UNIVERSITETET I OSLO

Institutt for informatikk

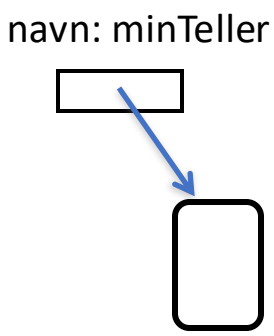
IN1010 våren 2025
30. januar

Del 1: Referanseverdier
Del 2: Mer om metoder og parameteroverføring i Java
Del 3: Definisjon av grensesnitt/tjenester
Del 4: Enhetstesting
Del 5: Invarianter i objekter og litt om arrayer
Del 6: To klasser fra Javas bibliotek

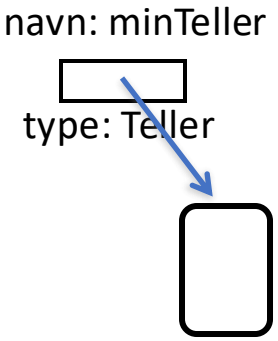
Stein Gjessing

Del 1: Om referanse-verdier (peker-verdier) og referanse variabler

Python:

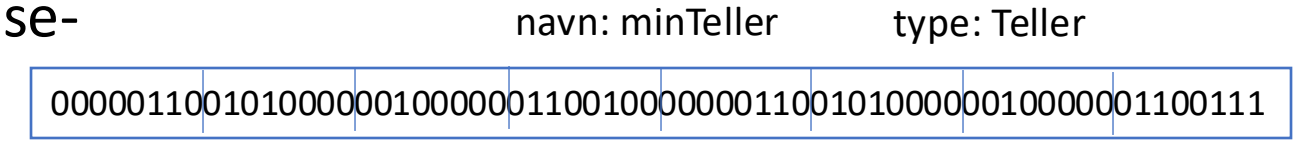


Java:



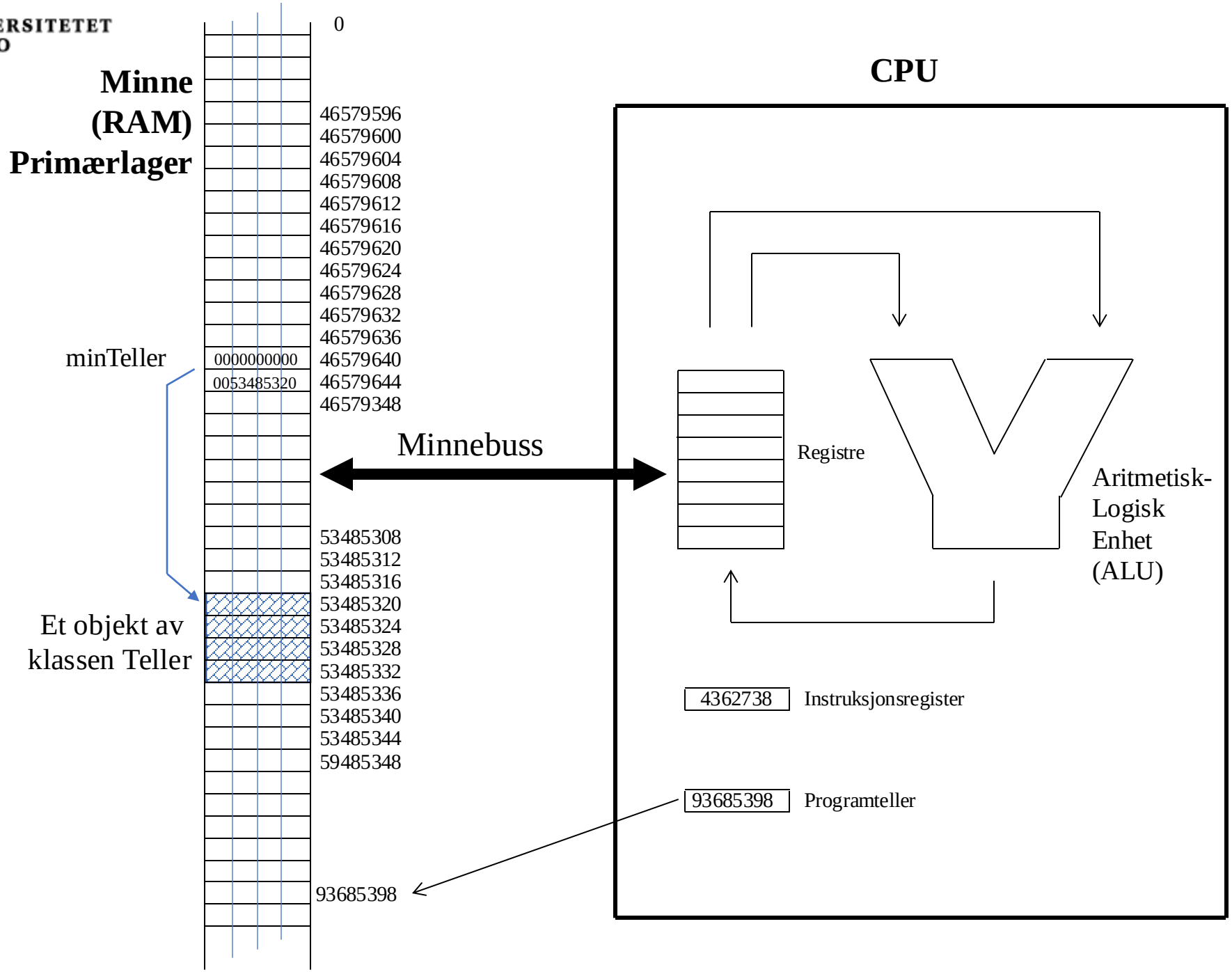
F.eks. deklarasjonen i Java: `Teller minTeller = new Teller();`

En referanse-variabel: som innholder verdien:



00000110010100000010000001100100000011001010000001000000 1100111

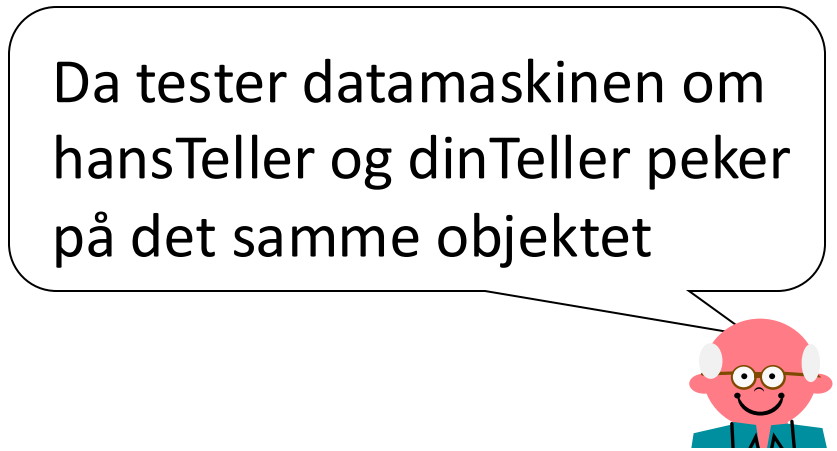
En referanse(verdi) er en adresse til et sted i minnet der det ligger et objekt (eller en array)
I Java bruker vi *referanse* og *peker* synonymt
Java er **et sterkt typet språk:** Referanse-variablen i dette eksemplet **har bare lov å inneholde en peker til et** Teller-objekt (sjekkes av kompilatoren)

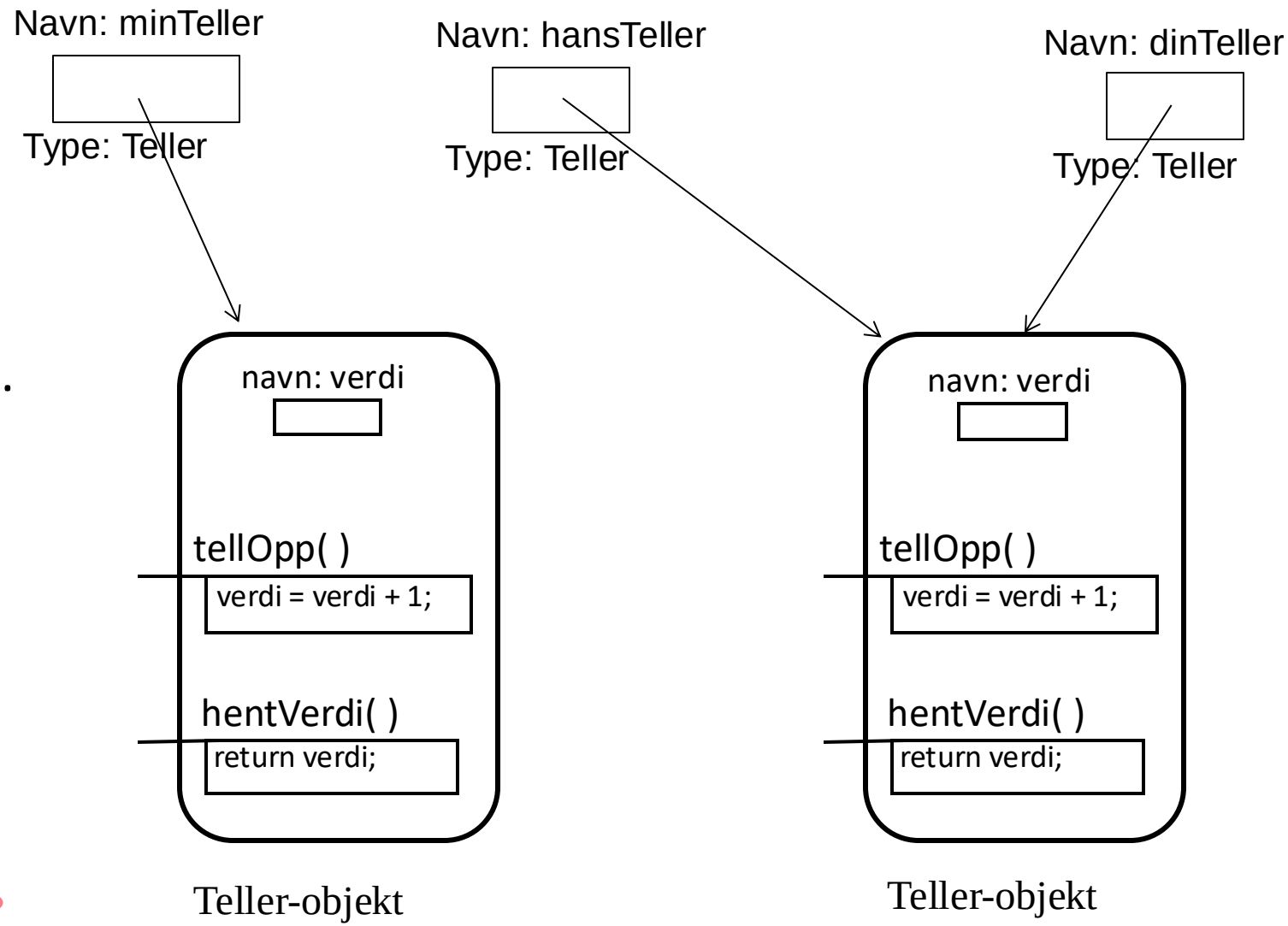

Minne
(RAM)
Primærlager
0
46579596
46579600
46579604
46579608
46579612
46579616
46579620
46579624
46579628
46579632
46579636
minTeller
0000000000
46579640
0053485320
46579644
46579348
Minnebuss
53485308
53485312
53485316
53485320
53485324
53485328
53485332
53485336
53485340
53485344
59485348
Et objekt av
klassen Teller
93685398
CPU
Registre
Aritmetisk-
Logisk
Enhet
(ALU)
4362738
Instruksjonsregister
93685398
Programteller

Referanser og likhet

Hva skjer når vi utfører

if (hansTeller == dinTeller) . . .



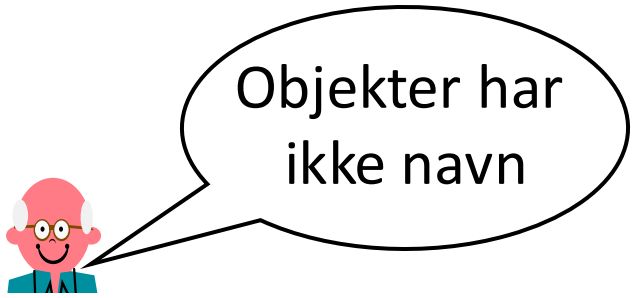


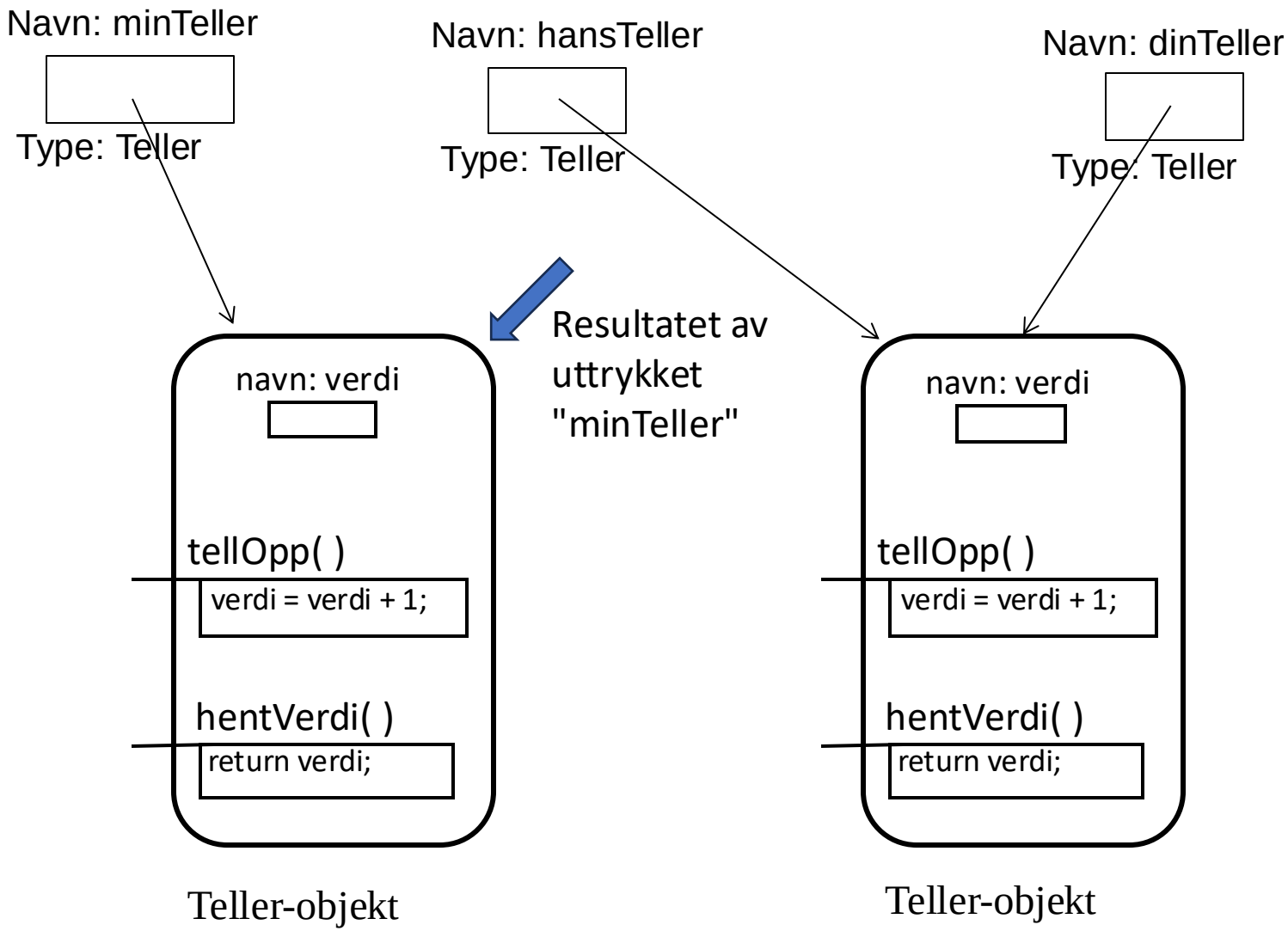
Referanser og tilordning

Hva skjer om vi nå utfører

dinTeller = minTeller;

?

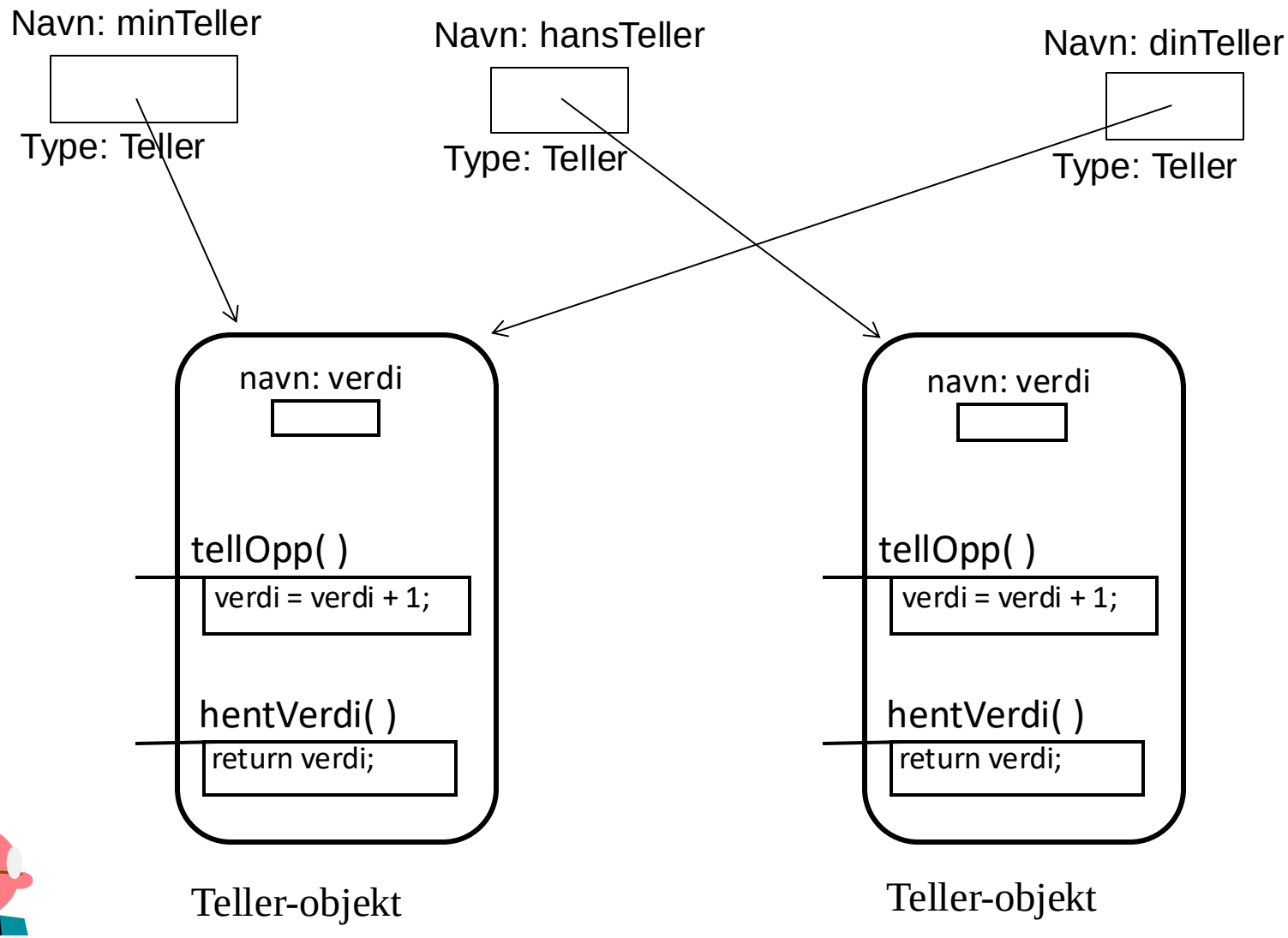




Referanser og tilordning

dinTeller = minTeller;

Dette er en referanse-tilordning (pekertilordning). Etter dette er situasjonen som vist på tegningen.



Jeg liker å uttale "dinTeller = minTeller;" slik:

"dinTeller settes til å peke på det samme som minTeller peker på"



Stein

Referanser og likhet

Hva skjer om vi nå utfører

if (hansTeller == dinTeller) . . .

Navn: minTeller

53485320

Type: Teller

Navn: hansTeller

9483567

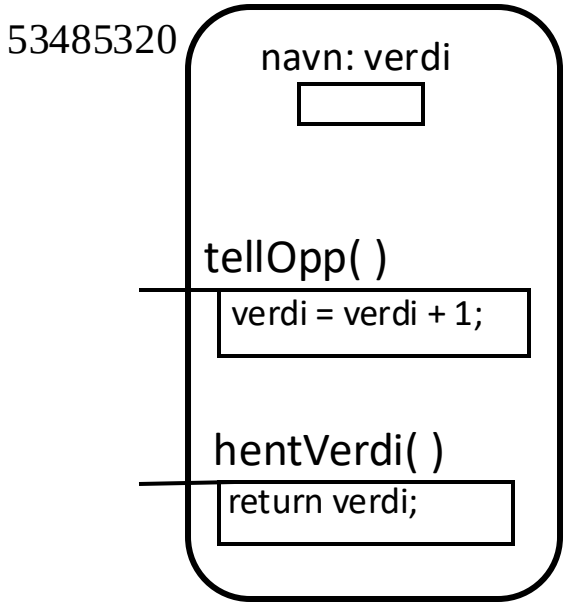
Type: Teller

Navn: dinTeller

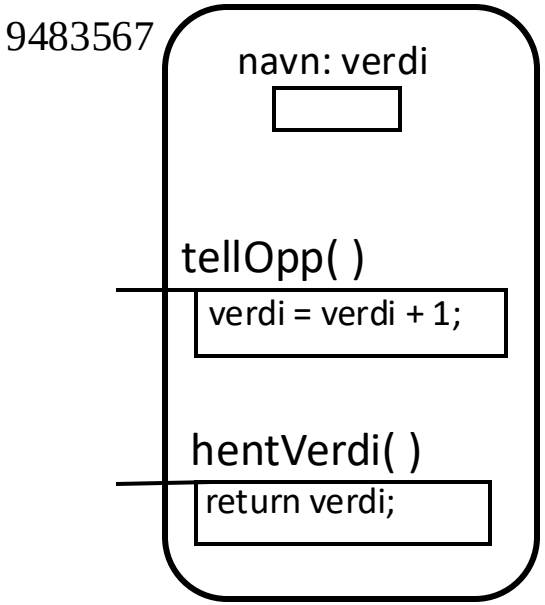
9483567

Type: Teller





Teller-objekt



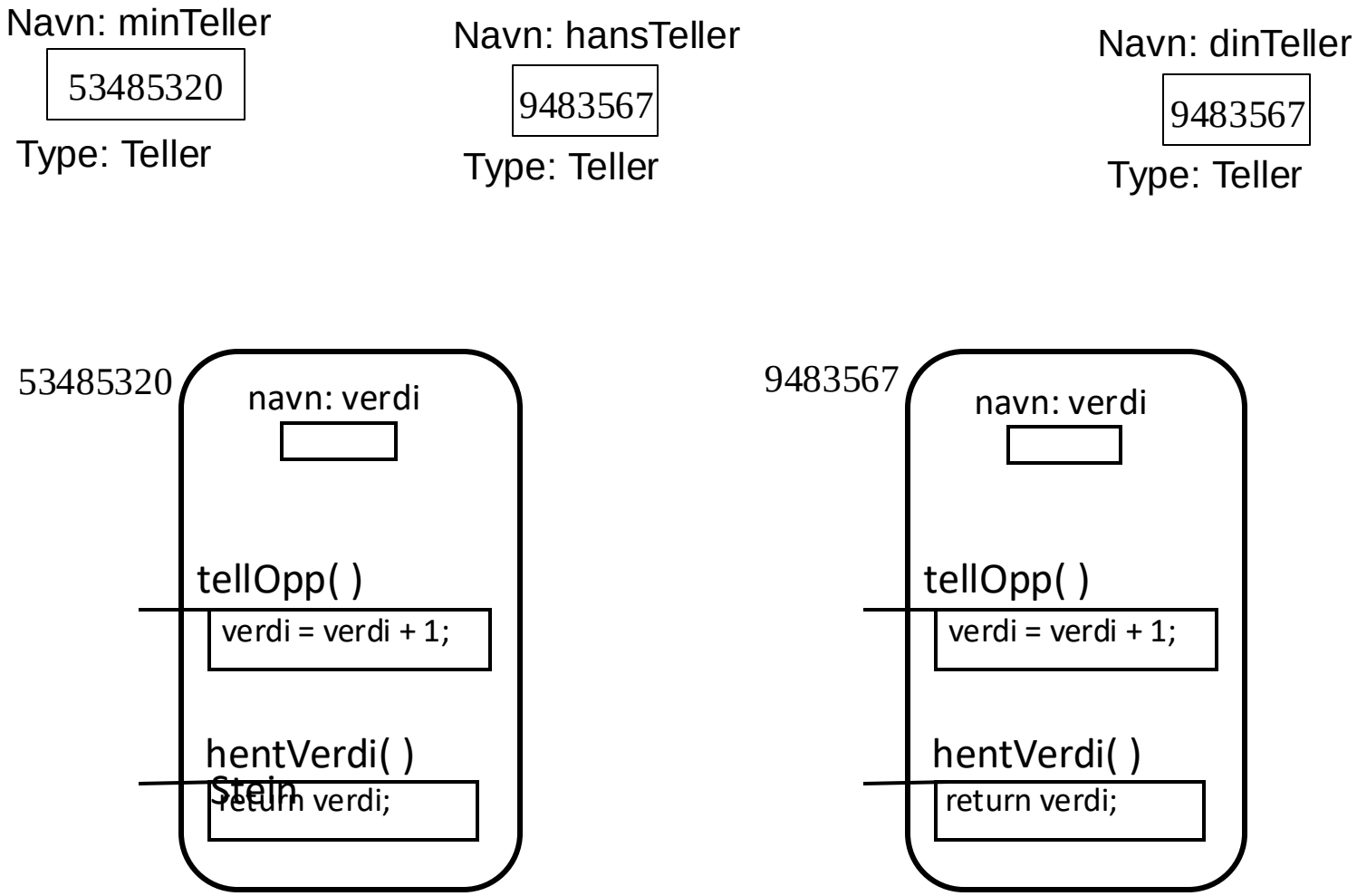
Teller-objekt

Referanser og likhet

Hva skjer om vi nå utfører

if (hansTeller == dinTeller) . . .

9483567 9483567



Vi sammenligner innholdet av de to variablene (egentlig uttrykkene) hansTeller og dinTeller !



8

Referanser og tilordning

Hva skjer når programmet utfører

dinTeller = minTeller;

?

Navn: minTeller

53485320

Type: Teller

Navn: hansTeller

9483567

Type: Teller

Navn: dinTeller

9483567

Type: Teller

53485320

navn: verdi

tellOpp( )

verdi = verdi + 1;

hentVerdi( )

return verdi;

Teller-objekt

9483567

navn: verdi

tellOpp( )

verdi = verdi + 1;

hentVerdi( )

return verdi;

Teller-objekt

En tilordning:
Verdien av uttrykket minTeller (53485320) blir puttet inn i variabelen dinTeller

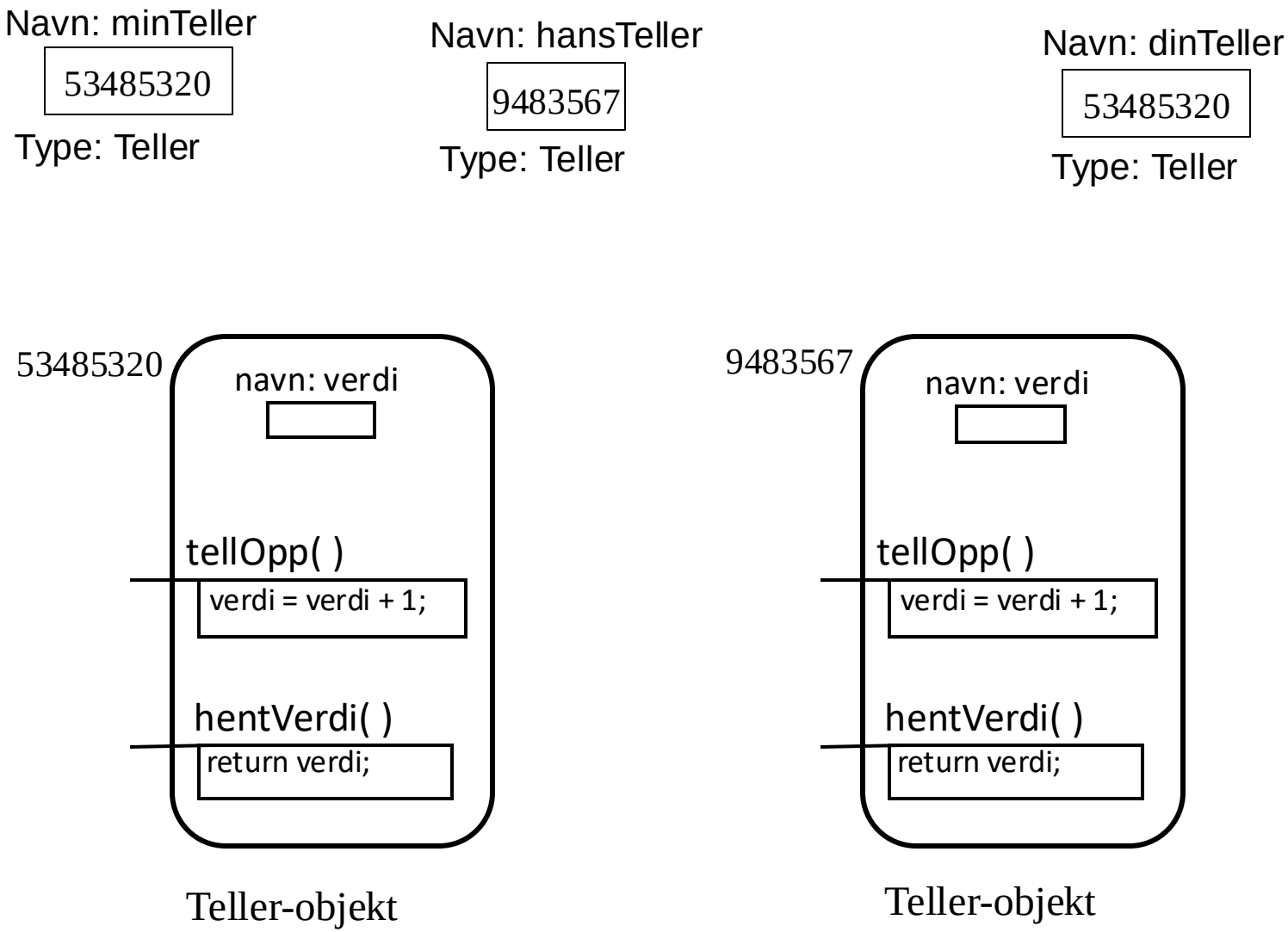


Referanser

Etter

dinTeller = minTeller;

er tilstanden slik:



Del 2: Metodeinstanser og parameteroverføring

Et objekt = en instans av en klasse

- Men hva er en instans av en metode (funksjon, prosedyre . . .)?
- Hva skjer når en metode blir kalt og utført?
  - Parameteroverføring
  - Metodeinstanser

La oss modifisere Teller-klassen litt


```
class Teller {
      private int verdi;

      public void tellOpp( ) {
            verdi = verdi + 1;
       }

      public int hentVerdi( ) {
            return verdi;
      }
}
```



```
class Teller {
      private int verdi;

       public void tellOpp( int antall ) {
            verdi = verdi + antall;
       }

       public int hentVerdi( ) {
            return verdi;
      }
}
```

Formell parameter der metoden er deklarert

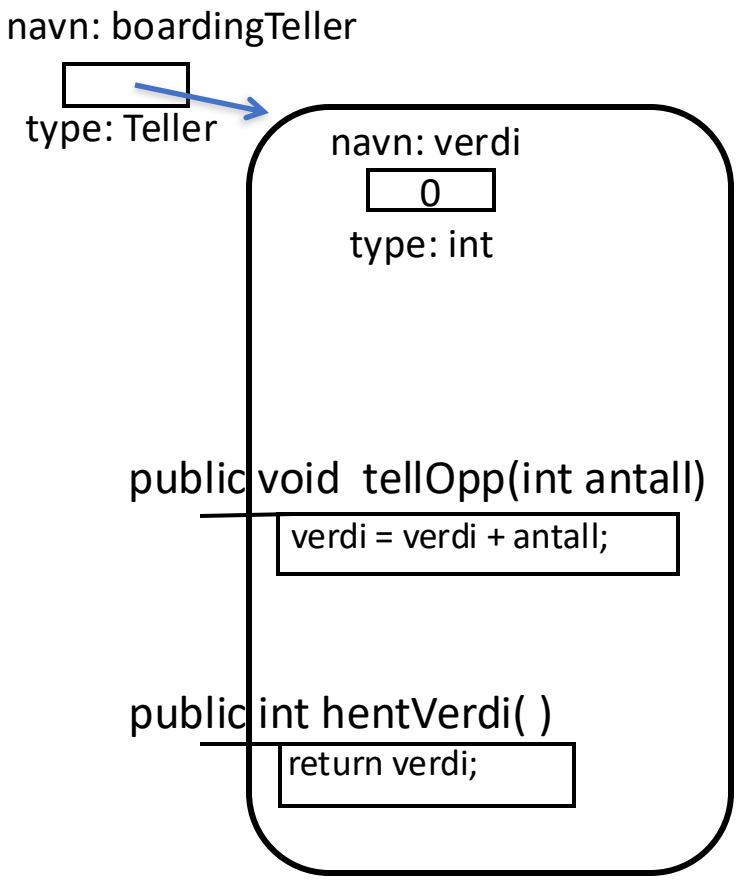
Aktuell parameter der metoden er kalt

```
Teller boardingTeller = new Teller( );

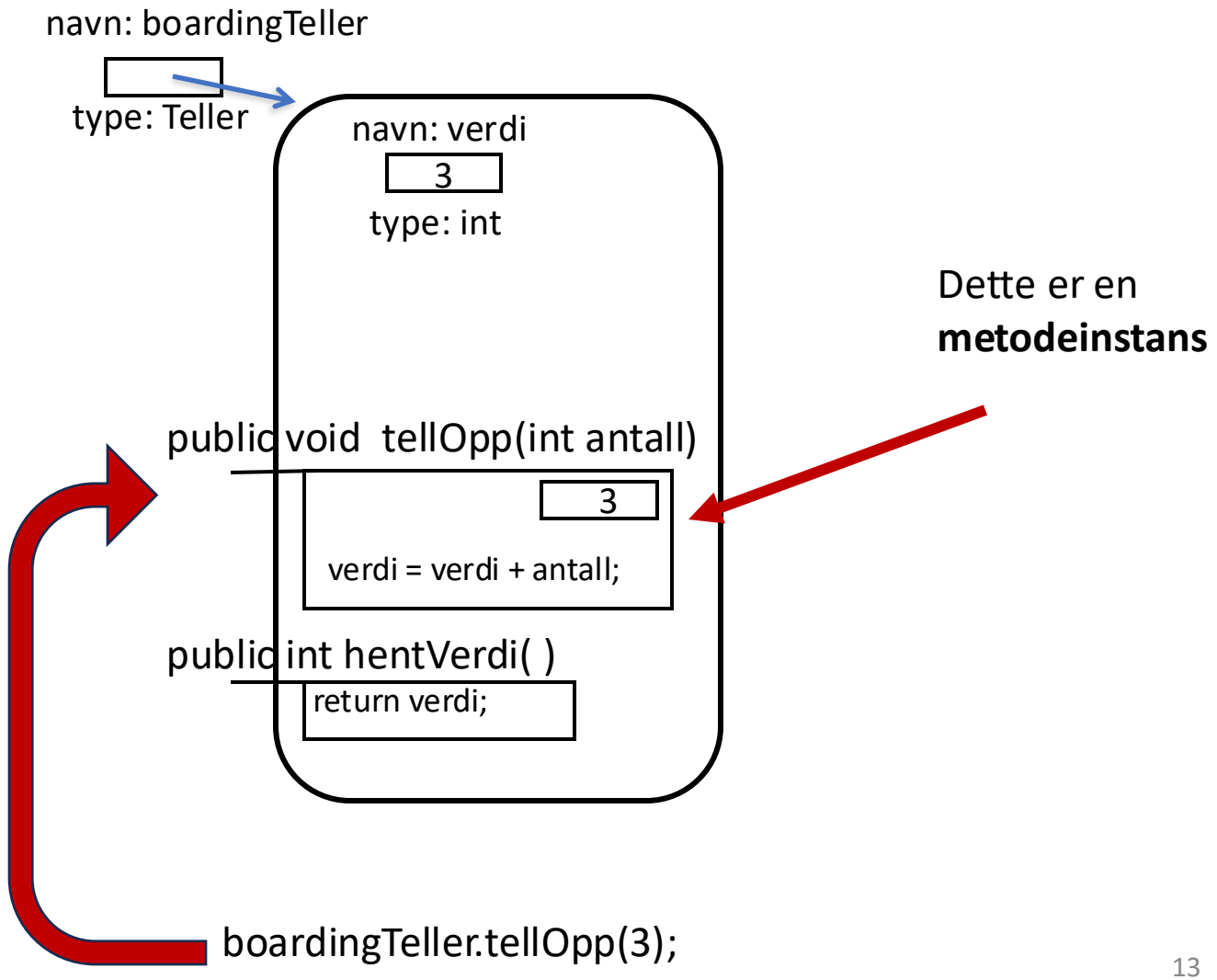
boardingTeller.tellOpp( 3 );

System.out.println(boardingTeller.hentVerdi( ));
```

Objekt med illustrasjon av metoder som **kan kalles**

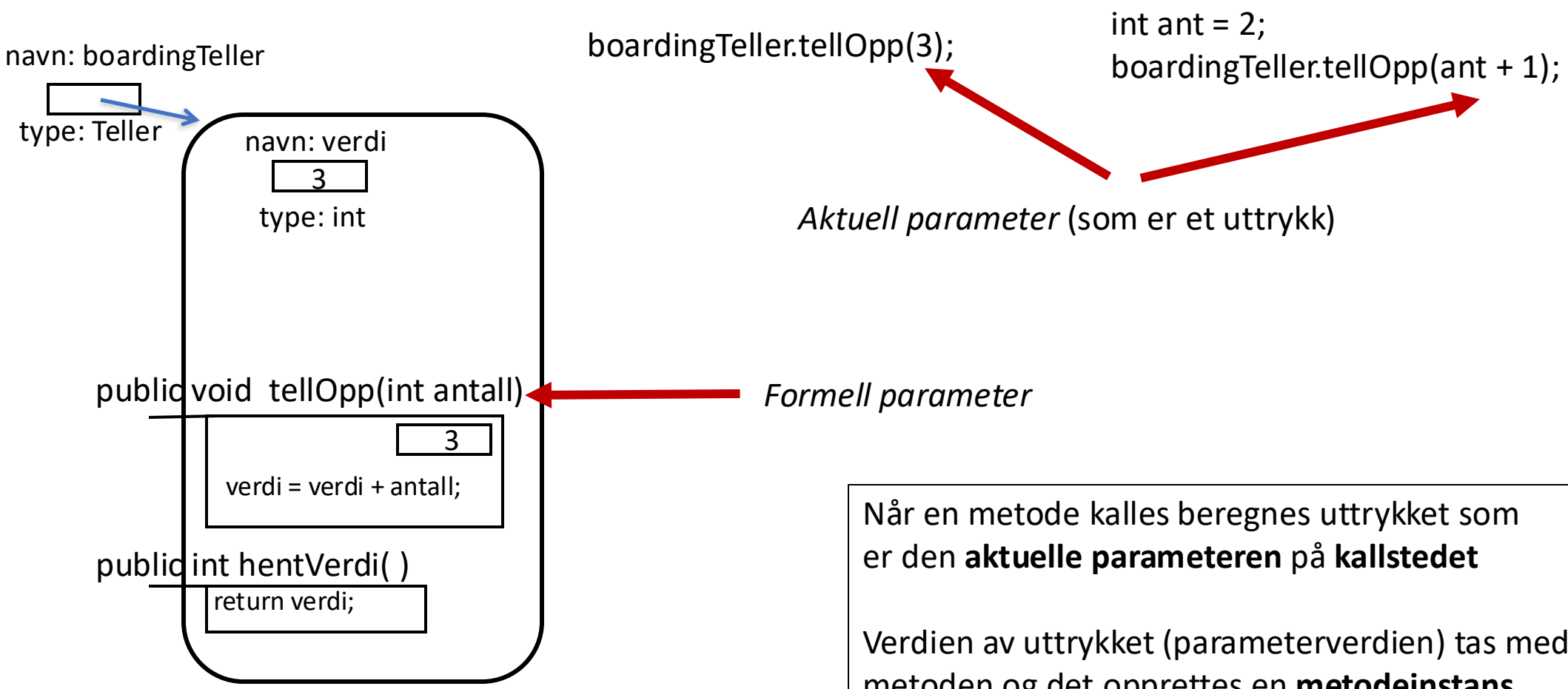


Objekt med illustrasjon av metode som **er kalt (som er under utførelse)**



Dette er en **metodeinstans**

Metodinnstanser Metodekall



Når en metode kalles beregnes uttrykket som er den **aktuelle parameteren** på **kallstedet**

Verdien av uttrykket (parameterverdien) tas med til metoden og det opprettes en **metodeinstans** der den **formelle parameteren** blir en **lokal variable** med parameterverdien som startverdi

Les mer i notatet: “Mer om objekter og klasser i Java”

Et objekt er en instans av en klasse

- Det settes av plass i minnet til instansvariabler hver gang vi sier new . . . og det opprettes et **objekt** (= instans av en klasse)
  - Den plassen i minnet som brukes til dette kalles ”haugen” (eng: the heap)
  - Her blir alle variabler initialisert med sine standardverdier

- Det samme skjer når vi oppretter en array
  - Arrayer legges også på haugen
  - Arrayer blir initialisert med standardverdier.

Metodeinstanser

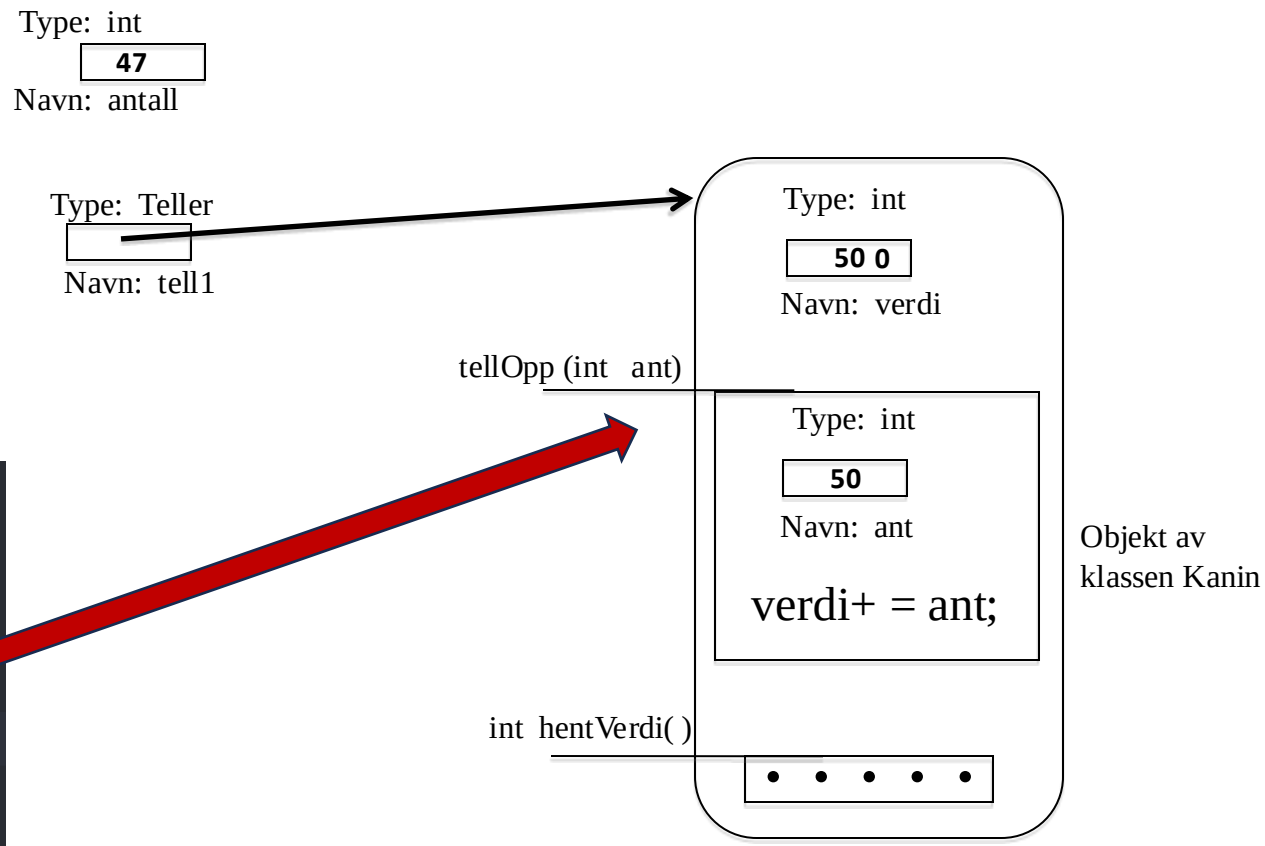
- Det settes av plass i minnet til *lokale variabler* (inkludert formelle parametre) hver gang vi kaller en metode og det opprettes en **metodeinstans**
  - I IN1010 forestiller vi oss at en metodeinstans oppstår inne i et objekt
  - I objektorientering gir dette oss en god tankemodell for hvilke egenskaper i objektet metoden har tilgang til (skop)
    - Men objektet blir ikke utvidet med mer data. Dataene til metodeinstansen blir lagt på stakken (se under)
  - Den plassen i minnet som brukes til lokale variabler i metodeinstanser kalles for stakken (eng: the stack)
  - På stakken blir variabler ikke initialisert (de får tilfeldige startverdier)
  - Se mere i Steins notat "Mer om objekter og klasser i Java"

Eksempel: Parameteroverføring - int

BrukTeller.java

```
class Teller {
      private int verdi = 0;
      public void tellOpp(int ant) {
           verdi = verdi + ant;
       }
      public int hentVerdi( ) {
           return verdi;
      }
}
```

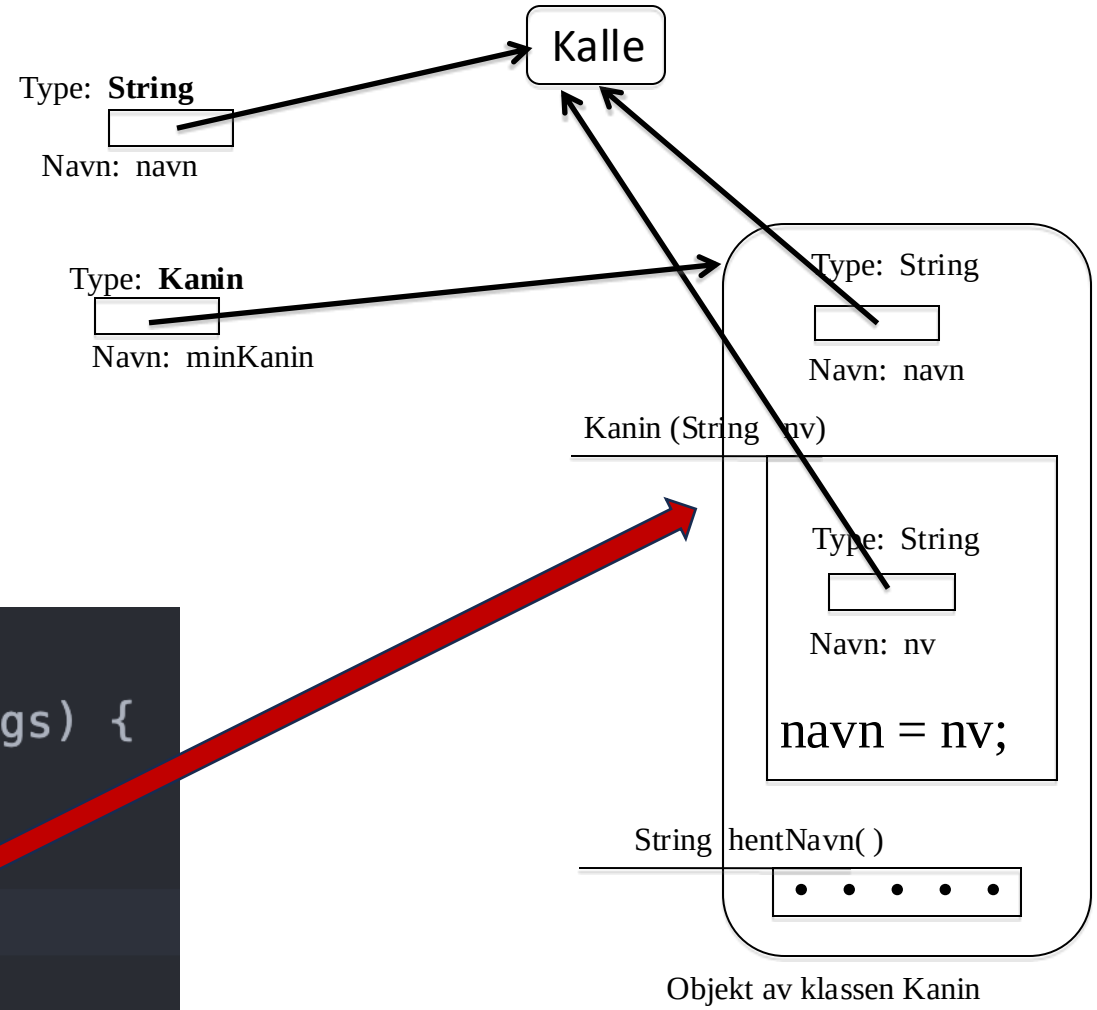
```
class BrukTeller {
     public static void main (String [ ] arg) {
          Teller tell1 = new Teller( );
          int antall = 47;
          tell1.tellOpp(antall+3);
      }
}
```



Parameteroverføring - referanse


```
class Kanin{
      private String navn;
      public Kanin(String nv) {
            navn = nv;
      }
      public String hentNavn()  {
          return navn;
      }
}
```

```
class BrukKanin {
    public static void main(String[]args) {
        String navn = "Kalle";
        Kanin minKanin;
        minKanin = new Kanin(navn);
    }
}
```



Sett dere sammen 2 – 3 – 4 – 5 og diskuter:

Stemmer det vi nå har lært om

formelle parametre,
aktuelle parametre og
metodeinstanser

med hva vi har lært før om parameteroverføring (Python - IN1000)

Del 3: Software engineering

- Hvilke tjenester skal et program tilby?
- Hvilke tjenester skal et **objekt** tilby?
  - Virkningen av denne tjenesten

- Hvilken oppgave er det vi skal løse?
- Hva er det vi skal programmere?

Ikke alltid lett å vite hva vi skal programmere



Når dere får en oppgave
vil dere ofte synes at
oppgaveteksten er
alt for lang og komplisert.

Men det er for at oppgaven
skal være så presis at
dere programmerer
den “riktig”.

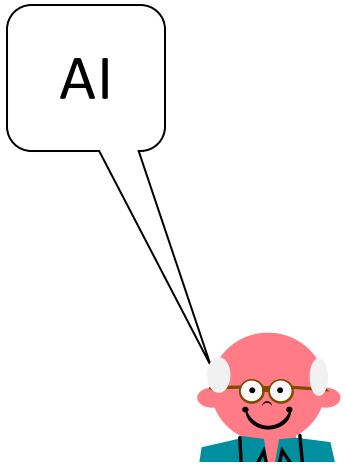
Derfor smidig
systemutvikling

Vi har enda ikke fått snakket så mye om

. . . at det er dere som skal lage fremtiden

Etikk
Moral

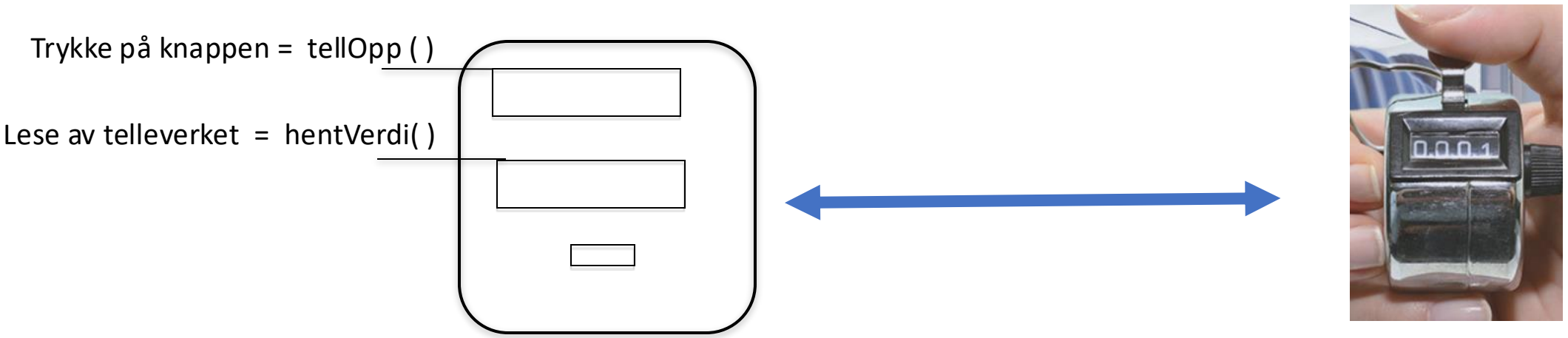




For en uke siden programmerte vi en teller

- Hvilke tjenester ønsker brukeren seg
- Har vi laget de tjenestene brukerene trenger?
- Hvem bestemmer hva grensesnittet skal være?
- Har vi programmert riktig?
- ??????????

Grenessnitt = tjeneste

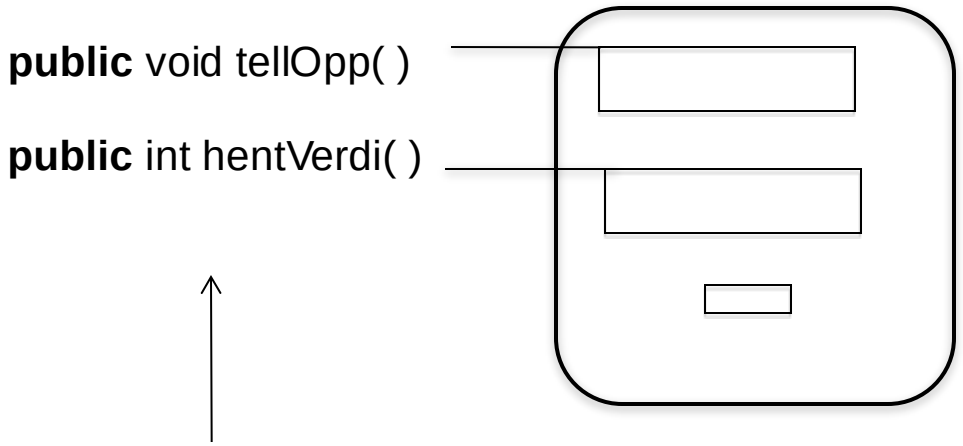


Det er ikke alltid vi får en så veldefinert oppgave



Når vi lager tjenester:

En metodes *signatur*



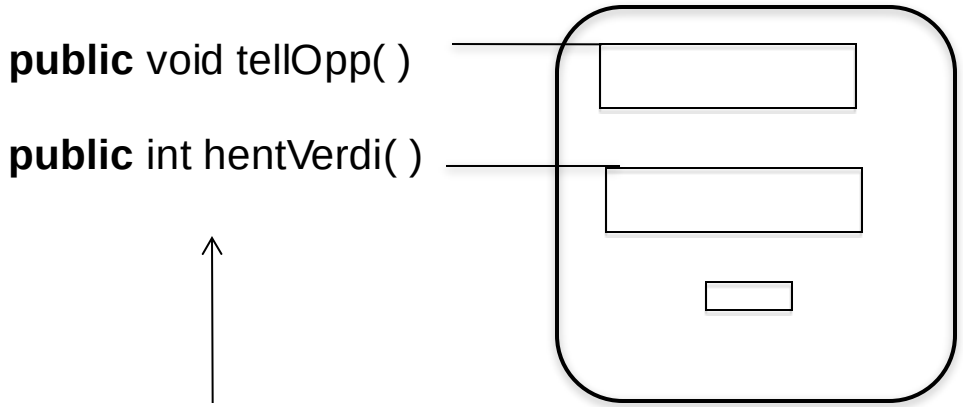
Dette kaller vi metodenes **signaturer** (skrivemåte, syntaks)

Signaturen til en metode er

- navnet på metoden
- typene, rekkefølgen og navnene til parametrene
- retur-typen (ikke i Java)

Når vi lager tjenester:

En metodes *semantikk*



Semantikk betyr
virkemåte



Hva gjør disse metodene? Hvordan virker de? Hvilke **tjenester** tilbyr de?
Hvordan beskrives disse tjenestene? Hva er sematikken til objektet / klassen?

Forslag til semantikk for et Teller-objekt (norsk):
- Når objektet opprettes er verdien 0
- Metoden hentVerdi returnerer objektets verdi
- Metoden tellOpp øker objektets verdi med 1

Men her snakker vi litt om (en tenkt) implementasjon

Hva er en referanseimplementasjon? (slå opp på nettet)
(reference implementation)

- # Informatikkens 3. lov: ☺

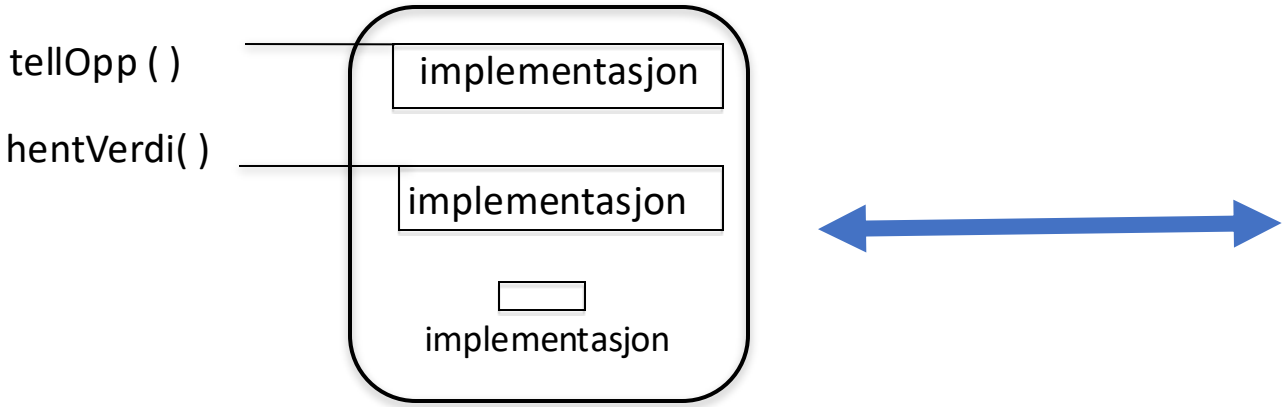
1. Først betemmer vi semantikken og signaturene til metodene
2. Deretter implementerer vi metodene
   samtidig som vi bestemmer hva de **private** dataene skal være

Dette gjelder for alle programmeringsspråk - dette er ikke Java-spesifikt.

☺ Dette er en spøk. Informatikken har ikke nummererte lover

Bare du, som er et menneske*, kan sjekke at implementasjonen overholder de SEMANTISKE KRAVENE til metodene. De semantiske kravene er gitt i naturlig språk (?)





Semantikk:

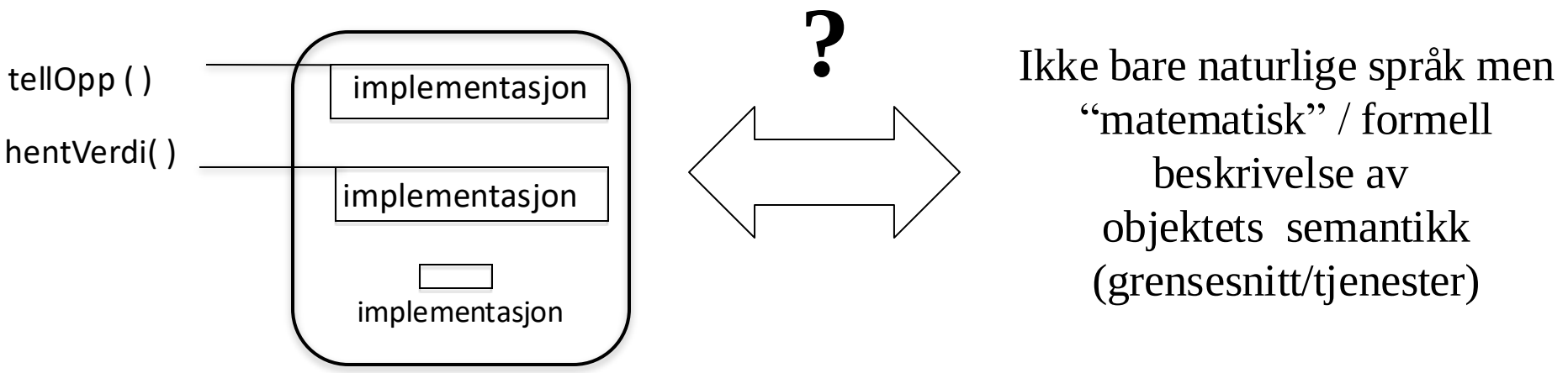
- Når objektet opprettes er verdien 0
- Metoden hentVerdi returnerer objektets verdi
- Metoden tellOpp øker objektets verdi med 1

* Se neste side

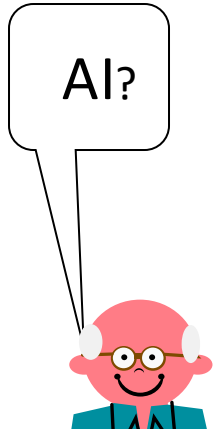
Institutt for informatikk har 17 forskningsgrupper.

En av disse heter "Pålitelige systemer" (PSY).

Her arbeider de bl.a. med å formalisere disse sematiske kravene, slik at du kan få hjelp av datamaskinen til å sjekke at implementasjonen overholder de semantiske kravene. Litt mer på en senere forelesning.



*De sematiske kravene kalles også en **"kontrakt"***
(mellom brukerene av objektet og objektet selv)

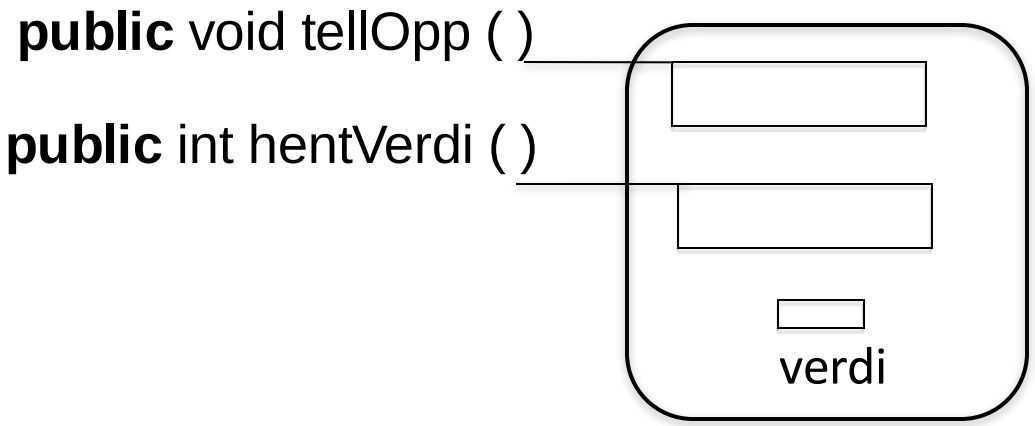


Del 4: Enhetstesting av objekter

Nå skal du lære litt om OO-programmering samtidig som du lærer mer Java

Hvordan vet vi at programmet gjør det det skal?
Hvordan vet vi at objektet gjør det det skal?


```
class Teller {
      private int verdi = 0;
      public void tellOpp( ) {
            verdi++;
      }
      public int hentVerdi( ) {
            return verdi;
      }
}
```



Et Teller-objekt

Vi må gjøre 2 ting:

- *Vi må tenke og resonere nøye når vi designer og skriver programmet*
- *Vi må **teste** programmet*

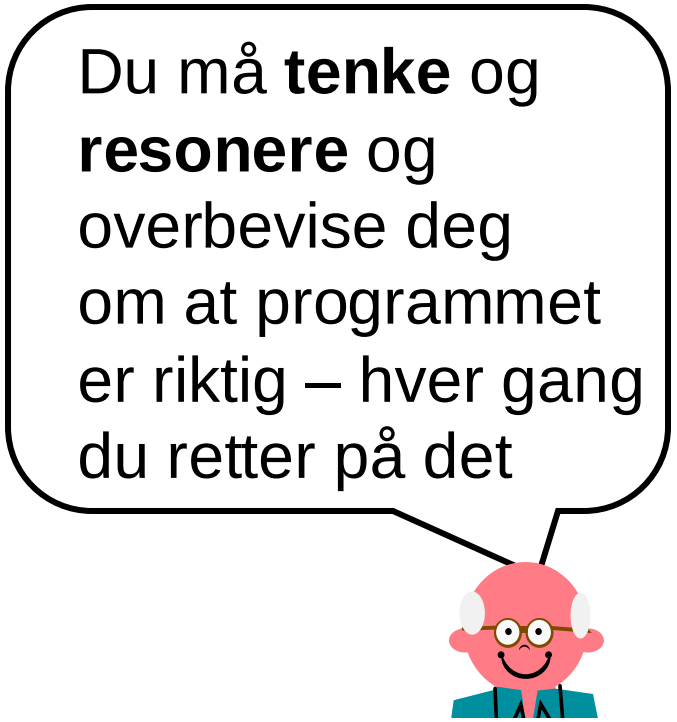
Testing -- Enhetstesting

- Når vi planlegger og skriver programmer prøver vi å overbevise oss selv (og dem vi skriver sammen med) at den koden vi skriver kommer til å utføre det vi ønsker (oppfylle semantikken)
- Men vi kommer alltid til å tenke og skrive feil
- Derfor må vi **teste** programmene våre
- Objektorientering / modularisering:
  - Teste et objekt eller en modul om gangen - **enhetstesting**
    - Sørg for at den er så riktig som mulig
  - Deretter kan vi test sammensettingen av objektene / modulene
    - **integrasjonstesting**

Mer om testing 3. april

Det er IKKE lov å

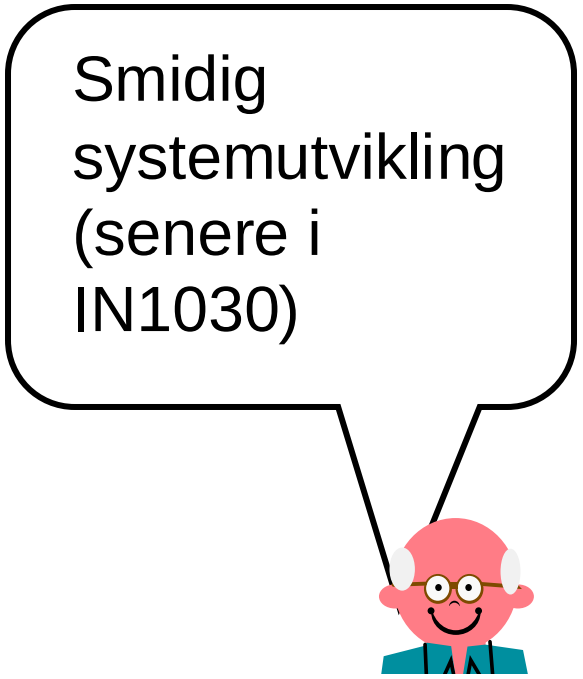
- Skrive et program til du tror at det kanskje virker
- Teste det, men det feiler
- **Gjette** på hva som er feil og rette én ting
- Teste på ny, men det feiler
- **Gjette** på hva som er feil og rette én ting
- Teste på ny, . . .
- . . .



Når du skriver et større program SKAL du gjøre slik som Dag gjorde på mandag

- Utvid programmet litt etter litt og kompiler og kjør det hele tiden

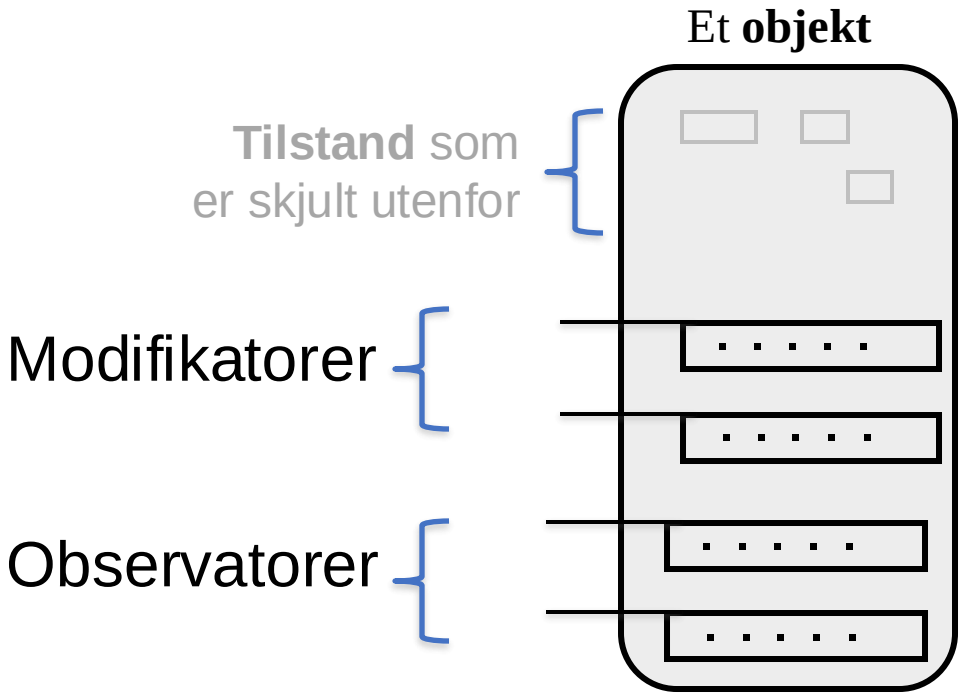
- Smidig systemutvikling (IN1030):
  - Bygg systemt opp gradvis og kommuniser med den/de som skal bruke programmet at du/dere er på riktig vei i systemutviklingsprosessen



Om grensesnitt/tjenester: Modifikatorer og Observatorer i objekter

(Accessors and mutators)

- En modifikator-metode forandrer (modifiserer) tilstanden til et objekt
- En observator-metode leser av (observerer). tilstanden uten å forandre den

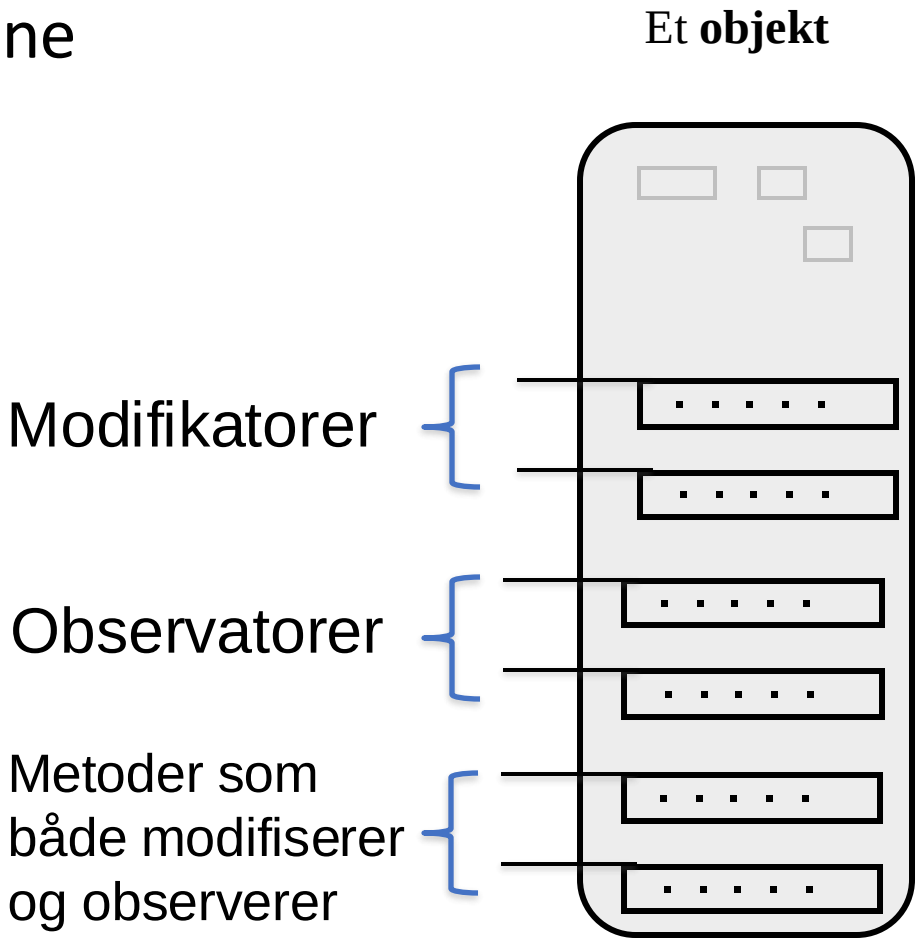


Modifikator, f.eks. tellOpp(), settVerdi()

Observator, f.eks. hentVerdi()

Testing - - Enhetstesting

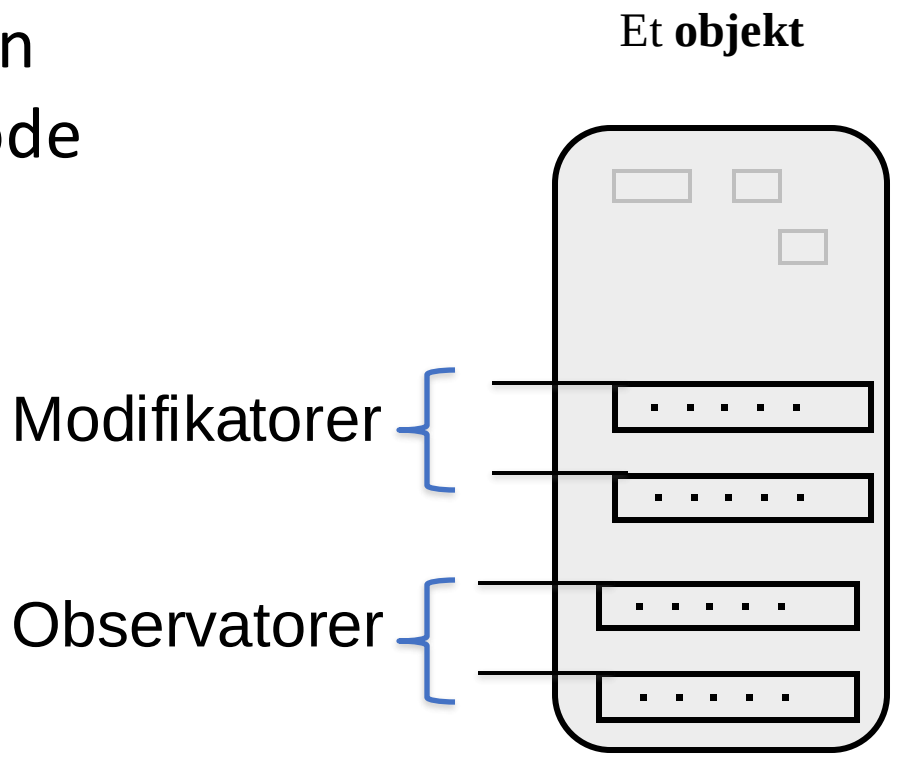
- Lag et objekt og se hva som skjer når du kaller metodene
- Du må selv sette opp rekkefølgene av kallene og vite hva resultatet bør bli
- Hva er de mulige feilene?
  - Ekstern testing vs. intern testing.
- Prøv å ta med så mange forskjellige kall-rekkefølger som mulig som du tror kan vise feil



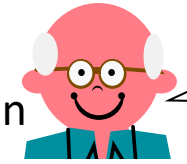
Testing - - Enhetstesting

Hvis mulig:

- Når vi skal teste et objekt kan vi først kalle en observator-metode, så en modifikator-metode og så igjen en observator-metode og se om vi observerer det ønskede resultatet - den ønskede forandringen i **tilstanden**.



For de som er spesielt interessert:

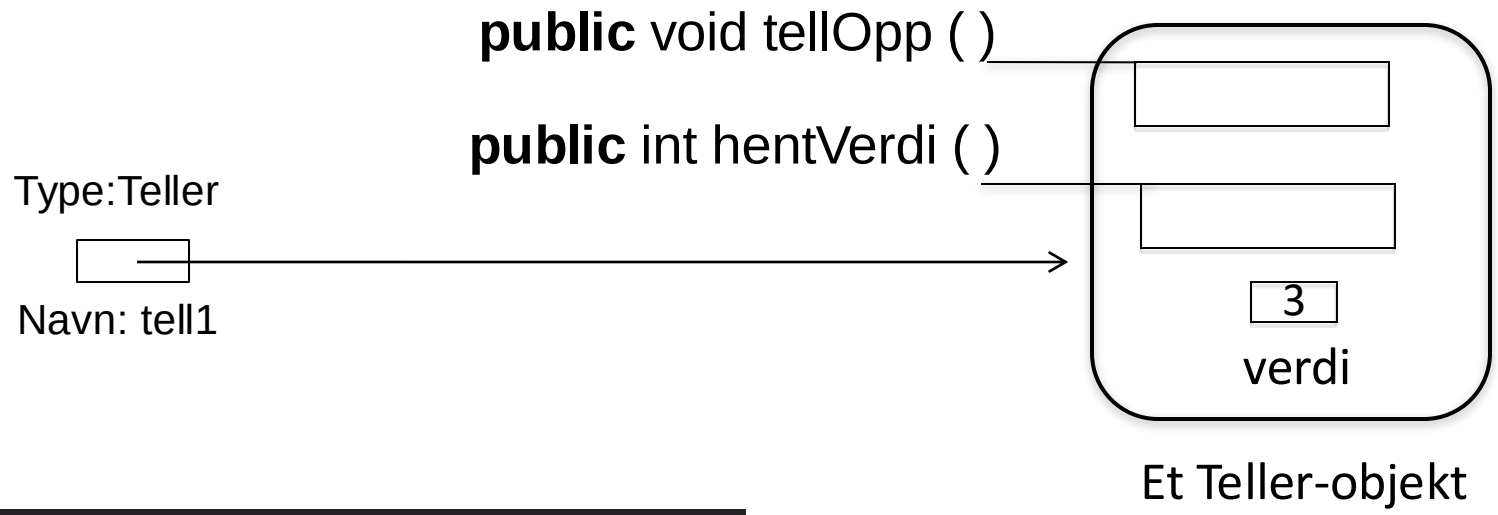


Ole-Johan Dahl

Objektets semantikk (og tilstand) kan beskrives av den historiske sekvensen av operasjoner på objektet

Om å teste et objekt / klasse

```
class Teller {
      private int verdi = 0;
      public void tellOpp( ) {
            verdi++;
      }
      public int hentVerdi( ) {
            return verdi;
      }
}
```



Et Teller-objekt

```
class BrukTellerTest1 {
     public static void main (String [ ] arg) {
           Teller tell1 = new Teller( );
           tell1.tellOpp();
           tell1.tellOpp();
           tell1.tellOpp();
           // test:
           System.out.println("Telleren er nå " + tell1.hentVerdi());
           System.out.println("Telleren skal være 3");
           if (tell1.hentVerdi() == 3) System.out.println("Svaret er riktig");
              else System.out.println("Svaret er feil");
      }
}
```

Når skal vi skrive testprogrammet?

Om du er alene?
Om dere er flere?

Samme – men med egen (statisk) metode for å teste


```
class Teller {
      private int verdi = 0;
      public void tellOpp( ) {
            verdi++;
      }
      public int hentVerdi( ) {
            return verdi;
      }
}
```




```
class BrukTellerTest2 {
     public static void main (String [ ] arg) {
           Teller tell1 = new Teller( );
           tell1.tellOpp();
           tell1.tellOpp();
           tell1.tellOpp();
           // test:
           test(tell1.hentVerdi(),3);
      }
      private static void test (int resultat, int fasit) {
           if (resultat == fasit) System.out.println("Svaret er riktig");
              else System.out.println("Svaret er feil");
      }
}
```

Flere eksempler på enhetstesting: Om å lage et kaninbur

```
class Kanin{
   private String navn;
   public Kanin(String nv) {
      navn = nv;
   }
   public String hentNavn() {
      return navn;
   }
}
```





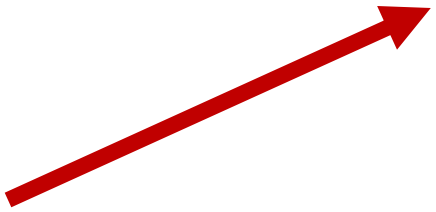
```
class Kaninbur {
   private . . . ;
   public boolean settInn(Kanin k) {
      . . .
   }
   public Kanin taUt( ) {
      . . .
   }
}
```

Kaninburets tjenester / grensesnitt – Signaturene og semantikken til metodene i kaninburet

Signaturer (syntaks)

```
class Kaninbur {
      public boolean settInn(Kanin k) {. . .}
      public Kanin taUt( ) { . . . }
}
```

Kan det beskrives enklere ?



Semantikk (virkemåte):

- Hvis objektet (buret) er tomt vil metoden "settInn" gjøre at objektet tar vare på kaninen som er parameter til metoden, og metoden returnerer sann.
  Hvis objektet allerede inneholder en kanin gjør metoden ingen ting med objektet, og metoden returnerer usann.

- Metoden "taUt" tar ut kaninen som er i objektet og returnerer en peker til denne kaninen**.** Metoden returnerer null hvis objektet (buret) var tomt.

Kaniner og kaninbur: Full kode


```
class Kanin{
   private String navn;
   public Kanin(String nv) {
      navn = nv;
   }
   public String hentNavn() {
      return navn;
   }
}
```



Men enda ikke et fullt program (mangler klasse med main).



```
class Kaninbur {
   private Kanin denne = null;
   public boolean settInn(Kanin k) {
     if (denne == null)  {
       denne = k;
       return true;
     }
     else return false;
   }
   public Kanin taUt( ) {
      Kanin k = denne;
      denne = null;
      return k;
   }
}
```

Enhetstesting av class Kaninbur

```
class Kanin{
   private String navn;
   public Kanin(String nv) {navn = nv;}
   public String hentNavn() {return navn; }
}
```

```
class Kaninbur {
   private Kanin denne = null;
   public boolean settInn(Kanin k) {
     if (denne == null)  {
       denne = k;
       return true;
     }
     else return false;
   }
   public Kanin taUt( ) {
      Kanin k = denne;
      denne = null;
      return k;
   }
}
```

Testprogram:

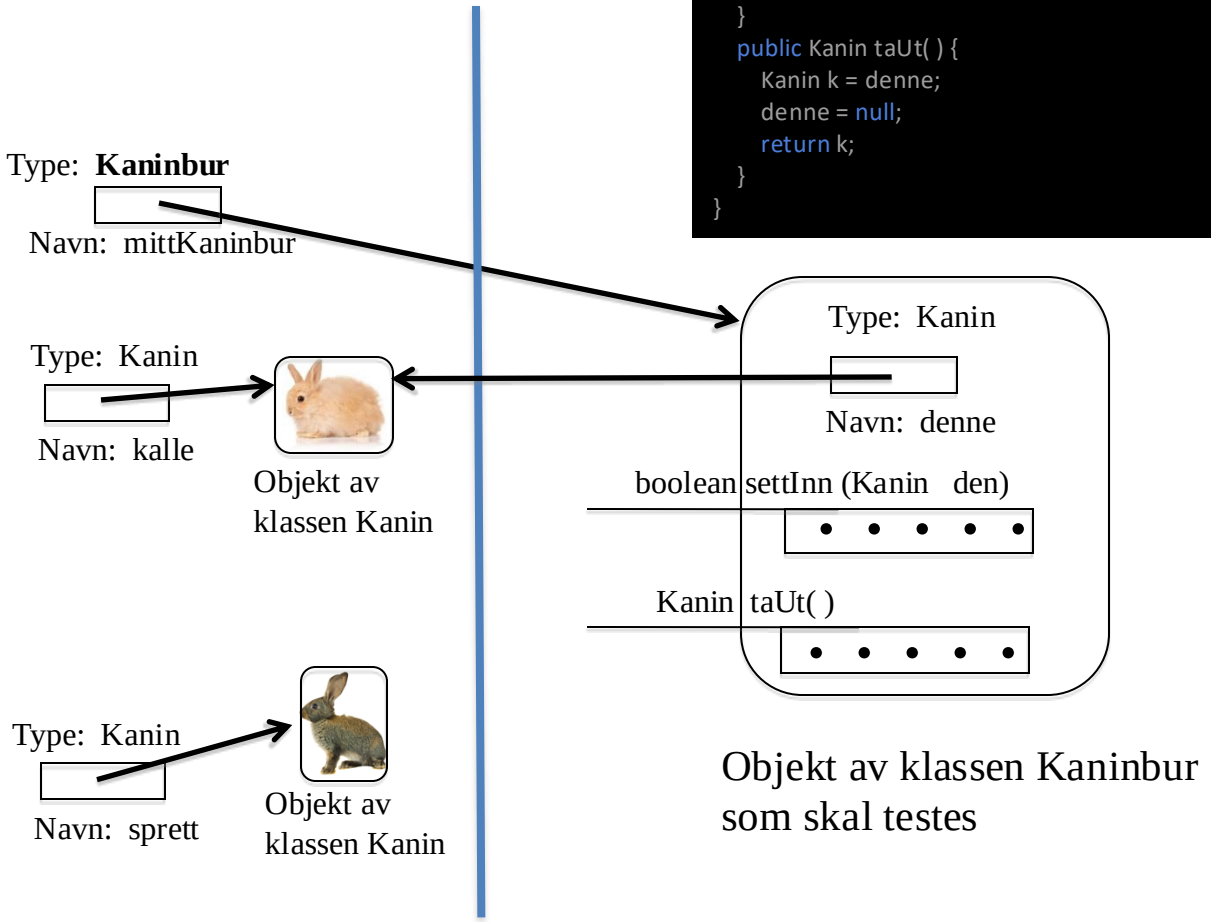
```
class Kanintest {
  public static void main ( String []args) {
     Kaninbur mittKaninbur = new Kaninbur( );




    Kanin kalle = new Kanin("Kalle");
    // Test å sette inn i tomt bur:
    boolean settInnOK = mittKaninbur.settInn(kalle);
    test("Test inn i tomt bur", settInnOK);


    Kanin sprett = new Kanin("Sprett");
    // Test å sette inn i fullt bur:
    boolean settInnOK = mittKaninbur.settInn(sprett);
    test("Test inn i fullt bur", !settInnOK);
    . . .
    . . .
    . . .

  }
  static void test(...) { . . . }
}
```



Sett dere sammen 2 – 3 – 4 – 5 og diskuter i 5 minutter

- Er det slik dere testet programmer i IN1000?
- Er det slik dere testet klasser / objekter i IN1000?
- Er dere flinke til å spille “djevelens advokat” og tro at programmet inneholder feil?

Del 5: Om invarianter på data og litt mer om bruk av arrayer

- Om invarianter
- Mer om arrayer

Norsk: Antagelser / Betingelser
Engelsk: Assertions / Conditions

- En beskrivelse av tilstanden på et bestemt sted i programmet kalles en antagelse eller en betingelse, f.eks.:

```
int tall = 0;
while (tall < 10) {
      tall++;
}
// Nå er !(tall < 10)
// dvs.   tall >= 10
```

tall >= 10 kalles en antagelse eller en betingelse

Et godt program inneholder kommentarer med gyldige antagelser om tilstanden til variablene i programmet på dette stedet i koden

WIKIPEDIA:
Invariant (computer science)

- In computer science, an **invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a **loop invariant** is a condition that is true at the beginning and end of every execution of a loop.
  . . . . .
- Programmers often use assertions in their code to make invariants explicit. Some object oriented programming languages have a special syntax for specifying **class invariants**.

"Loop invariants" skal vi ikke snakke om nå men er på planen den 15. mai.

Invarianter i objekter

- Ofte kalt (noe unøyaktig) "class invariants"
  - Handler om forholdet mellom instansvariablene

- En invariant på datastrukturen i et objekt hjelper oss å programmere metodene bedre og mer oversiktlig

- En invariant på datastrukturen i et objekt sier oss noe om instansvariablenes verdier, forholdet mellom verdiene til instansvariablene og generelt om datastrukturen inne i objektet – at datastrukturen er konsistent

- En invariant på datastrukturen i et objekt forklarer oppgaven til instansvariablene og begrenser verdiene de kan ha
  - Litt slik som typen til en variabel gjør

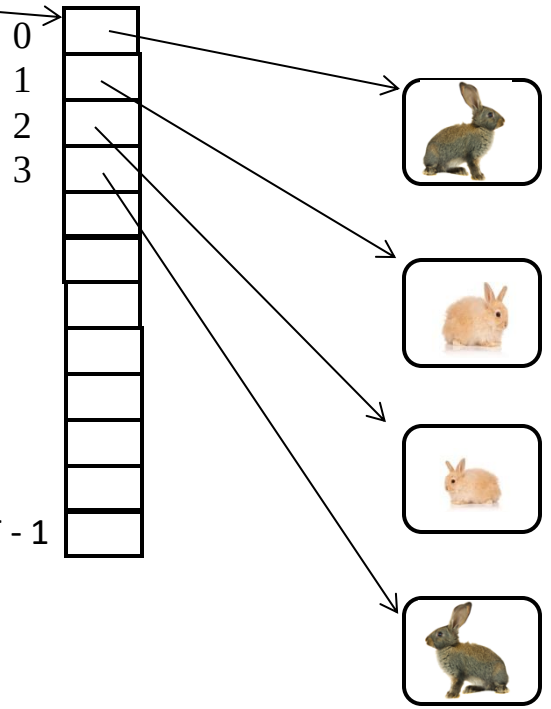
En kaningård som inneholder 4 kaniner

Navn: antall

4

Type: int

Kanin [ ] alle



public void settInn (Kanin en)

```
if (antall < MAXANT) {
        alle [antall] = en;
        antall ++;
}
```

public Kanin taUt( )

```
if (antall > 0)  {
        antall --;
        return alle [antall];
} else return null;
```

public Kaningård ()

```
antall = 0;
```

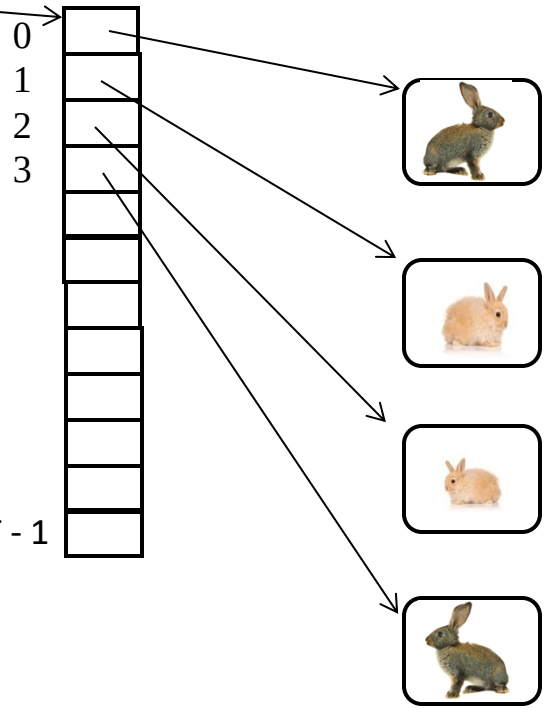
Invarianten holder i det objektet opprettes OG alle metodene bevarer innvarianten -> Da holder invarianten alltid

Navn: antall

4

Type: int

Kanin [ ] alle



public void settInn (Kanin en)

Pre-condition: Invarianten gjelder

```
if (antall < MAXANT) {
        alle [antall] = en;
        antall ++;
}
```

Post-condition: Invarianten gjelder

Invariant:
0 <= antall <= MAXANT og
alle kaninene er lagret fom.
alle[0] tom. alle[antall-1]

public Kanin taUt( )

Pre-condition: Invarianten gjelder

```
if (antall > 0) {
        antall --;
        return alle [antall];
} else return null;
```

Post-condition: Invarianten gjelder

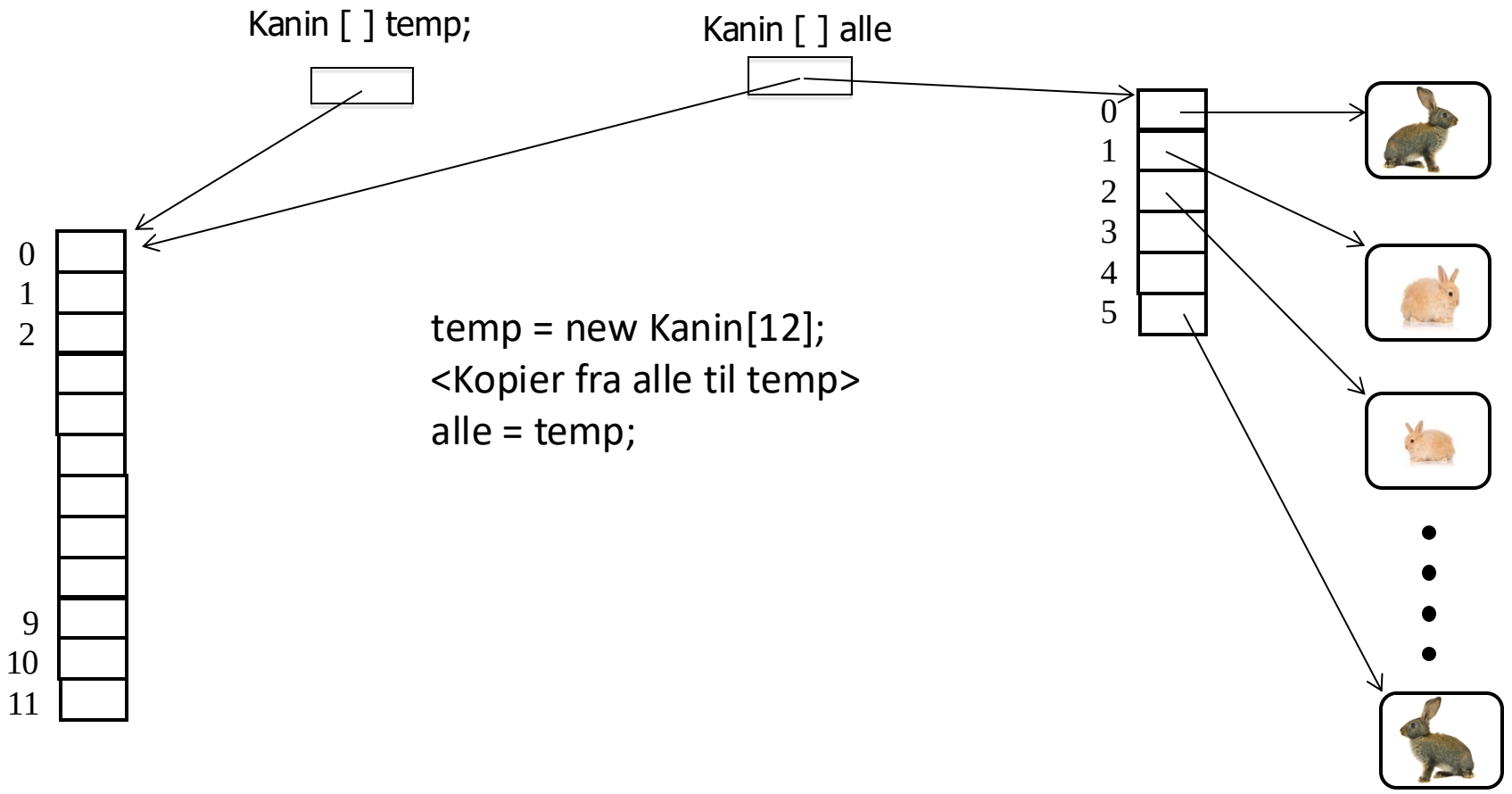
public Kaningård ()

Pre-condition: true

```
antall = 0;
```

Post-condition:
Invarianten gjelder

Array-tilordning kan være en løsning om kaninburet blir for lite ☺



Del 6: To samlinger (beholdere/collections) fra Javas bibliotek:

ArrayList og **HashMap**

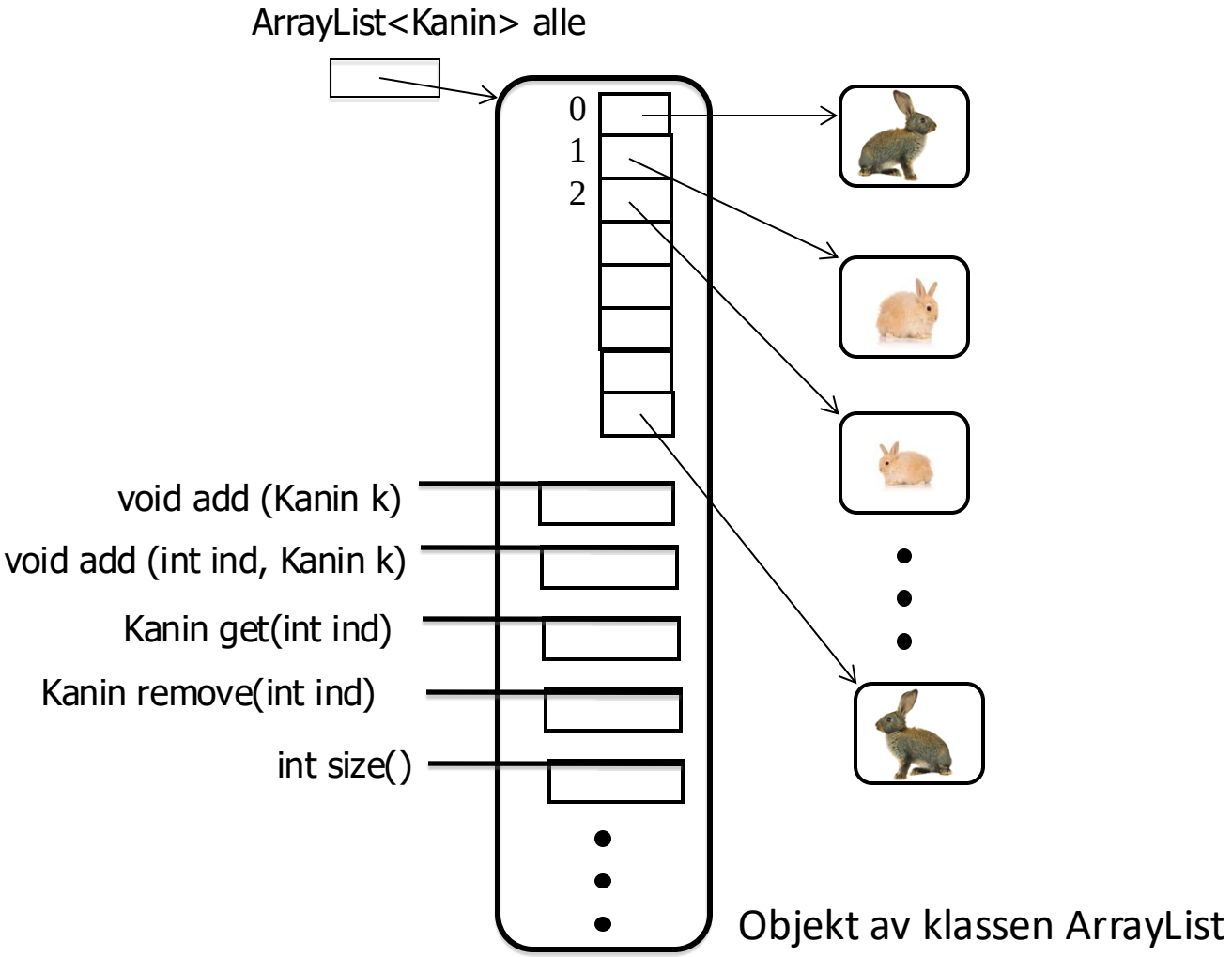
- ArrayList er en fleksibel array som utvider seg og trekker seg sammen etter behov
- ArrayList <Kaniner> mineKaniner = new ArrayList <Kaniner> ();
- Metoder: add, get, remove, . . . se Java-biblioteket

- HashMap er en samling der elementene identifiseres ved en nøkkel / navn
- HashMap<String,Kaniner> alleKaninene = new HashMap<String, Kaniner> ( );
- Metoder: put, get, remove, . . . se Java-bilblioteket

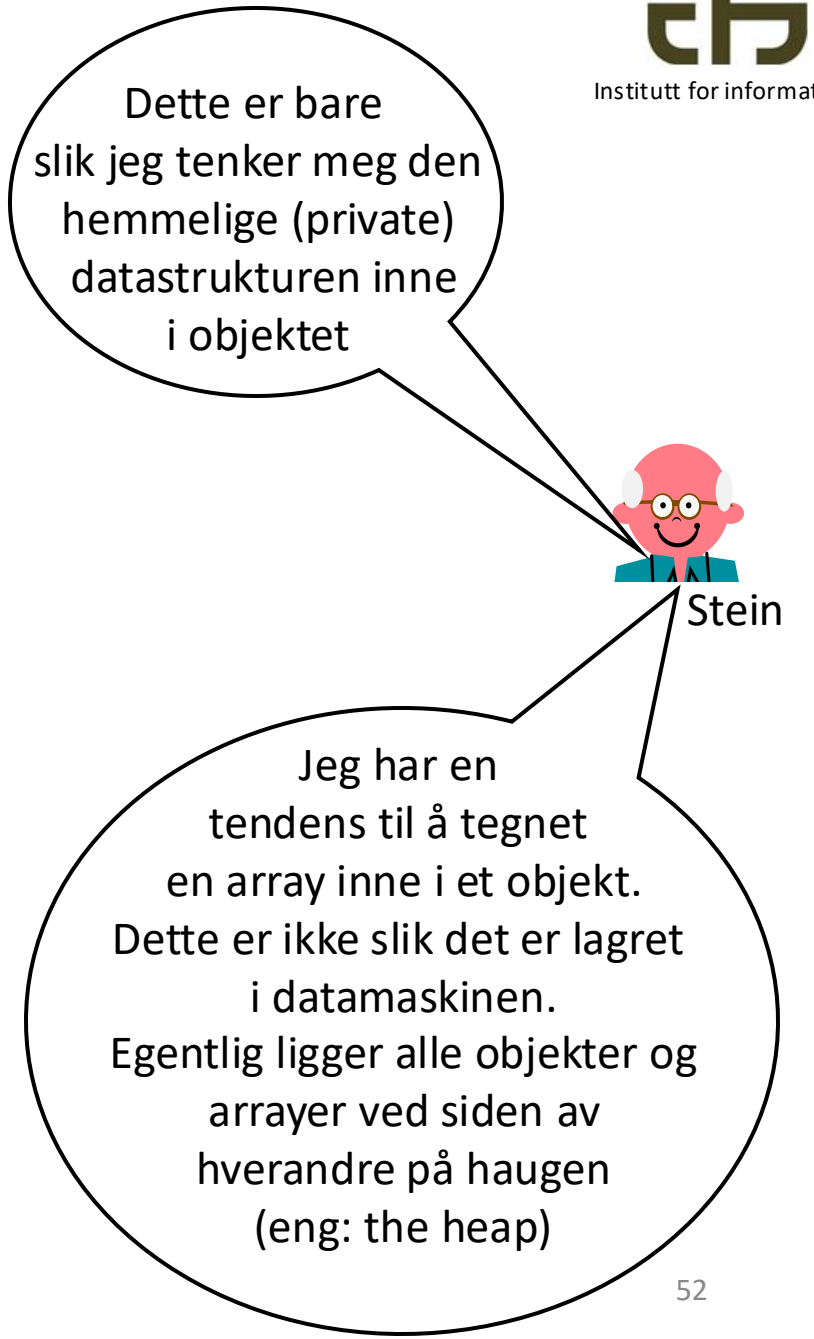
ArrayList

Deklarasjon:

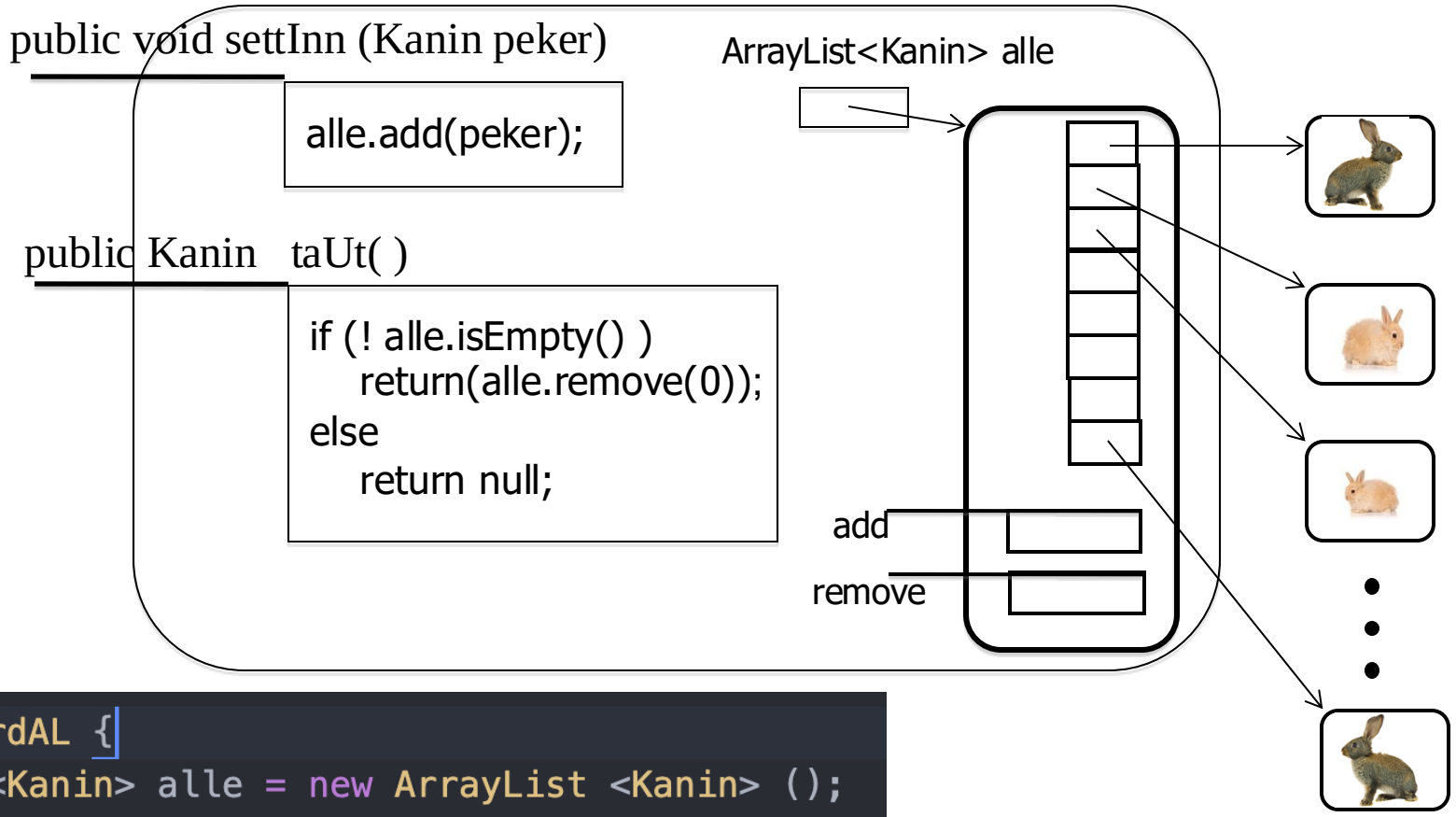
```
ArrayList<Kanin> alle = new ArrayList< >( );
```



Objekt av klassen ArrayList



Objekt av klassen ForenkletKaninGårdAL



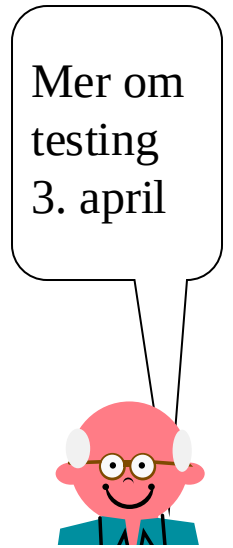

```
class ForenkletKaninGardAL {
    private ArrayList <Kanin> alle = new ArrayList <Kanin> ();
    public void settInn(Kanin peker) {alle.add(peker);}
    public Kanin taUt() {
      if (! alle.isEmpty()) {return (alle.remove(0));}
      else  return null;
    }
}
```

```java
import java.util.ArrayList;

class Kanin{
    private String navn;
    Kanin(String nv) {navn = nv;}
    public String hentNavn ( ) {return navn;}
}

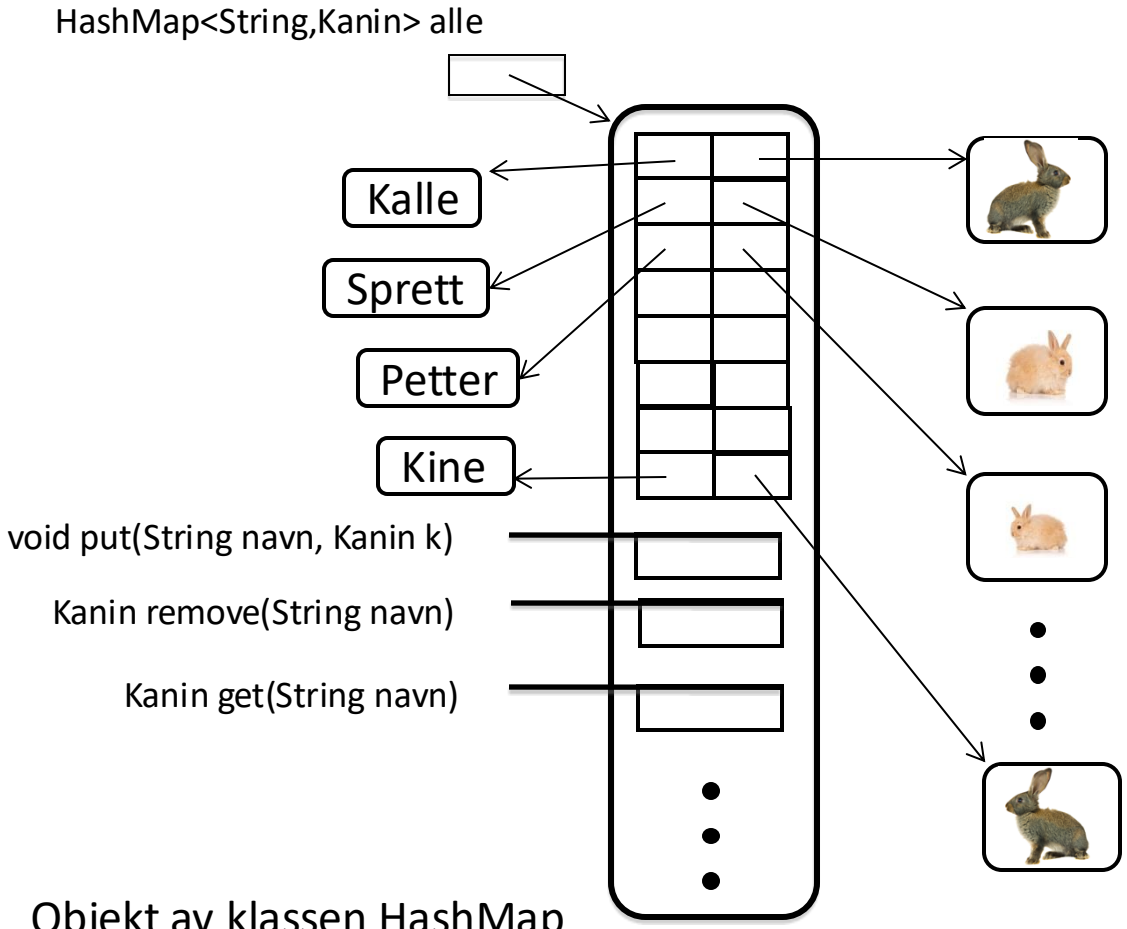
class ForenkletKaninGardAL {
    private ArrayList <Kanin> alle = new ArrayList <Kanin> ();
    public void settInn(Kanin peker) {alle.add(peker);}
    public Kanin taUt() {
      if (! alle.isEmpty()) {return (alle.remove(0));}
      else  return null;
    }
}
```

```
class  KaningardTestArrayList {
  public static void main (String [ ] args) {
    ForenkletKaninGardAL mittKaninbur = new ForenkletKaninGardAL( );
    Kanin kalle = new Kanin("Kalle");
    mittKaninbur.settInn(kalle);
    Kanin sprett = new Kanin("Sprett");
    mittKaninbur.settInn(sprett);
    // test at først inn kommer først ut
    Kanin enKanin = mittKaninbur.taUt();
      test(((enKanin != null) && enKanin.hentNavn().equals("Kalle")),1);
    // test at neste inn nå kommer ut
    enKanin = mittKaninbur.taUt( );
      test(((enKanin != null) && enKanin.hentNavn().equals("Sprett")),2);
    // test at buret nå er tomt
    enKanin = mittKaninbur.taUt();
      test((enKanin == null),3);
  }
   static void test(boolean riktig, int testNr) {
      if (riktig) {
          System.out.println("Riktig test nummer " + testNr);
      }  else {
        System.out.println("Feil test nummer " + testNr);
    }
  }
}
```

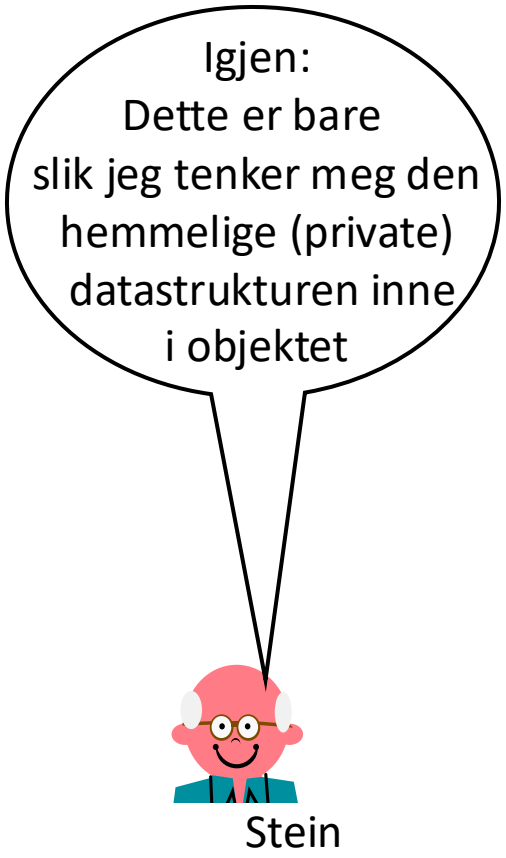


HashMap

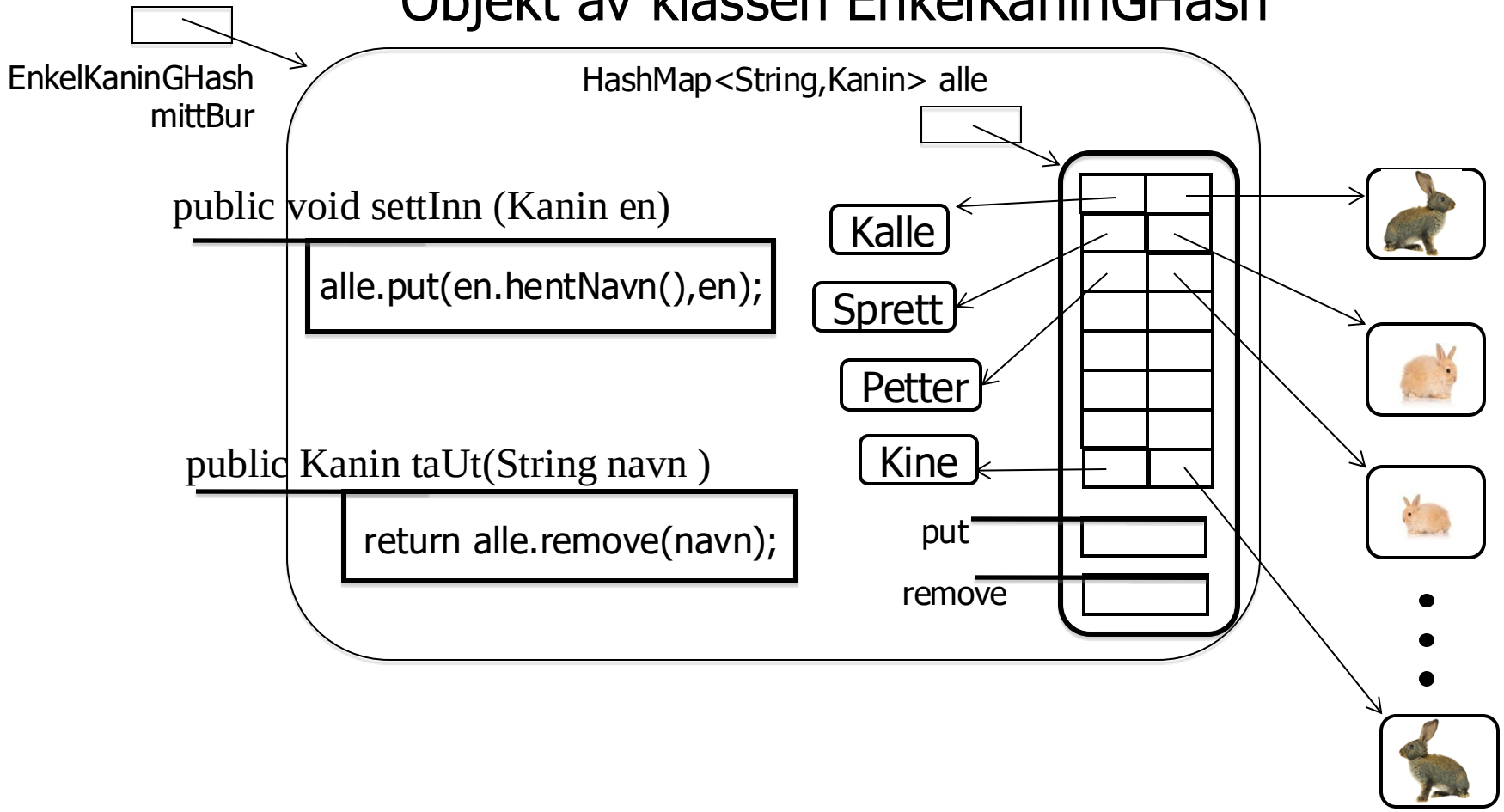
Deklarasjon:
HashMap<String,Kanin> alle = new HashMap<String, Kanin>( );



Objekt av klassen HashMap



Objekt av klassen EnkelKaninGHash




```
class EnkelKaninGHash {
    private HashMap <String, Kanin> alle = new HashMap < >();
    public void settInn(Kanin en) {alle.put(en.hentNavn(), en);}
    public Kanin taUt(String navn ) { return alle.remove(navn); }
}
```

```java
import java.util.HashMap;

class Kanin{
  private String navn;
  Kanin(String nv) {navn = nv;}
  public String hentNavn() {return navn;}
}

class EnkelKaninGHash {
      private int antall = 0;
      private HashMap <String, Kanin> alle = new HashMap <String, Kanin>();
      public void settInn(Kanin peker) {alle.put(peker.hentNavn(), peker);}
      public Kanin taUt(String navn) {return alle.remove(navn); }
}
```

```
class  KaningardTestHash {
  public static void main (String [ ] args) {
    EnkelKaninGHash mittBur = new EnkelKaninGHash();
    Kanin kalle = new Kanin("Kalle");
    mittBur.settInn(kalle);
    Kanin sprett = new Kanin("Sprett");
    mittBur.settInn(sprett);
    // Tester at en som er satt inn kan tas ut
    Kanin enKanin = mittBur.taUt("Kalle");
      test (((enKanin != null) &&  enKanin.hentNavn().equals("Kalle")), 1);
    // Tester at den som er tatt ut virkelig er ute
    enKanin = mittBur.taUt("Kallle");
      test ((enKanin == null),2);
    // . . .
  }
   static void test(boolean riktig, int testNr) {
      if (riktig) {
          System.out.println("Riktig  test nummer " + testNr);
      }  else {
        System.out.println("Feil test nummer " + testNr);
    }
  }
}
```

Enhetstesting - oppsummering

- Først: Tenk og planlegg og tenk og planlegg . . . når du programmerer
- Små deler av programmet må testes før de settes sammen til et større program
- En slik liten del kan være et objekt / en klasse
- Intern testing vs. ekstern testing (se på grensesnitt vs. implementasjon)
- Du bør særlig teste grensetilfeller
  - F.eks: Ta ut av tom beholdere – sette inn i full beholder
- Du bør teste at alle metoder som forandrer tilstanden til objektet virker slik du ønsker – men umulig å teste alle tilfeller
- Bruk observator-metodene til å observere at modifikator-metodene gjør det du ønsker

I dag har vi lært

- Om referanseverdier og referansevariabler
- Om metodinstanser og parameteroverføring
- Om hva en invariant på en datastruktur i et objekt (klasse) er
- At enhetstesting er viktig i programmering
- Mer om å bruke arrayer (innebygget i Java (og i datamaskiner)) og HashMap og ArrayList fra Javabiblioteket.
- Nå kan dere «alt» om klasser og objekter (og typer og variabler)
  - Etter denne uken må dere kunne Java like bra som dere kan Python (Horstmann tom. kap 8)

- Om en uke begynner vi «sakte» på subklasser og arv.

Jo før du blir hengende etter -

jo mer tid har du til å ta igjen det tapte

