2.8 Definition och initiering av en array

Följande program testar allt vi sagt i förra avsnitt om array speciellt indexregeln. Utöver det visas ytterligare en egenskap hos array som relaterar den till objekt, nämligen en egenskap **`Length`** som lagrar arrayens storlek när den skapas. Programmet demonstrerar också vad som händer om man överskrider arrayens maximala index: Man kan kompilera, men inte exekvera – ett tecken på att arrayens allokering sker vid *run time*.

```
// Array.cs
// Definierar en array av 4 int-värden, visar default-initie-
// ringsvärdena 0, tilldelar och skriver ut de nya värden
// Skriver ut arrayens storlek med Array-egenskapen Length
// Överskridning av arrayens index leder till exekveringsfel
using System;

class Array
{
   static void Main()
   {
      int[] no;                      // Deklarerar en referens
                                     // utan att skapa arrayen
      no = new int[4];               // new skapar arrayen vars
                                     // adress tilldelas referens
      // int[] no = new int[4];      // Alternativt i EN sats
      Console.WriteLine("Default-initiering:");
      for (int i=0; i < no.Length; i++)
          Console.WriteLine("Arrayens " + (i+1) + ":a element" +
                    " med index " + i + " har värdet " + no[i]);
      no[0] = 64;                   // Tilldelar 1:a elementet
      no[1] = 86;                   // värdet 64 osv. Överskriver
      no[2] = 34;                   // default-initieringen
      no[3] = -6;
      Console.WriteLine("\nEfter tilldelning:");
      for (int i=0; i < no.Length; i++)
          Console.WriteLine("Arrayens " + (i+1) + ":a element" +
                   " med index " + i + " har värdet " + no[i]);
      Console.WriteLine(
        "\n\tÖverskridning av arrayens index leder till "      +
        "exekveringsfel!\n\n\tEx.: no[4] inte definierad\n\t"  +
        "     Index 4 överskrider gränsen: Programavbrott!")   ;
      no[4] = 1;                    // no[4] kan kompileras, men
   }                                // leder till exekveringsfel
}
```

Inte alla satser i programmet **`Array`** exekveras. Det blir avbrott när den kompilerade koden **`no[4]`** i allra sista satsen ska exekveras där index **`4`** överstiger arrayens tillåtna maximala indexgräns som är **`3`** därför att **`new`** i början av programmet allokerar endast **`4`** minnesceller åt arrayen, nämligen de med index **`0`**, **`1`**, **`2`** och **`3`**. Någon minnescell med index **`4`** är inte allokerad. Därför kan vi inte heller referera till den med **`no[4]`**. Men ef-

tersom arrayens allokering sker med **new** och därmed under exekveringstid (eng. *run time*) leder detta till exekveringsfel, medan kompilatorn godtar den syntaxmässigt korrekta koden **no[4]**. Programmet **Array** ger följande utskrift när man kör det:

```
Default-initiering:
Arrayens 1:a element med index 0 har värdet 0
Arrayens 2:a element med index 1 har värdet 0
Arrayens 3:a element med index 2 har värdet 0
Arrayens 4:a element med index 3 har värdet 0

Efter tilldelning:
Arrayens 1:a element med index 0 har värdet 64
Arrayens 2:a element med index 1 har värdet 86
Arrayens 3:a element med index 2 har värdet 34
Arrayens 4:a element med index 3 har värdet -6

       Överskridning av arrayens index leder till exekveringsfel!

       Ex.: no[4] inte definierad
            Index 4 överskrider gränsen: Programavbrott!

Unhandled Exception: System.IndexOutOfRangeException: Index was
outside the bounds of the array.
at Array.Main(String[] a) in C:\C#\MyProject\54Array.cs:line 33
```

Vi drar slutsatsen:

> Att referera till icke-definierade element i en array leder till exekveringsfel.

Man kan även säga att C#-interpretatorn (VM) kontrollerar indexgränserna och inte tillåter åtkomsten till icke-allokerade minnesplatser, vilket ur allmän datasäkerhetssynpunkt är en fördel. Programmen blir stabilare. Andra programmeringsspråk som C++ har i detta avseende en mer liberal attityd. Där ligger ansvaret för kontroll av indexgränserna helt och hållet hos programmeraren.

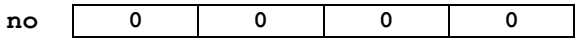
Man kan ju undra varför **no[4]** inte är definierat – som vi hävdar ovan – fast talet **4** ”förekommer” i definitionssatsen **new int[4]**. Detta beror på att hakparenteserna **[]** i **no[4]** inte har samma betydelse som i **new int[4]**. Den korrekta tolkningen av **[]** beror på sammanhanget. Man kan också säga att **[]** är symbolen för tre olika operatorer som överlagrar varandra dvs betyder olika i olika sammanhang (sid 139/139):

Hakparentesernas tre olika betydelser

1. **[] som storleksoperator** omsluter i definitioner med **new** *antalet* element i arrayen specificerar därmed arrayens *storlek*. T.ex. innebär koden

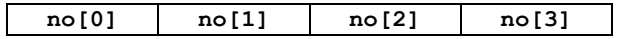
```
new int[4]
```

i programmet **`Array`** att **`new`** skapar en array av **`int`** med **`4`** element dvs att **`4`** minnesceller reserveras för lagring av **`int`**-värden. Det gemensamma för alla dessa element är att de lagras en efter den andra vid adressen eller referensen **`no`**:



Här är frågan om ”Hur många element?”. I matematiken kallas detta *kardinaltal*.

2. **`[]` som indexeringsoperator** omslutar *indexet* till varje element av en array. Här handlar det om ett elements *position* i arrayen. Man anger index inom hakparenteser för att referera till elementet när man vill hämta eller tilldela det ett värde. Indexregeln (sid 99) tillämpas enligt vilken indexeringen börjar med **`0`**. Därför är **`no[4]`** i arrayen ovan inte definierat:



Här är frågan om ”Vilket element?”. I matematiken kallas detta *ordinaltal*.

3. **`[]` som en del av datatypen** ”referens till array” omsluter ingenting utan är tom och skrivs direkt efter en datatyp för att definiera en ny referenstyp. T.ex. innebär satsen

```
int[] no;
```

i programmet **`Array`** att en minnescell allokeras (en referensvariabel med namnet **`no`** definieras) för lagring av en adress till en **`int`**-array. Vi kan i fortsättningen använda namnet **`no`** för att komma åt arrayen vid denna adress. I satsen ovan har referensen **`no`** inte initierats. Det sker inte heller automatiskt, för **`no`** är en lokal variabel i **`Main()`**. Det sker först med tilldelningen **`no = new int[4];`** som initierar rederensen explicit.

Default-initiering av en array

Det anmärkningsvärda är nu att det som gäller för referensen **`no`** – att den är oinitierad när den skapas – inte gäller för själva arrayen. Referensen **`no`** är oinitierad och måste initieras explicit eftersom den är en lokal variabel i **`Main()`**. Men trots att även arrayen är lokal i **`Main()`** initieras den till de default-värden vi nämnde för datamedlemmar i objekt (sid 83), vilket är ett tecken på att array även i detta avseende behandlas som objekt. Programmet **`Array`** skriver ut arrayelementens värden en gång *innan* och en andra gång *efter* att de har fått värdena 64, 86, 34 och -6. Utskriften på förra sidan visar för arrayens alla element initialvärdet **`0`** som är den föreskrivna default-initieringen för variabler av typ **`int`** vilket även gäller för element i en **`int`**-array. Generellt gäller:

> Alla element i en array initieras automatiskt till default-värden (precis som datamedlemmar i ett objekt) även om arrayen skapas lokalt i en metod.

Initieringslista

Precis som det finns skillnader i *definitionen* av arrayvariabler (referenser) jämfört med vanliga variabler, finns även skillnader vid *tilldelningen*. T.ex. är tilldelningen av arrayen **no** i programexemplet **Array** (sid 101) – en sats för varje element – inte särskilt effektiv, speciellt om man skulle tillämpa samma teknik på större arrayer med 100-tals eller fler element. Men just hanteringen av stora datamängder var ju motiveringen för att syssla med array. Man kan effektivisera hanteringen genom att använda sig av **for**-satser och av en s.k. *initieringslista*, en kortform som slår ihop definitionen med initieringen. Exempel på båda visas i följande program:

```
// ArrayInit.cs
// Kortform för definition och initiering av en array i EN och
// samma sats med hjälp av en initieringslista
// Elementvis utskrift av en array kan med fördel göras med en
// for-sats: Arrayens index = for-satsens räknare
using System;

class ArrayInit
{
   static void Main()
   {
      int[] no = { 64, 86, 34, -6 };      // Initieringslista:
                                           // Definition OCH ini-
                                           // tiering av en array
//    int[] no = new int[4] { 64, 86, 34, -6 };     // Gör samma
                                                    // sak
      Console.Write("\nVärdena från arrayen skrivs ut med" +
                    " referensen:\n\n\t");
      for (int i = 0; i < no.Length; i++)
          Console.Write(no[i] + "\t");
      int[] copy = no;                     // Ny referens till
                                           // samma array
      Console.Write("\n\nVärdena från arrayen skrivs ut"    +
                    " med den nya referensen copy:\n\n\t");
      for (int i = 0; i < copy.Length; i++)
          Console.Write(copy[i] + "\t");
      Console.WriteLine('\n');
   }
}
```

En körning visar att värdena i initieringslistan som först tillelas arrayen **no** verkligen kopierats över till arrayen **copy**, för det är de som skrivs ut:

```
Värdena från arrayen skrivs ut med referensen no:

        64      86      34      -6

Värdena från arrayen skrivs ut med den nya referensen copy:

        64      86      34      -6
```

Både definitionssatsen och initieringssatserna i programet **Array** (sid 101) – det är de 5 första satserna i **Main()** – kan slås ihop till en enda sats:

```
int[] no = { 64, 86, 34, -6 };
```

Satsen ovan är bara en förkortning på:

```
int[] no = new int[4] { 64, 86, 34, -6 };
```

Dvs initieringslistan kan skrivas efter **new int[4]** som egentligen skapar eller definierar arrayen. Men **new int[4]** får utelämnas. Detta visar att den förkortade versionen gör två saker: Först, fram till tilldelningstecknet definieras referensen **no** (*utan* någon uppgift om arrayens storlek). Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom klamrarna **{ }** som kallas arrayens *initieringslista*. Kortformen gör precis samma sak som satsen med **new**. Kompilatorn får informationen om arrayens storlek genom att i initieringslistan räkna antalet element inom klamrarna **{ }**. Det är inte ens tillåtet att explicit ange det korrekta antalet element inom hakparenteserna **[ ]**. Det blir kompileringsfel om man gör det, därför att **no** endast är en referens till en array, inte arrayen själv. Observera även att man inte får använda initieringslistan separat utan endast i samma sats som definitionen.

Valet av variabelnamnet **copy** kan vara missledande i följande sats av programmet **ArrayInit** om man inte beaktar skillnaden mellan referens och array:

```
int[] copy = no;
```

copy blir nämligen en kopia av referensen **no** i satsen ovan, inte av arrayen – en ny referens som kommer att peka på samma array som den gamla referensen **no** pekar på. Det skapas ingen ny array eftersom det varken finns någon **new** eller någon initieringslista som skulle ersätta **new**. Anledningen till detta är – som vi konstaterat tidigare – följande viktigt faktum:

> En array i C# är alltid ett objekt som behöver en referens.

För att skapa ett objekt måste en **new**-sats skrivas. En referens definieras utan **new**.

Minnesmässigt lagras arrayen på *en och samma* adress som från programmet kan nås med referenserna **no** eller **copy**: