Abstrakt programmering

. . . ok, i JAVA, dvs. i-110 med mer struktur

I. Typer, Data Typer og Abstrakte Data Typer

I.1 ADT = grensesnitt (Type) : synlig **Hva**

Interface

I.2 DT = grensesnitt + implementasjon på en DS

Class : Hva + Hvordan

I.3 DS = intern organisering av data : usynlig **Hvordan**

I.4 Bruk, Dokumentasjon og Implementasjon av ADT

I.5 Package = en samling av tilhørende ADT

II. Arv og ... Sterk Typing

II.1 Arv, overskriving, overlasting og polymorfisme

Synlighet: private, public, protected

enkel arv

II.2 Abstract class = DT med delvis implementasjon

II.3 Interface

multipel typing (arv av grensesnitt)

III. Bruk og ... Tilpassing

III.1 import, pakker, klassehierarki – og Javadoc

III.2 Arv ...

III.3 Casting

III.4 Exceptions

En Type

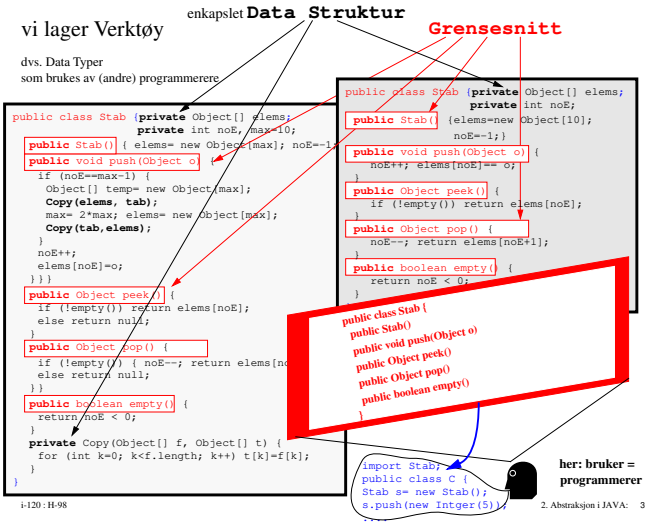
```
boolean, char, String, int, double

boolean a, b;      a && b   !b   false||b
int x, y;          x + y    x/y
String s, t;       s.equals(t)  s+'c'

            !!   s.equals(a)   t+x
```

Primitive DataTyper (i et programmeringsspråk) er implementert (Hvordan?) men det eneste viktige for programmerer er Hva han kan gjøre med dem

```
PUBLIC CLASS STAB {public Object[] elems; int noE;
          public Stab() {elems= new Object[10]; noE=-1;}
          public void push(Object o) {
            noE++; elems[noE]= o;
          }
          public Object peek() {
            if (!empty()) return elems[noE];
            else return null;
          }
          public Object pop() {
            noE--; return elems[noE+1];
          }
          public Object empty() {
            return noE < 0;
          }
}
```

introduserer en **ny DataType – utvider språket**

```
Stab p,r;   p=new Stab();   p.push(new Integer(5))
            r=new Stab();
        !!  p.equals(r);    p.push(5)
            r.noE= 5;       if (!r.empty()) ...
```

Data Type =

boolean, char, Set, ...

Grensesnitt

operasjoner tilgjengelig for bruker
som er implementert på en bestemt

+ Data Struktur

privat for Data Typen, dvs. skjult for bruker
– han kan forandre på tilstanden i DS kun v.hj.a. grensesnitt-operasjoner

NB! Bruker er interesert i – importerer *– kun grensesnittet*

Interface = en "Abstrakt" Data Type

kun deklarasjoner av grensesnitt operasjoner – med dokumentasjon !!!

```
/** LIFO kø av vilkaarlige Objekter */
public interface Stack {   // ingen konstruktører
/** legger nye Objekter på toppen av stabel
 *  @param o Objektet som skal settes inn */
    public void push(Object o);
/** fjerner top (siste) Objektet fra stabel
 *  @return top Objektet – null hvis empty() */
    public Object pop();
/** returnerer (uten å fjerne) top Objektet fra stabel
 *  @return top Objektet – null hvis empty() */
    public Object peek();
/** @return true hviss stabel er tom */
    public boolean empty();
}
```

importeres *og programmeres med (brukes) som alle andre Data Typer*

. . . men :

har ingen konstruktører – nye objekter kan ikke opprettes

skal nye objekter opprettes, må man bruke en implementasjon av interface

Javadoc

```
/**   LIFO kø av vilkaarlige Objekter
 *    første Objektet er det som ble innsatt sist
 *    @author      Michal Walicki
 *    @version     1.2, Aug 19 1998
 *    @see         Stack
 */
public class Stab implements Stack {

/**   legger nye Objekter på toppen av stabel
 *    @param o     Objektet som skal settes inn
 */
      public void push(Object o);

/**   fjerner top (siste) Objektet fra stabel
 *    @return      top Objektet – null hvis empty()
 */
      public Object pop();

/**   returnerer (uten å fjerne) top Objektet fra stabel
 *    @return      top Objektet
 *    @exception NullPointerException kastes hvis empty()
 */
      public Object peek();

/**   @return      true hviss stabel er tom
 */
      public boolean empty();
}
```

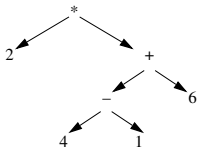
> javadoc Stab.java

Bruk av Interface

2 * ((4 – 1) + 6)

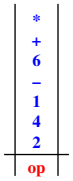
*** 2 + – 4 1 6** "polsk notasjon"
* (2, +(–(4,1), 6))

2 4 1 – 6 + * "omvendt polsk notasjon"



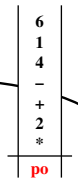
Evaluer et aritmetisk uttrykk gitt i omvendt polsk notasjon

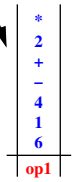
```
import Stack;
int Opolish(Stack op) {
    o = op.pop();
    if (o er et tall) return o;
    else if (o er *) {
        a1= Opolish(op); a2= Opolish(op);
        return a1 * a2;
    } else if (o er -) {
        a1= Opolish(op); a2= Opolish(op);
        return a2 – a1;
    } .... }
```



Les et aritmetisk uttrykk i polsk notasjon og evaluer

```
/*  Les fra venstre og
 *    push hvert symbol på stabel po

 *  Reverser stabel:
 *    while (!po.empty())
 *        op1.push(po.pop());
 *  !! får omvendt rekkefølge !

 *  return Opolish(op1);
 *    tilpasset, f.eks:
 *    if (o er –) {    a1= Opolish(op1);
 *                     a2= Opolish(op1);
 *                     return a1 – a2;     }
 */
```

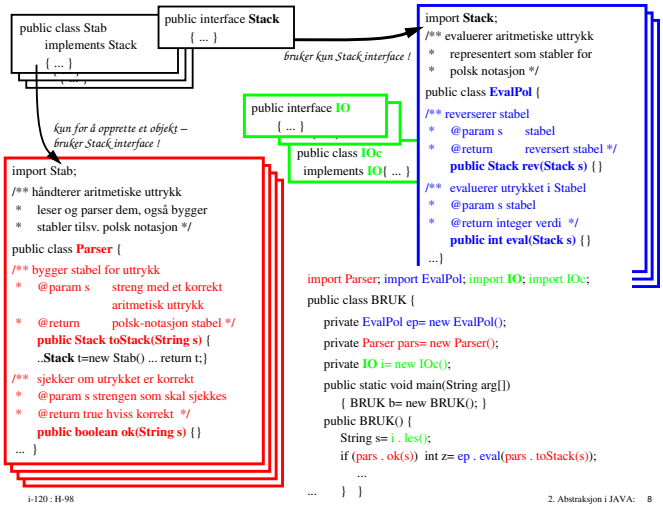




Implementasjon av Interface

en Stabel Data Type

```
public class Stab IMPLEMENTS STACK {
    private Object[] elems;
    private int noE, max=10;

    public Stab() { elems= new Object[max]; noE= -1; }
    public Object peek() {
        if (empty()) return null;
        else return elems[noE];
    }
    public void push(Object o) {
        if (noE=max-1) {
                Object[] temp= new Object[max];
                Copy(elems, tab);
                max= 2*max; elems= new Object[max];
                Copy(tab,elems);  }
        noE++;
        elems[noE]= o;
    }
    public Object pop() {
        if (empty() return null;
        else { noE--; return elems[noE+1]; }
    }
    public boolean empty() { return noE < 0;
    }
/** kopierer innhold av array fra til array til
    @param PRE: fra.lenght <= til.length */
    private Copy(Object[] fra, Object[] til) {
        for (int k=0; k<fra.length; k++)  til[k]= fra[k]; }
}
```

Package

- *En ADT (interface) samler noen relaterte funksjoner*
- *Dens implementasjon implementerer disse på en valgt Data Struktur*
- *Flere ADT'er og DT'er vil ofte utgjøre en helhet – en pakke – for håndtering av spesielt sett av problemer.*

1. **Opprett i din hjemmekatalog en underkatalog og kall den, f.eks. 'Pakker' – alle dine pakker skal ligge i denne katalogen**
2. **I Unix, kjør kommando**
   ' > setenv CLASSPATH $HOME/Pakker : /usr/java/lib '
   (dette kan legges til din .cshrc filen)
3. **Når du lager en ny pakke, si, niceIO**
   - **opprett katalog 'niceIO' i katalogen 'Pakker'**
   - **alt som hører til 'niceIO'-pakken skal legges i katalogen**
     ' $HOME/Pakker/niceIO/ '
   - **enhver fil 'klass.java' i denne katalogen skal starte med**
     package niceIO ;
4. **For å bruke en pakke, niceIO, i en klass, skriv**
   import niceIO.* ;
   på toppen – før public class klass { ... i filen 'klass.java'

For å bruke bare en spesifikk klasse, terminalIO, fra denne pakken:
import niceIO.terminalIO ;

5. **F.eks. java.awt er en pakke for grafisk brukergrensesnitt;**
   - **alle dens klasser ligger i katalogen ' /usr/java/src/java/awt '**
   - **du bruker den/importerer ved å starte din 'klass.java' med**
     import java.awt.* ;

Klasse-synlighet

Klasse A deklarert:	`public`	*default / pakke
i samme pakken	Ja	Ja
i andre pakker	Ja	Nei

* Ingen adgangsmodifikator for klassen.

- En `public`-klasse er tilgjengelig i alle andre pakker, dvs er synlig der pakken den er deklarert i er synlig.
  - Høyst én `public`-klasse pr. fil.
- En klasse *uten* `public`-synlighetsmodifikator er kun synlig i den pakken den er deklarert i, dvs klassen har `package`-synlighet.

```
package niceIO;
public class Allmen { // synlig i andre pakker der denne pakken er synlig.
   // …
}
class Gradert { // kun synlig i pakken den er definert i.
   // …
}
```


Synlighet av medlemmer: *variabler* og *metoder*

Klasse A, medlemmer deklarert : og forsøkt brukt i:		`public`	*default / pakke	`protected`	`private`
§samme pakken	klasse A	Ja	Ja	Ja	Ja
	subklasse B	Ja	Ja	Ja	Nei
	ikke-subklasse C	Ja	Ja	Ja	Nei
andre pakker	subklasse D	Ja	Nei	‡Ja	Nei
	ikke-subklasse E	Ja	Nei	Nei	Nei

* ingen adgangsmodifikator

‡ Subklassen kan ikke aksessere `protected` medlemmer i instanser av superklassen, bare i instanser av seg selv og sine subklasser.

§ ingen pakke-navn medfører *"default" pakken* som vanligvis er innværende katalog i filsystem-hierarkiet.

```
package niceIO;
public class Allmen {              // synlig der niceIO er synlig.
   public          Allmen() {..}   // alle kan opprette nye instanser
   protected int   minMax;         // jeg har minMax og mine barn har sine
   private   void decMax() {..}    // bare jeg kan se og minke Max
   public    int   Max();          // men alle kan se dens verdi
   protected void incMax() {..}    // jeg kan øke minMax,
}                                  // og mine barn kan øke sine
```


Oppsummering av I. ADT

Vi programmerer ADT'er

- moduler som samler noen relaterte funksjoner
- et endelig program er bare en sammensetting av forskjellige moduler

En modul brukes kun gjennom grensesnitt

- en modul skiller skarpt mellom grensesnitt og implementasjon (intern Data Struktur, valgte algoritmer og deres implementasjon)
- modulens Data Struktur skal være private – skjult for brukeren
- enhver modul burde implementere en designert interface
- kun moduler som oppretter nye objekter av en Interface bruker konkret Data Type (class) som implementerer en interface

  andre burde bruke minst mulig konkrete Data Typer og mest mulig ADT (interface)

II. OO – Arv

synlighet	binding	type	hvem	
Grensesnitt				
public			**interface**	alle
Klasser som bruker-definerte typer				
public	.....		**class**	alle
———	.....		**class**	i samme pakken
Klasse-medlemmer				
public	.....	**void**	method	alle
private	.....	int	meth/var	kun jeg
———	.....	char	meth/var	i samme pakke
protected	.....	String	meth/var	jeg og subklasser – også i andre pakker
	Abstrakte klasser			
.....	**abstract**	.....	**class/meth**	kan ikke instansieres – opprettes nye Objekter
				har klasse en abstract-method må den selv være abstract
.....	**final**	.....	**class/meth**	kan ikke ha barn – ingen subklasser
				en ikke-final klasse kan ha final-method
.....	**native**	.....	method	skrevet i et annet programmeringsspråk

Abstract class

= en mellomting mellom en interface og en super-klasse

– innfører en ny type (som inteface eller class)
– kan ha Data Struktur og implementasjon av (noen) metoder
– har minst en abstrakt metode
– kan ikke instansieres, selv om kan definere konstruktører

```
public interface Srt{
    int[] sort() ;
    void setA(int[] X) ;
}
```

	public grensesnitt	data struktur	implementerer metoder	privt/protectd medlemer	har konstruktr	kan instansieres
interface	+	kun **final public** variable	–	–	–	–
abstract class	+	kan ha **så mye du vil**	minst 1 **abstract** **– / + ellers**	+	+ / –	–
super-class	+	+	+	+	+	+

```
public abstract class Srt {
    protected int[] A;
    protected void swap(int i, int j) {
      if (i < A.length && j < A.length)
        {int k= A[i]; A[i]=A[j]; A[j]=k;} }
    public void setA(int[] X) { ... }
    public abstract int[] sort();
```

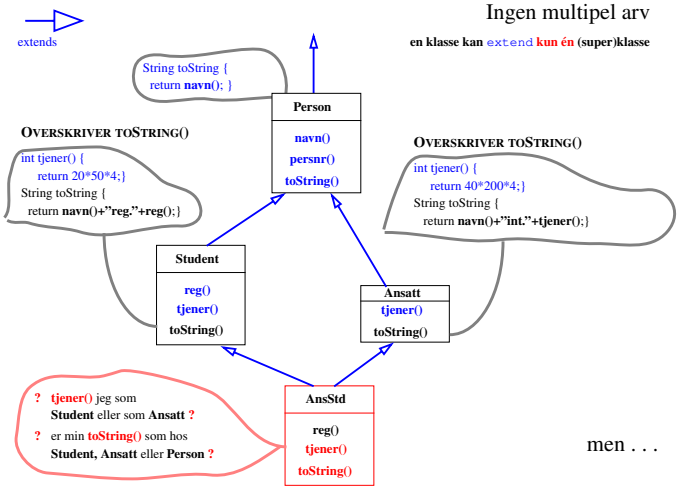
tvinger implementasjon i enhver – ikke-abstrakt – subklasse

```
public class Srt {
    protected int[] A;
    public Srt() { ... }
    protected void swap(int i, int j) { }
    public void setA(int[] X) { ... }
    public int[] sort() { return A; }
... }
```

en "dum" subklasse – og enhver instans – kan bruke denne

Ingen multipel arv

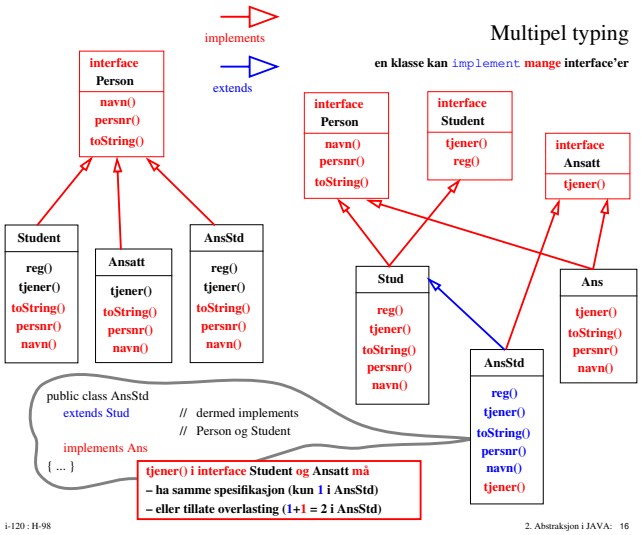
en klasse kan `extend` kun én (super)klasse



men . . .

Multipel typing

en klasse kan `implement` mange interface'er



Overskriving vs. overlasting

Overskriving (overriding) **:**

samme navn, samme parametre

en metode fra superklasse skrives om – på nytt

```
public class Super {
    public int tall() { return 100; }
}

public class Sub extends Super {
    public int tall() { return 50; }
    public int tallS() { return super.tall(); }
}
```

Overlasting (overloading) **:**

samme navn, forskjellige parametre

samme metodenavn brukes igjen – med forskjellige parametre

```
public class Ov {
    public int tall() { return 50; }
    public int tall(int k) { return k+1; }
/*
 *  public char tall() {...} er ulovlig !!
 *  men
 *  public char tall(int k, char c) {...} er ok
 */
}
```

(sub- eller superklasser av parametere i metoden som overlastes kan brukes)

'Dynamisk binding – Statisk overlasting'

```
public class Point{
  public void hei(){...}                          // 1h
  public boolean equal(Point x){...}              // 1e
}

public class ColorPoint extends Point{
  public void hei(){...}                          // 2h
  public boolean equal(ColorPoint x){...}         // 2e
}

Point p1 = new Point();
Point p2 = new ColorPoint();
ColorPoint cp = new ColorPoint();
```

A. p1.hei(); // 1h
B. p2.hei(); // 2h
C. cp.hei(); // 2h

Overskrevne metoder bindes dynamisk (run-time)

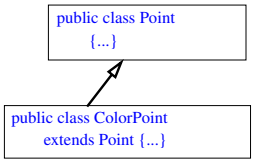
1. p1.equal(p1);
2. p1.equal(p2);
3. p2.equal(p1);
4. p2.equal(p2);
5. cp.equal(p1);
6. cp.equal(p2);
7. p1.equal(cp);
8. p2.equal(cp);
9. cp.equal(cp); // 2e

Overlastede metoder bindes statisk (compile-time)

Arv: oppsummering

- *tillatter å samle "felles" egenskaper i en "abstrakt" superklasse*
- *og dermed designe mer abstrakte programmer som*
  - *avhenger kun av de relevante, abstrakte egeneskaper*
  - *kan brukes på objekter av nye, spesifikke subklasser*

```
public class mittProgram {
    ...
    void proc(Point p) {...}
}
```



Dog:

- *disse kvalitetene sikres utelukkende gjennom type-arv (grensesnitt)*
- *og kan ivaretaes også ved bruk av* `interface`
  (`implements` *er ren arv av type*)

OO-Arv

- *innebærer i tillegg implementasjon-arv av operasjoner*
- *og dermed må håndere ting som overskriving/overlasting*
- *og forbyr multipel arv*

(`extends` *er arv av implementasjon og type*)

III. Bruk og tilpassing

Bruk

- importer kun de pakker/klasser som du trenger
- vær obs på klassehierarki

  Javadoc viser bare metoder deklarert i klassen

  – dermed ikke fullstendig grensesnitt for klassen

  – men viser hvilke klasser denne arver fra

Gjenbruk og Tilpassing

- Arv, overlasting, overskriving ...
- "Parametrisering" – Kast
- Unntak

Kast

– en sikringsmekanisme

```
/** LIFO kø av vilkaarlige Objekter */
public interface Stack {
    void push(Object o);
    Object pop();
    Object peek();
    boolean empty();
}
```

"parameter"-type
"tillater" å lage
Stack[String]
Stack[Integer] ...

*
+
6
–
1
4
2
op

Leser (fra terminal) : **2 4 1 – 6 + *** og **push**'er på stabel

Leser hva ? Si en **String**, som er passelig avgrenset, dvs.

etterfølgende kall til **lesToken()** vil returnere: **"2", "4", "1", "–", "6", "+", "*"**

```
/*  while (mer) {
        String s= lesToken();
        op.push(s);   }
 */

import Stack;
int Opolish(Stack op) {
    o = op.pop();
    if (o er et tall) return o;
    else if (o er *) {
        a1= Opolish(op));
        a2= Opolish(op);
        return a1 * a2;
    } else  .... }
```

```
import Stack;
int Opolish(Stack op) {
    String  o =    (String)    op.pop();
    if (parseInt(o)) return toInt(o);
    else if (o.equals("*")) {
        a1= Opolish(op);
        a2= Opolish(op);
        return a1 * a2;
    } else  .... }
```

Vet du ikke hvilken klasse Objekter fra stabelen tilhører kan du bruke

```
if (o instanceof String) ...
else if (o instancof Klasse) ...
```


Tilpassing

(adapter klasser)

```
public interface Stack
{   void push(Object o);
    Object pop();
    Object peek();
    boolean empty();
}

public class Stab implements Stack
{   void push(Object o) {...}
    Object pop() {...}
    Object peek() {...}
    boolean empty() {...}
...}
```

Men jeg vil nå bare ha en Stabel med String.....

```
public class StringStab extends Stab
{   void sPush(String o) { push(o); }
    String sPop() {return (String) pop(); }
    String sPeek() { return (String) pop(); }
}
```


Robusthet

Feiltoleranse: evnen til å motstå angrep fra ytre og indre fiender.

Ytre fiender (brukere og sluttbrukere) :

- brukere med "fingertrøbbel"
- feil bruk av parameterlister

kan oppdages ved å kontrollere inndata

- er de gyldige (oppfyller forkrav, invarianter) ?
- er de rimelige (selv urimelige data må kunne godtas)

Indre fiender (interne i programmet) :

- feil som medfører avbrudd
- feil i programkode som medfører subtile feil i resultatene
  (brudd på invarianter, uinitierte variable, evige løkker, avrundingsfeil, typekonverteringsfeil)

Robusthet økes ved defensiv programmerig:

- legg inn tester for å oppdage feil
- gi melding om hva feilen er og hvor den oppsto
- prøv å korrigere for feil slik at programmet kan fullføre med nyttige resultater
- dø ærerikt

Ved modulær (abstrakt) programmering:

- enten reager til feil så snart den oppdages
- eller signaliser til en som kaller metoden der feilen oppdages

Unntak (Exception)

For systematisk og modulær feilhåndtering

```
public class Stab IMPLEMENTS STACK {
    private Object[] elems;
    private int noE, max=10;
    ...
    public Object peek() {
        if (empty()) return null;
        else return elems[noE];
    }
    public Object pop() {
        if (empty()) return null;
        else { noE--; return elems[noE+1]; }
    }
...}
```

bruker av Stab-klassen må kjenne til alle 'spesielle objekter' som kan returneres i feilsituasjoner !
Disse er ikke beskrevet i interface !

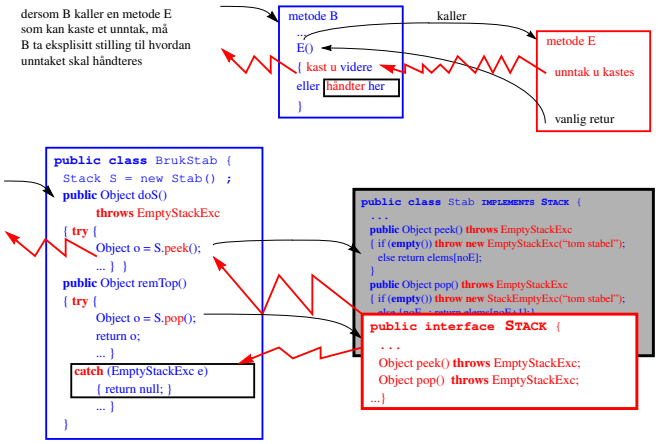
istedenfor slike 'spesielle objekter', markerer man uvanlige/feil situasjoner ved å kaste unntak

```
public interface STACK {
  ...
  Object peek() throws EmptyStackExc;
  Object pop()  throws EmptyStackExc;
...}
```

```
public class Stab IMPLEMENTS STACK {
  ...
  public Object peek() throws EmptyStackExc
  { if (empty()) throw new EmptyStackExc("tom stabel");
    else return elems[noE];
  }
  public Object pop() throws EmptyStackExc
  { if (empty()) throw new EmptyStackExc("tom stabel ved pop");
    else {noE--; return elems[noE+1];}
  }
...}
```

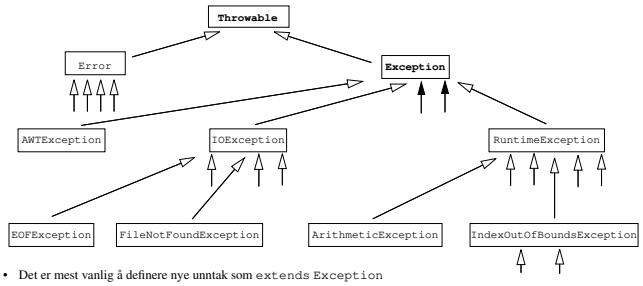
Unntakshåndtering

dersom B kaller en metode E
som kan kaste et unntak, må
B ta eksplisitt stilling til hvordan
unntaket skal håndteres



Unntaks-hierarki

- Unntak er objekter av klasser utledet fra klassen `Throwable`
- Et unntak som ikke `catch`'es må deklareres i `throw`-klausul



- Det er mest vanlig å definere nye unntak som `extends Exception`

```
public class EmptyStackExc extends Exception {
    public EmptyStackExc() { super("Tom Stabel!") ; }
    public EmptyStackExc(String s) { super(s); }
}
```