Mer om Arrayer, Matriser, Enum och Omslagtyper

Vecka 3, Bildserie 1

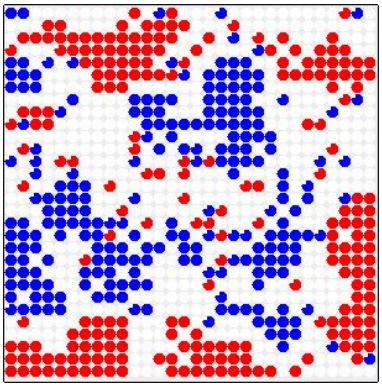
Innehåll

- Blandade inläsningar
- Array-objekt
- Matriser (2D array:er)
- Instansvariabler och final
- Permutering av Arrayer
- Enum
- Generiska metoder
- Omslagstyper

Mål

Kunna lösa problem som kräver mer avancerade algoritmer
- Problemen innehåller matriser (2d array:er)
- Ökad förståelse för typsystemet

Veckans Produkt



Vi skall göra ett program som simulerar segregering.
Se vidare :
http://nifty.stanford.edu/2014/mccown-schelling-model-segregation/

Kortfattat

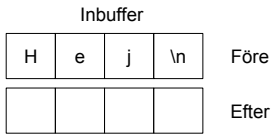
- Röda prickar trivs om tillräckligt många röda grannar …
- … samma med blå
- Om en prick är missnöjd flytter den slumpmässigt till en annan ledig position
- När (om) alla är nöjda stannar simuleringen.

Att Läsa i Boken

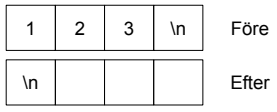
- 2.7
- 8.1-8.2 (ej 8.2.3)
- 8.3-8.4
- 10.7-10.8
- enum och generiska metoder verkar inte finnas i boken
  - Det räcker med det som finns i presentationen och i kodexemplen.

Blandade Inläsningar

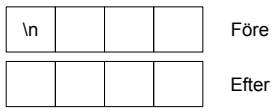
```
String line = sc.nextLine();
```




```
int i = sc.nextInt();
```




```
// Will read \n
String line = sc.nextLine();
```



För att läsa strängar används en Scanner (som tidigare)

- scan.nextLine() läser hela inbufferten inkl. det avslutande '\n'-tecknet som genereras av Enter-tangenten

Blandade inläsning av strängar och tal kan ge problem!

- nextInt() tar bara med tecken som kan ingå i en int '(\n' kan inte ingå)
  - Resten ligger kvar i inbufferten
- Om vi anropar nextLine() då något finns i inbufferten kommer det kvarvarande att läsas
  - Inläsningen är klar , … programmet stannar inte och väntar på tangentbordsinmatningen!
  - Vi uppfattar det som att programmet hoppar över en inläsning
  - Lösning: Lägg in extra sc.nextLine() för att bli av med "\n"

Array-objekt

Deklaration och initiering

```
int[] points = {0, 0, 0};  // Implicit instantiation
```

Instansiering

```
final int MAX = 10;
int[] points = new int[MAX];  // Explicit instantiation

int a = …
int[] values = new int[a];   // Use variable for size
int[] values = new int[arr.length]; // Use other array
```

I Java är array:er objekt

- Vid deklaration och initiering skapas en variabel som refererar ett objekt (med att antal variabler i rad)
  - points är en referensvariabel (innerhåller en “pil” till objektet)
- Vi kan även göra en explicit instansiering med hjälp av operatorn **new**
  - I instansieringsuttrycket anger man typ (t.ex. int-array) samt storlek inom hakparentesen
    - För int-arrayer gäller att alla variabler initieras till 0;
    - double blir 0.0, boolean blir false och referenser blir null.
  - Storleken kan vara ett fixt värde eller en variabel (av heltalstyp >= 0, annars körningsfel)
    - Praktiskt då vi på förhand inte vet storleken MEN …
  - Som tidigare, efter det att vi skapat array:en kan vi inte ändra storleken
- Anger man bara int[] arr; har vi BARA en variabel, den som kan innehålla referensen!

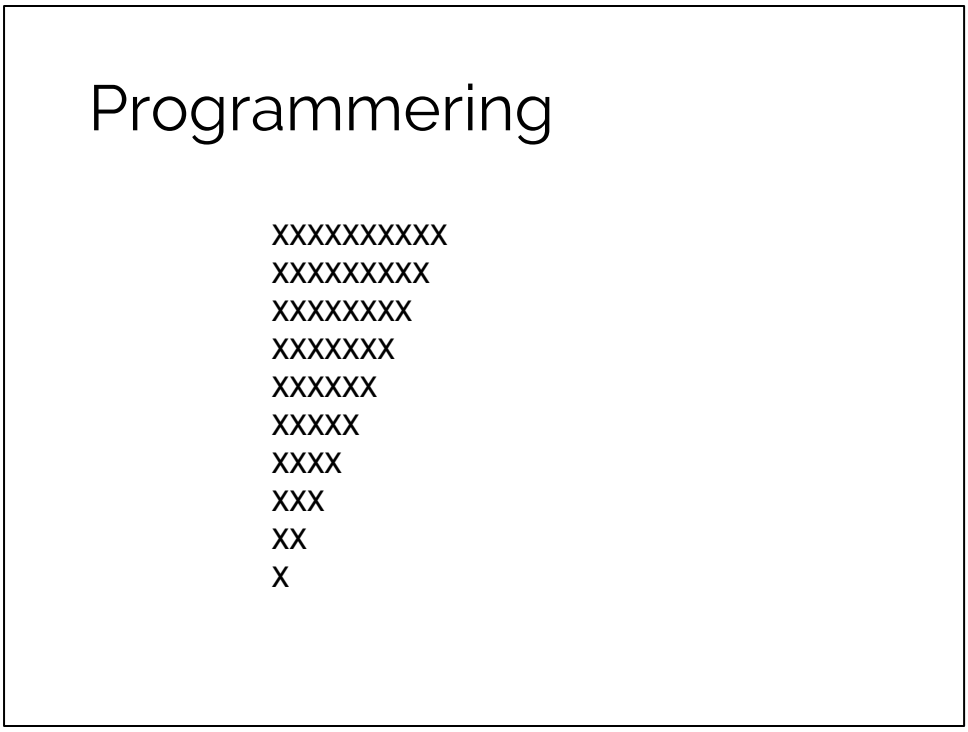
Programmering

```
int[] i1 = { 5, 4, 2, 1, 7 };
int[] i2 = { 8, 3, 8 };

int[] r = merge(i1, i2);

// Result [ 1, 2, 3, 4, 5, 7, 8, 8 ]
out.println(Arrays.toString(r));
```

Skriv metoden merge!

- Givet två arrayer, slå ihop dessa. Returnera som sorterad array.
- Använd explicit instansiering av arrayer (för att skapa en ny array för resultatet)
- TIPS: Använd sortering som ett försteg till själva sammanslagningen.

Programmering



Hur skriver vi programmet?

- Programmet ritar en halv kvadrat m.h.a. tecknet X.
- Enbart for-loop:ar, out.print(“ “), out.print(“X”) och out.println() får användas.

Matriser (2D Array:er)

Deklaration och initiering (implicit)

```
int[][] m = {       // An array of arrays
       { 1,2,3,},          // m[0]
       { 4,5,6,},          // m[1]
       { 7,8,9,},          // m[2]
   };
```

Deklaration och instansiering (explicit)

```
int[][] m = new int[3][3];
```

I Java kan man ha array:er av godtycklig dimensioner (< 255)

Tvådimensionella (2D) array:er vanligt

- Tekniskt en array av array:er
- Kan instansieras implicit eller explicit
- Deklareras med dubbla [ ] -parenteser efter typen
  - Först index anger vilken array (rad), …
  - … andra anger, element i aktuell array (kolumn)
- Vi använder bara rektangulära matriser (inget krav, "ragged 2D arrays" tillåtet)

Om man betraktar en 2D-array som en matris

- Anger första index rad och andra index kolumn

Indexering

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

[rad][kolumn]

```
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,}}
```

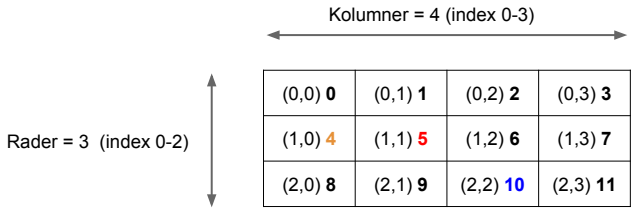
Traversering

```
for(int row = 0 ; row < m.length ; row++){
   for( int col = 0 ; col < m[row].length; col++){
       out.print( m[row][col]);
   }
   out.println();
}
```

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

- Alltid rad, kolumn
- Traversering sker med två (nästlade) for-loopar och length.

Index/Rad/Kolumn



rad = index / kolumner (5 / 4 = 1)
kolumn = index % kolumner (10 % 4 = 2)

index = rad * kolumner + kolumn (1 * 4 + 0 = 4)

Omvandling mellan index i en array och rad/kolumn i en matris kan behövas

- Ibland lättare att arbeta med array-format …
- … ibland lättare med matris.

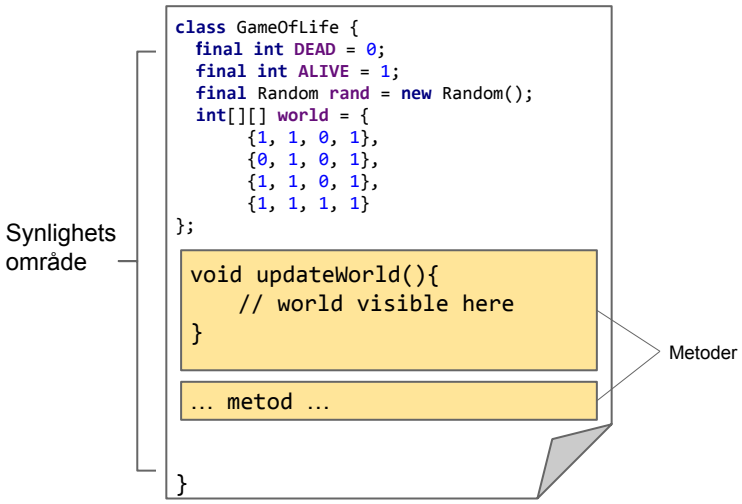
Programmering

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

int[][] m = toMatrix(arr, 3, 3);

/*
   123
   456
   789
*/
printMatrix(m);
```

Skriv en metod som givet en array samt antal rader och kolumner skapar en matris av arrayens element

- Vi antar som vanligt att användare anger korrekt antal rader och kolumner
- Original-arrayen för inte förstöras.

Instansvariabler och final



Variabler deklarerade utanför all metoder kallas **instansvariabler** (instance variables)

- Vi skriver dem vanligen överst i klassen
- Ordningen spelar ingen roll (förenklat).
- Initieras automatiskt (int, double → 0, referensvariabler → null).
- Är åtkomliga i alla metoder, synlighetsområdet är hela "programmet" (mer senare)
- IntelliJ visar instansvariabler i lila fetstil
- … mer senare …

Eftersom alla metoder kan komma åt (och ändra) variablerna innebär det en stor risk…

- .. om något blir fel? Var uppstod felet (i vilken metod?)
- Kan leda till långvarigt felsökande.
- Vi undviker i möjligaste mån instansvariabler

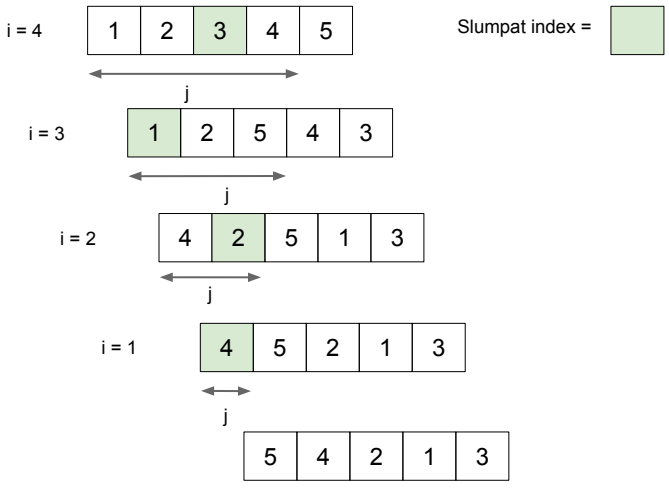
Ett sätt att öka säkerheten är att deklarera variablerna som **final**

- Innebär att de inte kan ändras (och därmed måste de initieras)
- Kallas konstanta variabler.
- Vi kan alltså bara läsa av final variabler (read only)

Vi ersätter vanligen literaler med konstanta variabler

- Vill inte ha literaler utspridda i koden.
- Konstanta variabler underlätter förståelsen eftersom vi ger ett värde ett namn (variabelnamnet)
- Dessutom: Lätt att ändra ett värde överallt genom att ändra på ett ställe (vid deklarationen)

Fisher-Yates Algoritm



Om man behöver en permutation av en array kan man använda Fisher-Yates algoritm

- Här visas Durstenfeld's version av algoritmen.
  - i räknar nedåt från sista index till 1
  - j slumpas mellan 0 och i-1
  - I varje steg byter värdena i och j plats
- Se kodexempel

Uppräkningstyper

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

WeekDay d1 = WeekDay.FRI;
WeekDay d2 = WeekDay.THU;
//WeekDay d3 = "SUN";  // No “SUN” is a string, wrong type
WeekDay d4 = WeekDay.FRI;

out.println(d1 != d2); // == Work!
out.println(d1 == d4);
```

Ibland behövs att 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 int 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) referenstyp med ett antal (uppräknade) värden.

- 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.
  - 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)
- Enum kan användas i switch-satser

Enum och Typomvandlingar

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

// Loop throught all days
WeekDay[] days = WeekDay.values();
for (int i = 0; i < days.length ; i++) {
   out.println(w);    // Will print “MON”, “TUE”, ...
}

// Convert String to WeekDay
WeekDay d = WeekDay.valueOf("SUN");
```

Finns fördefinierat hur man omvandlar mellan enum-typer och String.

- Vid konvertering från sträng måste strängen vara exakt samma som namnet på värdet.

Typproblem

```
// Enumeration types
public enum WeekDay {
   MON, TUE, WED, THU, FRI, SAT, SUN
}
public enum Color {
   RED, BLUE, GREEN, YELLOW, WHITE, BLACK
}
// Fisher Yates
void shuffle(WeekDay[] days) {
   for (int i = days.length - 1; i > 0; i--) {
       int j = rand.nextInt(i);
       WeekDay d = days[i];
       days[i] = days[j];
       days[j] = d;
   }
}
```

Att permutera en array är helt oberoende av typen på elementen

- Vi gör ingen operation på något element, vi bara flyttar runt dem.
- Problemet i bilden är att shuffle bara accepterar WeekDays-arrayer …
- Vill vi permutera en Color-array så säger typsystemet stopp!
- Forts följer …

Generiska Metoder

```
// Overload methods
void shuffle(WeekDay[] days) { … }
void shuffle(Color[] days) { … }
void shuffle( … ) { … }  // Potentially very many …?!?!


// Better use generic method
<T> void shuffle(T[] arr) {   // T stands for some type
   for (int i = arr.length - 1; i > 0; i--) {
       int j = rand.nextInt(i);
       T k = arr[i];
       arr[i] = arr[j];
       arr[j] = k;
   }
}
```

En lösning på typproblemet är att överlagra metoden shuffle

- Tyvärr kan det bli väldigt många överlagrade metoder

Ett bättre sätt är att använda en generisk metod

- En generisk metod är en metod som har en typparameter (T i bilden)
- Metoden kan användas för vilken referenstyp, T, som helst!

Mer Typproblem

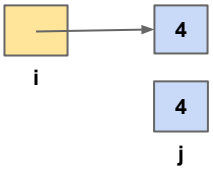
```
// Enumeration (reference) types
public enum WeekDay {
   MON, TUE, WED, THU, FRI, SAT, SUN
}

WeekDay[] days = { ... };
shuffle( days );   // Ok!

int[] arr = { ... };
shuffle( arr );   // No! only reference types allowed


// Fisher Yates
<T> void shuffle(T[] arr) { ... }
```

Generiska metoder fungerar bara för referenstyper!

- Forts följer ...

Omslagstyper

```
// Wrapper type Integer
Integer i = 4;    // Boxing
int j = i;        // Unboxing
```




```
Integer[] is = { 1, 2, 3, 4, 5 };
shuffle(is);      // Ok, reference type
```

Lösningen till föregående typproblem är att tillhandahålla **omslagstyper** (**wrapper types**)

- Omslagstyper är referenstypsversioner av primitiva typer
- De paketerar in ett primitivt värde i en referenstyp, t.ex. int packas in i en Integer-objekt
- Omvandling mellan omslagstyp och primitiv typ sköts automatiskt **(boxing/unboxing)**
- Objekt av omslagtyper kan inte ändras (read only)

Det finns omslagstyper för många primitiva typer, några är …

- Double och Character.