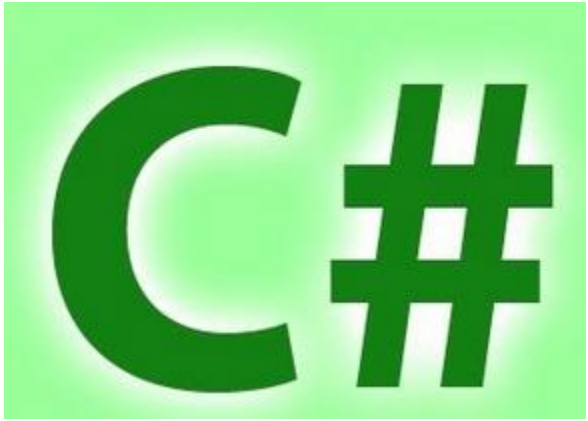
Stack & Heap

Speicherverwaltung in der Programmiersprache C#

Überblick

- Parameterübergabe
  - Call by Value
  - Call by Reference
  - Eingangsparameter
  - Ausgangsparameter
  - Übergansparameter
  - Array als Parameter
- Speicherverwaltung
  - Stack
  - Heap
  - Referenztypen
  - Wertetypen



Parameterübergabe

Call By Value

Call By Referenz

Parameter Wunschkonzert ;)

Es soll die Möglichkeit geben zu wählen ob ...

- eine Funktion einen Wert erhält um ihn zu benutzen, weiterzuverwenden, beliebig zu verändern ohne, dass es den Originalwert beeinflußt
- eine Funktion einen Wert erhält den man in der Funktion „befüllt“ damit man mehrere Rückgabewerte haben kann
- eine Funktion einen Wert erhält, den verändert und der Originalwert der außerhalb der Funktion existent ist wird mit aktualisiert.

Call By Value

- Parameterübergabe per Wert
- Kopie der Variable wird an die Methode übergeben
- genannt:
- Eingangsparameter

```
 static void Method( int i)
{
    i = 44;
}
static void Main()
{
    int value=0;
    Method(value);
    // value is now 0
    Console.WriteLine(value);
}
```

Call By Referenz

- Parameterübergabe per Referenz
- Speicherplatz wo die Variable gespeichert wird, wird an die Methode übergeben
- genannt:
- Übergangsparameter

```
static void Method( ref int i)
{
    i += 44;
}
static void Main()
{
    int value=22;
    Method( ref value);
    // value is now 66
    Console.WriteLine(value);
}
```

Arten von Parameter in C#

- Call by Value mit Wertübergabe
- Call by Referenz mit Adressübergabe
  - Ausgangsparameter:
    - Schlüsselwort: out
  - Übergangsparameter:
    - Schlüsselwort: ref
    - Unterschied zu out: Variable **muss** vor dem Methodenaufruf initialisiert sein

```
static void Method( out int i)
{
    i = 44;
}
static void Main()
{
    int value;
    Method( out value);
    // value is now 44
    Console.WriteLine(value);
}
```

Ausgangsparameter:

```
static void Method(out int i, out string s1, out string s2)
{
    i = 44;
    s1 = "I've been returned";
    s2 = null;
}
static void Main()
{
    int value;
    string str1, str2;
    Method(out value, out str1, out str2);
    Console.WriteLine("V: {0}, S1: {1}, S2: {2}",
        value, str1, str2);
}
```

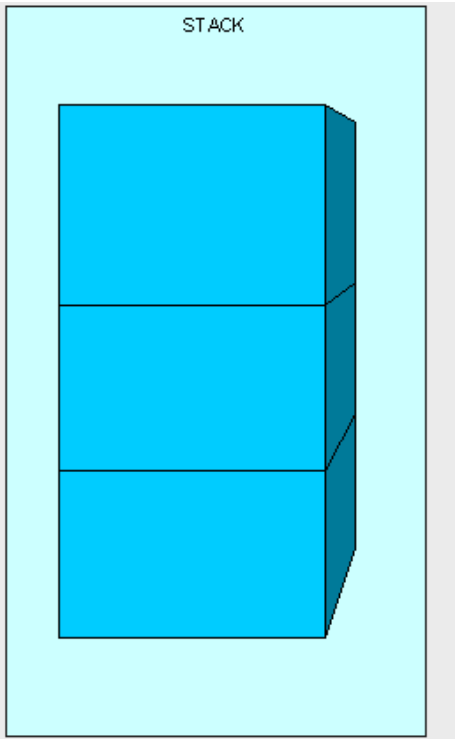
Wie lautet die Ausgabe?

Speicherverwaltung

Stack & Heap

Speicherverwaltung

- Speicher teilt sich in der Verwaltung durch den Hauptprozessor in zwei Bereiche auf:
  - **Stack:**
    - Datenelemente werden nach dem Prinzip Last-in-First-out verwaltet
    - wie ein Stapel (Stack)

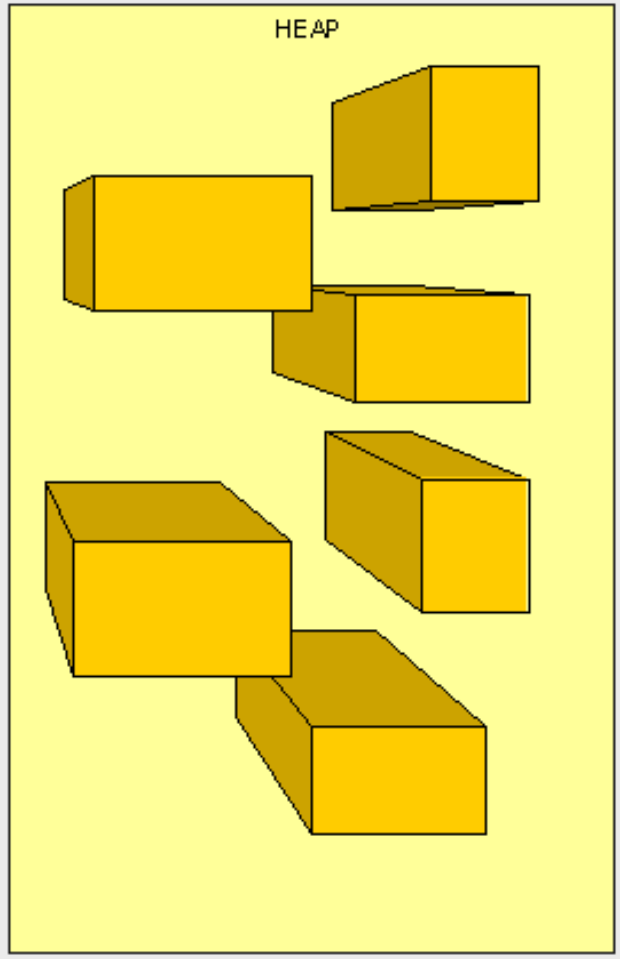


Stack

- Variablen von Wertetypen werden auf dem Stack gespeichert
- Stack ist ein Speicher mit einfacher Speicherverwaltung
  - Bei einem Funktionsaufruf wird im Stack ein fester Speicherbereich reserviert für alle Parameter und Wertetyp-Variablen dieser Funktion
  - Nach dem Funktionsaufruf wird dieser Speicherbereich wieder freigegeben
    - VT: effizient
    - NT: Lebensdauer der Variablen beschränkt
  - Wünschenswert -> Lebensdauer über Funktionsaufrufe hinweg

Speicherverwaltung

- Heap
  - Objekte, sog. Referenztypen, werden im Heap angelegt
  - Speicherallokierung findet an einem freien Platz im Heap statt, keine konkrete Reihenfolge
  - (in einer Referenz wird die Speicheradresse vermerkt, wo sich das Objekt im Heap befindet)



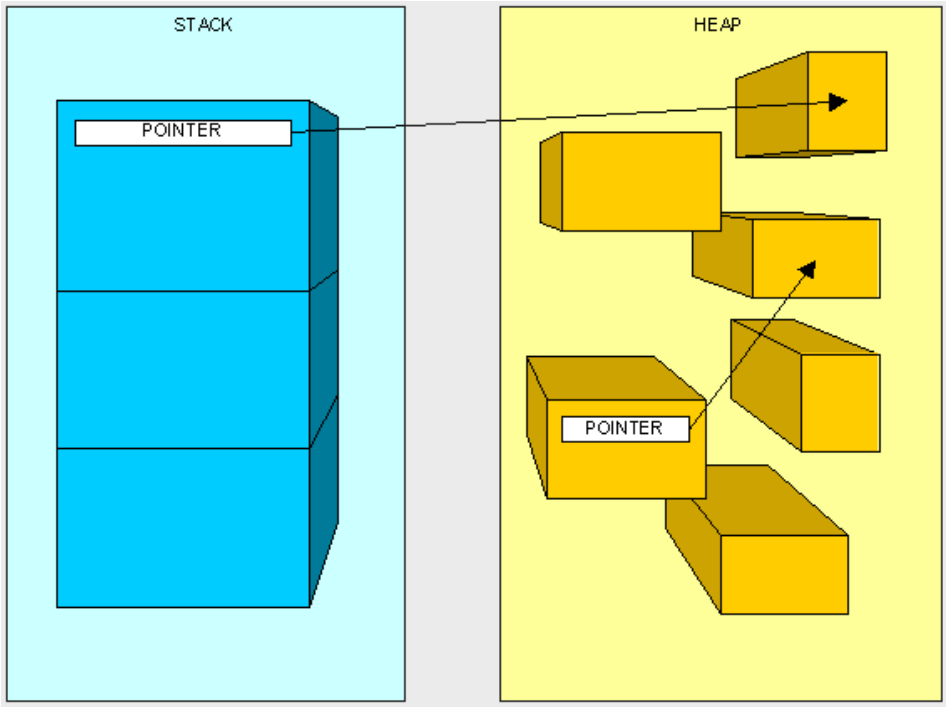
Heap

- Referenztypen werden am Heap gespeichert:
  - ein Objekt der Klasse wird am Heap gespeichert, die Objektvariable, die den Verweis (Referenz) auf das Objekt im Heap enthält wird am Stack abgelegt
  - Objekte am Heap bleiben solange bestehen, wie es Referenzen gibt die darauf verweisen
  - Dadurch ist die Lebensdauer über mehrere Funktionsaufrufe hinweg möglich!
    - Funktionen müssen nur die Objektverweise untereinander austauschen

Zeiger (Pointer) oder Referenzen

- Um auf einen Speicher im Heap zugreifen zu können benötigt man die Speicheradresse
  -> Referenz

- Referenzen
  können im Heap oder im Stack gespeichert werden.

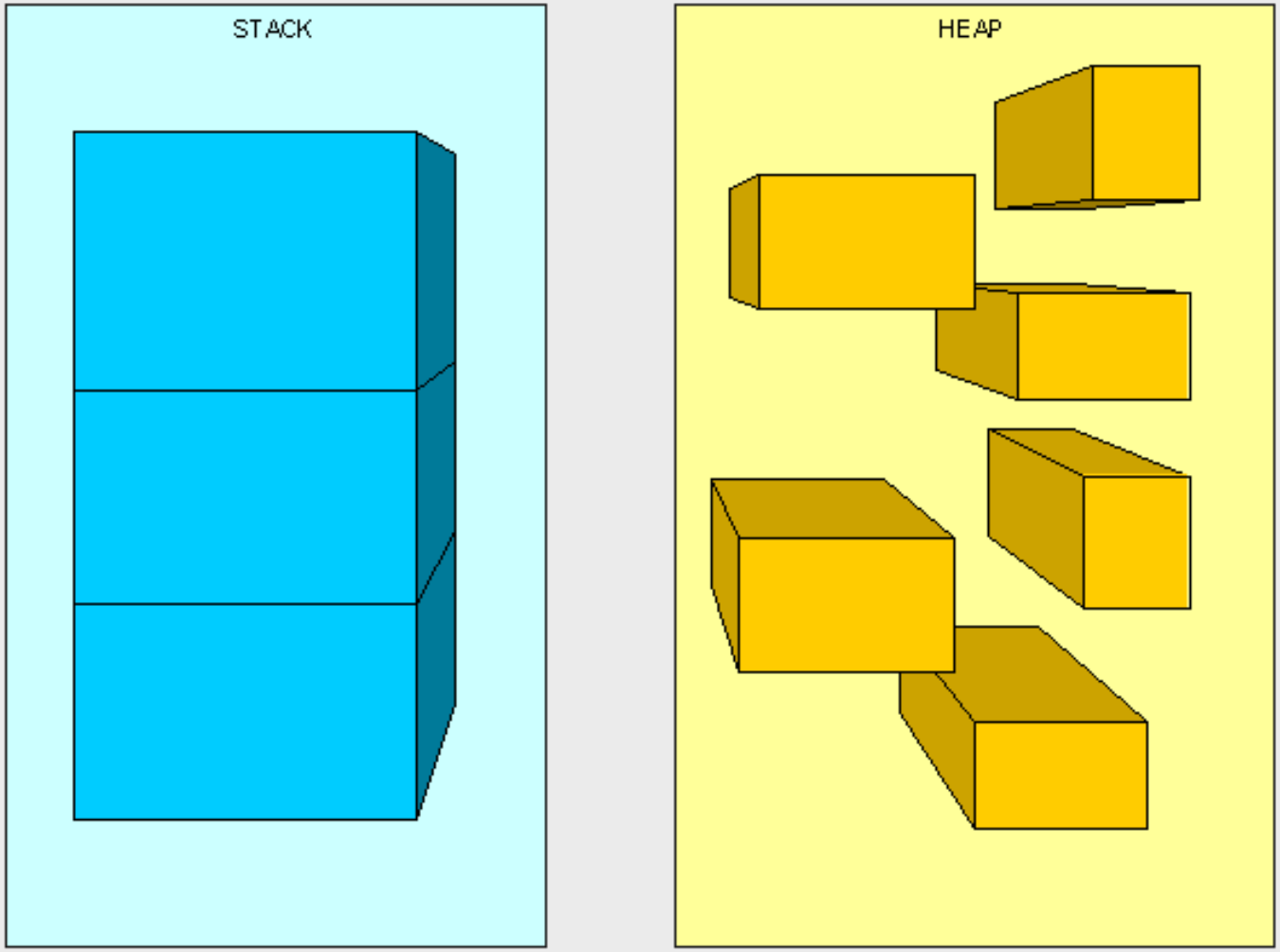


- en: refer
- de: verweisen, beziehen

Garbage Collection

- Automatische Speicherbereinigung des .NET Frameworks
- Wenn es keine Referenz mehr auf ein Objekt gibt, wird es vom Garbage Collector (GC) aufgelöst
- Nachteil der Heap-Speicherverwaltung:
  - relativ kompliziert & zeitaufwändiger

Was landet am Stack und am Heap?



Wie wird entschieden, was landet wo?

- Zwei goldene Regeln:
  - Referenztypen landen immer am HEAP (einfache Regel)
  - Wertetypen und Referenzen landen dort wo sie deklariert werden
  - (benötigt eine genauere Erklärung)

Unterscheidung Referenz- und Wertetypen

Wertetypen

- primitive Datentypen:
  - bool
  - byte
  - char
  - decimal
  - double
  - enum
  - float
  - int
  - long
  - sbyte
  - short
  - struct
  - uint
  - ulong
  - ushort
- Datentypen von System.ValueType

Referenztypen

zB:

- String
- Random
- StringBuilder
- Arrays
  - alle Objekte

    class

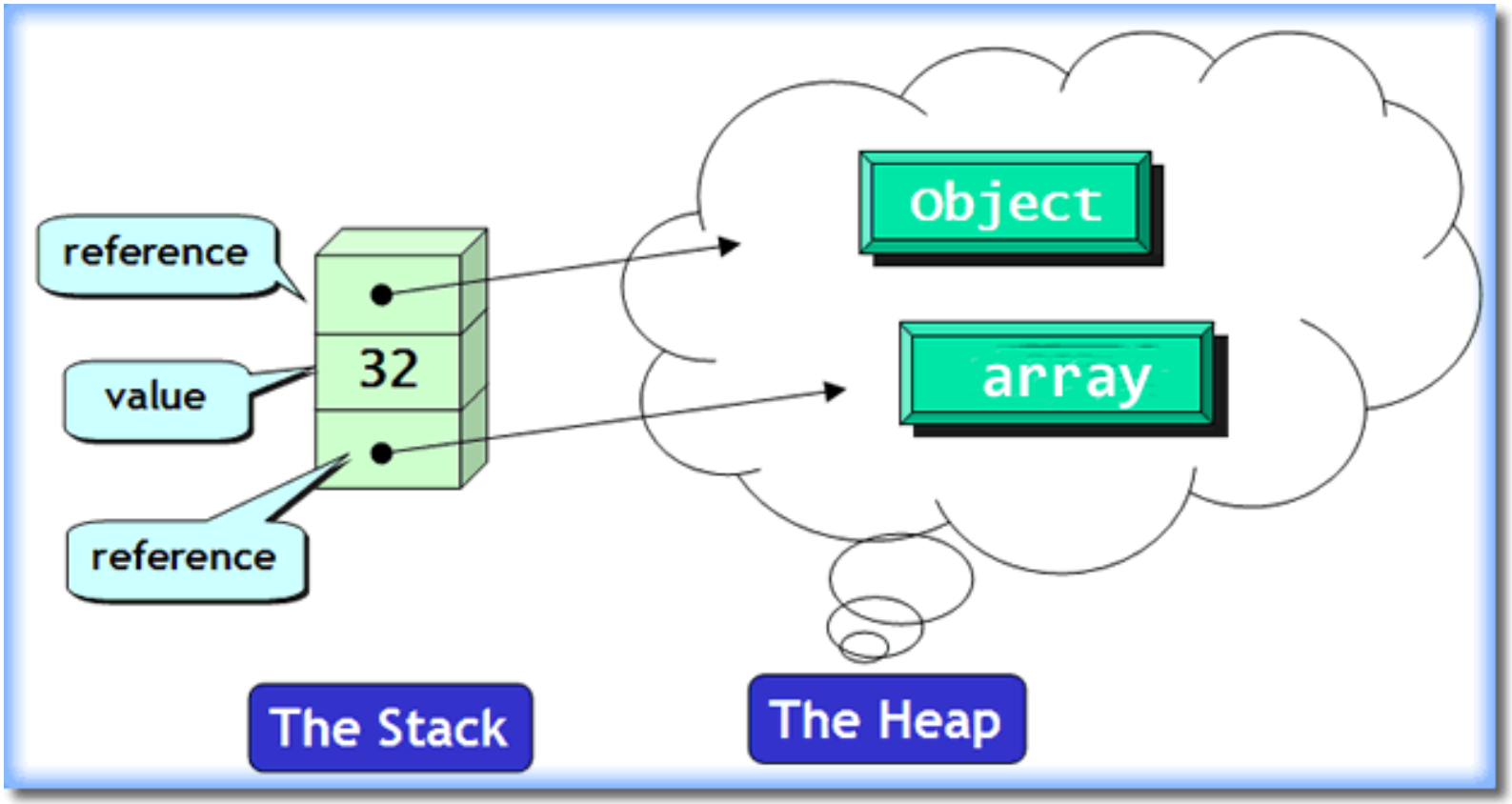
    interface

    delegate

    object

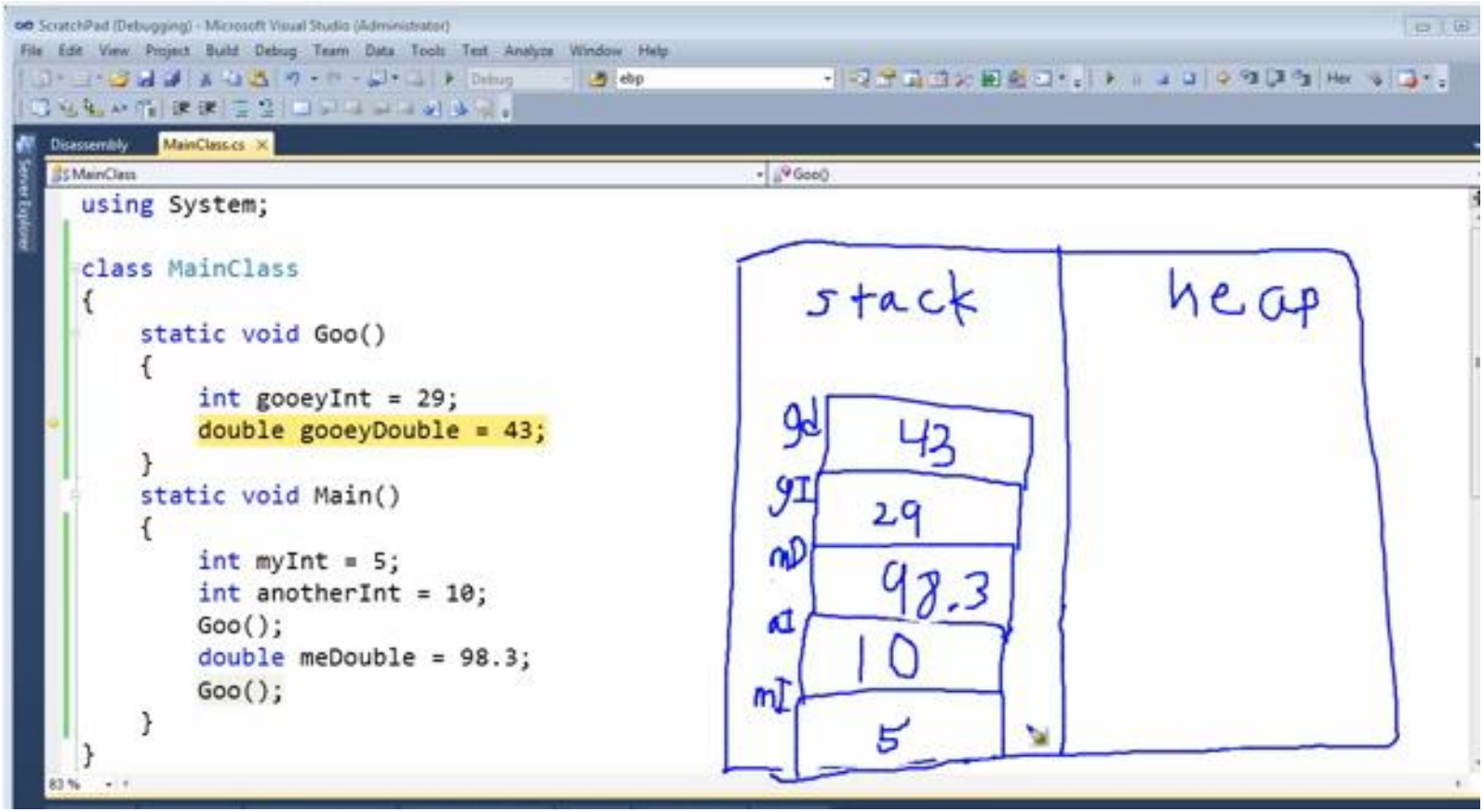
    string

Stack vs Heap



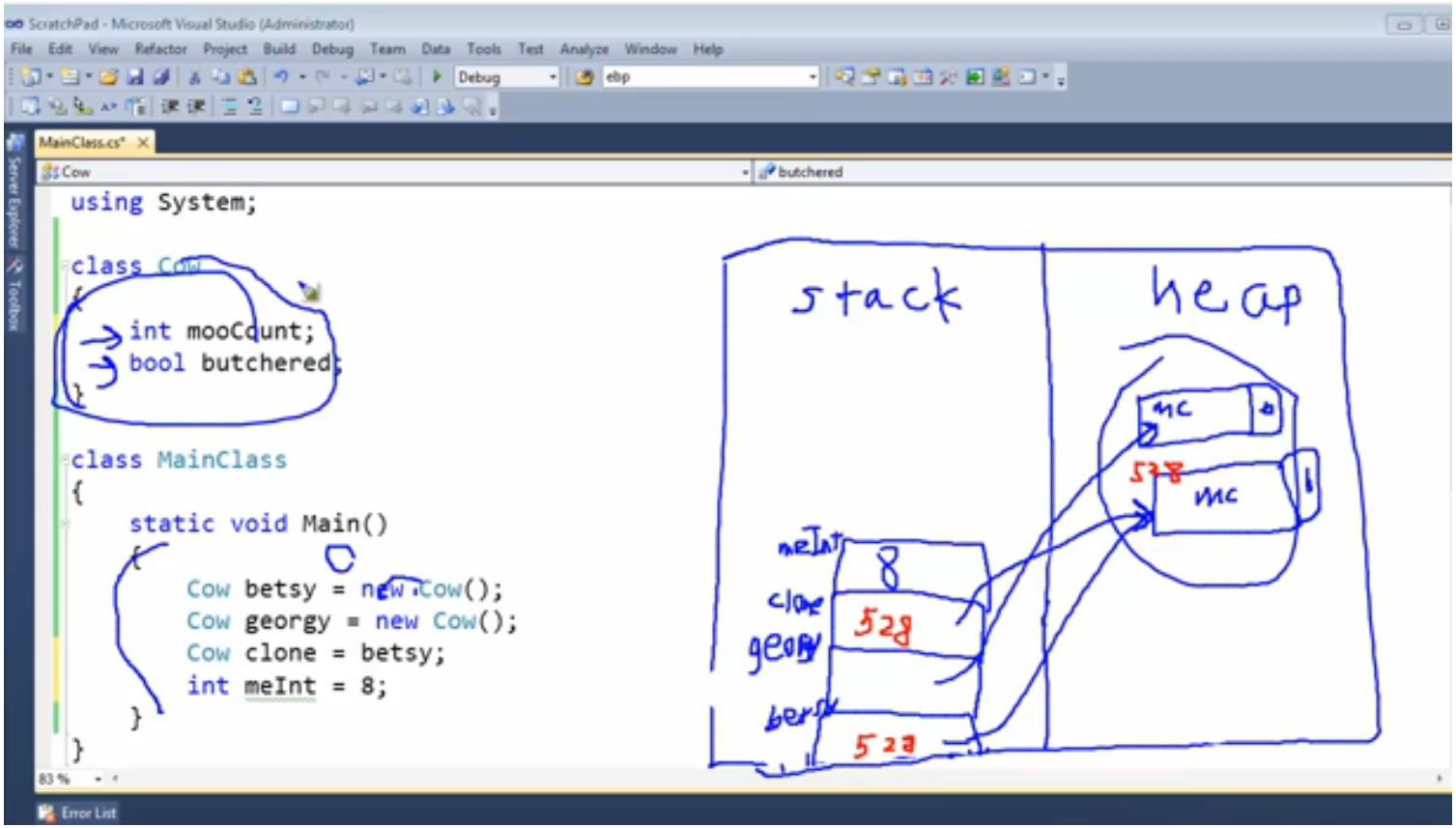
Stack

- https://www.youtube.com/watch?v=WOlrjNvM4jg



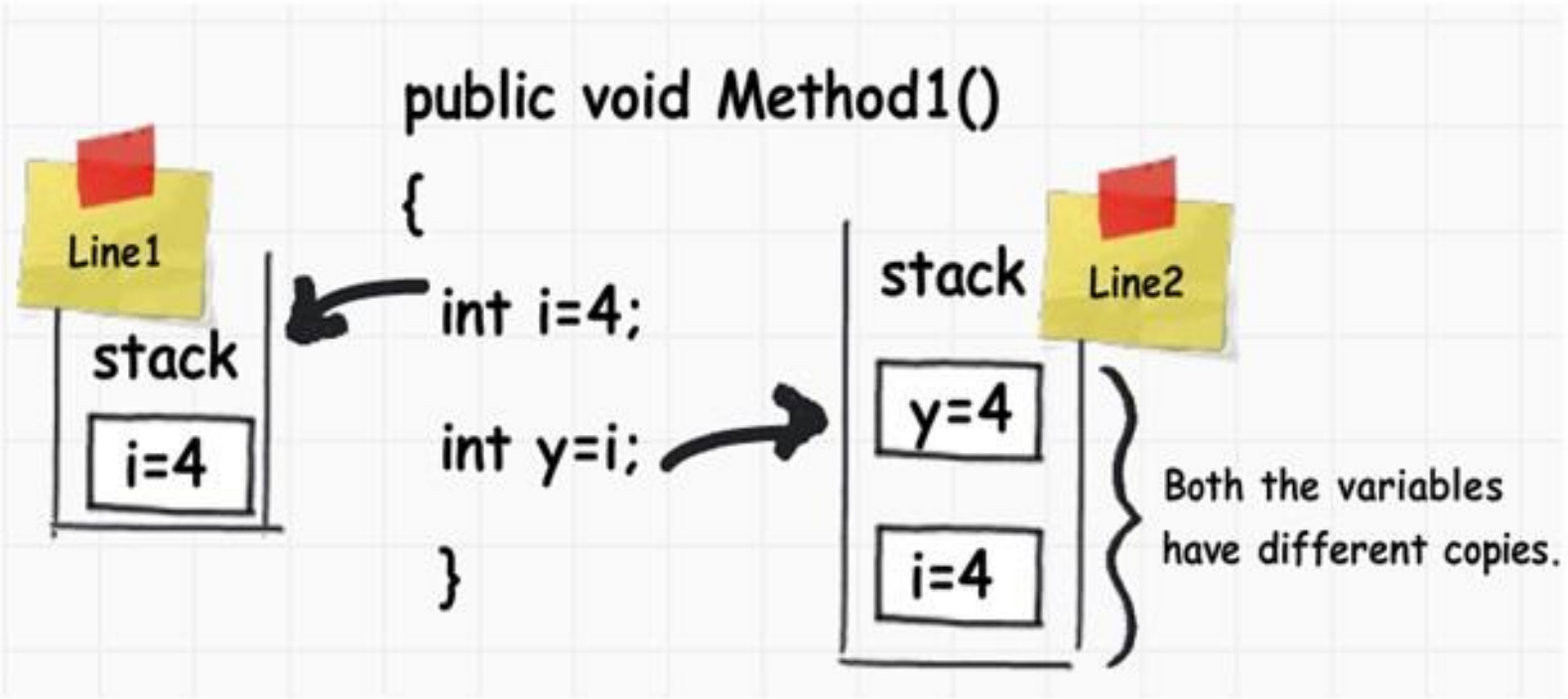
Heap

- https://www.youtube.com/watch?v=lLbKkClhzbU

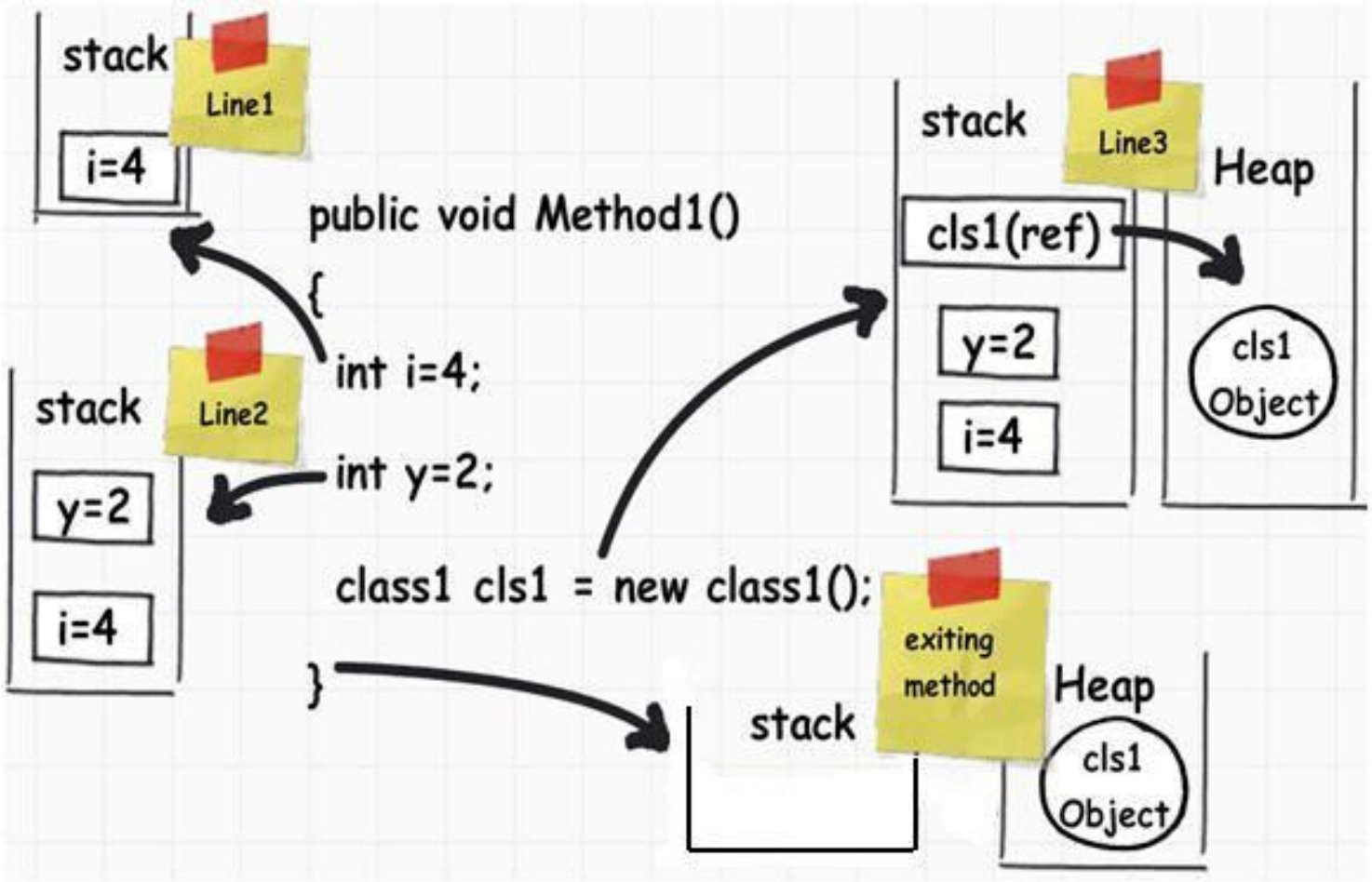


Wertedatentypen

- Primitive Datentypen landen am Stack:

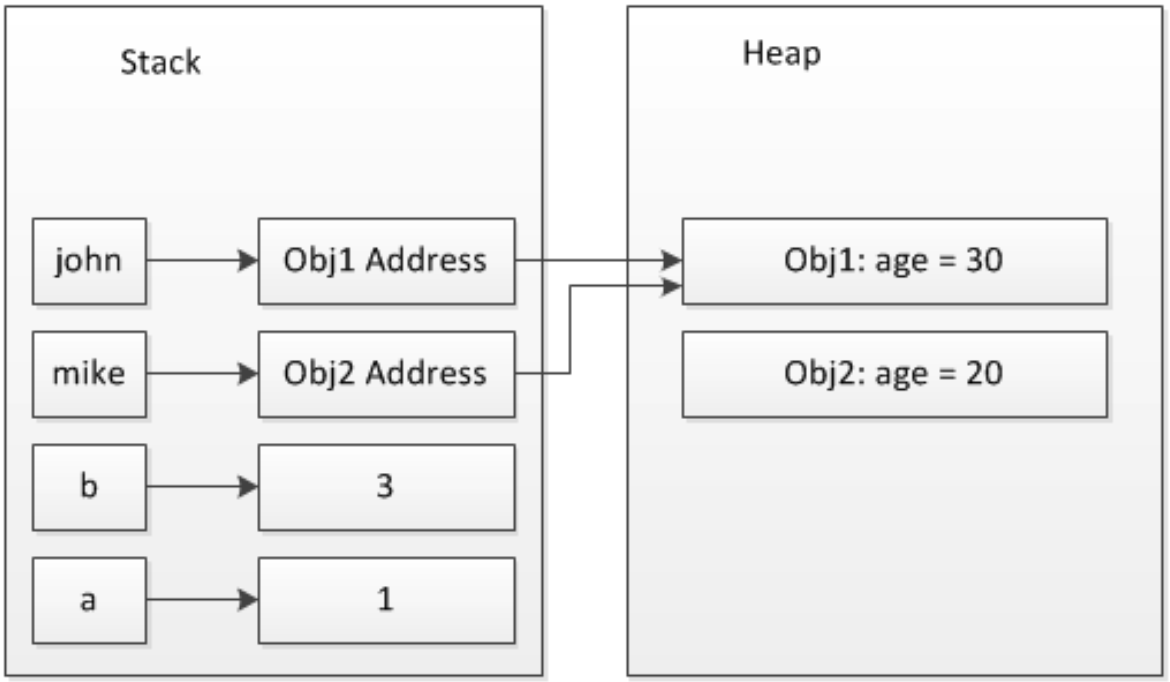


Wertetypen & Rerferenztypen



Stack & Heap

- String, StringBuilder & Random sind Referenztypen und entsprechen „Objekten“



Arrays als Parameter

ref & out bei Arrays als Parameter?!

Array als Parameter

- In der Main wird ein Array instantiiert, welches anschließend verworfen wird in der FillArray-Methode -> Sinnvoller out verwenden!

- Wie lautet die Ausgabe?
- Array elements are:
- 1, 2, 3, 4, 5

```
static void FillArray(int[] arr)
{
    // Initialize the array:
    arr = new int[5] { 1, 2, 3, 4, 5 };
}

static void Main()
{
    int[] theArray = new int[5];  // Initialization is not required

    // Pass the array to the callee using out:
    FillArray(theArray);

    // Display the array elements:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Ausgangsparameter mit Array

- Mit Hilfe von out kann ein Array in der Funktion instantiiert werden!

- Wie lautet die Ausgabe?
- Array elements are:
- 1 2 3 4 5

```
static void FillArray(out int[] arr)
{
    // Initialize the array:
    arr = new int[5] { 1, 2, 3, 4, 5 };
}

static void Main()
{
    int[] theArray; // Initialization is not required

    // Pass the array to the callee using out:
    FillArray(out theArray);

    // Display the array elements:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Übergangsparameter mit Arrays?

- Es macht keinen Sinn Arrays mit „ref“ zu übergeben, auch ohne ref ändern sich die Werte im Array, da es sich um einen Referenzdatentyp handelt.

- Wie lautet die Ausgabe?
- Array elements are:
- 1111 2 3 4 5555

```
static void FillArray(ref int[] arr)
{
    // Create the array on demand:
    if (arr == null)
    {
        arr = new int[10];
    }
    // Fill the array:
    arr[0] = 1111;
    arr[4] = 5555;
}

static void Main()
{
    // Initialize the array:
    int[] theArray = { 1, 2, 3, 4, 5 };

    // Pass the array using ref:
    FillArray(ref theArray);

    // Display the updated array:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Arrays als Parameter

- Ein Array kann ohne ref an eine Funktion übergeben werden, es gibt nur eine Version des Arrays am Heap, diese wird direkt verändert.
- Als Parameter wird eine Kopie der Adresse des Arrays übergeben.
- Wie lautet die Ausgabe?
- Array elements are:
- 10, 20, 30 111, 555, 50

```
static void FillArray(int[] arr)
{
    // Create the array on demand:
    if (arr == null) {
        arr = new int[10];
    }
    // Fill the array:
    arr[3] = 111;
    arr[4] = 555;
}

static void Main()
{
    // Initialize the array:
    int[] theArray = { 10, 20, 30, 40, 50 };

    // Pass the array using ref:
    FillArray(theArray);

    // Display the updated array:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

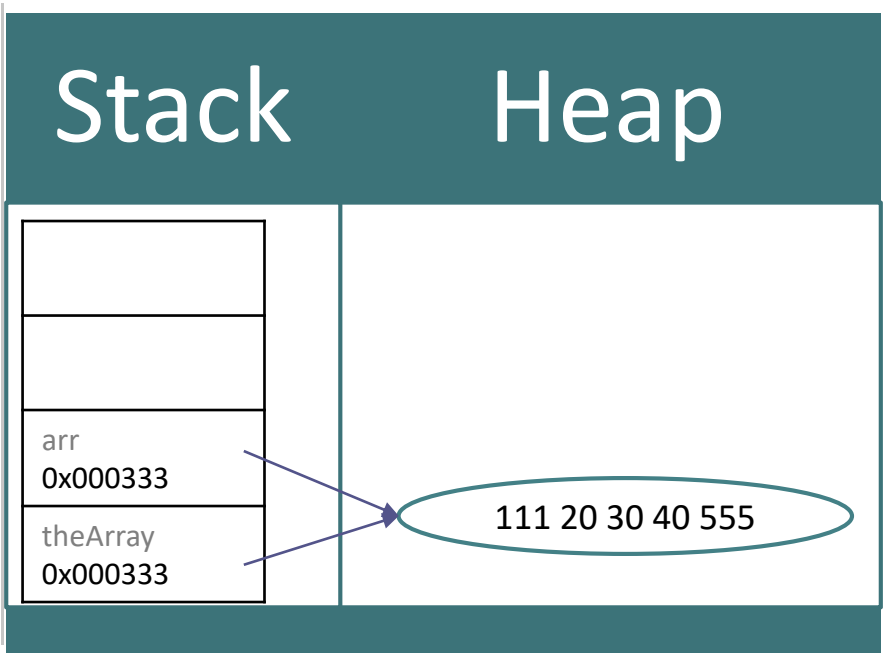
Ref & Out bei Arrays

- Auch Arrays kann man mit ref und out übergeben, allerdings sind Arrays Referenzdatentypen.
- Es wird die „Adresse des Arrays im Heap“ übergeben

- Ref macht für die Übergabe von Arrays keinen Sinn, kann weggelassen werden
- Out hat den Vorteil, dass man ein Array, welches nur deklariert jedoch nicht instantiiert ist übergeben kann – ohne out würde ein Compilerfehler auftreten…

Array als Parameter

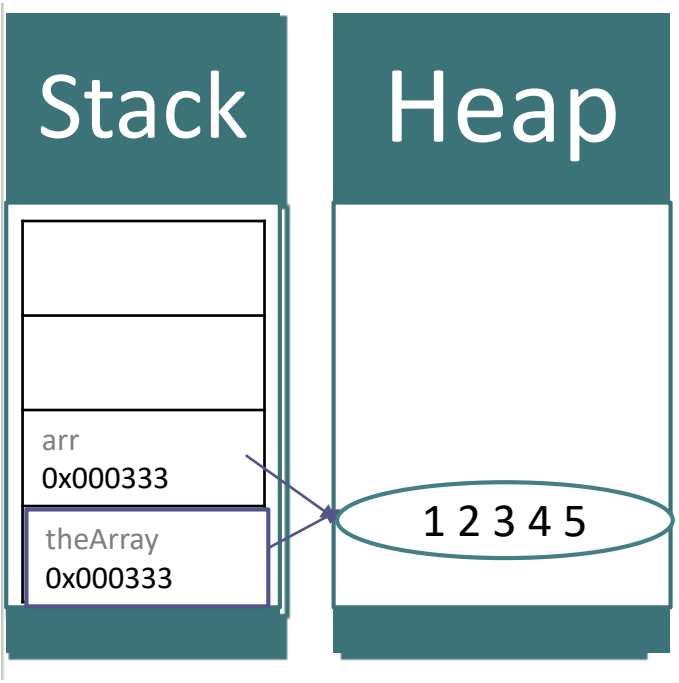
- Array elements are:
- 1111 2 3 4 5555




```
static void FillArray(int[] arr)
{
    // Create the array on demand:
    if (arr == null) {
        arr = new int[10];
    }
    // Fill the array:
    arr[3] = 111;
    arr[4] = 555;
}

static void Main()
{
    // Initialize the array:
    int[] theArray = { 10, 20, 30, 40, 50 };

    // Pass the array using ref:
    FillArray(theArray);

    // Display the updated array:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Ausgangsparameter mit Array

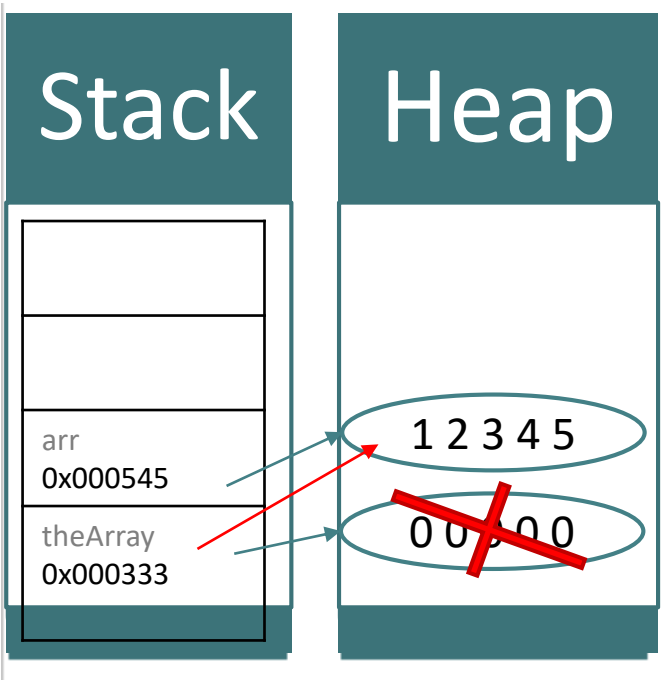
- Array elements are:
- 1 2 3 4 5




```
static void FillArray(out int[] arr)
{
    // Initialize the array:
    arr = new int[5] { 1, 2, 3, 4, 5 };
}

static void Main()
{
    int[] theArray; // Initialization is not required

    // Pass the array to the callee using out:
    FillArray(out theArray);

    // Display the array elements:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Eingangsparameter mit Array

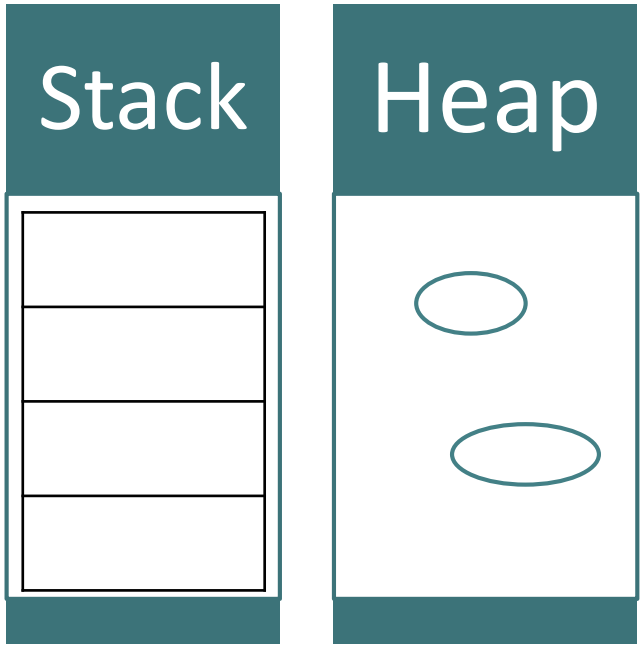
- Array elements are:
- 0 0 0 0 0




```
static void FillArray(int[] arr)
{
    // Initialize the array:
    arr = new int[5] { 1, 2, 3, 4, 5 };
}

static void Main()
{
    int[] theArray = new int[5];  // Initialization is not required

    // Pass the array to the callee using out:
    FillArray(theArray);

    // Display the array elements:
    System.Console.WriteLine("Array elements are:");
    for (int i = 0; i < theArray.Length; i++)
    {
        System.Console.Write(theArray[i] + " ");
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
```

Zusammenfassung

- Speicher ist in Stack und Heap geteilt:
  - Referenztypen landen immer am HEAP (einfache Regel)



  - Wertetypen und Referenzen landen dort wo sie deklariert werden

Mehrdimensionale Arrays

- im Speicher: