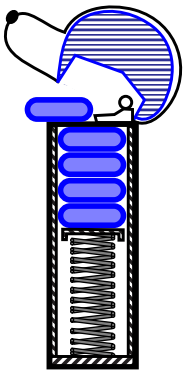
STABLER OG REKURSJON

- Abstrakte Data Typer (ADT)
- Stabler
- Eksempel: Aksjeanalyse
- Unntakshåndtering i Java
- To stabel-implementasjoner
- Rekursjon

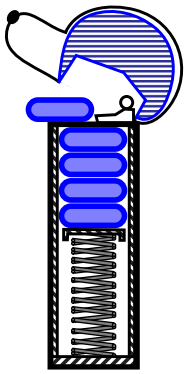


Abstrakte Data Typer (ADT)

- En **Abstrakt Data Type** er en abstraksjon av en data struktur: ingen implementering er involvert.
- ADT spesifiserer:
  - hva kan lagres i denne ADT
  - hvilke operasjoner kan utføres på/av ADT
- Det finnes flere standard ADTer, og i dette kurset vil vi se noen (stabler, køer, trær...)

Stabler

- En stabel er en beholder for objekter som settes inn og taes ut etter last-in-first-out (LIFO) prinsippet.
- Objekter kan innsettes når som helst, men bare det sist innsatte objektet kan taes ut.
- Å sette inn et objekt kalles å "PUSH"e objektet på stabelen. Å ta et objekt ut av en stabel kalles å "POP"e det.
- En PEZ® sukkertøysbeholder (eller en stabel med tallerkener på en kafeteria) er en analogi:

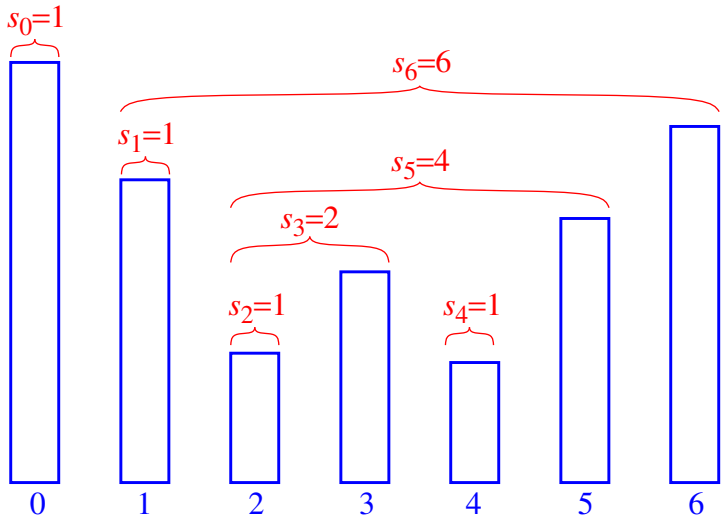


ADT Stabel

- ADT stabel har to hovedmetoder:
  - push(o): Inserts object o onto top of stack
  - pop(): Removes the top object of stack and returns it; if stack is empty an error occurs
- Følgende hjelpemetoder kan også defineres:
  - size(): Returns the number of objects in stack
  - isEmpty(): Return a boolean indicating if stack is empty.
  - top(): return the top object of the stack, without removing it; if the stack is empty an error occurs.

Eksempel

- En aksjes ***spenn (eng span)*** for en gitt dag d, er det høyeste antall påfølgende dager fram til dag d hvor prisen på aksjen ikke oversteg prisen på dag d.



En første Algoritme for spenn

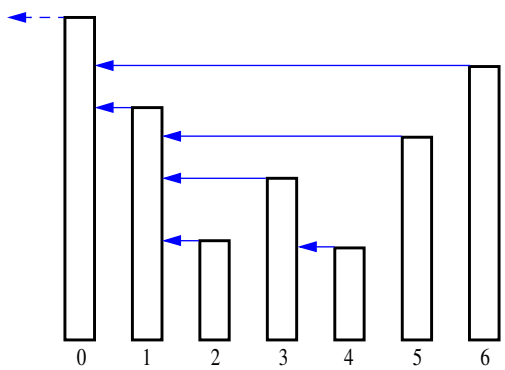
- Vi beregner spennet for hver av n dager:

```
Algorithm computeSpans1(P):
  Input: An n-element array P of numbers
  Output: An n-element array S of numbers such that
          S[i] is the span of the stock on day i.
  Let S be an array of n numbers
  for i=0 to n−1 do
    k ←0
    done←false
    repeat
      if P[i-k] ≤P[i] then
        k←k+1
      else
        done←true
    until (k=i) or done
    S[i]←k
  return array S
```

- Kjøretid er $O(n^2)$. Hvorfor?

Bruk en stabel!

- For en gitt dag i, må vi holde tritt med h_i = 'den nærmeste dagen, forut for dag i, hvor prisen er høyere enn den er for dag i'
- Spennet for dag i blir da ganske enkelt $s_i = i - h_i$



Vi kan bruke en ***stabel*** for å beregne h_i

En bedre algoritme vha stabel:

```
Algorithm computeSpan2(P):
  Input: An n-element array P of numbers
  Output: An n-element array S of numbers such that
          S[i] is the span of the stock on day i.
  Let S be an array of n numbers and D an empty stack
  for i=0 to n-1 do
    done←false
    while not(D.isEmpty() or done) do
      if P[i]≥P[D.top()] then
        D.pop()
      else
        done←true
    if D.isEmpty() then
      h← -1
    else
      h←D.top()
    S[i]←i-h
    D.push(i)
  return array S
```

- Hva med computeSpan2's kjøretid...?

Et grensesnitt for stabler i Java

- Stabel (stack) er en innebygget klasse i Java's java.util pakke, men man kan også definere sin egen stabel:

```
public interface Stack {

  // accessor methods
    public int size(); // return the number of
                       // elements in the stack
    public boolean isEmpty();  // see if the stack
                               // is empty
    public Object top() // return the top element
      throws StackEmptyException; // if called on
                                  // an empty stack

  // update methods
    public void push (Object element); // push an
          // element onto the stack. Note that
          // the type of the parameter is
          // specified as an Object
    public Object pop() // return and remove the
                        // top element of the stack
      throws StackEmptyException; // if called on
                                  // an empty stack
  }
```


Unntak (eng: Exceptions)

- **Unntak** er et nyttig programmeringsbegrep.
- Brukes for håndtering av feil. Feilsituasjoner markeres ved å ***heve (eng: throw)*** et unntak.
- Eksempel

```
public void spisPizza() throws OverSpistException
{
  ...

  if (spistForMye)
    throw new OverSpistException("Raaap");

  ...
}
```

- Såsnart unntaket heves, vil kontrollflyten endres ved at inneværende metode forlates.
- Når OverSpistException heves, forlater vi spisPizza() og fortsetter i den metoden hvor kallet kom fra.

Unntak

- Anta spisPizza() ble kallt som følger:

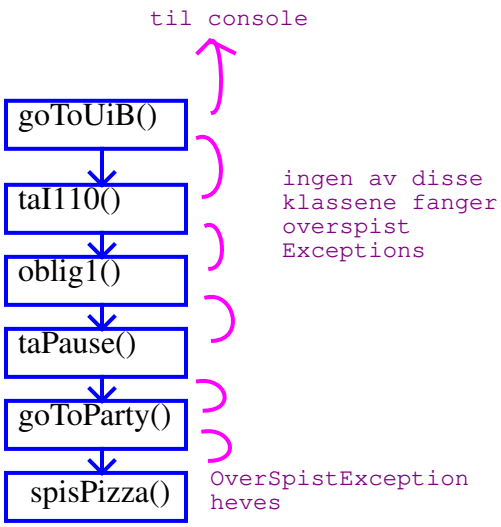
```
private void simulateMeeting()
{
  ...
  try
    {
      graadig.spisPizza();
    }
  catch(OverSpistException e)
    {
      System.out.println("Noen har overspist");
    }
  ...
}
```


Mer om unntak

- Kontrollen returnerer til graadig.spisPizza(); fordi et unntak ble hevet i spisPizza().
- Gjennom programblokkene try og catch venter vi på unntak som spesifisert i catch parameteren.
- Siden catch venter pår OverSpistException, vil kontorllflyten gå til catch og System.out.println vil utføres.
- En catch kan inneholde hva som helst, ikke nødvendigvis System.out.println. Programmereren bestemmer selv hvordan unntaket skal håndteres, unntaket kan også bli hevet videre.
- Hvis en metode kan heve et unntak må den ha throws rett etter metodenavnet.
- Hvorfor bruke unntak? Fordi feilhåndtering da kan foretas på et 'høyere nivå' i programmet, ansvaret sendes tilbake til den som gjorde kallet. Og dette er tydelig i grensesnittet for metoden, gjennom throws.

Mer om unntak

- Om et unntak aldri håndteres, men bare heves videre, vil det propagere oppover i metodekallene inntil brukeren ser det:



Siste om unntak

- Men hva er et unntak i Java? Svar: En klasse.
- Se på OverSpistException.

```
public class OverSpistException extends
  RuntimeException {
  public OverSpistException(String err)
    {
      super(err);
    }
}
```


En tabell-basert stabel

- Lag en stabel vha en array med max-størrelse N f.eks. $N = 1{,}024$, og en peker t til 'øverste' elementet i stabelen.
-

- Array indeks starter på 0, så vi initialiserer t til -1
- Pseudokode

Algorithm size():
  return $t + 1$

Algorithm isEmpty():
  return $(t < 0)$

Algorithm top():
  if isEmpty() **then**
    throw a StackEmptyException
  return $S[t]$
...

Tabell-basert stabel

- Pseudokode

Algorithm push(o):
  if size() = N **then**
    throw a StackFullException
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Algorithm pop():
  if isEmpty() **then**
    throw a StackEmptyException
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ **null**
  $t \leftarrow t - 1$
  return e

- Hver metode har konstant kjøretid $O(1)$.
- Array-implementeringen er enkel og effektiv..
- Men...vi har en øvre grense, N, på stabelstørrelsen. For en gitt applikasjon kan N være for liten, eller kanskje for stor...

Java-kode

```
public class ArrayStack implements Stack {
  // Implementation of the Stack interface
  // using an array.

  public static final int CAPACITY = 1000; // default
                                // capacity of the stack
  private int capacity; // maximum capacity of the
                                // stack.
  private Object S[ ]; // S holds the elements of
                                // the stack
  private int top = -1; // the top element of the
                                // stack.

  public ArrayStack( ) { // Initialize the stack
      this(CAPACITY);// with default capacity
  }

  public ArrayStack(int cap) { // Initialize the
                                // stack with given capacity
      capacity = cap;
      S = new Object[capacity];
  }
```


Tebell-basert stabel i Java

```
public int size( ) { //Return the current stack
                                // size
    return (top + 1);
}
public boolean isEmpty( ) {  // Return true iff
                                // the stack is empty
    return (top < 0);
}

public void push(Object obj) {  // Push a new
                                // object on the stack
    if (size() == capacity) {
      throw new StackFullException("Stack overflow.");
    }
    S[++top] = obj;
}

public Object top( ) // Return the top stack
                                // element
    throws StackEmptyException {
    if (isEmpty( )) {
      throw new StackEmptyException("Stack is
        empty.");
      }
    return S[top];
}
```


Tabell-basert stabel i Java

```
public Object pop() // Pop off the stack element
  throws StackEmptyException {
  Object elem;
    if (isEmpty( )) {
     throw new StackEmptyException("Stack is Empty.");
    }
  elem = S[top];
  S[top--] = null; // Dereference S[top] and
                // decrement top
  return elem;
  }
```


Tabell-basert stabel og utvidelse

- Istedetfor å heve en StackFullException, kan vi bytte ut array S med en større array og fortsette med flere push-operasjoner..

Algorithm push(o):
  if size() = N **then**
    $A \leftarrow$ *new array of length* $f(N)$
  for $i \leftarrow 0$ **to** $N - 1$
    $A[i] \leftarrow S[i]$
  $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

- ***Hvor stor skal den nye array være?***
  - ***konstantvekst-strategi*** : $f(N) = N + c$, c konstant
  - ***doblingsvekst-strategi*** : $f(N) = 2N$

- For å sammenlikne de to strategier, bruker vi følgende kostnadsmodell:

vanlig push operasjon: ett nytt elmt	1
spesiell push operasjon: lag array med $f(N)$ elmt, kopier over N elmt, og legg til ett nytt elmt	$f(N) + N + 1$

Konstantvekst-Strategi (c=4)

- begynner med array størrelse 0
- spesiell push koster $2N + 5$

push	fase	n	N	kost
1	1	0	0	5
2	1	1	4	1
3	1	2	4	1
4	1	3	4	1
5	2	4	4	13
6	2	5	8	1
7	2	6	8	1
8	2	7	8	1
9	3	8	8	21
10	3	9	12	1
11	3	10	12	1
12	3	11	12	1
13	4	12	12	29

Kjøretid for konstantvekst-strategi

- Anta k faser, hvor $k = n/c$, n antall push totalt
- Hver fase tilsvarer en ny array størrelse
- Fase i koster $2ci$
- Totalkostnad av n push operasjoner er totalkostnad for k faser, hvor $k = n/c$:

  $2c\ (1 + 2 + 3 + ... + k)$,

  som er $O(k^2)$ og $\mathbf{O(n^2)}$.

Doblingsvekst-strategi

- begynn med array størrelse 0, så 1, 2, 4, 8, ...
- spesiell push koster nå $3N + 1$ for $N > 0$

push	fase	n	N	kost
1	0	0	0	2
2	1	1	1	4
3	2	2	2	7
4	2	3	4	1
5	3	4	4	13
6	3	5	8	1
7	3	6	8	1
8	3	7	8	1
9	4	8	8	25
10	4	9	16	1
11	4	10	16	1
12	4	11	16	1
...	...	...	...	...
16	4	15	16	1
17	5	16	16	49

Kjøretid for doblingsvekst-strategi

- Anta k faser, hvor $k = \log n$, n antall push totalt
- Hver fase tilsvarer en ny array størrelse
- Fase i koster 2^{i+1}
- Totalkostnad for n push operasjoner er totalkostnad for k faser, hvor $k = \log n$

  $2 + 4 + 8 + ... + 2^{\log n + 1} =$

  $2n + n + n/2 + n/4 + ... + 8 + 4 + 2 = 4n - 1$

- Dette er O(n). Doblingsvekst-strategien er best!

Støping (eng: Casting) med en generisk stabel

- Bruk ArrayStack som lagrer Integer objekter.
- For å bruke en generisk stabel, må return objektet støpes om til den korrekte data type.
- Java kode eksempel (reverser en tabell av integer):

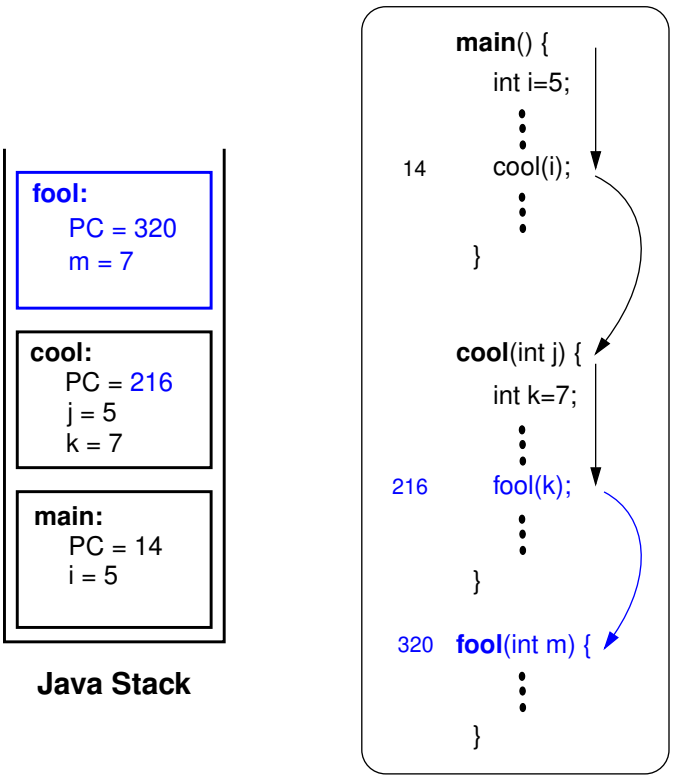
```
public static Integer[] reverse(Integer[] a) {
  ArrayStack S = new ArrayStack(a.length);
  Integer[] b = new Integer[a.length];
  for (int i = 0; i < a.length; i++)
    S.push(a[i]);
  for (int i = 0; i < a.length; i++)
    b[i] = (Integer)(S.pop()); // the popping
      // operation gave us an Object, and we
      // casted it to an Integer before
      // assigning it to b[i].
  return b;
}
```


Stabler i Java Virtual Machine

- Hver prossess i et Java program har sin egen Java Method Stack.
- Hver gang en metode kalles, push'es informasjon om metoden (eng: frame) på stabelen.
- Ved å bruke en stabel kan Java oppnå:
  - Utføring av rekursive metodekall
  - Skriv ut stabel 'trace' for å oppdage feil
- Java har også en operand-stabel som brukes til å evaluere aritmetiske uttrykk:
- Mer om dette i kap.5. Veldig forenklet:

```
Integer add(a, b):
  OperandStack Op
  Op.push(a)
  Op.push(b)
  temp1 ← Op.pop()
  temp2 ← Op.pop()
  Op.push(temp1 + temp2)
  return Op.pop()
```


Java Method Stack



Rekursjon

- Metode kaller seg selv som subrutine.
- n! = 1*2*3*...*(n-1)*n (eng: n factorial)
- Defineres induktivt ved 1!=1 og n!=n*(n-1)!, n>1.

```
Algorithm fact(n);
    Input: integer n>0
    Output:n!
  if (n <= 1) return 1;
  else return n*fact(n-1);
```

- Metoden fact kaller seg selv rekursivt med ny parameterverdi n-1. Når dette rekursive kallet returnerer med (n-1)! vil denne verdien multipliseres med n og gi svaret n!.
- Metodens stabel vil inneholde n kall, fact(n)-fact(n-1)-...fact(3)-fact(2)-fact(1), og det siste kallet fact(1) vil returnere 1.
- fact(n) vil alltid terminere. Hvorfor? Kjøretid?
- Hva med

```
Algorithm hack(n);
  if (n <= 1) return 1;
  else return n*hack(n+1);
```