

# ADT og OO programmering

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

## **I. ADT I JAVA - INTERFACE**

- I.1 grensesnitt skal dokumenteres – Javadoc
- I.2 bruk av interface
- I.3 implementasjoner av interface

## **II. OO**

- II.1 Arv av type og implementasjon
- II.2 Abstrakte klasser i Java

## **III. BRUK OG ... TILPASSING**

- III.1 Arv ...
- III.2 Casting (omstøping)
- III.3 Exceptions (unntak)

# I. Javadoc

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

# I. Javadoc

```
public interface Stack {  
    /** legger nye Objekter på toppen av stabel  
  
    void push(Object o);  
  
    Object pop();  
  
    Object peek();  
  
    boolean empty();  
}
```

# I. Javadoc

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

```
Object pop();
```

```
Object peek();
```

```
boolean empty();
```

```
}
```

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

# I. Javadoc

```
public interface Stack {  
/** legger nye Objekter på toppen av stabel  
 * @param o Objektet som skal settes inn */  
void push(Object o);  
  
/** fjerner top Objektet fra stabel  
 * @return top (=sist innsatte) Objektet (?null hvis empty())  
  
Object pop();  
  
Object peek();  
  
boolean empty();  
}
```

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

# I. Javadoc

```
public interface Stack {  
  /** legger nye Objekter på toppen av stabel  
   * @param o Objektet som skal settes inn */  
  void push(Object o);  
  
  /** fjerner top Objektet fra stabel  
   * @return top (=sist innsatte) Objektet (?null hvis empty())  
   * @exception NullPointerException hvis empty() */  
  Object pop();  
  
  Object peek();  
  
  boolean empty();  
}
```

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

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

# I. Javadoc

```
public interface Stack {  
/** legger nye Objekter på toppen av stabel  
* @param o Objektet som skal settes inn */  
void push(Object o);  
  
/** fjerner top Objektet fra stabel  
* @return top (=sist innsatte) Objektet (?null hvis empty())  
* @exception NullPointerException hvis empty() */  
Object pop();  
  
/** returnerer (uten å fjerne) top Objektet fra stabel  
* @return top Objektet  
* @exception NullPointerException hvis empty() */  
Object peek();  
  
/** @return true hviss (hvis og bare hvis) stabel er tom */  
boolean empty();  
}
```

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

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

# I. Javadoc

```
/** LIFO (Last In First Out) kø av vilkårlige Objekter – første Objektet er det som ble innsatt sist
 * @author Jan Arne Telle
 * @version 1.2, Aug 19 2000
 */
public interface Stack {
/** legger nye Objekter på toppen av stabel
 * @param o Objektet som skal settes inn */
void push(Object o);

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

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

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

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

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



# I. Javadoc

```
/** LIFO (Last In First Out) kø av vilkårlige Objekter – første Objektet er det som ble innsatt sist
 * @author Jan Arne Telle
 * @version 1.2, Aug 19 2000
 */
public interface Stack {
/** legger nye Objekter på toppen av stabel
 * @param o Objektet som skal settes inn */
void push(Object o);

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

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

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

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

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

> javadoc Stack.java

## *Interface* = en “Abstrakt” Klasse

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

*importeres og programmeres med (brukes) som alle andre klasser . . . men :*

- *har **ingen konstruktører** – nye objekter kan ikke opprettes*
- *skal nye objekter opprettes, må man bruke **en klasse** som implementerer dette interface*
- *dog, interface kan også tilby metode som newContainer for “kloning” av eksisterende objekter*

## *Interface* = en “Abstrakt” Klasse

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

*importeres og programmeres med (brukes) som alle andre klasser . . . men :*

- *har **ingen konstruktører** – nye objekter kan ikke opprettes*
- *skal nye objekter opprettes, må man bruke **en klasse** som implementerer dette interface*
- *dog, interface kan også tilby metode som newContainer for “kloning” av eksisterende objekter*

*Klasse =*

### *Grensesnitt*

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

# *Interface* = en “Abstrakt” Klasse

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

*importeres og programmeres med (brukes) som alle andre klasser . . . men :*

- *har **ingen konstruktører** – nye objekter kan ikke opprettes*
- *skal nye objekter opprettes, må man bruke **en klasse** som implementerer dette interface*
- *dog, interface kan også tilby metode som newContainer for “kloning” av eksisterende objekter*

*Klasse =*

## *Grensesnitt*

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

## *+ DataStruktur*

- *som skal være privat for klassen, dvs. skjult for bruker*
- *kan forandre tilstanden til DS kun v.hj.a. grensesnitt-operasjoner*

# *Interface* = en “Abstrakt” Klasse

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

*importeres og programmeres med (brukes) som alle andre klasser . . . men :*

- *har **ingen konstruktører** – nye objekter kan ikke opprettes*
- *skal nye objekter opprettes, må man bruke **en klasse** som implementerer dette interface*
- *dog, interface kan også tilby metode som newContainer for “kloning” av eksisterende objekter*

*Klasse =*

## *Grensesnitt*

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

## *+ DataStruktur*

- *som skal være privat for klassen, dvs. skjult for bruker*
- *kan forandre tilstanden til DS kun v.hj.a. grensesnitt-operasjoner*

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

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public Object peek() throws NullPointerException {
```

```
public void push(Object o) {
```

```
public Object pop() throws NullPointerException {
```

```
public boolean empty() {
```

```
public Stack newContainer() {
```

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {
```

```
    public Stab() {
```

```
        public Object peek() throws NullPointerException {
```

```
            public void push(Object o) {
```

```
                public Object pop() throws NullPointerException {
```

```
                    public boolean empty() {
```

```
                        public Stack newContainer() {
```

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {  
    private Object[] elems;  
    private int antall, max=10;  
    /** -1 <= antall < max; elems[antall] - sist innsatte elementet */  
    public Stab() {  
  
    public Object peek() throws NullPointerException {  
  
    public void push(Object o) {
```

```
        public Object pop() throws NullPointerException {  
  
    public boolean empty() {  
  
    public Stack newContainer() {
```





## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {  
    private Object[] elems;  
    private int antall, max=10;  
    /** -1 <= antall < max; elems[antall] - sist innsatte elementet */  
    public Stab() {  
        elems= new Object[max]; antall= -1; }  
  
    public Object peek() throws NullPointerException {  
        if (empty())  
            throw new NullPointerException("Tom stabel");  
        else return elems[antall];  
    }  
    public void push(Object o) {
```

```
        public Object pop() throws NullPointerException {  
  
        public boolean empty() {  
            return antall < 0; }  
  
        public Stack newContainer() {  
            return new Stab(); }  
    }
```

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** -1 <= antall < max; elems[antall] - sist innsatte elementet */
    public Stab() {
        elems= new Object[max]; antall= -1; }

    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom stabel");
        else return elems[antall];
    }
    public void push(Object o) {
```

```
        public Object pop() throws NullPointerException {
            if (empty())
                throw new NullPointerException("Tom!!");
            else {
                antall--;
                return elems[antall+1]; }
        }
    public boolean empty() {
        return antall < 0; }

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

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** -1 <= antall < max; elemes[antall] - sist innsatte elementet */
    public Stab() {
        elems= new Object[max]; antall= -1; }

    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom stabel");
        else return elems[antall];
    }

    public void push(Object o) {
        if (antall==max-1) {
            Object[] temp= new Object[max];
            Copy(elems, temp);
            max= 2*max;
            elems= new Object[max];
            Copy(temp, elems); }
        antall++;
        elems[antall]= o;
    }
}
```

```
public Object pop() throws NullPointerException {
    if (empty())
        throw new NullPointerException("Tom!!");
    else {
        antall--;
        return elems[antall+1]; }
}

public boolean empty() {
    return antall < 0; }

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

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

# En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {
```

```
    public Stab() {
```

```
        public Object peek() throws NullPointerException
```

```
        public void push(Object o)
```

```
        public Object pop() throws NullPointerException
```

```
        public boolean empty()
```

```
        public Stack newContainer()
```



*en programmerer bruker  
Stab kun som en **Stack!***

```
...  
Stack po = new Stab();  
po.push(...); ...  
metode(po.pop());
```

## En implementasjon av *interface Stack* (en Stack-Klasse)

```
public class Stab implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** -1 <= antall < max; elemes[antall] - sist innsatte elementet */
    public Stab() {
        elems= new Object[max]; antall= -1; }

    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom stabel");
        else return elems[antall];
    }

    public void push(Object o) {
        if (antall==max-1) {
            Object[] temp= new Object[max];
            Copy(elems, temp);
            max= 2*max;
            elems= new Object[max];
            Copy(temp, elems); }
        antall++;
        elems[antall]= o;
    }
}
```

```
public Object pop() throws NullPointerException {
    if (empty())
        throw new NullPointerException("Tom!!");
    else {
        antall--;
        return elems[antall+1]; }
}

public boolean empty() {
    return antall < 0; }

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

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

## En annen implementasjon av *interface Stack* ?

```
public class Koe implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** -1 <= antall < max; elemes[antall] - sist innsatte elementet */
    public Koe() {
        elems= new Object[max]; antall= -1; }

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

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

public boolean empty() { return antall < 0; }

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

/** @param fra.length <= 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<= antall; k++)
        elems[k-1] = elems[k];
}
}
```

## En annen implementasjon av *interface Stack* ? IKKE BRA!

```
public class Koe implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** -1 <= antall < max; elemes[antall] - sist innsatte elementet */
    public Koe() {
        elems= new Object[max]; antall= -1; }

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

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

public boolean empty() { return antall < 0; }

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

/** @param fra.length <= 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<= antall; k++)
        elems[k-1] = elems[k];
}
}
```



## En annen implementasjon av *interface Stack* ? IKKE BRA!

```
public class Koe implements Stack {
    private Object[] elems;
    private int antall, max=10;
    /** 1 <= antall < max; elems[antall] = sist innsatte elementet */
    /** returnerer (uten å fjerne) top (=sist innsatte) Objektet
     * @return top Objektet
     * @exception NullPointerException hvis empty() */
    public Object peek() throws NullPointerException {
        if (empty())
            throw new NullPointerException("Tom koe");
        else return elems[0];
    }
    /** legger nye Objekter på toppen av stabel */
    public void push(Object o) {
        if (antall==max-1) {
            Object[] temp= new Object[max];
            Copy(elems, temp);
            max= 2*max;
            elems= new Object[max];
            Copy(temp, elems); }
        antall++;
        elems[antall]= o;
    }
}
```

```
public Object pop() throws NullPointerException {
    /** fjerner top Objektet fra stabel
     * @return top (=sist innsatte) Objektet
     * @exception NullPointerException hvis empty() */
    antall--;
    return o; }
}

public boolean empty() {

}

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

/** @param fra.length <= 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<= antall; 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();  
}
```

# 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();  
}
```

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

```
public interface Queue extends LiFi {  
  /** FIFO subtype : peek() – objektet som lagret lengst  
   * if X.empty() : X.add(o).peek() = o  
   * else X.add(o).peek() = X.peek() */ }  
}
```

# 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 : peek() – sist innsatte objektet  
   X.add(o).peek() = o */ }  
}
```

```
public interface Queue extends LiFi {  
  /** FIFO subtype : peek() – objektet som lagret lengst  
   if X.empty() : X.add(o).peek() = o  
   else X.add(o).peek() = X.peek() */ }  
}
```

# Oppsummering av ADT

## *Vi programmerer ADT'er*

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

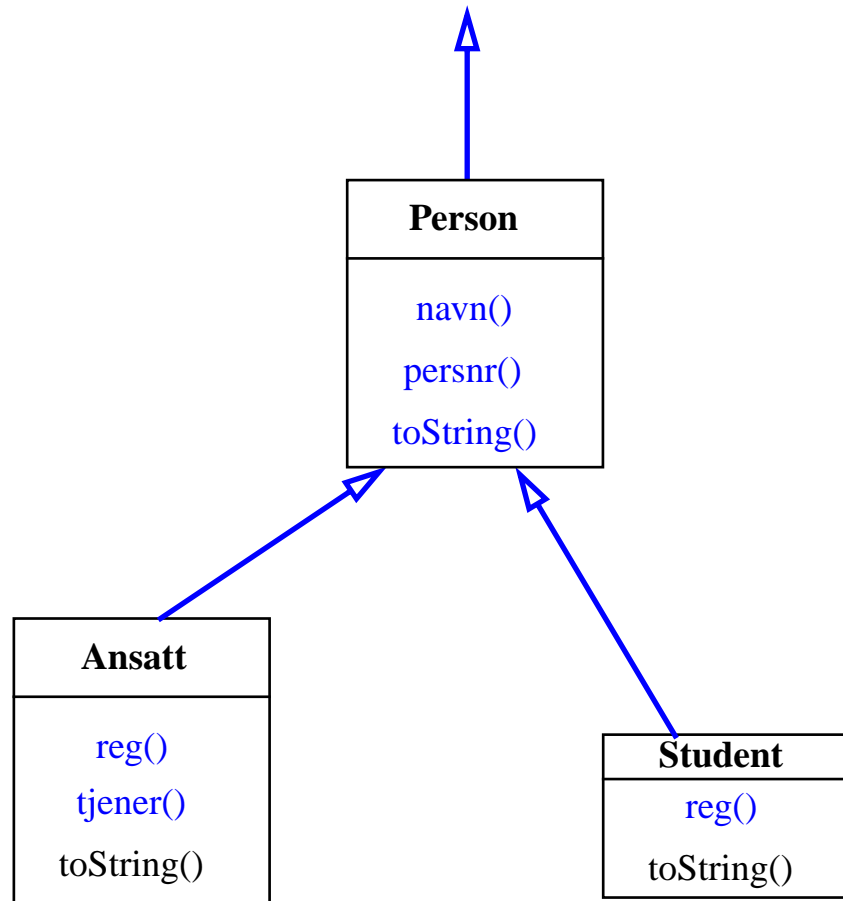
## *En modul brukes kun gjennom grensesnitt*

- 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 *opprettet nye objekter* av et Interface *braker konkret Klasse* (class) som implementerer interfacet
- andre burde bruke minst mulig konkrete Klasser og mest mulig ADT (interface)

## II. OO – arv av type & kode

Ingen multippel arv (av kode)

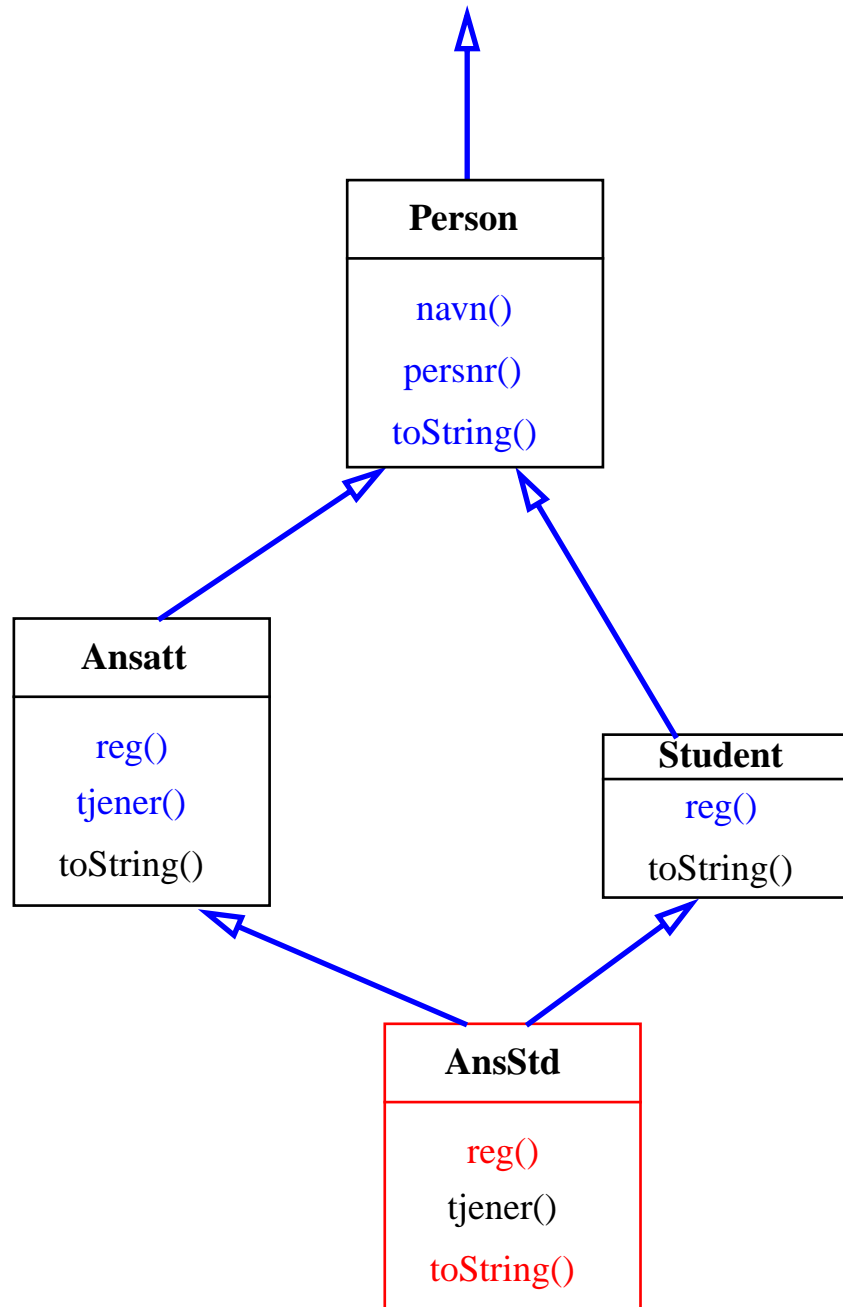
extends 



## II. OO – arv av type & kode

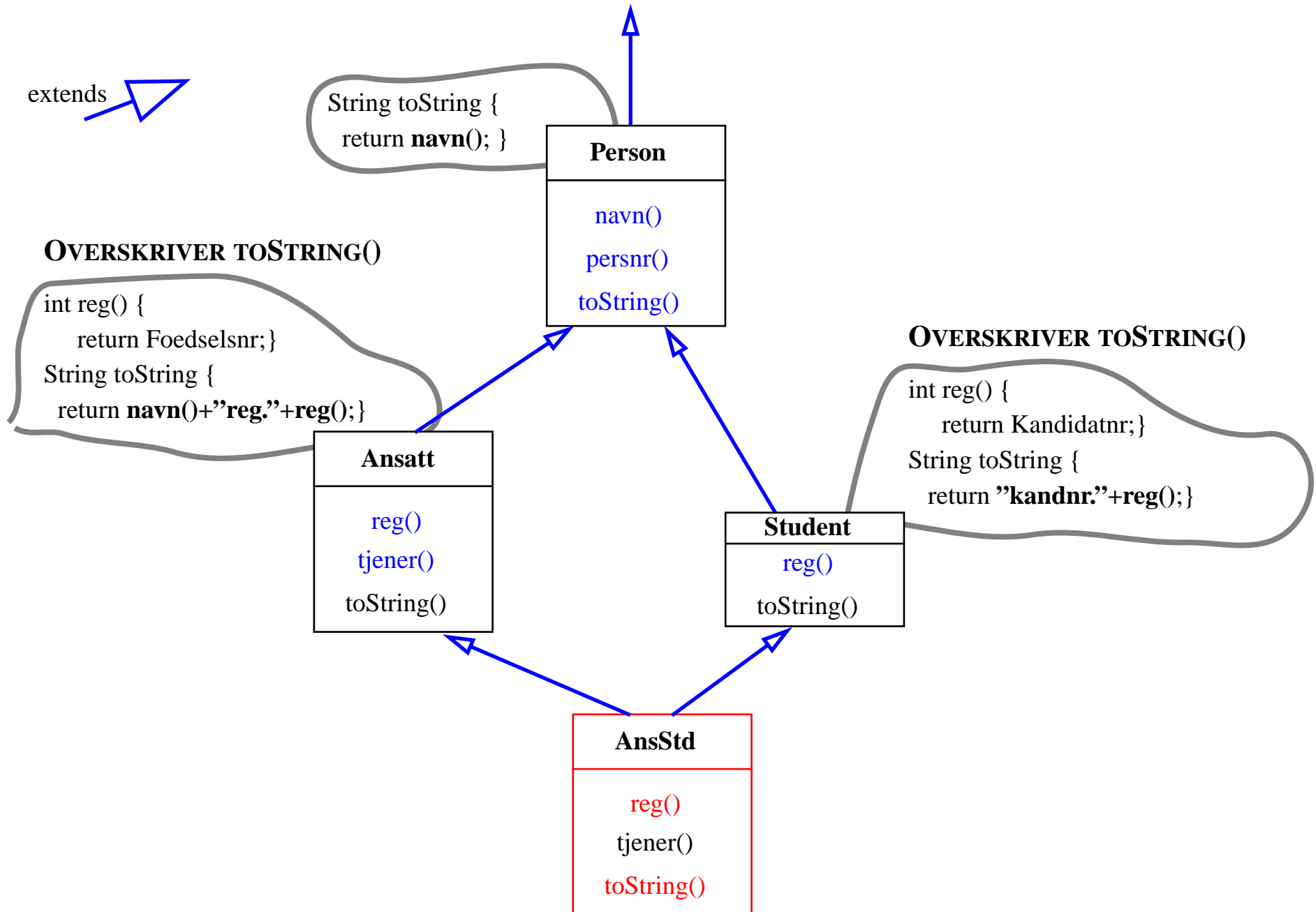
Ingen multippel arv (av kode)

extends 



## II. OO – arv av type & kode

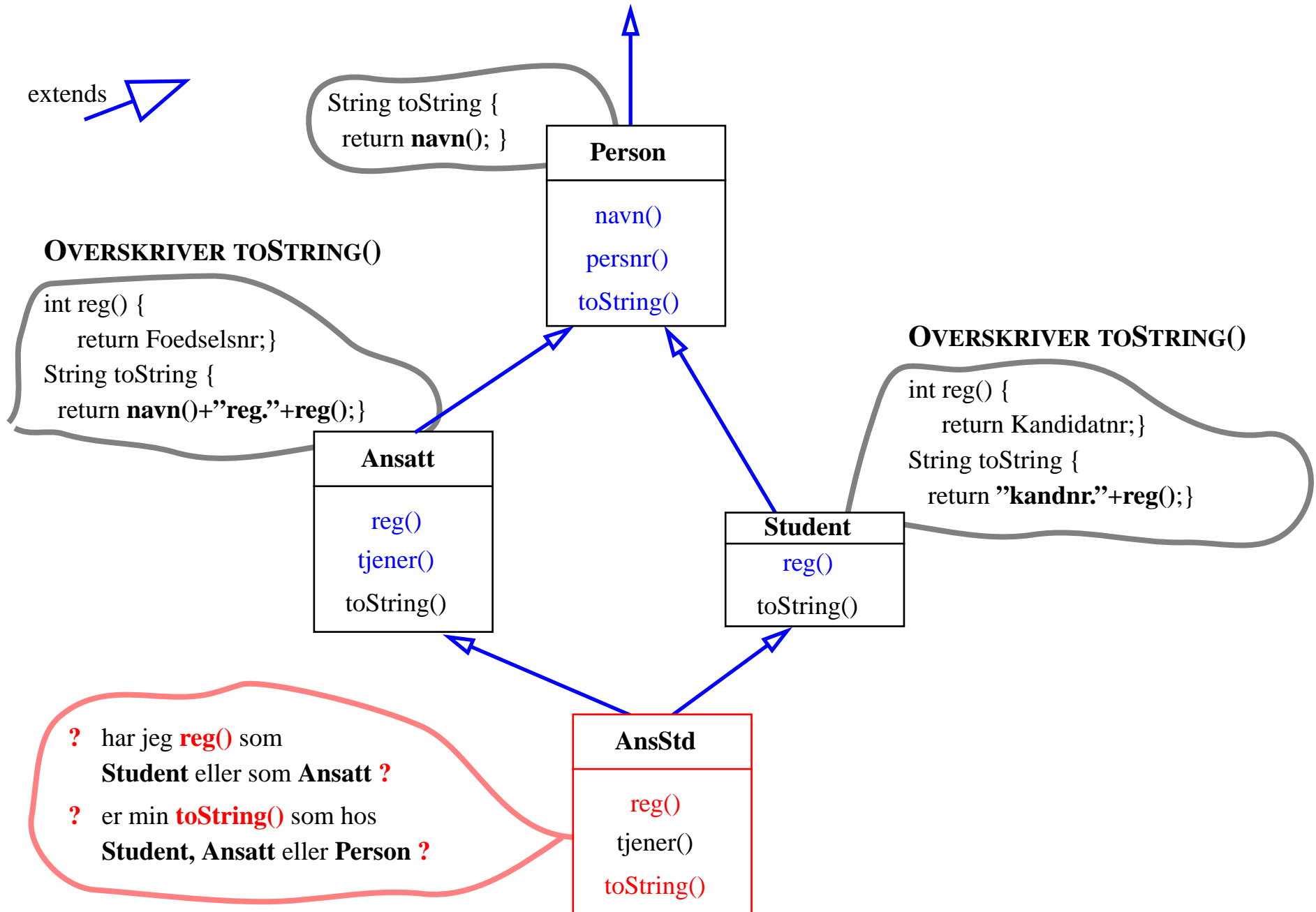
## Ingen multippel arv (av kode)





## II. OO – arv av type & kode

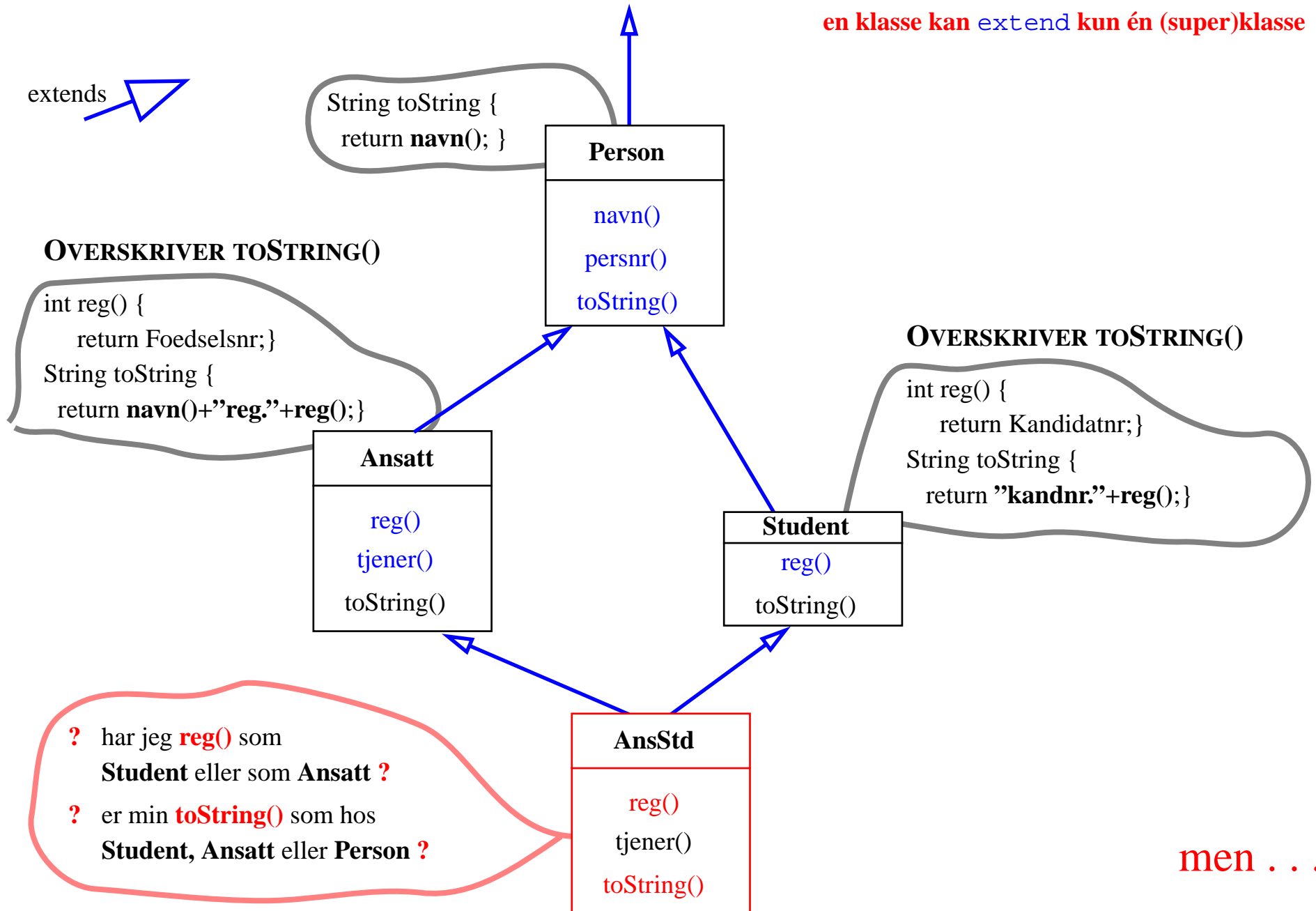
## Ingen multippel arv (av kode)



## II. OO – arv av type & kode

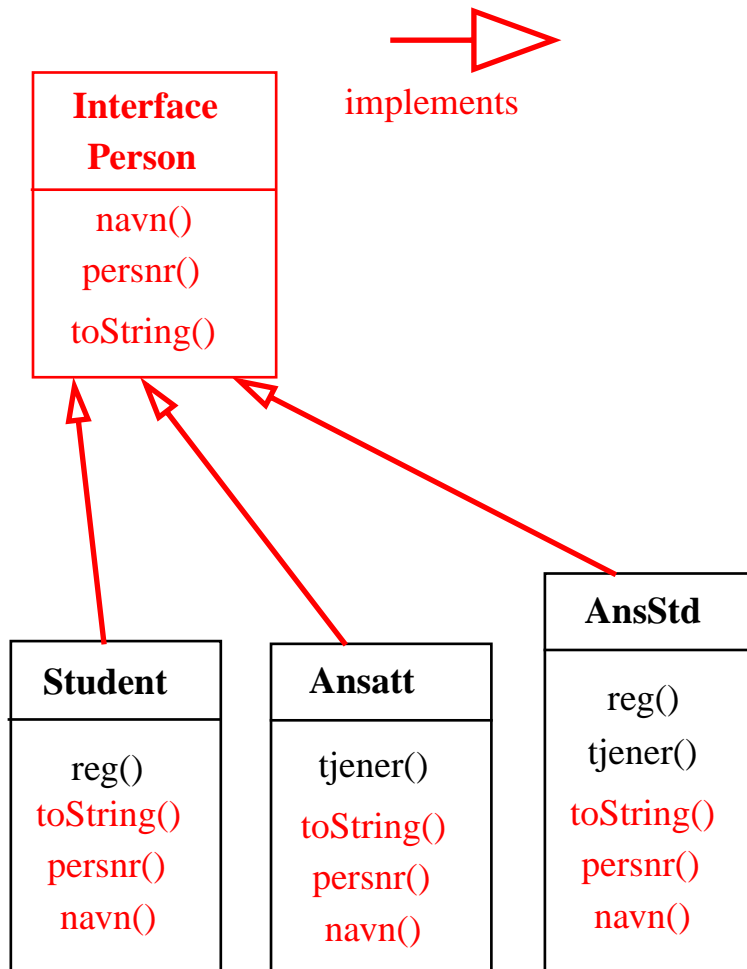
## Ingen multippel arv (av kode)

en klasse kan extend kun én (super)klasse



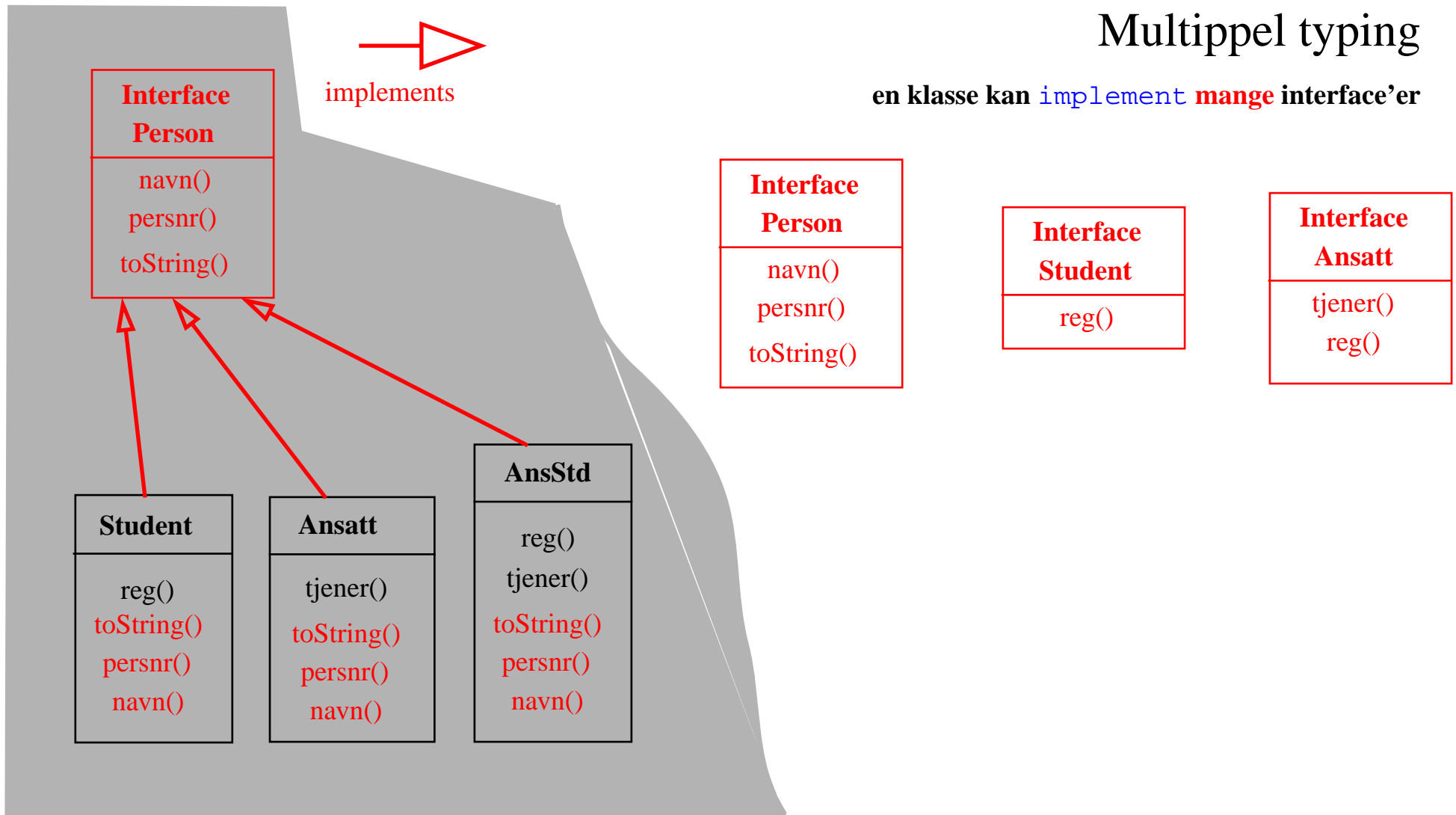
# Multipel typing

en klasse kan `implement` **mange** interface'er



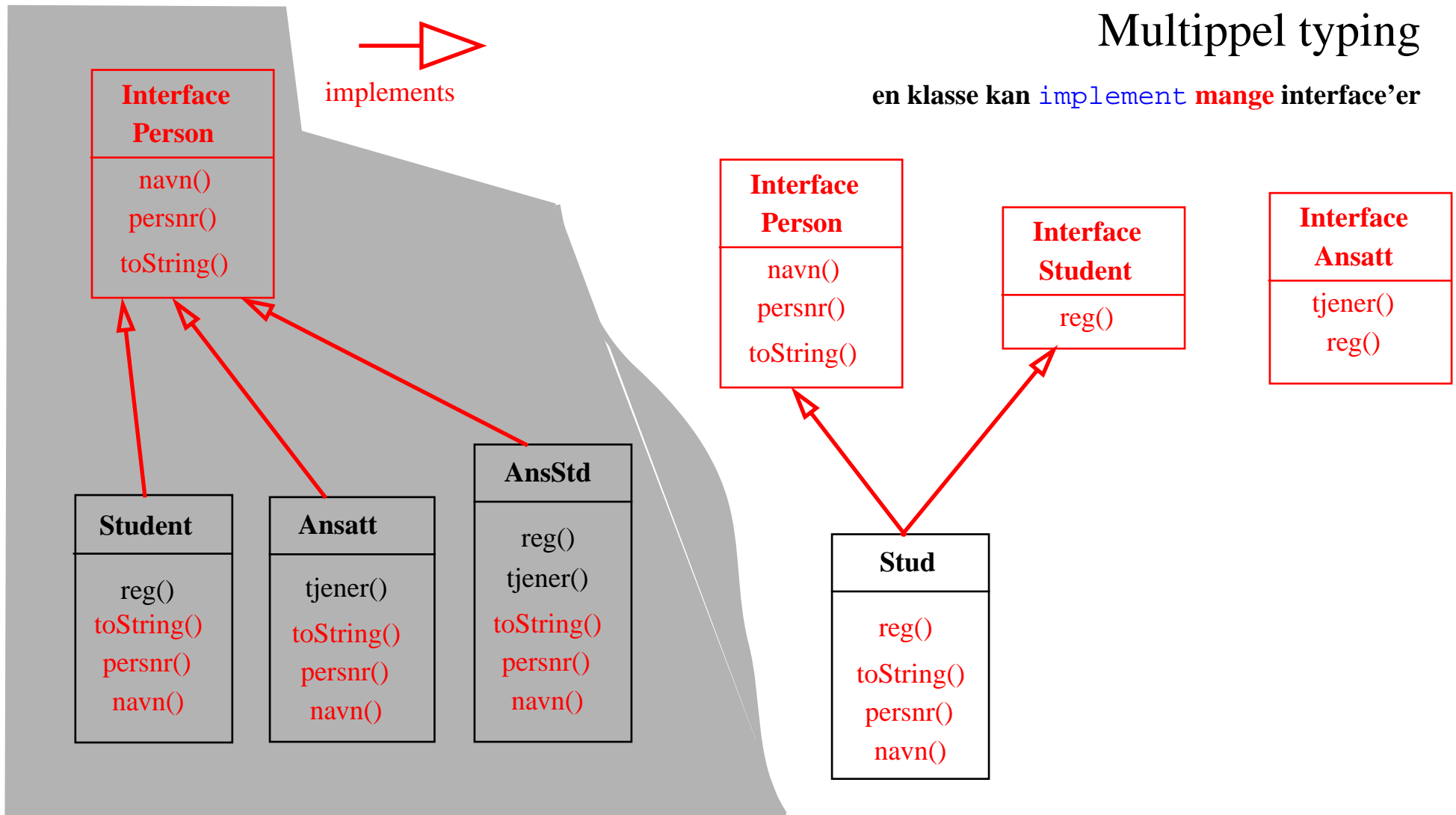
# Multipel typing

en klasse kan **implement** **mange** interface'er



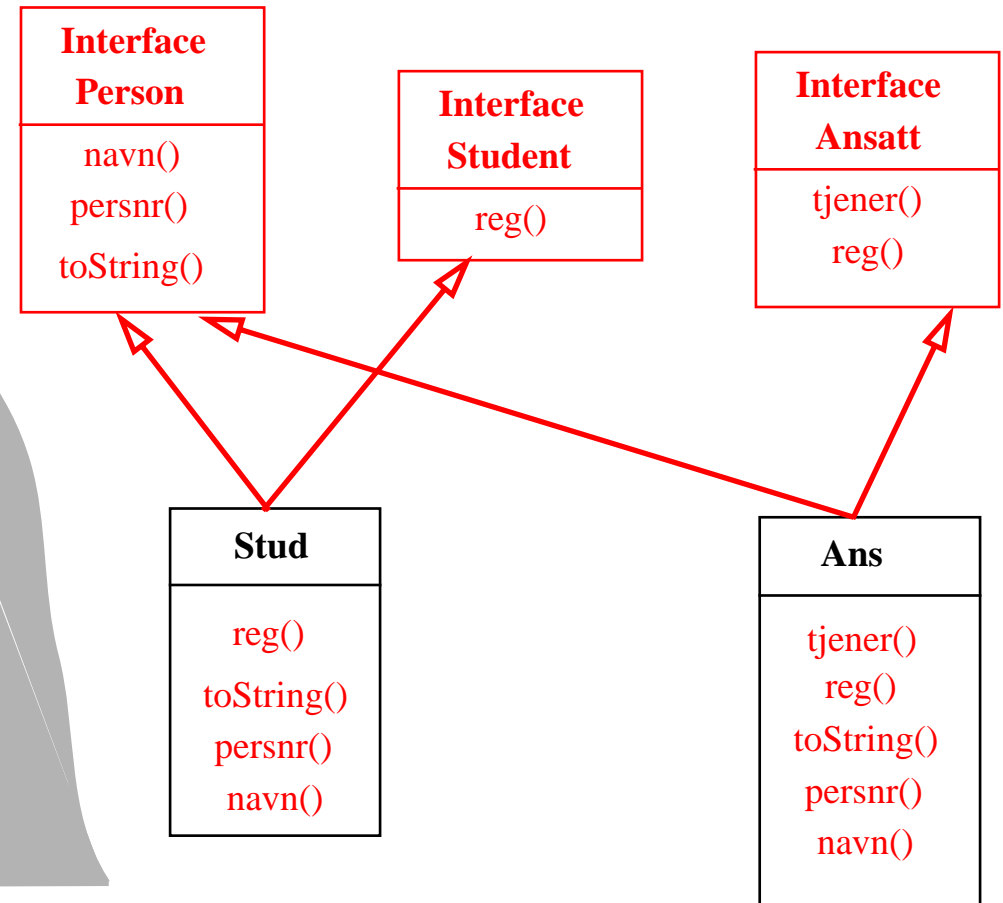
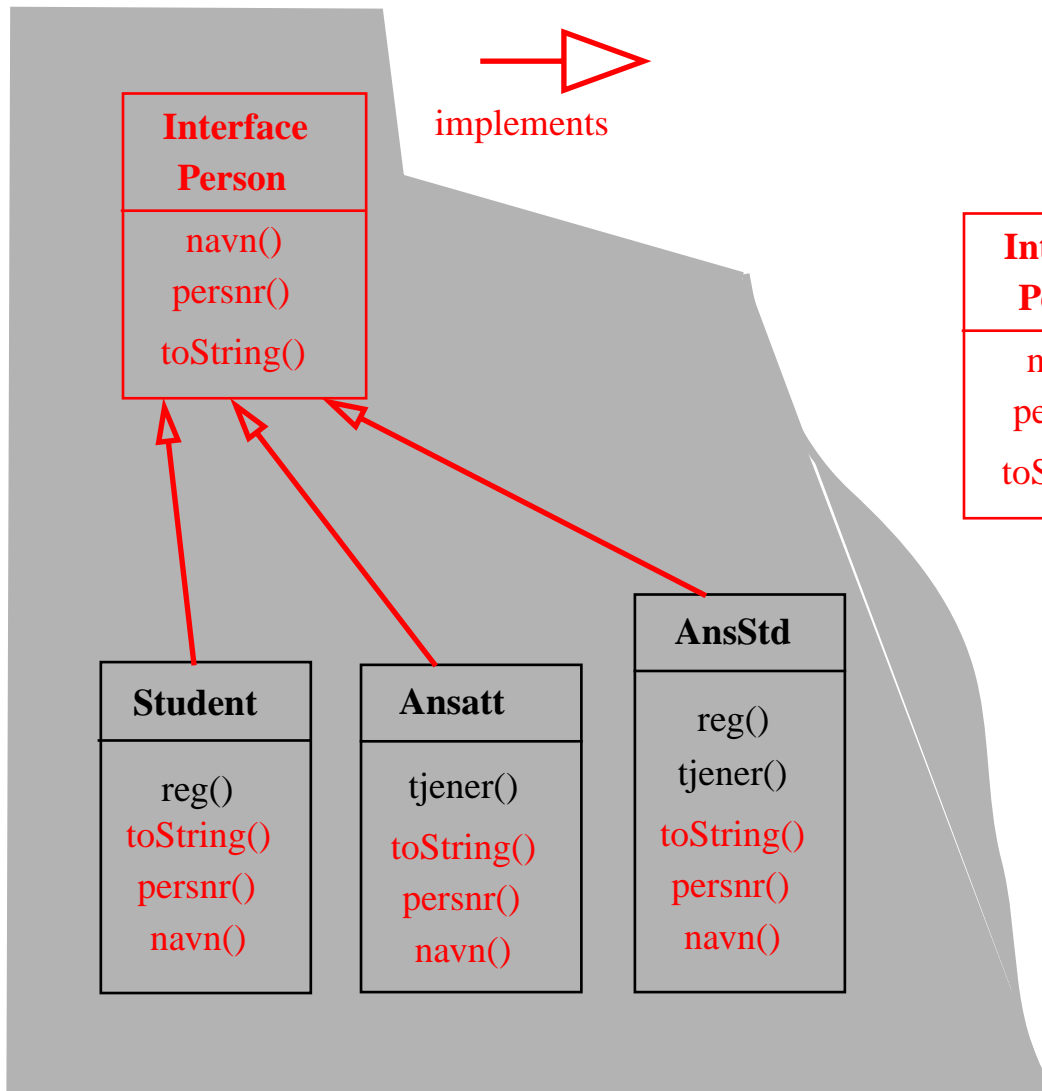
# Multipel typing

en klasse kan **implement** **mange** interface'er



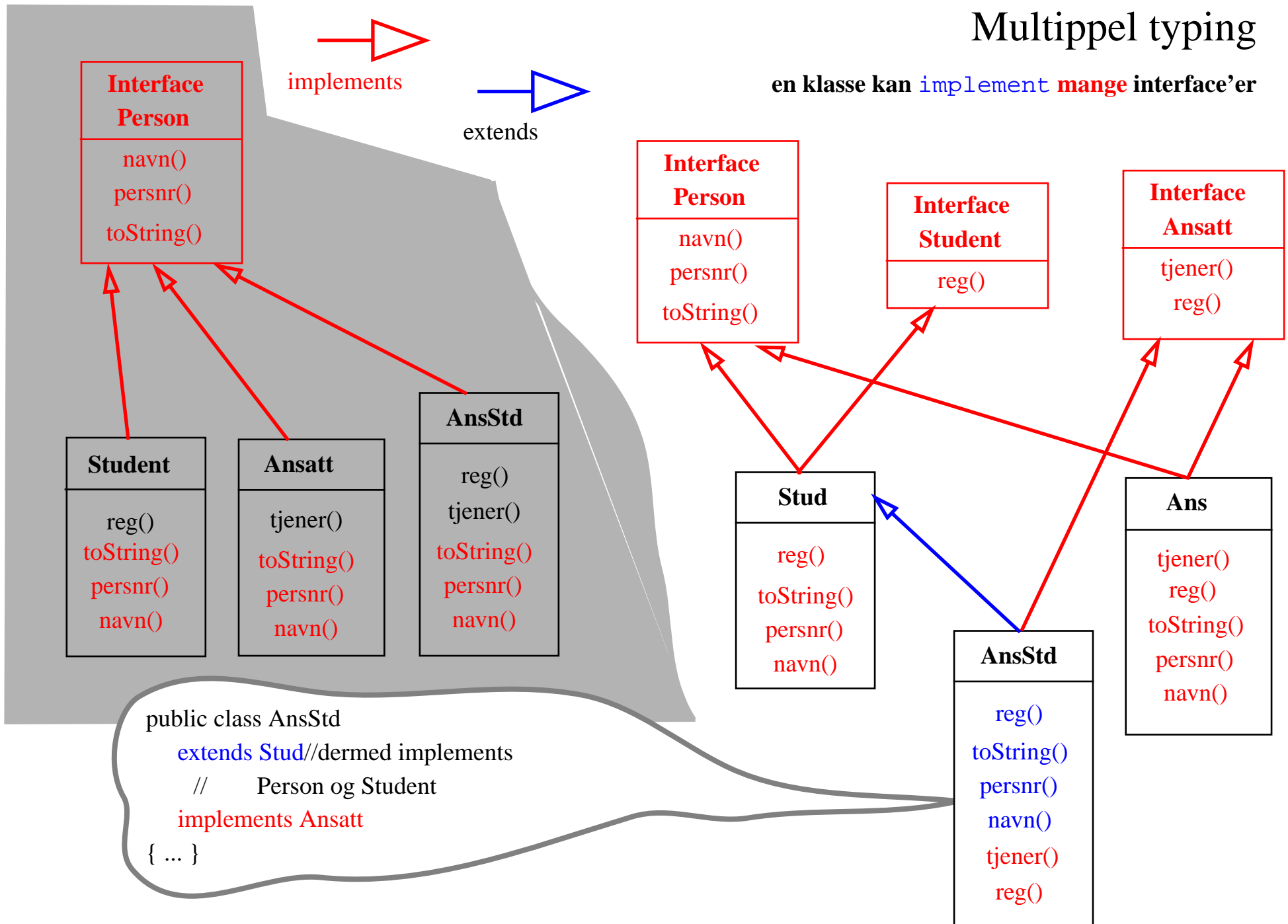
# Multipel typing

en klasse kan **implement** **mange** interface'er



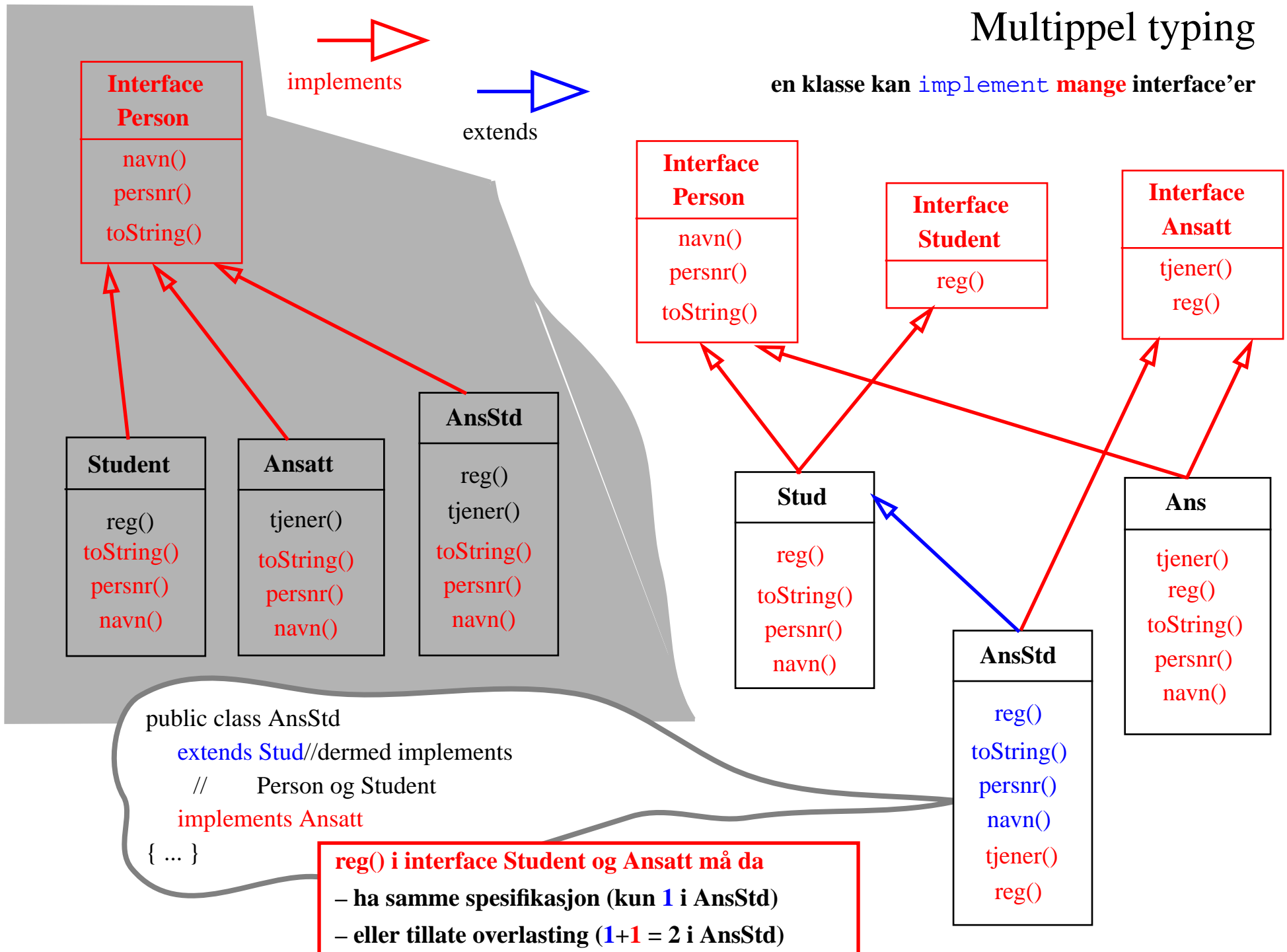
# Multipel typing

en klasse kan **implement** **mange** interface'er



# 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(); }  
}
```

# 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 deklarerer igjen  
(muligens i samme klasse)  
– med forskjellige parametre*

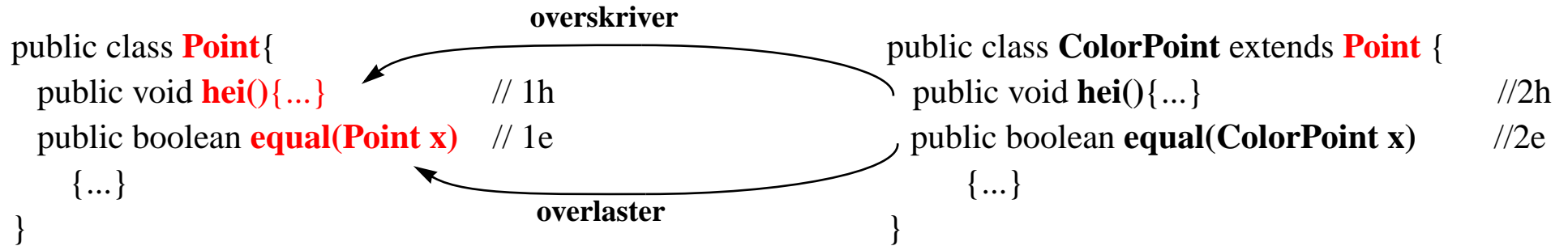
```
public class Overl {  
    public int tall() { return 50; }  
    public int tall(int k) { return k+1; }  
/*  
 * public char tall() {...} er ulovlig !! (hvorfor?)  
 * men  
 * public char tall(int k, char c) {...} er ok  
 */  
}
```

## ‘Dynamisk binding – Statisk overlasting’

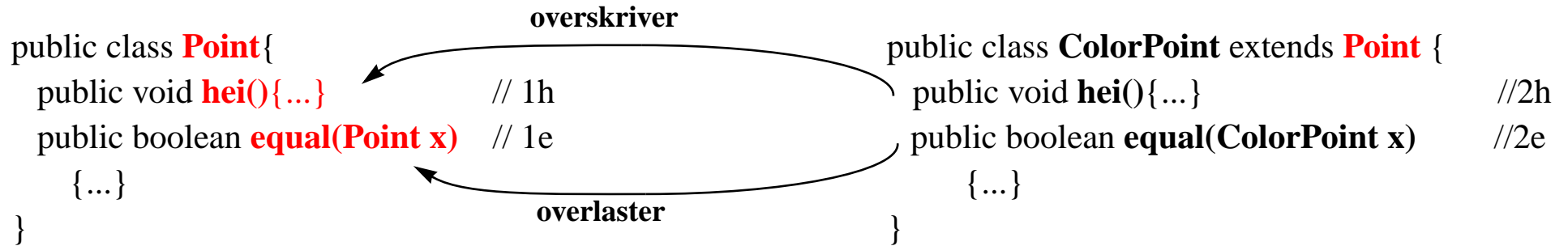
```
public class Point{  
    public void hei(){...}  
    public boolean equal(Point x)  
        {...}  
}
```

```
public class ColorPoint extends Point {  
    public void hei(){...}  
    public boolean equal(ColorPoint x)  
        {...}  
}
```

# ‘Dynamisk binding – Statisk overlasting’



# ‘Dynamisk binding – Statisk overlastering’



**Point p1** = new **Point**();

**ColorPoint cp** = new **ColorPoint**();

**Point p2** = cp;





# ‘Dynamisk binding – Statisk overlastering’

```
public class Point{
```

```
    public boolean equal(Point x) // 1e
```

```
    {...}
```

```
}
```

```
public class ColorPoint extends Point {
```

```
    public boolean equal(ColorPoint x) //2e
```

```
    {...}
```

```
}
```



```
Point p1 = new Point();
```

```
ColorPoint cp = new ColorPoint();
```

```
Point p2 = cp;
```

```
a. p1.hei(); // 1h
```

```
b. cp.hei(); // 2h
```

```
c. p2.hei(); // 2h
```

```
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);
```

*Overskrevne metoder bindes dynamisk  
(run-time) – “**p2** peker på ColorPoint”*



# ‘Dynamisk binding – Statisk overlastering’

```
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  
    {...}  
}
```

overskriver

overlaster

**Point p1** = new **Point**();

**ColorPoint cp** = new **ColorPoint**();

**Point p2** = **cp**;

- p1**.hei(); // 1h
- cp**.hei(); // 2h
- p2**.hei(); // 2h

*Overskrevne metoder bindes dynamisk  
(run-time) – “**p2** peker på ColorPoint”*

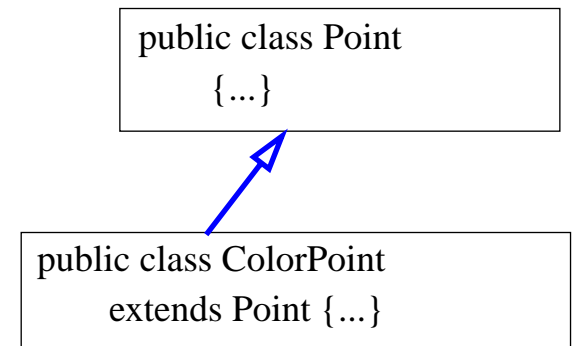
- p1**.equal(**p1**);
- p1**.equal(**p2**);
- p2**.equal(**p1**);
- p2**.equal(**p2**);
- cp**.equal(**p1**);
- cp**.equal(**p2**);
- p1**.equal(**cp**);
- p2**.equal(**cp**);
- cp**.equal(**cp**); // 2e

*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 egenskaper*
  - *kan brukes på objekter av nye, spesifikke subklasser*

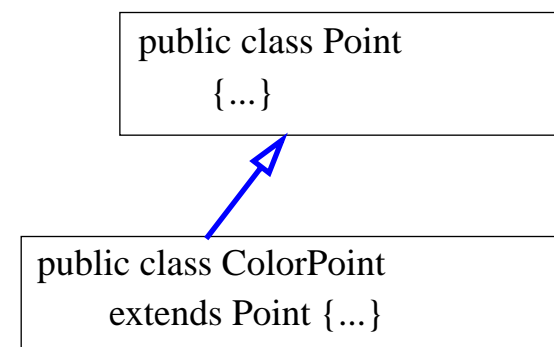
```
public class mittProgram {  
    ...  
    void proc(Point p) {...}  
}
```



## Arv: oppsummering

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

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



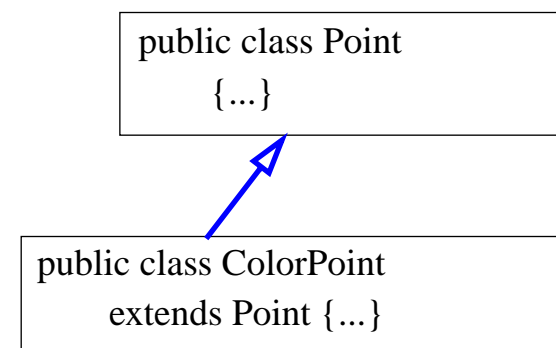
*Dog:*

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

# Arv: oppsummering

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

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

## 3 mulige løsninger (interface, class, abstract class) – hvilken er best?

*Vil ha: forskjellige sorteringsmetoder ...*

*– de skal sortere tabeller av noen objekter*

*– og sortering skal skje mht.*

*... en eller annen sammenlikningsoperasjon*

## 3 mulige løsninger (interface, class, abstract class) – hvilken er best?

*Vil ha: forskjellige sorteringsmetoder ...*

*– de skal sortere tabeller av 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) }  
}
```

### 3 mulige løsninger (interface, class, abstract class) – hvilken er best?

Vil ha: forskjellige sorteringsmetoder ...  
– de skal sortere tabeller av 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) }
```

### 3 mulige løsninger (interface, class, abstract class) – hvilken er best?

Vil ha: forskjellige sorteringsmetoder ...  
– de skal sortere tabeller av 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) }
```

```
g= new Sort() !  
g.sort() !
```



### 3 mulige løsninger (interface, class, abstract class) – hvilken er best?

Vil ha: forskjellige sorteringsmetoder ...  
– de skal sortere tabeller av 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() !
```

## 3dje løsning er her best: 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; }  
}
```

## 3dje løsning er her best: 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 konstruktør 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...

## 3dje løsning er her best: 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 konstruktør 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 deklareere nye konstruktorer – og kan bruke super...

### interface

ingen implementasjon  
ingen datastruktur  
ingen konstruktører  
(og ingen instansiering, :- new...)  
multipel arv (av type)

### abstract class

delvis implementasjon  
mulighet for en datastruktur  
mulighet for konstruktører  
(men ingen instansiering!)  
enkel arv (som for klasser)

### 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 nesteTegn()

vil returnere: “2”, “1”, “4”, “-” ...

```
/* while (mer) {  
    String s= nesteTegn();  
    st.push(s); }  
*/
```

*
+
6
-
4
1
2
st

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

# 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 nesteTegn()  
vil returnere: “2”, “1”, “4”, “-” ...

```
/* while (mer) {  
    String s= nesteTegn();  
    st.push(s); }  
*/
```

*
+
6
-
4
1
2
st

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

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

# 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 nesteTegn()  
vil returnere: “2”, “1”, “4”, “-” ...

```
/* while (mer) {  
    String s= nesteTegn();  
    st.push(s); }  
*/
```

*
+
6
-
4
1
2
st

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

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



# 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

	*
	+
	6
	-
	4
	1
	2
st	

Leser hva ? Si en **String**,  
passelig avgrenset, dvs.  
etterfølgende kall til nesteTegn()  
vil returnere: “2”, “1”, “4”, “-” ...

```
/* while (mer) {  
    String s= nesteTegn();  
    st.push(s); }  
*/
```

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

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

Vet du ikke hvilken klasse Objekter tilhører kan du bruke  
if (o instanceof String) ...  
else if (o instanceof Klasse) ...

# Tilpassing

(Adapter design pattern)

```
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**.....*

# Tilpassing

(Adapter design pattern)

```
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) peek(); }
}
```

# Unntak (Exception)


*For systematisk og modulær feilhåndtering*

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

# Unntak (Exception)

*For systematisk og modulær feilhåndtering*

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



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

## Unntak (Exception)

*For systematisk og modulær feilhåndtering*

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

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


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

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

## Unntak (Exception)

*For systematisk og modulær feilhåndtering*

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



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

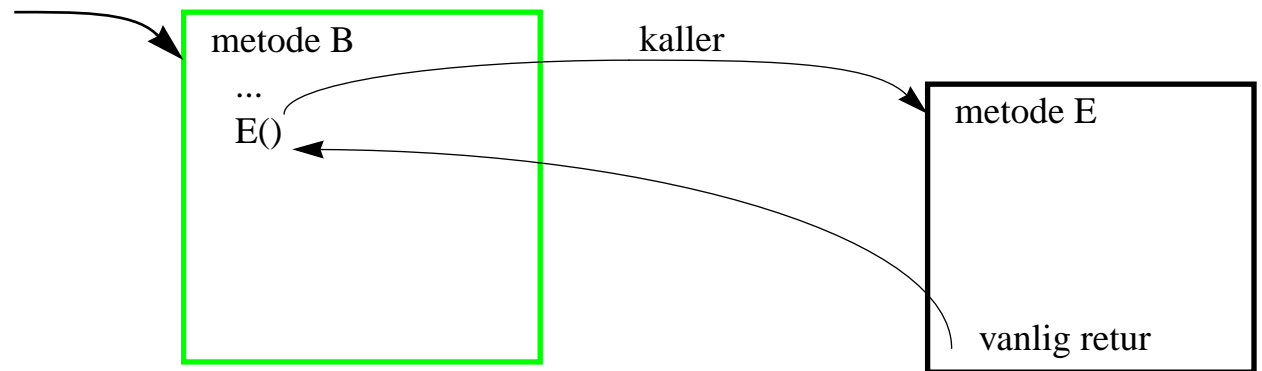
```
public interface Stack {  
    Object peek() throws EmptyStackExc;  
    Object pop() throws EmptyStackExc;  
    ...  
}
```

*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[antall];  
    }  
    public Object pop() throws EmptyStackExc  
    { if (empty())  
        throw new EmptyStackExc("tom");  
        else { antall--; return elems[antall+1]; }  
    }  
    ...  
}
```

# Unntakshåndtering

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



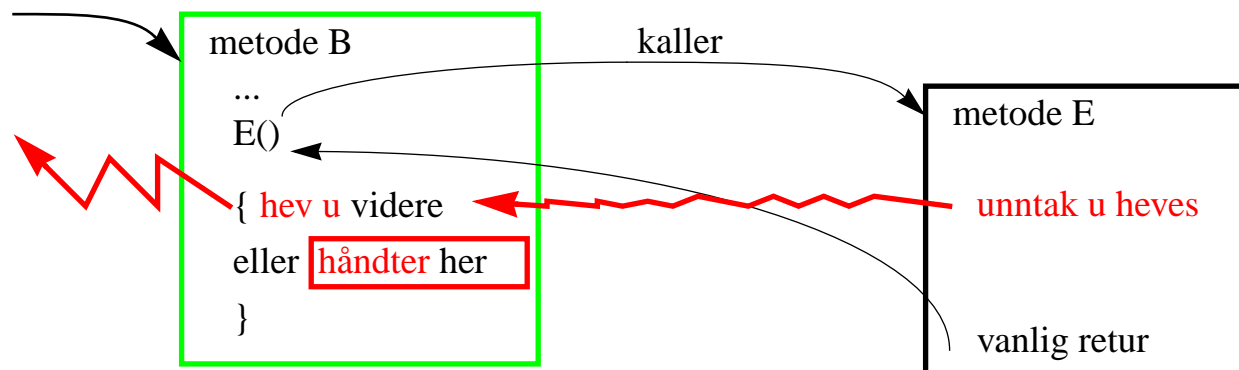


# Unntakshåndtering

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

1. Unntaket kan fanges **catch**'es, eller

2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul

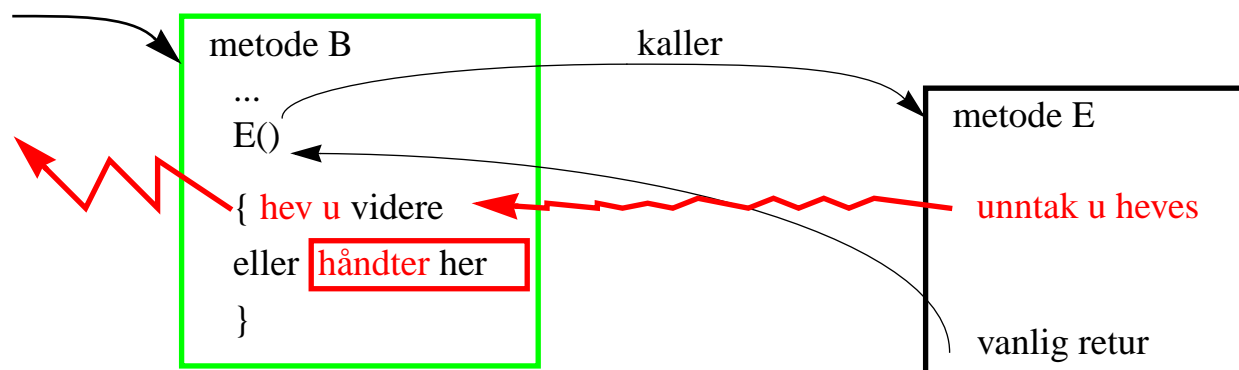


# Unntakshåndtering

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

1. Unntaket kan fanges **catch**'es, eller

2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul



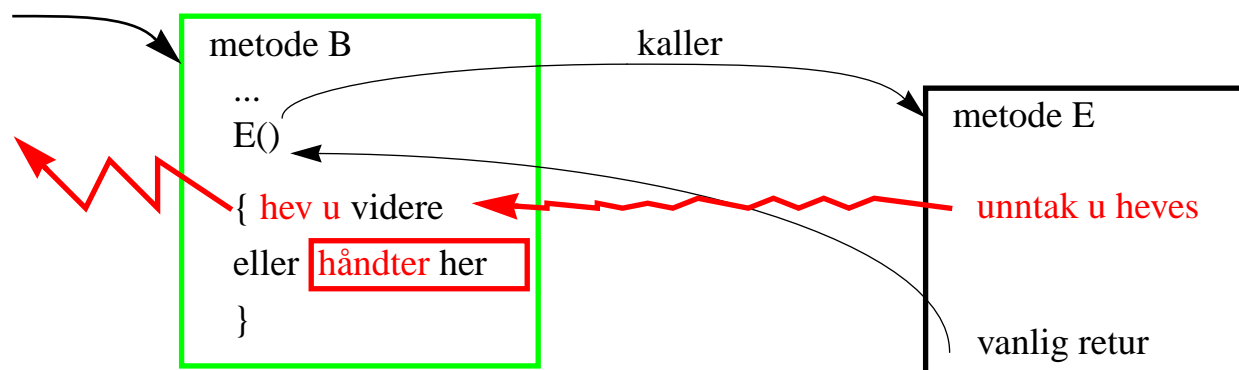
```
public class Stab IMPLEMENTS STACK {  
    ...  
    public Object peek() throws EmptyStackExc  
    { if (empty()) throw new EmptyStackExc("tom stabel");  
      else return elems[antall];  
    }  
    public Object pop() throws EmptyStackExc  
    { if (empty()) throw new EmptyStackExc("tom stabel");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

# Unntakshåndtering

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

1. Unntaket kan fanges **catch**'es, eller

2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul

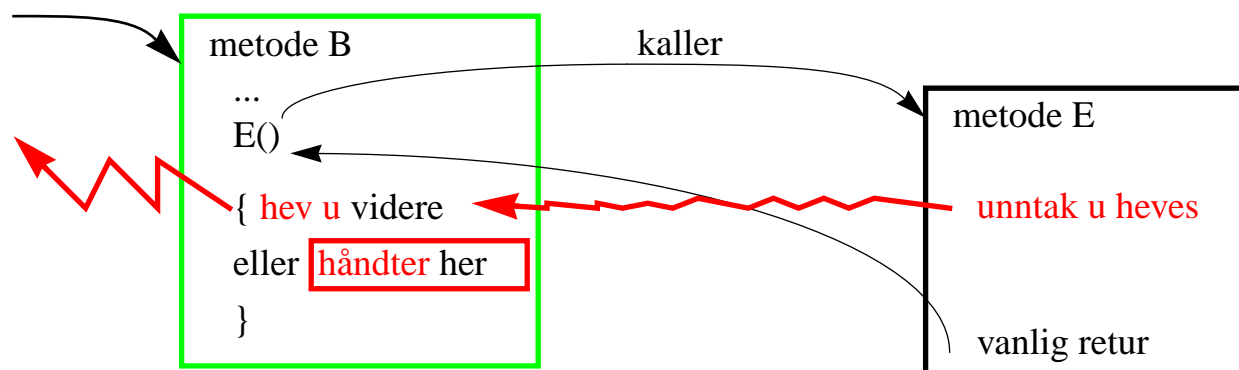


```
public class Stab IMPLEMENTS STACK {  
    ...  
    public Object peek() throws EmptyStackExc  
    { if (empty()) throw new EmptyStackExc("tom stabel");  
      else return elems[antall];  
    }  
    public Object pop() throws EmptyStackExc  
    { if (empty()) throw new EmptyStackExc("tom stabel");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

# Unntakshåndtering

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

1. Unntaket kan fanges **catch**'es, eller
2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul



```
public class B {  
  Stack S = new Stab() ;  
  public Object doS()  
    throws EmptyStackExc  
  { ...  
    Object o = S.peek();  
    ... }  
}
```

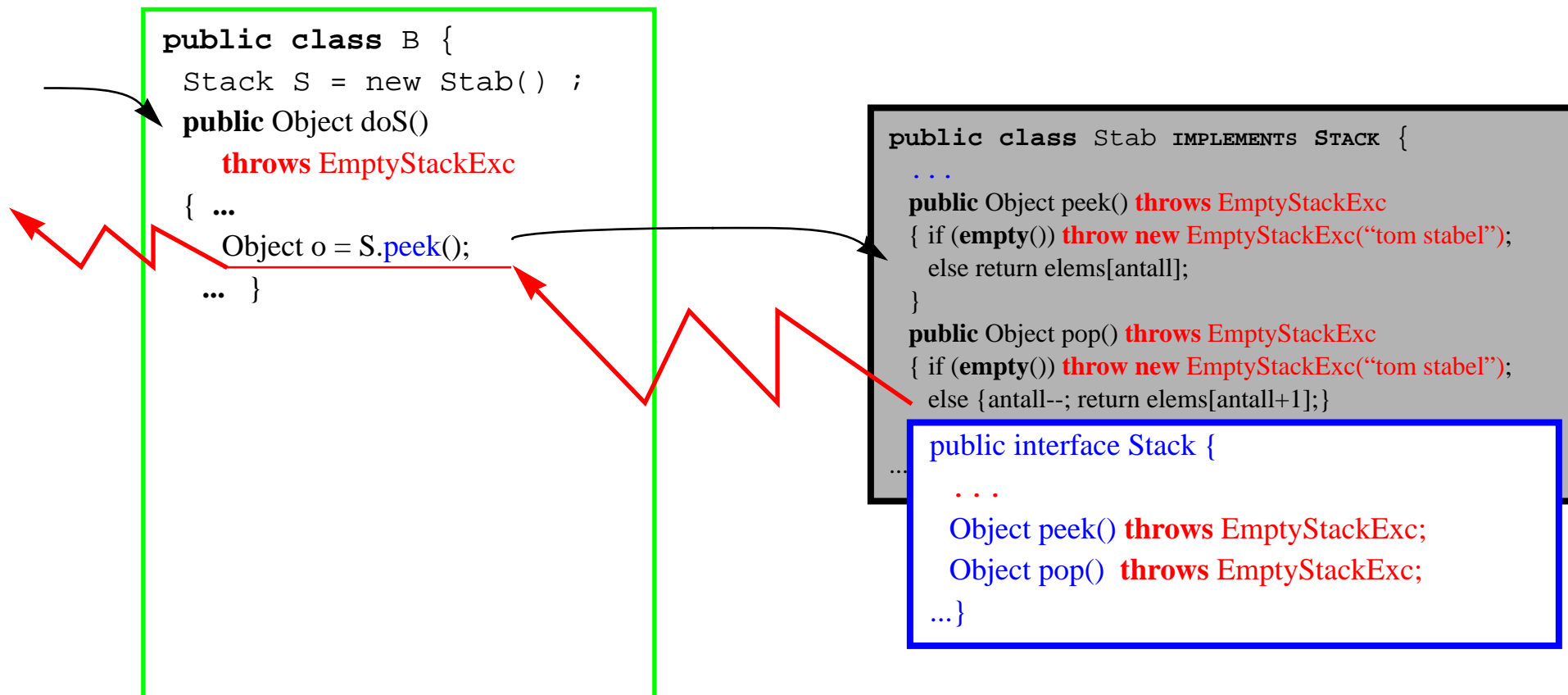
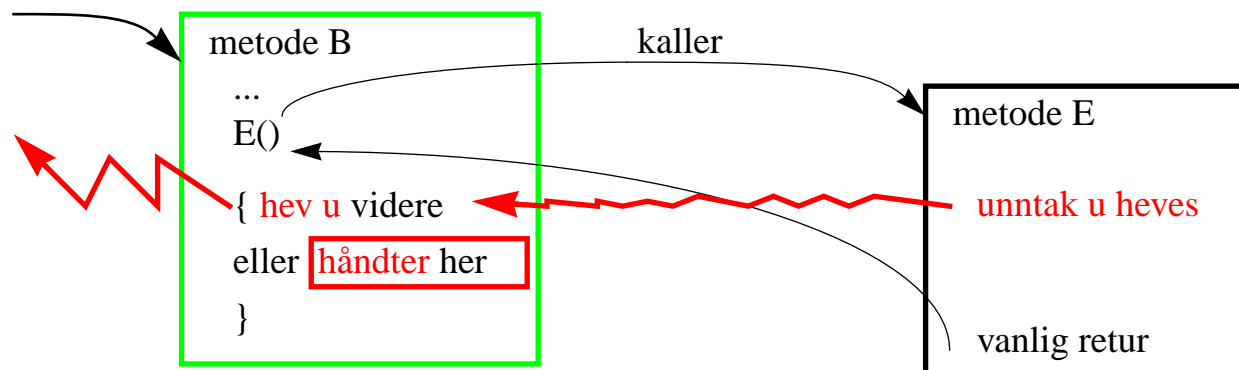
```
public class Stab IMPLEMENTS STACK {  
  ...  
  public Object peek() throws EmptyStackExc  
  { if (empty()) throw new EmptyStackExc("tom stabel");  
    else return elems[antall];  
  }  
  public Object pop() throws EmptyStackExc  
  { if (empty()) throw new EmptyStackExc("tom stabel");  
   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

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

# Unntakshåndtering

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

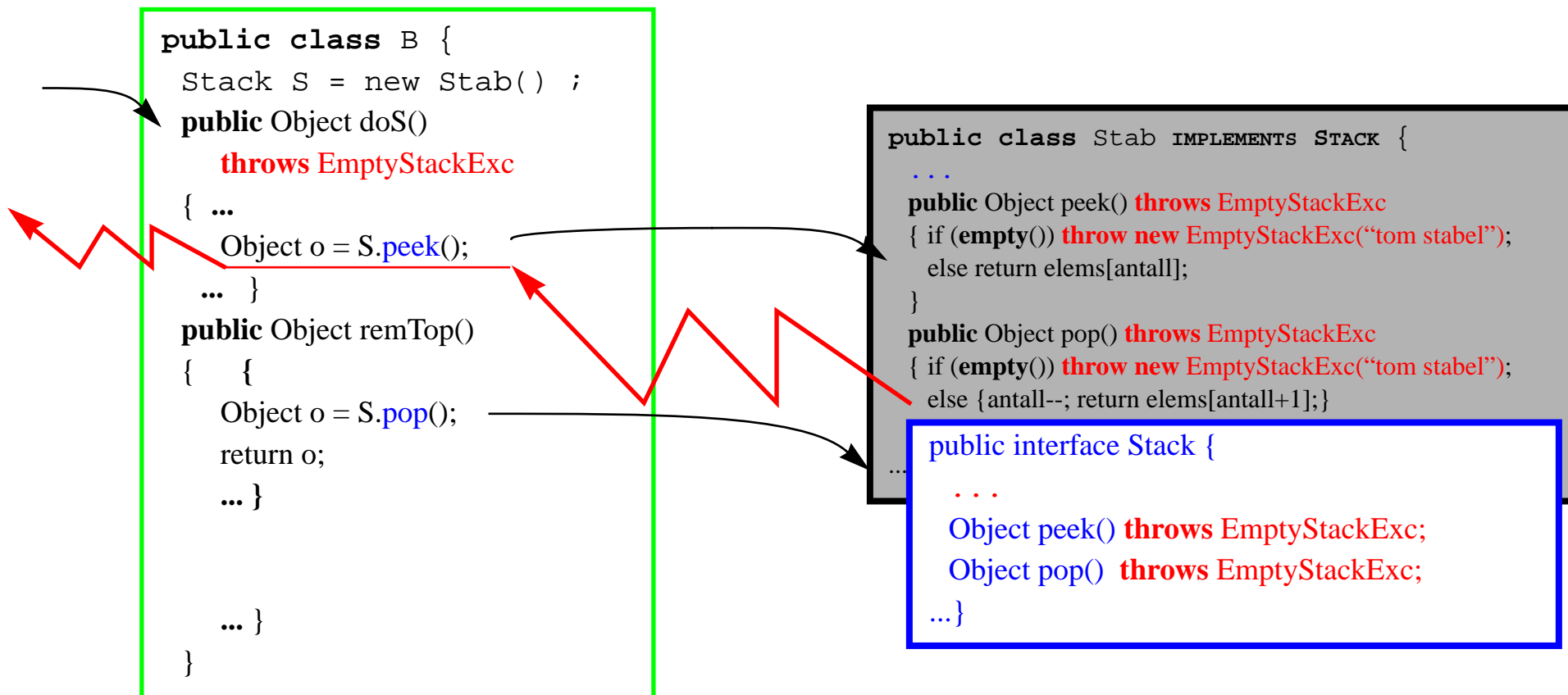
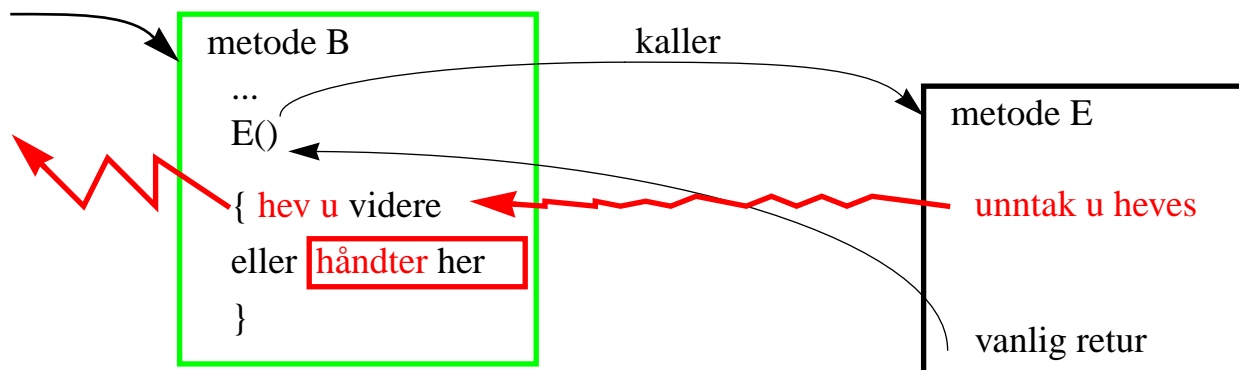
1. Unntaket kan fanges **catch**'es, eller
2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul



# Unntakshåndtering

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

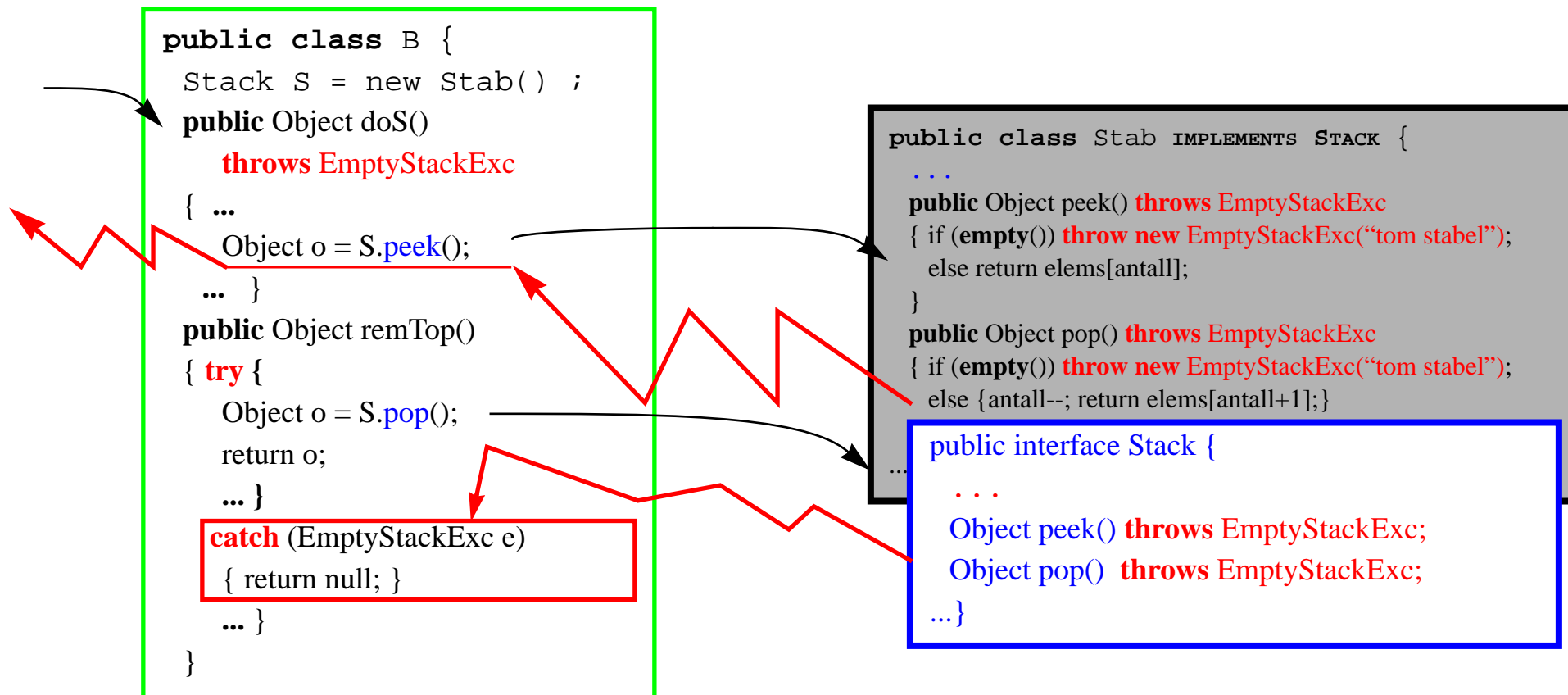
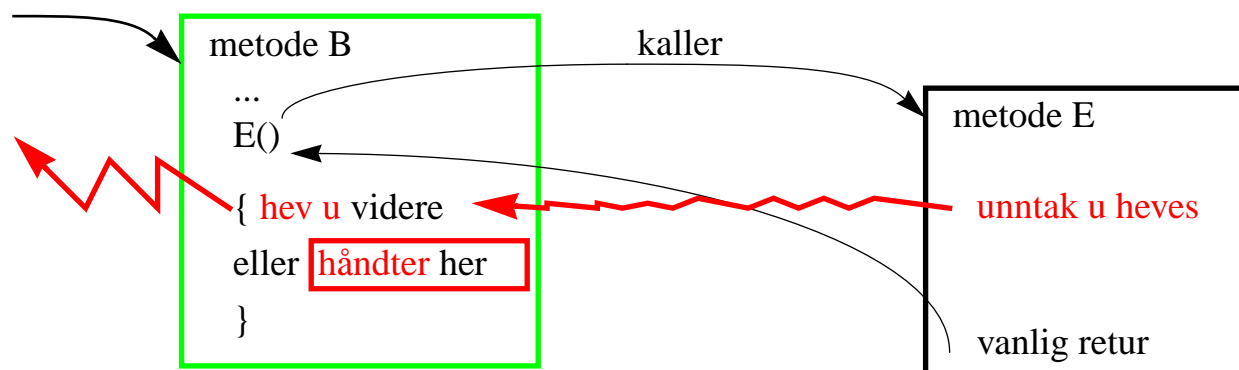
1. Unntaket kan fanges **catch**'es, eller
2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul



# Unntakshåndtering

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

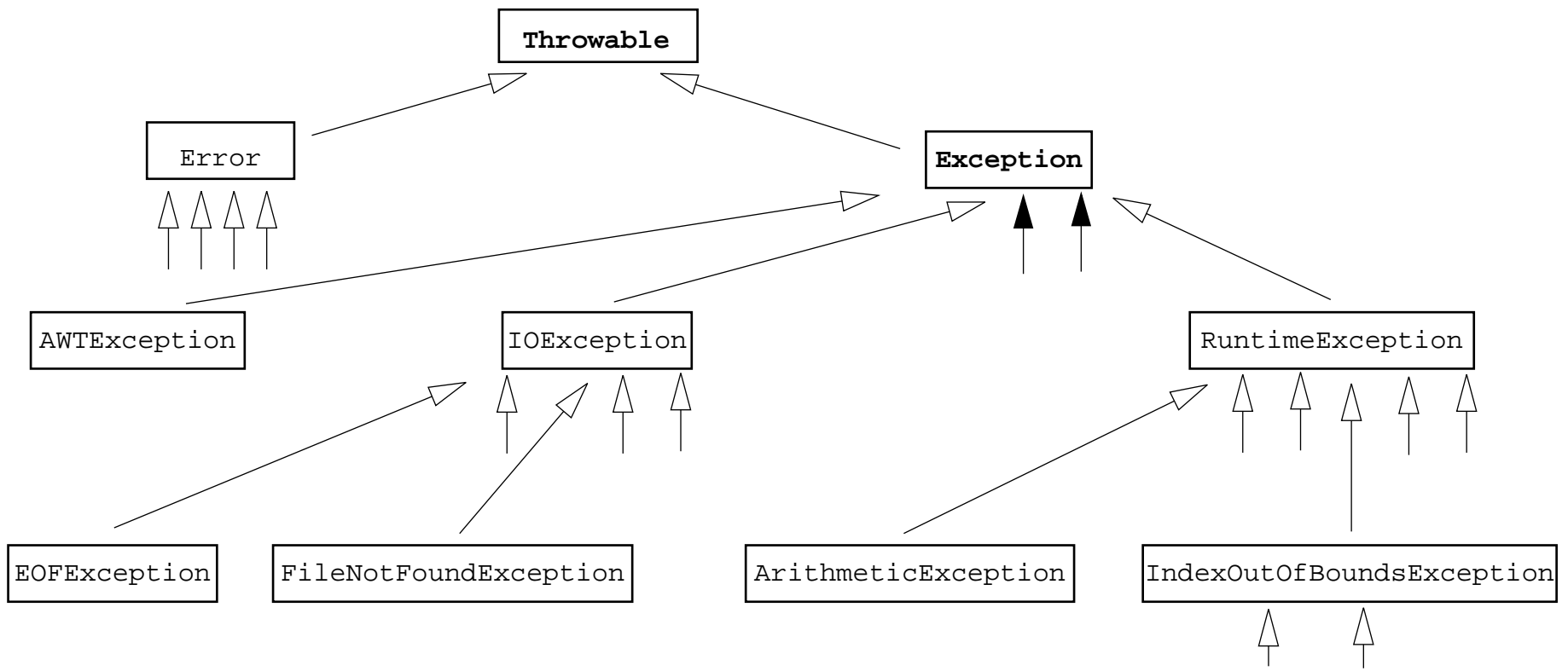
1. Unntaket kan fanges **catch**'es, eller
2. Et unntak som ikke **catch**'es må deklarereres i **throw**-klausul



# Unntaks-hierarki

- Unntak er objekter av klasser utledet fra klassen **Throwable**
- 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 multippel typing*
  - *et interface kan utvide flere andre,*
  - *en klasse kan implementere flere interface*

# 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 multippel typing*
  - *et interface kan utvide flere andre,*
  - *en klasse kan implementere flere interface*

## **Implementasjon av interface**

- *av alle grensesnitt metoder – iht. 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*
- *implementeres 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)*

# 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 multippel typing*
  - *et interface kan utvide flere andre,*
  - *en klasse kan implementere flere interface*

## **Implementasjon av interface**

- *av alle grensesnitt metoder – iht. 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*
- *implementeres 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)*

## **OO og arv**

- *ingen multippel arv*
  - *arv av type og kode*
- *synlighetsbegrensninger*
- *overskriving vs. overlasting*
  - *dynamisk vs. statisk binding*
- *abstrakte klasser*

# Det finnes ingenting

(ingen konkrete program)

som kan gjøres med **interface**

men som ikke kan gjøres uten

# Det finnes ingenting

(ingen konkrete program)

som kan gjøres med bruk av **unntak**

men som ikke kan gjøres uten

# Det finnes ingenting

(ingen konkrete program)

som kan gjøres med **typer/omstøping**

men som ikke kan gjøres uten

# Det finnes ingenting

(ingen konkrete program)

som kan gjøres med **klasser/hierarki**

men som ikke kan gjøres uten

# Det finnes ingenting

(ingen konkrete program)

som kan gjøres **Objekt-Orientert**

men som ikke kan gjøres uten objekter



# Det finnes ingenting

(ingen konkrete program)

som kan **programmeres**

men som ikke kan gjøres med bare

```
0011010101000101010100000111010110001  
0100001010101010010101001001001001010001  
0100010010 1101010....
```