Oslo, januar 2025

Mer om objekter og klasser i Java

Versjon 2. januar, 2025

Av Stein Gjessing

1 Innledning

Dette notatet går mer i dybden enn notatet «Objekter og klasser i Java». Mange av temaene er de samme men det er en god del overlapp. De to notatene kan leses uavhengige av hverandre.

Målet med begge notatene er at du skjønner og kan programmere med klasser og objekter i Java. Det er mange detaljer i dette notatet, så er du nybegynner i Java lønner det seg nok å lese notatet «Objekter og klasser i Java» først.

Det notatet du nå leser starter med to kapitler med introduksjon til objekter og objektorientert tenkning. Kapittel 3 og 4 inneholder mange detaljer om hva som skjer når et Javaprogram utføres. Her er det mange tegninger som har som oppgave å gi deg en mental modell for dette.

Objektorientert programmering (OOP) ble funnet på av Ole-Johan Dahl og Kristen Nygaard for 60 år siden. I IN1010 skal vi lære om mange aspekter ved OOP, men i dette notatet skal vi bare ta for oss basale og enkle objekter, ikke objekter laget fra subklasser og med arv.

Kristen og Ole-Johan var opptatt av å modellere den virkelige verden inne i datamaskinen. I den virkelige verden har mange objekter, f.eks. personer og biler, eget liv eller oppførsel. Ole-Johan og Kristen var opptatt av at den oppførselen objektene har i den virkelige verden, også skulle finnes i objektene inne i datamaskinen.

Et eksempel: Kristen fikk i oppdrag å finne ut hvor mange bensinpumper det burde være på en bensinstasjon. Da laget han bensinpumper og biler inne i datamaskinen. Hvis vi antar at det f.eks. er tre bensinpumper på en bensinstasjon, at det kommer en ny bil til bensinstasjonen hvert minutt og at det tar fire minutter å fylle en tank, så skjønner alle at det blir kø. Men når situasjonen blir mer kompleks med tilfeldige ankomsttidspunkter og tidene det tar å fylle er variable så er svarene ikke så opplagte.

Det systemet Kristen simulerte inne i datamaskinen kan beskrives slik: Når en bil kommer til bensinstasjonen ser den først om det er noen ledig pumpe. Er det en ledig pumpe fyller bilen bensin, betaler og kjører videre. Men hvis det ikke er noen ledig pumpe stiller bilen seg i kø.

Når bilen foran i køen rykker fremover, må vår bil også gjøre det. Til slutt er bilen ved en bensinpumpe, og da fyller sjåføren opp tanken, betaler og kjører videre. En bil må altså kunne sette seg inn i en kø, rykke fremover i køen, fylle bensin, betale og kjøre videre.

Alle disse handlingene som en bil kan gjøre i den virkelige verden ønsket Ole-Johan og Kristen at bilen også skulle kunne gjøre inne i datamaskinen. For å programmere slike simuleringer inne i datamaskinen laget Ole-Johan og Kristen et eget programmeringsspråk kalt Simula. Det språket vi bruker i IN1010, Java, bruker klasser og objekter svært likt slikt det ble gjort i Simula.

La oss se på deler av et slikt program:

```
class Bil {
     private int literPåTanken;
     private Person sjåfør;
     private Bensinpumpe derJegFyller;
     public void settInnIKø( . . . ) {. . . }
     public void rykkFramover( . . . ) {. . . }
     public void fyllBensin( . . . ) {derJegFyller.fyll(); }
     public void betale( . . . ) {sjåfør.betale();}
     public void kjørVidere( . . . ) {. . .}
}

class Person {
     private int penger;
     public void betale( . . . ) {. . . }
}

class Bensinpumpe {
     private Bil denSomFyllerNå;
     public void fyll( . . . ) { . . . }
}
```

(Ikke alle systemer liker de norske bokstavene æ, ø og å; så når du programmerer anbefaler vi at du ikke bruker disse.)

Når du skal programmere løsningen av et problem må du dele problemet (og løsningen) inn i håndterbare deler. En slik del kan være et objekt. I IN1010 skal vi etter hvert lære hvordan vi passer på at et objekt har et fornuftig ansvarsområde, at objektene kan kommunisere seg imellom ved å kalle på hverandres synlige (public) metoder og hvordan de totalt sett kan løse vårt problem. For å få til dette må vi kunne reglene for hvordan objekter lages og oppfører seg. Dette skal vi se på i dette notatet.

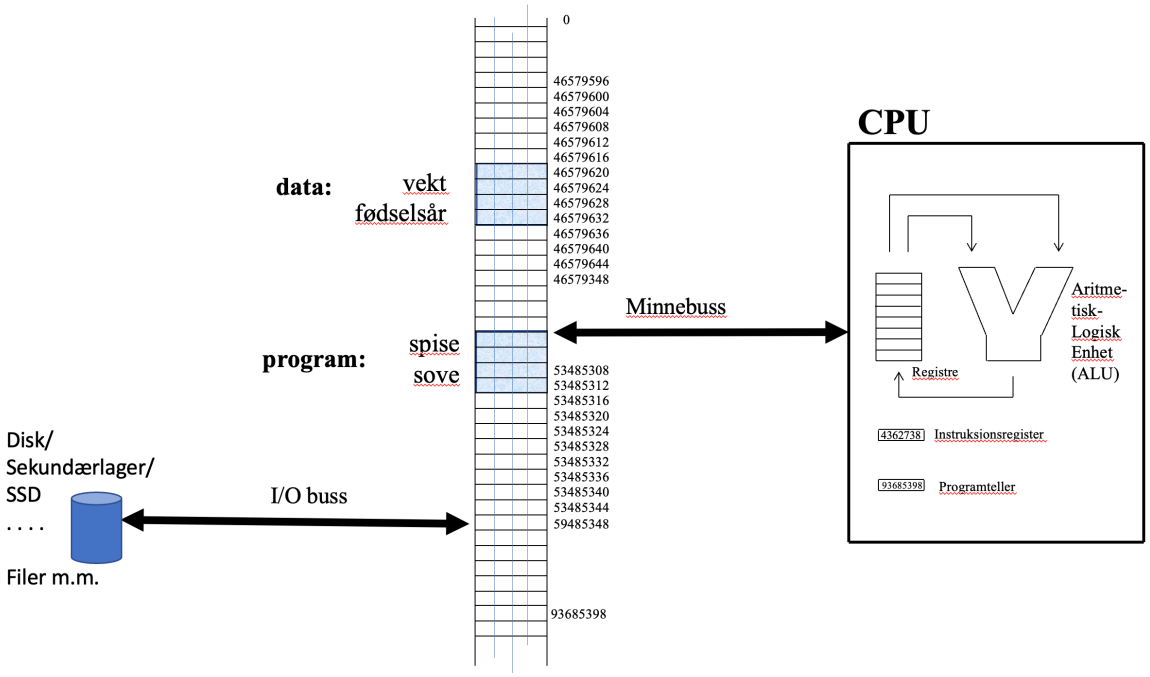
2. Enkle objekter

La oss se nøyere på en enkel klassedeklarasjon:

```
class Katt {
      private doble vekt = 0.5;
      final int fødselsår = 2023;
      public void spise (double mengde) {
             vekt =+ mengde;
      }
      public void sove( . . . ) { . . . }
}
```

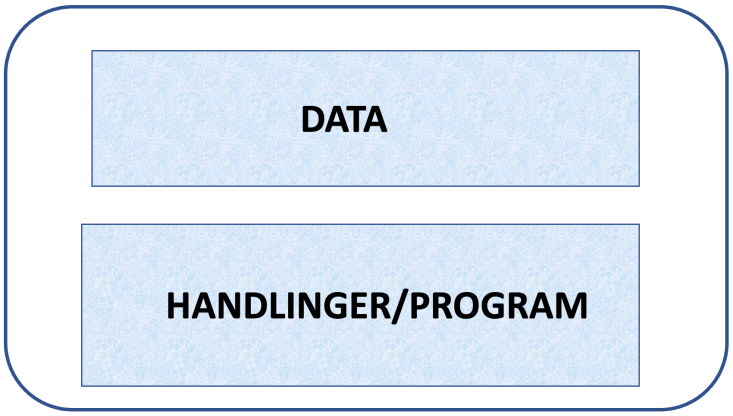
Vi oppretter et objekt ved å bruke nøkkelordet new, f.eks. `new Katt();`. Et slikt objekt er en del av det vi kaller programmets datastruktur. Senere skal vi se hvordan vi setter f.eks. fødselsvekt og fødselsår ved for eksempel kunne si `new Katt(0.5,2023);`.

Et program (når det utføres) oppretter en datastruktur som består av objekter med metoder, variabler og konstanter som programmets instruksjoner først oppretter og deretter forandrer. Det er kombinasjonen av programmet (algoritmen) og datastrukturene som gjør at problemet ditt blir løst. For å skjønne hvordan disse datastrukturene opprettes og manipuleres må vi vite hvordan de ser ut. Et objekt er en samling av variabler og metoder. I virkeligheten består et objekt av 1-ere og 0-er inne i primærlageret (minne, RAM (Random Access Memory)). En svært enkel datastruktur får vi ved å bare opprette ett Katte-objekt ved å si `new Katt();` som over. Figur 1 viser dette objektet inne i datamaskinen vår.



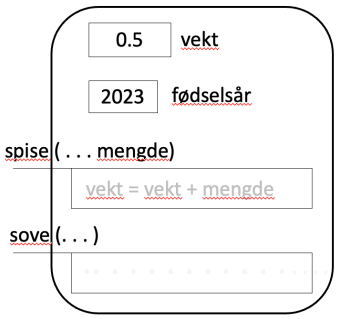
Figur 1. En datamaskin med sekundærlager, primærlager (minne) og prosessor (CPU). Et objekt er lagret i primærlageret med sine data og sitt program.

Vi kan ikke alltid tenke på objekter så komplisert som vi gjør i primærlageret i figur 1. I stedet må vi lage oss en modell av et objekt som et menneske kan forstå og tenke på på en enklere måte. Vi må abstrahere. Generelt består et objekt av en samling av data og program, og denne samlingen kan vi illustrere som i figur 2.



Figur 2. Generelt består et objekt av en samling av data og handlinger

Dataene i et objekt er variabler og konstanter og kalles instansvariabler. Når vi snakker om variabler (og da særlig instansvariabler) i IN1010 mener vi vanligvis både variabler og konstanter (en konstant deklareres ved å bruke modifikatoren `final`). Disse tar opp plass i primærlageret slik vi ser i figur 1. Kristen Nygaard kalte dette *substans*. I Java er handlingene i objekter metoder. I figur 3 ser vi et objekt av klassen Katt slik vi tenker på det i IN1010. Vi samler sammen variablene og metodene i et rektangel med avrundede hjørner:



Figur 3. Et objekt av klassen Katt der data er de to instansvariablene vekt og fødselsår og handlingene er de to metodene spise og sove.

Om metoder og metodeinstanser

Objektet i figur 3 kan opprettes med setningen `Katt min= new Katt();` Et objekt kalles også en instans av en klasse. Men ordet «instans» brukes også i en annen sammenheng (når vi starter en metode, se nedenfor).

Metodene i et objekt finner vi i klasse-deklarasjonen. Senere skal vi se at noen metoder i en klasse deklareres med modifikatoren `static`. Disse metodene skal vi behandle senere. Nå skal vi bare se på metoder som ikke er deklarert som static (eller statisk på norsk). Vi skal **tenke** oss at metodene som er deklarert i en klasse finnes i alle objekter av denne klassen. Men i virkeligheten, når programmet blir uført, finnes disse metodene bare ett sted i minnet. Metodene tar altså ikke opp mer plass i minnet når vi lager flere objekter, det er bare variabler (kalt instansvariabler) som tar opp plass for hvert nye objekt.

Hvis programmet kaller metoden spis() slik: `min.spis(0.2);` opprettes det en *metodeinstans* som utføres. Denne metodeinstansen inneholder lokale variabler i metoden, dvs. de variablene som er parametere og eventuelle variabler deklarert inne i metoden. I vårt tilfelle vil den formelle parameteren mengde være den lokale variabelen som opprettes i det metodeinstansen opprettes. Startverdien til denne variabelen er verdien av den aktuelle parameteren slik dette uttrykket er beregnet på kallstedet. I eksemplet over er det 0.2 som er den aktuelle parameteren (ja, 0.2 er et veldig enkelt uttrykk) og derfor vil variabelen mengde få denne verdien som startverdi.

Når alle instruksjonene i en metode er utført terminerer metoden. Da blir metodeinstansen borte og alle lokale variabler blir også borte, og plassen disse har tatt opp i minnet kan bli gjenbrukt av kjøretidsystemet senere. Hvis metoden blir kalt på nytt vil det bli opprettet en ny metodeinstans og det vil bli satt av ny plass til lokale variabler, osv.

Ut fra en objektorientert tankegang er det best å **tenke seg** at metodeinstansene opprettes inne i objektene. Da kan vi tenke oss at handlingen foregår inne i objektet, og det er lettere å skjønne at handlingene i metodeinstansen ser instansvariablene i det objektet metoden utføres inne i.

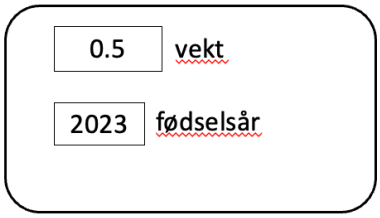
I virkeligheten legger kjøretidsystemet alle lokale variabler i en metodeinstans på det vi kaller en stakk eller en stabel. Når en metode kaller en annen vil den kallende metodens variabler bli liggende på stakken, og de lokale variablene i den metoden som kalles vil bli lagt oppe på toppen av stakken. Sammen med de lokale variablene ligger det på stakken en referanse til det objektet metoden utføres inne i. Når en metode terminerer plukkes de lokale variablene av (toppen av) stakken og kontrollen returnerer til den metoden som nå blir liggende på toppen av stakken. Senere i IN1010 skal vi se mer på de datastrukturene som heter stakker.

Om å tegne objekter

Hvis programmet ditt er enkelt behøver vi ikke tegne datastrukturen, det er ofte nok å bare tenke seg den. Men er datastrukturen mer komplisert eller du skal kommunisere den med

andre, er det ofte lurt å tegne den. Det viktigste med tegningen er ikke at den følger en spesiell mal, men at de som ser den skjønner hva den skal illustrere. Det finnes ingen fasit for hvordan en datastruktur skal tegnes. I IN1010 vil vi ofte tegne datastrukturer svært nøyaktig som et pedagogisk virkemiddel i en læresituasjon for å vise nøyaktig hvordan et Javaprogram fungerer. Når vi skjønner hvordan Java fungerer og vi ønsker å kommunisere spesielle løsninger kan vi lage enklere tegninger som bare illustrerer de poengene vi da vil belyse.

Om vi skal tegne et objekt behøver vi vanligvis ikke å tegne metodene inne i objektene fordi vi lett ser hvilke det er fra klassedeklarasjonen. Men instansvariablene i forskjellige objekter har vanligvis forskjellige verdier, derfor tegner vi ofte instansvariabler i noen av objektene. Dette gjelder kanskje mest referansevariable der det kan være nyttig å se hvilke objekter som refererer andre objekter. Noen ganger ønsker vi å legge vekt på spesielle egenskaper ved et objekt, og da tegner vi bare disse egenskapene. I figur 3 har vi tegnet et Katte-objekt svært nøyaktig. Her har vi til og med tatt med kode (`vekt= vekt+ mengde;`) inne i en av metodene. Dette er slik vi tenker oss det: At koden i metodene finnes og utføres inne i objektene. Men du vil vanligvis ikke få som oppgave i IN1010 å tegne kode inne i metoder. I høyden vil vi be deg antyde hvilke metoder som finnes i hvilke objekter. I figur 4 har vi tegnet et Katte-objekt slik vi vanligvis vil gjøre det med de to viktige instansvariablene (egenskapene) vekt og fødselsår. På denne måten ser vi raskt hvor mye denne katten veier og hvilket år den ble født.



Figur 4. Et objekt av klassen Katt med de to instansvariablene vekt og fødselsår.

Hva som skjer når et Javaprogram utføres

Et Javaprogram blir oversatt til såkalt byte-kode før det blir utført. Denne koden blir også lagret i primærlageret, og det er derfra Javas virtuelle maskin (JVM) henter instruksjoner som skal utføres. Dette skal vi imidlertid ikke legge vekt på i resten av dette notatet. I dette notatet skal vi se på hva som skjer inne i datamaskinen (i primærlageret / minnet / RAM) når et Javaprogram utføres. Vi skal ikke tenke så detaljert som figur 1 viser. Derimot skal vi tenke slik det antydes i figur 3 og figur 4. Når et program starter opp er primærlageret der data blir lagret blankt og prosessoren vil starte med den første instruksjon i programmet. Rett før programmet terminerer vil primærlageret ofte være fullt av objekter som er opprettet for å løse problemet.

I eksemplene i dette notatet er identifikatorer (navn) som vi som programmerere selv lager, skrevet på norsk. Den pedagogiske hensikten med dette er at leseren lett kan slutte at alle engelske navn allerede finnes i Java-språket eller i Java-biblioteket. Når du skal programmere sammen med andre, må dere bli enig om hva slags konvensjoner dere har for egendefinerte identifikatorer. Mange program skal leses av personer som ikke kan norsk, og i slike tilfeller kan det være lurt å unngå norske navn i programmene.

Vi følger den vanlige konvensjonen i Java om at alle klassenavn starter med Stor bokstav. Sammensatte ord skrives i ett med storBokstav for hvert nye ord. Konstanter skrives vanligvis med bare STORE bokstaver, men variabler som får verdier som senere ikke skal forandres skrives som andre variabler men vil også bli deklarert med modifikatoren `final`.

I eksemplene i dette notatet i er det meningen å illustrere Java-språket og datastrukturer i Java. Vi tar derfor ikke med kommentarer i programmene. Det betyr at programmer slik de skrives her ikke er slik du skal skrive dem. Når du programmerer skal du alltid programmere med kommentarer og ofte kan det være fornuftig å bruke Java-doc.

Litt på siden av temaet i dette notatet kan vi bemerke at en form for kommentarer som kalles **invarianter** er svært nyttig. Vi skal behandle invarianter nøyere utover i semesteret, men her sier vi bare at en invariant kan være et utsagn om en del av datastrukturen i programmet vårt, og dette utsagnet om hvordan datastrukturen ser ut er gyldig under (nesten) hele programutførelsen. Senere i IN1010 kommer vi tilbake til eksempler som illustrerer dette.

I dette notatet utvikler vi ikke programmer fra problemer, men vi tar for oss ferdiglagede programmer og ser hvordan datastrukturen i primærlageret utvikler seg når disse programmene blir utført. Dette gjør vi for at du skal lære deg hva som skjer når et Java-program blir utført og sammenhengen mellom Java-kode og datastrukturer.

Når du selv skal bruke det du lærer her, er det imidlertid den **omvendte** aktiviteten som er viktigst. Når du skal løse et problem skal du finne opp og se for deg, og også tegne hvis nødvendig, den datastrukturen som trengs og hvordan denne datastrukturen skal utvikle seg i primærlageret for å finne løsningen på problemet ditt. Instruksjonene som beskriver hvordan en datastruktur utvikler seg kaller vi en *algoritme*. Først etter at du har funnet ut hva slags datastruktur du trenger, og hvordan denne skal utvikle seg for å ende opp i en ferdig løsning, først da bør du starte å skrive den nøyaktige Javakoden som skal til for å løse dette problemet. De fleste studentene synes det er vanskeligere å lære seg å utvikle programmer fra problemer enn å lære seg programmeringsspråket Java.

Dette notatet inneholder mange detaljer, og ingen er uviktige. Når du tror du kan programmere med de mest basale konstruksjonene i Java, kan du igjen raskt lese gjennom notatet og sjekke at du skjønner alt.

Når du leser dette notatet første gang anbefaler jeg at du lese raskt over kapittel 3, som handler om såkalte statiske egenskaper i en klasse. Det viktigste, om objekter i Java, starter i kapittel 4. Kom så tilbake til kapittel 3 om det er noe om statiske egenskaper du lurer på.

3 Om statiske variabler/konstanter og statiske metoder.

I en klassedeklarasjon i Java kan du i tillegg til instansvariabler som finnes inne i alle objekter, deklarere variable som bare finnes **en gang** (og ikke blir laget på nytt hver gang du sier «new»). Slike variabler brukes blant annet til å lagre egenskaper som er felles for alle objekter av denne klassen. For å lage en slik variabel i en klasse bruker vi nøkkelordet (eller modifikatoren) «static» (statisk på norsk). Nedenfor skal vi først forklare litt teknisk om hva slike statiske variabler (og statiske konstanter og statiske metoder) er. Så skal vi ta noen få eksempler. Grunnen til at vi behandler «static» tidlig i dette notatet er at ordet forekommer tidlig i ethvert Java-program: «public static void main . . . ».

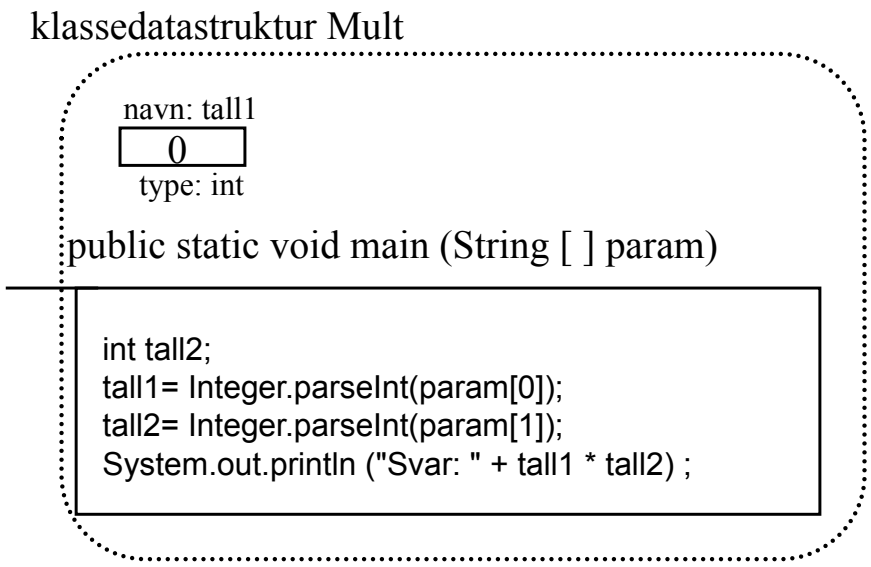
Det første som skjer når ethvert Javaprogram starter er at det settes av plass til alle statiske variabler, statiske konstanter og statiske metoder. Denne plassens settes av inne i datastrukturer vi kaller *klassedatastrukturer*.

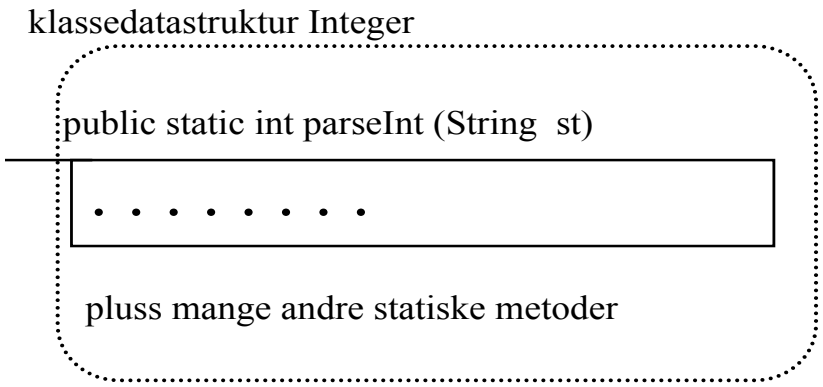
La oss se på et program som multipliserer to tall. Dette er et fullstendig Java-program, litt vanskeligere enn «Hello world»:

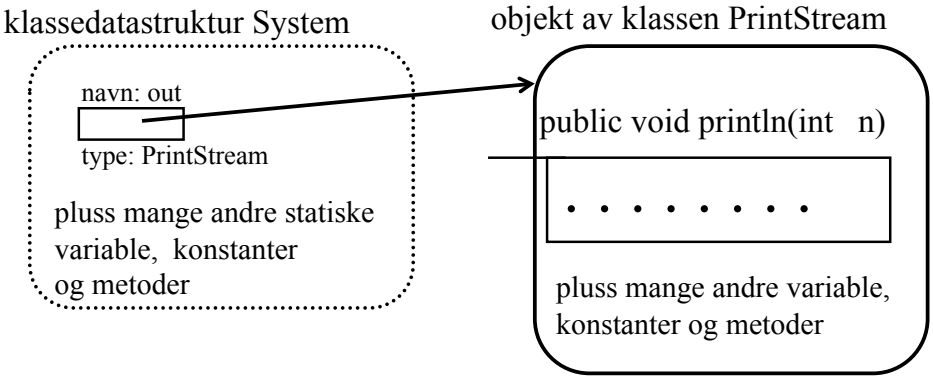
```
class Mult{
       static int tall1;
       public static void main (String [ ] param) {
            int tall2;
            tall1= Integer.parseInt(param[0]);
            tall2= Integer.parseInt(param[1]);
            System.out.println ("Svar: " + tall1 * tall2);
       }
}
```

Når et Java-programmet starter lager kjøretidsystemet (Java run time system) en klassedatastruktur for alle klasser som programmet skal bruke. Hvis vi ønsker å illustrere dette tegner vi disse datastrukturene som prikkede eller stiplede rektangler med avrundede hjørner. Inne i disse rektanglene ligger alle static-egenskapene i klassen. I programmet over brukes minst fire klasser: Mult, Integer, String og System. Det vil følgelig bli opprettet fire klassedatastrukturer i det programmet starter opp.

Å illustrere alle static-egenskapene i alle klassedatastrukturer i et Java-program er nærmest umulig. Nedenfor tegner vi en noen variabler og metoder i noen klassedatastrukturer. Grunnen til at det er tegnet så nøyaktig i dette notatet er at du som leser skal skjønne hva som foregår inne i datamaskinen når et Javaprogram utføres. Du behøver selv aldri tegne det så nøyaktig. Til enhver tid behøver du bare tegne det du (og dem du samarbeider med) trenger for å skjønne hva som foregår**.** Allerede i tegningene nedenfor utelater vi detaljer i klassedatastrukturene for Integer, String og System som vi ikke trenger for å forklare hva dette programmet gjør.







Alle variabler og konstanter i klassedatastrukturene blir initialisert i det programmet starter opp. Hvis vi ikke har gitt en variabel en startverdi vil de blitt gitt sin standard startverdi (f.eks. får int verdien 0 og referanser verdien `null`). I en av figurene over ser vi klassedatastrukturen til Mult etter at den statiske variabelen `tall1` er blitt initialisert til 0.

Grunnen til at vi tegner metoder er at det gjør det klart hvor metodene ligger og hvilket skop (engelsk scope) koden i metodene har. Vi ser f.eks. lett fra tegningene over at static-variabler i

en klasse er i skopet til en statisk metode i samme klasse, men at instansvariabler i samme klasse ikke er i skopet.

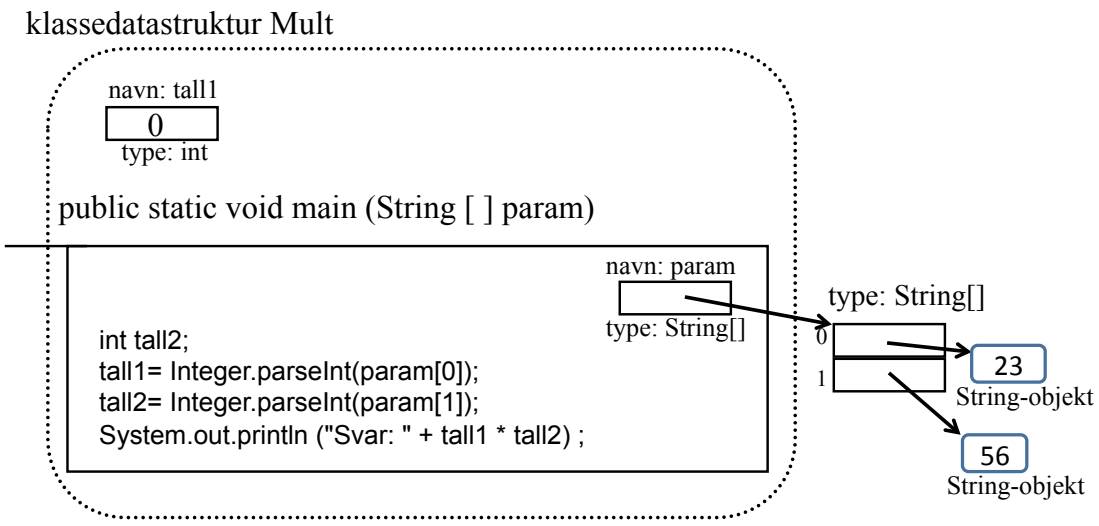
Det som dessverre ikke er så lett å se fra tegningene er at static-egenskaper (variabler og metoder) i en klasse er i skopet til alle metodene i alle objektene av den samme klassen.

Det er Javas kjøretidsystem som starter opp programmet vårt ved å kalle hovedmetoden i hovedklassen (metoden med navnet main() i klassen med det samme navnet som Javafilen vi kjører). Når en metode kalles oppstår det en *metodeinstans*, og i denne metodeinstansen blir lokale variabler opprettet.

I det en metode starter opp, vil alle *formelle parametre* oppstå som lokale variabler og initialiseres med verdiene til de *aktuelle parametrene* (vi kan tenke oss dette som tilordninger). Når main() starter ligger den aktuelle parameteren inne i kjøretidsystemet, og verdien av denne er en referanse til String-arrayen som inneholder argumentene til kallet på utføringen av programmet. Programmet over starter vi f.eks. ved å skrive:

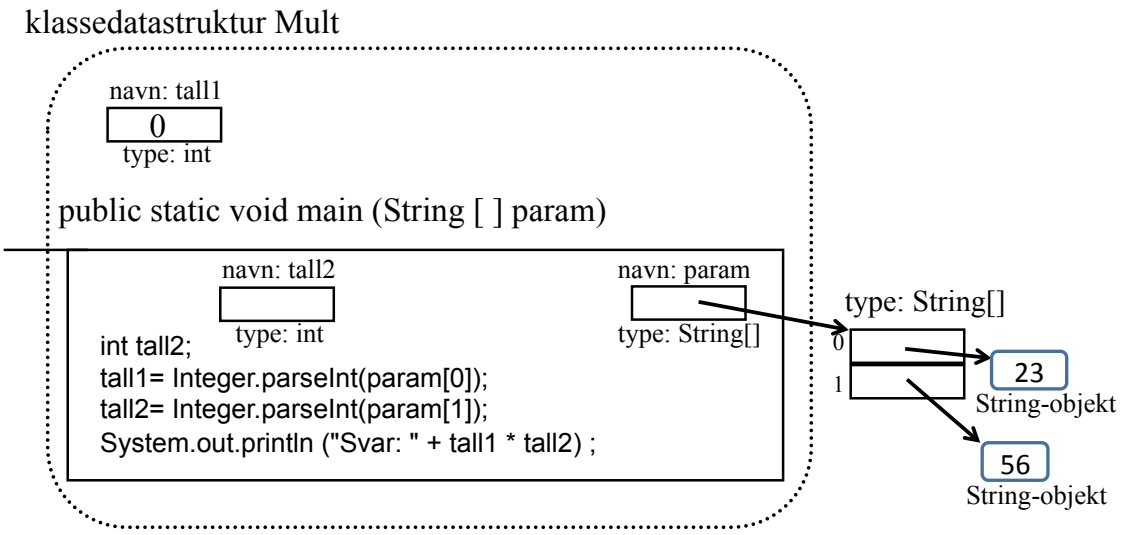
```
>java Mult 23 56
```

I det main-metoden starter opp er situasjonen denne:



I klassedatastrukturen Mult over er det nå oppstått en instans av main-metoden. Legg merke til den variablen som er tegnet øverst i høyre hjørne av denne metodeinstansen. Det er den formelle parameteren som heter param og som nå er blitt en lokal variabel med samme navn og med typen array av referanser til String-objekter og med initialverdi en referanse til arrayen som inneholder parametrene til programmet (tekstene "23" og "56").

Det neste som så skjer i main()-metoden er at deklarasjonen int tall2 blir utført. Da blir det opprettet en variabel inne i metoden, med navn tall2 og type int:



Legg merke til at det ikke er noen verdi inne i denne variabelen. Det er fordi den ikke er initialisert. Variabler i metoder blir ikke initialisert uten at vi eksplisitt ber om det.

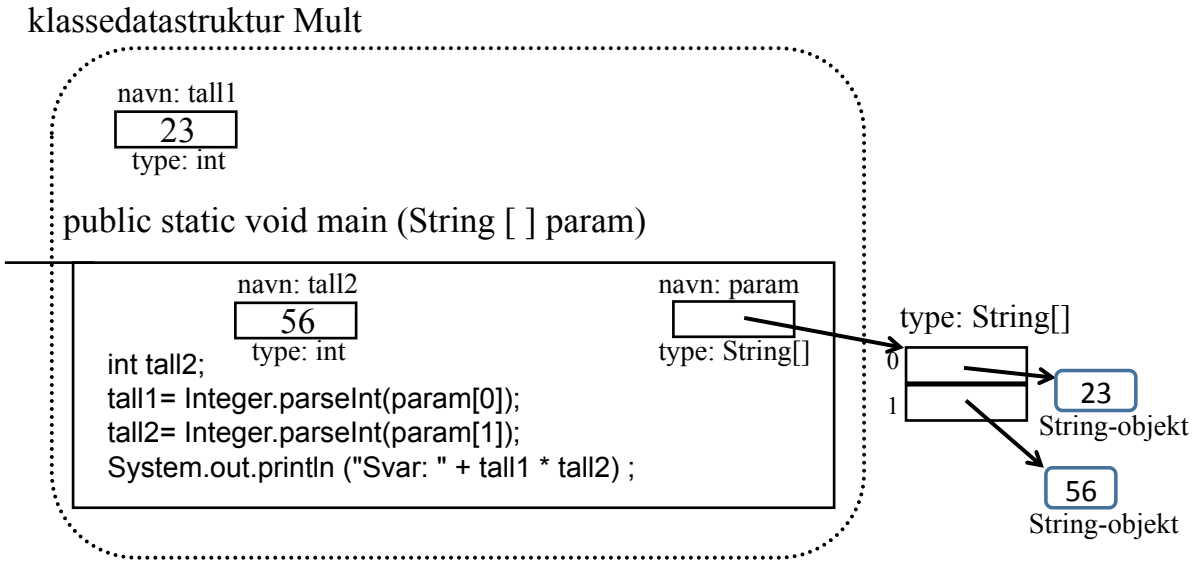
At vi har deklarert tall1 som statisk variabel i klassen, og tall2 som *lokal variabel* i metoden main() har ingen dypere mening. Variabelen tall1 er ment å vise hvordan en statisk variabel deklareres og hvor plass til denne allokeres, og variablen tall2 er ment å vise hvordan en lokal variabel deklareres og hvor plass allokeres. Siden tall1 bare brukes i metoden main() burde den kanskje, på samme måte som tall2, vært deklarert som lokal variabel i metoden main().

Dernest blir disse to setningene utført:

```
tall1= Integer.parseInt(param[0]);
tall2= Integer.parseInt(param[1]);
```

Oppgave 1. For å vite virkningen av den statiske metoden parseInt i klassen Integer, må vi slå opp denne klassen i Java-biblioteket. Gjør det, og tegn hvordan den formelle parameteren blir en lokal variabel og blir tilordnet verdien til den aktuelle parameteren.

Etter at disse to setningene er utført er situasjonen denne:



Neste linje er:

```
System.out.println ("Svar: " + tall1 * tall2);
```

Oppgave 2. Slå opp i Java-biblioteket, finn klassen System, variabelen out, klassen PrintStream og metoden println() (det er 10 stykker, hvilken og hvorfor?).

Til slutt skrives altså "Svar: 1288" ut på skjermen, og metoden main() terminerer. I det en metode terminerer bli alle lokale variabler inne i metoden borte, i dette tilfellet de to variablene param og tall2. I det hele programmet terminerer blir hele klassedatastrukturene (og alle objekter) også borte.

Oppgave 3: Lag et mer robust program, som ikke kræsjer når du ikke gir med riktige antall (for få) parametere. Er det andre ting du kunne gjort for å lage programmet mer robust?

Vi har nå lært at når en statisk variabel deklareres i en klasse, vil det bare finnes én slik variabel, selv om det senere blir opprettet mange objekter av klassen. Den fornuftige bruken av en slik variabel er derfor som felles variabel for alle objektene av klassen. Hvis programmet for eksempel er interessert i å vite hvor mange objekter det er laget av en klasse, kan en statisk variabel i klassen brukes til dette.

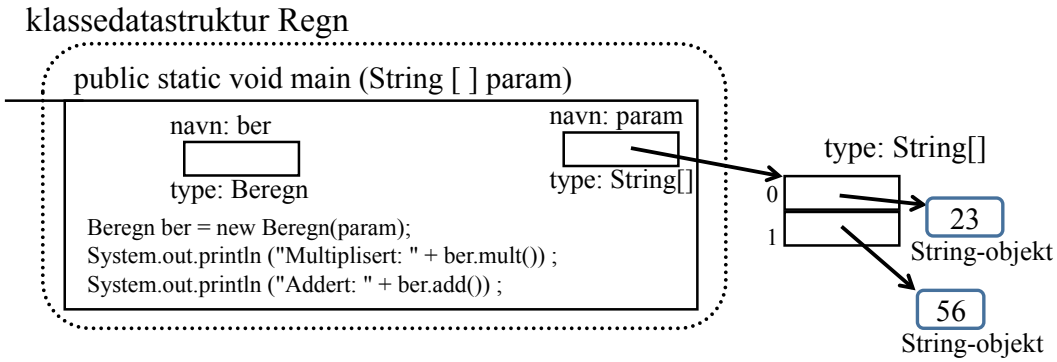
4 Mer om objekter i Java

La oss se på dette programmet som oppretter og manipulerer noen objekter.

```
class Regn{
   public static void main(String [ ] param) {
     Beregn ber= new Beregn(param);
     System.out.println("Multiplisert: " + ber.mult()) ;
     System.out.println("Adert: " + ber.add()) ;
   }
}
class Beregn {
   private String [ ] tall;
   public Beregn(String [ ] par) {
      tall= par;
   }
   public int mult( ) {
      int tall1, tall2;
      tall1= Integer.parseInt(tall[0]);
      tall2= Integer.parseInt(tall[1]);
      return tall1 * tall2;
   }
   public int add( ) {
      int tall1, tall2;
      tall1= Integer.parseInt(tall[0]);
      tall2= Integer.parseInt(tall[1]);
      return tall1 + tall2;
   }
}
```

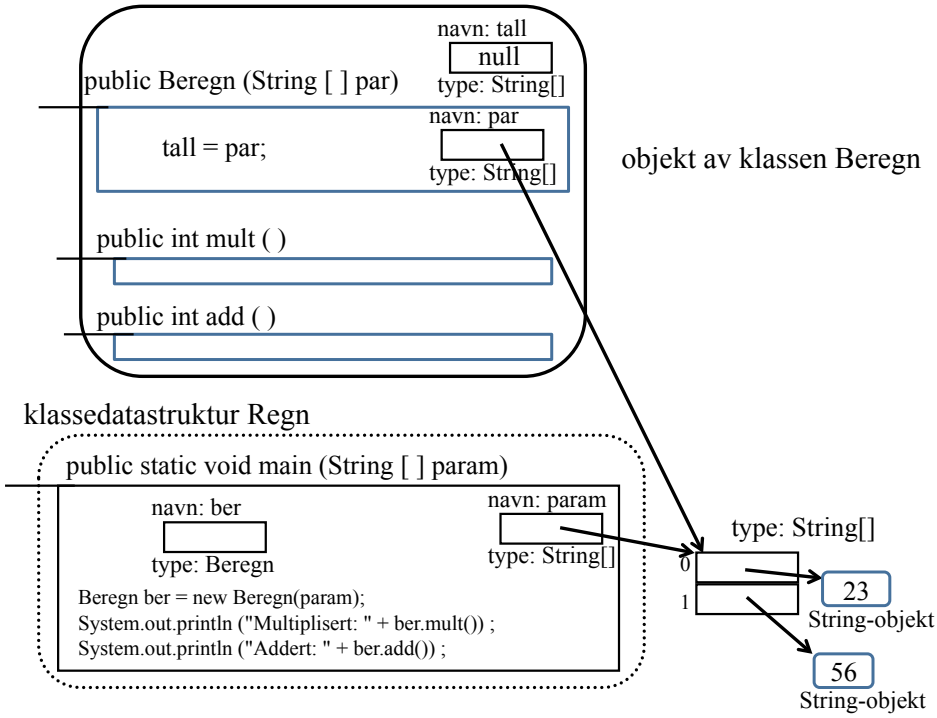
Vi kjører dette programmet på den samme måten som det forrige programmet:
>java Regn 23 56

I det main() starter opp har vi samme situasjon som i Mult-programmet. Vi bryr oss ikke om de to klassedatastrukturene til Integer og System denne gangen. Vi tenker heller ikke på klassedatastrukturen til klassen Beregn, for den er jo tom (det er ingen statiske egenskaper i denne klassen).

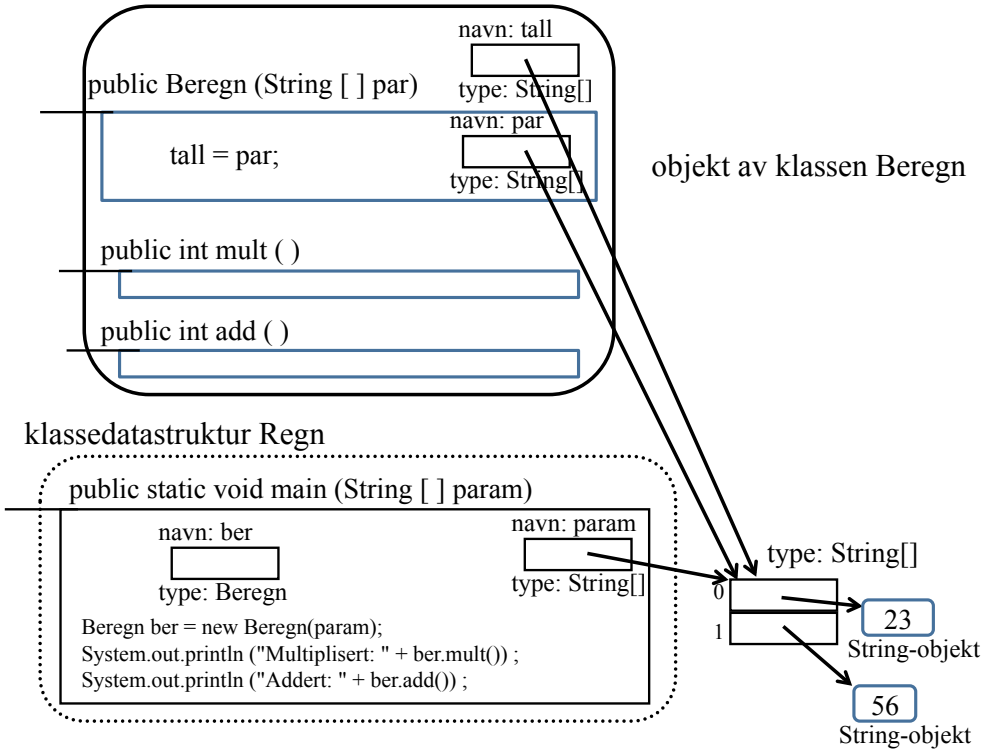


Så utføres deklarasjonen og tilordningen:
`Beregn ber = new Beregn(param);`

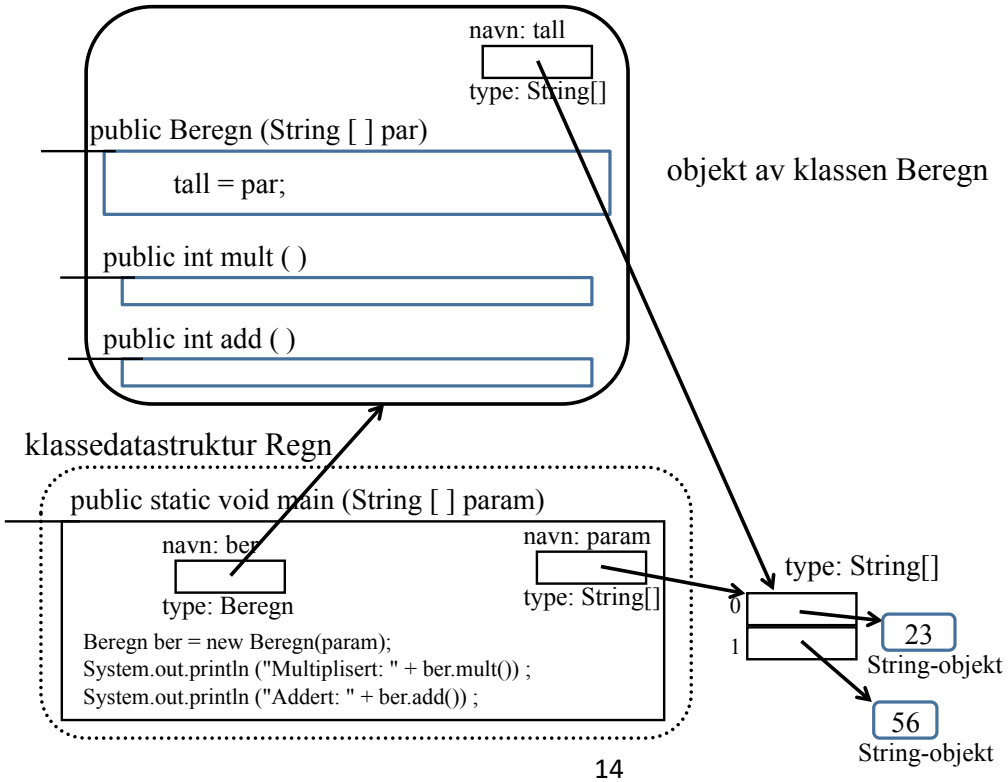
Uttrykket på høyresiden har den viktige (side)effekten å opprette et objekt av klassen Beregn. Vi sier også at objektet er en *instans* av klassen Beregn. Aktuell parameter til konstruktøren er en referanse til arrayen med to String-referanser. Men før konstruktøren blir utført blir det laget en instansvariabel inne i objektet med navn tall, og med initialverdi `null`. Nedenfor har vi tegnet dette objektet med én instansvariabel, én konstruktør og to metoder. I Javas minne blir alltid nye instansvariabler opprettet hver gang et objekt opprettes. Men i minnet vil ikke metodene opprettes på nytt inne i hvert objekt. I Java-systemet finnes disse metodene bare lagret ett sted. Men vi skal **forestille oss** og tenke oss at metodene finnes inne i alle objekter. Når vi tenker objektorientert, så må vi tenke oss at alle objekter har sine egne handlinger.



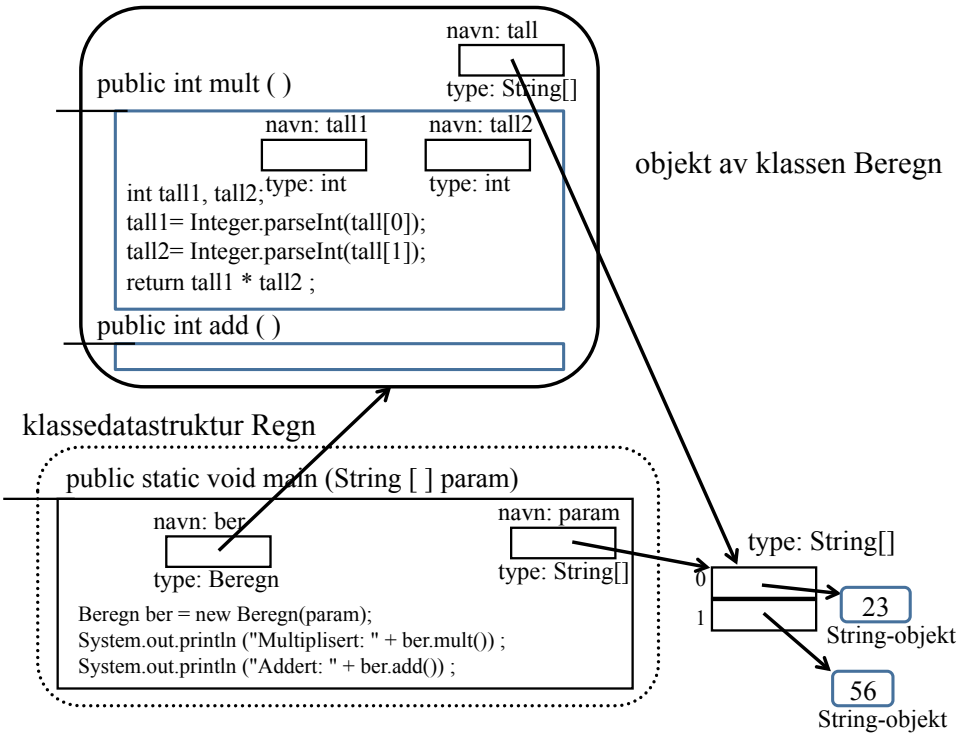
Etter at tilordningen inne i konstruktøren (`tall = par;`) er utført er situasjonen denne:



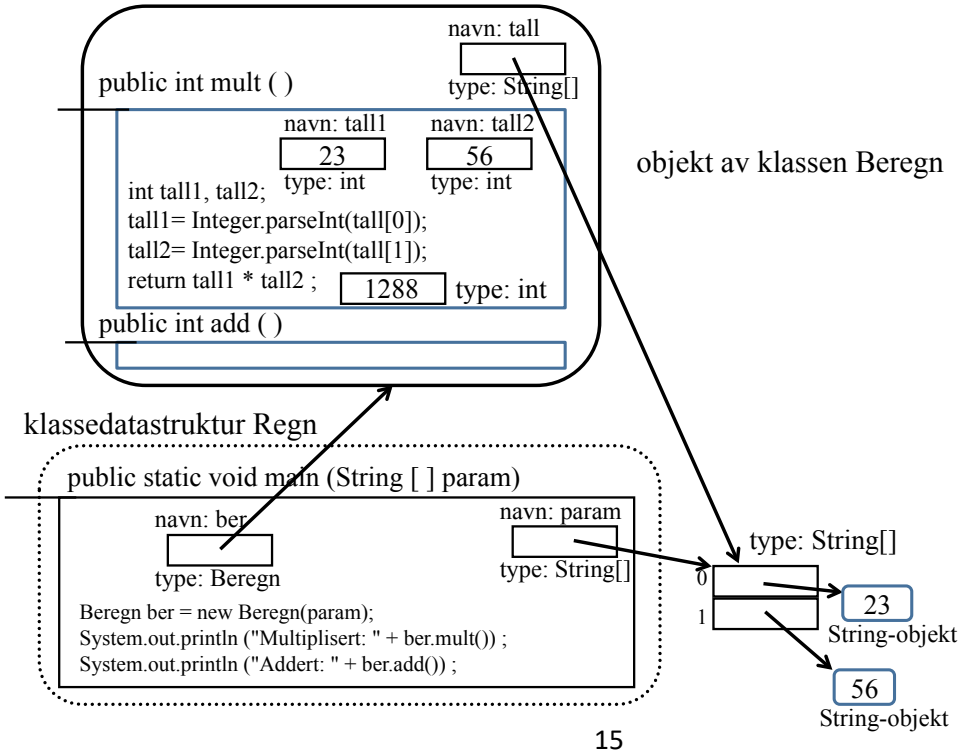
Etter at tilordningen er utført terminerer konstruktøren, og objektet er ferdig laget. I main() vil så en referanse til det nylagde objektet bli tilordnet variabelen ber:



Når objektet er laget vil konstruktøren ikke bli kalt igjen, og vi tegner den derfor ikke lenger. I neste setning i main skal uttrykket ber.mult() beregnes. Vi følger ber-referansen til et objekt, og dette objektet inneholder metoden mult() som så blir utført. Metoden har ingen parametre, men det første som skjer i metoden er at to lokale variabler blir deklarert:



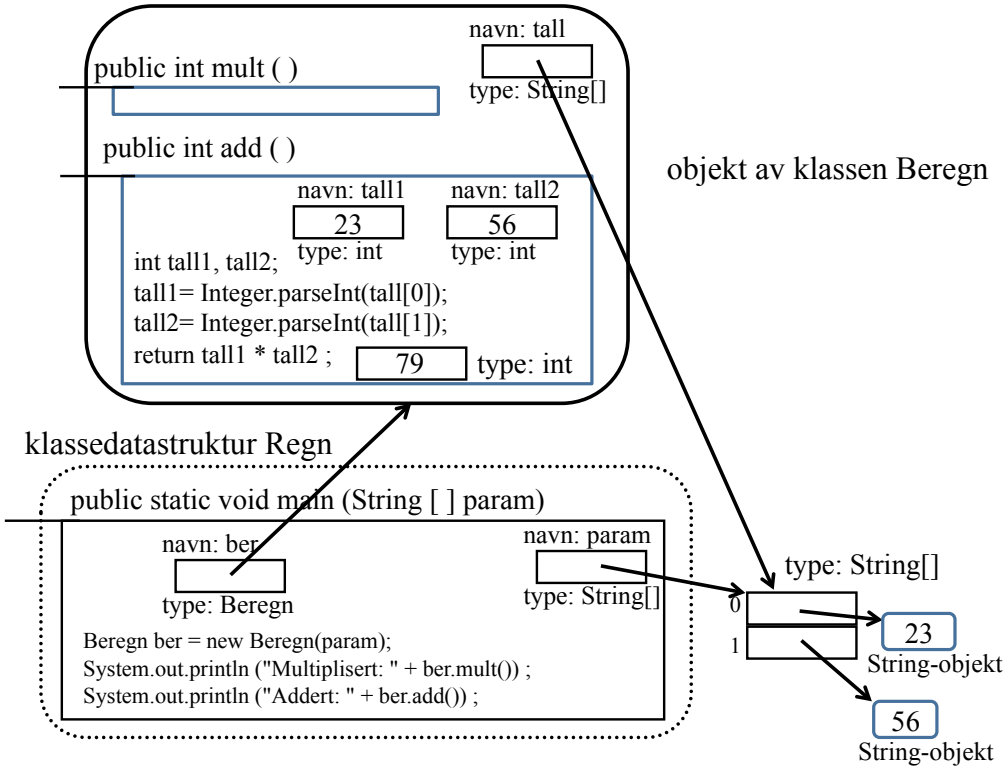
Disse to variablene får ingen startverdi fordi de er lokale variabler inne i en metode. De to tilordningene gir dem imidlertid verdier, og til slutt beregnes tall1*tall2, og resultatet av denne beregningen skal returneres som metodens resultat:



Når metoden er ferdig blir de lokale variablene tall1 og tall2 fjernet, og denne metodeinstansen av mult() terminerer. Metoden main() skriver så ut på skjermen:

Multiplisert: 1288

I siste linje av main() kalles ber.add(), og nesten det samme skjer igjen:



Når metoden add() er ferdig blir også denne metodeinstansen borte, og main() skriver ut: Addert: 79. Deretter terminerer metodeinstansen til main(), og kontrollen går tilbake til kjørtetidsystemet (som er en del av JVM) som så avslutter hele Java-maskinen.

La oss se på et program som gjør nøyaktig det samme, men der klassen Beregn er skrevet litt annerledes og kalt BergenT.

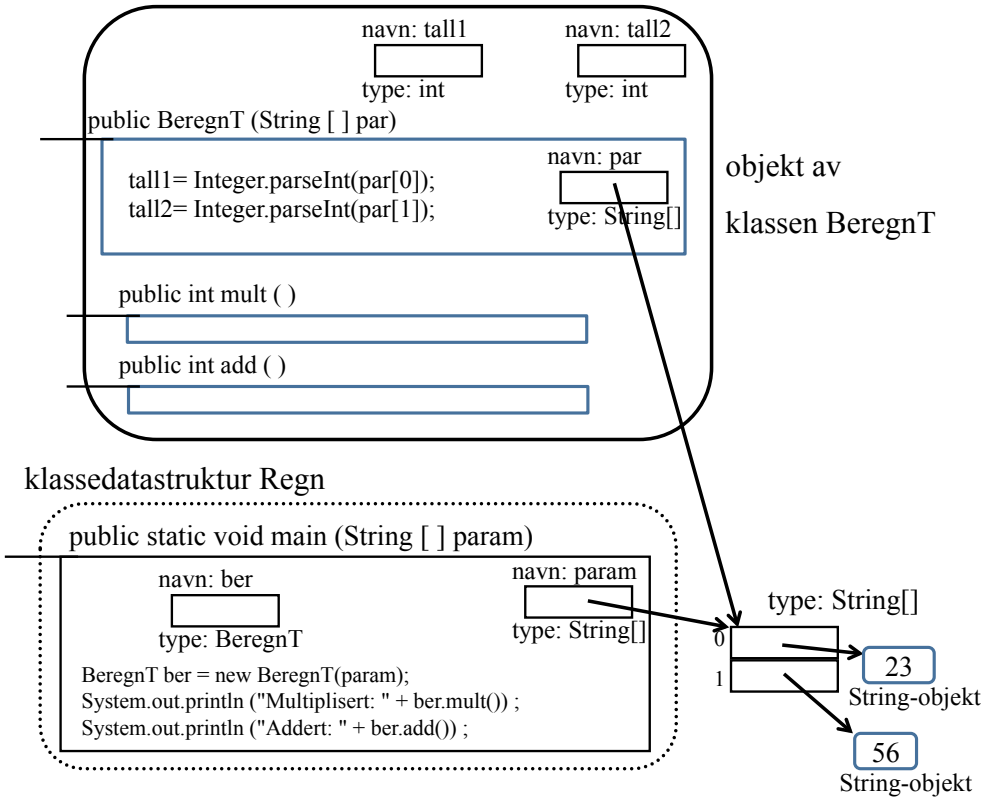
```
class Regn{
     public static void main (String [ ] param) {
           BeregnT ber = new BeregnT(param);
           System.out.println ("Multiplisert: " + ber.mult()) ;
           System.out.println ("Adert: " + ber.add()) ;
      }
}
```

```
class BeregnT {
     private int tall1, tall2;
     public BeregnT (String [ ] par) {
          tall1= Integer.parseInt(par[0]);
          tall2= Integer.parseInt(par[1]);
     }

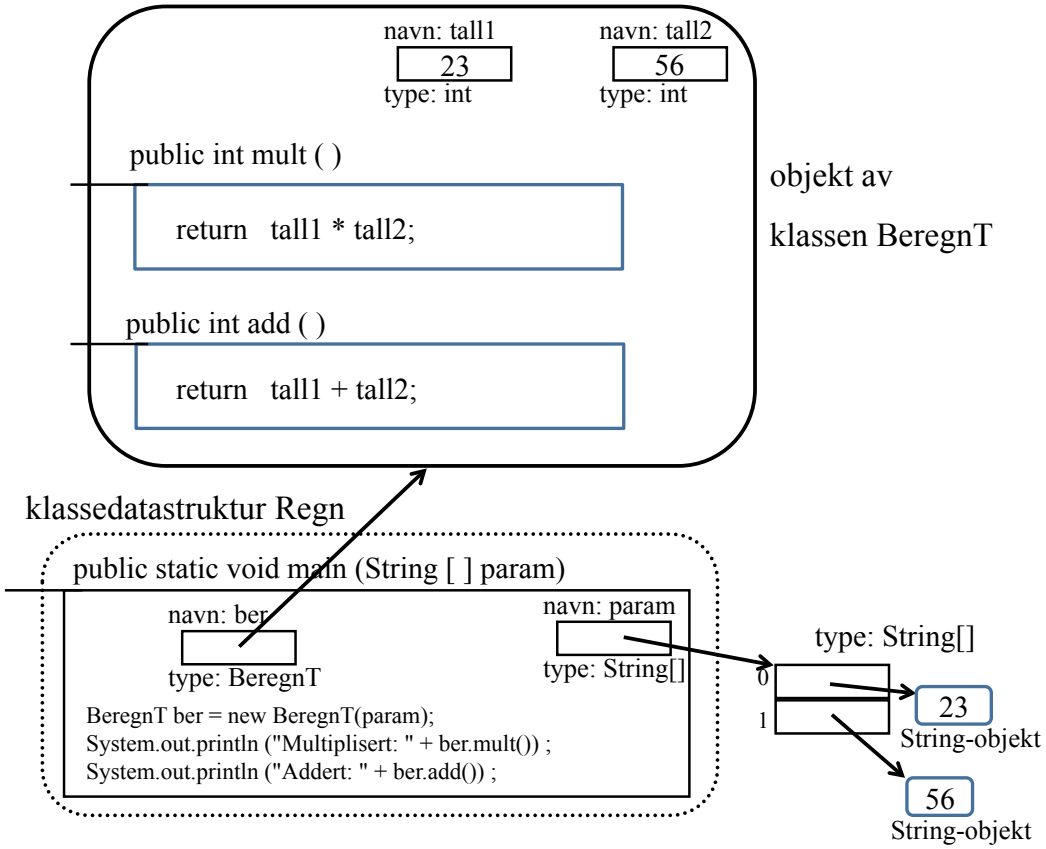
     public int mult ( ) {
          return tall1 * tall2 ;
     }
     public int add ( ) {
          return tall1 + tall2 ;
     }
}
```

Vi ser at BeregnT og Beregn har to offentlige (public) metoder med samme *signatur*. Vi sier at to metoder har samme signatur når de har samme navn og samme antall og type parametre (i samme rekkefølge). Java krever også at to metoder med samme signatur skal returnerer et resultat av samme type. Forskjellen på de to klassene BeregnT og Beregn er de private variablene og implementasjonen av konstruktøren og metodene. Vi skal senere lære om grensesnitt (interface) i Java, og kan da si mer om slike klasser som oppfører seg likt offentlig, men som har forskjellige (private) implementasjoner.

Etter at main() har startet opp, og konstruktøren i objektet av klassen BeregnT skal til å utføre sin første instruksjon, ser datastrukturen slik ut:



Så utfører konstruktøren sine to instruksjoner, konstruktøren terminerer og referansevariablen ber blir satt til å peke på det nylagde objektet:



Metoden main() utfører så sine to siste linjer, og med dette blir uttrykkene ber.mult() og ber.add() utført, og utskriften blir:

Multiplisert: 1288

Addert: 79

Til slutt terminerer dette programmet på samme måte som det forrige.

Oppgave 4. Kan du si noe generelt om forskjellen på de to implementasjonene? Når vil du velge den ene, og når vil du velge den andre type implementasjon?

5 Mer om metoder og metodeinstanser

Vi vet nå at når en metode kalles, oppstår det en metodeinstans som inneholder alle lokale variabler (inklusive parametre) i metoden. Koden til metodene i et objekt finnes bare ett sted. Selv om det oppstår mange objekter er det bare instansvariabler som opprettes på nytt for hvert objekt og som tar opp plass i primærlageret. Men når vi skal forstå hvordan programmet vårt oppfører seg når det utføres, må vi tenke oss at metodene finnes inne i – og utføres inne i – de forskjellige objektene. På denne måten ser vi enkelt hvilke variabler og konstanter metoden har tilgang (skopet til programmet).

Når programutførelsen er inne i en metodeinstans (la oss kalle denne instansen nr. 1) , kan programmet kalle en annen (eller den samme) metoden, og det opprettes en ny metodeinstans (nr. 2). Denne kan igjen kalle en metode og det opprettes enda en ny metodeinstans (nr. 3) og så videre. Når vårt java-program utføres vil utføringen av main-metoden være metodeinstans nr. 1. Hvis for eksempel programmet vårt er inne i instans nr. 7, vil det finnes totalt 7 metodeinstanser med hvert sitt sett av lokale variabler. Når metodeinstans nr. 7 er ferdig utført, vil programkontrollen gå tilbake til kallstedet i metodeinstans nr. 6. Kaller denne instansen (nr. 6) enda en metode vil dette bli nåværende aktive metodeinstans nr. 7, osv. En metode kan gjerne kalle seg selv. Da kaller vi det et rekursivt metodekall. Dette er temaet i en senere IN1010-forelesning.

Metodeinstanser er altså som tallerkener stablet oppå hverandre. Hver gang vi kaller en ny metode legger vi en tallerken på toppen av bunken. Hver gang en metode er ferdig tar vi av tallerkenen på toppen. Den første (og nederste) tallerkenen er utføringen av main-metoden. En slik organisering kaller vi gjerne en *stakk* (engelsk: *stack*). Vi sier gjerne at lokale variabler i metoder legges på en stakk, gjerne i bestemt form: *stakken (the stack)*. I dette tilfellet er det heldigvis kjøretidsystemet som holder orden på stakken av metodeinstanser og lokale variabler. Senere i IN1010 skal du lære å lage egne datastrukturer som er organisert som stakker.

Når et objekt opprettes må det også settes av plass til variabler – i det tilfellet instansvariabler. Den plassen i primærlageret der data i objekter får plass kalles gjerne *haugen,* eller mer vanlig på engelsk: *the heap.*

ArrayList oog HashMap - Om å ta vare på mange verdier av samme type

I IN1010 skal du lære to basale måter å ta vare på mange verdier av samme type: Arrayer og lenkede lister. Den siste måten blir tema i slutten av februar, arrayer må du lære med en gang.

Arrayer har den fordelen at de kan ta vare på både primitive verdier og referanse-verdier. Arrayer er svært effektive hvis du skal slå opp og lagre verdier med gitte indekser, men f.eks. å lagre verdier inne i mellom andre er ikke så lett. Å kunne arbeide med arrayer er et viktig læringsmål i IN1010. I Javas bibliotek er ofte den hemmelige datastrukturen bygget opp ved hjelp av arrayer. Et læringsmål i IN1010 er å kunne lage klasser som ligner på dem du finner i Javas bibliotek, og bl.a. derfor må du kunne programmere med arrayer.

Nå skal vi se på to klasser fra Javas bibliotek, uten å se på hvordan tjenestene er implementer.

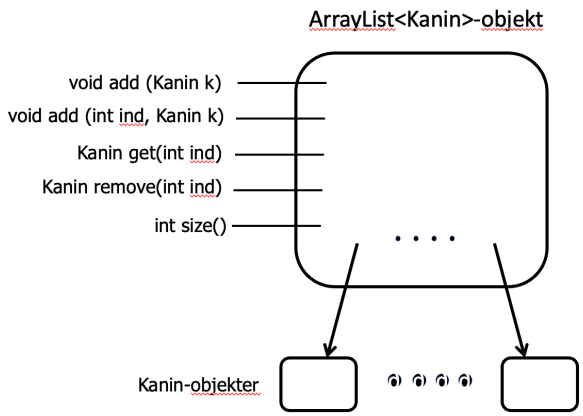
ArrayList

En ArrayList fungerer som en fleksibel array. Alle elementene i en ArrayList er referanser til objekter. Du kan sette disse referansene inn først i listen, sist i listen eller på en gitt indeks. Du kan også ta elementer ut på tilsvarene måter.

Siden Java er et sterkt typet språk (vi skal etter hvert i IN1010 finne ut hva dette betyr) må vi si fra hva slags (hvilken klasse) objekter referansene i listen vi lager kan peke på. Hvis vi f.eks. har deklarert en klasse Kanin, kan vi deklarere en liste med navnet kaninene slik:

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

Da er det bare lov å lagre (referanser til) Kanin-objekter i denne listen. Det er lov å forenkle deklarasjonen ved å sløyfe den andre forekomsten av «Kanin». Slå opp i Javas bibliotek for å se hvilke tjenester ArrayList tilbyr. Du kan gjøre det ved f.eks. å putte inn i en søkemoter: «Java API 8 ArrayList». Grunnen til at vi leter etter definisjonen i versjon 8 av Java er at vi ikke ønsker en tidligere versjon. Senere og nyere versjoner er helt OK. Figuren nedenfor viser noen av tjenestene ArrayList tilbyr, og skisserer en abstrakt versjon av datastrukturen inne i et slikt objekt.



HashMap

Hvis du ønsker å finne igjen elementer basert på en nøkkel, er en HashMap en mulighet. Den hemmelige datastrukturen i en HashMap virker slik at nøkkelen blir komprimert til et heltall. Dette heltallet blir så brukt til å slå opp i en array der elementet har sin plass. På engelsk kalles dette hashing, og funksjonen som brukes kalles a hash function. På norsk brukes ofte ordet folding (å folde) eller å komprimere. Folding er ikke pensum i IN1010, men å kunne bruke en HashMap er pensum. Folding brukes mange steder i informatikken der en ønsker å slå nesten direkte opp i en array ved å komprimere / folde / hashe en nøkkel.

For spesielt interesserte: Noen ganger vil to forskjellige nøkkelverdier foldes til samme verdi. Da vil de slå opp på samme sted i arrayen, og vi må tilsynelatende lagre de to forskjellige elementene på samme sted i arrayen. Dette er ikke mulig og vi må derfor ha en strategi for slike kollisjoner. Hashing er pensum i IN2010 (Algoritmer og datatstrukturer).

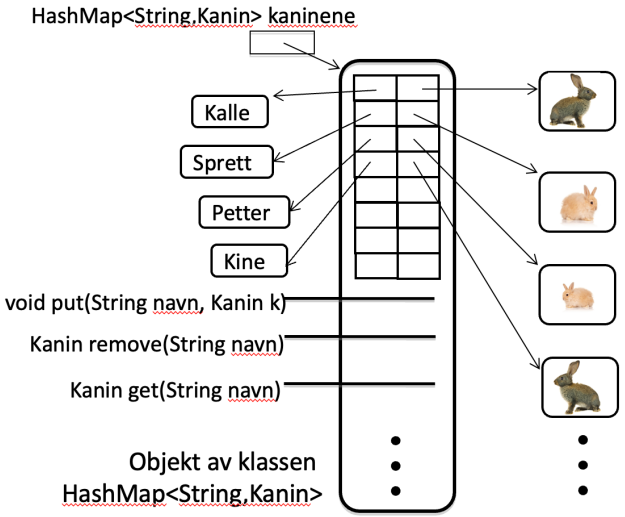
I en HashMap kan alle typer objekter brukes som nøkkel, men i IN1010 bruker vi nesten alltid String-er som nøkler. Hvis du taster «Java API 8 Object» inn i en søkemotor, vil du finne klassen Objekt. Denne skal vi snakke mye om i IN1010. En av operasjonene vi kan gjøre på et objekt av klassen Object er å spørre om dets hash-verdi: public int hashCode(). Når vi lærer om subklasser i februar vil vi lære at alle andre klasser arver metodene til klassen Object. F.eks. vil String-klassen arve metoden public int hashCode().

Hvis kaninene våre har unike navn kan vi lagre og finne dem igjen ved å bruke navnet som nøkkel. Vi kan bare bruke nøkler som er unike i en HashMap. Siden kaninenes navn er en String, deklarerer vi en HasMap av kaniner slik:

```
HashMap<String,Kanin> kaninene=new HashMap<String,Kanin>();
```

Også her kan vi sløyfe det som står inne i den siste < > -parentesen.

Nedenfor ser du et forslag til en illustrasjon av et HashMap-objekt som kan ta vare på par av navn og kaniner. Vi vet ikke hvordan datastrukturen inne i objektet er, men vi spekulerer i at det kan være noe som ligner på to arrayer ved siden av hverandre der den venstre inneholder nøklene (navnene) og den andre inneholder referanser til de tilsvarende kanin-objektene.



6 Om viktigheten av forenkling og abstraksjon – splitt og hersk!

Husk at når du tegner Java datastrukturer skal dette være et hjelpemiddel, ikke en byrde. Igjen kan det påpekes at du ikke behøver tegne mer enn det som er nødvendig for at du selv eller en annen som skal lese tegningene, kan forstå dem.

Når du skal lære å programmere er det svært viktig at du skjønner nøyaktig hva som skjer i datamaskinens primærlager når et Javaprogram blir utført. Når du skal løse et problem må du først finne ut hvordan datastrukturen som løser problemet skal være, og hvordan den skal utvikle seg når programmet utføres. Når du har dette klart for deg kan du skrive programmet som løser oppgaven ved å først opprette og deretter manipulere denne datastrukturen. Underveis i programmet vil du sikkert også opprette nye datastrukturer som du manipulerer og forandrer.

Ofte kan den totale datastrukturen som trengs bestå av mange deler og være svært kompleks. Da er det viktig å ikke prøve å forstå alt på en gang, men dele programmet opp i deler og skjule (ikke bry seg om) detaljene i en del for resten av programmet. Dette kalles gjerne å abstrahere. Resten av programmet kjenner bare *grensesnittet* og de tjenestene hver av de

andre komponentene (objektene) i programmet tilbyr. Resten av programmet har et abstrakt syn på hver av de andre komponentene.

Om du programmerer ”nedenfra og opp” lager du først moduler med mange detaljer i. Når du senere skal bruke disse modulene skal du glemme disse detaljene, og bare bry deg om grensesnittene til disse modulene (en modul kan f.eks. være en klasse); du skal abstrahere. Eksempelet med klassene Beregn og BeregnT illustrerer dette. Om metodene inne i klassen er programmert slik eller sånn spiller ikke så mye rolle for den som bruker klassen (det kan imidlertid innvirke på plass- og tidsbruken). Brukeren må forholde seg til grensesnittet, dvs. metodenes signatur og hvordan metodene virker.

Om du programmerer ”ovenfra og ned”, postulerer du at du har en modul med det grensesnittet du ønsker deg, og mens du programmerer skriver du ned signaturene og virkningene til metodene som utgjør grensesnittet til de modulene/klassene du trenger. Så blir det din (eller en annens) oppgave å implementere disse klassene senere.

Lykke til med programmeringen din og med kommunikasjon og samarbeid med andre i programutviklingsprosessen.

Appendiks: Om synligheten av variabler - skop (Engelsk: scope)

Variabler som er deklarert inne i en blokk er synlige i resten av blokken.
Variabler inne i en metode er synlige i hele metoden (etter at de er deklarert).
Variabler i metoder kaller vi i IN1010 lokale variabler.
Når en blokk eller en metode terminerer forsvinner alle variabler som er opprettet.

Inne i en metode i et objekt er alle variabler og metoder i objektet (instansvariabler og instansmetoder) synlige, i tillegg til alle variabler og metoder i klassedatastrukturen.

Så lenge synligheten ikke er restriktert (med for eksempel ”private”) er alle metoder og variabler i klassedatastrukturen tilgjengelig over alt i programmet ved å si

<Klassenavn>.<navnPåMetodeEllerVariabel>.

Alle regler for variabler gjelder også for konstanter.

SLUTT