# ADT og OO programmering

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

## I. ADT I JAVA - INTERFACE

I.1 grensesnitt skal dokumenters – Javadoc

I.2 bruk av interface

I.3 implementasjoner av interface

## II. OO

II.1 pakker og synlighet

II.2 Arv av type og implementasjon

II.3 Abstrakte klasser i Java

## III. BRUK OG ... TILPASSING

III.1 Arv ...

III.2 Casting

III.3 Exceptions



# Javadoc

```
/** LIFO kø av vilkaarlige Objekter – første Objektet er det som ble innsatt sist
 *  @author  Michal Walicki
 *  @version  1.2, Aug 19 1998
 */
public interface Stack {
/** legger nye Objekter på toppen av stabel
 *  @param o Objektet som skal settes inn */
    public void push(Object o);

/** fjerner top Objektet fra stabel
 *  @return  top (=sist innsatte) Objektet (?null hvis empty())
 *  @exception NullPointerException  hvis empty() */
    public Object pop();

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

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

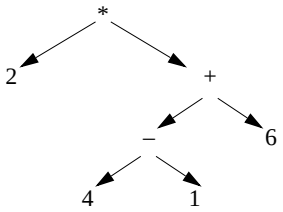
*forbetingelse (pre-condition)*
*– krav til parametre*

*bakbetingelse (post-condition)*
*– beskrivelse av resultatet*

**> javadoc Stack.java**

# Bruk av *Interface*

**2 * ( (4 – 1) + 6 )**

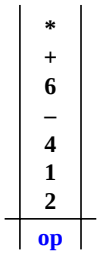



*** 2 + – 4 1 6** "polsk notasjon"
* ( 2, +( –(4,1), 6 )) .

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

*Evaluer et aritmetisk uttrykk lagret i en stabel i **omvendt polsk notasjon***

```
import Stack;
int Opolish(Stack op) {
    Object 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 a1 – a2;
    } else ...
.... }
```

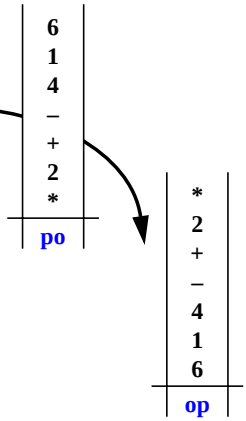



*Les et aritmetisk uttrykk i **polsk notasjon** og evaluer*

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

*  Reverser stabel:
*     Stack op = po.newContainer();
*     while (!po.empty())
*        op.push(po.pop());

*  return Opolish(op);
*/
```






# *Interface* = en "Abstrakt" Data Type

***kun deklarasjoner av grensesnitt operasjoner – med dokumentasjon !!!***

*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*
- *med mindre interface tilbyr en metode som* newContainer *for "kloning" av tilgjengelige instanser*

## DataType =

boolean, char, Stack, ...

Grensesnitt

- *operasjoner tilgjengelig for bruker*
- *som er implementert på en bestemt*

+ DataStruktur

- *som skal være privat for DataTypen, 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*

# En implementasjon av *interface Stack* (en Stack DataType)

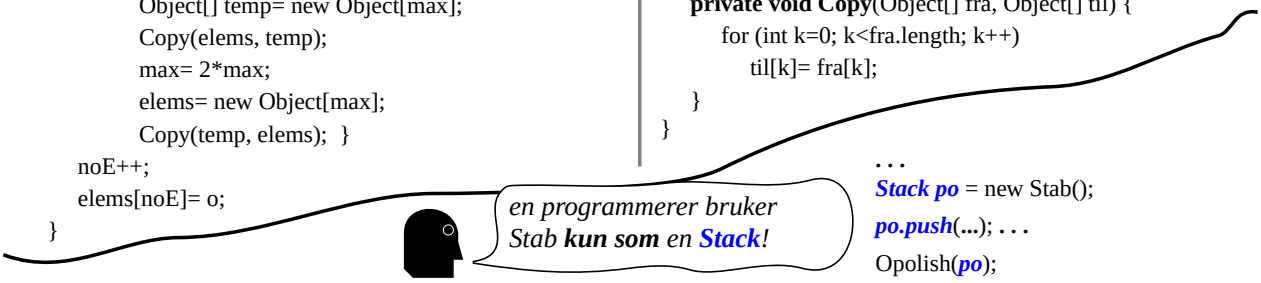
```
public class Stab implements Stack {
    private Object[] elems;
    private int noE, max=10;

    public Stab() {
        elems= new Object[max]; noE= -1; }

    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom stabel");
        else return elems[noE];
    }
    public void push(Object o) {
        if (noE=max-1) {
                Object[] temp= new Object[max];
                Copy(elems, temp);
                max= 2*max;
                elems= new Object[max];
                Copy(temp, elems);  }
        noE++;
        elems[noE]= o;
    }
    public Object pop() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom!!");
        else {
            noE--;
            return elems[noE+1]; }
    }
    public boolean empty() {
        return noE < 0; }

    public Stack newContainer() {
        return new Stab(); }

/** @param fra.lenght <= til.length */
    private void Copy(Object[] fra, Object[] til) {
        for (int k=0; k<fra.length; k++)
            til[k]= fra[k];
    }
}
```




```
. . .
Stack po = new Stab();
po.push(...); . . .
Opolish(po);
```



# En annen implementasjon av *interface Stack* ?

```
public class Koe implements Stack {
    private Object[] elems;
    private int noE, max=10;

    public Koe() {
        elems= new Object[max]; noE= -1; }

    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom koe");
        else return elems[0];
    }
    public void push(Object o) {
        if (noE=max-1) {
                Object[] temp= new Object[max];
                Copy(elems, temp);
                max= 2*max;
                elems= new Object[max];
                Copy(temp, elems);  }
        noE++;
        elems[noE]= o;
}
```

```
    public Object pop() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom!!");
        else {  Object o = elems[0];
            flytt();
            noE--;
            return o; }
    }
    public boolean empty() { return noE < 0; }

    public Koe newContainer() {
        return new Koe(); }

/** @param fra.lenght <= til.length */
    private void Copy(Object[] fra, Object[] til) {
        for (int k=0; k<fra.length; k++)
            til[k]= fra[k];
    }
    private void flytt() {
        for (int k=1; k<= noE; k++)
            elems[k-1] = elems[k];
    }
}
```

# En *LIFO/FIFO ADT*

```
public interface LiFi {
/** legger nye Objekter inn i LiFi
 *  @param o Objektet som skal settes inn */
    void add(Object o);

/** fjerner et Objektet fra  LiFi
 *  @return     Objektet = peek()
 *  @exception NullPointerException  hvis empty() */
    Object remove();

/** returnerer (uten å fjerne) et Objekt fra LiFi
 *  @return     Objektet  som remove() ville fjernet
 *  @exception NullPointerException  hvis empty() */
    Object peek();

/** @return     true hviss LiFi er tom */
    boolean empty();

/** @return     et nytt LiFi Objekt */
    LiFi newContainer();
}
```

```
LiFi copy(LiFi inn) {
    LiFi ut = inn.newContainer();
    while (!inn.empty())
       ut.add(inn.remove());
    return ut;
}

LiFi st = new Stab();
LiFi ko = new Koe();
...
LiFi ko1 = copy(ko);  // kopierer ko
LiFi st1 = copy(st);   // reverserer st
```

```
public interface Stack extends LiFi {
/*   LIFO subtype : add(o).peek() = o  */   }
```

```
public interface Queue extends LiFi {
/*   FIFO subtype : if empty() : add(o).peek() = o
                    else add(o).peek() = peek()   */  }
```



# Oppsummering av 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 DataStruktur, valgte algoritmer og deres implementasjon)*

*– modulens DataStruktur skal være private – skjult for brukeren*

*– ofte vil en modul implementere et designert interface*

*– kun moduler som oppretter nye objekter av et Interface  bruker konkret DataType (class) som implementerer interfacet*

*– andre burde bruke minst mulig konkrete DataTyper og mest mulig ADT (interface)*

# II. OO – Arv (implementasjon av ADTer)

| 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 |



# 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.

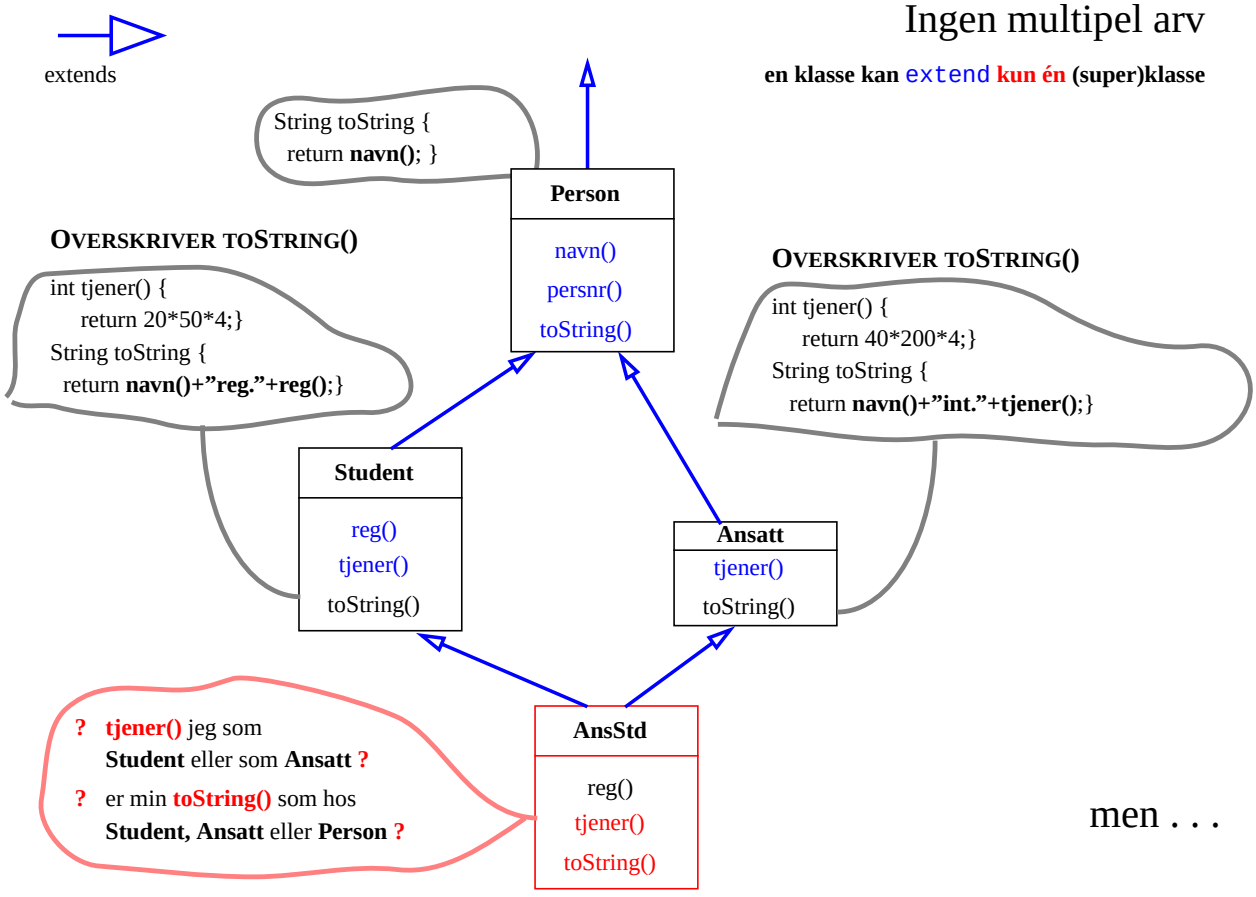
§ ingen pakke-navn medfører *"default" pakken* som vanligvis er innværende katalog i filsystem-hierarkiet.

```
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 minke Max
   public    int  Max();        // men alle kan se dens verdi
   protected void incMax() {..} // jeg kan øke minMax,
}                                // og mine barn kan øke sine
```



## Ingen multipel arv

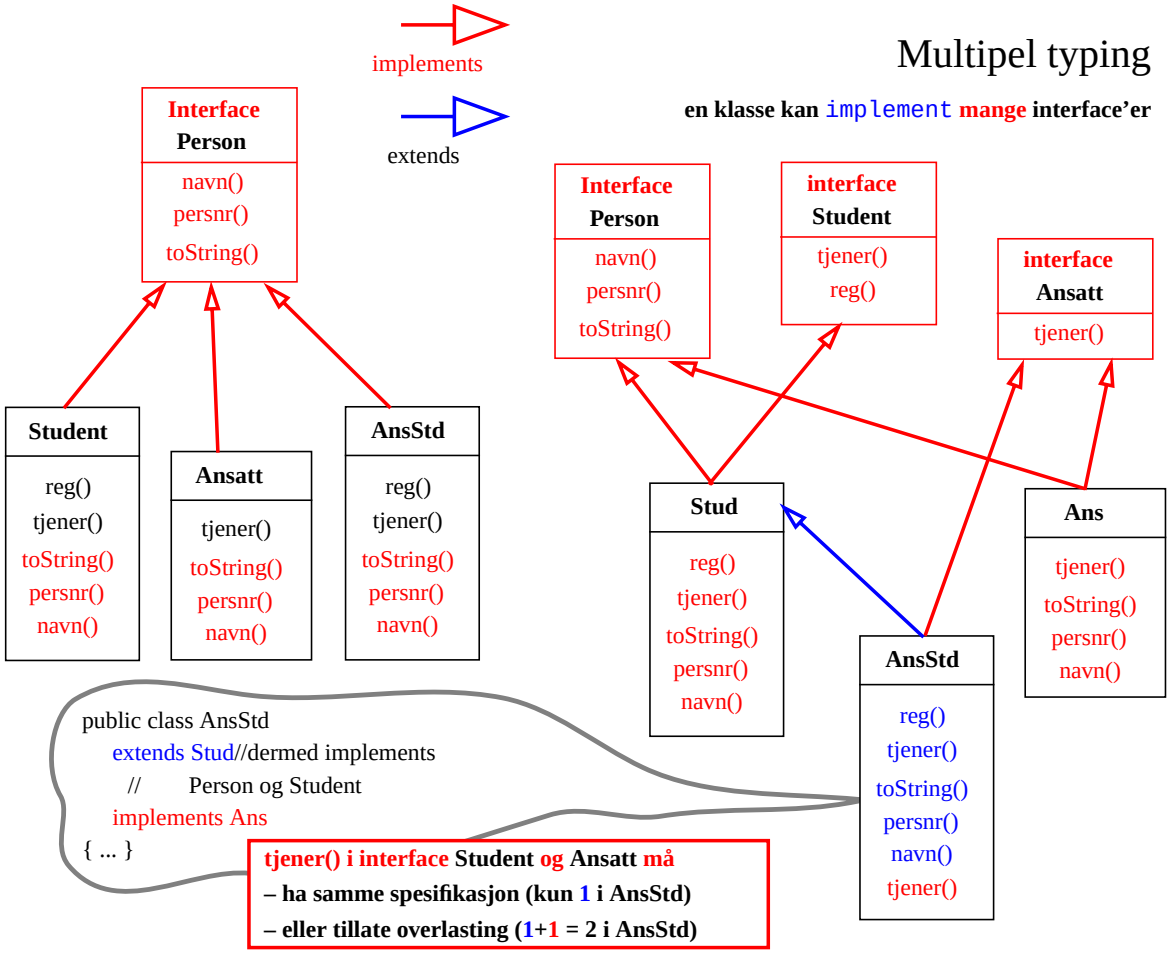
**en klasse kan `extend` kun én (super)klasse**




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 i subklassen – 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(); }
}
```

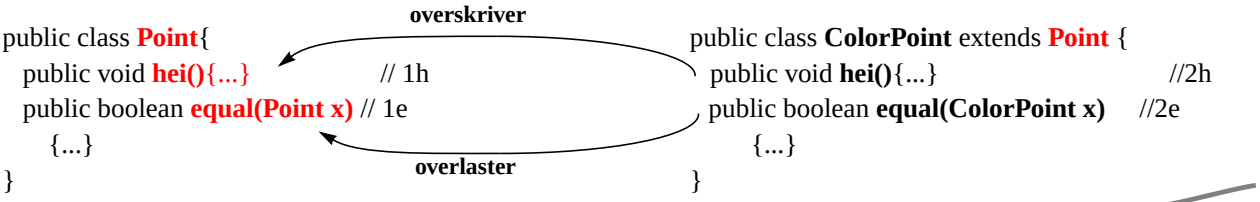
## **Overlasting** (overloading) **:**

samme navn, forskjellige parametre

*samme metodenavn brukes igjen – med forskjellige parametre*

```
public class Overl {
    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
 */
}
```

# ‘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) – “*p2 ***peker på*** ColorPoint*”*

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

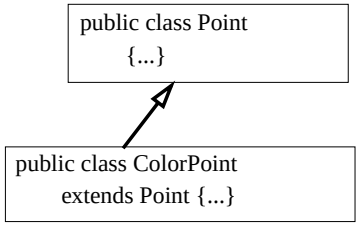
***Overlastede*** *metoder bindes* ***statisk*** *(compile-time) – “utfra* ***deklarert*** *type”*



# 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 kode*
- *og dermed må håndere ting som overskriving/overlasting*
- *og forbyr multipel arv* (`extends` *er arv av implementasjon og type*)

# Jeg vil ...

*ha forskjellige sorteringsmetoder ... for å sammenlikne de*
*de skal sortere tabeller .... med noen objekter*
*og sortering skal skje mht.*
*... en eller annen sammenlikningsoperasjon*

```
public interface Sort {
      void sort();
      void show();
      void set(Object[] t);
}
```

```
public class SortM implements Sort {
      Object[] tab;
      void sort() { mergesort(tab) }
public class SortS implements Sort {
      Object[] tab;
      void sort() { seleksjonsort(tab) }
}
```

```
public class Sort {
      Object[] tab;
      set(Object[] t) {
         tab= new Object[t.size];
         for (int k=0; k<t.size; k++) tab[k]= t[k]; }
      void swap(int i,j) {...}
      void show() { skriv ut tabellen }
      void sort() { }
}
```

```
public class SortM extends Sort {
      void sort() { mergesort(tab) }
public class SortS extends Sort {
      void sort() { seleksjonsort(tab) }
}
public class SortL extends Sort {
      void sort() { gjør det i morgen }
}
```

**g= new Sort() !**
**g.sort() !**
**new SortL().sort() !**



# abstract class

```
abstract class Sort {
  protected Object[] tab;
  protected Comparator cp;
  public void set(Object[] t) {
      set tab = t}
  public void show() { skriv tab }
  public void swap(int i,j) {
      bytt om i og j }
  public abstract void sort();
  public Sort(int m, Comparator c) {
      tab= new Object[m];
      cp= c; }
}
```

*Dette begrenser enhver (programmerer) som skal bruke klassen:*

– `Sort s=new Sort()` – *er ulovlig (selv om konstruktor finns der)*
`s.sort()` – *vil aldri forekomme*

– `class SortS extends Sort { ...`
`public void sort() {` ***må implementeres*** `} ...`
*med mindre man sier*
`abstract class SortS extends Sort {`
*... med nye ting men fortsatt noen* `abstract` *metode(r)*

– `SortS` *kan deklarere nye konstruktorer – og kan bruke super...*

| **interface** | **abstract class** |
|---|---|
| ingen implementasjon | delvis implementasjon |
| ingen datastruktur | mulighet for en datastruktur |
| ingen konstruktorer | mulighet for konstruktorer |
| multipel arv (av type) | enkel arv (som for klasser) |

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

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

# 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** (ikke implementert)

**– kan ikke instansieres, selv om kan definere konstruktører**

| | 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** | + | + | + | + | + | + |

***interface*** brukes for å definere **"abstrakte typer"** = **ingen føringer** på implementasjon = kun **grensesnitt**

***abstract class*** brukes når

– data typen er **ikke "helt abstrakt"** : **en del** av implementasjon er bestemt

– **noen metoder**, vesentlige for data typen, har **ingen "generisk/typisk"** implementasjon

– "*abstrakt*" er en beskjed til programmerer – og kompilator! – om dette

**vanlig *class*** brukes når

– data typen er **"konkret"** : **alle metoder** har en implementasjon – subklasser kan overskrive disse

– for å implementere interface og abstract class



# III. Gjenbruk og tilpassing

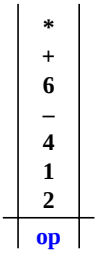
- *Arv, overlasting, overskriving ...*
- *"Parametrisering" – Omstøping (cast)*
- *Unntak*

# Omstøping (cast) – en sikringsmekanisme

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

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

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




*Leser hva ? Si en **String**,*
*passelig avgrenset, dvs.*
*etterfølgende kall til* nextToken()
*vil returnere:* **"2", "4", "1", "–" ...**

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

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

```
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 tilhører kan du bruke*
if (o **instanceof** String) ...
else if (o **instanecof** Klasse) ...



# Tilpassing
(adapter klasser)

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

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

---

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

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

# 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** !*

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

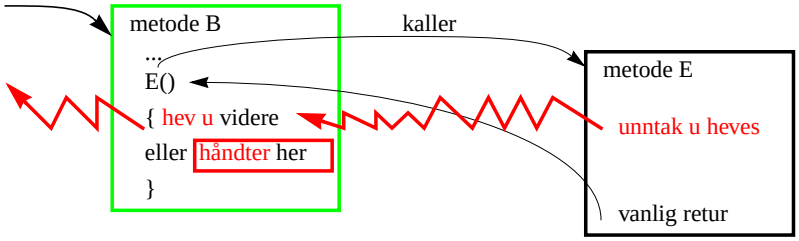
*istedenfor slike 'spesielle objekter' markerer man feilsituasjoner ved å heve et unntak*

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



# Unntakshåndtering

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



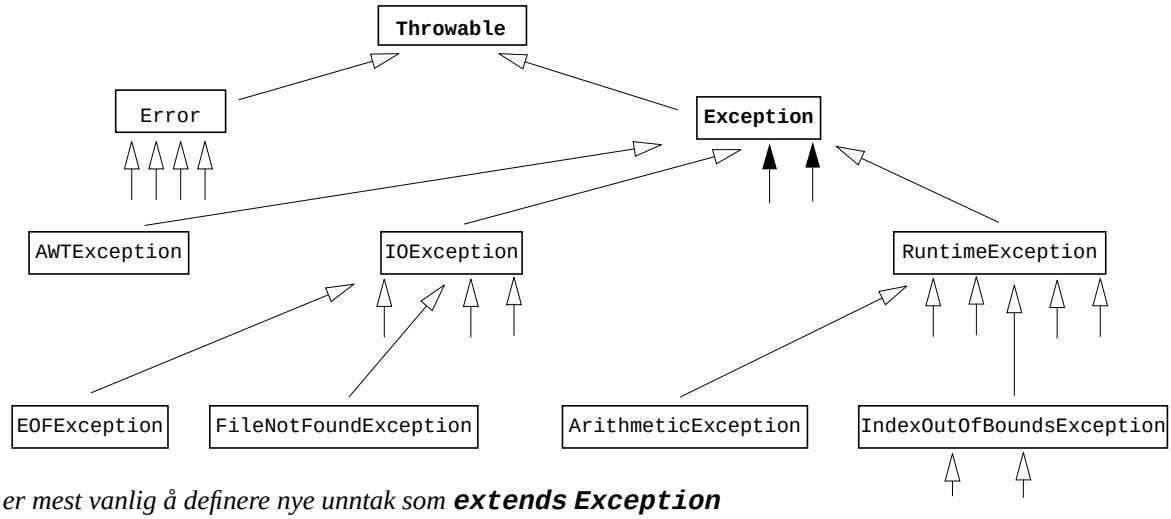

```
public class B {
 Stack S = new Stab() ;
 public Object doS()
    throws EmptyStackExc
 { ...
    Object o = S.peek();
   ...  }
 public Object remTop()
 { try {
    Object o = S.pop();
    return o;
    ... }
   catch (EmptyStackExc e)
    { return null; }
    ... }
 }
```

```
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");
   else {noE--; return elems[noE+1];}
...
```

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

# 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); }
}
```



# Oppsummering

***Interface = ADT***

*– kun spesifikasjon av grensesnittet*
- *med dokumentasjon (for- og bakbetingelser)*

*– kan brukes i andre programmer som vanlige typer*
- *så lenge det ikke trengs å opprette helt nye instanser*
- *spesielt som parametre*

*– tillatter multipel typing*
- *et interface kan utvide flere andre,*
- *en klasse kan implementere flere interface*

***OO og arv***

*– ingen multipel arv*
- *arv av type og kode*

*– synlighetsbegrensninger*

*– overskriving vs. overlasting*
- *dynamisk vs. statisk binding*

*– abstarkte klasser*

***Implementasjon av interface***

*– av alle grensesnitt metoder – mht. dokumentasjon !!*

*– et interface kan ha flere implementasjoner*
- *som kan byttes ut uten å endre resten av programmet*
- *som kan resultere i forskjellig oppførsel av hele programmet*

*– implementers som en klasse*
- *med passende synlighetsbegrensninger*
- *private/protected datastruktur og hjelpemetoder*
- *med unntak istedenfor "eksplisitt" errorhåndtering*

*– kan kreve tilpassing av eksisterende klasser (cast = omstøping, adapter klasser)*

Det finnes ingenting

(ingen konkret program)

som kan gjøres med **interface**

men som ikke kan gjøres uten



Det finnes ingenting

(ingen konkret program)

som kan gjøres med bruk av **unntak**

men som ikke kan gjøres uten

Det finnes ingenting

(ingen konkret program)

som kan gjøres med **typer/kast**

men som ikke kan gjøres uten



Det finnes ingenting

(ingen konkret program)

som kan gjøres med **klasser/hierarki**

men som ikke kan gjøres uten

Det finnes ingenting

(ingen konkret program)

som kan gjøres **Objekt-Orientert**

men som ikke kan gjøres uten objekter



Det finnes ingenting

(ingen konkret program)

som kan **programmeres**

men som ikke kan gjøres med bare

0011010101000101010100000111010110001
0100001010101010010101001001001010001
0100010010 1101010....