objekt - en instans af **`Heltal`**; hvilket fremgår af typeangivelsen efter **`new`**. Betydningen af paranteserne vender vi tilbage til senere.

Kald via reference

Efter erklæring af referencen og instantiering af et objekt, følger en anvendelse. Man sender requests til et objekt ved at kalde metoder. Disse **metoder kaldes via referencer** til objektet. Det sker ved anvendelse af følgende syntaks:

Syntax 1: Kald gennem reference

```
<reference>.<metodekald>
```

Det er det vi gør i kaldet:

Source 5: Kald af metode via reference

```
vorTal.set( 5 );
```

<reference> var **`vorTal`** og <metodekald> var **`set( 5 ).`**

Flere referencer

Ligesom vi med arrays kunne have flere referencer til det samme array, kan vi have flere referencer til det samme objekt. Vi kunne f.eks. ændre anvendelsen til:

Source 6: To referencer til samme objekt

(Main.java)

```
1  Heltal vorTal = new Heltal();
2  Heltal osseVor;
3
4  osseVor = vorTal;
5  vorTal.set( 5 );
6
7  System.out.println( osseVor.get() );
```

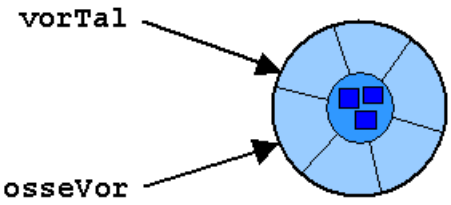
```
5
```

Hedder ikke noget

Dette demonstrerer en anden egenskab ved objekter, eller man skulle måske snarere kalde det en manglende egenskab. De hedder ikke noget!

Det er referencerne der hedder **`vorTal`** og **`osseVor`**, objektet selv har ikke noget navn.

Figur 3:
To referencer til samme objekt



Sædvanligvis har man kun én reference til et objekt, og der er normalt ikke gjort nogen skade ved, at man bruger referencens navn til også at benævne objektet.

3. Konstruktorer

Skal "sættes i gang"

Vores **`Heltal`**-objekt havde brug for at vi initialiserede det med en passende værdi. Derfor må den første anvendelse forventes at være et kald af set-metoden. Det er ikke specielt for klassen **`Heltal`**, langt de fleste objekter i denne verden har brug for at "starte op" og gøre sig klar til tilværelsen i et objektsystem. Derfor har man en speciel metode-lignendende indretning der kaldes en konstruktor. Den er ikke en metode, men den ligner. Lad os se eksemplet igen, med en passende konstruktor:

Source 7: Konstruktor

Heltal.java

```
1  public class Heltal {
2    private int tal;
3
4    public Heltal( int t ) {
5      tal = t;
6    }
7
8    public void set( int t ) {
9      tal = t;
10   }
11
12   public int get() {
13     return tal;
14   }
15 }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Heltal vorTal = new Heltal( 5 );
5
6      System.out.println( vorTal.get() );
7    }
8  }
```

```
5
```

Ingen retur-type

Den metodelignende indretning i starten af klassen adskiller sig kun syntaktisk fra en metode ved at den ikke har nogen retur-type, den kan nemlig ikke returnere noget. En anden speciel syntaktisk egenskab ved konstruktoren er at den altid skal hedde det samme som klassen.

Kan ikke kaldes

Semantisk er forskellene fra en metode større. En konstruktor kan ikke kaldes direkte. Den kaldes indirekte ved instantieringen som det ses i anvendelsen. Ved at anføre 5 i instantieringen bliver konstruktoren kaldt med denne parameter. Datakernen bliver initialiseret og der "returneres". Rent syntaktisk ligner instantieringen efter **`new`** et kald af konstruktoren og der er ikke noget problem i at tænke på den som sådan.

Navlestreng

Instantieringen er den eneste situation hvor konstruktoren kan blive udført, derefter træder den fuldstændig ud af billedet, som en "navlestreng" der har udtjent sit formål.

3.1 Kode-redundans

Gør det samme

Redundans (eng.: redundancy, dk.: overflødighed) betyder generelt at noget forekommer flere gange/steder. **Kode-redundans** er når flere dele af kildeteksten gør det samme, at de derfor i virkeligheden er den samme kode.

Kilde til fejl

Af hensyn til vedligeholdelsen af programmer er det generelt en fordel, at kode som gør det samme kun forekommer ét sted, at man undgår koderedundans! Koderedundans bevirker nemlig, at man skal foretage eventuelle ændringer flere steder. Ud over at det forøger arbejdsbyrden ved vedligeholdensen, er det også en potentiel kilde til fejl - husker man at få ændret alle steder?

I vores eksempel sætter vi to steder direkte `tal` til en værdi: I konstruktoren og i set-metoden. Vi kunne nøjes med at have denne kode i set-metoden, og i konstruktoren kalde denne metode:

Source 8: Metode-kald fra konstruktor

(Heltal.java)

```
1  public Heltal( int t ) {
2    set( t );
3  }
4
5  public void set( int t ) {
6    tal = t;
7  }
```

Kun ét sted

Nu står tildelingen kun ét sted, og vi skal nu kun huske at rette ét sted; hvis vi vil ændre på hvad der sker ved tildelingen.

Kan være meget

Eksemplet er *meget* simpel, og kun lidt er vundet, men det illustrerer princippet og det kunne have været meget mere. Jeg har set projekter hvor op til en side var gentaget flere steder i programmet.

4. `class Tidspunkt`

Vi vil i det flg. lave en klasse til at repræsentere et tidspunkt. Vi starter med at overveje designet og ser dernæst på implementationen.

4.1 Design-overvejelser

De første overvejelser man kunne gøre sig, ville måske være i retning af: "Et objekt til at repræsentere et tidspunkt, det må være noget med at jeg kan sætte den til at huske et tidspunkt, og jeg så senere kan hente det igen".

Vi er aktive og objektet er passivt

Det er forståeligt at man kan gøre sig sådanne tanker, men de er tydeligvis ikke særlig objektorienterede. Der er grundlæggende en ting der er galt. Lad os tage det en gang til med kursivering: "Et objekt til at repræsentere et tidspunkt, det må være noget med at ***jeg kan sætte*** den til at huske et tidspunkt, og ***jeg*** så senere ***kan hente*** det igen". Problemet er, at vi er på vej til at lave et ***passivt*** objekt!

Hvad kan objektet gøre for mig

Naturligvis skal vi også kunne gøre de nævnte ting, men perspektivet overfor tager udgangspunkt i én selv, og det ender hurtigt med at objektet blot holder data, mens vi gør arbejdet. Den grundlæggende design-idé for ethvert rigtigt objekt er ikke: "Hvad kan jeg gøre ved objektet", men: "Hvad kan objektet gøre for mig". Objekter skal være levende, selvstændige entiteter, der fungerer i et samspil med hinanden.

Som nævnt skal man trods alt kunne det, der i de første overvejelser lægges op til, men det er en uvæsentlig del af objektet. Det må være en bi-ting! Det første og mest væsentlige for vores objekt kan ikke være set- og get-metoder til tidspunktet. Det må i stedet være den rolle objektet skal udfylde.

Objektets rolle

For at kunne beskrive den rolle som objektet skal spille, og dernæst designe det så det udfylder denne rolle, kræver naturligvis et godt kendskab til hvad objektet skal kunne. Her har vi kun en meget generel idé ud fra klassens navn: `Tidspunkt`, og må i første omgang nøjes med det, da der er tale om et eksempel.

Vi kan derfor kun gøre os generelle overvejelser. "Objektet skal repræsentere et tidspunkt, det skal kunne ændres, kunne sammenlignes med andre tidspunkter osv.".

Puste liv i objektet

Hvordan kan vi puste liv i objektet? Hvordan kan det komme til at spille en aktiv rolle, og ikke hensygne i passivitet?

Hvis vi f.eks. ønsker at ændre tidspunktet, så det sættes en time længere frem, må vi lade objektet selv gøre det, sig: "Stil dig en time frem!". Det værste af alt ville være: "Fortæl mig hvad tid du er, så ændrer jeg det, og fortæller dig hvad din nye tid skal være".

Eller hvis vi f.eks. ønsker at sammenligne to tidspunkter, så lad dem selv løse opgaven. Man tager det ene tidspunkt, og "giver" det til det andet, idet man siger: "Er du et senere tidspunkt end det her tidspunkt?"

Eller hvis vi vil lægge to tidspunkter sammen. Så giv det ene tidspunkt til det andet, og sig: "Giv mig et nyt tidspunkt, som er dig og det her tidspunkt lagt sammen"

Normalt vil man gøre designet mere fuldendt før man begynder at implementere klassen, men da `Tidspunkt` er relativ simpel, og det kun er den anden klasse vi ser implementeret, vil vi forlade designovervejelserne og vende os mod implementationen.

4.2 Implementation

I det følgende vil vi starte med datakernen, dernæst konstruktorerne og endelig operationerne. Det er en naturlig rækkefølge, ikke alene ved implementationen men også ved opbygningen af et mere detaljeret design end det vi har lavet ovenfor.

4.2.1 Datakernen

Klokkeslæt

Datakernen skal indeholde en tidsangivelse. Vi vil afgrænse det til et vist antal timer, minutter og sekunder, og dermed bliver tidspunktet et klokkeslæt.

Den første idé er ikke altid den bedste

Spørgsmålet er så, om vi skal lade datakernen bestå af disse tre værdier: timer, minutter og sekunder, f.eks. i form af tre integers? Det er en nærliggende idé, da den ligger tæt op af vores forståelse af hvad objektet skal repræsentere. Problemet er blot, at det ikke altid giver den simpleste implementation. Den del af objektet vi kommer til at se er interfacet, dvs. metoderne. Hvordan selve datakernen er opbygget kommer vi som "brugere" af tidspunkter aldrig til se - det er kun den der designer og implementerer selve klassen, der arbejder med den side af sagen. Man bør derfor altid overveje hvilken repræsentation i datakernen der gøre det mest enkelt at implementere metoderne, da dette både sparer tid under udviklingen, og gør det nemmere at sikre kodens korrekthed.

En bedre idé

Hvilke andre muligheder er der? Et alternativ kunne være at repræsentere tidspunktet med en enkelt integer. Det ville gøre det enklere at lægge tidspunkter sammen, at sammenligne dem osv. Men hvordan skulle et tidspunkt kunne repræsenteres som én enkelt integer? Det kan det fordi der er en enkel sammenhæng mellem timer, minutter og sekunder. Et minut er 60 sekunder, og en time er 60 minutter. Man kan derfor i stedet regne det hele sammen i sekunder; hvad vi vil kalde **total-sekunder**. F.eks. kan tidspunktet 2:12:34 omregnes til total-sekunder som: 2*3600 + 12*60 + 34 = 7954 sekunder, idet en time er 60 minutter á 60 sekunder = 60*60 = 3600 sekunder.

Fordele og ulemper

Uanset hvilken løsning man vælger på et problem, vil der være fordele og ulemper forbundet med den. Spørgsmålet er naturligvis altid hvilken løsning der giver det bedste trade off mellem de to.

Hvis vi vælger løsningen med timer, minutter og sekunder har vi data på den form der naturligt passer til interfacet. Til gengæld skal vi internt foretage en del ekstra arbejde da der er tale om tre integers. Dette i sig selv er ikke nogen stor ulempe, til gengæld får vi et andet problem som er langt større. Når vi foretager ændringer af timer, minutter og sekunder kan det ske at minutter og sekunder bliver mere end 59, eller evt. at de bliver negative. Det betyder at vi ved ændring af disse variable skal foretage en efterjustering, en normalisering af tidspunktet. En sådan normalisering er ikke helt enkelt at implementere, selvom den naturligvis kan laves.

Vælger vi derimod at arbejde med total-sekunder, bliver normaliseringen meget enkel, da minutter og sekunder ikke er direkte repræsenteret og man derfor blot kan regne løs. Til gengæld skal man kunne omregne total-sekunder til timer, minutter og sekunder af hensyn til interfacet; hvor en "bruger" af objektet må kunne forvente at få disse oplysninger. En sådan omregning er relativ enkel at lave vha. heltals-division og modulus, så det er kun en mindre ulempe.

Konklussionen på vores overvejelser omkring datakernen, bliver derfor at vi vil repræsentere et tidspunkt vha. total-sekunder.

Source 9:

MENU

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    private int totalSekunder;
3
4    ...
5  }
```

Indkapsling

Vi indkapsler som altid instansvariablen med **`private`**, for at beskytte den mod direkte tilgang udefra.

4.2.2 `toString`

Nyttig til test-formål

Inden vi går videre til konstruktorerne og metoderne, vil vi først lave en meget nyttig metode. Metoden anvendes primært til test-formål; hvor den giver os mulighed for på en simpel måde at observere objektet efterhånden som vi arbejder med det. Denne metode er en **toString-**metode der skal returnere en tekststreng som beskriver objektets tilstand. I vores eksempel bliver det selve tidspunktet, som tekststreng, der skal returneres:

Eftersom tidspunktet i datakernen er repræsenteret som total-sekunder støder vi dog allerede her på den ulempe der er nævnt ovenfor - at vi kan være nød til at regne tilbage til timer, minutter og sekunder afht. interfacet - i dette tilfælde **toString**-metoden. Vi vil derfor lave tre get-metoder der returnerer netop disse tre værdier:

Source 10: get-metoder til timer, minutter og sekunder

(Tidspunkt.java)

```
1  public int getTimer() {
2    return totalSekunder / 3600;
3  }
4
5  public int getMinutter() {
6    /*
7     * tager først modulus med TIME for at få fjernet timerne, og dermed kun
8     * have minutter og sekunder regnet sammen til totalsekunder.
9     *
10    * dernæst tages division med MINUT for at få hele minutter.
11    */
12   return ( totalSekunder % 3600 ) / 60;
13 }
14
15 public int getSekunder() {
16   return totalSekunder % 60;
17 }
```

Vi genkender de 3600 sekunder der udgør en time, og de 60 sekunder der ligeledes udgør et minut. Formlerne for timer og sekunder er relativ simple; hvis man ellers er fortrolig med heltals-division og modulus, mens formlen for minutter er lidt mere kompliceret. Da emnet for vores gennemgang er objektorienteret programmering i Java, og ikke matematiske formler, overlades det til læseren at forvisse sig om at disse formler er korrekte. Derimod er der en anden ting ved formlerne vi vil berøre, nemlig at der gentagne gange optræder tal-konstanter i dem. I skrivende stund er vi godt klar over hvad der menes med 3600 og 60, men det ville være lettere at huske, hvis der i stedet var anvendt navne.

Vi vil derfor indføre en række konstanter:

Source 11: Konstanter

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    private static final int MINUT = 60;
3    private static final int TIME = 60 * MINUT;
4
5    ...
6
7    public int getTimer() {
8      return totalSekunder / TIME;
9    }
10
```

```
11    public int getMinutter() {
12      return ( totalSekunder % TIME ) / MINUT;
13    }
14
15    public int getSekunder() {
16      return totalSekunder % MINUT;
17    }
18
19    ...
20  }
```

Konstanter

Erklæringen af konstanterne ligner erklæringen af instansvariable, men der er ikke tale om variable. Betegnelsen **final** betyder, at der er tale om konstanter — at de ikke kan ændres! Betydningen af **static** skal vi ikke berøre her, men blot notere os, at man normalt erklærer konstanter i klasser som værende **static**. Ønsker man at kende betydningen af **static**, kan man læse kapitlet: "Klasser som objekter" under "Java specielt", men det anbefales at man venter indtil man er mere fortrolig med objektorienteret programmering.

Vi er nu klar til at lave vores **toString**-metode:

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public String toString() {
5      return "[" + getTimer() + ":" + getMinutter() + ":" + getSekunder() + "]";
6    }
7  }
```

Source 12: toString

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt();
5
6      System.out.println( vorTid.toString() );
7    }
8  }
```

```
[0:0:0]
```

Kantede paranteser

Vi har her valgt at **`toString`** sætter kantede paranteser omkring beskrivelsen af objektets tilstand. Det er en konvention jeg selv har indført, da den hjælper til at identificere teksten som en tilstand i udskrifter.

Initialiseres automatisk til 0

Vi ser her at timer, minutter og sekunder fra starten er 0; hvilket skyldes at **totalSekunder** er 0, selvom vi ikke noget sted har tildelt **totalSekunder** denne værdi. Det skyldes at instans-variable, der er integers, automatisk initialiseres til 0 ved instantieringen. Det er en egenskab ved instans-variable som vi normalt ikke vil udnytte, da det gør programmet vanskeligere at læse: "Er det meningen at de skal være 0, eller har vi glemt at sætte dem til noget?". Hvis vi selv sætter dem til 0, er der ikke noget at være i tvivl om - vi har kommunikeret et klart budskab til læseren!

Inden vi går videre, vil vi løse et mindre problem der viser sig i udskriften. Vi ønsker normalt ikke at få angivet et tidspunkt som 0:0:0, men i stedet: 0:00:00; hvor minutter og sekunder altid angives med to cifre. Da vi ønsker dette problem løst for både minutter og sekunder, er det hensigtsmæssigt at lave en metode til formålet. Metoden skal tage en integer mellem 0 og 99 som parameter og returnere en teksstreng der viser tallet med to cifre:

Source 13: Metode der returnerer tal med to cifre

(Tidspunkt.java)

```
1  // PRE: x tilhører [0:99]
2  private String twoDigits( int x ) {
3    if ( x >= 10 ) // er allerede to-cifret
4      return "" + x;
5    else
```

```
    return "0" + x;
}
```

Precondition

Vi har her placeret en kommentar før metoden der fortæller, at det er en forudsætningen for at metoden fungerer korrekt, at parameteren **x** ligger i intervallet fra 0 til 99. Forkortelsen PRE står for **precondition**, der er den normalt anvendte engelske betegnelse for en sådan forudsætning i forbindelse med programmering. Enhver der bruger metoden er selv ansvarlig for at overholde en sådan precondition i forbindelse med kald af metoden.

Selve implementationen af metoden skelner mellem om tallet allerede er to-cifret, eller der skal sættes et "0" foran.

Service-metode

Man bemærker at metoden er **private** i modsætning til de almindeligvis **public** metoder vi normalt laver. Det skyldes, at vi kun har lavet metoden med henblik på, at andre metoder i klassen skal kunne kalde den. Vi ønsker ikke at man skal kunne kalde metoden udefra, da den ikke repræsenterer en egenskab ved et tidspunkt, men blot er en funktionalitet, der er nyttig at anvende internt i klassens egen implementation. En sådan metode kaldes generelt en **service-metode**.

Lad os se hvordan **toString**-metoden kan anvende **twoDigits**-metoden:

(Tidspunkt.java)

```
public class Tidspunkt {
  ...

  public String toString() {
    return "[" + getTimer() + ":" +
              twoDigits( getMinutter() ) + ":" +
              twoDigits( getSekunder() ) + "]";
  }
}
```

Source 14: toString med to cifre

Main.java

```
public class Main {

  public static void main( String[] argv ) {
    Tidspunkt vorTid = new Tidspunkt();

    System.out.println( vorTid.toString() );
  }
}
```

```
[0:00:00]
```

Vi har nu opnået det ønskede resultat.

4.2.3 Fra timer, minutter og sekunder til totalsekunder.

Vi så ovenfor, at det med rimelig simple matematiske beregninger, var muligt at lave get-metoder til timer, minutter og sekunder.

set-metoder

Da den interne repræsentation med totalsekunder under alle omstændigheder må ligge noget fra det vi ønsker at præsentere i interfacet (en get- eller set-metode der returnerer eller sætter datakernens totalsekunder må formodes ikke at stå særlig højt på ønskelisten) vil det være nærliggende at have tilsvarende set-metoder, der kan sætte timer, minutter og sekunder. Sådanne metoder vil være nyttige i forbindelse med implementationen af resten af klassen.

Disse metoder laves nemmest hvis vi har en service-metode der omregner timer, minutter og sekunder til totalsekunder:

Source 15: Servicemetode der beregner totalsekunder

(Tidspunkt.java)

```
1  private int getTotalSekunder( int timer, int minutter, int sekunder ) {
2    return timer * TIME + minutter * MINUT + sekunder;
3  }
```

Vi ser her, hvordan vi igen anvender konstanterne som blev erklæret tidligere. **TIME**, der er antallet af sekunder i en time (3600), og **MINUT**, der er antallet af sekunder i et minut (60).

Før vi laver de tre set-metoder til timer, minutter og sekunder, vil vi lave en samlet set-metode:

Source 16: Samlet set-metode

(Tidspunkt.java)

```
1  public void set( int timer, int minutter, int sekunder ) {
2    totalSekunder = getTotalSekunder( timer, minutter, sekunder );
3  }
```

Denne metode anvender service-metoden: **getTotalSekunder**, som vi netop har lavet, til at sætte datakernens totalsekunder ud fra **timer**, **minutter** og **sekunder**.

De tre set-metoder bliver nu relativ simple at implementere:

Source 17: De tre set-metoder

(Tidspunkt.java)

```
1  public void setTimer( int timer ) {
2    set( timer, getMinutter(), getSekunder() );
3  }
4
5  public void setMinutter( int minutter ) {
6    set( getTimer(), minutter, getSekunder() );
7  }
8
9  public void setSekunder( int sekunder ) {
10   set( getTimer(), getMinutter(), sekunder );
11 }
```

Regner frem og tilbage

Man ser hvordan de tre set-metoder alle løser deres opgave ved først at omregne fra totalsekunder til timer, minutter og sekunder, og dernæst igen tilbage til totalsekunder, idet den ene af de tre erstattes med parameteren.

F.eks. kalder **setTimer**-metoden: **getMinutter()** og **getSekunder()** og supplerer selv med antallet af timer, der er givet med som parameter. Dernæst regnes det hele tilbage til totalsekunder vha. den samlede **set**-metode der tager alle tre som parametre.

Totalsekunder er stadig bedre end timer, minutter og sekunder

Man kan stille spørgsmålstegn ved hvor effektivt det er at foretage denne beregning frem og tilbage, men det er en enkel måde at løse problemet på. Til gengæld virker det måske ikke så overbevisende at valget af totalsekunder som datakerne frem for tre integers: timer, minutter og sekunder skulle være så godt. Det viser sig dog, at disse indledende besværligheder nemt opvejes af, at langt de fleste af de metoder vi skal implementere i det følgende bliver så simple, at de endog kun indeholder én linie! Dette ville ingenlunde være tilfældet hvis vi havde valgt en datakerne med timer, minutter og sekunder!

4.2.4 Konstruktorer

Det er oplagt at lave en konstruktor der tager timer, minutter og sekunder som parametre:

Source 18: Konstruktor med timer, minutter og sekunder

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    private int totalSekunder;
5
```

```
  public Tidspunkt( int timer, int minutter, int sekunder ) {
    set( timer, minutter, sekunder );
  }

  ...
}
```

Main.java

```
public class Main {

  public static void main( String[] argv ) {
    Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );

    System.out.println( vorTid.toString() );
  }
}
```

```
[12:30:00]
```

4.2.4.1 Overloading af konstruktorer

Man kan udemærket have flere konstruktorer, og det er mere reglen end undtagelsen.

Det gøres ved at overloade konstruktorer på samme måde som det gøres med metoder.

Lad os lave endnu en konstruktor, der kan bruges når man ikke ønsker at angive et antal timer, minutter og sekunder, men i stedet ønsker at tidspunktet skal initialiseres til 0, dvs. 0:00:00:

Source 19: Overloading med konstruktor, der ikke tager nogen parameter

(Tidspunkt.java)

```
public class Tidspunkt {
  ...

  private int totalSekunder;

  public Tidspunkt() {
    totalSekunder = 0;
  }

  public Tidspunkt( int timer, int minutter, int sekunder ) {
    set( timer, minutter, sekunder );
  }

  ...
}
```

Main.java

```
public class Main {

  public static void main( String[] argv ) {
    Tidspunkt vorTid = new Tidspunkt();

    System.out.println( vorTid.toString() );
  }
}
```

```
[0:00:00]
```

Ligesom metoder

Overloading fungerer efter helt de samme mekanismer som overloading af metoder, idet der vælges den konstruktor som matcher de aktuelle parametres antal og typer.

Man bemærker at **totalSekunder** eksplicit bliver sat til 0, selvom det ellers sker automatisk. Det er gjort for at

tydeliggøre at det er *hensigten* - for at gøre programmet lettere at læse/forstå. Som tidligere nævnt øger det læsbarheden at man initialiserer alle instans-variable i konstruktoren, i stedet for at falde tilbage på den automatiske initialisering til 0.

Kode-redundans

Hvis man sammenligner de to konstruktorer observer man en vis lighed. De sætter begge **totalSekunder** til en værdi. Konstruktoren med de tre parametre gør det indirekte ved at kalde **set**-metoden, mens konstruktoren uden parametre gør det direkte ved assignment. Selv om ligheden er meget begrænset, vil vi bruge den som et eksempel på hvordan man kan fjerne kode-redundans - et fænomen der ofte er mere omfattende i konstruktorer end det ses her.

I eksemplet med klassen **`Heltal`** fjernede vi redundansen ved at lade konstruktoren kalde **set**-metoden. Noget sådant kunne vi også prøve her:

Source 20: Konstruktor der kalder set-metoden

(Tidspunkt.java)

```
1  public Tidspunkt() {
2    set( 0, 0, 0 );
3  }
```

Problemet er blot, at var kode-redundansen lidt søgt i vores tidligere eksempel, så er den nu direkte udtalt, selvom det skal indrømmes at den stadig er beskeden i omfang.

For helt at fjerne redundansen skal vi have den ene konstruktor til at kalde den anden. Kan man det? Ja!

4.2.4.2 Konstruktorer der kalder konstruktorer

Det gøres ved noget der *ligner* et metodekald:

Source 21: Kald fra konstruktor til konstruktor

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public Tidspunkt() {
5      this( 0, 0, 0 );
6    }
7
8    public Tidspunkt( int timer, int minutter, int sekunder ) {
9      set( timer, minutter, sekunder );
10   }
11
12   ...
13 }
```

`this`-kald

Der er ikke nogen metode der hedder **`this`**, men kaldet betyder at den *konstruktor*, der passer med parametrene bliver kaldt. På den måde er det den anden konstruktor, der foretager initialiseringen ud fra de aktuelle parametre i kaldet.

Det skal bemærkes at det kun er konstruktorer, der kan kalde andre konstruktorer. Metoder kan f.eks. ikke kalde konstruktorer, hverken med **`this`** eller på anden måde. Til gengæld kan konstruktorer udemærket kalde metoder, som vi også tidligere har set.

Kode-redundansen er nu væk, og vil man senere foretage ændringer, skal man kun gøre det ét sted. En sådan ændring kunne f.eks. være at indføre en kontrol af at minutter og sekunder ikke overstiger 59. I den aktuelle situation vil man placere en sådan kontrol i **set**-metoden.

4.2.4.3 Default-konstruktor

Definition: Default-konstruktor

En default-konstruktor, er en konstruktor der ikke har nogen formelle parametre.

Før vi lavede den første konstruktor til klassen **`Tidspunkt`** kunne vi allerede have lavet instanser med f.eks.:

Source 22: Instantiering med default-konstruktor

```
Tidspunkt vorTid = new Tidspunkt();
```

Java laver selv en default-konstruktor

Her er der ikke anført nogen aktuelle parametre, og den konstruktor der bliver kaldt er default-konstruktoren. Men hvordan kan det så lade sig gøre før vi overhovedet har lavet nogen konstruktor? Det kan det, fordi Java selv laver en default-konstruktor til alle klasser. Denne konstruktor er på formen:

Source 23: Automatisk default-konstruktor

(Tidspunkt.java)

```
1  public Tidspunkt() {
2  }
```

En konstruktor der intet gør!

Kun hvis vi ikke selv laver nogen

Hvis vi selv laver en eller flere konstruktorer til en klasse, vil Java undlade at lave en default-konstruktor, uanset om vi selv har lavet en default-konstruktor.

4.2.4.4 Set-konstruktor

Definition: Set-konstruktor

En set-konstruktor [1], er en konstruktor der tager datakernen som parameter.

Direkte til datakernen

Set-konstruktoren repræsenterer en primitive form for initialisering af objektet; hvor man sender de data med, som uden videre skal placeres i datakernen. Navnet skyldes naturligvis den store lighed med set-metoder.

Om det er hensigtsmæssigt at lave en set-konstruktor afhænger ofte af hvor intuitiv den repræsentation man har valgt i data-kernen er. I vores klasse: **Tidpunkt**, må vi nok indrømme, at selvom total-sekunder gør implementationen simplere, så kan selve begrebet total-sekunder ikke forventes at være intuitivt for den der anvender klasse. Det er blot en intern data-repræsentation, der ikke bør have betydning for klassens interface. Selvom det derfor ikke er videre oplagt at lave en set-konstruktor til **Tidspunkt**-klassen, vil vi gøre det for at illustrere hvad en set-konstruktor er:

Source 24: Set-konstruktor

(Tidspunkt.java)

```
1  public Tidspunkt( int totalSekunder ) {
2    this.totalSekunder = totalSekunder;
3  }
```

Parameter-navne

Når man laver set-konstruktorer, der helt eller delvist tager datakernen som parameter, skal man vælge navne til de formelle parametre. Det kan man gøre på mange måder - vi vil se på de mest almindelige.

Begyndelses-bogstav

Man kan vælge at forkorte parametrenes navne til start-bogstavet for den tilsvarende instans-variabel, evt. flere start-bogstaver hvis flere variable starter med det samme bogstav. Det går ud over læsbarheden med én-bogstavs parameter-navne, men da en set-konstruktorer har et trivielt indhold er læsbarheden i forvejen så høj, at man ikke har noget behov for beskrivende parameternavne.

Underscore

Man ser også nogle gange at der sættes en underscore foran navnet på instans-variabelen og at man bruge det som parameternavn:

Source 25: Parameter-navne med underscore

(Tidspunkt.java)

```
1  public Tidspunkt( int _totalSekunder ) {
2    totalSekunder = _totalSekunder;
3  }
```

Vanskeligt at se fejl

Jeg har selv anvendt denne metode en kort overgang, men kasserede den fordi jeg synes det kan være vanskeligt, at se om man har husket underscore i assignments.

4.2.4.4.1 `this`-referencen

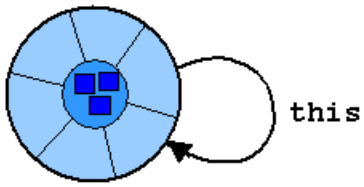
`this` har to betydninger

En tredie mulighed er at bruge en **`this`**-reference. Vi har allerede set ordet **`this`** brugt til kald mellem konstruktorer i samme klasse, men det har også en anden betydning.

Reference til sig selv

Enhver instans har en reference til sig selv, der hedder **`this`**. Vha. denne reference kan objektet referere til sine metoder og variable.

Figur 4: **`this`-*referencen***



Det er det man benytter i den tredie måde at navngive de formelle parametre til en set-konstruktor:

Source 26: Parameter med samme navn

(Tidspunkt.java)

```
1  public Tidspunkt( int totalSekunder ) {
2    this.totalSekunder = totalSekunder;
3  }
```

Parametre "skygger" for instans-variable

Ved at give parametre de samme navne som instans-variablene afskærer vi os umiddelbart fra at bruge instans-variablene. Når vi skriver **`totalSekunder`** i konstruktoren (eller hvis det havde været en metode), vil det være parameteren der bliver anvendt, og instans-variabelen med samme navn vil ikke være tilgængelig.

Her er det, at vi bruge **`this`**-referencen til at tilgå objektet selv. Vi gør det ved at anvende referencen som ved et metodekald, men nu er det en variabel, og ikke et kald, der optræder efter punktum.

Syntax 2: Tilgå instans-variabel direkte gennem reference

```
<reference>.<instans-variabel>
```

hvor <reference> er **`this`** og <instans-variabel> er f.eks. **`totalSekunder`**.

Source 27: Tilgang til instans-variabel med reference

```
this.totalSekunder
```

`this` som et visitkort

`this` kan også anvendes som en reference man kan give til andre objekter, så de får mulighed for at kommunikere med objektet.

4.2.4.5 Copy-konstruktor

MENU

Definition: Copy-konstruktor

En copy-konstruktor, er en konstruktor der tager en anden instans af samme klasse som parameter, og kopierer datakernen.

Samme tidspunkt som andet objekt

En sådan copy-konstruktor kunne være praktisk til vores klasse **`Tidspunkt`**. Så ville man ved instantiering kunne angive en instans af **`Tidspunkt`** (som allerede var lavet), som vores nye objekt skulle kopiere sin tilstand fra. Dvs. den skulle sætte sit tidspunkt til at være det samme.

Overføre reference

For at kunne gøre det må vi kunne overføre objekter som parametre. Det kan man ikke i Java, men man kan gøre noget der på sin vis er bedre: Man kan overføre referencer som parametre!

Assignment

I virkeligheden er det *nøjagtig* det samme der sker som ved parametre af primitive typer, f.eks. integers. Det der sker, er et **assignment** fra den aktuelle parameter til den formelle parameter. Lad os se skelettet af vores copy-konstruktor med en anvendelse:

Source 28: Skelettet til en copy-konstruktor

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    private int totalSekunder;
5
6    public Tidspunkt( Tidspunkt other ) {
7      ...
8    }
9
10   ...
11 }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt tid1 = new Tidspunkt( 12, 0, 0 );
5      Tidspunkt tid2 = new Tidspunkt( tid1 );
6    }
7  }
```

I anvendelsen laver vi først en instans **`tid1`** med tidspunktet 12:00:00, og dernæst laver vi endnu en instans; hvor vi overfører referencen til den første som parameter, for at **`tid2`** skal få samme tidspunkt.

Hvis vi tænker på det som et assignment, svarer overførslen til:

Pseudo 1: Paramter-overførsel svarer til assignment

```
Tidspunkt other = tid1
```

To referencer til samme objekt

Her er det netop et assignment mellem reference-variable, og både **`other`** og **`tid1`** vil nu referere til det samme objekt. På den måde har **`main`**-metoden fortalt konstruktoren "hvor" det objekt er som den skal kopiere sin datakerne fra befinder sig. Når vi skal tilgå tidspunktet i det andet objekt, gør vi det ved at bruge denne reference. Det færdige resultat bliver:

Source 29: Copy-konstruktor

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    private int totalSekunder;
3
4    public Tidspunkt( Tidspunkt other ) {
5      this.totalSekunder = other.totalSekunder;
6    }
7
```

```
  ...
}
```

Main.java

```
public class Main {

  public static void main( String[] argv ) {
    Tidspunkt tid1 = new Tidspunkt( 12, 0, 0 );
    Tidspunkt tid2 = new Tidspunkt( tid1 );

    System.out.println( tid1 );
    System.out.println( tid2 );
  }
}
```

```
[12:00:00]
[12:00:00]
```

Der er her to ting der kræver en forklaring.

"Friends"

Den første er brudt på indkapslingen. Den ene instans af **`Tidspunkt`** læser direkte fra instans-variabelen i den anden instans af **`Tidspunkt`**. At det kan lade sig gøre skyldes et fænomen som kaldes "**friends**". To instanser er friends hvis de er af samme klasse, og venner deler jo! Mellem friends er alt **`public`**, uanset hvad der ellers måtte være angivet i klassen.

Bryde indkapsling

Det forklarer hvorfor det er muligt - men brud på indkapslingen!? Der er ganske unike situationer; hvor (jeg mener) det er acceptabelt at man bryder indkapslingen - copy-konstruktorer er én af den. Det skyldes en række egenskaber ved en copy-konstruktor:

> En copy-konstruktor læser kun fra den anden instans, den *ændrer ikke noget*.
>
> Alt foregår *inden for samme klasse*, så navne på instans-variable er ikke fremmede. Skal disse navne f.eks. ændres, spreder ændringerne sig ikke uden for klassen.

Grunden til at man så vidt muligt skal holde sig til streng objektorientering, er for at opnå de foredele der er ved objektorientering og undgå de problemer man skaber når man ikke bruger det. Det er derfor argumentationen ovenfor tager udgangspunkt i de fordele og ulemper der spiller ind, i relation til det objektorienterede.

`println` er overloaded

Den anden ting, der kræver en forklaring i eksemplet ovenfor, er det manglende kald af **`toString`** i **`System.out.println`**. Metoden **`println`** er en instans-metode, men vi kalder den på en speciel måde som vi skal se nærmere på i forbindelse med **`static`** i et senere kapitel. Metoden er overloaded med et utal af forskellige type (se evt. afsnittet "PrintStream" i kapitlet "Streams"). To af disse typer er **`String`** og **`Object`**. Når vi tidligere har kaldt **`toString`**, har det været den version af **`println`** der tager en **`String`** som blev kaldt og den har udskrevet den tekststreng der beskrev objektet. Når man i stedet undlader **`toString`**, er det den version der tager et **`Object`** som bliver brugt (vi skal senere i kapitlet "Nedarvning" se hvorfor dette sker). Denne version af **`println`** kalder **`toString`** på ethvert objekt den modtager som parameter, og vi opnår derved samme effekt som hvis vi selv kaldte **`toString`**.

Source 30: println overloaded

(Tidspunkt.java)

```
public void println( Object obj ) {
  println( obj.toString() );
}
```

Vi vil derfor normalt ikke selv kalde **`toString`**, da det forkorter notationen, at udnytte denne overloading.

I Java ligger det fast, at parameter-overfører man en primitiv type bliver værdien altid kopieret over i den metode man kalder, og parameter-overfører man et objekt bliver det altid en reference der kopieres over til den metode man kalder. Som nævnt er det, det samme der sker - et **assignment** fra en aktuel parameter til en formel parameter.

MENU

Pass by ...

I f.eks. C++ ligger det ikke fast, og man kan efter eget valg f.eks. overføre en integer vha. en reference, eller et objekt som en kopi. I den forbindelse har man to betegnelser, der er gode at kende: **Pass by value** og **pass by reference**. Pass by value, er at man **kopierer** data over i metodens formelle parametre, mens pass by reference, er at man i stedet **overfører en reference** til data.

Hvis man derfor skal beskrive mulighederne i Java med disse termer, kan det gøres med:

> Alle primitive typer overføres med pass by value.
>
> Alle ikke-primitive typer overføres med pass by reference.

Med alle ikke-primitive typer, menes objekter og arrays.

Altid som assignment

Disse to forskellige former for parameter-overførsel kan i starten forvirre, men husker man én ting, er det i virkeligheden enkelt at forstå: **I Java fungerer det *altid som assignment!***

Pass by reference findes ikke i Java

Det er samme enkle huskeregel, der i virkeligheden viser *at pass by reference ikke findes i Java*. Når vi overfører en reference til et objekt, angiver vi *selv* referencen som parameter. Det betyder at al parameteroverførsel i Java reelt er pass by value.

4.2.5 Metoder

Vi har allerede lavet en **`toString`**-metode, men hvad skal **`Tidspunkt`** ellers have af metoder?

4.2.5.1 Ændringer

Tidspunktet skal kunne ændres.

Hvis vi f.eks. ønskede at stille tidspunktet et vist antal timer frem, kunne det være nyttigt med følgende metode:

(Tidspunkt.java)

```
1  public class Tidspunkt {
2
3    ...
4
5    public void addTimer( int timer ) {
6      totalSekunder += timer * TIME;
7    }
8  }
```

Source 31: Ændring af timer

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );
5      vorTid.addTimer( 1 );
6      System.out.println( vorTid );
7    }
8  }
```

```
[13:30:00]
```

Tilsvarende kunne vi lave to andre metoder **`addMinutter`** og **`addSekunder`** til at forøge tidspunktet med et vist antal af disse enheder.

Source 32: Ændring af minutter og sekunder

(Tidspunkt.java)

```
1  public class Tidspunkt {
```

```
2
3     ...
4
5     public void addMinutter( int minutter ) {
6       totalSekunder += minutter * MINUT;
7     }
8
9     public void addSekunder( int sekunder ) {
10      totalSekunder += sekunder;
11    }
12  }
```

Metoderne er udemærkede, men de kan laves med én, hvis man slår dem sammen.

Source 33: Ændring med alle tre

(Tidspunkt.java)

```
1  public class Tidspunkt {
2
3    ...
4
5    public void add( int timer, int minutter, int sekunder ) {
6      totalSekunder += getTotalSekunder( timer, minutter, sekunder );
7    }
8  }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );
5
6      vorTid.add( 1, 0, 0 );
7
8      System.out.println( vorTid );
9    }
10 }
```

```
[13:30:00]
```

Mere objekt-orienteret

Når vi lægger f.eks. 1 time, 0 minutter og 0 sekunder til et tidspunkt, er det i virkeligheden et helt **`Tidspunkt`**-objekt vi angiver. Derfor er det mere objektorienteret, at lade metoden tage en anden instans af **`Tidspunkt`** som parameter:

Source 34: Ændring angivet ved tidspunkt

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public void add( Tidspunkt other ) {
5      this.totalSekunder += other.totalSekunder;
6    }
7  }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );
5      Tidspunkt enTime = new Tidspunkt( 1, 0, 0 );
6
7      vorTid.add( enTime );
8
9      System.out.println( vorTid );
10   }
```

```
11 }
```

```
[13:30:00]
```

Man bemærker at vi igen udnytter, at de to instanser af **`Tidspunkt`** er friends fordi de er instanser af samme klasse. Igen er det i orden, fordi evt. ændringer ikke spreder sig ud af klassen, da det er den samme klasse det hele foregår i.

Bemærk ligeledes linien:

Source 35: Holdbar linie

(Main.java)

```
vorTid.add( enTime );
```

Holdbarhed

I selve dette kald står der nu ikke noget om hvor mange timer, minutter og sekunder der er tale om. Selve kaldet er blevet mere abstrakt formuleret og skulle det senere vise sig at vi havde lavet en tastefejl i linien hvor vi instantierer **`enTime`**, så den f.eks. var blevet 1:01:00 i stedet, så behøver vi ikke rette i kaldet af **`add`**-metoden. Man ser hvorledes linien er blevet mere holdbar. Når man programmerer gælder det derfor om at lave så mange linier så holdbare som muligt, så ændringer kun skal laves få steder, mens resten af programmet automatisk "retter sig ind".

Vi vil lave en enkelt ændring mere, der betyder at vi mister den holdbare linie ovenfor, men til gengæld sker det ved at vi sammenskriver den med en anden:

Source 36: Instantiering i kaldet

(Tidspunkt.java)

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );
5
6      vorTid.add( new Tidspunkt( 1, 0, 0 ) );
7
8      System.out.println( vorTid );
9    }
10 }
```

Direkte overførsel

Det er første gang vi anvender en instantiering som aktuel parameter, men det er ganske uproblematisk. Vi kan se præcist hvad der sker; hvis vi anvender vores enkle regel vedrørende parameter-overførsel: **"Det er fuldstændig ligesom et assignment"**:

Pseudo 2: Parameter-overførsel er som assignment

```
Tidspunkt other = new Tidspunkt( 1, 0, 0 )
```

Mister objektet

Nu er det enkelt at se, at den formelle parameter er den eneste reference der får fat i objektet. I særdeleshed er der ingen reference fra det sted hvor der kaldes, som har mulighed for at få fat i instansen, men det er heller ikke nødvendigt for vores anvendelse.

Hvis vi ønsker at stille et tidspunkt en time tilbage kan vi gøre det med:

Source 37: "Hack"

```
vorTid.add( new Tidspunkt( -1, 0, 0 ) );
```

Modstridende signaler

Hvad med læsbarheden? I linien er der modstridende signaler. **`add`** indikerer at vi lægger noget til og **`-1`** indikerer at vi trækker noget fra. I virkeligheden er det på grænset til et hack, at vi bruger **`add`** til at stille tidspunktet tilbage. Vi vil derfor foretrække at lave en ny metode til dette formål:

Source 38: Speciel metode til substraktion

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public void sub( Tidspunkt other ) {
5      this.totalSekunder -= other.totalSekunder;
6    }
7  }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt vorTid = new Tidspunkt( 12, 30, 0 );
5
6      vorTid.sub( new Tidspunkt( 1, 0, 0 ) );
7
8      System.out.println( vorTid );
9    }
10 }
```

```
[11:30:00]
```

sub-metoden er en enkel omskrivning af **add**.

4.2.4.2 Sammenligninger

Vi skal også kunne sammenligne tidspunkter. Normalt opdeler man sammenligning i to grupper: lighed og ulighed. Vi vil først se på lighed.

Efter de overvejelser vi gjorde os i forrige afsnit, har vi allerede lært nok, til med det samme at kunne rynke lidt på næsen af følgende forslag til en metoder der sammenligner med henblik på lighed:

Source 39: Primitiv equals

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public boolean equals( int timer, int minutter, int sekunder ) {
5      return totalSekunder == getTotalSekunder( timer, minutter, sekunder );
6    }
7  }
```

Det er naturligvis de ikke objektorienterede parametre, der er svagheden i denne løsning.

Løsningen er i stedet den mere objektorienterede:

Source 40: Objektorienteret equals

(Tidspunkt.java)

```
1  public boolean equals( Tidspunkt other ) {
2    return this.totalSekunder == other.totalSekunder;
3  }
```

Lad os dernæst implementere sammenligning ved ulighed. Umiddelbart lyder det som to sider af den samme sag. Måske kan det endog lyde lidt mærkeligt at det er to forskellige metoder, men i nogle sammenhænge er sammenligning af lighed betydelig hurtigere end sammenligning mht. ulighed, derfor er det normalt at man laver to forskellige metoder, og vi vil gøre det samme.

MENU

Metoden der ser på ulighed kaldes normalt **`compareTo`**, og returner et tal der giver "resultatet" af sammenligningen. Man giver det ene objekt til det andet og siger: "Sammenlign dig med det her objekt!". Svaret er et af tre mulige:

Værdi	Betydning
-1	"Den er større end mig"
0	"Vi er lige store"
1	"Jeg er størst"

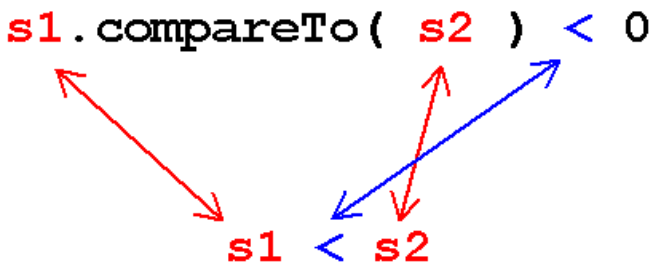
Tabel 1:
Retur-værdier fra **`compareTo`**

Trække fra

Enkelt nok, men umiddelbart lidt mærkeligt med værdierne. Man skal forstå det som om det objekt man sender med som parameter "trækkes fra", og det der returneres er et udtryk for fortegnet på resultatet.

Hvis vi f.eks. beder 5 om at lave en **`compareTo`** med 8 (hvis vi et øjeblik ser bort fra, at 5 og 8 ikke er objekter), vil den svare tilbage -1 ("Den er større end mig"). Det vil den, fordi den udfører regnestykket: "den selv" minus "den anden", altså 5 - 8 = -3.

Når man anvender metoden er det bekvemt at bruge følgende standard-udtryk:



Figur 5:
Anvendelse af **`compareTo`**

Fordelen ved dette udtryk, er at rækkefølgen af de to operander er den samme som i det ækvivalente udtryk og at ulighedstegnet ligeledes er det samme.

Analogt for andre uligheder

Ovenfor er anvendt **<**, men man kan gøre det samme for andre uligheder og ligheder, blot ved at udskifte **<** med det ønskede.

Når vi implementerer metoden begynder vi ikke at trække sekunder fra sekunder, minutter fra minutter osv. Det er meget enklere at omregne begge tidspunkter til sekunder og sammenligne dem.

Lad os se metoden:

Source 41: compareTo

(Tidspunkt.java)

```
1  public class Tidspunkt {
2    ...
3
4    public int compareTo( Tidspunkt other ) {
5      int dif = this.totalSekunder - other.totalSekunder;
6
7      if ( dif < 0 )
8        return -1;
9      else if ( dif > 0 )
10       return 1;
11     else
12       return 0;
13   }
14 }
```

Pænere med service-metode

De nederste if-sætninger i **compareTo** omregner til de tre værdier. Denne afbildning kalder man signum-funktionen og vi vil vælge at lave den som en service-metode[1], selvom den ikke anvendes andre steder i vores klasse:

Source 42: Med signum service-metode

(Tidspunkt.java)

```
1  public int compareTo( Tidspunkt other ) {
2    return signum( this.totalSekunder - other.totalSekunder );
3  }
4
5  private int signum( int x ) {
6    if ( x != 0 )
7      return x / Math.abs( x );
8    else
9      return x; // eller 0
10 }
```

Vi indfører her et begreb: "fortegns-tal" (signum: **sig**n-**num**ber), og dette øger abstraktionsniveauet i implementationen af **`compareTo`**-metoden. Hvis man ellers ved hvad **`signum`** betyder, er **`compareTo`** nu mere læsbar.

4.2.5 Testanvendelse

Vi afslutter med en samlet testanvendelse af klassen: **`Tidspunkt`**:

Source 43: Testanvendelse af Tidspunkt

Main.java

```
1  class Main {
2
3    public static void main( String[] argv ) {
4      Tidspunkt t1 = new Tidspunkt( 12, 30, 0 );
5      Tidspunkt t2 = new Tidspunkt( t1 );
6
7      System.out.println( "t1 lig t2: " + t1.equals( t2 ) );
8
9      t1.add( new Tidspunkt( 1, 0, 0 ) );
10
11     System.out.println( "t1: " + t1 );
12     System.out.println( "t1 compare to t2: " + t2.compareTo( t1 ) );
13
14     t1.setMinutter( 55 );
15
16     System.out.println( "t1: " + t1 );
17     System.out.println( "t1's timer: " + t1.getTimer() );
18   }
19 }
```

```
1  t1 lig t2: true
2  t1: [13:30:00]
3  t1 compare to t2: -1
4  t1: [13:55:00]
5  t1's timer: 13
```

5. Arrays af objekter

Referencer

Man kan lave et array af heltal, men kan man også lave et array af objekter? Set i relation til afsnittets lovende overskrift, er det måske overraskende, at det præcise svare faktisk er: Nej! I stedet kan man lave et **array af referencer** til objekter (man betegner dem dog alligevel som "arrays af objekter").

Erklæring

Hvis vi f.eks. ønskede at organisere ti **`Tidspunkt`**-objekter i et array, kunne vi erklære arrayet på følgende måde:

Source 44: Allokering af

```
Tidspunkt[] tider = new Tidspunkt[10];
```

array af referencer

Samme syntaks

Ved erklæringen af arrayet bruger vi den samme syntaks som for primitive typer, idet vi placerer klassens navn hvor typen skulle være. Arrayet bliver et array af referencer til **`Tidspunkt`**-objekter; hvilket stemmer over ens med at

Source 45: Erklæring af reference

```
Tidspunkt vorTid;
```

erklærer én reference til ét **`Tidspunkt`**-objekt.

Refererer ikke til noget

Arrayets referencer refererer endnu ikke til noget, vi har ingen instanser lavet endnu. Når en reference ikke refererer til noget har den en speciel værdi. Denne værdi kan vi f.eks. få frem ved at udskrive det første element i arrayet, der er en reference.

Source 46: Anvendelse af reference fra array

```
1  Tidspunkt[] tider = new Tidspunkt[10];
2
3  System.out.println( tider[0] );
```

```
null
```

`null`

Værdien hedder **`null`**, og man kan selv anvende den. Hvis vi f.eks. ville initialisere alle referencerne i arrayet til **`null`** (hvilket jo er ganske overflødigt), så kunne det gøres ved følgende:

Source 47: Initialisering med null

```
1  for ( int i=0; i<tider.length; i++ )
2    tider[i] = null;
```

Hvis vi i stedet ville initialisere arrayet, så hver reference kom til at referere til en instans af **`Tidspunkt`** kunne vi i stedet gøre følgende:

Source 48: Referencer sættes til instanser

```
1  for ( int i=0; i<tider.length; i++ )
2    tider[i] = new Tidspunkt( 2*i, 0, 0 );
```

Vi sætter her de enkelte instanser til et antal timer, der svarer til den indgang hvor de er placeret gange to. Det er naturligvis kun for eksemplets skyld. En efterfølgende udskrift af samtlige instanser bliver derved også lidt mindre triviel, end hvis vi blot havde brugt default-konstruktoren:

Source 49: Udskriving af objekter i array

```
1  for ( int i=0; i<tider.length; i++ )
2    System.out.println( tider[i] );
```

```
1  [0:00:00]
2  [2:00:00]
3  [4:00:00]
4  [6:00:00]
5  [8:00:00]
6  [10:00:00]
7  [12:00:00]
8  [14:00:00]
9  [16:00:00]
10 [18:00:00]
```

Hvis vi vil kommunikere med de enkelte objekter som arrayets elementer refererer til, gøres det på normal vis:

Source 50: Kald

af metode på objekt i array

```
1  tider[3].add( tider[7] );
2
3  System.out.println( tider[3] );
```

```
[20:00:00]
```

Intet nyt under solen

Som det ses er der ikke rigtig noget nyt i et array af referencer, det er som et array af enhver anden type. Konstruktionen med kantede paranteser binder, som altid, lige så stærkt som et variabelnavn.

6. Garbage Collector

I dette sidste afsnit skal vi se på noget af det mest specielle ved objekter, et hjørne hvor man ikke kommer så tit, men som man bør kende.

Hvad sker der når objekter dør? Kommer de ud på de evige bit-marker? Nej, de bliver taget af garbage collectoren!

Rydder op efter os

Garbage collectoren er en del af Java's runtime-system - den virtuelle maskine. Garbage collectoren søger med mellemrum efter objekter, der ikke findes referencer til. Når den finder sådanne, frigives den del af lageret der ellers var allokeret til dem. Det betyder at man kan bruge objekter ud fra "brug og smid væk" mentaliteten - garbage collectoren skal nok komme og rydde op efter os.

`finalize` kaldes altid før...

Vi skal i senere kapitler se, at objekter kan have allokeret andre resourcer end lager, f.eks. netværksforbindelser, filer og andet. Derfor kan det være hensigtsmæssigt at objekter får en sidste chance for at "rydde op efter sig" inden de bliver taget af garbage collectoren. Man har mulighed for at lave en metode, der hedder **`finalize`**, som garbage collectoren altid vil kalde lige før den "gør det onde" ved objektet.

Et simpelt eksempel kunne være:

C.java

```
1  public class C {
2    public void finalize() {
3      System.out.println( "signing off..." );
4    }
5  }
```

Main.java

```
1  public class Main {
2
3    public static void main( String[] argv ) {
4      C c;
5
6      while ( true )
7        c = new C();
8    }
9  }
```

Source 51: Uendelig løkke, der laver reference-fri objekter

```
1  Signing off...
2  Signing off...
3  Signing off...
4  ...
```

Programmet kører i en uendelig løkke, der hele tiden sætter referencen til en ny instans. På den måde vil alle tidligere instanser være uden reference og kandiderer til at blive garbage collected.

Garbage collectoren ligger i dvale

Hvis man kører eksemplet vil man observere at det varer lidt før garbage collectoren begynder at samle objekterne ind. Garbage collectoren kører ikke hele tiden. Den vågner i yderste konsekvens først op til dåd når lagret er ved at være helt fyldt med objekter.

MENU

Man har mulighed for at appelere til garbage collectoren om at rydde op, men i hvilken udstrækning den vælger at gøre det, er op til den selv - garbage collectoren har man aldrig styr på! Man gør det ved kaldet:

Source 52: Direkte kald af garbage collectoren

```
System.gc();
```

Brug ikke `finalize`-metoden

Det er derfor i realiteten umuligt at sige hvornår **`finalize`**-metoden bliver kaldt, og man bør derfor helt undlade at anvende den. I stedet bør man lave en "rydde op"-metode, som man explicit kalder når man er færdig med at anvende objektet; hvis objektet har resourcer som det er væsentligt at få frigivet med det samme.

Verbose mode

Hvis man anvender Suns JDK kan man få garbage collectoren til at fortælle om sin "gøren og laden" ved at bruge option **-verbosegc.**

Støjer meget

Det kan være interessant at se hvad den skriver, men normalt er det ikke en option man har slået til, da den "støjer" meget.

Epilog

Læseren har i dette kapitel været udsat for en streng objektorienteret indoktrinering. Måske for streng efter nogens mening. Jeg foretrækker dog at starte kompromisløst, så kan man senere bløde lidt op på principperne, når man ved hvad man gør, og specielt hvad det koster!

---

fodnoter:

1 I virkeligheden behøver vi ikke være så omhyggelige, at vi laver en endelig normalisering til -1 og 1, med **`signum`**-metoden.

F.eks. foretager **`String`**'s **`compareTo`** ikke denne normalisering:

Source 53: String's compareTo

```
1  String s1 = "Her";
2  String s2 = "Herning";
3  String s3 = "Hyrning";
4
5  System.out.println( s1.compareTo( s2 ) );
6  System.out.println( s1.compareTo( s3 ) );
```

```
1  -4
2  -20
```

`"Her"` og **`"Herning"`** er ens så langt som **`"Her"`** rækker, og metoden returnerer **`-4`**. **`4`** fordi det er forskellen i længde mellem de to tekststrenge, og *minus* fordi den længste regnes som den største.

I den anden sammenligning er 20 forskellen mellem **`'e'`** og **`'y'`** i Unicode, og *minus* fordi **`'y'`** har den største værdi i Uncode (dette er ganske enkelt **`e`**'s Unicode minus **`y`**'s Unicode).

Repetitionsspørgsmål

1 Hvad svarer til en skabelon i Java?

2 Hvordan implementeres datakernen?

3 Hvordan implementeres operationer?

4 Hvad betyder klassens virkefelt for metoderne i en klasse?

MENU

5 Hvad betyder **`public`**?

6 Hvad betyder **`private`**?

7 Hvorfor undergraver set-/get-metoder objektorientering?

8 Hvad skyldes ofte et stort behov for set-/get-metoder?

9 Hvorfor er det bedre at bruge set-/get-metoder i stedet for at gøre instans-variable **`public`**?

10 Hvilket formål tjener klassen med **`main`**?

11 I hvilken forbindelse har vi tidligere brugt reference-variable?

12 Hvordan sender man en request til et objekt?

13 Hvor mange referencer kan man have til ét objekt?

14 Hedder objekter noget?

15 Hvad er formålet med en konstruktor?

16 Hvad specielt er der ved en konstruktor i forhold til en metode?

17 Hvad er kode-redundans?

18 Hvorfor er kode-redundans en ulempe?

19 Hvilken opfattelse af et objekt giver et dårligt objektorienteret design?

20 Hvad skal man vide om et objekt for at kunne lave et godt design?

21 Hvad er **`toString`**'s primære formål?

22 Hvordan fungerer overloading af konstruktorer?

23 Hvordan kan en konstruktor kalde en anden konstruktor?

24 Hvad er en default-konstruktor?

25 Hvornår laver Java selv en default-konstruktor?

26 Hvad er en set-konstruktor?

27 Hvordan kan man navngive set-konstruktorens formelle parametre?

28 Hvad er **`this`**-referencen og hvad kan den bruges til?

29 Hvad er en copy-konstruktor?

30 Hvordan overføres et objekt som parameter?

31 Hvad er friends?

32 Hvorfor må man godt bryde indkapslingen i en copy-konstruktor?

33 Hvorfor behøver man ikke bruge **`toString`** i forbindelse med **`println`**?

34 Hvad er pass by value og pass by reference?

35 Findes der pass by reference i Java?

36 Hvorfor er det mere objektorienteret at overføre en instans af en klasse som parameter, end at overføre hvad der svarer til datakernen? (se evt. diskussionen om **`add`**)

37 Hvad vil det sige nogle sætninger er mere holdbare end andre?

38 Parameteroverførsel er simpelt at forstå, hvis man husker én grundregel - hvilken?

39 Hvad er en service-metode?

40 Hvordan skal retur-værdierne fra **`compareTo`** fortolkes?

41 Hvordan anvender man **`compareTo`**?

42 Kan man lave arrays af objekter?

43 Hvad er den syntaktiske forskel på arrays af objekter og arrays af f.eks. heltal?

44 Hvad betyder **`null`**?

45 Hvad er garbage collectorens opgave?

46 Hvornår kaldes **`finalize`**-metoden?

Svar på repetitionsspørgsmål

1 En klasse.

2 Som instans-variable.

3 Som instans-metoder.

4 At de kan tilgå instans-variable og kalde hinanden.

5 At noget kan tilgås udefra objektet.

6 At noget kun er tilgængeligt indefra objektet.

7 Fordi de skaber indirekte adgang til objektets instans-variable.

8 Mangelfuldt objektorienteret design.

9 Fordi de trods alt giver mulighed for senere at lave en kontrol af adgangen, det gør **`public`** ikke.

10 Den skal kunne sætte programmet igang - ikke andet.

11 I forbindelse med arrays.

12 Man kalder en metode vha. en reference til objektet.

13 Vilkårligt mange.

14 Nej, objekter hedder ikke noget, men der er sjældent noget forgjort ved, at man kalder dem det referencen hedder

15 At initialisere objektet - at gøre det klar i forbindelse med instantieringen.

16 En konstruktor kan *kun* "kaldes" i forbindelse med instantieringen og kun af andre konstruktorer.

17 At kode der gør det samme forekommer flere gange i kildeteksten.

Fordi ændringer i redundant kode skal foretages alle de steder det findes; hvilket giver større mulighed for fejl.

18

19 At objektet er passivt og det er vores opgave at gøre noget ved det.

20 Man skal kende dets rolle i de objektsystemer det skal indgå.

21 At kan se tilstanden under test.

22 På samme måde som for metoder.

23 Den laver noget der ligner et metodekald, blot bruger man "metodenavnet" **`this`**.

24 En konstruktor der ikke tager nogen parametre.

25 Den laver kun en default-konstruktor hvis vi ikke laver nogen konstruktorer overhovedet.

26 En konstruktor, der tager datakernen som parameter.

27 Der anvendes normalt en af tre måder: Forbogstavs-, underscore eller **`this`**-metoden.

28 Det er en reference til objektet selv. Det bruges normalt til at sende andre objekter en reference, så de kan sende request til én.

29 En konstruktor, der tager en anden instans af samme klasse som én selv som parameter.

30 Vha. en reference til objektet.

31 Objekter er friends hvis de kan tilgå hinandens private variable og metoder. To instanser af samme klasse er friends.

32 Fordi det sker indenfor samme klasse og fordi der kun læses fra de instans-variable man bryder ind til.

33 Fordi **`println`** er overloaded med en version der tager et objekt som parameter, og som selv kalder **`toString`** for os.

34 Det er betegnelser der anvendes i forbindelse parameteroverførsel. Pass by value kopierer værdien af variablen, mens pass by reference kun sender en reference til variablen med over.

35 Rent teknisk gør der ikke. Det skyldes at man *selv* angiver en reference i forbindelse med overførsel af objekter. Det er dog godt at kende begrebet pass by reference fordi det er relevant i den sammenhæng.

36 Fordi objektorienteret programmering i sin reneste form kun beskæftiger sig med objekter.

37 At de er mere holdbare overfor ændringer andre steder i kildeteksten.

38 Det er fuldstændig lige som et assignment.

39 En metode der kun er til for de andre metoder i objektet. Man gør derfor service-metoder **`private`** for at indkapsle dem.

40 -1 betyder "den er større end mig", 0 betyder "vi er lige store" og 1 betyder "jeg er størst".

41 Ved at lave et boolsk udtryk hvor man sammenligner med 0. 0 skal står sidst og uligheden kan dermed fortolkes som stod dem mellem de to objekter.

42 Nej, rent teknisk kan man kun lave arrays af referencer til objekter, men kalder dem alligevel arrays til objekter.

43 Ingen. Man anfører blot klassenavn i stedet for typenavn.

44 At en reference ikke refererer til noget.

45 At fjerne de objekter der ikke bruges. Den gør det ved at fjerne dem der ikke er nogen referencer til.

46 Den kaldes af garbage collectoren lige før den fjerner objektet.

MENU