Referenser

TDA548/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 ... !!!
```

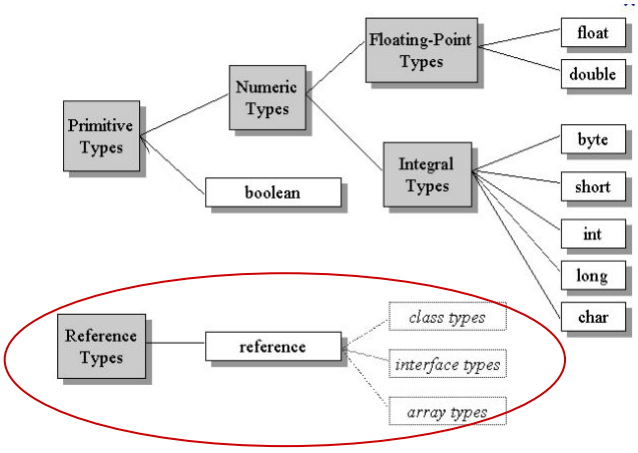
Vi har sett att vid tilldelning och metodanrop/återhopp sker det en kopiering av data

- 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 …

Referenstyper



3

Referenstyper kan vara array-typer (t.ex. int[]), klasstyper (t.ex. String) eller gränssnittstyper (interface)

Man kan skapa nya referenstyper!

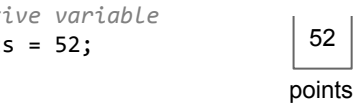
- Typsystemet är utbyggbart för array-, klass- och gränssnittstyper
  - Kallas också **egendefinierade typer** (vi definierar dem)
  - Genom att deklarera arrayer skapas nya typer utifrån en grundtyp.
  - Genom att deklarera klasser och gränssnitt skapas nya klass- respektive gränssnittstyper.

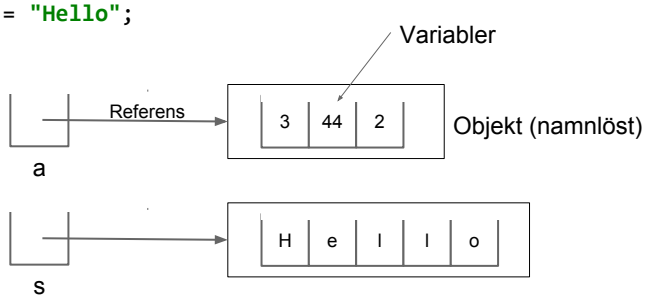
Referenstyper har alla samma min och maxvärden, de är alla lika stora

- 32 bitar för 32-bitars maskiner
- 64 bitar för 64-bitars maskiner

Referensvariabler

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

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



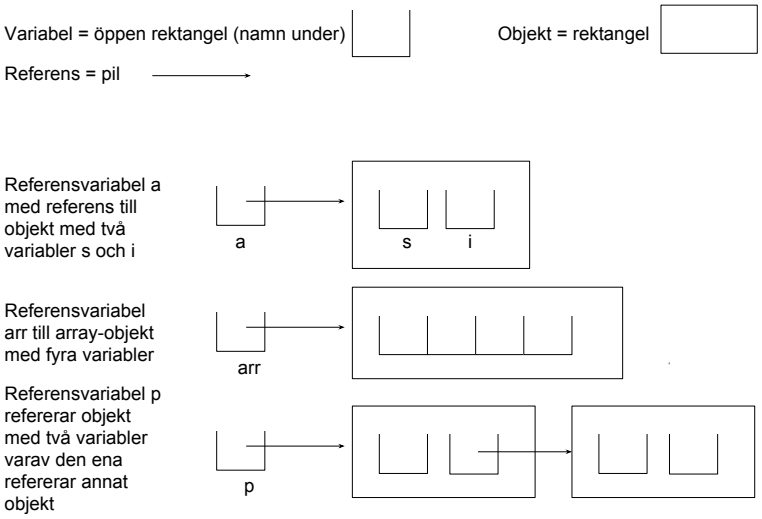


Variabler med primitiv typ, **primitiva variabler**, innehåller värdet, värdet finns i variabeln (t.ex. värdet 52 en int)

Variabler deklarerade med referenstyper innehåller inte värdet …

- … de innehåller en **referens** till ett namnlöst objekt som innehåller variabler med värdet/värdena
  - Vi säger också att referensen "pekar" på ett objekt.
- Variabler med referenstyper, kallas för **referensvariabler**
  - Enda sättet att komma åt det namnlösa objektet 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)**

Visualisera Referenser



För att visualisera referenser, referenssemantik (betydelsen, se vidare nedan) och objekt/variabler använder vi ett "bildspråk".

- Variabler ritar vi som tidigare.

Avreferering

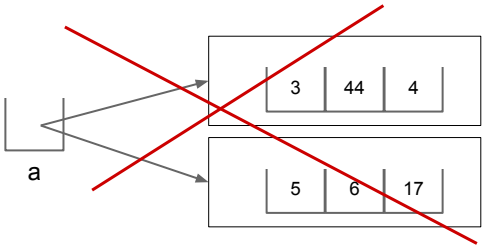
```
class Player {
  String name;
  int pts;
}

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 då vi använder indexering eller punktnotation. Därefter väljs variabel utifrån index eller namn.

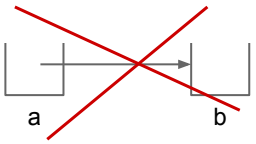
- Så sker alltid i Java!
- [ ]-operatorn och **.**-notationen gäller alltså för objektet, inte för referensen (eller variabeln)!

Saker som aldrig kan hända!

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

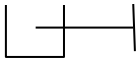


Detta kommer aldrig att ske i Java!
En referens pekar alltid på ett objekt, aldrig på en annan variabel.



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, se Typer.
- 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 kan få … **NullPointerException, NPE!**
- Ett ständigt och stort problem är att hålla reda på om referenser är null eller inte

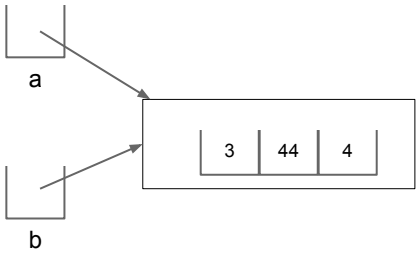
För alla objekt, o, skall gälla att o.equals(null) är false!

- Se bildserie klasser.

En artikel (kanske inte 100% rätt men intressant …)

Referenser och Tilldelning

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

int[] b;

b = a;    // Oops!
```

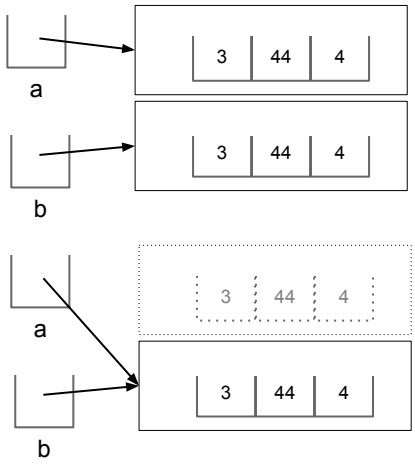


Tilldelning för referensvariabler fungerar som för primitiva typer

- Värde kopieras från vänster till höger MEN …
- .. "värdet" är en referens, det är referensen som kopieras!
- Effekten blir att två referenser pekar på samma objekt!

Referenser och Likhet

```
int[] a = { 3, 44, 4 };

int[] b = { 3, 44, 4 };


// False
out.println(a == b);


a = b;

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



Likhet för referensvariabler fungerar som för primitiva variabler

- Innehållet i variablerna jämförs MEN …
- … innehåller är nu referenser!
- Likhet innebär att referenserna refererar samma sak (pekar på samma objekt)
    - Alt. innehåller samma adress!

Efter tilldelningen, a = b ovan, pekar referenserna på samma objekt.

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

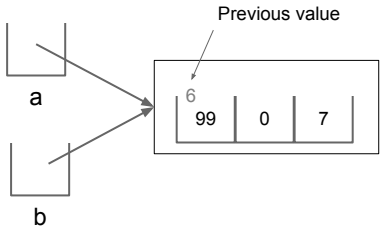
- Semantiken (betydelsen) blir annorlunda pga av vi använder referenser
- För primitiva typer säger man **värdesematik**

OBS! String är en referenstyp, innebär att == inte "fungerar" för strängar!

- Normalt är de ju själva texten vi vill jämföra men == jämför referenserna.
- Se bildserie om strängar.

Alias-problem

```
a = b;


a[0] = 99;


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

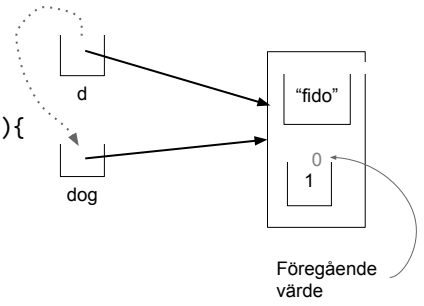


Att flera referenser kan peka på samma objekt innebär att det finns flera sätt att ändra variablerna i objektet.

- 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)

Mer om Objekt som Parameter

```
Dog d = new Dog();
incAge(d);

void incAge( Dog dog ){
   dog.age++;
}

class Dog {
   String name;
   int age;    // 0 default
}
```



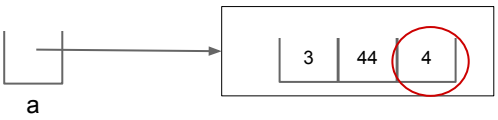
Eftersom referensen till objektet skickas till metoden kommer den åt objektet (utanför metoden)

- På samma sätt som för en array

Gäller även strängar men dessa kan inte förändras (icke muterbara) så ingen risk.

Konstanta referenser

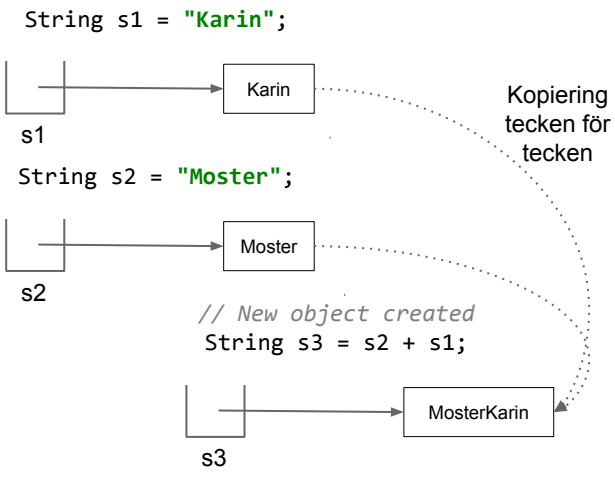
```
// Constant reference variable a
final int[] a = { 3, 44, 2 };

// 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!

Strängar och +-operatorn



14

Konkatenering med +-operatorn innebär att ett nytt strängobjekt skapas och en referens till detta returneras.

- Eftersom strängar inte kan ändras (tecknen kan inte ändras)
- Tecknen från operanderna kopieras till det nya objektet.
- +-operatorn kan vara ineffektiv t.ex. i en loop med många varv (kopierar samma sak och mer och mer för varje varv)
-

Kedjade metodanrop

```
Complex c = new Complex(1, 2);
// Chained calls (invisible objects)
c.add(c).add(c).equals(new Complex(3, 6));


// Returns an object
Complex add(Complex other) {
   return new Complex(this.re + other.re,
                      this.img + other.img);
}
```

15

Om metoden returnerar en referens till ett objekt kan vi direkt anropa en metod på objektet (utan att spara referensen i en variabel).

- Sker detta i flera steg
  - Om så finns det ett antal "osynliga" mellanliggande objekt.
- Kan ge smidig kod i vissa fall.

En annan variant för att möjliggöra kedjade anrop är att returnera this

- Inga mellanliggande objekt kommer då att skapas
- Ingen bra idé för koden i bilden, där vill vi ha nya objekt, eftersom resultatobjektet skall vara helt skilt från operanderna.

Metodreferenser

```
// References to method with int param end return type
Function<Integer, Integer> addRef = this::add;

// Reference to void method with int param
Consumer<Integer> doItRef = this::doIt;

// Sadly can't do like this … must use methods as below
// addRef(6);
// doItRef(6);

out.println(addRef.apply(6));  // Execute referenced method
doItRef.accept(6);             // Execute referenced method

void doIt(int i) {
   out.println(i);
}

int add(int i) {
   return i + 1;
}
```

16

Man kan ha referenser till metoder i Java

- Vi går inte in på detaljerna här (man kan spara referenser i variabler d.v.s. det finns typer för olika sorters metoder m.m.)
- En metodreferens skrivs: objektet :: metoden (dubbla kolon)
    - Objektet vi använder är this.

Användning

- För intern iteration av samlingar (forEach)
    - Man anger vilken metod som skall anropas för varje element i samlingen (elementet skickas som argument till metoden, d.v.s. metoden måste ha en parameter av elementtypen)
- I JavaFX då vi skapar händelsestyrda program
- Finns i sista labben (given kod).