

Litt om datastrukturer i Java

Av Stein Gjessing, Institutt for informatikk, Universitetet i Oslo

1 Innledning

Dette notatet beskriver noe av det som foregår i primærlageret når et Javaprogram utføres. Det er denne beskrivelsen eller modellen vi skal bruke i INF1010. Modellen kan illustreres ved hjelp av papir og blyant. Papiret er vår modell av primærlageret (RAM) i datamaskinen. Du er prosessoren(e) som utfører Java-programmet ved å tegne på papiret. Av og til må du også forandre ting på papiret, og da trenger du et viskelær.

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 se bort fra i resten av dette notatet. Vi skal tenke oss at du, som prosessor, kan utføre Java-programmer direkte slik de er skrevet. Når du skal starte å utføre et Javaprogram har du altså fire ting: programmet, et blankt ark, en blyant og et viskelær.

I eksemplene i dette notatet er identifikatorer som vi som programmerere selv lager, skrevet på norsk. Den pedagogiske hensikten med dette er at leseren da lett skal kunne slutte at alle engelske navn 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 programmer skal leses av personer som ikke kan norsk, og da 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. Konstanter skrives med bare STORE bokstaver. Sammensatte ord skrives i ett med storBokstav for hvert nytt ord.

I eksemplene i dette notatet 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 programmere med kommentarer og dokumentasjon (Java-doc).

2 Om statiske variable/konstanter og statiske metoder.

Det første som skjer når ethvert Javaprogram starter er at det for hver klasse settes av plass til alle statiske variable, statiske konstanter og statiske metoder som finnes i alle klassene som brukes av programmet. Denne plassens settes av inne i datastrukturer vi kaller *klassedatastrukturer*.

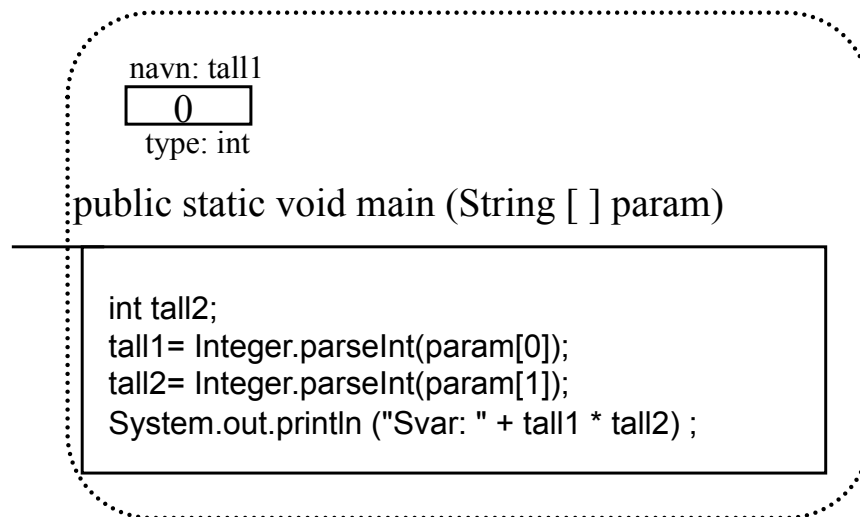
La oss se på et program som multipliserer to tall:

```
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);
    }
}
```

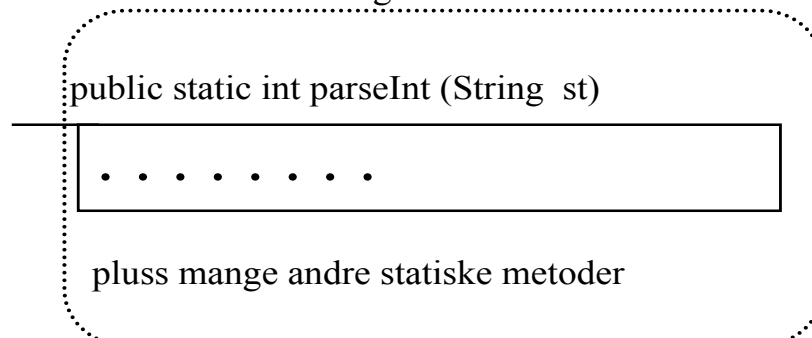
I det Java-programmet starter lager kjøresystemet en klassesdatastruktur for alle klasser som programmet skal bruke. I INF1010 tegner vi disse datastrukturene som prikkete rektangler med avrundede hjørner. I programmet over brukes tre klasser: Mult, Integer og System. Vi må altså tegne tre klassesdatastrukturer.

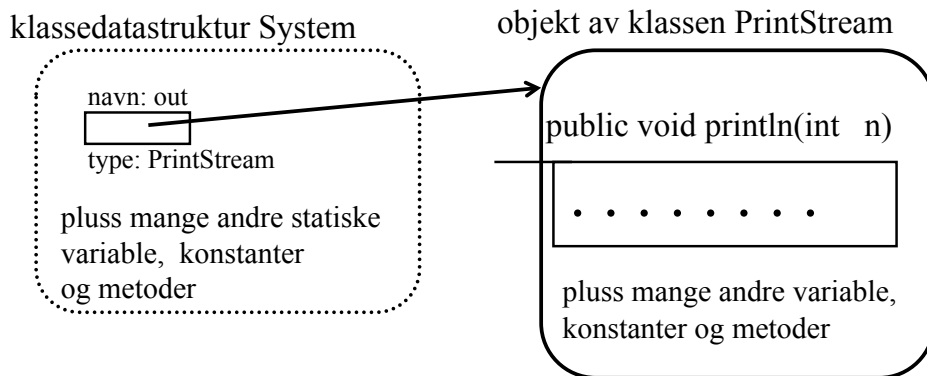
Når vi tegner opp utførelsen av et Javaprogram på denne måten kan det bli svært mye å tegne. Til enhver tid behøver du bare tegne det du (og de du samarbeider med) trenger for å skjønne hva som foregår. I begynnelsen kommer vi (og det bør du også) være svært nøyaktig med tegningene. Etter hvert som vi blir flinkere kan unødvendige og opplagte detaljer utelates. Allerede i tegningene nedenfor utelater vi detaljer i klassesdatastrukturene for Integer og System som vi ikke trenger i dette programmet.

klassesdatastruktur Mult



klassesdatastruktur Integer





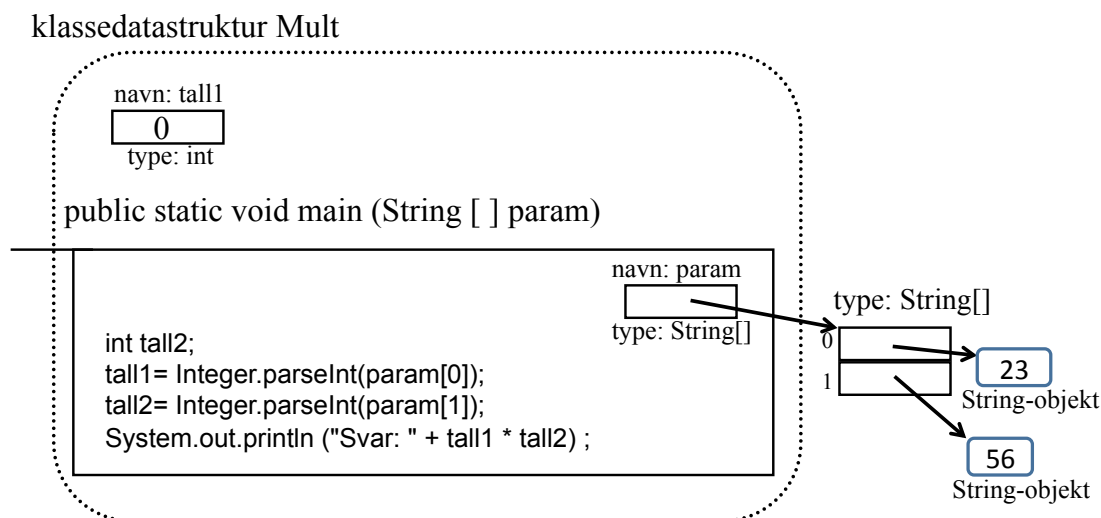
Alle variable og konstanter i klassesdatastrukturene blir initialisert i det programmet starter opp. Hvis vi ikke har gitt variable noen startverdi vil de blitt gitt sine standard startverdier (f.eks. får int verdien 0 og pekere verdien null). I figuren over ser vi klassesdatastrukturen til Mult etter at den statiske variabelen tall1 er blitt initialisert til 0.

Det er Javas kjøretidsystem (run time system) 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 variable opprettet. Vi kommer tilbake til dette i forbindelse med opprettelse av klasseobjekter med metoder i. I tillegg blir det litt mer om dette i kapittel 4.

I det en metode starter opp, vil alle *formelle parametre* oppstå som lokale variable 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 peker til argumentene til kallet på utføringen av programmet. Programmet over starter vi f.eks. ved å skrive

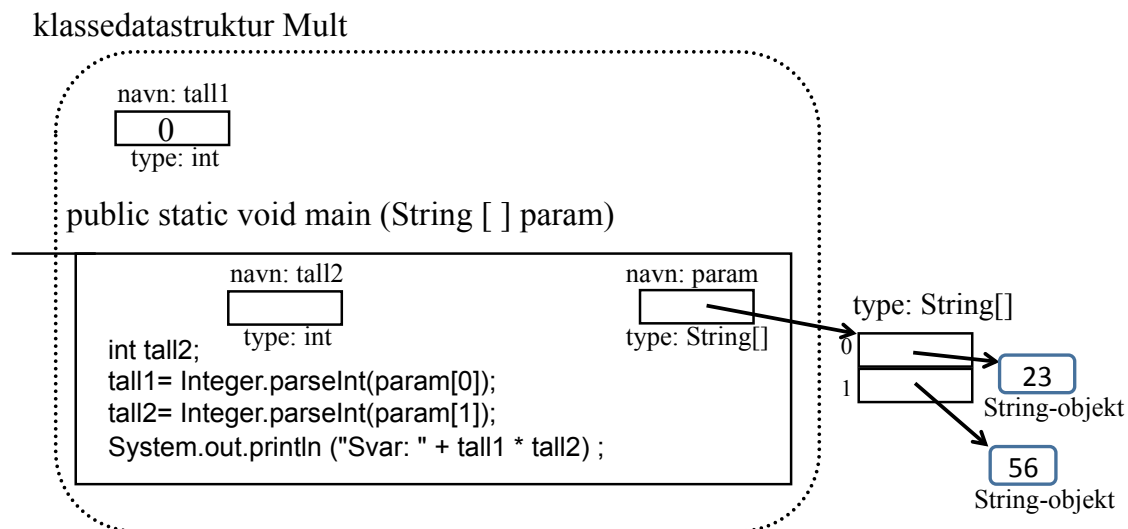
```
>java Mult 23 56
```

I det main-metoden starter opp er situasjonen denne:



Legg merke til at den boksen som er kommet øverst i høyre hjørne av metoden main er den formelle parameteren som heter param og som nå er blitt en lokale variabelen med samme navn og med typen tabell (array) av pekere til String-objekter og med initialverdi en peker til tabellen som inneholder parametrene til programmet.

Det neste som så skjer i main-metoden er at deklarasjonen `int tall2` blir utført. Det blir da 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. Variable i metoder blir ikke initialisert uten at vi eksplisitt ber om det.

At vi har deklartert `tall1` som statisk variabel i klassen, og `tall2` som lokal variable i metoden `main` har ingen dypere mening. Variabelen `tall1` er ment å vise hvordan en statisk variabel deklarerer og tegnes. Siden `tall1` bare brukes i metoden `main` burde den nok, på samme måte som `tall2`, vært deklartert som lokal variabel i metoden `main`.

Derneft blir disse to setningene utført

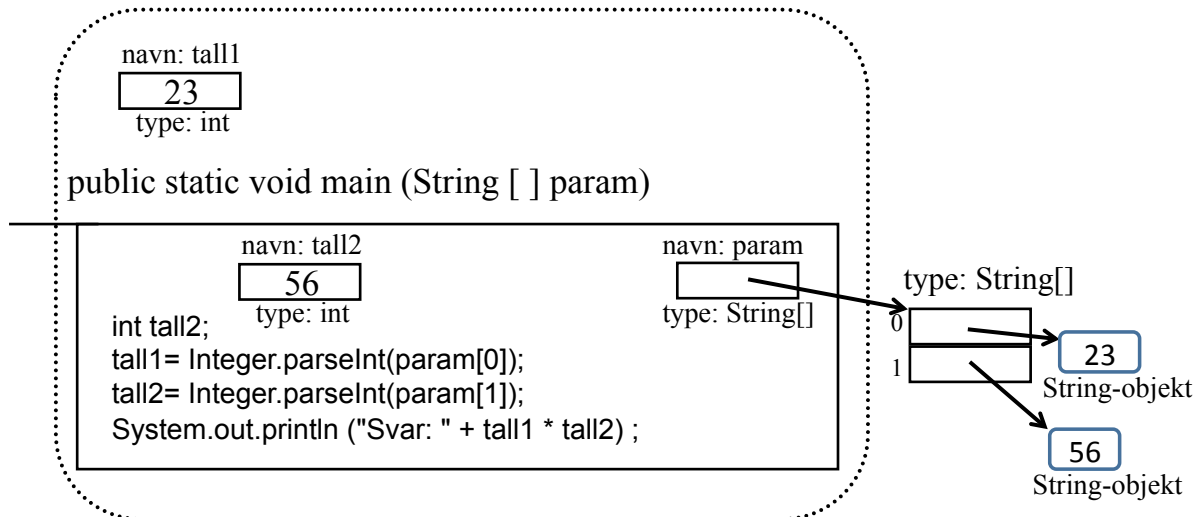
```

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:

klasedatastruktur Mult



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 variable (og konstanter) borte, i dette tilfellet de to variablene param og tall2. I det hele programmet terminerer blir alle klasedatastrukturene også borte.

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

3 Eksempel med klasse-objekt

La oss se på programmet

```
class Regn{
    public static void main (String [ ] param) {
        Beregn1 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 ;
}
}

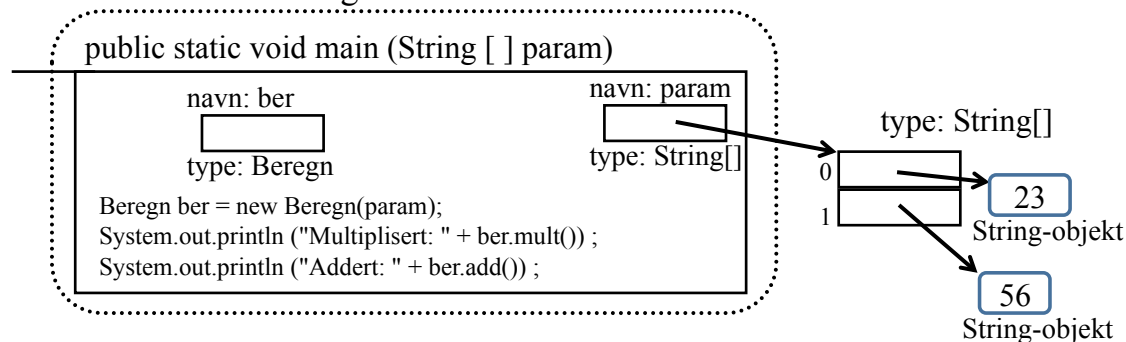
```

La oss kjøre 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 tegner ikke de to klassedatastrukturene til Integer og System denne gangen. Vi tegner heller ikke klassedatastrukturen til klassen Beregn, for den er jo tom (det er ingen statiske egenskaper i denne klassen).

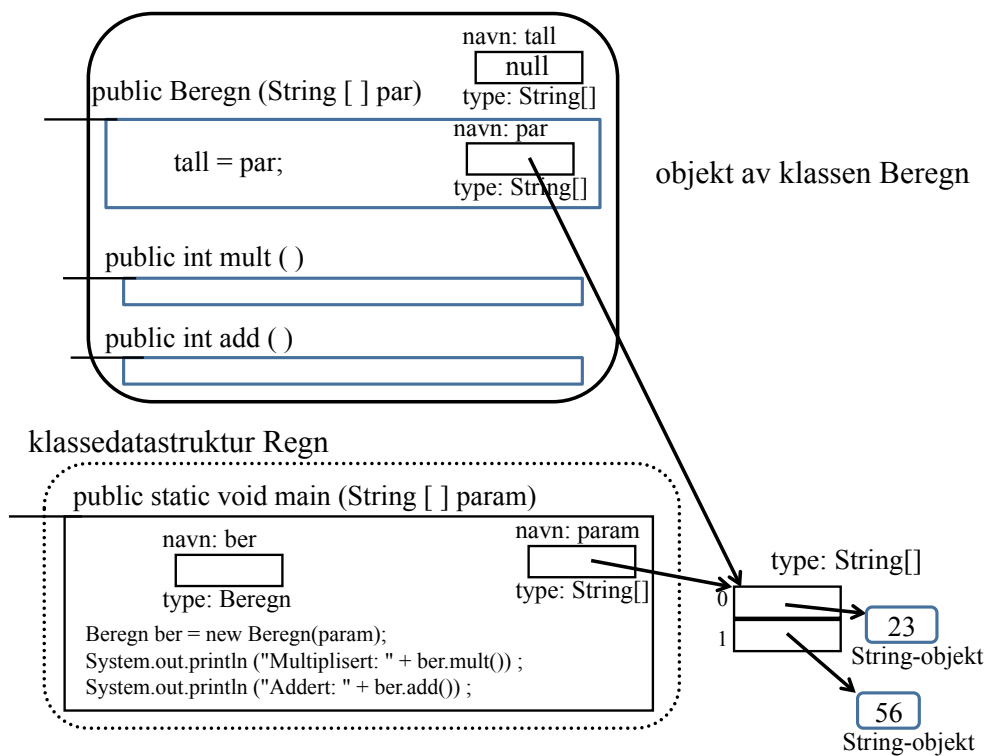
klassedatastruktur Regn



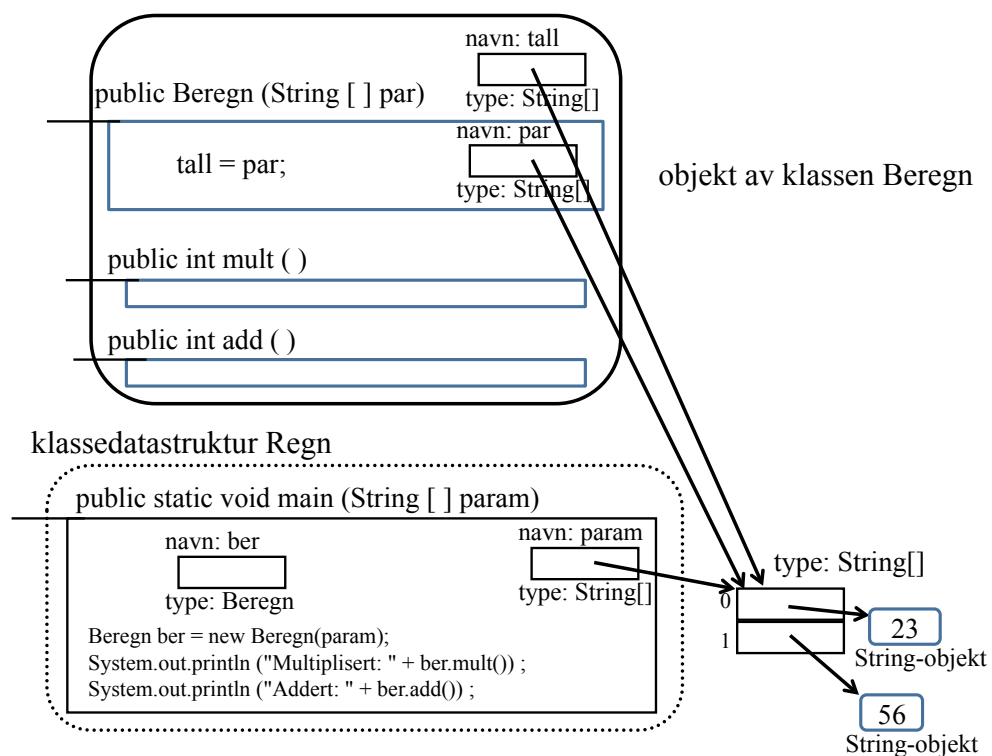
Så utføres deklarasjonen og tilordningen:

```
Beregn1 ber = new Beregn(param);
```

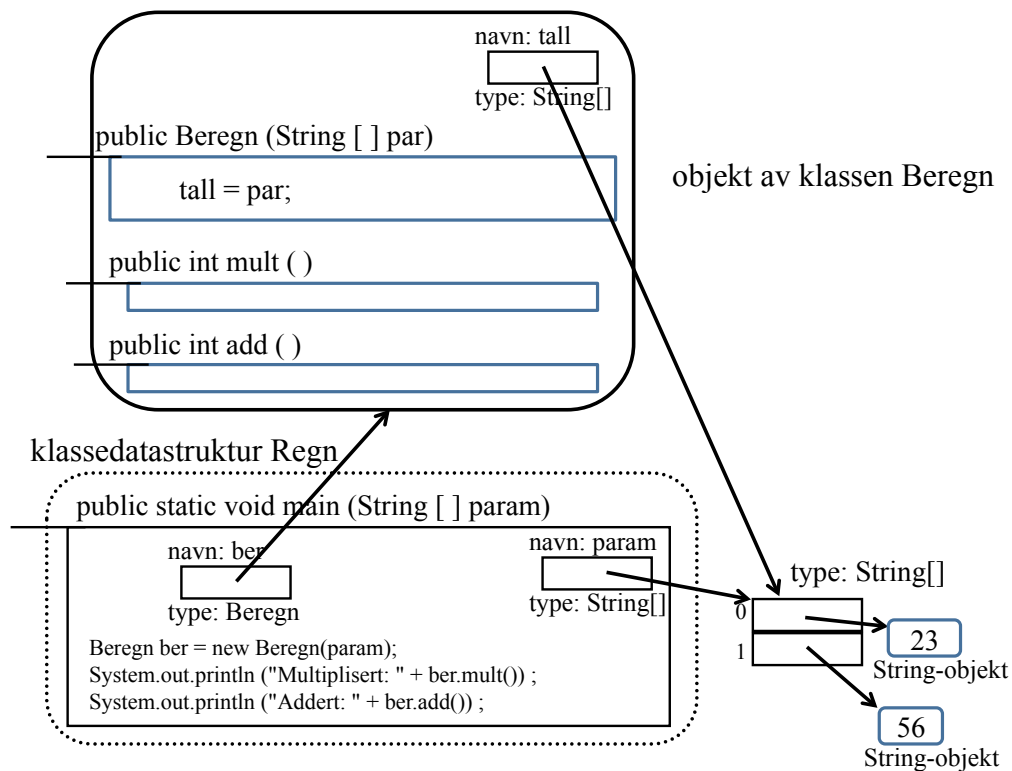
Uttrykket på høyresiden har den viktige sideeffekten å opprette et objekt av klassen Beregn, med aktuell parameter til konstruktøren som er en peker til tabellen av to String-pekere. Men før konstruktøren blir utført blir det laget en variabel inne i objektet med navn tall, og med initialverdi null:



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

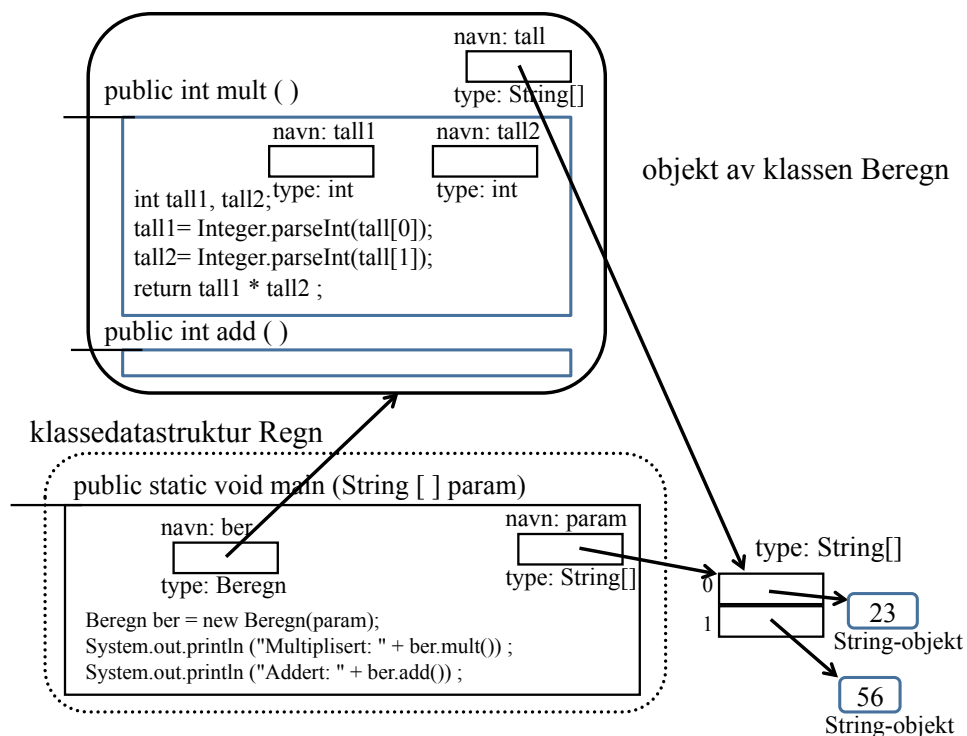


Etter at tilordningen er utført terminerer konstruktøren, og objektet er ferdig laget. I `main` vil så en peker til det nylagde objektet bli tilordnet variabelen `ber`:

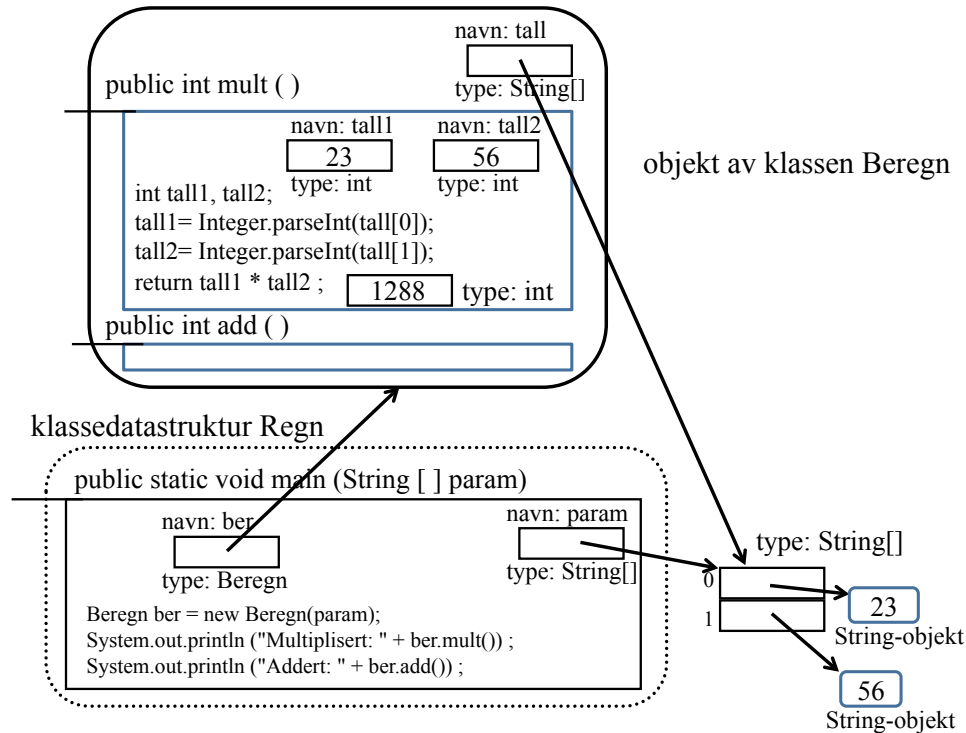


Når objektet er laget vil konstruktøren aldri kunne bli kalt igjen, og vi tegner den derfor ikke mer i objektet.

I neste setning i main skal uttrykket `ber.mult()` beregnes. Vi følger `ber`-pekeren til et objekt, og dette objektet inneholder metoden `mult` (oppgave: hvordan kan vi være sikre på at det er en metode med navnet `mult` der?), som så blir utført. Metoden har ingen parametre, men det første som skjer i metoden er at to variable blir deklartert:

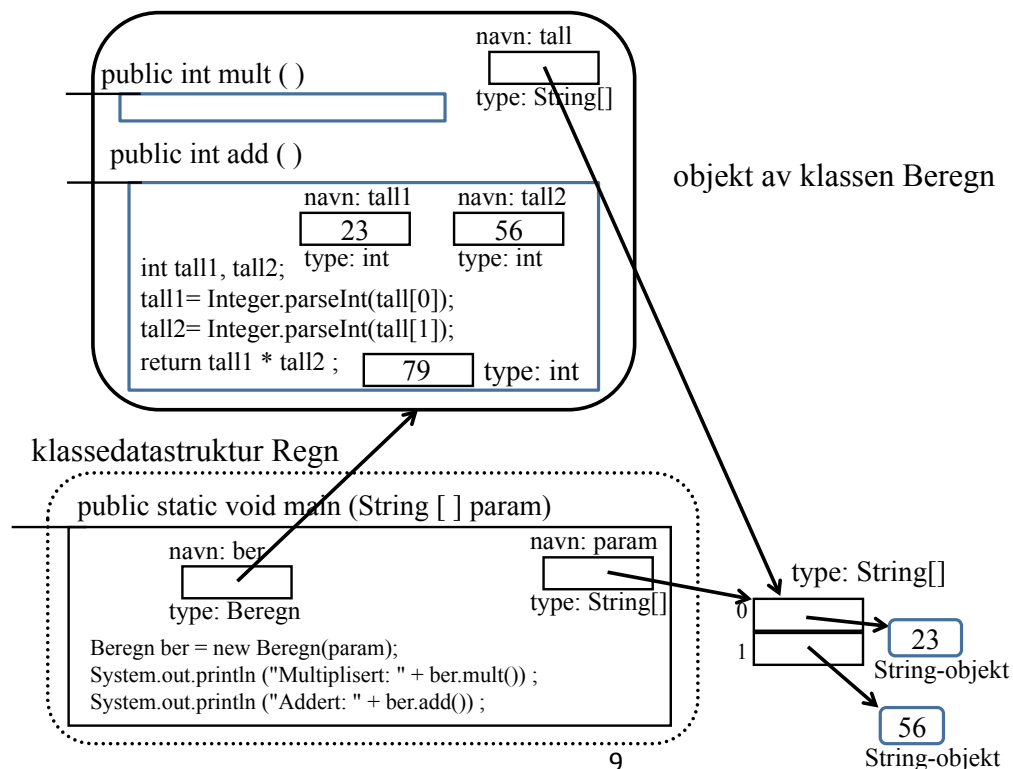


Disse to variable får ingen startverdi fordi de er inne i en metode. De to tilordningene gir dem imidlertid verdier, og til slutt beregnes $tall1 * tall2$, og resultatet av denne beregningen skal returneres som metodens resultat:



Når metoden har returnert blir de lokale variablene `tall1` og `tall2` fjernet, og metoden `mult` sitter igjen uten lokale variable. Metoden `main` skriver så ut på skjermen: `Multiplisert: 1288`.

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



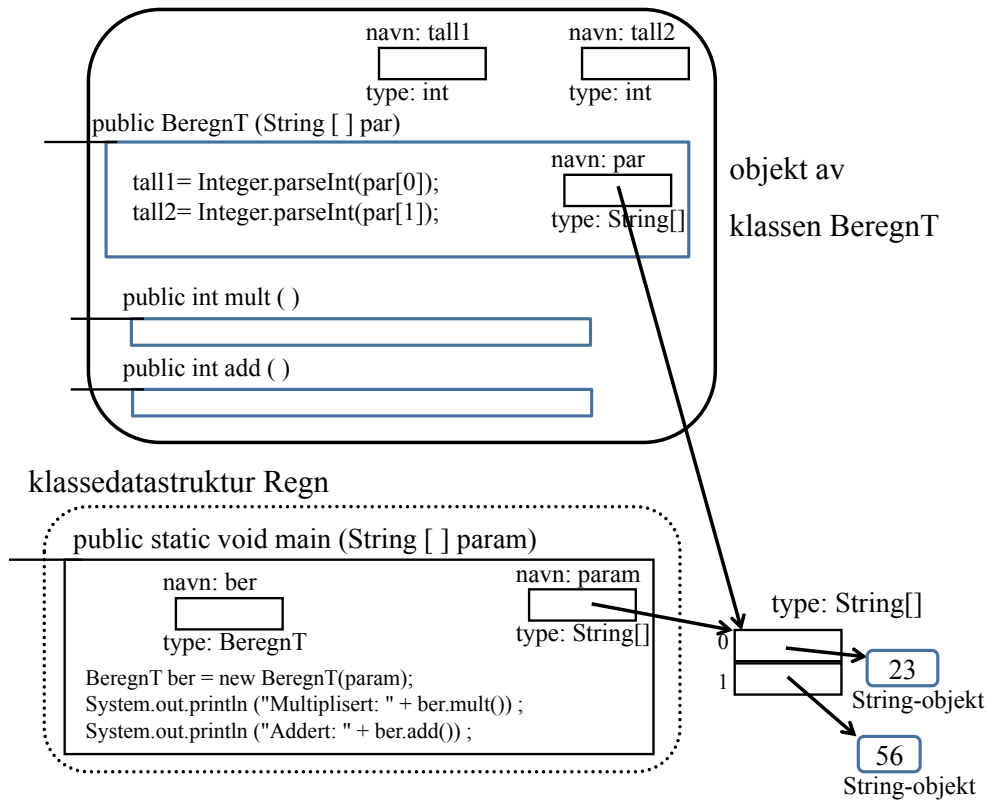
Når metoden `add` er ferdig blir også denne metodens lokale variable borte, og `main` skriver ut: `Addert: 79`. Deretter avsluttes metoden `main` og alle lokale variable blir fjernet (`ber` og `param`), og kontrollen går tilbake til kjøretidssystemet (som er en del av JVM) som avslutter hele Java-maskinen.

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

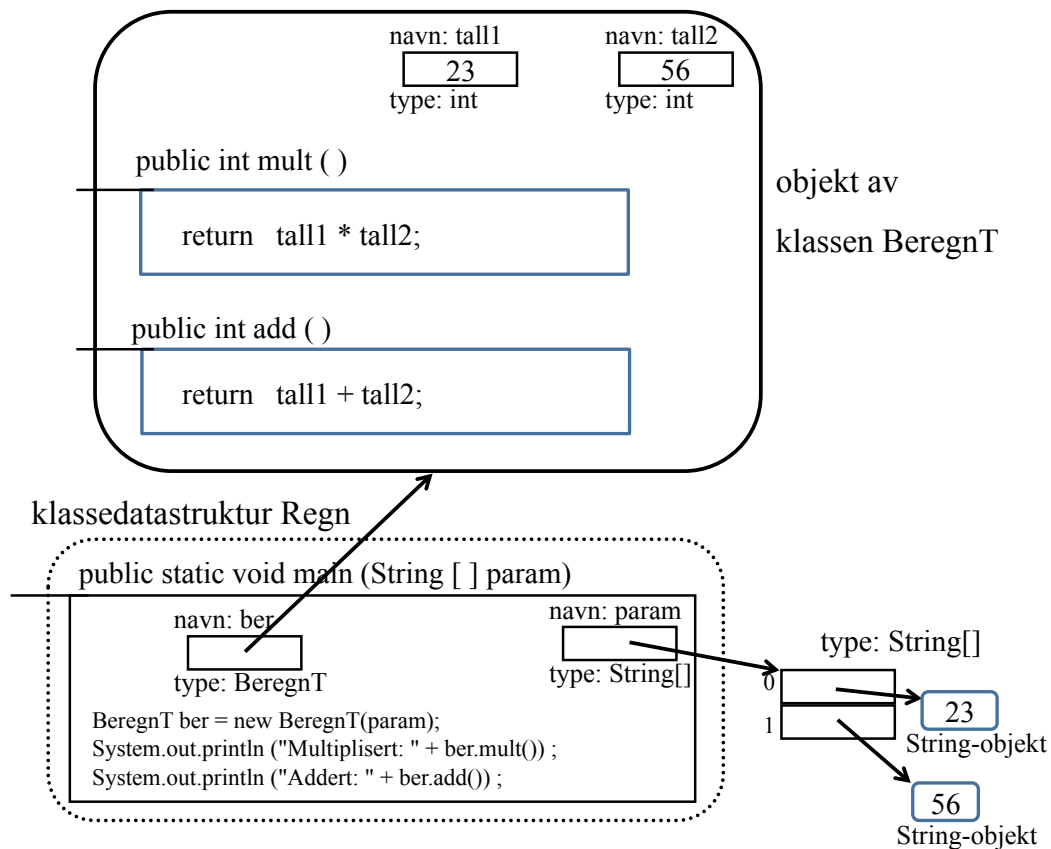
```
class Regn{
    public static void main (String [ ] param) {
        BeregnT ber = new BeregnT(param);
        System.out.println ("Multiplisert: " + ber.mult()) ;
        System.out.println ("Adert: " + ber.add()) ;
    }
}
class BeregnT {
    private int tall1, tall2;
    public BeregnT (String [ ] 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, samme antall og type parametre (i samme rekkefølge) og returnerer et resultat av samme type. Forskjellen på de to klassene `BeregnT` og `Beregn` er de private variable 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 pekeren 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?

4 Om metoder og metodeinstanser

Vi har sett at når en metode kalles oppstår det en metodeinstans som inneholder alle lokale variable (inklusive parametre) i metoden. Når vi tegner klassesdatastrukturer og objekter tegner vi alltid opp variable, konstanter og metoder også (de statiske egenskapene tegner vi i klassesdatastrukturene, de andre egenskapene tegner vi i alle objektene som blir opprettet). Å tegne en (ikke-statisk, gjerne kalt en objekt-) metode i alle objektene som opprettes er egentlig ikke riktig. Koden til metoden finnes bare ett sted, og når det oppstår mange objekter er det bare variable og konstanter som dupliseres 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 variable og konstanter metoden har tilgang til i det objektet metodeinstansen omslutes av.

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 variable. Når metodeinstans nr. 7 er ferdig utført, vil programkontrollen gå tilbake til kalletstedet i metodeinstans nr. 6. Kaller denne instansen (nr. 6) enda en metode vil også dette bli metodeinstans nr. 7.

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*). I dette tilfellet er det heldigvis kjøretidsystemet som holder orden på stakken av metodeinstanser. Senere i INF1010 skal du lære å lage egne datastrukturer som er organisert som stakker.

5 Etterord

Det er 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 gjør dette.

På den annen side er det også viktig å skjule (å ikke bry seg om) detaljer. Det er helt umulig å holde oversikt over alle detaljer i et program på en gang. Derfor må du programmere en ting om gangen. Om du programmerer ”nedenifra 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). 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 tidsbruken). Brukeren må forholde seg til grensesnittet, dvs. metodenes signatur og hvordan metodene virker.

Om du programmerer ”ovenifra 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.