# Der Stack oder Stapel

… ist eine wichtige Datenstruktur in der Informatik.

Vom "Boden" ausgehend legen wir Objekte oben auf einen Stapel drauf, die Operation "push()",

oder nehmen sie oben vom Stapel herunter unter Rückgabe des Inhaltes: "pop()".

Typisch für eine Stapel ist,

dass jeweils nur oben auf den Stapel etwas hinzugefügt oder weggenommen werden kann.

Die Datenstruktur ist somit eingeschränkter als z.B. eine lineare Liste,

bei der auf alle enthaltenen Elemente zugegriffen werden kann.

Der Inhalt von Stackelementen ist beliebig,

kann z.B. auch ein Objekt sein.

Hier benutzen wir der Einfachheit halber Ganzzahlen.

Beispiel:

leerer Stack:

push (1):

1

push (2):

2

1

pop(): 2 wird zurück gegeben

1

Wir implementieren den Stack auf einem Feld,

schon deshalb, weil wir keine andere Datenstruktur dafür haben.

Frage: Ist deshalb ein Stack gleich einem Feld?

Antwort: Nein! Ein Feld ist eine mögliche Struktur, in dem ich einen Stack speichere.

Eine lineare Liste wäre z.B. eine Alternative.

Vergleich: Ein Regal

Ich baue ein Regal aus Holz, weil das leicht zu verarbeiten ist.

Ist deshalb ein Regal zwangsweise aus Holz?

Nein, es könnte auch aus Metall oder Acrylglas oder Stein bestehen.

Wir benutzen Holz als Material wegen der leichten Verarbeitbarkeit.

Wir benutzen ein Feld zur Implementierung eines Stacks,

weil das einfach ist,

und wir noch keine Alternative haben.

Weil wir ein Feld zu Grund legen, ist damit eine fixe Größe verbunden.

Wir wählen eine bewusst kleine Größe,

um einen Overflow provozieren zu können.

Welche Operationen wollen wir implementieren?

* push(), um ein Element oben auf den Stapel zu legen
* pop(), um das oberste Element des Stapels zu entfernen u. zurück zu geben
* get\_max\_size(), um die maximale Größe des Stapels zu erfragen
* get\_size(), um die aktuelle Größe zu erfragen
* is\_empty() gibt "true" zurück, wenn der Stapel leer ist
* is\_full() liefert "true", wenn er voll ist.
* print\_stack(), um ihn grafisch auszugeben

Diese Methoden implementieren wir in einer abhängigen Klasse "Stack".

In der dazugehörigen Hauptklasse, z.B. "Stack\_Haupt\_1",

soll eine oder mehrere Instanzen eines Stacks erzeugt u. benutzt werden.

Wo orden wir die Elemente an?

class Stack {

// Def Stack

… push(…){…}

… pop (…){…}

… get\_size()…

… get\_max\_size()…

… is\_full()…

… is\_empty()…

… print\_stack()…

}

class Stack\_Haupt\_1{

… main … {

new Stack …

…

push();

…

pop();

}

}

Dem Benutzer soll die Datenstruktur eines Stacks zur Verfügung gestellt werden.

Er soll nur auf einen definierten Satz an Operationen bzw. Methoden zugreifen können.

Die eigentliche Struktur bleibt dem Benutzer verborgen.

Der "naive Benutzer" weiß nicht, was ein Feld ist.

Wir kapseln deshalb alles außer dem, was wir explizit freigeben.

Was brauchen wir für Variablen?

* Zuerst das Feld mit einem beliebigen aber fixen Grundtyp,  
  hier z.B. "int", als Aufgabe "char", für den Benutzer verborgen
* max\_size, z.B. 5, gibt die maximale Stapelgröße bzw. Feldgröße an.  
  Sie soll vom Benutzer zugänglich sein u. wird zur Dimensionierung des Feldes benutzt.
* size, mit 0 initialisiert, gibt die aktuelle Größe an, vor dem Benutzer verborgen.

int

stack:

Index: 0 1 2 3 4

Deklaration des Feldes:

public int max\_size = 5;

private size = 0;

private int[] stack = new int[max\_size];

Wie merken wir uns,

an welcher Stelle ein Element mit "push()" eingefügt oder mit "pop()" entfernt wird?

Das Attribut "size" bekommt hier ein Doppelrolle:

Neben der aktuellen Größe des Stack, ist sie wie ein Zeiger auf den Index,

an dem mit push() ein neues Element eingefügt wird:

int

stack:

Index: 0 1 2 3 4

**size = 0**

Die Befehle push() u. pop():

Beim push()-Befehl sind zwei Aktionen zu tätigen,

wobei es auf die Reihenfolge ankommt:

1. Schreibe das zu pushende Element an die Stelle des Stacks,  
   worauf der size-Zeiger zeigt,  
   beim ersten Einfügen in den leeren Stack an Indexposition 0.
2. Inkrementiere size.

Beim pop()-Befehl umgekehrt:

1. Dekrementiere size
2. Gebe das Elemen zurück,  
   das an der Indexstelle steht,  
   worauf der size-Zeiger zeigt.

Hier sieht man schön den Komplementären Charakter von den beiden Befehlen,

ählich wie bei getter- u. setter-Methoden:

push() greift erst auf den Stack an der Stelle size zu u. verändert den Zeiger dann,

pop() umgekehrt verändert zuerst den Zeiger u. greift dann an der size-Stelle zu.

Weiterer Aspekt des komplementären Charakters:

push() hat einen Parameter, aber keinen Rückgabewert,

pop() umgekehrt hat einen Rückgabewert, aber keine Parameter.

Beispiel:

Auf dem neu angelegten Stack kommt der Befehl "push(17);"

Vorher:

int

stack:

Index: 0 1 2 3 4

**size = 0**

Nachher:

int

stack: 17

Index: 0 1 2 3 4

**size = 1**

Fortsetzung Beispiel:

Nun kommt der Befehl "pop()":

Vorher:

int

stack: 17

Index: 0 1 2 3 4

**size = 1**

Nachher: unter Rückgabe der 17

int

stack: 17

Index: 0 1 2 3 4

**size = 0**