ex2references

Joachim von Hacht

Effektivitet

```
int[] a1 = new int[1000000000];  // 32 Gb !!!
int a2[];

a2 = a1;           // Assignments copies! Much to copy!!
a1 = add(a1, a2);  // Method call/return copies!

// This is *not* feasible ... !!!
```

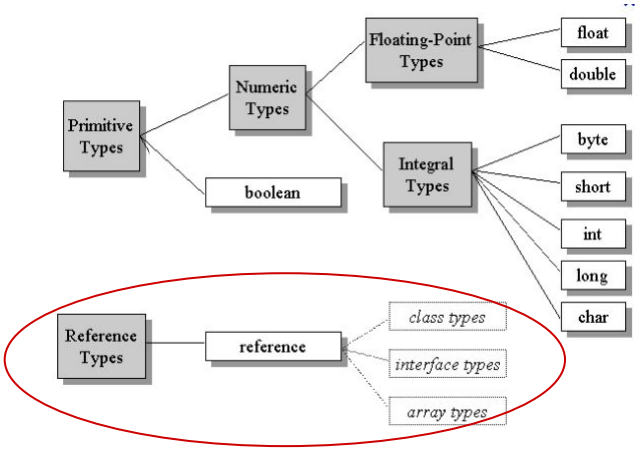
Vi har sett att vid tilldelning och metodanrop/återhopp sker det en kopiering av data

- Innebär att data kopieras från en plats i minnet till en annan plats
- Tilldelar vi en int till en heltalsvariabel kopierar vi 32 bitar/4 bytes

Antag att det som skall kopieras är mycket stort i bytes t.ex. en bild eller en video

- Kan röra sig om många MB eller GB
- En ren kopiering skulle i detta fall bli väldigt resurskrävande och ineffektiv.
- En lösningen består i att använda referenser … mer strax.

Referenstyper



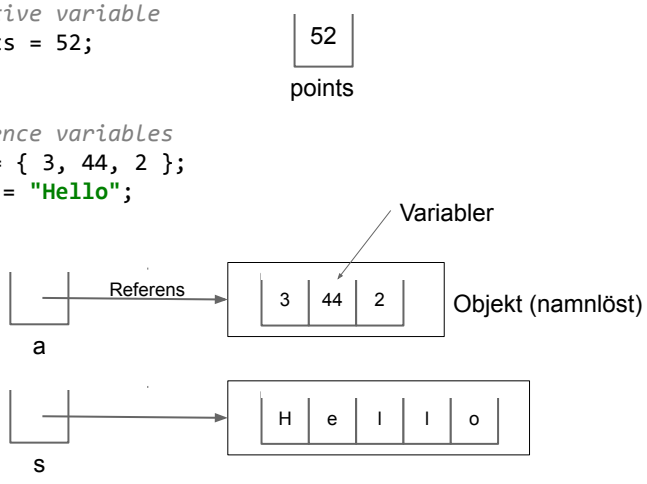
3

Referenstyper

- Allt som inte är primitiva typer är referenstyper.
- Kan vara t.ex. klasstyper, som String eller Dog, eller array-typer t.ex. int[].

Referensvariabler

```
// Primitive variable
int points = 52;

// Reference variables
int[] a = { 3, 44, 2 };
String s = "Hello";
```



Variabler med primitiv typ, **primitiva variabler**, innehåller värdet, värdet finns i variabeln (t.ex. värdet 52 en int)

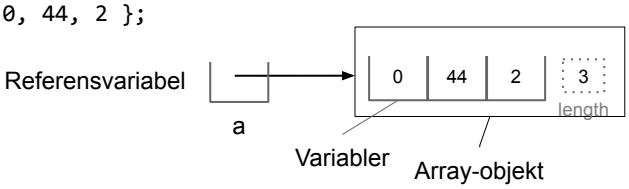
Variabler deklarerade med referenstyper innehåller inte värdet …

- … de innehåller en **referens** till ett namnlöst objekt med variabler
  - Vi säger också att referensen "pekar" på ett objekt.
- Variabler med referenstyper, kallas för **referensvariabler**
  - Enda sättet att komma åt objektet (variablerna) är via referensen (genom att använda variabelnamnet)
  - Tappar vi referensen i referensvariabeln är objektet oåtkomligt.
    - Objektet kommer då automatiskt att tas bort ur minnet, **skräpsamlas (garbage collect)**

I bilden: int[] och String är inte primitiva typer alltså är de referenstyper.

Array-Objekt

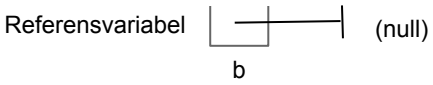
```
// Declaration and initialization, array-object created
int[] a = { 0, 44, 2 };
```



Variabler

Array-objekt

```
// Declaration, NO array-object!
int[] b;
```



5

En deklaration av en array-variabel ger en (enda) referensvariabel.

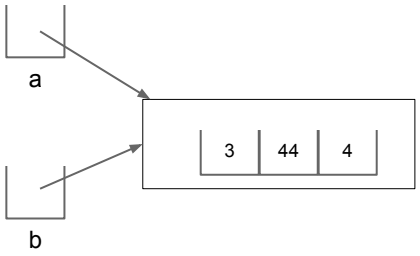
- Om vi initierar variabeln kommer den att innehålla en referens till ett (namnlöst) **array-objekt** som i sin tur innehåller ett antal (namnlösa) variabler
- Vi kan bara nå objektet indirekt, via referensen.
  - Indexering innebär: Gå till objektet, välj variabel utifrån index
  - Tappar vi bort referensen har vi tappat bort objektet, vi kan aldrig komma åt det igen.
- Vi kan välja om vi vill att variabeln skall peka på ett array-objekt eller ej
  - Om inte sätter vi värdet till **null** (mer strax).
- Array-objektet innehåller en konstant variabel: lenght, (som vi sett tidigare) för att komma åt den använder vi punktnotation.
  - Vi ritar aldrig ut length förutom i denna bild.

Att det skapas ett objekt i samband med deklaration och initieringen kallas att objektet instansieras (objektet är en instans av typen array)

- Samma effekt har operatorn new, den instansierar ett objekt som vi därefter kan initiera

Array-variabler och Tilldelning

```
int[] a = { 3, 44, 2 };

int[] b;

b = a;   // Oops!
```

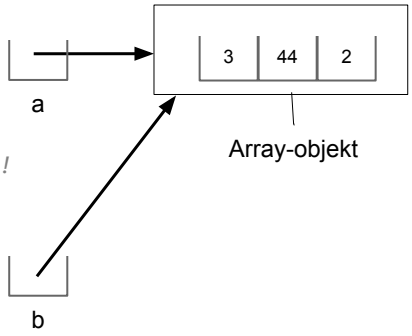


Tilldelning:

- Tilldelning innebär alltid: Kopiera från höger sida till vänster!
- Eftersom array-variabler är referenser kommer en tilldelning att göra så att två referenser pekar på samma array-objekt, referensen kopieras!

Arrayer-variabler och Likhet

```
// Declaration and initialization
int[] a = { 3, 44, 2 };
```




```
// Assign, reference copied!
int[] b = a;
```



```
out.println(a == b);   // true (identical)
```

7

Referenslikhet

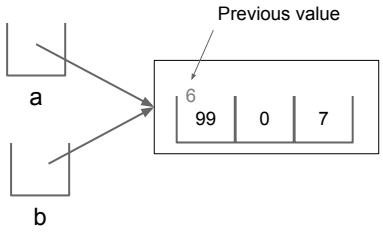
- Om två variabler innehåller samma "värde" är de lika (d.v.s. som vanligt)
- Variablerna i bilden innehåller samma värde, nämligen en referens till objektet
  - Denna typ av likhet kallar vi referenslikhet (equal by reference)
- Vill vi definiera någon annan typ av likhet för arrayer, t.ex. lika långa eller alla har samma värde för samma index, får vi själva implementera detta (t.ex. skapa en metod)

Att tilldelning och likhet för referenser får en speciell betydelse sammanfattas som **referenssemantik**

- Semantiken (betydelsen) blir annorlunda pga av vi använder referenser
- För primitiva typer säger man **värdesemantik**

Alias-problem

```
a = b;


a[0] = 99;


// b changed!!!
if( b[0] == 99 ){
    // true
}
```

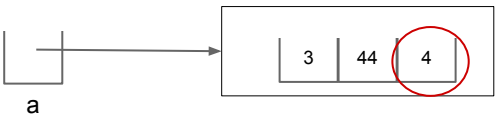


Att flera referenser kan peka på samma objekt innebär att det finns flera sätt att ändra variablerna i objektet.

- Ändringen kan ske “bakom ryggen” på oss.
- Kallas **alias-problem,** den ena referensen är ett alias för den andra
- Kan leda till svårlösta fel i program (går inte att undvika i språk med referenser)

Konstanta referenser

```
// Constant reference variable a
final int[] a = { 3, 44, 2 };

// Error! a is final
a = new int[5];

// Ok, variables in object not final
a[2] = 4;
```

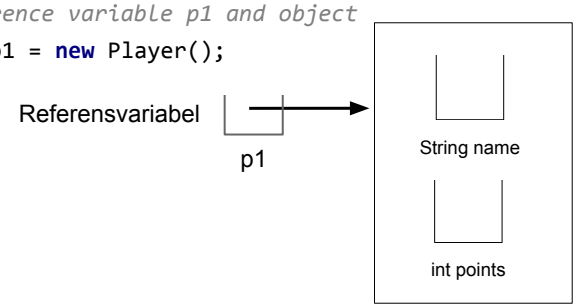


En konstant referensvariabel kan inte ändras.

- Objektet den referera kan däremot ändras!

Variabler för Objekt

```
// Reference variable p1 and object
Player p1 = new Player();
```




```
// Variable but no object!
Player p2;
```

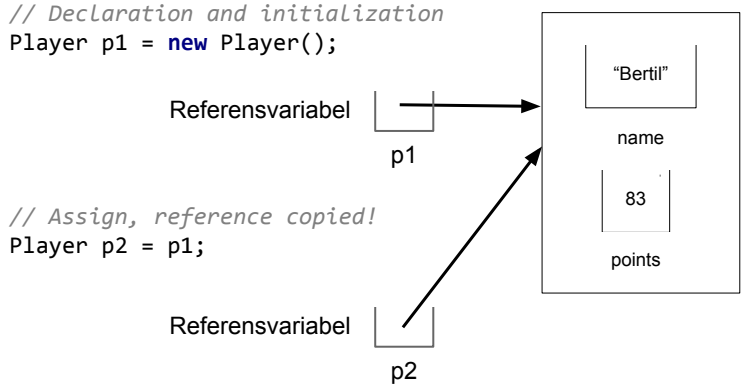
10

Detta fungerar på samma sätt som för arrayer!

- Variabler för objekt är referensvariabler, klasstyper är referenstyper.
- En deklaration ger en referensvariabel (bara).
- Initieringen (till höger) instansierar ett namnlöst objekt och returnerar en referens till detta.
  - Värdet är alltså en referens …
  - .. sidoeffekten är att ett (namnlöst) objekt skapas (i minnet).
- Punktnotation innebär att man går till objektet och därefter väljer variabel utifrån namn m.h.a. punktoperatorn.
  - Punkten använd på objektet (inte på variabel eller referens)!

Tilldelning och likhet med Objektvariabler

```
// Declaration and initialization
Player p1 = new Player();

// Assign, reference copied!
Player p2 = p1;

out.println(p1 == p2);   // true
```



Fungerar på samma sätt som arrayer, det är referensen som kopieras!

- D.v.s vi får två olika variabler som refererar samma objekt
- Ger samma aliasproblematik som för arrayer

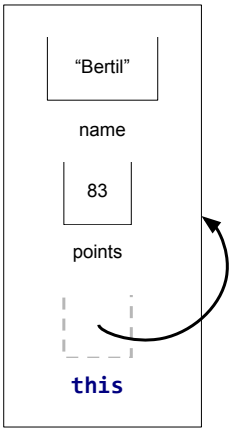
Avreferering

```
class Player {
  String name;
  int pts;
}

Player p = new Player();
int[] a = { 3, 44, 2 };


out.println( p.pts );  // Implicit dereferencing
out.println( a[0] );   // Implicit dereferencing
```

Avreferering innebär (i Java) att programmet implicit (automatiskt) “följer” referensen till objektet (går till) då vi använder t.ex. indexering eller punktnotation. Därefter väljs variabel utifrån index.

- [ ]- och .-operatorn gäller alltså för objektet, inte för referensen (eller variabeln)!
- Detta gäller inte för alla språk i t.ex. C måste man ange att man vill avreferera.

this



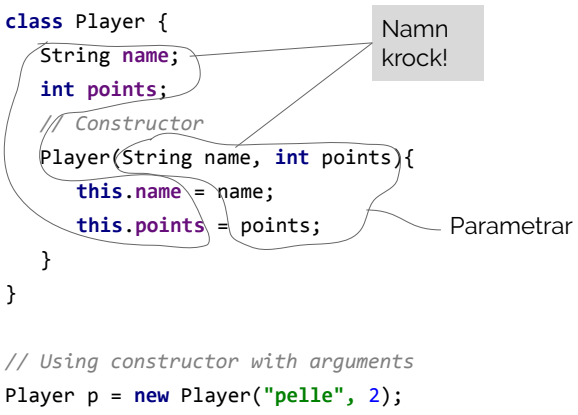

```
// Implicit "this" used
out.println(name);
out.println(points);

// Using this explicit
// Same output as above
out.println(this.name);
out.println(this.points);

// No! can't change
this = ...
```

13

Alla klassobjekt i Java har en dold konstant referensvariabel till sig själv.

- Referensen används automatiskt (osynligt) då man t.ex. anger en instansvariabel, underförstått är det variabeln för det aktuella objektet som avses.
- Ibland använder man referensen explicit genom att skriv **this** i koden.
  - Används t.ex. vid namnkrockar, mer strax
- this är ett reserverat ord
- Synlighetsområdet är i klassen.

Konstruktor och this

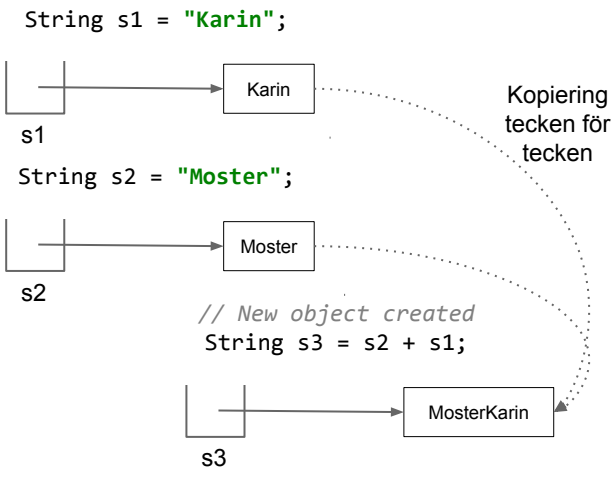


14

Ofta är det naturligt att parametrarna till konstruktorn har samma namn som instansvariablerna de skall initiera (så att vi slipper hitta på olika namn för samma koncept).

- För att skilja på parametrarna och instansvariablerna anger vi i så fall explicit **this** för instansvariablerna.
- OBS! IntelliJ kan generera konstruktor (högerklicka > Generate … ger en konstruktor som använder this)

Strängar och +-operatorn



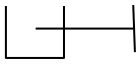
15

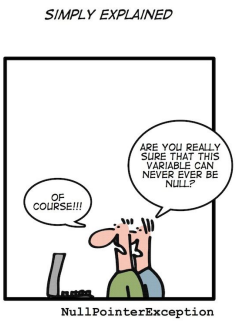
Strängar hanteras som referenser (de är objekt).

- Konkatenering med +-operatorn innebär att ett nytt strängobjekt skapas och en referens till detta returneras. Eftersom strängar inte kan ändras
- Tecknen från operanderna kopieras till det nya objektet.
- +-operatorn kan vara ineffektiv t.ex. i en loop med många varv (kopierar mer och mer för varje varv)
- Mer senare...

null och NPE

```
int[] a;

// Will print null
out.println( a )

// Ok, b also null
int[] b = a;

// ... but this is *bad* !!!
out.println( a.length );   // NPE!!!
out.println( a[1] );       // NPE!!!
```





För att beteckna att en referensvariabel inte refererar något används det speciella värdet **null**

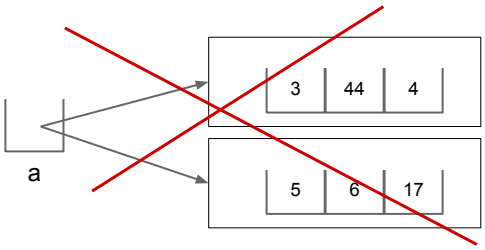
- Att tilldela en variabel null eller att skriva ut ett null-värde går alltid bra..
- Instansvariabler med referenstyp ges automatiskt värdet null

Ett undantag uppstår om man försöker gör något med en null-referens (d.v.s. avreferera den, … var skall man gå??? Finns inget objekt …)

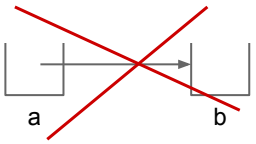
- Vi kommer att få **NullPointerException, NPE!**
- Ett ständigt och stort problem är att hålla reda på om referenser är null eller inte

Saker som aldrig kan hända!

Detta kommer aldrig att ske!
En variabel innehåller ett och endast ett värde (en referens i detta fall)



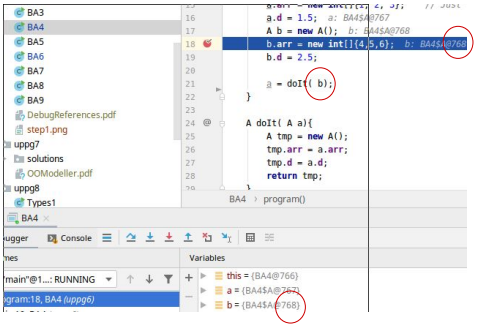
Detta kommer aldrig att ske i Java!
En referens pekar alltid på ett objekt, aldrig på en annan variabel.



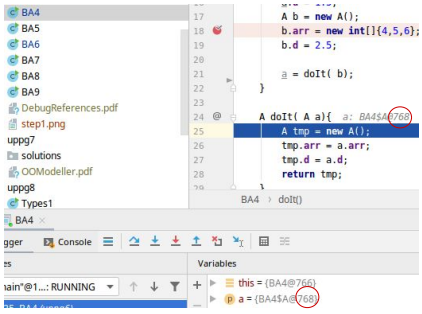
En referens finns alltid i en referensvariabel och pekar alltid på ett objekt (eller inget alls, d.v.s. null)

Spåra Referenser (1)

1) Vad pekar parametern a i metoden doIt på? Starta debugger. Anteckna objektid för variable b eftersom den är argument till doIt() (d.v.s. 768)



2) Stega in i metoden och undersök värdet för parameter a. Eftersom värdet är samma pekar a och b på samma objekt.



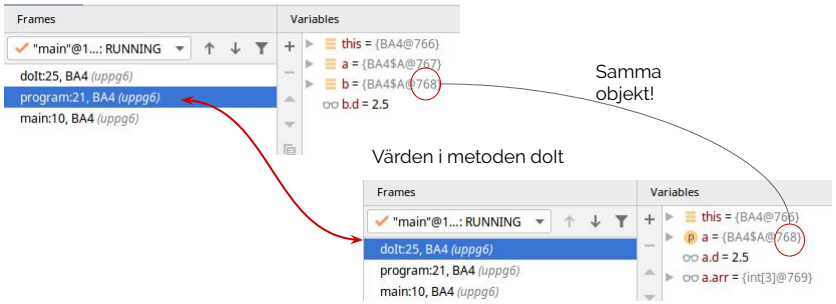
Vi kan använda debuggern i IntelliJ för att se var “pilarna pekar”.

- Alla objekt har ett unikt id under körningen.
- Om värdet i två referensvariabler innehåller samma id pekar de på samma objekt.

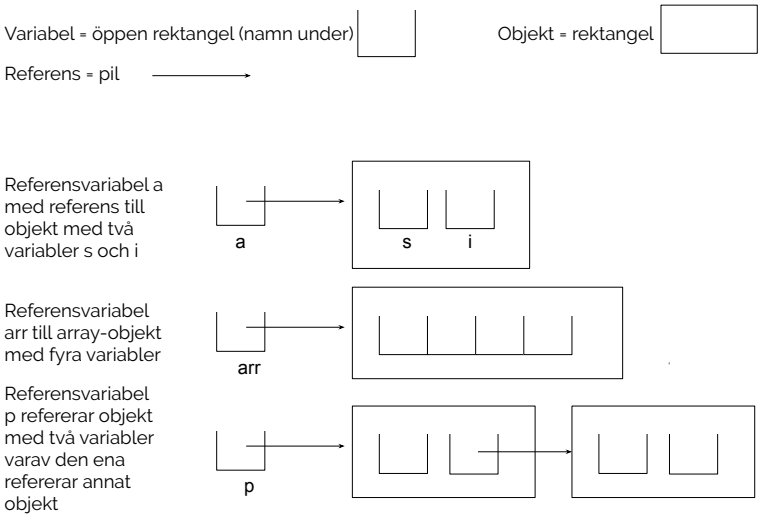
Spåra Referenser (2)

I stället för att anteckna objektid:n kan man hoppa mellan metoder och inspektera värden. Markera metoden i fönstret Frames så visas värden i Variables

Värden i metoden program



Visualisera Referenser



En sammanfattning

- För att visualisera variabler, värden, referenser och objekt använder vi ett “grafiskt språk” enlig bilden

Primitiv kontra Referenstyp

Reference types	Primitive types
Unlimited number of reference types, as they are defined by the user.	Consists of `boolean` and numeric types: `char`, `byte`, `short`, `int`, `long`, `float`, and `double`.
Memory location stores a reference to the data.	Memory location stores actual data held by the primitive type.
When a reference type is assigned to another reference type, both will point to the same object.	When a value of a primitive is assigned to another variable of the same type, a copy is made.
When an object is passed into a method, the called method can change the contents of the object passed to it but not the address of the object.	When a primitive is passed into a method, only a copy of the primitive is passed. The called method does not have access to the original primitive value and therefore cannot change it. The called method can change the copied value.

Sammanfattning