

# Klasser og nedarvning

Hvad er formålet?

Typer, generisk kode, typeparameterisering

Kritisk kig på, hvordan man gør i Java.

---

Eftermiddagens opgave: Designe, implementere, diskutere klassehierarki

## Formål klasser og nedarvning (bl.a.):

- at kunne genbruge kode (generisk, “parameteriseret”)
- at beskytte data mod uautoriseret tilgang til repræsentation
  - uautoriserede ændringer => ødelægger (måske) korrekthed
    - Hvis repræsentation ændres duer program ikke :(
  - uautoriserede læsninger => ødelægger modularitet:  
Hvis repræsentation ændres duer program ikke :(
- sikkerhed: compiler finder så mange fejl som muligt,
  - f.eks. forkert operationer på forkerte slags data,
    - forkerte slags data i forkerte slags variable)

## Generisk kode (kode = klasser/metoder/datatyper...):

- kode som kan anvendes på forskellige (men men “beslægtede”) datatyper
- kræver kraftige *parameteriseringsmekanismer*  
eller *nedarvning*

## Eksempler på, hvad vi ikke kan i Java:

```
void sortér(type T)( T [] array data, boolean ≤(T x, T y) ) // T er en typeparameter  
{ ..... }
```

```
a = sortér(int)(new Integer [] {12,87,435,2,-17,76})
```

```
class Set_of(type T) // T er en typeparameter  
{ public void insert(T) {.....}  
  public T get {...}  
}
```

```
s Set_of(Set_of(BlaBla)) = new Set_of(Set_of(BlaBla))
```

Hvis vi kunne noget-i-den-retning i Java undgik vi:

- at benytte den generelle klasse Object
- explicit definitioner af specialiserede klasser hele tiden og så igen!

## Generelle/generiske “ting” (klasser, metoder i klasser, interface) i Java:

- enhver klasse/metode som ikke er final
- en abstrakt klasse (pr. def klasse med abstrakte metoder)
- interface (en slags totalt abstrakt klasse definition)

## Brug af generelle/generiske “ting”:

- specialisering med nye eller andre egenskaber
- Nedrivning  $\approx$  med mindre andet er nævnt, overtager specialiseringen egenskaber fra det generelle

## Generel syntax i Java

```
class KlasseNavn {egenskaber}
class KlasseNavn extends Navn1 {noget nye egenskaber}
interface InterFaceNavn {navne-på-egenskaber}
class KlasseNavn implements InterFaceNavn1, InterFaceNavn2, ... {udfylde egenskaber}
```

**Princip:** Erklæring af klasse eller interface skaber ny “type”  $\approx$  referencer til objekter i klasse, værdier (= referencer til objekter) skabes ved new

**Princip:** specialisering skaber undertyper. Hvor T kan bruges kan undertype of T også!

**Designproblemet i Java: simple typer og klasser er ikke helt kompatible (kommer vi til)**

## Til design af sprog (som Java) med klasser og nedarvn. hører konventioner for/syntax for:

- hvilke navne kan ses hvor ( $\approx$  hvilke egenskaber kan refereres)
  - o når et objekt benyttes
  - o når en klasse specialiseres
- eller omvendt, når der står “x” i en programtekst, hvilken (om nogen) egenskab refereres til?
- hvordan løses navnekonflikter (flere ting kaldet “x”)
- hvor stor grad af overloading tillades?

## Hvad er et “navn” Java?

navn på variabel eller klasse

signatur: metode-navn(parameter-type 1, ... )

obs: resultattype ikke en del af signaturen

## Java benytter signatur til at identificere metode

```
int f(); {return 5} og float f(); {return 7.7} har samme signatur
```

```
float f(float x) har en anden signatur
```

Reference til “f” afgøres så vidt mulig statisk; hvis flere lige gode muligheder afgøres det dynamisk.

## (U)synlighed kontrolleres i Java ved:

static, final, abstract, public, protected, ... (læs selv hvis I ikke kan dem allerede)

## Særligt problem ved multipel nedarvning

Følgende kan man ikke i Java:

```
class A {public int f(); {return 5} ...};  
class B {public int f(); {return 7} ; ...}  
class C extends A,B {...}  
new A,B().f() == ??? // hvad for en f???
```

Problem kunne løses med ekstra syntaks “bla.f of A()” eller “bla.f of B()”

Noget der minder om multipel nedarvning kan opnås ved at implementere flere interfaces:

- der må ikke være overlappende signaturer med forskellig resultat-type i forskellige interfaces som kombineres
- class AB extend IF1, IF2 {... kun én metode for hver signatur+resultat}

**... resten af forelæsning: lidt mere detaljer om hvordan det ser ud i Java.**

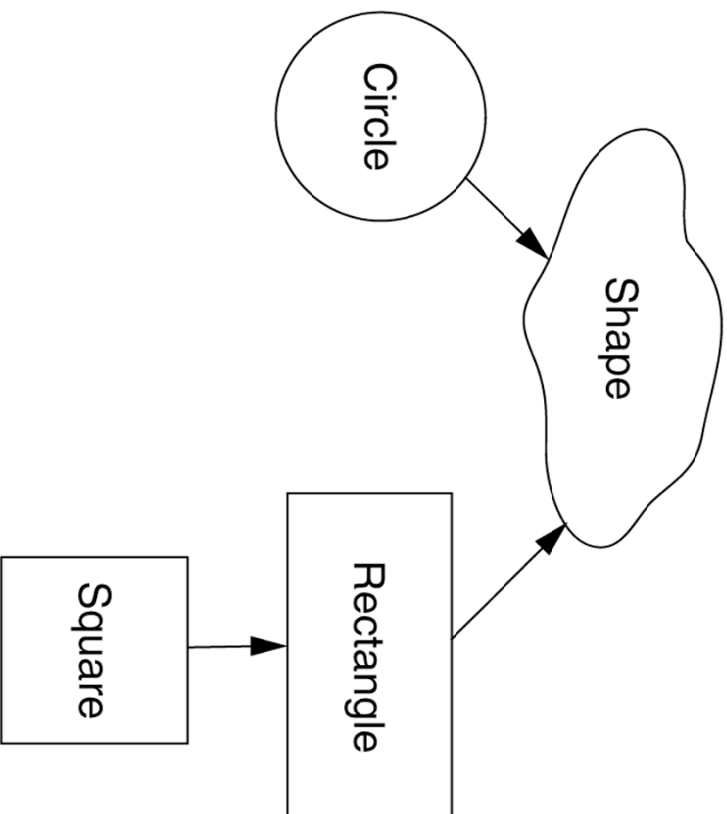
## Eksempel på overloading (konstruktører)

```
public class Point {  
    private double x, y;  
  
    public Point() {  
        x = 0.0; y = 0.0;  
    }  
  
    public Point(double x, double y) {  
        this.x = x; this.y = y;  
    }  
  
    ...  
}
```

## Eksempel på overloading; metoder

```
public class Point {  
    private double x, y;  
    // ...  
  
    public double distance(Point other) {  
        double dx = this.x - other.x, dy = this.y - other.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
  
    public double distance(double x, double y) {  
        double dx = this.x - x, dy = this.y - y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

## Eksempel på nedarving



```
public abstract class Shape
{
    public abstract double area();
    public abstract double perimeter();
    public double semiperimeter() {return perimeter() / 2; }}
```

```

public class Circle extends Shape
{
    public Circle( double rad ) {radius = rad;}
    public double area( ) {return Math.PI * radius * radius;}
    public double perimeter( ) { return 2 * Math.PI * radius;}
    public String toString( ) { return "Circle: " + radius;}
    private double radius;
}

```

```

public class Rectangle extends Shape
{
    public Rectangle( double len, double wid ){length = len; width = wid;}
    public double area( ){return length * width;}
    public double perimeter( ){ return 2 * ( length + width );}
    public String toString( ){return "Rectangle: " + length + " " + width;}
    public double getLength( ){return length;}
    public double getWidth( ) { return width;}
    private double length;
    private double width;
}

```

```

public class Square extends Rectangle
{
    public Square( double side ){super( side, side );}
    public String toString( ){return "Square: " + getLength();}
}

```

## Klassen **Object** (std. Java),

Object er overklasse for alle andre klasser

```
public class Object {  
    public Object();  
    public String toString();  
    public boolean equals(Object obj);  
    public int hashCode();  
    ... }  
}
```

På grund af manglende typeparameterisering, benyttes **Object** til generiske klasser/metoder  
Prisen: Typedisciplinen sættes til dels over styr!

## Eksempel: ArrayList (std. Java klasse)

```
public class SimpleArrayList
{
    public SimpleArrayList() {clear();}
    public int size() {return theSize;}
    public Object get( int idx )
    { if( idx < 0 || idx >= size() )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + size() );
      return theItems[ idx ]; }
    public Object set( int idx, Object newVal )
    { if( idx < 0 || idx >= size() )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + size() );
      Object old = theItems[ idx ]; theItems[ idx ] = new Val; return old; }
    public boolean add( Object x )
    { if( theItems.length == size() )
      { Object [ ] old = theItems;
        theItems = new Object[ theItems.length * 2 + 1 ];
        for( int i = 0; i < size(); i++ ) theItems[ i ] = old[ i ];}
      theItems[ theSize++ ] = x; return true; }
    public Object remove( int idx )
    { Object removedItem = theItems[ idx ];
      for( int i = idx; i < size() - 1; i++ ) theItems[ i ] = theItems[ i + 1 ];
      theSize--; return removedItem; }
}
```

```

public void clear() { theSize = 0; theItems = new Object[ DEFAULT_CAPACITY ];}
private static final int DEFAULT_CAPACITY = 10;
private static final int NOT_FOUND = -1;
private int theSize;
private Object [] theItems; }

```

### Det store problem når vi bruger ArrayList:

- vi kan ikke specialisere til versioner som begrænser hvilken undertype Object'er listen en given ArrayList indeholder
- dvs. dynamisk typecheck, massevis af "type casting"
- dvs. større risiko for updagede fejl, som først opdages på runtime!!!

**Nok et problem:** Simple værdier er ikke klasser, derfor behov for "wrapper classes" (std. Java)

```

public final class Integer {
    public Integer(int v) { value = v;}
    public int intValue() {return value;}
    .... diverse andre gode ting for heltal
    private int value; }

```

**Andre bøjede søm i form af standardklasser for at kompensere for manglende typeparameterisering:**

```
public interface Comparable {int compareTo( Object other); }  
    // =0 hvis ens; positiv hvis “selv” < other; negativ ellers
```

**Eksempel:** Antag vi skriver en klasser (ikke std. java.util):

```
public class ArrayList_with_sorting extends ArrayList;  
    { public void sort(); { ... må typecaste Object'er til Comparable } }  
  
public class Shape extends Comparable;  
    { ... som tidligere Shape men med en compareTo metode ... }
```

Korrekt brug:

```
a = new ArrayList_with_sorting(); a.add(new triangle(...)) ; a.add(new square(...));a.sort;
```

Men runtime-fejl i tilfælde af:

- der indsettes et Object som ikke er i underklasse af Comparable
- der indsettes et Object som er i underklasse af Comparable men ikke sammenlignelig med Shape.

Beslægtet std. java klasse til subtile tilfælde hvor Comparable ikke fungerer:  
**public interface Comparator** { int compare( Object lhs, Object rhs );}

Læs også i bogen om “Anonymous Classes” – som gør det lidt nemmere at skrive generisk kode.

---

Programmering i Java: Kræver grundig debuging og disciplin for metodisk brug af nedrivningsmekanismene

---

Eftermiddagens opgave: Designe, implementere, diskutere klassehierarki