Kursus 02199: Programmering
Mere om metoder, klasser og objekter: afsnit 4.5, 5.1-5.4

Anne Haxthausen, IMM, DTU

1. Referencer (afsnit 5.1)
   - **null** referencen
   - **this** referencen
   - objekt aliasing
     (a) ved tildelingssætniger
     (b) ved metode-kald (hvor parametrene er objekter)
     (c) specielle implikationer ved aliasing
     (d) lighed for referencer versus lighed for objekter
2. Modifieren **static** (afsnit 5.2)
3. Indre klasser (læs selv afsnit 5.3)
4. Metode overloading og overload resolution baseret på signaturer (afsnit 4.5)
5. Interfaces (afsnit 5.4)

Time eksemplet fra forelæsning 5

```
class Time {
  private int hours, min; //timer og minutter siden midnat

  public Time(int h, int m) {hours = h; min = m;}

  public int gethours() {return hours ;}

  public int getmin() { return min; }

  public String toString() { return hours + "." + min; }

  public void passtime(int m) {
    int totalmin = 60 * hours + min + m;
    hours = (totalmin / 60) % 24; min = totalmin % 60;  }
}
```

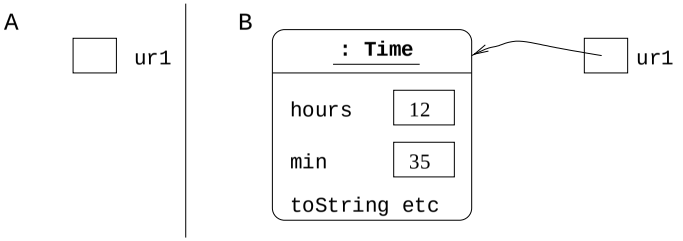

null referencen

En variable af en klasse type indeholder en reference/henvisning til (dvs. *adressen* på) et objekt af den givne klasse.

Inden variablen bliver initialiseret, vil den indeholde den specielle reference værdi **null**, der angiver, at den (endnu) ikke henviser til et objekt.

Eksempel

```
Time ur1;                //A: efter erklaering er  ur1 == null
ur1 = new Time(12, 35); //B: efter initialisering er  ur1 != null
```



this referencen

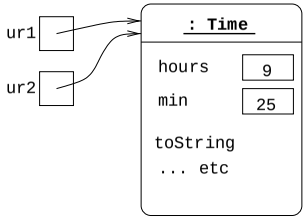
```
class Time {
  private int hours, min; //min #1

  ...

  public void passtime(int min) { //min #2  (skygger for min #1)
    int totalmin = 60 * this.hours + this.min + min;
    this.hours = (totalmin / 60) % 24; this.min = totalmin % 60;  }
  }
}
```

Objekt aliasing ved tildeling

To variable kan henvise til det samme objekt – de siges at være *aliaser*.

Eksempel

```
Time ur1 = new Time(9, 25);
Time ur2 = ur1;
```



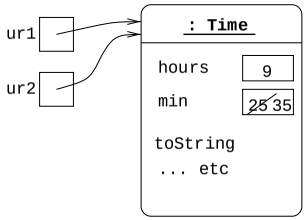
Specielle implikationer ved aliasing

Pas på

Hvis man bruger aliaser, så betyder en ændring af et objekts tilstand en ændring for *alle* de referencer, der peger på det objekt.

Eksempel

```
Time ur1 = new Time(9, 25);
Time ur2 = ur1;
ur1.passtime(10);
```



Lighed for referencer versus lighed for objekter

Reference lighed: `==`

```
ur1 == ur2
```

afgører om `ur1` og `ur2` refererer til det samme objekt — d.v.s. er aliaser.

Objekt lighed: `equals`

I `Time` klassen kan vi definere vores egen lighed som en metode ved navn `equals`:

```
boolean equals(Time ur2) {
  return this.min == ur2.min &&  this.hours == ur2.hours;
}
```

herved kan to objekter sammenlignes således:

```
ur1.equals(ur2)
```


Lighed: eksempler

```
public class TimeAliasing {
  public static void main(String[] args) {
    Time ur1, ur2;

    ur1 = new Time(9, 10);  ur2 = new Time(9, 10);
    udskrivtider(ur1, ur2);
    ur1.passtime(40);  udskrivtider(ur1, ur2);
    ur2 = ur1;  udskrivtider(ur1, ur2);
    ur1.passtime(40);  udskrivtider(ur1, ur2);
  }
  static void udskrivtider(Time ur1, Time ur2) {
    System.out.println("ur1 viser " + ur1);
    System.out.println("ur2 viser " + ur2);
    System.out.println("ur1 == ur2: " +  (ur1 == ur2));
    System.out.println("ur1.equals(ur2): " + ur1.equals(ur2) +"\n");
} }
```

Uddata fra TimeAliasing

```
ur1 viser 9.10
ur2 viser 9.10
ur1 == ur2: false
ur1.equals(ur2): true

ur1 viser 9.50
ur2 viser 9.10
ur1 == ur2: false
ur1.equals(ur2): false

ur1 viser 9.50
ur2 viser 9.50
ur1 == ur2: true
ur1.equals(ur2): true

ur1 viser 10.30
ur2 viser 10.30
ur1 == ur2: true
ur1.equals(ur2): true
```


Hvad sker der med variable, der bruges som aktuelle parametre til metoder?

Java bruger *call-by-value*, når der overføres værdier til en metode — svarer til kopiering af den *aktuelle* parameters værdi til den *formelle* parameter i metoden. Efter et kald af metoden har den aktuelle parameter (her: variabel) den samme værdi som før kaldet af metoden.

Betydning:

- når variable af klasse typer er argumenter til en metode, kopieres *referencen* til objektet, ikke objektet selv
- hvis metode kroppen ændrer den formelle parameter reference (sætter den til at pege på et nyt objekt), får det *ikke* betydning for argumentet (se eksempel1)
- hvis metode kroppen ændrer tilstanden af det objekt, den formelle parameter peger på, sker det samme for argumentet (se eksempel2)

Primitive værdier som argumenter til metoder: eksempel

Hvad udskriver følgende program?

```
public class Metodekald {
  public static void main(String[] args) {
    int a = 2;
    System.out.println("a er " + a);
    f(a);
    System.out.println("a er " + a);
  }

  static void f(int x) { x = x + 10; }
}
```


Objekter som argumenter til metoder: eksempel1

Hvad udskriver følgende program?

```
public class Metodekald1 {
  public static void main(String[] args) {
    Time a = new Time(9, 25);
    System.out.println("a er " + a);
    f(a);
    System.out.println("a er " + a);
  }

  static void f(Time x) { x = new Time(8, 17); }
}
```

Objekter som argumenter til metoder: eksempel2

Hvad udskriver følgende program?

```
public class Metodekald2 {
  public static void main(String[] args) {
    Time a = new Time(9, 25);
    System.out.println("a␣er␣" + a);
    f(a);
    System.out.println("a␣er␣" + a);
  }

  static void f(Time x) { x.passtime(10); }
}
```


Static felter

Vi har allerede set, at *metoder* kan erklæres **static**, så de kan bruges via klassenavnet uden at lave et objekt.

Det samme gælder *felterne* i en klasse. Da vil alle objekter, der er instanser af klassen, for dette felt have en fælles plads i lageret, ej hver deres kopi.

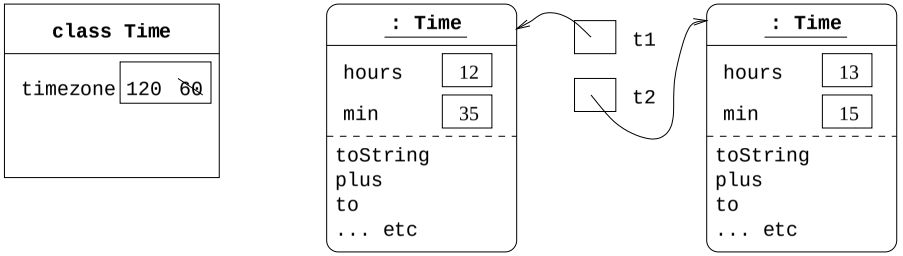
Eksempel

Et klassefelt `timezone` kunne bruges til at angive tidszonen fælles for alle tidspunkter.

```
class Time {
  static int timezone = 60; //min. øst for Greenwich
  ... alt andet som før ...
}
```


Static felter, eksempel fortsat

```
Time t1 = new Time(12, 35);
Time t2 = new Time(13, 15);

Time.timezone = 120;
```



Metode signaturer

Signaturen af en Java metode består af: metodenavnet, samt listen af parameter typer

Eksempler

`gethours()`

`passtime(int)`

`max(int, int)`

`is_ok(boolean)`

Grundbegreb: overloading

Når flere metoder erklæres med

- samme navn, men
- forskellig signatur

siges navnet at være *overloaded* (overlæst).

Eksempel i Java:

`Time` klassen kunne have haft to konstruktorer:

```
public Time(int h, int m) {hours = h; min = m;}
public Time(int h) {hours = h; min = 0;}
```

De har samme navn, men forskellige signaturer:

`Time(int, int)` hhv. `Time(int)`

Overload resolution

Hvis et overloaded navn kaldes, hvilken erklæret metode hører den så til?

Ved *overload resolution* forstås afgørelsen af dette.
I Java afgøres det ved at matche argument typerne med typerne i signaturen. Andre sprog har andre regler.

Eksempler i Java

`Time(12, 10)` hører til 1. erklæring

`Time(12)` hører til 2. erklæring

Interfaces

Hidtil har vi brugt begrebet *interface* om et objekts **public** konstanter og metoder. Dette stemmer overens med det mere formelle interface begreb i Java.

Definition: Et Java *interface* består af:

- en samling af public konstanter og/eller
- en række public *abstrakte* metoder, dvs. metoder uden kroppe

Bemærk: ingen konstruktorer, ingen variabel felter.

Definition: En klasse *implementerer* et interface ved at give metode implementeringer (kroppe) for *alle* de abstrakte metoder i interfacet, desuden evt. yderligere felter, metoder og konstruktorer.

Bemærk: et interface kan have flere implementeringer og en klasse kan implementere flere interfaces.

Interfaces: eksempel

```
interface TimeInterface {

   public int gethours();

   public int getmin();

   public String toString();

   public void passtime(int m);
}
```

Implementering af interfaces: eksempel 1

```
class Time1 implements TimeInterface {
  private int hours, min; //timer og minutter siden midnat

  public Time1(int h, int m) {hours = h; min = m;}

  public int gethours() {return hours ;}

  public int getmin() { return min; }

  public String toString() { return hours + "." + min; }

  public void passtime(int m) {
    int totalmin = 60 * hours + min + m;
    hours = (totalmin / 60) % 24; min = totalmin % 60;
  }

}
```


Implementering af interfaces: eksempel 2

```
class Time2 implements TimeInterface {
  private int min; //minutter siden midnat

  public Time2(int h, int m) {min = h * 60 + m;}

  public int gethours() {return ... ;}

  public int getmin() { return ... ; }

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

  public void passtime(int m) { min = ... ;  }
}
```


Hvad bruger man interfaces til (1)?

Et interface specificerer public metoder og konstanter, som en (eller flere) implementerende klasser skal tilbyde (som et minimum).

I progammeludvikling kan interfaces bruges til HVAD før HVORDAN princippet:

1. Først laves et interface: det er en specifikation af HVAD en klasse skal tilbyde
2. bagefter laves klasser, der implementerer interfacet (HVORDAN er data og algoritmer)

Interfacet kan bruges som en kontrakt mellem de, der laver og de der bruger klassen.

Hvad bruger man interfaces til (2)?

private modifieren kan bruges i klasser til *"information hiding"*: kun public metoder og konstanter eksporteres. Ved at gøre variabel felterne i en klasse **private** bliver det muligt at erstatte dem med andre felter og erstatte metodekroppene tilsvarende, uden at kald af metoderne behøver at blive ændret.

Interfaces bruges også til *"information hiding"* med tilsvarende effekt. Men hvis man fortryder en implementering f.eks. `Time1` skal man ikke ændre den, man kan blot lave en ny implementering f.eks. `Time2` og erstatte brugen af `Time1` med `Time2`.