Litt om enkle generiske klasser i Java

Versjon 20. februar, 2025

Av Stein Gjessing

Del 1 Enkle referanseverdier (pekere)

Før vi tar fatt på det som er nytt i dette notatet, skal vi repetere litt om klasser og objekter. Anta at vi har skrevet en klasse Hund, som beskriver hunder:

```
class Hund {
    public final String navn;
    public Hund(String n) {
        navn = n;
    }
}
```

Hvis vi oppretter en HashMap der vi ønsker å legge inn hunder (av klassen Hund), og vi ønsker å finne disse igjen basert på navn (av klassen String), kan vi gjøre dette slik:

```
HashMap<String, Hund>  mineHunder = new HashMap<String, Hund>( );
```

Vi sier at String og Hund er *parametre* til klassen HahMap. Klassen HashMap er laget så fleksibel at den kan ta vare på objekter av alle slags klasser, og disse kan finnes igjen basert på objekter av en annen klasse (i dette tilfellet klassen String).

Ser vi i Javas bibliotek, finner vi at overskriften til klassen HashMap ser slik ut

```
class HashMap<K,T>  . . . {
    . . .
}
```

Her er K og T parametre til klassen. Nå har vi snakket om både String og Hund som parametre, og vi har også snakket om K og T som parametre. For å skille disse fra hverandre kaller vi K og T *formelle* parametre, og String og Hund *aktuelle* parametre. Hvis vi trenger å skille disse parametrene fra parametre til metoder, kaller vi de parametrene vi her snakker om for klasse-parametre eller type-parametre. Det siste fordi noen kaller en klasse en type (ADT: Abstract Data Type). Vi sier også at klassen HashMap er en generisk klasse (fordi den er generell, den kan brukes med mange klasser som parameter).

Som kjent skriver vi parametre til metoder inne i vanlige parenteser slik: "(" og ")". Også når det gjelder parametre til metoder skiller vi mellom formelle parametre (i definisjonen av metoden) og aktuelle parametre (uttrykkene på kallstedet). De nye klasse-parametrene skrives inne i andre slags parenteser, nemlig "<" og ">".

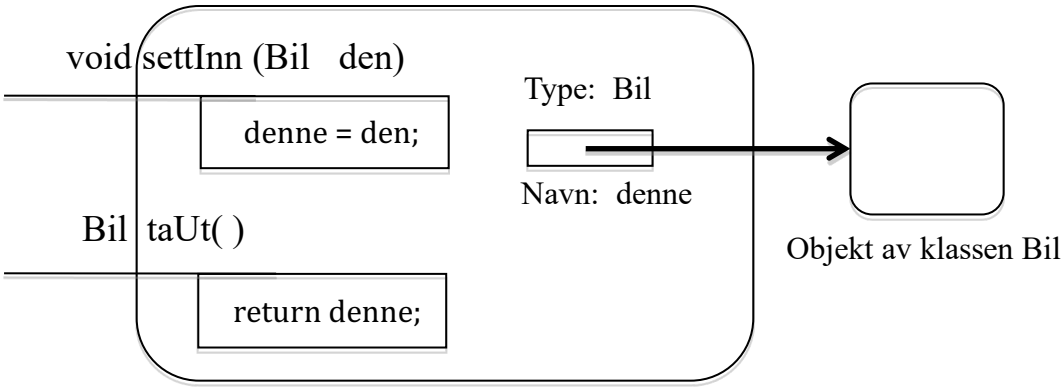
La oss se på et annet eksempel. Anta at vi skal lage et program som behandler biler. Vi må da lage et mønster:

```
class Bil {
      public final String regNr;
      public Bil(String r) {
            regNr = r;
      }
}
```

Vi skal lage en garasje som kan inneholde en bil. Et mønster til en slik garasje kan være:

```
class Garasje {
      private Bil denne;
      public void settInn (Bil den) {denne = den;}
      public Bil taUt ( ) { return denne;}
}
```

Når programmet kjører kan en garasje med en bil i se slik ut:



Objekt av klassen Garasje

Hvis det ikke er noen bil i garasjen når vi tar en bil ut, vil metoden taUt returnere null. Det er OK. Andre ting med denne enkle klassen er derimot ikke så bra, bl.a. er det ikke bra at det er fullt mulig å sette en ny bil inn når det allerede er en bil i garasjen, og det er ikke bra at om vi tar ut en bil, så står den fremdeles også inne i garasjen.

Oppgave 1: Rett opp klassen Garasje slik at objekter av denne klassen oppfører seg bedre når vi prøver å sette en bil inn i en full garasje, og når vi tar ut en bil.

Husk: Overskriften til en metode kalles *signaturen* til metoden. Dette omfatter navnet på metoden og typen til parametrene (i riktig rekkefølge). I mange andre språk, men ikke i Java, er typen til den verdien som returneres (eller void) også

en del av signaturen. Hvis signaturen ellers er den samme i en metode i en subklasse, er det i Java ikke lov å forandre på typen til returverdien.

La oss gå tilbake til hundene, og lage et hundehus. Vi har allerede laget klassen Hund, så mønsteret til hundehuset kan være:

```
class Hundehus {
      private Hund denne;
      public void settInn (Hund den) {denne = den;}
      public Hund taUt ( ) { return denne;}
}
```

Legg merke til hvor like klassene Garasje og Hundehus er.

(At dette hundehuset har de samme feilene og manglene som Garasje-klassen over skal vi ikke bry oss om her).

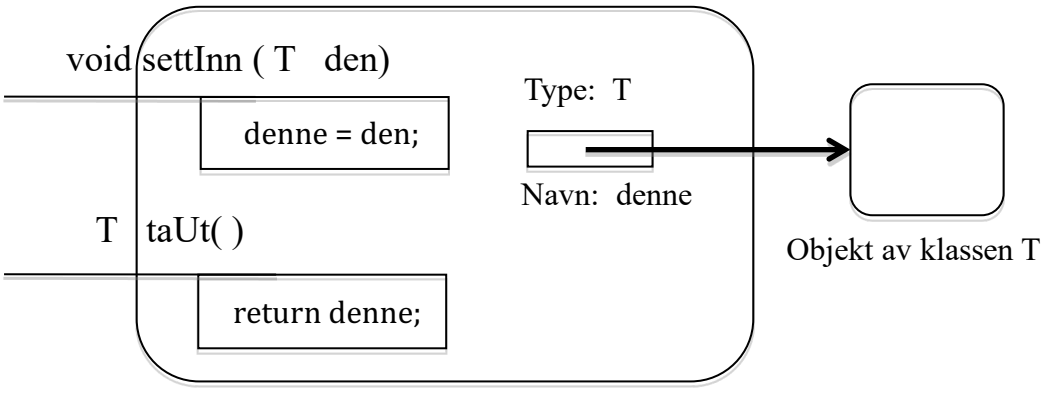
Alt vi har behandlet hittil er kjent stoff. Nå kommer det nye.

Klassene Garasje og Hundehus er ganske like. De brukes begge til å oppbevare en ting, en bil eller en hund. Objekter av disse to klassene er alle beholdere til en ting. Å programmere består ofte i å kjenne igjen felles egenskaper, og deretter lage ett begrep for ting som er nesten like.
Her kommer en klasse som både kan bli en garasje og et hundehus:

```
class BeholderTilEn <T> {
      private T denne;
      public void settInn (T den) {denne = den;}
      public T taUt ( ) { return denne;}
}
```

Sammenlign denne klassen med klassene Garasje og Hundehus!

Vi kan ikke si "new BeholderTilEn<T>( );" så det finnes ikke objekter av denne klassen, men hadde det eksistert slike, ville de kanskje sett slik ut:



Objekt av klassen BeholderTilEn<T> (men slike finnes IKKE)

BeholderTilEn<T> kalles en parametrisert klasse, en klasse med parameter, en generisk klasse eller en generisk type ("kjært barn har mange navn"). BeholderTilEn<T> er et mønster for en beholder som kan ta vare på én ting (ett objekt av klassen T). T kalles den formelle parameteren til klassen. Når vi oppretter et objekt av denne beholderen må vi fortelle hva den skal kunne inneholde:

new BeholderTilEn<Bil> ( )

og vi kan lage et hundehus:

new BeholderTilEn<Hund> ( )

Bil og Hund er da aktuelle parametre til klassen.

For å få tak i et objekt må vi ha en referanse til det. I et program vil vi derfor vanligvis deklarere en referansevariabel først:

BeholderTilEn<Bil> minGarasje;

Deretter kan vi opprette et objekt som minGarasje peker på:

minGarasje = new BeholderTilEn<Bil> ( );

Vi kan også slå disse to setningene sammen til én:

BeholderTilEn<Bil> minGarasje. = new BeholderTilEn<Bil> ( );

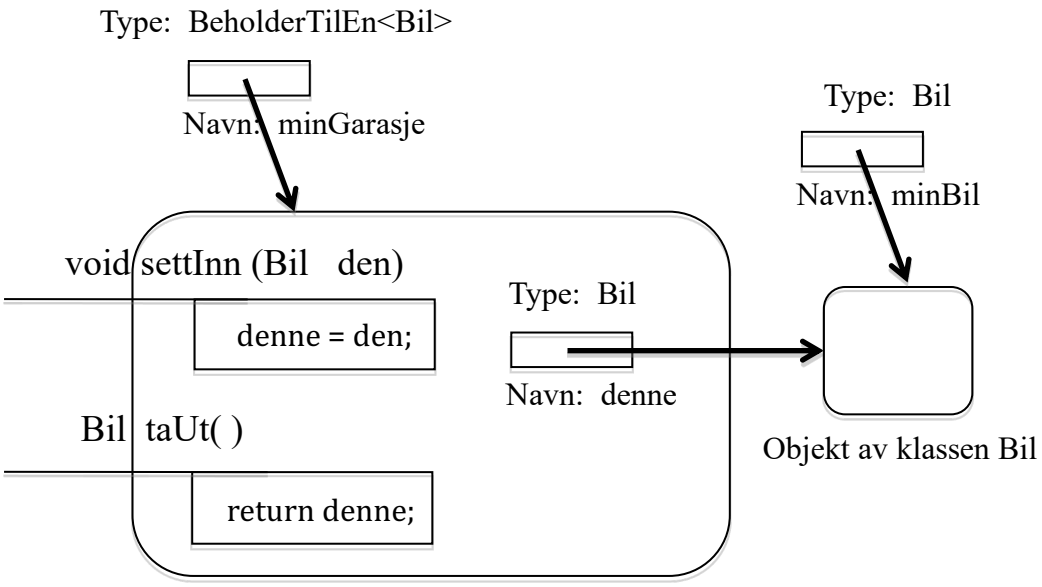
For enkelhets skyld er det lov å utelate Bil på høyre side:

BeholderTilEn<Bil> minGarasje. = new BeholderTilEn< > ( );

Nå kan vi lage en bil og kjøre den inn i garasjen:

```
Bil minBil = new Bil("AB12345");
minGarasje.settInn(minBil);
```

Etter dette er Javaprogrammets datastruktur slik:

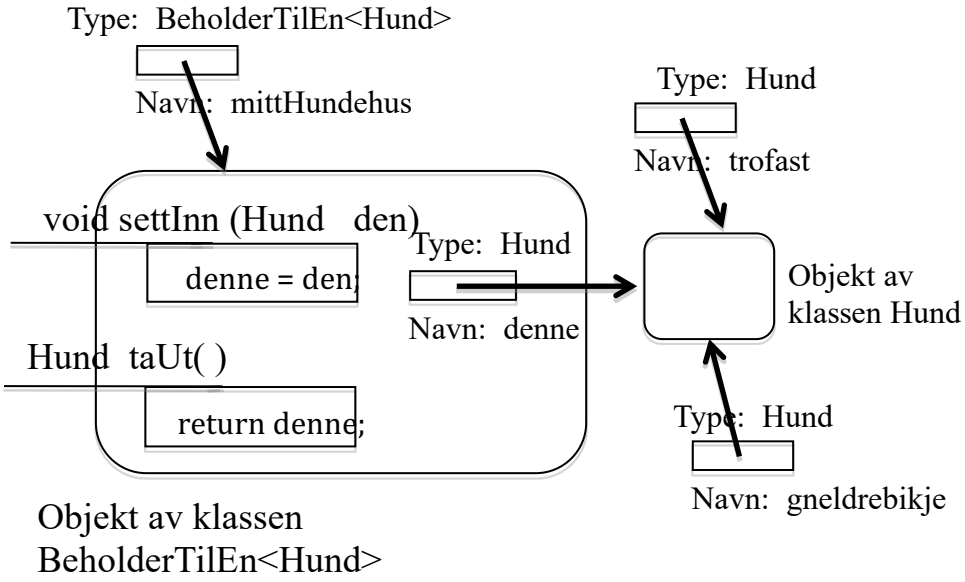


Objekt av klassen BeholderTilEn<Bil>

La oss gjøre det samme med et hundehus og en hund:

```
BeholderTilEn<Hund> mittHundehus = new BeholderTilEn< > ( );
Hund trofast = new Hund("Trofast");
mittHundehus.settInn(trofast);
Hund gneldrebikje;
gneldrebikje = mittHundehus.taUt( );
```

Etter dette er datastrukturen denne (husk det vi sa over om at hunden ble igjen inne i hundehuset selv om vi har tatt den ut):



Objekt av klassen BeholderTilEn<Hund>

Nå har du sett ett eksempel på en enkel generisk klasse. Slike generiske klasser med parametre bruker vi når vi ønsker å lage en "generell" klasse, en klasse som kan virke i samspill med en (eller flere (som i HashMap)) annen klasse, og der vi ønsker å utsette valget av denne andre (de andre) klassene. Vi bruker gjerne klasser med parametre når vi lager verktøy, slik at først i det vi tar i bruk verktøyet (dvs. i det vi lager et objekt) bestemmer vi hvilken klasse som skal være aktuell parameter.

Gjennom dette eksemplet håper jeg du skjønner det mest grunnleggende om hvordan parametre til klasser virker. I IN1010 vil du etter hvert se mange eksempler, og du skal lage mange klasser med parametre selv.

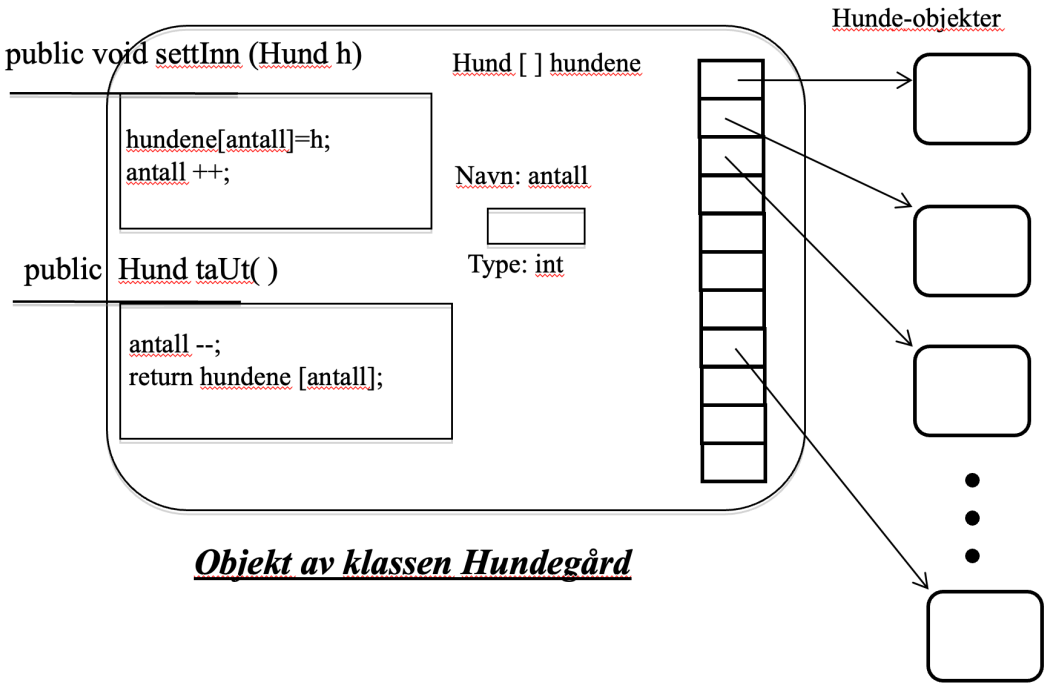
Del 2 Arrayer av referanser

Når vi bruker klasser med parametre sammen med arrayer er det en sær ting i Java vi må passe på. Derfor kommer det her en egen del om dette temaet.

Vi har fra før lært hvordan vi putter referanser til mange objekter inn i en array. La oss lage en klasse som vi kan bruke til å opprette hundegårder med plass til 100 hunder. Dette er kjent stoff:

```
class Hundegård {
    private Hund [ ] hundene = new Hund [100];
    private int antall = 0;
    public void settInn(Hund h) {
        hundene[antall] = h;
        antall ++;
    }
    public Hund taUt( ) {
        antall -- ;
        return hundene[antall];
    }
}
```

Hvis vi lager et objekt av denne klassen og setter inn mange hunder ser Javas datastruktur slik ut:



Oppgave 2: Lag en konstruktør slik at vi kan bestemme størrelsen på hundegården i det vi oppretter en, f.eks. slik: new Hundegård(50);

Legg merke til at når vi tar en hund ut av en slik hundegård, så er dette alltid den hunden som ble satt inn sist. En slik beskrivelse av virkemåten til metodene settInn og taUt kaller vi *semantikken* til disse metodene (i motsetning til bare signaturen). Siden disse metodene er de eneste i klassen, kan vi også kalle dette semantikken til hele klassen. Vi skal andre steder i IN1010 drøfte hvordan vi egentlig ønsker at taUt-metoden skal virke. Dvs. vi skal lage en semantikk som brukerne av klassen er mer fornøyd med, og vi skal programmere en slik klasse slik at den oppfører seg i henhold til denne semantikken.

Skulle vi laget en garasje til mange biler, kunne vi bare byttet ut Hund med Bil i klassen over, og så hadde vi hatt et fult virkende mønster for en stor garasje med plass til 100 biler.

Oppgave 3: Klassen Hundegård over er ikke noe robust program, dvs. den kan feile når den brukes på bestemte måter. Finn to slike feilsituasjoner og modifiser programmet slik at disse feilene ikke oppstår. Må vi forandre signaturen og semantikken til klassen for å få til dette?
(Hint: En feil oppstår når vi skal legge inn en ny hund, og det allerede er 100 hunder der, og en annen feil oppstår når vi tar ut en hund, men det ikke er noen hunder i hundegården.)

Siden vi alt har lært om parametre til klasser, så kan vi prøve å gjøre om hundegården til en parametrisert klasse.

Vi prøver oss med et program, men NB! Dette er IKKE helt riktig:

```
class StorBeholder <T> {
    private T [ ] alle = new T [100];
    private int antall = 0;
    public void settInn(T  det) {
        alle[antall] = det;
        antall ++;
    }
    public T taUt( ) {
        antall -- ;
        return alle[antall];
    }
}
```

Dette ser jo veldig bra ut, og det er ergerlig at Java ikke tillater oss å skrive dette. Grunnen behøver bare de spesielt interesserte å kjenne til: Under utføring av programmet har ikke kjøretidsystemet noen kjennskap til hvilken klasse T er, og derfor klarer den ikke å lage en ny tabell (array) av klassen T (Dette kalles ”type erasure”).

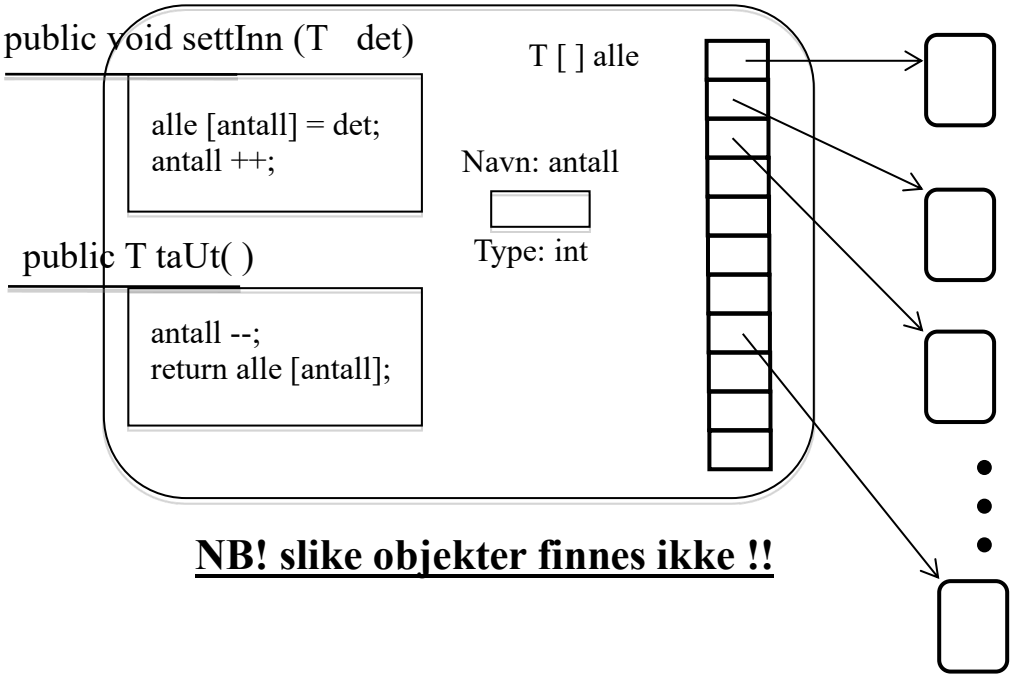
Vi må derfor erstatte linjen som oppretter en tabell, med en mer komplisert linje. Hva denne gjør skal vi ikke bry oss om nå, men bare ta det som en besvergelse, og regne med at virkningen er den samme i programmet under, som den ville ha vært i programmet over, hvis dette var lovlig.
Dette er et riktig program:

```
class StorBeholder <T> {
    @SuppressWarnings("unchecked")
    private T [ ] alle = ( T [ ] )  new Object [100];
    private int antall = 0;

    public void settInn(T  det) {
        alle[antall] = det;
        antall ++;
    }
    public T taUt( ) {
        antall -- ;
        return alle[antall];
    }
}
```

Dette programmet gjør akkurat som det vi ville at programmet over skulle gjøre. Hvis du oversetter dette programmet uten annotasjonen @SuppressWarnings vil du få en advarsel som sier at programmet ditt "uses unchecked or unsafe operations". Dette er pga. «type erasure», dvs. at Javas virtuelle maskin (JVM) ikke har full kontroll over referansers typer når disse er parametre til klasser. De som vil vite mer om dette, kan slå opp i en beskrivelse av Java.

Nedenfor ser du et tenkt objekt av klassen StorBeholder<T> etter at det er lagt inn mange tenkte objekter av type T. NB! T må først bindes til et klassenavn (vi må ha en aktuell parameter) før et objekt kan bli laget.



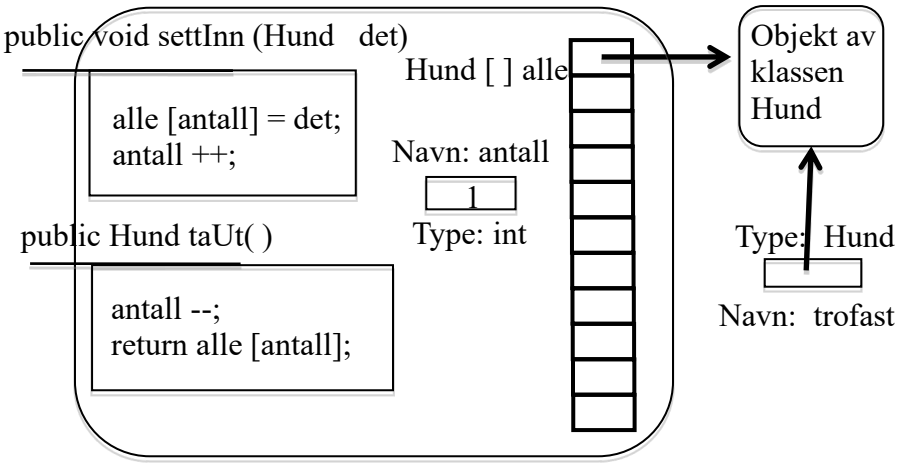
Nå kan vi lage både store hundegårder og fellesgarasjer:

StorBeholder<Hund> minHundegård = new StorBeholder< > ( );
og
StorBeholder<Bil> fellesGarasjen = new StorBeholder< > ( );

Så kan vi si, på samme måte som for et hundehus:

minHundegård.settInn(trofast);

Nedenfor ser du et objekt av klassen new StorBeholder<Hund> ( ) etter at Trofast er lagt inn. Slike objekter finnes virkelig:



Objekt av klassen StorBeholder<Hund>

Nå kan vi ta ut den hunden vi har lagt inn, og la referansevariablen "gneldrebikkje" peke på den hunden vi tar ut:

gneldrebikkje = minHundegard.taUt();

Nå vil begge de to referansevariablene trofast og gneldrebikkje peke på samme hund. Legg merke til at nå har vi virkelig tatt hunden ut av hundegården, for nå vil variablen "antall" inne i objektet ha verdien 0, som betyr at det ikke er noen hunder i gården. Og har du løst oppgave 3 har du en bedre hundegård, dvs. en som bl.a. ikke feiler om du prøver å ta en hund ut av en tom hundegård.

Vi kan selvfølgelig gjøre det samme med bilene i fellesgarasjen, f.eks.:

```
fellesGarasjen.settInn(minBil);
fellesGarasjen.settInn(dinBil);
denneBilen = fellesGarasjen.taUt();
```

Oppgave 4: Skriv en konstruktør til klassen slik at størrelsen på beholderen blir bestemt i det beholderen blir opprettet, f.eks. slik:
StorBeholder<Bil> fellesGarasjen = new StorBeholder<Bil> (500);

SLUTT