Föreläsning 7

for-satsen
Fält

for-satsen

for-sats är en styrsats för iterationer.

for-sats har följande generella utseende:

```
for (initiering; villkor; ändring)
    sats;
```

for-satsen är ekvivalent med följande **while**-sats:

```
initiering;
while (villor) {
    sats;
    ändring;
}
```

for-satsen är en villkorsloop, men använd **for**-satsen *endast vid räkneloopar*!!!

for-satsen

```
for (int i = 1; i <= 5; i = i + 1)
    System.out.println(i);
```

Ger utskriften:

1
2
3
4
5

```
for (char c = 'a'; c <= 'd'; c = (char) (c + 1))
    System.out.println(c);
```

Ger utskriften:

a
b
c
d

```
for (double x = 0.5; x < 0.8; x = x + 0.1)
    System.out.println(x);
```

Ger utskriften:

0.5
0.6
0.7
0.7999999999999999

Problemexempel

Skriv ett program som visar hur kapitalet på ett bankonto växer då räntan beräknas och läggs till kapitalet en gång per år. Rimlighetskontroll av indata skall göras.

Analys:

Indata: Startår, slutår, startkapital samt ränta.

Utdata: En tabell över kapitalets tillväxer från startår till och med slutår.

Exempel: En programkörning har följande utseende

```
Ge startår och slutår: 2012 2016
Ge startkapital: 1000
Ge räntesats i procent: 2.25
   År          Kapital
 2012         1000.00
 2013         1022.50
 2014         1045.51
 2015         1069.03
 2016         1093.08
```

Problemexempel

Design:

Algoritm:

1. Läs *startår*
2. Läs *slutår*
3. Läs *kapital*
4. Läs *räntesats*
5. Skriv rubriker
6. Upprepa från *startår* till *slutår*
   - 6.1. Skriv årtal och kapital vid årets ingång
   - 6.2. Beräkna årets ränta och lägg denna till kapital

Diskussion: Vid inläsningen måste det gälla att att *kapital* och *räntesats* är större än noll.

Datarepresentation:

startår och *slutår* är av datatypen **int**.
kapital och *räntesats* är av datatypen **double**.

Implementation:

```
import javax.swing.*;
import java.text.*;
public class Capital {
   public static void main(String[] arg)    {
      int startYear = readStarYear();
      int endYear = readEndYear();
      double capital = readCapital();
      double interest = readInterest();
      writeTable(startYear, endYear, capital, interest);
   } // main

   private static int readStarYear() {
      String input = JOptionPane.showInputDialog("Ange startår: ");
      return Integer.parseInt(input);
   }//readStartYear

   private static int readEndYear() {
      String input = JOptionPane.showInputDialog("Ange slutår: ");
      return Integer.parseInt(input);
   }//readEndYear
```

Implementation:

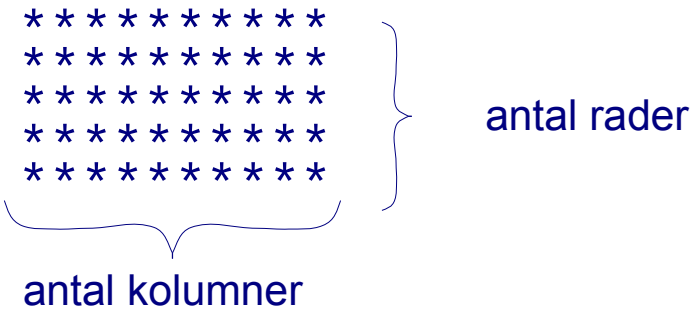
```
private static double readCapital() {
    double capital;
    while (true) {
        String input = JOptionPane.showInputDialog("Ge startkapital: ");
        capital = Double.parseDouble(input);
        if (capital > 0)
            break;
        JOptionPane.showMessageDialog(null, "Kapitalet måste vara positivt!!");
    }
    return capital;
}//readCapital
private static double readInterest() {
    double interest;
    while (true) {
        String input = JOptionPane.showInputDialog("Ange räntesats:");
        interest= Double.parseDouble(input);
        if (interest> 0)
            break;
        JOptionPane.showMessageDialog(null, "Räntesatsen måste vara positivt!!");
    }
    return interest;
}//readInterest
```

Implementation:

```
   private static void writeTable(int startYear, int endYear, double capital, double interest) {
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      String output = " År       Kapital\n";
      for (int year = startYear; year <= endYear; year = year + 1) {
         output = output + year + "       " + r.format(capital) + "\n";
         capital = capital * (1 + interest/100);
      }
      JOptionPane.showMessageDialog(null, output);
   }//writeTable
} // Capital
```

Problemexempel

Skriv ett program som läser in två heltal, som representerar antal rader respektive antal kolumner, och skriver ut följande mönster:



Design:

Algoritm:

1. Läs antal rader *r* och antal kolumner *k*
2. För varje rad
   2.1. För varje kolumn
      2.1.1. Skriv ut tecknet '*'
   2.2. Skriv ut radslut

Implementation:

```
import javax.swing.*;
import java.util.*;
public class WriteStars {
   public static void main(String[] args) {
      while (true) {
         String indata = JOptionPane.showInputDialog("Ange antal rader och antal kolumner: ");
         if (indata == null)
            break;
         Scanner sc = new Scanner(indata);
         int nrOfRows = sc.nextInt();
         int nrOfColumns = sc.nextInt();
         writeStars(nrOfRows, nrOfColumns);
      }
   } // main
   private static void writeStars(int nrOfRows, int nrOfColumns) {
      for (int row = 1; row <= nrOfRows; row = row +1) {
        for (int col = 1; col <= nrOfColumns; col = col +1)
           System.out.print("*");
         System.out.println();
      }
   }//writeStars
} //WriteStars
```

Fält

I ett program hantera man ofta samlingar av objekt av samma typ.

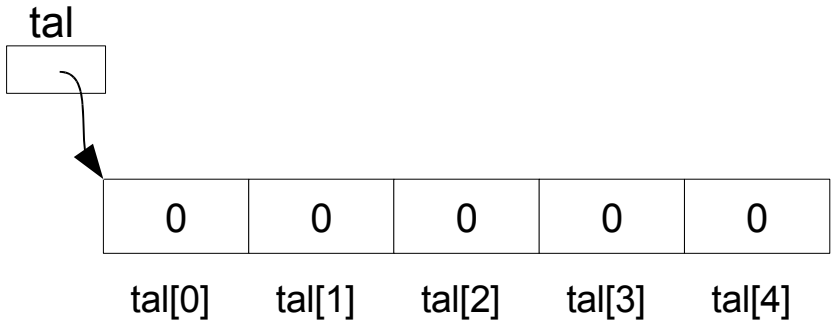
Sådana samlingar vill man vanligtvis kunna gruppera ihop till en sammanhängande *struktur*.

För detta ändamål tillhandahåller Java språkkonstruktioner för att hantera *fält*.

Fält

Ett fält är en numrerad samling av objekt, där varje delobjekt är av samma datatyp och delobjekten selekteras med *index*.

```
int [] tal = new int[5];
```

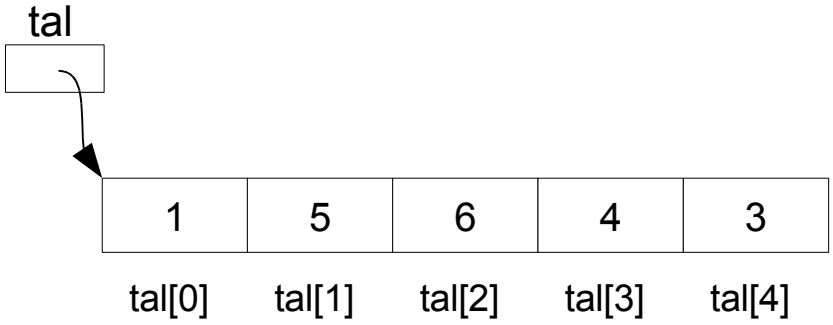


Indexering sker *alltid* från 0.

Oinitierade heltal får värdet 0.

Varje enskilt element i ett fält kan handhas individuellt via sitt index:

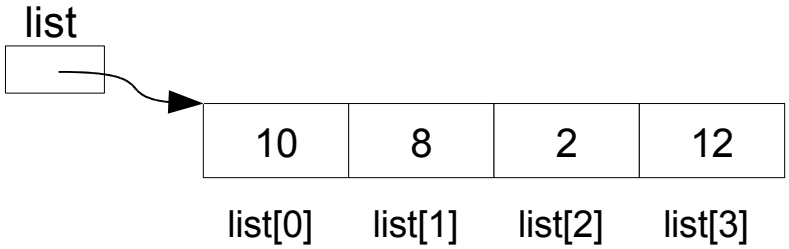
```
tal[0] = 1;
tal[1] = 5;
tal[2] = 6;
tal[3] = 4;
tal[4] = 3;
```



Fält

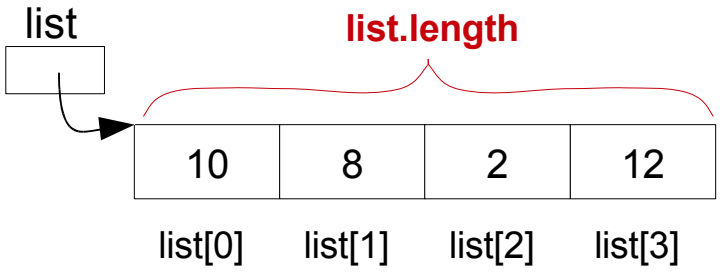
Istället för att skapa ett fält med **new** kan fältet skapas genom att initiera värden till fältet vid deklarationen. Antalet värden som då deklareras bestämmer fältets storlek.

```
int[] list ={10, 8, 2, 12};
```



Längden av ett fält fås av instansvariabeln *length*

```
int nrOfElements = list.length;
```



Fält

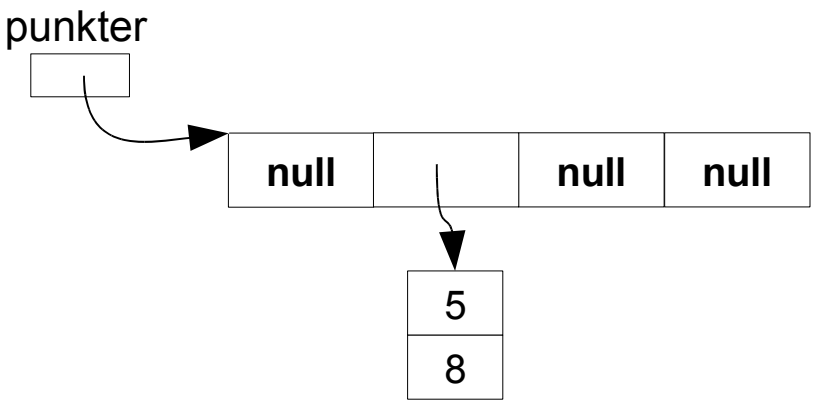
Fält kan skapas av godtycklig typ.

```
double [] numbers = new double[20];
String[] namn = {"Adam", "Beda", "Cesar", "David", "Erik", "Fabian"};
Punkt[] punkter= new Punkt[4];
```

Klassen `Punkt` är definierad enligt:

```
public class Punkt {
    private int x, y;
    public Punkt(int x, int y) {
        this.x = x;
        this.y = y;
    }
    . . . //fler metoder
}
```

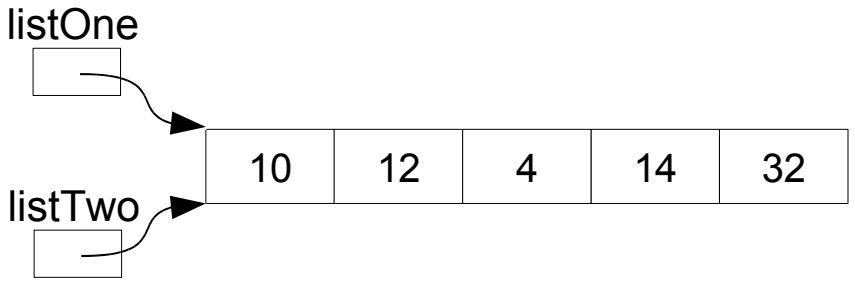
```
Punkt[] punkter = new Punkter[4];
Punkter[1] = new Punkt(5, 8);
```



Fält

Tilldelning ger ingen kopia, utan en referens till samma fält!

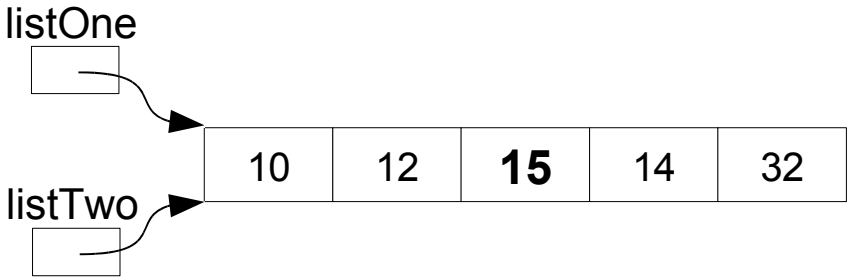
```
int[] listOne = {10, 12, 4, 14, 32};
int[] listTwo = listOne;
```



Satsen

```
listTwo[2] = 15;
```

resulterar således i att även elementet `listOne[2]` får värdet 15!



Klassen `java.util.Arrays`

I klassen `java.util.Arrays` finns ett antal klassmetoder som är användbara när man arbetar med fält

boolean equal(**int**[] f1, **int**[] f2)

returnerar **true** om fälten *f1* och *f2* är lika långa och motsvarande komponenter är lika, annars returneras **false**

int[] copyOf(**int**[] org, **int** newLength)

returnerar en kopia av fältet `org` med längden `newLength`. Om det kopian är kortare än `org` sker en trunkering och är kopian längre fylls kopian ut med 0:or.

int[] copyOfRange(**int**[] org, **int** from, **int** to)

returnerar ett fält som innehåller de elementen från index `from` till index `to` i fältet `org`.

String toString(**int**[] f)

returnerar en strängrepresentation av fältet *f*

Klassen `java.util.Arrays`

void fill(**int**[] f, **int** v)

sätter alla element i fältet *f* till värdet *v*

void fill(**int**[] f, **int** i1, **int** i2, int v)

sätter elementen från index *i1* till index *i2*-1 i fältet *f* till värdet *v*

void sort(**int**[] f)

sorterar fältet *f* i stigande ordning

void sort(**int**[] f, **int** i1, **int** i2)

sorterar elementen från index *i1* till index *i2*-1 i fältet *f* i stigande ordning

void binarySearch(**int**[] f, **int** k)

söker efter komponenten *k* i fältet *f*. Ger index för *k* om *k* finns i fältet *f*, annars fås ett värde < 0. Obs fältet måste vara sorterat!

Samtliga dessa metoder finns också för andra typer av fält, t.ex. **double**[], **boolean**[] och **char**[]!

Användning av metoder i `java.util.Arrays`

Exempel: Att sortera ett fält

```
import java.util.*;
. . .
   int[] list ={5, 4, 3, 8, 1, 9, 6, 7, 2};
   Arrays.sort(list);
```

Ordningen på elementen i fältet `list` är nu

1, 2, 3, 4, 5, 6, 7, 8, 9

Exempel: Att lokalisera ett element i ett sorterat fält

```
import java.util.*;
   . . .
   int[] list ={5, 4, 3, 8, 1, 9, 6, 7, 2};
   Arrays.sort(list); //sortera fältet
   int index = Arrays.binarySearch(list, 9);
   if (index >= 0)
      System.out.println("Talet 9 finns i index" + index);
   else
      System.out.println("Talet 9 finns INTE i fältet!");
```

Observera att sortering kan innebära att man ”förstör” fältet, ifall om den inbördes ordningen av elementen i fältet har betydelse för applikationen.

Att genomlöpa ett fält.

För att genomlöpa alla elementen i ett fält används normalt en **for-** loop

```
int[] list = new int[20];
. . .
for (int i = 0; i < list.length; i = i + 1) {
    // gör de bearbetningar av elementen
    // som skall göras
}
```

Exempel: Summera talen i heltalsfältet `list`

```
int sum = 0;
for (int i = 0; i < list.length; i = i + 1)
    sum = sum + list[i];
```

Att söka i ett *osorterat* fält.

Ett första försök: Använd en **for**-sats för genomsökning av hela listan

```
int [] list;
. . . .
String indata = JOptionPane.showInputDialog("Ange talet som söks:");
int talet = Integer.parseInt(indata);
boolean funnen = false;
int index = 0; // måste initieras
for (int i = 0; i < list.length; i = i + 1) {
    if (talet == list[i]) {
        funnen = true;
        index = i;
    }
}
if (funnen)
    JOptionPane.showMessageDialog(null, "Talet finns i index " + index);
else
    JOptionPane.showMessageDialog(null, "Talet finns INTE !");
```

Innan vi börjar sökningen har vi **inte** funnit vad vi söker

Nu har vi funnit det vi söker

Anm: När vi funnit vad vi söker fortsätter sökningen ändock till slutet av listan! Onödigt! Sluta när vi hittat vad vi söker!

Att söka i ett *osorterat* fält.

En bättre lösning: Använd en **while**-sats och en boolsk flagga för att kunna sluta när vi funnit det vi söker

```
int [] list;
. . .
String indata = JOptionPane.showInputDialog("Ange talet som söks:");
int talet = Integer.parseInt(indata);
boolean funnen = false; //har inte hittat det vi söker
int index = 0, i = 0;
while (i < list.length && ! funnen) {
    if (talet == list[i]) {
        funnen = true; //hittar det vi söker
        index = i;
    }
    i = i + 1;
}
if (funnen)
    JOptionPane.showMessageDialog(null, "Talet finns i index " + index);
else
    JOptionPane.showMessageDialog(null, "Talet finns INTE !");
```

Avbryt sökningen när vi funnit det vi söker eller då hela fältet är genomsökt

Att söka i ett osorterat fält.

En svårbegriplig och därmed en dålig lösning:

```
int [] list;
. . . .
String indata = JOptionPane.showInputDialog("Ange talet som söks:");
int talet = Integer.parseInt(indata);
int index;
for (index = 0; index < list.length && talet != list[index]; index++)
    ;
if (index != list.length)
    JOptionPane.showMessageDialog(null, "Talet finns i index " + index);
else
    JOptionPane.showMessageDialog(null, "Talet finns INTE !");
```

Dessvärre ser man denna typ av lösningar i många läroböcker!!!

En metod som söker i ett *osorterat* fält.

Att ta reda på om ett specifikt element finns eller inte finns i en osorterat fält är i många tillämpningar ett återkommande delproblem, varför det kan vara lämpligt att skapa en metod som utför sökningen. Vi lägger metoden i en klass `ArrayUtils`.

```
public class ArrayUtil {
   // Metoden returnerar värdet true om elem finns i fältet theList,
   // annars returnerar metoden värdet false.
   public static boolean foundInList(int[] theList, int elem) {
      boolean found = false; //har inte hittat det vi söker
      int i = 0;
      while (i < theList.length && ! found) {
         if (elem == theList[i]) {
            found = true; //hittar det vi söker
         }
         i = i + 1;
      }
      return found;
   }//foundInList
   . . .
}//ArrayUtil
```

En metod som söker i ett *osorterat* fält.

I många tillämpningar behöver man även få reda på i vilket *index* ett eftersökt element finns. Vi lägger till en sådan metoden i en klass `ArrayUtils`.

```
public class ArrayUtil {
    . . .
    // Metoden returnerar index för första förekomsten av elem
    // i fältet theList, finns inte elem i theList returneras värdet -1
    public static int firstIndexOf(int[] theList, int elem) {
        boolean found = false; //har inte hittat det vi söker
        int index = -1, i = 0;
        while (i < theList.length && ! found) {
            if (elem == theList[i]) {
                found = true; //hittar det vi söker
                index = i;
            }
            i = i + 1;
        }
        return index;
    }//firstIndexOf
}//ArrayUtil
```

Problemexempel

En indatasekvens består av maximalt 100 osorterade positiva heltal. I indatasekvensen kan samma tal förekomma flera gånger. Skriv ett program som lagrar indatasekvensen i ett fält på så sätt att alla dubbletter är borttagna, dvs endast första förekomsten av varje unikt tal skall lagras i fältet. Därefter skall programmet skriva ut fältet.

Inmatningen görs med användning av dialogrutor och datasekvensen avslutas att trycka på `Cancel`-knappen.

Analys:

Indata:	En sekvens av maximalt 100 positiva heltal.
Utdata:	Den inlästa sekvensen med alla dubbletter borttagna.
Exempel:	Indatasekvensen 1 4 1 2 4 5 12 3 2 4 1 ger utskriften 1 4 2 5 12 3

Design:

Diskussion: För att veta om ett tal har förekommit i indatasekvensen tidigare måste vi lagra den första förekomsten av talen i ett fält som vi kan kalla *listan*. När ett tal läses kontrollerar vi om talet redan finns i fältet *listan*. Om talet inte finns skall det lagras i *listan*.

Algoritm:

1. *antalElement* = 0;
2. **while** (fler tal att läsa)
   - 2.1. Läs *talet*
   - 2.2. Kontrollera om *talet* finns i *listan*
   - 2.3. **if** (*talet* inte finns i *listan*)
     - 2.3.1. *listan[antalElement] = talet*;
     - 2.3.2. *antalElement = antalElement*+1;
3. Skriv ut *listan*

Datarepresentation:

talet är av datatypen **int**.

listan är av datatypen **int**[100].

antalElement är av datatypen **int**.

Implementation:

```
import javax.swing.*;
import java.util.*;
public class NoDublicate {
   private static int[] readLista() {
      final int MAX = 100;
      int[] listan = new int[MAX];
      int antalElement = 0;
      String indata = JOptionPane.showInputDialog("Ange  indataserien: ");
      if (indata != null) {
         Scanner sc = new Scanner(indata);
         while (sc.hasNextInt()) {
            int talet = sc.nextInt();
            if (!ArrayUtil.foundInList(Arrays.copyOf(listan, antalElement), talet)) {
               listan[antalElement] = talet;
              antalElement = antalElement + 1;
            }
         }//while
      }//if
      return Arrays.copyOf(listan, antalElement);
   } // readLista
```

Genomsök endast den del av fältet som innehåller element

Returnera endast den del av Fältet som innehåller element

Implementation: Fortsättning

```
public static void writeLista(int[] listan) {
        if (listan.length == 0)
            JOptionPane.showMessageDialog(null, "Listan är tom");
        else {
            String utdata = "";
            for (int i = 0; i < listan.length; i = i +1)
                utdata = utdata + listan[i] + "    ";
            JOptionPane.showMessageDialog(null, utdata);
        }
    } // writeLista

    public static void main(String[] args) {
        int [] lista = readLista();
        writeLista(lista);
    } // main
} //NoDublicate
```

Defensiv programmering

VAD HÄNDER OM VI FÖRSÖKER LÄSA IN MER ÄN 100 ELEMENT TILL FÄLTET??

ETT FEL UPPSTÅR OCH EXEKVERINGEN AV PROGRAMMET AVBRYTS MED EN FELUTSKRIFT!!

Vi försöker referera till ett index som ligger utanför fältets gränser och ett undantag (*exception*) av typen `ArrayIndexOutOfBounds` inträffar.

Vi kan skydda oss mot denna situation genom *defensiv programmering*!

Anm:

Java har en speciell konstruktion (**try - catch**) med vilken man kan fånga upp undantag och åtgärda det fel som uppstått på ett eller annat sätt.

Undantag behandlas dock i denna kurs.

Defensiv programmering: Om fältet är för litet för att rymma alla elementen, skapa ett nytt fält som är 50% större

```
private static int[] readLista() {
        int[] listan = new int[100];
        int antalElement = 0;
        String indata = JOptionPane.showInputDialog("Ange  indataserien: ");
        if (indata != null) {
            Scanner sc = new Scanner(indata);
            while (sc.hasNextInt()) {
                int talet = sc.nextInt();
                if (!ArrayUtil.foundInList(Arrays.copyOf(listan, antalElement), talet)) {
                    if (antalElement == listan.length) {
                        listan = Arrays.copyOf(listan, (int) (1.5*listan.length));
                    }
                    listan[antalElement] = talet;
                    antalElement = antalElement + 1;
                }
            }//while
        }
        return Arrays.copyOf(listan, antalElement);   // returnera den "använda delen" av fältet
    } // readLista
```

Anm: Med denna variant av metoden `readList` gäller inte längre begränsningen att indatasekvensen maximalt kan bestå av 100 tal, utan nu kan indatasekvensen vara godtyckligt lång.

Parallella fält.

Två fält kallas för *parallella fält* om den data som finns i motsvarande index i de båda fälten är logiskt relaterade till varandra på något sätt.

Exempel:

Antag att vi har en golftävling med 5 deltagare. Vi kan lagra deltagarnas namnen i en fält, deltagarnas startnummer i ett annat fält och deltagarnas resultat i ett tredje fält:

```
String[] namn = new String[5];
int[] startNr = new int[5];
int[] score = new int[5];
```

namn	startNr	score
"Kalle"	4	74
"Anna"	5	67
"Stina"	2	73
"Åsa"	1	70
"Sven"	3	68

Är dessa fält parallella gäller att varje index `k` i fältet `namn` är relaterat till index `k` i fälten `startNr` och `score`, dvs `"Stina"` hade startnummer `2` och gick banan på `73` slag.

Obs: I just detta exempel hade det givetvis varit bättre att skapat en klass `GolfPlayer` med instansvariablerna `namn`, `startNr` samt `score` och istället använt ett enda fält med objekt av denna klass.

Parametrar till `main`.

Metoden `main` har en parameterlista som utgörs av ett fält av strängar:

```
public static void main(String[] args)
```

Detta innebär att man kan ge indata till `main`-metoden via parameterlistan.

Betrakta `main`-metoden i klassen `Demo` nedan. Vad `main` gör är att skriva ut de strängar som finns i dess parameter `args`.

```
public class Demo {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i = i + 1)
         System.out.println("Argument " + i + " = " + args[i]);
   }//main
}//Demo
```

Innehåller parametern `args` strängarna "Anna", "Beda" och "Doris" blir utskriften

```
Argument 0 = Anna
Argument 1 = Beda
Argument 2 = Doris
```

Parametrar till `main`.

Metoden `main` har en parameterlista som utgörs av ett fält av strängar:

```
public static void main(String[] args)
```

Detta innebär att man kan ge indata till `main`-metoden via parameterlistan.

Betrakta `main`-metoden i klassen `Demo` nedan. Vad `main` gör är att skriva ut de strängar som finns i dess parameter `args`.

```
public class Demo {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i = i + 1)
         System.out.println("Argument " + i + " = " + args[i]);
   }//main
}//Demo
```

Innehåller parametern `args` strängarna "Anna", "Beda" och "Doris" blir utskriften

```
Argument 0 = Anna
Argument 1 = Beda
Argument 2 = Doris
```

Parametrar till `main`.

Argumenten till `main`-metoden ges när exekveringen av programmet startas. Startas exekveringen från kommandofönstret skriver man alltså

```
java Demo Anna Beda Doris
```

Sker exekveringen från jGrasp skapar man ett kommandorad i vilket argumenten ges, genom att trycka på "Run Arguments" i menyn "Run".