Grundlagen:
Heap und Stack
Value Types, Reference Types
Boxing and Unboxing

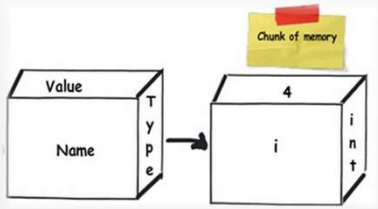
POS 1

Heap and Stack

Was passiert im Speicher, wenn eine Variable erzeugt wird?

```
int i = 4;
```

Diese Variable ist am Stack gespeichert!



Stack - Stapelspeicher

Der Stack ist ein Speicher, der nach dem LIFO – Prinzip organisiert ist.

Das bedeutet: Last In – First Out

Das Element, das zuletzt am Stapel gelegt wurde, muss als erstes wieder vom Stapel genommen werden.

Stack – LIFO – Principle

```
 9   class Program
10   {
11       static void Main(string[] args)
12       {
13           int x = 4;
14           int y = 7;
15
16           {
17               int z = 8;
18           }
19
20           funcA();
21       }
22
23       private static void funcA ()
24       {
25           int g = 9;
26           int h = 12;
27       }
```

Name	Type	Value
x	int	4

Abteilung Informatik
Future Inside

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7

Abteilung Informatik
Future Inside

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
z	int	8

Abteilung Informatik
Future Inside

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
~~z~~	~~int~~	~~8~~

Abteilung Informatik
Future Inside

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
g	int	9

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
g	int	9
h	int	12

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
g	int	9
~~h~~	~~int~~	~~12~~

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7
g	int	9

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

Name	Type	Value
x	int	4
y	int	7

Stack - LIFO – Principle

```
 9  class Program
10  {
11      static void Main(string[] args)
12      {
13          int x = 4;
14          int y = 7;
15
16          {
17              int z = 8;
18          }
19
20          funcA();
21      }
22
23      private static void funcA ()
24      {
25          int g = 9;
26          int h = 12;
27      }
```

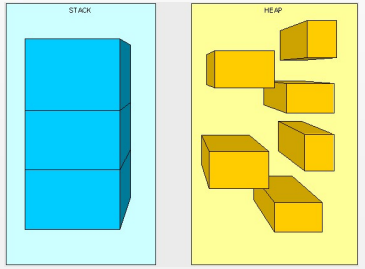
Name	Type	Value
x	int	4

Heap and Stack

Was passiert im Speicher, wenn ein Speicherbereich dynamisch erzeugt wird?

```
int[] myArray = new int[4];
```

Dieser Speicherbereich wird am Heap (Haufen) gespeichert!
Lediglich die Referenz (Zeiger) auf den Speicher wird am Stack gespeichert!

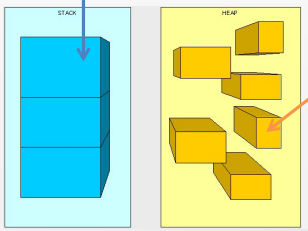


Heap and Stack

```
int[] myArray = new int[4];
```

Name	Type	Value
myArray	int[]	0x0008

Address	Name	Type	Value
...			
0x0008	myArray[0]	int	0
0x000C	myArray[1]	int	0
0x0010	myArray[2]	int	0
0x0014	myArray[3]	int	0
...			



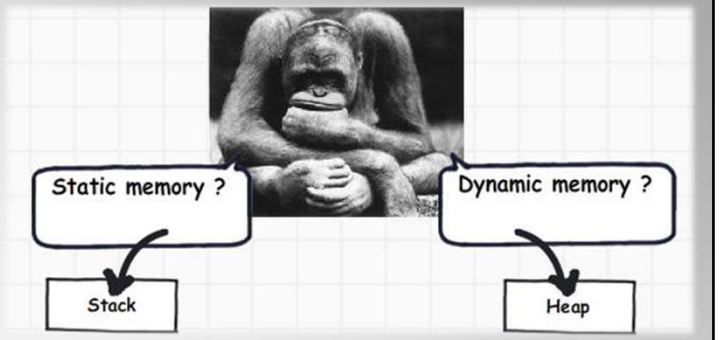
Heap und Stack

Merke:

Dynamisch reservierter Speicherplatz wird am HEAP angelegt. Wenn der Speicher nicht mehr benötigt wird, gibt der Garbage Collector diesen wieder frei!

Statischer Speicher (Werte-Datentypen, ...) werden am Stack angelegt und der Speicher wird nach dem LIFO-Prinzip freigegeben!

Heap und Stack



Value Types and Reference Types

Value Types

- speichern die Werte direkt in der Speicherzelle
- liegen am Stack
- Veränderungen der Werte können nur über diese Variable vorgenommen werden

Reference Types

- speichern die Werte am Heap, nur die Referenz auf die eigentlichen Werte liegen am Stack
- Veränderungen der Werte können über jede Referenz die auf den Speicher zeigt vorgenommen werden!

Value Types (Werte Datentypen)

Klein

Häufig verwendet

Wenig Ressourcen

Kein Heap-Speicher → keine Garbage-Collection

Große Wert-Typen sollten vermieden werden

- *Vor allem im Zusammenhang mit Funktionen (Parameter oder Rückgabeparameter)*

Eigenschaften

- *Wert-Typen werden bei Funktionsaufrufen immer kopiert.*
  Änderungen am Original finden nicht statt.
- *Es muss kein Konstruktor aufgerufen werden.*
- *Werden immer 0 oder null initialisiert.*
- *Um dasselbe Verhalten wie mit Objekten zu erzielen,*
  werden Value-Types oft geboxed (in Objekte eingepackt).

Reference Types (Referenzen)

Unterschiedliche Behandlung im Vergleich zu Wert-Typen

Besteht aus zwei Teilen:

- *Zeiger auf das Objekt im Heap*
- *Objekt im Heap selbst*

Mehr Ressourcen für Verwaltungsaufwand

Hohe Flexibilität

Geschwindigkeitsgewinn

- *Da kein Kopieren sondern nur Zeiger weiterreichen*

CLR (Common Language Runtime) kümmert sich um die Speicher-Allokation und liefert einen Zeiger zum Objekt

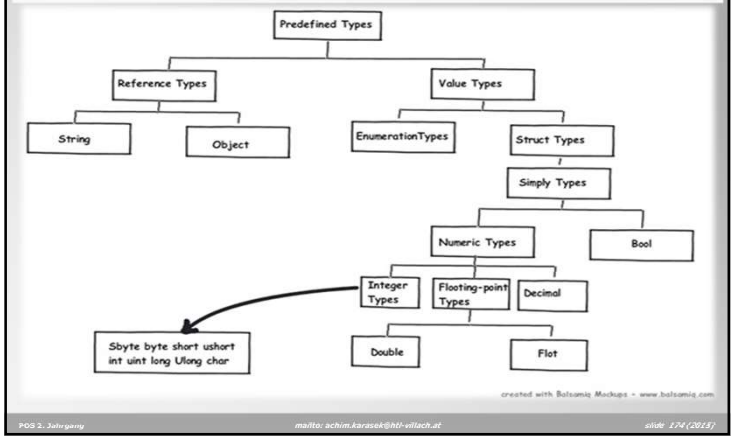
Da kein unendlicher Speicher, müssen nicht mehr benötigte Instanzen gelöscht werden. → Garbage Collector

Direkter Vergleich

eine Variable der Kategorie	Value Type	Reference Types
beinhaltet	direkt die Werte	die Referenz auf die eigentlichen Werte
ist gespeichert am	Stack	Heap
wird initialisiert mit	0, false,	null
macht bei einer Zuweisung	eine Kopie des Wertes	nur eine Kopie der Referenz. Die eigentlichen Werte werden NICHT kopiert

Beispiele:	`int x = 99;`
	`int y = x;`
	x: 99
	y: 99
		`int[] a1 = new int[2] {98, 99};`
		`int[] a2 = a1;`
		a1
		a2
		{ 98, 99 }

Übersicht Einteilung Datentypen (Wiederholung)



Boxing and Unboxing

Fortgeschrittenen Kapitel

wird bei Bedarf erledigt