Interface i Java

Versjon 14. februar 2025

Av Stein Gjessing

Dette notatet handler om noe av det viktigste ved objektorientering og programmering generelt, nemlig muligheten til å skjule detaljer. Å se på noe uten å bry seg om detaljene kalles gjerne abstraksjon. Å skjule detaljene i en modul eller et objekt kalles innkapsling. I tillegg skal vi se litt på generalisering (generelle egenskaper ved ellers forskjellig objekter).

Fra før vet vi at det finnes private og offentlige (public) egenskaper i et objekt. De offentlige metodene definerer det omverdenen har lov til å gjør med objektet, og derfor kan vi gjerne kalle disse metodene objektets grensesnitt mot omverdenen. Ved å implementere dette grensesnittet (på riktig måte) oppfyller objektet sin *kontrakt* med omverdenen. Datastrukturen som hjelper de offentlige metodene til å gjøre det de skal, er gjerne hemmelig (private). Det er dette vi kalte innkapsling over. Å bare kunne bruke de offentlige metodene gjør at vi har et abstrakt eller overordnet syn på objektet. Tidligere i IN1010 har vi sett hvordan vi enhetstester objekter ved å kalle de offentlige metodene (grensesnittet) til objektene, vi har snakket om disse metodenes signatur og vi har snakket om semantikken til de offentlige metodene. I del 1 av dette notatet skal vi behandle dette igjen i lys av en ny mekanisme, kalt *interface* i Java.

I første omgang er del 1 av dette notatet det klart viktigste. Delene 2, 3 og 4 bygger alle på del 1, men er rimelig uavhengige av hverandre. Del 4 trekker linjene til generiske (parametriserte) klasser som er nærmere beskrevet i et eget notat.

Del 1: Enkle interface - som grensesnitt

I den første delen av dette notatet ser vi på noen enkle eksempler på hvordan interface-begrepet i Java nettopp kan brukes til akkurat det – å beskrive grensesnittet til et objekt.

For å illustrere de programmeringsmetodene og -mekanismene vi skal lære, og spesielt hvordan disse mekanismene materialiserer seg i Java, er det greit å bruke enkle eksempler. Derfor starter vi også i dette notatet med små, enkle programmer.

Når vi har dette mønsteret kan vi lage kaniner:

```
class Kanin{
   final public String navn;
   Kanin(String nv) {navn = nv;}
}
```

Da kan vi også skrive en klasse som kan brukes til å lage et kaninbur med plass til én kanin:

```
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;
   }
}
```

Disse 15 linjene med kode er ikke særlig vanskelige å forstå, men det er allikevel en del detaljer her. Om du skal skrive et program for å håndtere kaninene dine, med flere kaniner og flere bur, må du da skjønne alle disse 15 linjene?
Svaret på dette ledende spørsmålet er "Nei". Det du trenger å vite er navnet på klassen, navnene på (og typen til parametrene til) de offentlige (public) metodene du kan kalle og hva disse metodene gjør (semantikken til metodene). Hvis vi ser bort fra semantikken, trenger vi dette:

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

Her er vi nede i 4 linjer kode, og det er lett å se at klassen Kaninbur har to offentlige metoder, settInn() og taUt(). I tillegg er det lett å se hvilken type returverdiene har, og antallet og typene til parametrene (det som til sammen kalles signaturen til metodene). Som det ble nevnt helt først i dette notatet, er dette offentlige metoder i objekter av klassen Kaninbur, og de definerer derved dette objektets grensesnitt mot omverdenen.

Selv om vi kjenner signaturene, vet vi likevel ikke hva metodene gjør. Før vi implementerer (og før vi bruker) en klasse må vi kjenne semantikken til (public)-metodene, og denne semantikken beskrives med ord. Den private datastrukturen vi bestemmer oss for når vi implementerer klassen må lages hånd i hånd med koden i metodene, slik at metodene oppfyller (oppfører seg i henhold til) semantikken. Etter at metodene er implementert, slik som i de 15 linjene med kode som er hele klassen Kaninbur, kan vi også si at semantikken til klassen er denne Java-koden. Denne semantikken bør da være den samme som semantikken til objektet beskrevet med norsk (eller engelsk).

Alt dette har vi behandlet grundig tidligere i IN1010.

Nå kommer det nye: Java inneholder en mekanisme som bl.a. kan brukes til å kunngjøre navnene på de offentlige metodene i en klasse:

```
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}
```

Ordet "interface" betyr grensesnitt på norsk, og det er nettopp dette vi trenger, vi må vite grensesnittet til de tingene vi lager. Grensesnittet eller interfacet KaninOppbevaring er noe (abstrakt) vi kan oppbevarer kaniner i: Vi kan sette inn en kanin og vi kan ta ut en kanin. Ordet «interface» er et nøkkelord i Java.
Metodene i et interface er alltid public, det er frivillig å skrive dette i interfacet, og vanligvis gjør vi det ikke.

Legg merke til at om vi skriver «class» istedenfor «interface» i koden over så ville vi nesten hatt en klassedeklarasjon. Vi har lov å skrive i Java:

```
abstract class KaninOppbevaring {
    abstract public boolean settInn(Kanin k);
    abstract public Kanin taUt( );
}
```

Dette er nesten det samme som interface-deklarajonen over. Hva forskjellen er skal vi komme tilbake til.

Et annet eksempel på et grensesnitt er en telefon. En gammeldags mobiltelefon uten trykkfølsom skjerm er enklest: Vi kan slå den av og på, og vi kan trykke på tastene.
Et tredje eksempel er en radio. Grensesnittet på en radio er slik at vi kan slå den av og på, vi kan justere volumet, og vi kan finne stasjoner, f.eks. slik:

```
interface Radio {
    public boolean trykkAvPaKnapp( );
    public void justerLyd(int forandring);
    public void setKanal(String kanalNavn);
}
```

Hvordan mobiltelefonen eller radioen er *implementert* innvendig bryr vi oss ikke om når vi bare skal bruke en mobiltelefon eller en radio. På folkemunne kalles det gjerne en ”sort boks” (engelsk: ”black box”). Vi regner med at innmaten gjør det vi forventer når vi trykker på knappene på utsiden, på samme måte som vi skal forvente at et Java-objekt gjør det vi ønsker når vi kaller de offentlige metodene.

I Java betyr dette bl.a. at vi kan dele programmet i (minst) to deler: Den delen som *bruker* objektet, og den delen som *implementerer* objektet. (Dette har vi også gjort tidligere i IN1010 da vi snakket om enhetstesting). I programmet nedenfor bruker vi det nye Java-nøkkelordet «implements» for å vise at et interface blir implementert i en klasse. Den delen som bruker objektet trenger bare vite om metodene i grensesnittet (og hva disse metodene gjør, dvs. semantikken), mens den delen som implementerer objektet må lage og kjenne til alle detaljer (dvs. implementere semantikken). Derfor: Detaljene trenger bare være kjent inne i klassen som implementerer grensesnittet. Her kommer et fullstendig Javaprogram (med en litt annen versjon av klassen Kaninbur):

```
// Definisjonen av kaninene må alle deler av programmet kjenne:
class Kanin{
    public final String navn;
    Kanin(String nv) {navn = nv;}
}

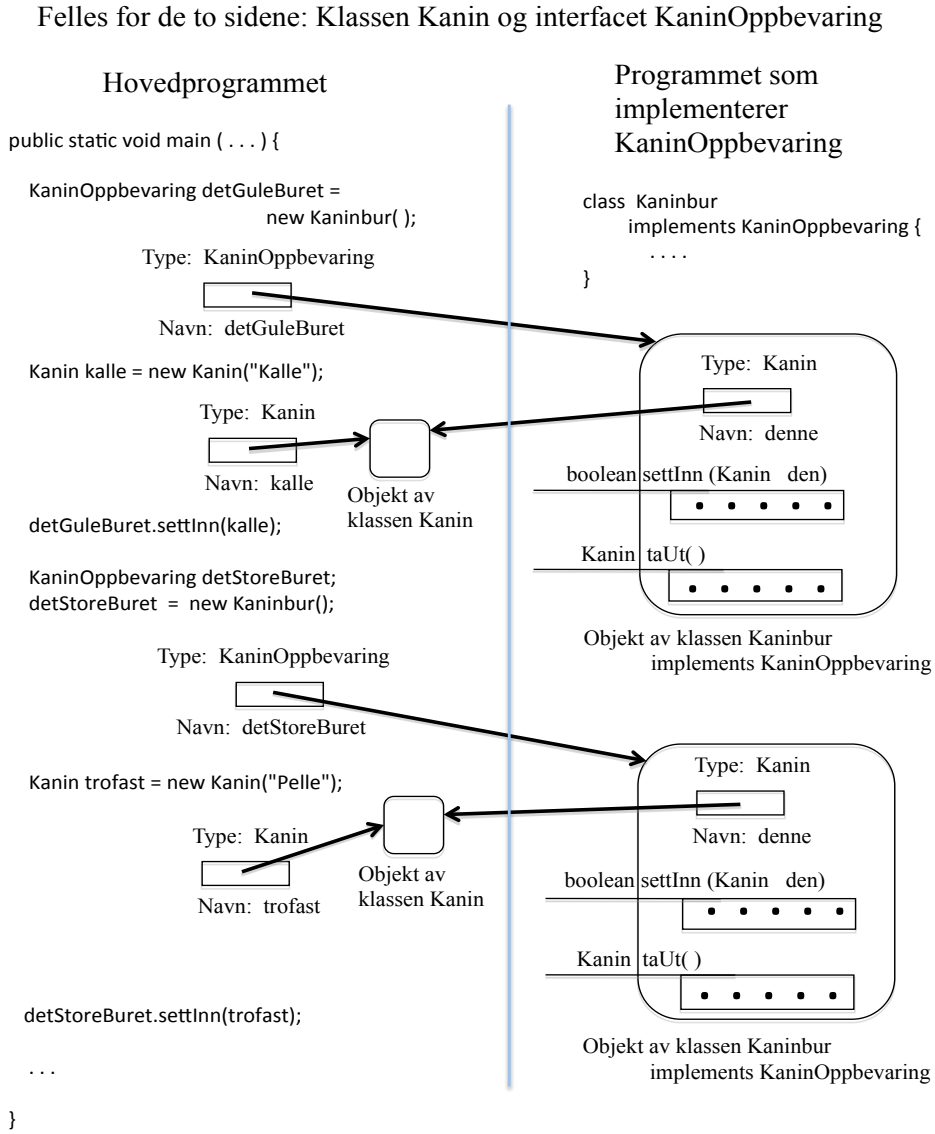
// Grensesnittet til kaninburene må også alle deler kjenne:
interface KaninOppbevaring {
    public boolean settInn(Kanin k);
    public Kanin taUt( );
}

// Her kommer programmet som er en implementasjon av KaninOppbevaring:
class Kaninbur implements KaninOppbevaring {
    private Kanin denne = null;
    @Override
    public boolean settInn(Kanin k) {
        if (denne == null)  {
            denne = k;
            return true;
        }
        else return false;
    }
    @Override
    public Kanin taUt( ) {
        Kanin k = denne;
        denne = null;
        return k;
    }
}

// Her kommer et lite hovedprogram som bruker kaninburet:
class BrukKaninbur {
    public static void main (String [ ] args) {
        KaninOppbevaring detGuleBuret = new Kaninbur( );
        Kanin kalle = new Kanin("Kalle");
        detGuleBuret.settInn(kalle);
        KaninOppbevaring  detStoreBuret;
        detStoreBuret = new Kaninbur();
        Kanin trofast = new Kanin("Pelle");
        detStoreBuret.settInn(trofast);

        // bytt kaninene i de to burene:
        Kanin forstUt = detGuleBuret.taUt( );
        Kanin utAvStore = detStoreBuret.taUt( );
        detStoreBuret.settInn(forstUt);
        detGuleBuret.settInn(utAvStore);
    }
}
```

Figuren under illustrerer de to delene av programmet når det kjører. Hovedprogrammet med main-metoden er til venstre og klassen Kaninbur og objektene av denne klassen er til høyre på figuren. Den vertikale streken er *kontrakten* som Kaninbur-objektet (og dets metoder) har med programmet på venstre side av linjen. Legg merke til at inne i main-metoden er det tegnet en del variabler og objekter. Variablene og objektene er tegnet omtrent der de oppstår når main-metoden utføres.



Når du programmerer bør du ofte tenke på en slik to-delingen. Fordelen er at når du programmerer en av sidene skal du IKKE bry deg om detaljene på den andre siden. Som beskrevet før: Det er bare klassen Kanin og interfacet KaninOppbevaring som er kjent for begge sider. Det eneste unntaket er når hovedprogrammet oppretter et objekt. Da må det vite navnet på klassen som implementerer objektet. Det er måter å unngå dette på, bl.a. ved å lage en såkalt "objektfabrikk" (engelsk: object factory).

Som også skrevet før: Grensesnittet KaninOppbevaring (med signatur og semantikk til metodene) er *kontrakten* mellom venstre og høyre side på figuren. Grensesnittet (interfacet) beskriver de *tjenestene* objektene som implementerer det tilbyr.

Legg også merke til at selv om metoden settInn returnerer en sannhetsverdi (Boolsk verdi), bruker vi ikke denne i dette programmet. Det er OK i Java å ikke gjøre noe med returverdien fra en metode.

Oppgave 1: Modifiser programmet over slik at de Boolske verdiene fra settInn blir tatt i mot av main-programmet, og programmet skriver ut om det gikk bra eller ikke å sette kaninene inn i burene. Utvid programmet slik at det prøver å sette to kaniner inn i samme bur, og skriv ut at dette ikke gikk så bra.

Legg spesielt merke til typen til referansene i hovedprogrammet. De to Kanin-objektene pekes på av to referansevariabler av typen Kanin, så her er alt som det pleier. Det vi har lært hittil om Java er at alle objekter må pekes på av variabler som har den samme typen (samme klassen) som objektet selv, eller samme typen som en superklasse til objektet. Men se på de to Kaninbur-objektene, de pekes på av to referansevariabler av typen KaninOppbevaring. Dette er et interface, og dette er nytt. Men det er helt i tråd med det vi har lært om subklasser og tanken om innkapsling og skjuling av detaljer. På venstre side er vi ikke interessert i alle detaljer i implementasjonen av KaninOppbevaring. Vi er bare interessert i de to metodene settInn og taUt, slik de er definert av interfacet KaninOppbevaring. Bruker vi en referansevariabel av typen KaninOppbevaring, er det nettopp bare egenskapene som er definert i interfacet KaninOppbevaring vi har tilgang til i de to objektene i den andre enden av pila. Klassen Kaninbur kunne gjerne hatt mange flere metoder, men disse hadde vi eventuelt ikke hatt tilgang til gjennom referansene av typen KaninOppbevaring, for KaninOppbevaring-grensesnittet definerer bare de to metodene settInn og taUt. Derfor, ved å bruke referanser av typen KaninOppbevaring, blir vi hindret i å bruke eventuelle andre metoder i objektene av klassen Kaninbur. Dette er nettopp hensikten: **Kjenn til så lite som mulig (men nok) om de objektene du bruker!** Da slipper du å tenke på unødvendige detaljer og du gjør minst mulig skade (fordi kompilatoren hindrer deg i å kalle metoder du ikke skal kalle).

Et interface er noe vi ikke ser så mye spor av under kjøringen av programmet. Men, navnet på interfacet bruker vi som typen til referansevariabler og vi kan teste (ved instanceof) om et objekt har egenskapene definert av et interface. Husk at et interface bare inneholder metoder. Klasser som implementerer interfacet (dvs. implementerer metodene i interfacet), og objekter av disse klassene ser vi som Java datastrukturer under kjøring. Vi ser interface-metodene i disse objektene, men det viktigste med interfacet er skjuling av detaljer og den ansvars-oppdelingen vi har når vi utvikler programmet.

For å summere opp: Et interface er en slags klasse men med bare metode-signaturer (metode-overskrifter). Istedenfor "class" står det "interface" foran, og istedenfor innmat i metodene (som står mellom ”{” og ”}”), står tegnet ”;” etter metodesignaturene. En klasse kan implementere et interface ved å skrive «implements» og så navnet på interfacet, etter klassenavnet. Denne klassen MÅ da inneholde metodene i interfacet (som offentlige metoder), og alle metodene må ha samme signatur som i interfacet (og også samme returtype).

Hvis vi ønsker at en klasse ikke skal implementere alle metodene i interfacet må vi deklarere klassen som **abstract.**

Viktig: Referanser med typen til et interface kan bare peke på objekter av klasser som implementerer dette interfacet. Gjennom disse referansene har vi bare tilgang til de metodene i objektet som er definert av interfacet. Ønsker du at programmet skal ha tilgang til andre egenskaper i objektet må programmet først typekonvertere (cast, class-cast) til en referansetype som gir tilgang til de ønskede egenskapene. Husk at det kan være lurt å teste med **instanceof** før du typekonverterer.
Polymorfi gjelder også for metoder definert i et interface, og vi kan (og bør) til og med annotere implementasjon av en interface-metode med @Override.

Del 2. Multippel arv

Noen ganger trenger en klasse å bygge på egenskaper fra flere superklasser. Men dette er ikke mulig i Java. Av forskjellige grunner (som det går for langt å drøfte i dette notatet) er det i Java bare lov å arve egenskaper fra én superklasse. Men det finnes en løsning i Java som baserer seg på interface. I tillegg til å arve fra sin superklasse, er det mulig for en klasse å arve egenskapene til et eller flere interface.

La oss igjen se på kaninbureksemplet. Du skal nå kjøpe deg en ny kanin, og ekspeditøren i butikken sier at denne nye kaninen trenger mye lys om dagen. Han anbefaler deg derfor å kjøpe et kaninbur med lys. Det finnes et interface som beskriver rom der lyset kan slås av og på:

```
interface Lys {
    void tennLyset ( );
    void slukkLyset ( );
}
```

Basert på dette selger han deg et kaninbur som både har arvet de vanlige kaninburegenskapene, men også har arvet lys-egenskapene:

```
class KaninburMedLys extends Kaninbur implements Lys {
    private boolean lys = false;
    @Override
    public void tennLyset ( )  {lys = true;}
    @Override
    public void slukkLyset ( ) {lys = false;}
}
```

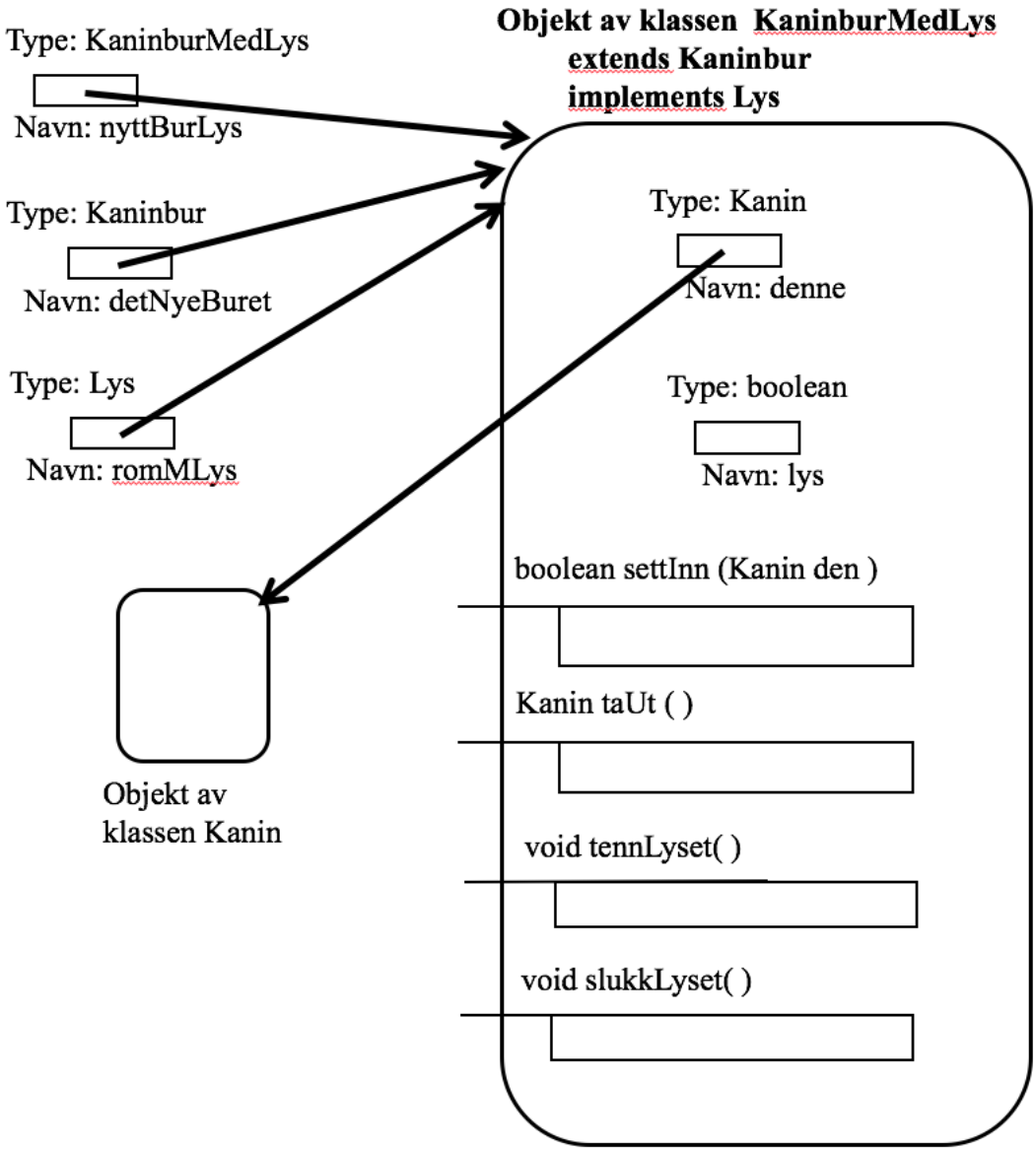
```
// Her er et hovedprogram med en feil i siste linje:

public static void main (String [ ] args) {
    KaninburMedLys nyttBurLys = new KaninburMedLys ( );
    Kanin pelle = new Kanin("Pelle");
    nyttBurLys.settInn(pelle);
    nyttBurLys.tennLyset( );
    Lys romMLys = nyttBurLys;
    romMLys.slukkLyset();
    romMLys.tennLyset();
    Kaninbur  detNyeBuret;
    detNyeBuret = nyttBurLys;
    Kanin trofast;
    trofast = detNyeBuret.taUt( );
    detNyeBuret.slukkLyset( );
}
```

Hvis du oversetter dette programmet får du feilmeldingen: "cannot find symbol : method slukkLyset()". På neste side ser du en datastruktur under kjøring av dette programmet (hvis vi tar vekk den siste linjen). Hvis vi følger referansen detNyeBuret kommer vi til et objekt som inneholder metoden slukkLyset, så hvorfor sier kompilatoren at den ikke finner dette symbolet? Grunnen er at pekeren detNyeBuret er av typen Kaninbur, og den klassen inneholder bare de to metodene settInn og taUt. Dvs. når du kommer til et objekt via referansen detNyeBuret så har du bare tilgang til Kaninbur-metodene. Alle andre detaljer ved dette objektet er ikke synlig via en referanse av denne typen. Vi sier gjerne at vi ser på objektet med Kaninbur-*briller*.

Dette er det samme prinsippet som vi har når en referanse av en superklassetype ikke får tilgang til en metode som er definert i en subklasse (om ikke metoden er polymorf).

I tillegg til å kalle det "briller", kaller vi også ofte de forskjellige måtene vi kan betrakte et objekt på for objektets forskjellige *roller*.
Det store objektet på neste side kan spille tre roller. Det kan spille sin originale KaninburMedLys-rolle. Da er alle de offentlige metodene i objektet med, og vi får tilgang til alle metodene eller operasjonene i rollen via referanser av typen KaninburMedLys. Men objektet kan også spille rollen Kaninbur. Bare metodene i denne klassen er med i denne rollen, dvs. settInn() og taUt(). Ved hjelp av referansevariablen detNyeBuret ser vi bare disse metodene, og objektet spiller da rollen KaninOppbevaring. Til slutt kan det spille rollen Lys. Gjennom referansevariablen romMLys synes bare de to Lys-metodene tennLyset() og slukkLyset().



Oppgave 2: Tegn opp klassehierarkiet til klassen KaninburMedLys. Marker interfacer med *kursiv* skrift. Bruk gjerne også "*kursive*" bokser til å representere dem. Ta med klassen Object øverst i klassehierarkiet.

Før vi avslutter denne delen må vi nevne hva forskjellen på et interface og en abstrakt klasse med abstrakte metoder er. Én forskjell er at en klasse bare kan arve fra én superklasse, men en klasse kan arve fra (implementere) flere interface. En annen forskjell er at i en abstrakt klasse kan det være deklarert variabler og metoder med innmat, men et interface er helt fri for slikt.

Del 3. Enkle interface - som felles oppførsel

Vi skal i dette kapittelet se hvordan interface-mekanismen i Java kan brukes til å definere at forskjellige ting kan ha noe felles oppførsel. Det vil si at vi skal se på hvordan forskjellige klasser kan implementere det samme interfacet, og på den måten spille den samme rollen (i tillegg til den eller de rollene objektene har basert på resten av egenskapene i klassene sine og som den har arvet via det vanlige klassehierarkiet). Vi har i noen grad allerede gjort dette i del 2 (multippel arv), så det vi skriver nedenfor blir på mange måter bare flere eksempler på det samme.
Hos Norsk Tipping er det f.eks. slik at både personer og hester kan delta i konkurranser med tidtaking. Vi kan skille ut de spesielle konkurranseegenskapene i et eget konkurranse-interface:

```
interface Konkurransedeltager {
    public int startnummer ( );
    public int resultatnummer ( );
    public float tid( );
}
```

Da kan vi skissere deklarasjoner av mønstre for hhv. personer og hester slik:

```
class Idrettsutøver implements Konkurransedeltager {
    private String navn;
    . . .
    @Override
    public int startnummer( ) {. . .  return . . . }
    @Override
    public int resultatnummer( ) { . . . return . . . }
    @Override
    public float tid( ) {return . . . }
}

class Veddeløpshest implements Konkurransedeltager {
    private int hestNr;
    private String eier;
    . . .
    @Override
    public int startnummer( ) { . . . return . . . }
    @Override
    public int resultatnummer( ) { . . . return . . . }
    @Override
    public float tid( ) { . . . return . . . }
}
```

I resten av dette kapittelet skal vi se nærmere og mer detaljert på et annet eksempel. Norge har toll eller importavgift på en rekke forskjellige varer. I programmet til tollvesenet er det derfor rimelig at det finnes et interface:

```
interface Importavgiftspliktig {
    public int avgift ( );
    public int pris ( );
}
```

Alle varer som det er importavgift på må implementere dette grensesnittet slik at vi kan spørre varene om hva deres avgift er og hvor mye varen koster i utsalg. La oss se på to eksempler. Biler har importavgift som egentlig er en funksjon av flere egenskaper ved bilen, så som pris, type motor, vekt, ytelse og utslipp. Her gjør vi det enkelt og sier at alle Biler har 100% importavgift:

```
class Bil implements Importavgiftspliktig {
    public final String registNr;
    private String eier;
    private int importprisen;
    Bil (String reg, int imp) {
        registNr = reg;
        importprisen = imp;
    }
    public void nyEier (String ei) {eier = ei;}
    public String regNr ( ) {return registNr;}
    @Override
    public int pris ( ) {return (importprisen + avgift ( ) ); }
    @Override
    public int avgift ( ) { return importprisen; }
}
```

For å beskytte norske oster er det kraftig importavgift på faste (og halvfaste) oster. La oss si at avgiften er 277%:

```
class FastOst implements Importavgiftspliktig {
    private int fettpr;
    private String nvn;
    private long kg;
    private int importprisPerKg;
    FastOst (int ft, String navn, int imp, long kg) {
        fettpr = ft; nvn = navn; importprisPerKg = imp; this.kg = kg;
    }
    public int fettprosent ( ) {return fettpr;}
    public String navn ( ) {return nvn;}
    @Override
    public int pris ( ) { return (Math.round(importprisPerKg * kg) + avgift ( ) ); }
    @Override
    public int avgift ( ) { return Math.round(importprisPerKg  * kg * 2.77F); }
}
```

I tollvesenets program er det en stor tabell (en array eller en annen datastruktur) som peker ut (refererer) alle varer på lager i øyeblikket, og som det er importavgift på. Hvis det er en array kan den være deklarert slik:

```
Importavgiftspliktig [ ] alleImpPliktVarer = new Importavgiftspliktig [1000];
```

Her har vi altså en hel array av referanser der hver referanse er av en interface-type, nemlig Importavgiftspliktig.

Skal vi teste programmet vårt kan vi legge inn noen varer:

```
Importavgiftspliktig denne;
int antall = 0;
denne = new Bil ("BD12345", 150000);
alleImpPliktVarer[antall] = denne;
antall ++;
denne = new FastOst(27,"Edamer",30,2000);
alleImpPliktVarer[antall] = denne;
antall ++;
```

Når vi ønsker å gå gjennom alle importpliktige varene på toll-lageret kan vi lage en for-løkke, og så f.eks. skrive ut summen av alle avgifter:

```
int totalAvgifter = 0;
for (Importavgiftspliktig den: alleImpPliktVarer)
      if (den != null) totalAvgifter+= den.avgift( );
System.out.println("Sum av avgifter: " + totalAvgifter);
```

Lærdommen i dette kapittelet er at vi kan implementere det samme interfacet i forskjellige klasser. Objektene av disse klassene får da litt lik oppførsel, dvs. de kan alle spille rollen definert av det interfacet de implementerer. Når vi i et slikt tilfelle har en array (eller en klasse fra Java-biblioteket) med referanser av denne interface-typen, vet vi bare at objektene som refereres spiller denne rollen, og vi kan følgelig bare kalle metodene i dette interfacet (hvis vi da ikke typekonverterer (caster)).

Del 4. Om å bruke flere grensesnitt samtidig, mer om multippel arv.

I det siste eksemplet i del 1 lagde vi et objekt av klassen KaninburMedLys, og hadde to pekere til dette objektet, en av typen KaninOppbevaring og en av typen KaninburMedLys. Gjennom referanser av den siste typen kan vi se alle de offentlige metodene i objektet, men gjennom referanser av typen KaninOppbevaring ser vi bare de metodene som er definert i grensesnittet KaninOppbevaring. Vi sier at dette objektet kan spille to roller, en rolle som KaninOppbevaring, og en rolle som KaninburMedLys.

I Java kan vi lage objekter som spiller vilkårlig mange roller. La oss se nærmere på eksemplet med radioen som vi så vidt så på i del 1. Her er radio-interfacet igjen:

```
interface Radio {
    public boolean trykkAvPaKnapp( );
    public void justerLyd(int forandring);
    public void setKanal(String kanalNavn);
}
```

I tillegg definerer vi at en StrømmeSpiller kan slås av og på og justeres lyden på (som på en radio). Som enda et tillegg kan vi finne fram album og enkeltsanger:

```
interface StrømmeSpiller {
    public boolean trykkAvPaKnapp( );
    public void justerLyd(int forandring);
    public void setAlbum(String albumnavn);
    public void setSang(String sang);
}
```

Så skal vi lage en kombinert radio og StrømmeSpiller. Den må da implementere begge disse to interfacene. Vi kaller det vi lager en KombiSpiller:

```
class KombiSpiller implements Radio, StrømmeSpiller {
    private int vekten = 125;
    private boolean av = true;
    private int lydstyrke = 0;
    @Override
    public boolean trykkAvPaKnapp( ) {av = !av; return av;}
    @Override
    public void justerLyd(int forandring) {lydstyrke += forandring;}
    @Override
    public void setKanal(String kanalnavn) {  }
    @Override
    public void setAlbum(String albumnavn) {  }
    @Override
    public void setSang(String sang) {  }
    @Override
    public int vekt( ) {return vekten;}
}
```

Oppgave 3: Innmaten i klassen KombiSpiller over er svært ufullstendig. Spesifiser og implementer en god datastruktur i klassen KombiSpiller.

Her skal vi bare se på KombiSpiller-klassen utenifra. Hvis vi et sted i programmet vårt sier:

```
    new KombiSpiller( )
```

får vi tre ting: Vi får en KombiSpiller, vi får en StrømmeSpiller og vi får en Radio, alt i ett og samme objekt. Dette objektet sier vi kan spille tre roller: KombiSpiller-rollen, StrømmeSpiller-rollen og Radio-rollen.

Anta at disse 6 kodelinjene finnes (f.eks. i en main-metoden):

```
KombiSpiller minDings;
Radio minLytter;
StrømmeSpiller minSpiller;
minDings = new KombiSpiller( );
minLytter = minDings;
minSpiller = minDings;
```

I figuren på neste side ser du datastrukturen som er et resultat av dette. Referansevariablen minDings er av typen KombiSpiller. Det betyr at vi gjennom den kan få fatt i alle (de offentlige) egenskapene til objektet (siden objektet er av klassen KombiSpiller). Vi kan f.eks. si minDings.setKanal( ) og minDings.vekt( ).

Gjennom referansene minLytter og minSpiller har vi ikke tilgang til alle objektets egenskaper. Siden minLytter er av typen Radio, har vi bare adgang til Radio-egenskapene til objektet og gjennom minSpiller har vi bare tilgang til StrømmeSpiller-egenskapene. Vi kan f.eks. si: minLytter. justerLyd( ), minLytter.setKanal( ), minSpiller.justerLyd( ) og MinSpiller.setAlbum( ). Vi kan derimot f.eks. ikke si minLytter.setSang( ), ikke minLytter.vekt( ) og ikke minSpiller.setKanal( ). Regelen er at gjennom en referanse av en gitt type (klasse- eller interface-navn) kan vi bare se de egenskapene til objektet som er definert i denne klassen eller dette interfacet.

Oppgave 4: Hvis du endrer siste linje i programmet over til minSpiller = minLytter; blir de tre siste linjene seende slik ut:

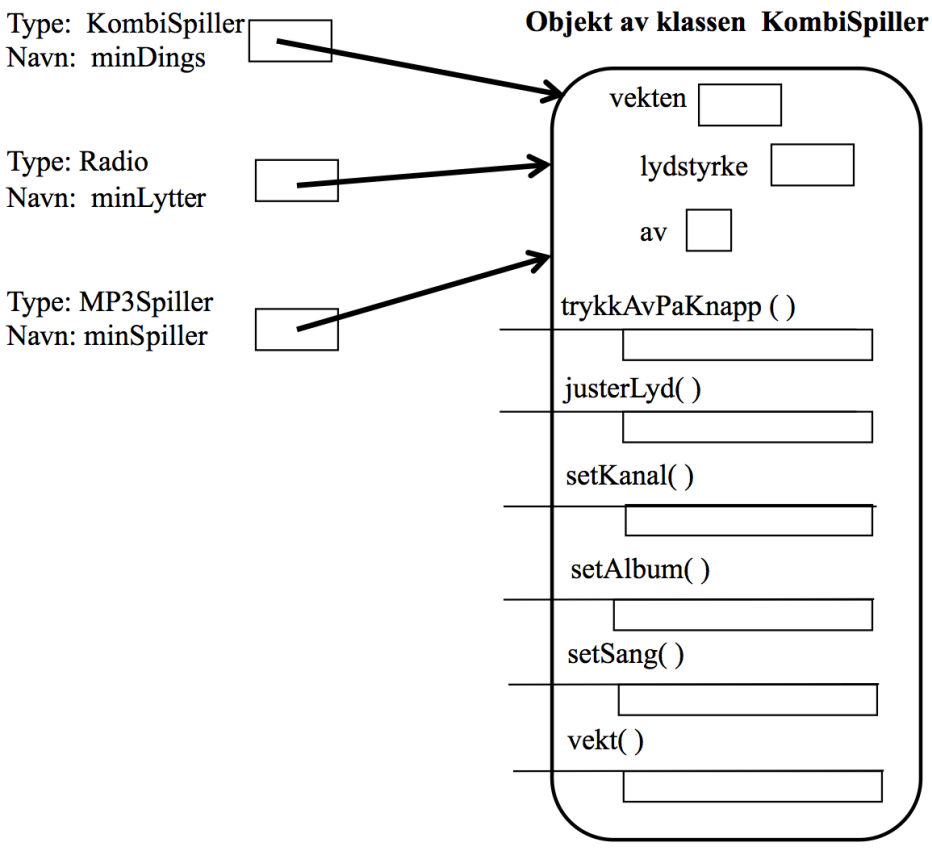
```
minDings = new KombiSpiller( );     // ingen forandring
minLytter = minDings;               // ingen forandring
minSpiller = minLytter;
```

Dette liker ikke kompilatoren. Hvorfor? Jo fordi minLyter i utgangspunktet kan referere et hvilket som helst objekt bare det implementerer Radio-interfacet.

På samme måte som med subklasser kan vi teste og typekonvertere:

```
minSpiller = (StrømmeSpiller) minLytter;
```

Da vil kjøretidsystemet sjekke om det objektet som minLytter refererer spiller rollen StrømmeSpiller. Hvis detter er OK (som det er her) vil tilordningen vær vellykket, og programmet eksekverer videre. Hvis ikke vil programmet terminere øyeblikkelig. Også her kan det være lurt å sjekke med **instanceof** før du prøver å typekonvertere.



Del 5. Generiske grensesnitt (Grensesnitt med parametre)

Dette kapitlet bør sees i sammenheng med notatet om generiske klasser (også kalt klasser med parametre og parametriserte klasser). Antagelig kan det lønne seg å lese det notatet før du leser videre her.

På samme måte som for klasser kan vi ha interface med parametre, også kalt generiske interface. Hvis du har lest notatet om generiske klasser, så vil du se at kaninburene i del 1 ligner veldig på garasjene og hundehusene vi så der. La oss hoppe i det og se på et interface med én parameter:

```
interface Beholder<E> {
        public boolean settInn (E den);
        public E taUt ( );
}
```

Her kommer tre klasser som alle implementerer dette grensesnittet på hver sin måte:

```
class EnkelBeholderTilEn <E> implements Beholder<E> {
        private E denne;
        @Override
        public boolean settInn (E den) {denne = den;  return true;}
        @Override
        public E taUt ( ) {return denne;}
}
```

```
class BedreBeholderTilEn<E> implements Beholder<E> {
    private E denne = null;
    @Override
    public boolean settInn(E  den) {
                if (denne == null)  {
                        denne = den;
                        return true;
                }
                else return false;
    }
    @Override
    public E taUt( ) {
                E den = denne;
                denne = null;
                return den;
    }
    public boolean erTomt( ) {return denne == null;}
}

class StorBeholder<E> implements Beholder<E> {
    private E [ ] alle = (E [ ] ) new Object [100];
    private int antall = 0;
    @Override
    public boolean settInn(E  det) {
            if (antall ==100) return false;
        alle[antall] = det;
        antall ++;
        return true;
    }
    @Override
    public E taUt( ) {
        if(antall == 0) return null;
        antall -- ;
        return alle[antall];
    }
}
```

Legg merke til at i alle disse eksemplene er den første E-en i klassen den definerende forekomsten av den formelle parameteren. Vi må selv finne på navn på de formelle klasse-parametrene. Vi har brukt E her istedenfor T bare for å vise at alle navn på klasseparametre er mulig. Javas konvensjon er at en formell klasseparameter skal være én store bokstav, og at T står for Type, E for Element, V for Value og K for Key. Navnet gir bare leseren et hint om hva meningen med parameteren er.

Oppgave 5: I klassen StorBeholder er tallet 100 brukt to ganger. Dette er ikke god programmeringspraksis. Deklarer en konstant (som du setter til 100), og erstatt tallet 100 med denne konstanten.

Oppgave 6: Skriv en konstruktør i klassen StorBeholder som har verdien på denne konstanten (som du deklarerte i oppgave 4) som parameter.
I tillegg til class Kanin, regner vi med at vi har class Hund og class Bil fra notatet om generiske klasser. Vi kan da lage et lite program:

```
BedreBeholderTilEn<Hund>   mittLilleHundehus;
mittLilleHundehus  =  new BedreBeholderTilEn<Hund>( );
Hund fido =  new Hund("Fido");
mittLilleHundehus.settInn(fido);
if (mittLilleHundehus.erTomt( ) ) System.out.println("Hundehuset er tomt");
Beholder<Hund> hhus = mittLilleHundehus;
Hund  sjuklingen = hhus.taUt( );
if (hhus.erTomt( ) ) System.out.println("Hundehuset er tomt");
```

Den siste linjen er ikke riktig. Svar på hvorfor i denne oppgaven:

Oppgave 7: Tegn opp hvordan Java-datastrukturen utvikler seg når disse linjene (rett over) med kode blir utført. Diskuter de to referansene mittLilleHundehus og hhus.

I Javas biblioteket finnes det et meget brukt grensesnitt med én parameter:

```
interface Comparable<T>{
    public int compareTo (T  med);
}
```

I Java-biblioteket står det at virkningen av metoden compareTo i et objekt av en klasse som implementerer dette grensesnittet er: "Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object." Med "the specified object" menes objektet som parameteren "med" peker på.

La oss bruke dette til å sammenligne kaniner. Når er en kanin større enn en annen? Er det når den ene er høyere enn den andre? Er det når den ene veier mer enn den andre? Dette må vi spørre den som bestiller programmet av oss om, og eventuelt de som skal bruke programmet. Slik kunnskap om den virkelige verden, kunnskap som vi skal avspeile i datasystemet vi lager, kalles *domene-kunnskaper.* La oss anta at vi får beskjed om at en kanin er større enn en annen hvis den veier mer enn den andre.

Vi kan da lage en klasse Kanin, der vi kan sammenligne objekter av denne klassen, dvs. vi kan sammenligne kaniner.
NB! Vi kan ikke ha to klasser i et program som heter det samme (i alle fall ikke om de ligger i samme katalog eller pakke). Vi må derfor erstatte den definisjonen av klassen Kanin som vi brukte i del 1, med denne:

```
class Kanin implements Comparable<Kanin>{
    public final String navn;
    private float vekt;
    Kanin(String nv, float v) {navn = nv; vekt = v;}
```

```
    @Override
     public int compareTo(Kanin med) {
         if(vekt < med.vekt) return -1;
         if(vekt > med.vekt) return  1;
         else return 0;
     }
}
```

Om instansvariablen vekt i Kanin er et heltall kan compareTo-metoden se slik ut:

```
     @Override
     public int compareTo(Kanin med) {
          return vekt – med.vekt;
     }
```

Spørsmål 8: Anta at instansvariablen vekt fortsatt er en float. Er det mulig at to float-verdier er like? Hva har dette å si for compareTo-metoden? Når kan vi da si at to kaniner er like? Er de for eksempel like om det bare skiller 10 gram?

Oppgave 9: Skriv først en metode som tar tre kaniner som parametre, sammenligner dem og skriver ut hvilken av de tre kaninene som er minst, nest minst og størst. Om du vil forenkle oppgaven litt kan du la være å ta hensyn til at noen av kaninene kan veie det samme. Hva må du gjøre om du skal skrive ut kaninenes navn i metoden? Skriv så et hovedprogram som oppretter tre kaniner og kaller denne metoden.

Til slutt skal vi se på et enkelt grensesnitt med to parametre. Dette er ikke viktig så tidlig i semesteret, og du kan gjerne vente med å lære dette til litt senere. Særlig det aller siste her (med arrayer) virker nok litt komplisert med en gang. Men prøv etter hvert å få med deg alle detaljene og bruk tid på å tegne opp datastrukturen.

Anta at vi skal lage et stort program, og så finner vi ut at flere steder trenger vi å ta vare på par av data, der de to elementene i paret er av forskjellig type. Da kan vi tenke oss at vi lager et grensesnitt som definerer de metodene (operasjonene) vi kan gjøre på slike par. Det viser seg at vi bare har behov for å sette inn de to verdiene samtidig, men vi må kunne lese av (hente) en og en verdi:

```
interface Par<S,T> {
    public void settInn (S en, T to);
    public S hentEn ( );
    public T hentTo ( );
}
```

En klasse som implementerer dette grensesnittet kan være:

```
class EnklesteParKlasse <S,T> implements  Par<S,T> {
     private S en;
     private T to;
     @Override
     public void settInn (S en, T to) {
          this.en = en;
          this.to = to;
     }
     @Override
     public S hentEn ( ) {return en;}
     @Override
     public T hentTo ( ) {return to;}
}
```

Nå har vi et begrep, et mønster, en klasse, til å lage objekter fra. Men det er et svært generelt (generisk) mønster. Fra dette mønsteret kan vi lage objekter av mange forskjellige typer. Hver gang vi putter noen ordentlige klassenavn inn istedenfor S og T, får vi en klasse av den gamle sorten, og alle objekter laget fra denne klassen er av samme type.

Oppgave 10: I notatet om generiske klasser tegner vi objekter av slike klasser selv om slike objekter ikke finnes. Gjør det samme her, dvs. lat som om det finnes objekter av klassene S og T, og tegn opp et liksom-objekt av klassen EnklesteParKlasse<S,T>. Tegn og fortell hva som hadde skjedd om du kunne utført hver av de tre operasjonene (kalle hver av de tre metodene) i objektet.

Først skal vi bruke EnklesteParKlasse til å lage en bitte liten del av et program for en dressurskole for hunder. Hit kommer personer med hundene sine:

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

Klassen Hund har vi fra notatet om generiske klasser. Hver gang en person med sin hund melder seg på dressurskolen lager vi et hundeobjekt og et personobjekt. Så tar vi vare på dette paret i et eget objekt, f.eks. slik:

```
Hund sisteHund = new Hund("Fido");
Person sistePerson = new Person("Petter");
Par<Person,Hund>  sistePar = new EnklesteParKlasse<Person,Hund>( );
sistePar.settInn(sistePerson, sisteHund);
```

Oppgave 11: Tegn opp datastrukturen slik den utvikler seg i programmet over. Det betyr nesten at du skal gjøre oppgave 7 om igjen, men nå med virkelige objekter slik de materialiserer seg som datastrukturer i dette Java-programmet.

Så skal vi se på deler av et program som skal brukes til å administrere data på en gård der det er kuer. I en del av programmet er klassen Ku definert, og her er hver ku bare identifisert med et nummer:

```
class Ku {
    final public int kuNummer;
    Ku (int nr) {kuNummer = nr;}
}
```

Men de som steller i fjøset kjenner best kuene med navn, så vi trenger noe som kan binde sammen navnet på en ku, med kua selv. Vi bruker Par-grensesnittet og EnklesteParKlasse til dette. Noen linjer kode vi lager ser da slik ut:

```
Ku miKu = new Ku(248765);
// Først lager vi et par uten innhold:
Par<String,Ku>   minKuInfo = new EnklesteParKlasse<String,Ku> ( );
// Så kobler vi sammen kua (som miKu peker på) med navnet Dagros i dette paret:
minKuInfo.settInn("Dagros", miKu);
System.out.println("Navnet på kua mi er " + minKuInfo.hentEn( ));
System.out.println("Nummeret på kua mi er " + minKuInfo.hentTo( ).kuNummer);
```

Oppgave 12: Modifiser tegningen fra oppgave 8 slik at du nå får en tegning av datastrukturen slik den utvikler seg når ku-programmet over utføres.

Under lager vi en array med plass til 100 par med navn og kuer. Grunnen til at den tredje linjen ser så stygg ut, er det som kalles "type erasure" av generiske klassenavn under kjøring av Javaprogrammer (i Java Virtual Machine, JVM). Men på samme måte som i notatet om generiske klasser behøver du ikke å forstå dette. Du vil også her få en advarsel fra oversetteren hvis du ikke ber om at den skal undertrykkes:

```
Par<String,Ku> [ ]  alleKuene;
@SuppressWarnings("unchecked")
alleKuene = (Par<String,Ku> [ ]) new Par [100];
```

Dette gir en tabell med referanser til par, der vi bare kan se Par-egenskapene til objektene. Vi kan også lage en tabell med referanser der vi kan se alle egenskapene til objektene av klassen EnklesteParKlasse:

```
EnklesteParKlasse<String,Ku> [ ] alleDineKuer =
                    (EnklesteParKlasse<String,Ku> [ ]) new EnklesteParKlasse [200];
```

I begge array-deklarasjonene over bør du i alle fall skjønne typen på arrayen, dvs. de to øverste linjene. Den første deklarasjonen gir oss en referansevariabel (med navnet alleKuene) til en array av referanser til Par av Strenger og Kuer. Denne gir bare tilgang til Par-egenskapene i de objektene tabellen peker på. Den andre deklarasjonen gir oss en array av referanser til EnklesteParKlasse-objekter av Strenger og Kuer. Gjennom denne får vi tilgang til alle egenskapene i de objektene arrayelementene

peker på. Men siden klassen EnklesteParKlasse i dette tilfellet inneholder de samme metodene som i grensesnittet Par er dette akkurat de samme egenskapene.

Til slutt setter vi inn to kuer på de to første plassene i arrayen alleKuene:

```
alleKuene[0] = new EnklesteParKlasse<String,Ku> ( );
alleKuene[0].settInn("Dagros", miKu);

alleKuene[1] = new EnklesteParKlasse<String,Ku> ( );
alleKuene[1].settInn("Litago", new Ku(12387));
```

Oppgave 13: Tegn opp datastrukturen slik den utvikler seg når Java-koden rett over blir utført.

Oppgave 14a: Lag et fullstendig program med en array av <String,Ku>-par, der du kan legge inn data om kuer (navn og kunummer), og der du kan finne kunummeret hvis du har navnet og omvendt. Kan flere kuer ha samme navn? Flere kuer kan ikke ha samme nummer. Hvordan sørger du for at denne betingelsen blir opprettholdt hver gang du legger inne en ny ku? Når du kjører programmet skal du først taste inn hvor mange kuer programmet maksimalt skal kunne ta vare på.

Oppgave 14b: Kunne du ha brukt en HashMap istedenfor en array?

Oppgave 14c: Hva tror du er den beste datastrukturen i dette tilfellet?

SLUTT