Bilder Vecka 2

TDA548/Joachim von Hacht

Grunderna

(som sagt, repetera serien Grunderna)

Styrande Satser

Namnge Booleska Uttryck

```
// Hmm, what does it mean? Bad
if( score1 >= 10 || score2 >= 10 ){
    if( score1 != score2) {
        …
    }
}
// Also bad
if( score1 >= 10 || score2 >= 10 && score1 != score2) {
...
}
// Better
boolean gameOver = score1 >= 10 || score2 >= 10 && score1 !=
score2;
if( gameOver ) {  // Much clearer!
...
}
```


Om de booleska uttrycken blir för komplexa, … skapa en boolesk variabel med ett beskrivande namn

- Mycket lättare att förstå

Nästlade for-satser

```
for (int i = 0; i < arr.length - 1; i++) {
   for (int j = 0; j < arr.length - i - 1; j++) {
       if (arr[j] > arr[j + 1]) {
           int temp = arr[j];
           arr[j] = arr[j + 1];
           arr[j + 1] = temp;
       }
   }
}
```

5

Nästlade for-satser mycket vanligt.

- Ofta i samband med arrayer, se Arrayer
- Ibland styrs den inre loop-variabeln av den yttre (den inre beror på den yttre).

För att förstå nästlade for-loopar är det vanligen enklast att börja “inifrån”, försök förstå vad som görs i den inre loopen först!

- Gäller även nästlade while

Metoder

Synlighetsområde

```
{
 int i = 0;

}

//Ok, other scope
{
 int i = 2;

}
```

```
{

 int i = 0;

    {
        int i = 2; // Name!
        int j = 3;
    }


    j = 4; // Not visible
}
```

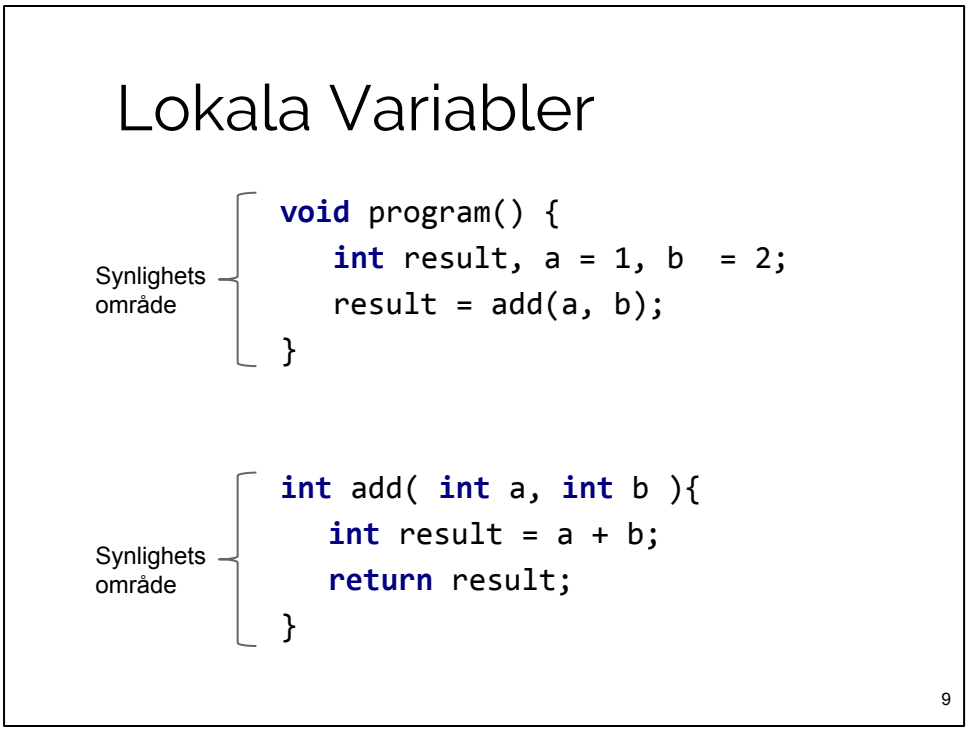
7

Synlighetsområdet (scope) anger var i programmet det är möjligt att använda en variabel

- I Java sammanfaller synlighetsområde och block (förenklat)

Följande gäller

- Variabler är bara synliga (kan bara användas) i det synlighetsområde de är deklarerade (fr.o.m. deklarationen och vidare), förutom …
  - … nästlade synlighetsområden
    - Ett inre block kommer åt variabler i ett yttre (som är deklarerade innan det inre blocket)
    - Yttre block kommer inte åt variabler i ett inre
- I Java gäller att variabler inte får ha samma namn inom samma synlighetsområde
- Inom olika synlighetsområden kan samma namn användas
  - Mycket praktiskt: Slipper hitta på nya namn hela tiden!



Variabler deklarerade i metoder kallas **lokala variabler**

- Vi räknar även parametrar som lokala variabler
- Synlighetsområdet är metodkroppen (ett block)
- Lokala variabler har en livslängd, de lever på anropsstacken och förstörs då programmet lämnar metoden.
- Lokala variabler måste ges ett värde (initieras eller tilldelas) innan de används, om ej kompileringsfel
- Inledningsvis tillåter vi bara lokala variabler i våra program! (d.v.s. inga variabeldeklarationer utanför metoder)

Bilden:

- "result" i koden ovan syftar på två olika variabler med samma namn, den ena i program() den andra i add()
- På samma sätt med a och b.
- Eftersom de finns i olika synlighetsområden kan vi använda samma namn!
- I exemplet finns totalt 6 variabler (2 st a, 2 st b och 2 st result)

Anm: I program() ovan deklarerar vi flera variabler på samma rad

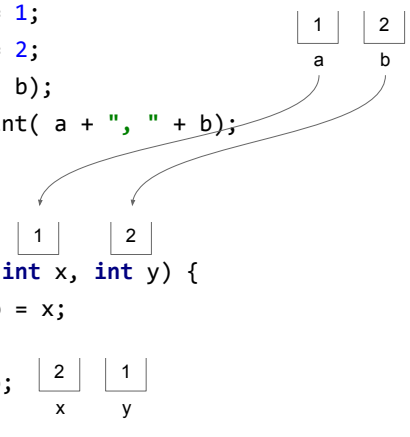
- Möjligt att göra men inte så bra, bättre med en/rad (görs av utrymmesskäl här)

Summa:

- Samma variabelnamn inom olika synlighetsområden syftar på på olika variabler
- Olika variabelnamn (ev inom olika synlighetsområden) kan syfta på samma sak, om referenser är inblandade, se Referenser

Variabler utanför Metod

```
void program() {
   int a = 1;
   int b = 2;
   swap(a, b);
   out.print( a + ", " + b);
}

void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

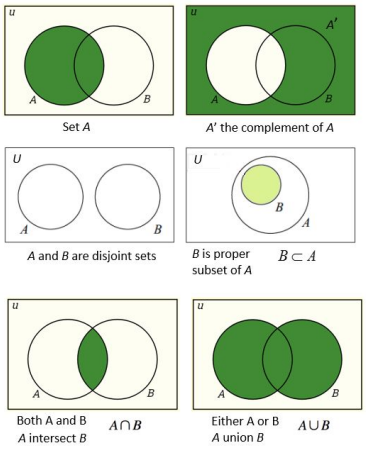


En metod kan aldrig komma åt lokala variabler i en annan metod.

- Antag att vi vill skriva en metod, swap, som byter plats på argumenten!
- När vi anropar metoden swap kopieras värdena i a och b till parametrarna (lokala variabler), därefter byter dessa värden
- … inget händer med a och b i program(),… swap kan aldrig komma åt a och b.
- Det går inte att skriva metoden swap med primitiva typer
  - Däremot kan metoder komma åt objekt “utanför” … om parametrarna är referenser.

Typer

Vad är en Typ?

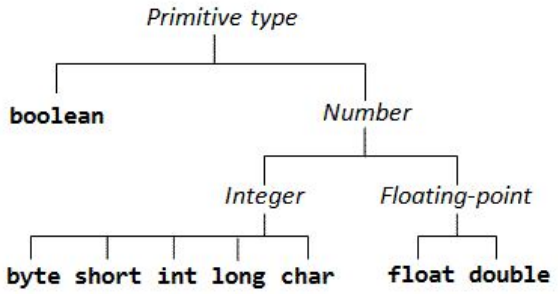


11

Finns inget enkelt svar.

- I denna kurs ser vi en typ som en mängd
- T.ex. mängden av reella tal, mängden av sanningsvärden o.s.v.

Primitiva Typer



12

Bilden visar de 8 primitiva typerna i Java. Dessa finns färdiga i språket och kan inte ändras (eller lägga till nya).

- boolean, är typen för sanningsvärden
  - Finns bara två värden i typen: true och false
- int, är typen för heltal (integer)
  - Heltalsliteraler ges automatisk denna typ
- char, är typen för enstaka tecken (character). Egentligen en teckenkod, därför räknas den som ett heltal.
  - Alla teckenliteraler ges denna typ
- double, är typen för reella närmevärden, kallas också flyttal (double precision, ca 15 decimaler)
  - Flyttalsliteraler får denna typ
- long, används för stora heltal
- byte, short, och float hoppar vi över så länge (behövs inte i kursen)

Namn på primitiva typer är reserverade ord

Referenser

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 ... !!!
```

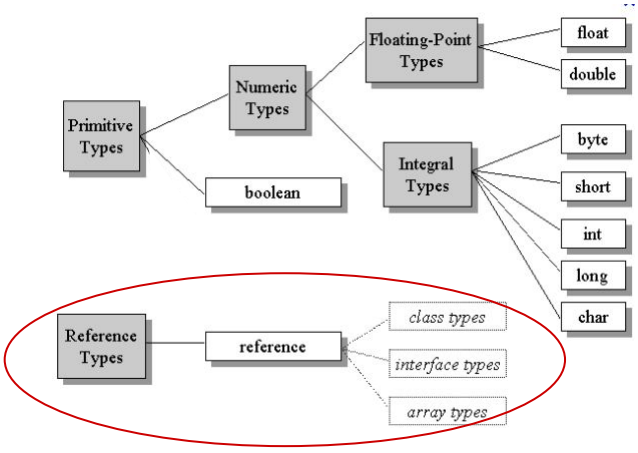
Vi har sett att vid tilldelning och metodanrop/återhopp sker det en kopiering av data

- 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



15

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

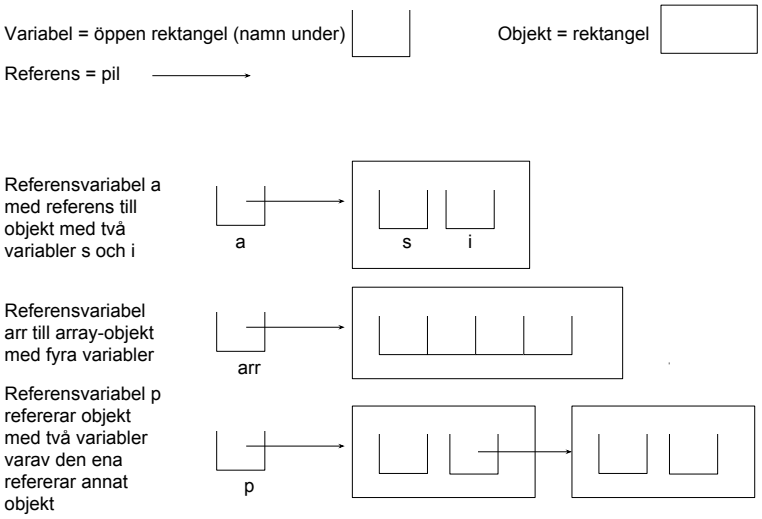
Man kan skapa nya referenstyper!

- 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

Visualisera Referenser

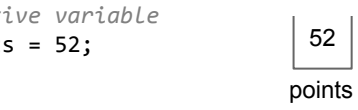


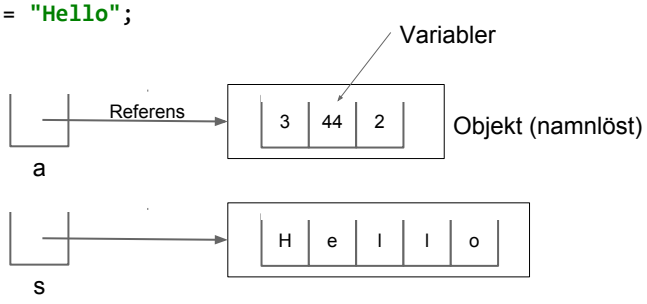
För att visualisera referenser, referenssemantik (betydelsen, se vidare nedan) och objekt/variabler använder vi ett "bildspråk".

- Variabler ritar vi som tidigare.

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)**

Avreferering

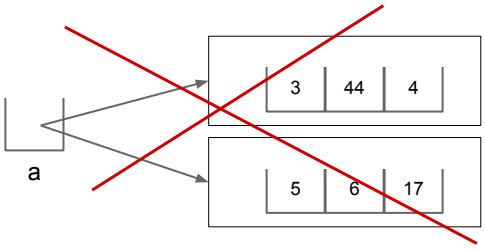
```
class Player {
  String name;
  int pts;
}

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.

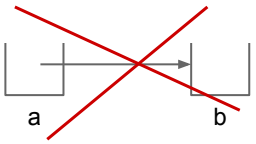
- Så sker alltid i Java!
- [ ]-operatorn och **.**-notationen gäller alltså för objektet, inte för referensen (eller variabeln)!

Saker som aldrig kan hända!

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

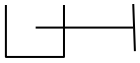


Detta kommer aldrig att ske i Java!
En referens pekar alltid på ett objekt, aldrig på en annan variabel.



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

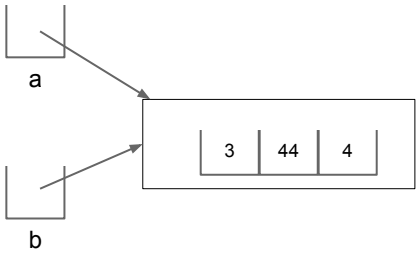
För alla objekt, o, skall gälla att o.equals(null) är false!

- Se bildserie klasser.

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

Referenser och Tilldelning

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

int[] b;

b = a;    // Oops!
```

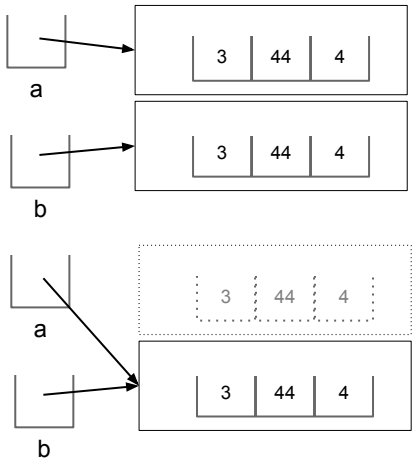


Tilldelning för referensvariabler fungerar som för primitiva typer

- 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**

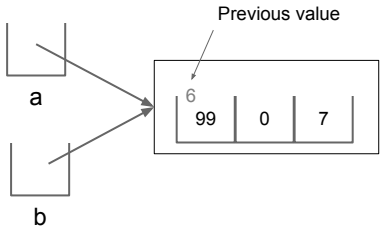
- Semantiken (betydelsen) blir annorlunda pga av vi använder referenser
- För primitiva typer säger man **värdesematik**

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
}
```



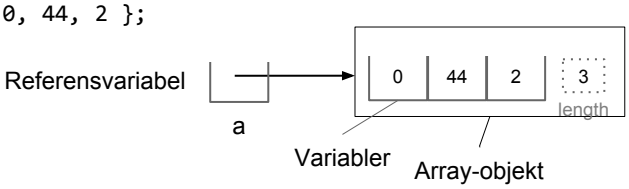
Att flera referenser kan peka på samma objekt innebär att det finns flera sätt att ändra variablerna i objektet.

- 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)

Arrayer

Array-Objekt

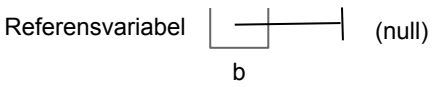
```
// Declaration and initialization, array-object created
int[] a = { 0, 44, 2 };
```



Variabler

Array-objekt

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



28

En variabeldeklaration ger en variabel!

- Men m.h.a. en array-deklaration kan vi plötsligt skapa flera variabler?
- Hur kommer det sig?

Förklaring:

- En deklaration av en array ger en (enda) referensvariabel
- Värdet i variabeln är en referens till ett (namnlöst) **array-objekt** som i sin tur innehåller ett antal (namnlösa) variabler
- Vi kan bara nå objektet indirekt, via referensen och variablerna via indexering.
    - Indexering innebär: Avreferera referensen (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.**
- 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.

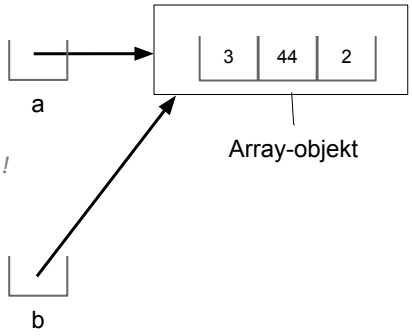
Att det skapas ett objekt i samband med deklaration och initieringen kallas

att objektet instansieras (objektet är en instans av typen array)

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

Tilldelning och Likhet

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




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



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

30

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!

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)

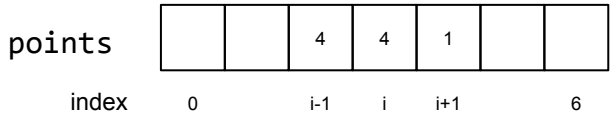
Testning av Arrayer

```
int[] arr = { … };


// Simple way to test expected values of array
// Convert to string and compare strings
out.println(Arrays.toString(arr)).equals("[1, 2, 3]"));
```


För att förenkla vid testning kan man omvandla arrayen till en sträng, på samma sätt som vid utskrifter (så slipper vi loopar)

- Nackdel: Det gäller att skriva rätt i det förväntade resultatet!

Array som Datastruktur




```
int[] points = {0, 0, 0, 0, 0, 0, 0};

int i = 3;
points[i] = 4;            // { 0,0,0,4,0,0,0 }
points[i+1] = 1;          // { 0,0,0,4,1,0,0 }
points[i-1] = points[i]; // { 0,0,4,4,1,0,0 }
```

28

En array har en struktur, det finns första/sista, före/efter, vänster/höger

- Vi säger att en array är en **datastruktur**.
- Ger oss möjlighet indirekta komma åt variabler: “grannen till”, “tre efter”, “ en före”, …
- Givet ett index i kommer vi åt
  - variabeln till vänster (före) med i-1
  - variabeln till höger (efter) med i+1
  - i-1 eller i+1 får inte hamna utanför array:en, om så: Undantag (som tidigare).

Matriser

```
// Declare and initialize (instantiated automatically)
int[][] m = {       // An array of arrays
       { 1,2,3,},          // m[0]
       { 4,5,6,},          // m[1]
       { 7,8,9,},          // m[2]
   };

int[][] m;          // Just a single variable
m = new int[3][3];  // Instantiate object, default values 0
m = new int[2][2]{  // Instantiation and initialization
    {11, 12},
    {21, 22}
};
```

29

I Java kan man ha arrayer av godtycklig dimensioner (< 255)

Tvådimensionella array:er är vanligt

- Tekniskt en array av array:er
- Vi kallar 2D arrayer för matriser
- En matrisvariabel deklareras med dubbla [ ] -parenteser efter typen
  - Först index anger vilken array (rad), …
  - … andra anger, element i aktuell array (kolumn)
  - Rad och kolumnindex börjar som vanligt på 0.
- Vi använder normalt bara rektangulära matriser i laborationerna (man kan använda “ragged 2D arrays”)

Indexering på Matriser

```
int[][] m = { { 1,2,3,}, { 4,5,6,}, { 7,8,9,} };

// [row][col]
m[0][2] = 99;  // {{ 1,2,99,},{ 4,5,6,},{ 7,8,9,}}
m[2][1] = m[0][2]; // {{ 1,2,99,},{ 4,5,6,},{ 7,99,9,}}

// Exception
m[0][3] = 45;
```


För att komma åt enskilda element används som tidigare indexering (men nu med två index)

- Alltid rad, kolumn.

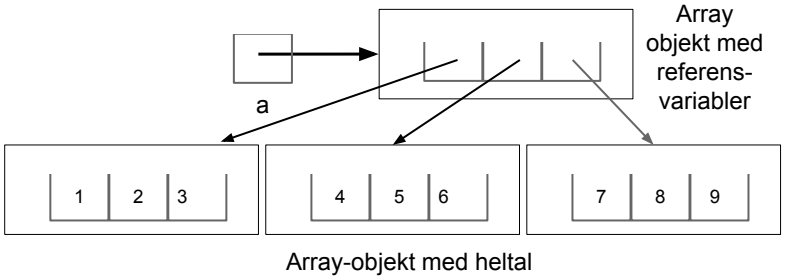
Traversera Matris

```
int[][] m = { { 1,2,3,}, { 4,5,6,}, { 7,8,9,} };

// Traverse (will work for ragged matrices)
for(int row = 0 ; row < m.length ; row++){
   for( int col = 0 ; col < m[row].length; col++){
       out.print( m[row][col]);
   }
   out.println();
}
```


Traversering använder nästlade for-loopar och length.

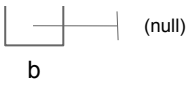
Matris-Objekt

```
int[][] a = { { 1,2,3,}, { 4,5,6,}, { 7,8,9,} };
```



Array-objekt med heltal

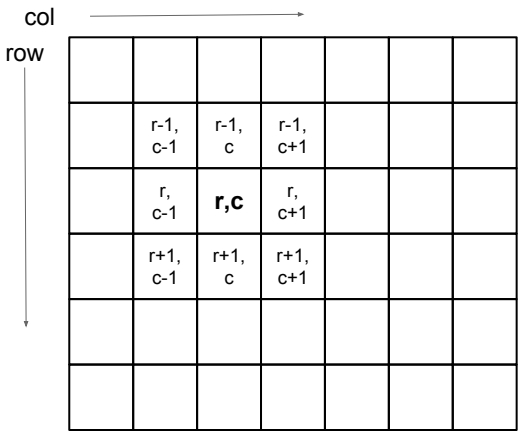
```
// NO matrix object
int[][] b = null;
```



En deklaration av en matrisvariabel ger, på samma sätt som en array, en referensvariabel

- Objektet referensen pekar på innehåller i sin tur variabler av referenstyp
- Som tidigare anger vi null om variabeln inte skall peka på något objekt.

Matris som Datastruktur



33

En matris ger oss en datastruktur

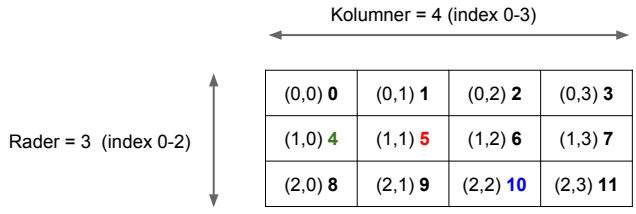
- Fler möjligheter: vänster/höger/över/under/snett över/snett under ...

Testning av Matriser

```
int[][] m = {
        {0, 1, 0,},
        {1, 1, 1,},
        {0, 1, 0,},
};

out.println(Arrays.toString(m[0]).equals("[0, 1, 0]"));
// m[1] false!
out.println(Arrays.toString(m[1]).equals("[1, 0, 1]"));
out.println(Arrays.toString(m[2]).equals("[0, 1, 0]"));
```


Ungefär som för arrayer fast vi får skriva ut alla rader.

Byte mellan Array och Matris



Array -> Matris:

radindex = index / kolumner (5 / 4 = 1)

kolumnindex = index % kolumner (10 % 4 = 2)

Matris -> Array:

index = radindex * kolumner + kolumnindex (1 * 4 + 0 = 4)

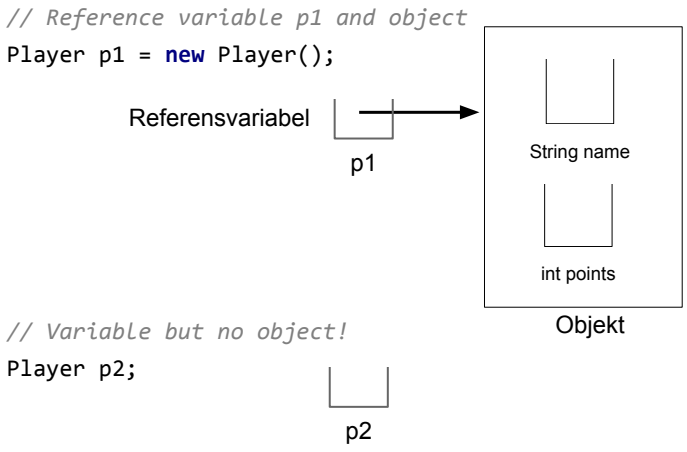
35

Omvandling mellan array och en matris kan behövas

- Datan är oförändrad det är bara strukturen som ändras.
- Ibland lättare att arbeta med array-format …
- … ibland lättare med matris.

Klasser

Mer om Variabler för Objekt



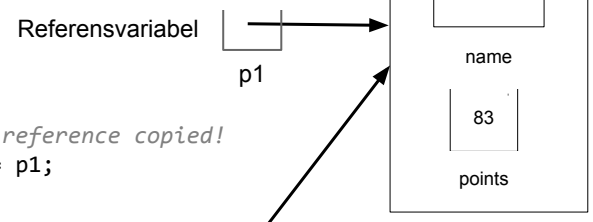
37

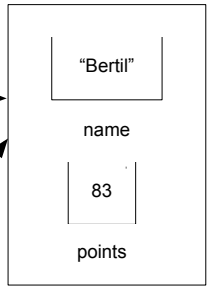
Detta fungerar på samma sätt som för arrayer!

- Variabler för objekt är referensvariabler
  - En deklaration ger en referensvariabel, instansieringen (instansieringsuttrycket) ger en referens till ett namnlöst objekt
    - Värdet är alltså en referens …
    - .. sidoeffekten är att ett (namnlöst) objekt skapas (i minnet).
- En deklaration av en variabel ger bara en referensvariabel, inget objekt!
- Punktnotation innebär att man avrefererar referensen och därefter väljer variabel utifrån namn m.h.a. punktnotation.
  - Punkten använd på objektet (inte på variabel eller referens)!

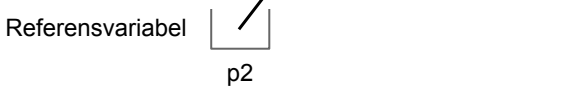
Tilldelning

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






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

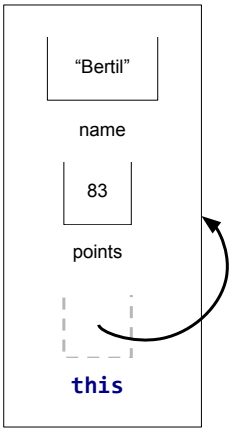



```
out.println(p1 == p2);   // true
```


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

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 = ...
```

39

Alla objekt i Java har en dold konstant referens 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
- Den statiska typen för this är typen som klassen introducerar (Player i bilden).
  - Den dynamiska typen kan vara någon subtyp, se Typer.
- this är ett reserverat ord

Konstruktor

```
class Player {
   String name;
   int points;
}
Player p = new Player();
// Tiresome to set values
p.name = "pelle";
p.points = 2;
```

```
// Better, use constructor
class Player {
   String name;
   int points;
   // Constructor
   Player(String name, int points){
      this.name = name;
      this.points = points;
   }
}

// Using constructor with arguments
Player p = new Player("pelle", 2);
```

40

För att initiera ett objekt på ett visst sätt kan vi skapa vi en speciell metod i klassdeklarationen, en **konstruktor**

- M.h.a. konstruktorn sätter vi värden för instansvariabler

En konstruktor

- Är en metod som automatiskt anropas i samband med instansiering
  - Måste ha samma namn som klassen.
  - Parameterlista som vanlig metod
  - Får inte ange returtyp.
  - Kan inte anropas som en vanlig metod
- Ofta är det naturligt att parametrarna till konstruktorn har samma namn som instansvariablerna de skall tilldelas (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**.

Det finns alltid en parameterlös konstruktor (den vi använt tidigare).

- Om man skapar en konstruktor med parametrar försvinner den parameterlösa konstruktorn.
- Vill man ha en sådan får man skriva dit den själv.

Mer om Typer

Uppräkningstyper

```
enum WeekDay {   // Enumeration type
   MON, TUE, WED, THU, FRI, SAT, SUN
}

WeekDay d1 = WeekDay.FRI;
WeekDay d2 = WeekDay.THU;
WeekDay d3 = WeekDay.FRI;

out.println(d1 != d2);   // == Works!
out.println(d1 == d3);  // Same object

// Loop through all days
for (int i = 0; i <  WeekDay.values(); i++) {
   // Do something
}
```

46

Ibland behövs ett begränsat antal värden av en viss typ (typen innehåller ett litet antal värden)

- T.ex. veckodagar, färger , etc
- VI skulle kunna använda String för dessa t.ex. “Mon”, “Tue” osv. men … (eller integer med Månd = 1, o.s.v.)
  - … detta blir inte typsäkert …
  - t.ex. om vi stavar fel, t.ex. “Tui”, så släpper kompilatorn igenom felet (felet kommer senare under körning)
  - Om vi använder int för dagar så kan någon av misstag tilldela en dag värdet -12, o.s.v. …
- Bättre att låta typsystemet se till att allt stämmer

Genom att deklarera en **uppräkningstyp** (enumeration**)** skapar vi en ny (egen) klasstyp med ett antal (uppräknade) värden (en enum är en slags klass).

- Vi kan därmed deklarera variabler av denna typ.
- Kompilatorn kan kontrollera att vi bara använder korrekta värden !

Deklarationen av uppräkningstyp görs men det reserverade order **enum**.

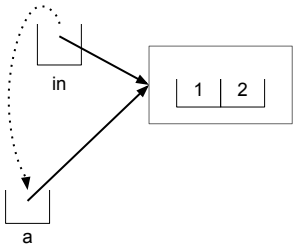
- De värden som tillhör typen räknas upp (vi skriver namnen på värdena)
  - I bilden: MON, TUE, … o. s.v. (stora bokstäver skall användas)

    - Värdena är av typen WeekDay (INTE String, inga citattecken runt)
    - Det finns exakt ett (icke-ändringsbart) objekt för varje namn vi räknar upp.
        - Likhet (==) fungerar därför mellan värden (eftersom det är identitet i detta fall)
- För att komma åt värdena måste vi använda **punktnotation** d.v.s. typnamn.värde
    - Går att förenkla genom att använda import static …. Weekday.* (som med System)

Mer om Metoder

Mer om Array-parametrar

```
void program() {
   int[] in = { 1,2 };
   zero(in);
    // in is now [ 0, 0 ]
}


void zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
       a[i] = 0;
    }
}
```



Om en metod har en array-parametrar sker precis samma sak som vanligt men …

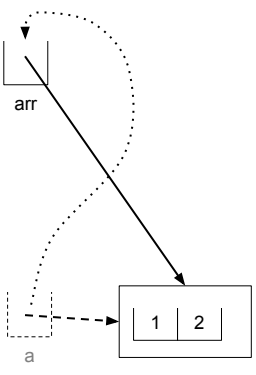
- … argumentet som kopieras är en referens!
- Parametern kommer därför att peka på samma objekt som argumentet!
- Metoden kan ändra på ett objekt utanför sig självt (variabeln “in” kan vi däremot aldrig ändra)

I koden ovan kommer array:en som variabeln in refererar att vara förändrad efter metodanropet

- Kallas **utparameter**
- Innebär en viss risk eftersom det kan vara svårt att inse att en metod har ändrat i objektet (eftersom den är void)

Mer om Array som Returtyp (1)

```
void program() {
   int[] arr = getArray();

}


int[] getArray(){
    int[] a = { 1, 2 };
    return a;
}  // a gone here
```



Om man returnerar en array sker samma sak som vid ett vanligt metodanrop

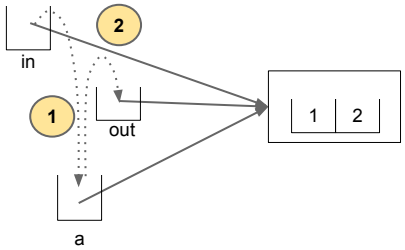
- Dock är <u>returvärdet en referens</u>

Vi kan skapa en array i en metod och skicka som returvärde.

- Den lokala variabeln a försvinner! …
- … men inte array-objektet
- För att kunna komma åt array-objektet måste vi spara den returnerade referensen

Mer om Array som Returtyp (2)

```
void program() {
   int[] in = { 1,2 };
   int[] out = zero(in);

}

int[] zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
       a[i] = 0;
    }
    return a;  // Return parameter!
}
```

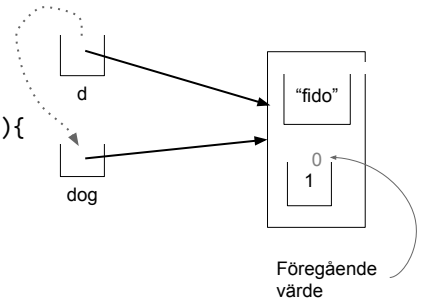


Här returnerar vi samma referens som vi skickar in.

- Kan i vissa fall ge smidigare kod.

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
}
```



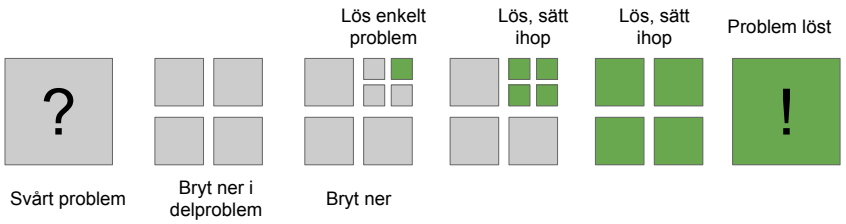
Eftersom referensen till objektet kopiera till metodens parameter kommer den åt objektet (utanför metoden)

- På samma sätt som för en array
- Samma problem som för en array

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

Arbetssätt

Bryta ner Problem



49

Normalt är ett problem (program) för stort för att direkt kunna kodas med minsta steget metodiken.

- Vi måste dela upp problemet.

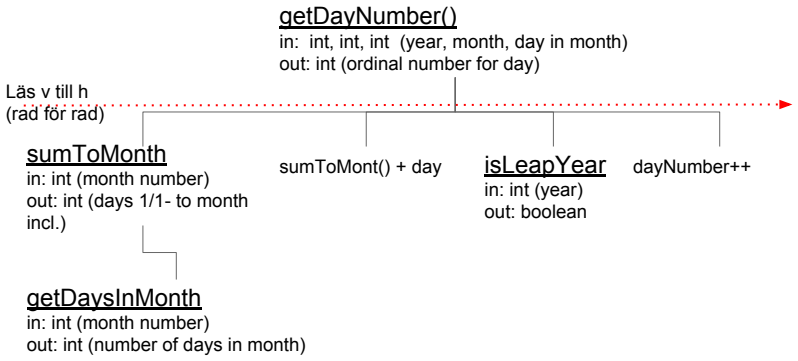
Ett klassiskt angreppssätt om man har ett stort/svårt problem.

- Bryt ner i mindre/enklare problem!
- Lös de enklare problemen (eller dela dessa ytterligare)
- Sätt ihop alla lösningar av delproblemen till en lösning för hela problemet.

Konkret gör vi detta genom att använda funktionell nedbrytning.

Funktionell Nedbrytning

Problem: Givet årtal och datum, beräkna ordningsnumret för dagen detta år (Exempel: 25/4, 2018 är dag 115 detta år)



50

Funktionell nedbrytning (functional decomposition) innebär att man:

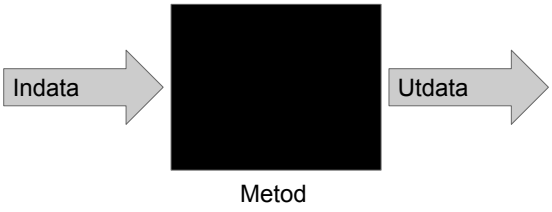
- Antar att man har en metod som löser hela problemet (i bilden: getDayNumber)
- Börjar skriva denna … när man stöter på ett nytt problem antar man att man har en metod som löser detta (sumToMonth, isLeapYear).
- O.s.v… tills metoderna man behöver är triviala t.ex. getDaysInMonth.
  - Implementera de triviala metoderna.
  - Implementera m.h.a. dessa metoderna "högre upp"
  - Genom att kombinera små enkla metoder har man löst ett stort problem. Klart!
- För att illustrera kan man rita ett diagram enligt bilden. Överst ritar man den första metoden man antog, därefter en ny nivå för varje nedbruten method

Funktionell nedbrytning är en top-down strategi. Man börjar med helheten och bryter ner i enklare delar.

Genom att använda funktionell nedbrytning (plus anteckningar/skisser) får vi en struktur på programmet.

- Ha alltid papper och penna uppe!

Funktionell Abstraktion



56

Under arbetet med funktionell nedbrytning söker vi metoder vi behöver.

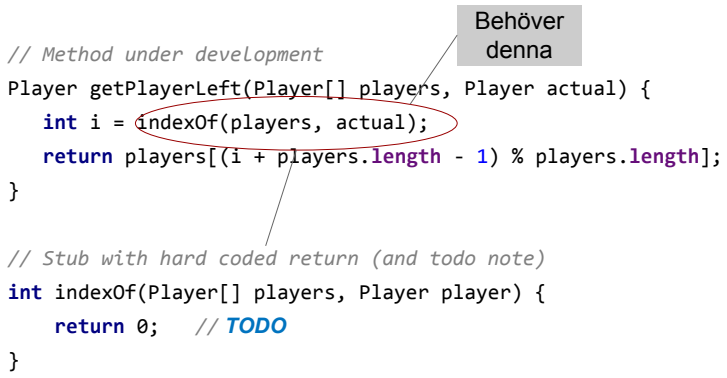
För att underlätta sökandet använder man **funktionell abstraktion**, en tankeform där man bara fokuserar på:

- Indata (vad har jag?)
- Utdata (vad vill jag ha?)
- … detta för att slippa alla detaljer om hur det skall gå till …
- Sikta på vad som skall göras inte hur!
    - Vi måste veta vad som skall göras innan vi börjar fundera på hur det skall göras.

Process för att skriva en metod:

1. Namnge metoden (med ett bra namn som säger vad den gör, ett verb är ofta inblandat).
    a. En metod skall vara bra på en sak, kan du inte hitta ett bra namn kan det bero på att metoden gör för mycket. Isf bryt ner den i mindre metoder.
2. Vilken data har du? Ange parametrar.
3. Vad vill du ha? Skriv metodens returtyp m.h.a. detta.
4. Skriv klart metoden m.h.a. minsta steget (och testa, se senare slide)

Metodstubbar



Då man arbetar med funktionell nedbrytning händer det t.ex. att man upptäcker flera metoder samtidigt.

- Eftersom vi bara kan implementera en metod i taget så skriver man "stubbar" för de övriga

En **metodstubbe** är en "tom" metod.

- Genom att använda stubbar kan vi gör anrop till metoder som inte är färdiga d.v.s programmet kompilerar och går att köra (men resultatet blir inte rätt)
- Genom att använda stubbar kan vi få ett körbart program.
  - Vid behov hårdkoda returdata, t.ex. return 0, return null, return "", o.s.v..

Testning

```
// Simple test for dices
int d = rollDice();
out.println(1 <= d && d <= 6);   // true?

// Test to get player to the left of first (there are 3 players)
Player[] players = {new Player("0lle", 3), new Player ...};
out.println(getPlayerLeft(players, players[0]) == players[2]);  // true?


// Check distribution of game chips given a known distribution and a result
String[] res = {"L", "C", "R"};  // The known result
actual = players[1];
distributeChips(res, players, actual); // Initially all have 3 chips
out.println(players[0].chips == 4 &&
            players[1].chips == 0 && players[2].chips == 4); // true?
```

58

Då vi använder funktionell nedbrytning kommer programmet att bestå av ett antal samverkande metoder.

- Att bara sätta ihop dessa utan att veta om de fungerar är dåligt!
  - För att få helheten att fungera måste delarna fungera!

Vår strategi är följande:

- Så fort vi implementerat en metod så testar vi den.
- IO-metoder testar vi, vid behov manuellt, vi provkör helt enkelt
  - IO metoder är (skall vara) enkla.
- För logikmetoder automatiserar vi testerna genom att skriva ut (out.println) ett uttryck som skall ge värdet true
  - I uttrycket gör vi en jämförelse mellan resultatet av ett metodanrop och ett förväntat korrekt värde (som vi måste veta)
    - Här krävs viss kreativitet för att komma på vilka jämförelser vi vill göra
    - Testerna måste vara så enkla som möjligt, vi vill inte introducera nya fel i själva testerna.
  - Vi skriver alla tester i en egen metod med namnet test() eller …
    - … senare i en egen klass men namnet Test.
  - Vi behåller alltid alla tester, … när som helst kan vi köra dessa igen!

I senare kurser får du lära dig att använda speciella ""testramverk".