

Denne forelæsning optages og gøres muligvis  
efterfølgende tilgængelig på Moodle  
MEDDEL VENLIGST UNDERVISEREN, HVIS  
DU IKKE ØNSKER, AT OPTAGELSE FINDER STED

This lecture will be recorded and afterwards may be  
made available on Moodle  
PLEASE INFORM THE LECTURER IF YOU DO NOT WANT  
RECORDING TO TAKE PLACE

# OOP 2020

Nedarvning

Thomas Bøgholm

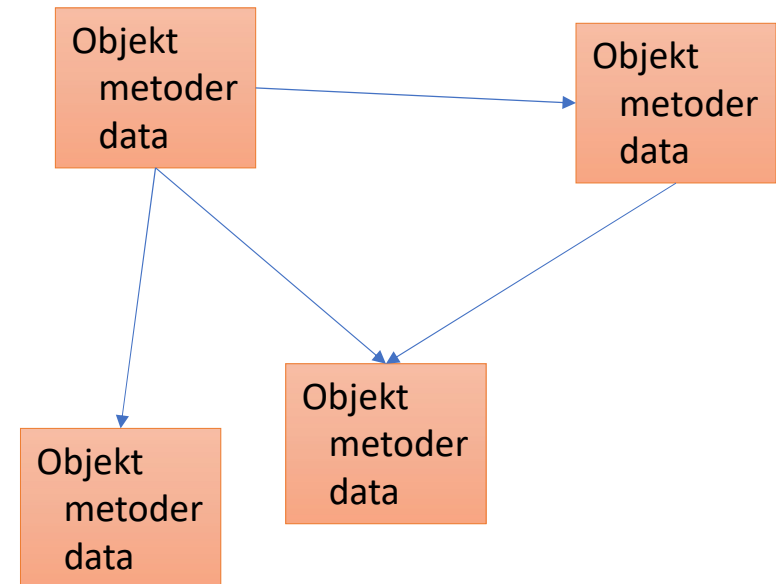
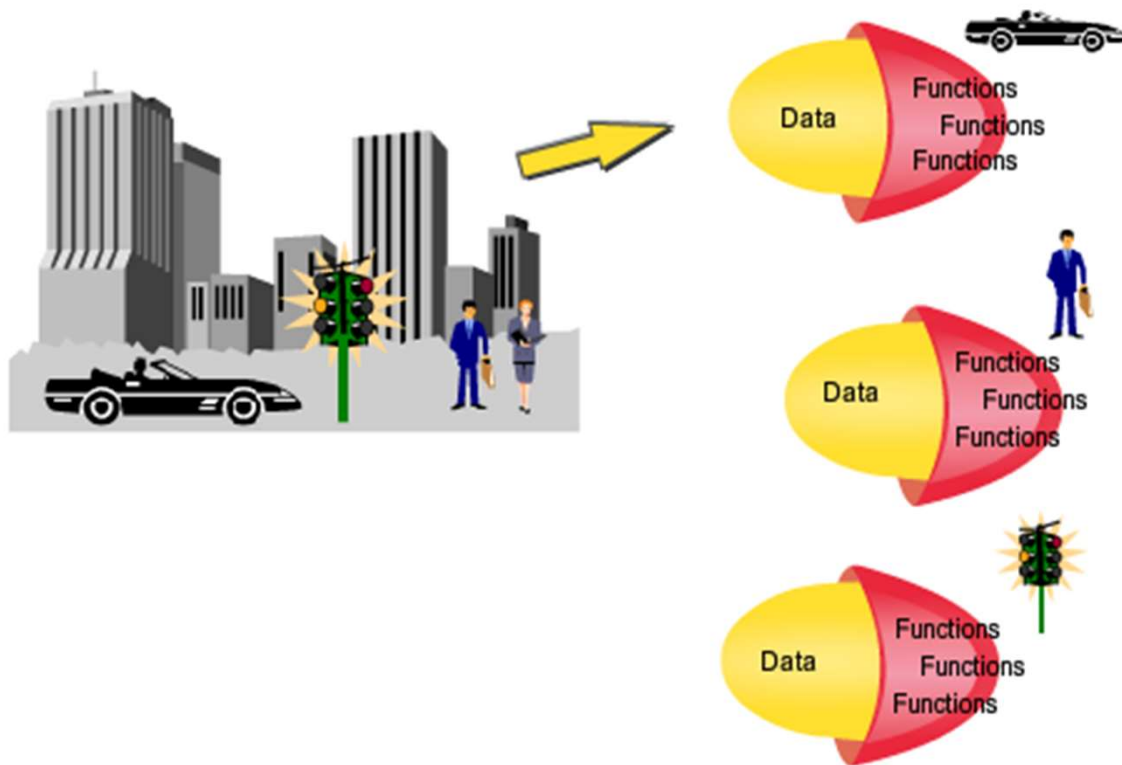
# Plan

- Opfølgning på opgaver
- Opfølgning på materiale fra sidste gang
  - Kræves yderligere forklaring eller mangler noget?
  - Properties? (get/set)
- Dagens emne: Nedarvning og polymorfi

# Grundsten i OOP

- Abstraktion
  - Gem alle irrelevante detaljer
- Indkapsling
  - Data og opførsel i en kapsel
- Nedarvning
  - Kodegenbrug
- Polymorfi
  - Objekter kan tage flere former

Resten foregår i Visual Studio  
- resten omhandler dagens emne

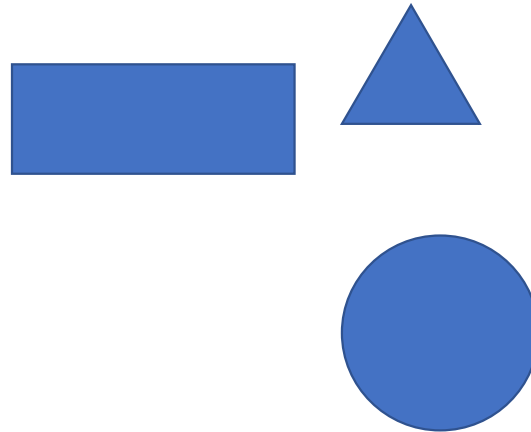


# Grundsten i OOP

- Abstraktion
  - Gem alle irrelevante detaljer
- Indkapsling
  - Data og opførsel i en kapsel
- Nedarvning
  - Kodegenbrug
- Polymorfi
  - Objekter kan tage flere former

# Firgur-eksempel

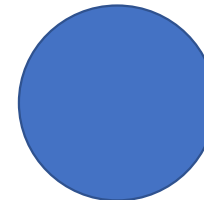
- Tre klasser beskriver disse figurer
  - Trekant
  - Firkant
  - Cirkel





# Klasser for figurer

- Tre klasser beskriver disse figurer
  - Trekant
  - Firkant
  - Cirkel



```
class Trekant
{
    public Trekant(int højde, int grundlinie){..}
    private double orientation;
    private int posX, posY;
    public void SetPosition(int x, int y) { }
    public void Rotate() {}
    public void Scale(double factor) {}

    public double Area() {}
}
```

# Andre figurers klasser?

Copy/paste!?

```
class Trekant
{
    Trekant(int h, int g){..}
    double orientation;
    int posX, posY;
    void SetPosition(...) { }
    void Rotate() { }
    void Scale(..factor) { }

    double Area() {return h*g/2;}
}
```

```
class Rektangel
{
    Rektangel(int l, int b){..}
    double orientation;
    int posX, posY;
    void SetPosition(...) { }
    void Rotate() { }
    void Scale(.. factor) { }

    double Area() { return l*b;}
}
```

```
class Cirkel
{
    Cirkel(int radius) {..}
    double orientation;
    int posX, posY;
    void SetPosition(...) { }
    void Rotate() { }
    void Scale(.. factor) { }

    double Area() { return PI*r*3;}
}
```

# Kodegenbrug!

- **Nedarvning!**
  - Samle generel kode, og genbruge det hvor det giver mening!
  - Introducér generel **Figur**-klasse

```
class Figur
{
    double orientation;
    int posX, posY;
    void SetPosition(int x, int y) { }
    void Scale(double factor) { }
    void Rotate() { }

    double Area() { }
}
```

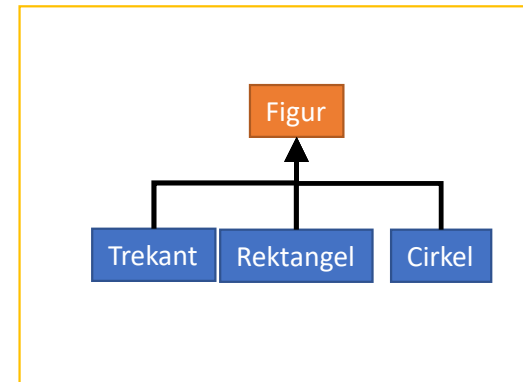
# Kodegenbrug!

- **Nedarvning!**

- Samle generel kode, og genbruge det hvor det giver mening!
- Introducér generel **Figur**-klasse

```
class Figur
{
    double orientation;
    int posX, posY;
    void SetPosition(int x, int y) { }
    void Scale(double factor) { }
    void Rotate() { }

    double Area() { }
}
```



# Klassehierarki og nedarvning i C#

```
class Figur
{
    double orientation;
    int posX, posY;
    void SetPosition(int x, int y) { }
    void Scale(double factor) { }
    void Rotate() { }

    double Area()
}
```

Bemærk! Detaljer kommer senere!  
- ingen accessmodifiers endnu

Specialisering af Figur

```
class Rektangel : Figur
{
    Rektangel(int l, int b){..}

    double Area() { }
}
class Cirkel : Figur
{
    Cirkel(int radius) {..}

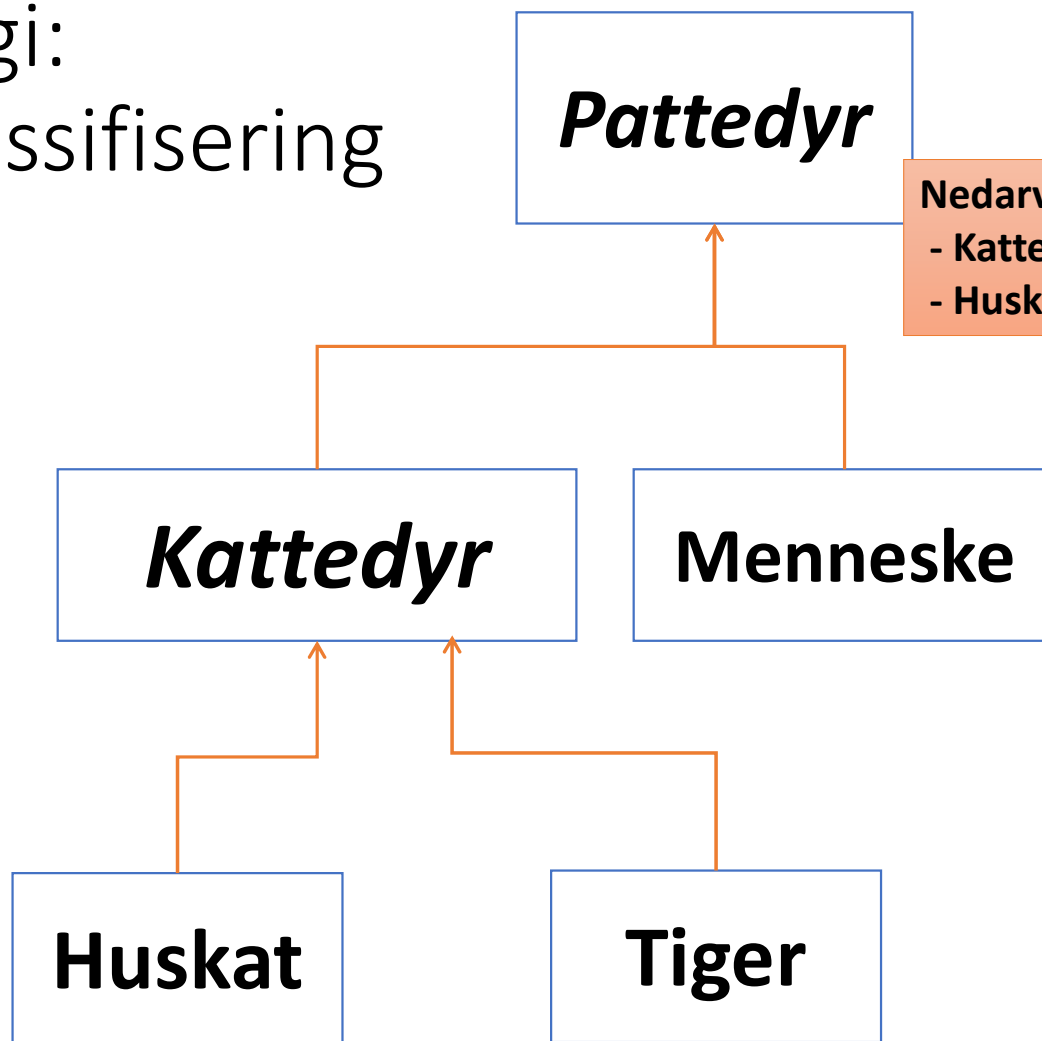
    double Area() { }
}
class Trekant : Figur
{
    Trekant(int h, int g){..}

    double Area() {}
}
```

# Nedarvnings-definitioner

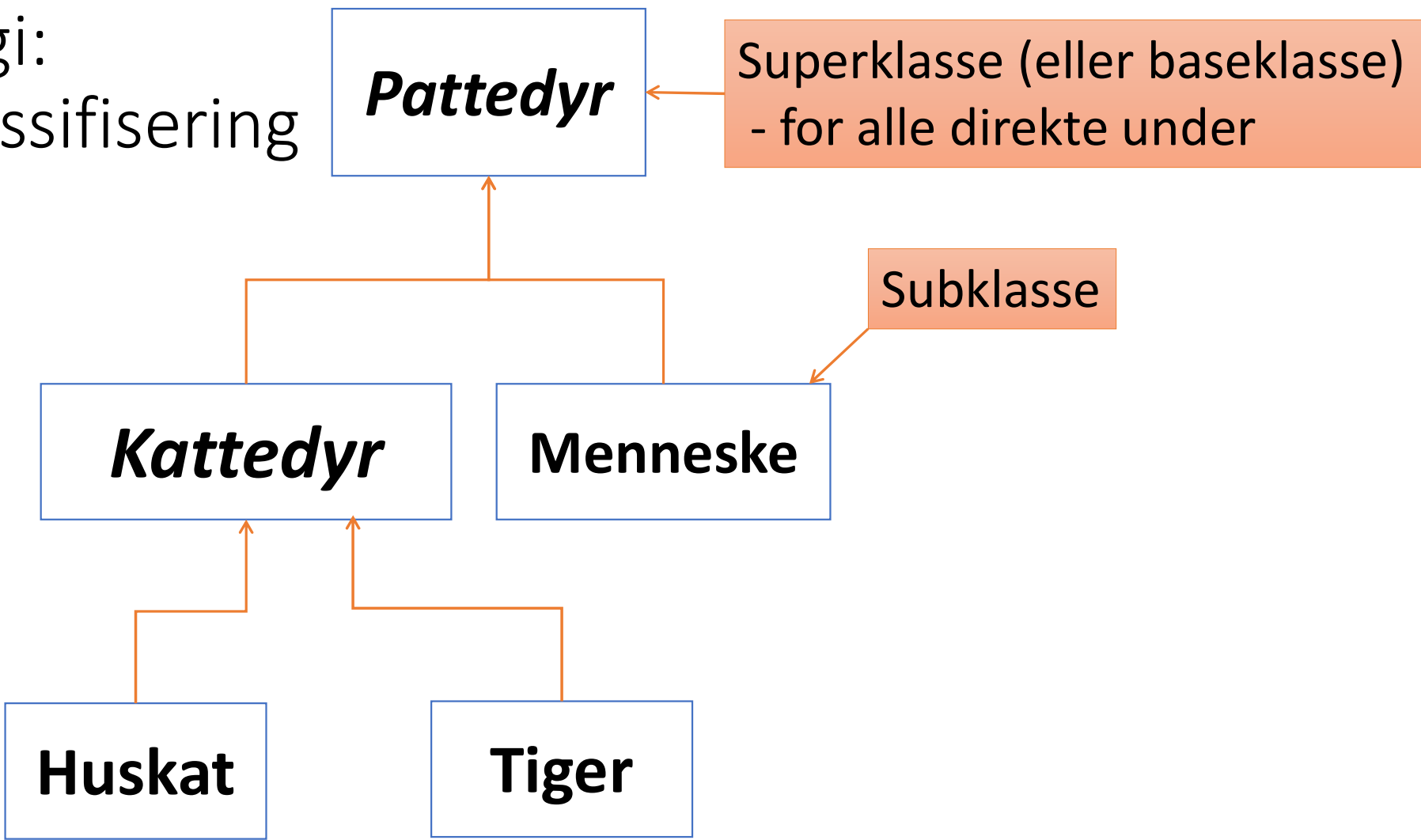
- **Nedarvning:** Beskriver “er-også-en” forhold mellem to typer.
- **Superklasse (type) / Base Type /Parent Type**
- **Subklasse (type) / Afledt Type /Child Type**
- Supertype: **generalisering** af subtyper.
- Subtyper: **specialisering** af supertype.
- Ide:
  - Fælles implementation i superklasse.
  - Specialisering i subklasser.
- -> Genbrug og dermed vedligeholdelse

# Analogi: art-klassifisering



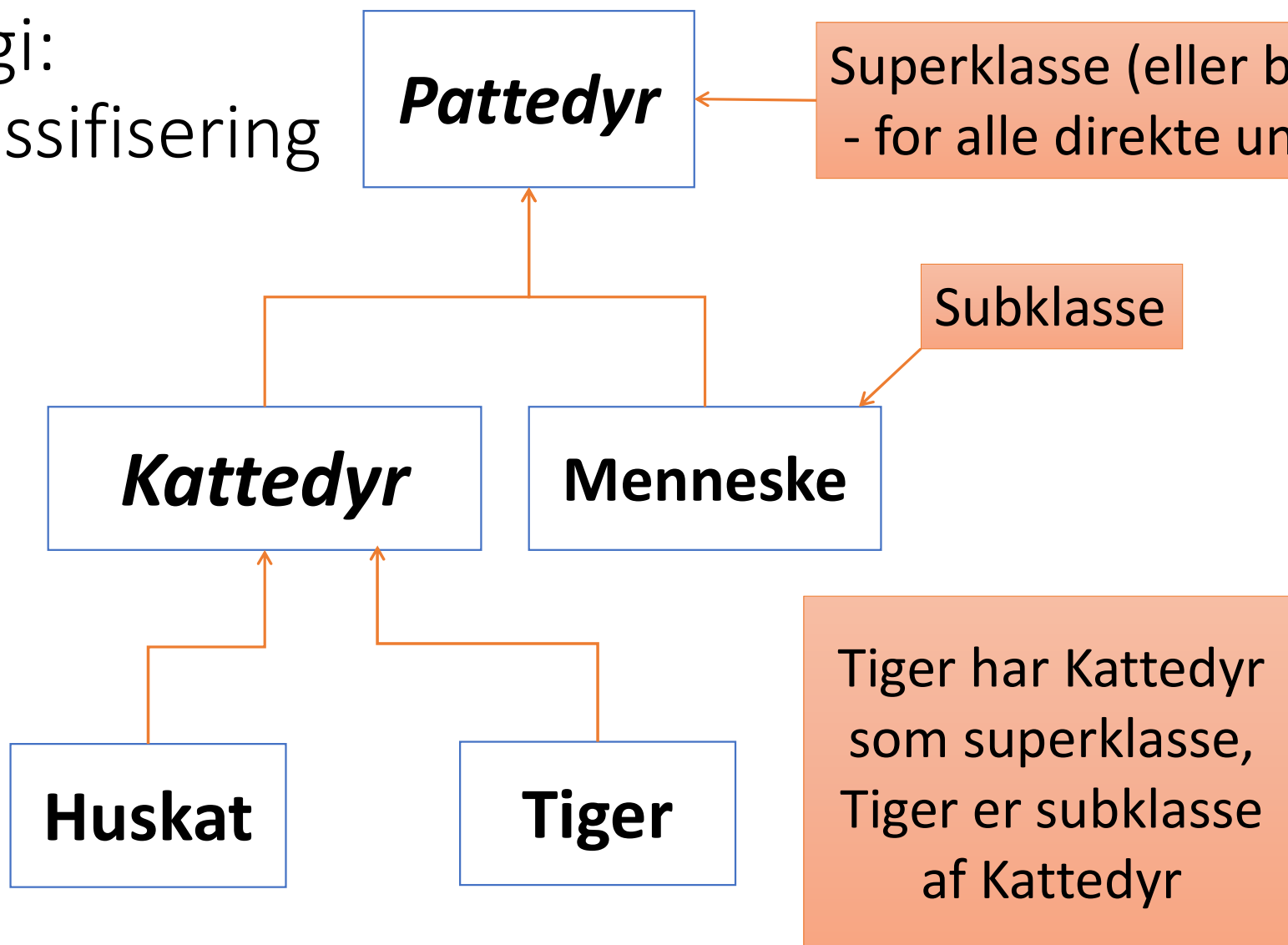
Nedarvning beskriver forholdet: "Er-en"(Is-a)  
- Kattedyr er pattedyr, mennesker er pattedyr  
- Huskat og Tiger er Kattedyr OG PATTEDYR

Analogi:  
art-klassifisering

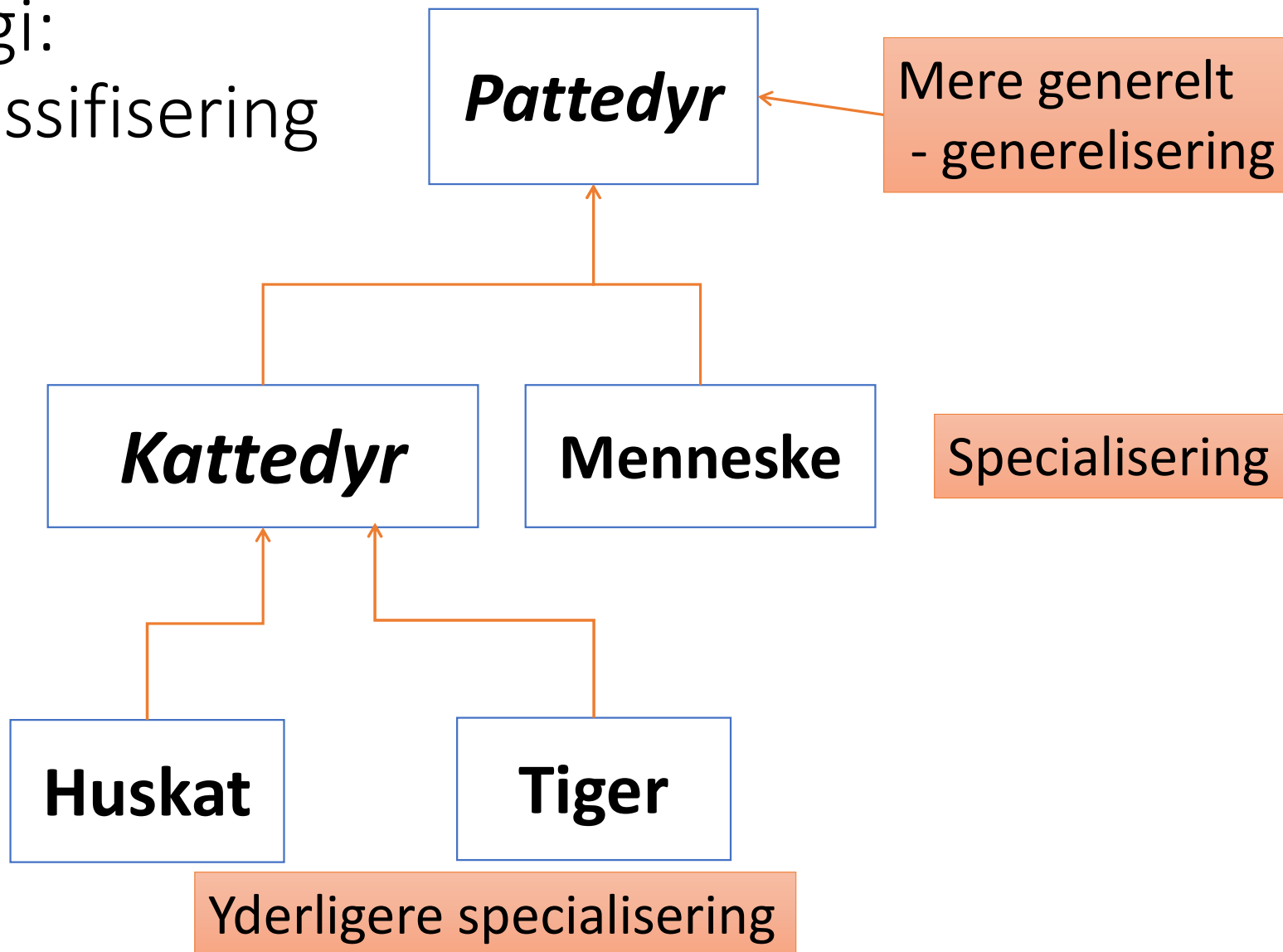




Analogi:  
art-klassifisering



Analogi:  
art-klassifisering



# Bruges som almindelige klasser

```
static void Main(string[] args)
{
    double radius, højde, grundlinie, længde, bredde;
    Cirkel c = new Cirkel(radius);
    Trekant t = new Trekant(højde, grundlinie);
    Figur f = new Rektangel(længde, bredde);

    List<Figur> figurer = new List<Figur>();
    figurer.Add(c);
    figurer.Add(t);
    figurer.Add(f);
    figurer.Add(new Cirkel(100.0));
}
```

# Gensyn med adgangsmodifikatorer

- **Synligheder:**
  - **public** medlemmer: Kan tilgås fra alle klasser.
  - **private** medlemmer: Kan ikke tilgås fra andre klasser
  - **protected** medlemmer: Kan kun tilgås fra klassen og afledte typer

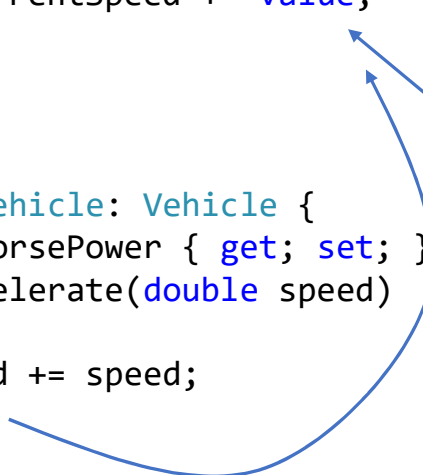
## Brug af **protected** access modifier

```
public class Vehicle {
    private double _currentSpeed; //kun synlig i klassen - ikke subklasser

    public double TopSpeed { get; set; } //synlig alle steder

    public double CurrentSpeed {
        get { return _currentSpeed; } //getter synlig overalt
        protected set { //setter kun synlig subklasser
            if (_currentSpeed + value < TopSpeed)
                _currentSpeed += value;
        }
    }
}

public class MotorVehicle: Vehicle {
    public double HorsePower { get; set; }
    public void Accelerate(double speed)
    {
        CurrentSpeed += speed;
    }
}
```



```
static void Main(string[] args)
{
    Vehicle v = new Vehicle();
    v.TopSpeed = 75;
    //setter er ikke tilgængelig
    //v.CurrentSpeed = 50;

    MotorVehicle mv =
        new MotorVehicle();
    mv.TopSpeed = 90;
    mv.Accelerate(35);
}
```

# Constructors og nedarvning

- Constructors bliver **ikke** tilgængelige i subklasser!
  - Constructors skal derfor (re)defineres for subklasser
- En subklasse **skal eksplicit kalde** en superklasse-constructor
  - Kun hvis superklassen ikke har en **tom constructor**
  - Superklasse-constructors kaldes med **base**-keyword
    - **Først member initialisering**
    - **Superklassens constructor udføres**
    - **Aktuel constructor til sidst**
  - *Constructor chaining*

```
class Konto {  
    public Konto(int balance) {  
        this.Balance = balance;  
    }  
    public int Balance { get; set; }  
}
```

Lovlig- og ulovligheder mht. constructorer

```
class Konto {  
    public Konto(int balance) {  
        this.Balance = balance;  
    }  
    public int Balance { get; set; }  
}
```

Lovlig- og ulovligheder mht. constructorer

```
class FirmaKonto : Konto { }
```

← ULOVLIGT: mangler construtor

```
class Firmakonto : Konto {  
    public Firmakonto(int b) { } ← ULOVLIGT: mangler kald til baseklassens constructorer  
}
```



```
class Konto {  
    public Konto(int balance) {  
        this.Balance = balance;  
    }  
    public int Balance { get; set; }  
}
```

Lovlig- og ulovligheder mht. constructorer

```
class FirmaKonto : Konto
```

```
{  
    public FirmaKonto(int b) : base(b)  
    {  
        .....  
    }  
}
```



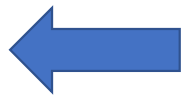
Kalder base-constructor, så alt er fint

“B er en A, så skal konstrueres som A”

## Lovlig- og ulovligheder mht. constructorer

```
class EmptyA {  
    ..... ingen constructor eller default constructor .....  
}
```

```
class B : EmptyA {  
    EmptyA(){  
        ...  
    }  
}
```



Behøver ikke base-kald, base-klassens default bliver kaldt

# Polymorfi

- Det andet og vigtigere argument for nedarvning

# Polymorfi

**Polymorfi** giver mulighed for at **overskrive (/ override)** medlemmer fra superklassen.

Polymorfi er et græsk ord, der betyder *mangeformet*

# Polymorfi

**Polymorfi** giver mulighed for at **overskrive (/ override)** medlemmer fra superklassen.

# Polymorfi

- To forskellige aspekter
  - På runtime kan instanser af en subtype ses som en supertype
- Virtual og override
  - Med virtual kan en supertype definere og implementere en metode
  - Med override kan en subtype give en ny implementation

# Polymorfi

1. **(Statisk polymorfi)** (metode overloading).  
**Statisk binding.** Metode-kald afgøres på compile-tidspunktet.
2. **(Dynamisk) polymorfi**  
**Dynamisk binding.** Metode-kald afgøres på runtime-tidspunktet.  
(denne proces kaldes også **dynamic dispatch**)

VIGTIGT

(Dynamisk) Polymorfi:

1. En subtype **er også** supertypen – subtypen kan bruges alle steder hvor supertypen forventes
2. Subtyper kan redefinere medlemmer fra supertypen
  - en implementation kan variere
  - **abstract** og **virtual** keywords

# Statisk polymorfi (ikke static-keyword)

```
class PersonFileCatalog
{
    public List<Person> ReadCatalogFromPath(string filename)
    {
        return ReadCatalogFromFileInfo(new FileInfo(filename));
    }

    public List<Person> ReadCatalogFromFileInfo(FileInfo file)
    {
        List<Person> result;
        Stream s = file.OpenRead();
        result = ReadCatalogFromStream(s);
        s.Close();
        return result;
    }

    public List<Person> ReadCatalogFromStream(Stream s)
    {
        List<Person> result = new List<Person>();
        StreamReader sr = new StreamReader(s);
        string name;
        while(null != (name = sr.ReadLine()))
            result.Add(new Person() {Name = name});
        return result;
    }
}
```

Logik kun skrevet én gang!



# Statisk polymorfi (ikke static-keyword)

```
class PersonFileCatalog
{
    public List<Person> ReadCatalogFromPath(string filename)
    {
        return ReadCatalogFromFileInfo(new FileInfo(filename));
    }

    public List<Person> ReadCatalogFromStream(Stream s)
    {
        List<Person> result;
        Stream sr = s.OpenRead();
        result = ReadCatalogFromFileInfo(sr);
        sr.Close();
        return result;
    }

    public List<Person> ReadCatalogFromStream(Stream s)
    {
        List<Person> result = new List<Person>();
        StreamReader sr = new StreamReader(s);
        string name;
        while(null != (name = sr.ReadLine()))
        {
            result.Add(new Person() {Name = name});
        }
        return result;
    }
}
```

## Metode-overloading

Vi har tidligere set det med constructors

- Det samme gælder metoder
- kendes som statisk polymorfi

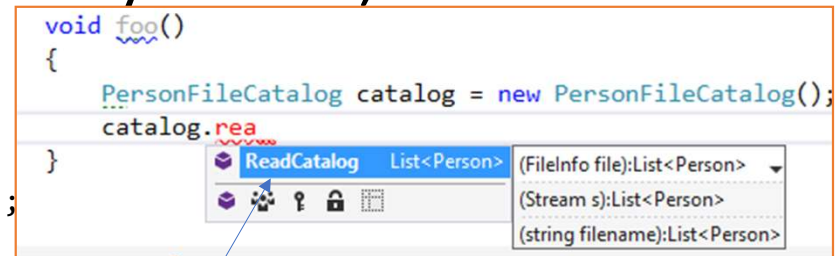
Logik kun skrevet én gang!

# Statisk polymorfi (ikke static-keyword)

```
class PersonFileCatalog
{
    public List<Person> ReadCatalogFromPath(string filename)
    {
        return ReadCatalogFromFileInfo(new FileInfo(filename));
    }

    public List<Person> ReadCatalogFromFileInfo(FileInfo file)
    {
        List<Person> result;
        Stream s = file.OpenRead();
        result = ReadCatalogFromStream(s);
        s.Close();
        return result;
    }

    public List<Person> ReadCatalogFromStream(Stream s)
    {
        List<Person> result = new List<Person>();
        StreamReader sr = new StreamReader(s);
        string name;
        while(null != (name = sr.ReadLine()))
        {
            result.Add(new Person() {Name = name});
        }
        return result;
    }
}
```



Samme navn  
- de gør jo det samme

Hvilken metode der kaldes  
afgøres af typen på parametre  
- og antallet!

Logik kun skrevet én gang!

# Figureksempel - gensyn

```
class Figur
{
    double orientation;
    int posX, posY;
    void SetPosition(int x, int y) { }
    void Rotate() { }
}
```

```
class Firkant : Figur
{
    void Draw() { }
    double Area() { }
}
```

```
class Cirkel : Figur
{
    void Draw() { }
    double Area() { }
}
```

```
class Trekant : Figur
{
    void Draw() {}
    double Area() {}
}
```

# Figureksempel - gensyn

```
abstract class Figur
{
    double orientation;
    int posX, posY;
    void SetPosition(int x, int y) { }
    void Rotate() { }
    abstract void Draw();
    abstract double Area();
}
```

```
class Firkant : Figur
{
    void override Draw() {...}
    double override Area() {...}
}

class Cirkel : Figur
{
    void override Draw() {...}
    double override Area() {...}
}

class Trekant : Figur
{
    void override Draw() {...}
    double override Area() {...}
}
```

# Figureksempel – med polymorfi

```
abstract class Figur
{
    ...
    public abstract void Draw();
    public abstract double Area();
}
```

```
Figur figur1 = new Trekant();
Figur figur2 = new Cirkel();
figur1.Draw();

Figur[] figArray = {new Trekant(),
                    new Rektangel(),
                    new Cirkel() };

foreach (Figur f in figArray)
    f.Draw();
```

```
class Firkant : Figur
{
    void override Draw() {...}
    double override Area() {...}
}

class Cirkel : Figur
{
    void override Draw() {...}
    double override Area() {...}
}

class Trekant : Figur
{
    void override Draw() {...}
    double override Area() {...}
}
```

## Virtuelle og Abstrakte Medlemmer

- **Virtuelt/abstrakt medlem:** Instans-metode/property der kan eller skal *specialiseres/redefineres* i subklasser.
- **virtual:** Medlem **kan** specialiseres i subklasser.
- **abstract:** Medlem **skal** specialiseres i subklasser. Kan kun defineres i *abstrakt klasse*.

# Override

- **override:** Medlem redefinerer/specialiserer virtuelt medlem i superklasse og **kan** selv yderligere specialiseres
  - (sjældent: kan kombineres med *abstract* til at tvinge subklasser til at levere implementation).
- Et redefineret medlem skal være identisk med baseklassens medlem (dvs., samme signatur, returtype og synlighed)
- Ingen af ordene: virtual/ abstract/ override -> ingen specialisering
- Statiske medlemmer kan ikke være virtuelle/abstrakte.

## Medlemmer med virtual keyword kan overrides

```
public class Figur
{
    //Default implementation af en figurs areal
    public virtual double Areal() { return 42; } //Skidt!
}

public class Rektangel : Figur
{
    public double Længde { get; set; }
    public double Bredde { get; set; }

    public override double Areal() { return Længde * Bredde; }
}

public class Trekant : Figur
{
    public double Højde { get; set; }
    public double Grundlinje { get; set; }

    public override double Areal() { return 0.5 * Højde * Grundlinje; }
}

public class Cirkel : Figur { }
```



# Eksempel: Virtual er ikke nok

Generel type

```
static void Main(string[] args)
{
    Figur f = new Figur();
    Console.WriteLine(f.Areal()); //42

    Figur r = new Rektangel() { Længde = 2, Bredde = 3 };
    Console.WriteLine(r.Areal()); //6

    Figur t = new Trekant() { Højde = 4, Grundlinje = 5 };
    Console.WriteLine(t.Areal()); //10

    Console.ReadLine();
}
```

# Abstrakte klasser

- Visse superklasser repræsenterer abstrakte begreber
  - Abstrakte begreber det ikke giver mening at lave objekter af.
- En klasse der er erklæret **abstract** kan ikke instantieres.
  - ...Men den kan godt have constructors
- En abstrakt klasse kan definere **abstrakte medlemmer**
  - med **abstract** keyword'et.
    - Metoder
    - Properties

# Abstrakte medlemmer

- Abstrakte medlemmer har *ingen* implementation
  - Ingen krop, kun signatur
- **Abstrakte medlemmer kan ikke være private**
- Afledte, konkrete, klasser skal implementere *alle* abstrakte medlemmer
- Kun abstrakte subklasser er fritaget for *kravet* om implementation

```

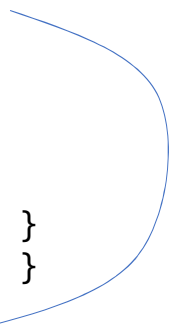
public abstract class Figur
{
    //SKAL overrides i konkrete subklasser
    public abstract double Areal();
}

public class Rektangel : Figur
{
    public double Længde { get; set; }
    public double Bredde { get; set; }
    //SKAL implementeres:
    public override double Areal() { return Længde * Bredde; }
}

//Behøver ikke implementere noget - er gjort i Rektangel klassen.
public class Firkant : Rektangel { }

//behøver ikke implementere noget - er selv en abstrakt klasse.
public abstract class SjovFigur : Figur { }

```



```

static void Main(string[] args)
{
    Figur f = new Figur(); //abstrakte klasser kan ikke instantieres
    Figur r = new Rektangel() { Længde = 2, Bredde = 3 };
    Console.WriteLine(r.Areal()); //6
}

```

# Mere polymorfi

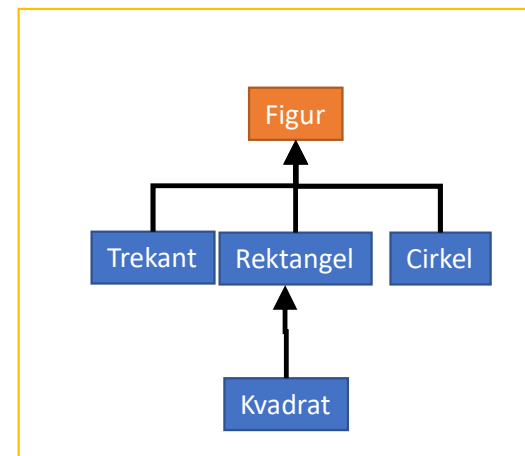
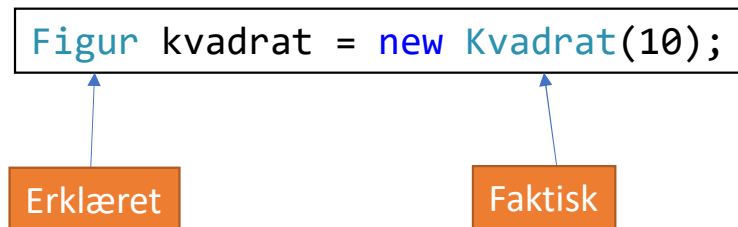
- Under udførsel afgøres metode-kald via *dynamisk binding*:
- Startende fra objektets dynamiske (faktiske) type gennemgås nedarvningskæden indtil implementation af metoden findes.

```
static void Main(string[] args)
{
    List<Figur> figurer = new List<Figur>()
    {
        new Rektangel() { Bredde = 2, Længde = 4 },
        new Trekant() { Grundlinje = 6, Højde = 4},
        new Cirkel() { Radius = 5 }
    };

    foreach (Figur f in figurer) //Udnytter "is-a": Alle figurer har Areal()
    {
        Console.WriteLine(f.Areal());
    }
}
```

# Erklærede og faktiske typer

- Vi skelner mellem **erklærede** og **faktiske** typer.
  - Den erklærede type afgør hvilke medlemmer der er tilgængelige
  - Den faktiske type afgør hvilken implementation af et medlem der kaldes på runtime.



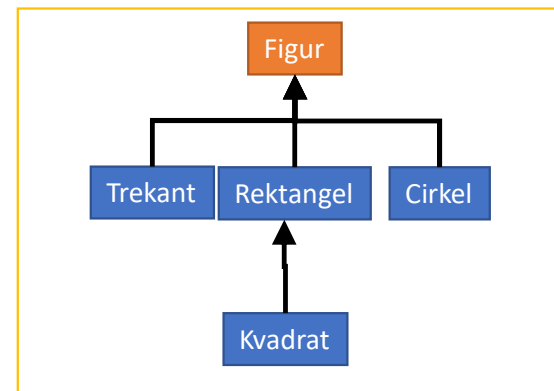
# Erklærede og faktiske typer

- **Typekompatibilitet**

- Typekompatibilitet følger nedarvningskæden.
  - En objektreference kan implicit konverteres til en basetype
  - En objektreference skal eksplicit konverteres til en subtype - via et (down) **cast**.
- En eksplicit typekonvertering giver en `InvalidCastException` hvis den faktiske type ikke er kompatibel med target-typen

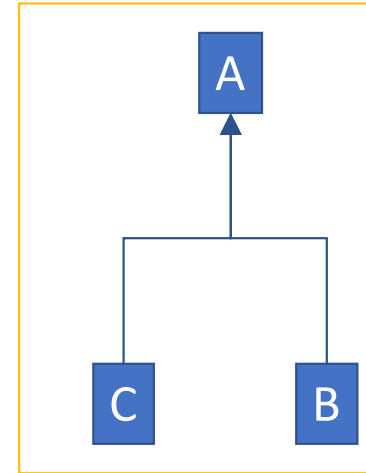
```
Figur kvadrat = new Kvadrat(10);  
Rektangel rekt = (Rektangel)kvadrat;
```

Erklærede type ændres!



# Typecasts

```
class A {  
    public virtual string virtualMethod() { return "A"; }  
}  
  
class B : A {  
    public override string virtualMethod() { return "B"; }  
    public string MethodB() { return "b"; }  
}  
  
class C : A {  
    public override string virtualMethod() { return "C"; }  
    public string MethodC() { return "c"; }  
}
```



```
static void Main(string[] args)  
{  
    A a1 = new A(); //erklæret type A, faktisk type A  
    A a2 = new B(); //erklæret type A, faktisk type B  
    A a3 = new C(); //erklæret type A, faktisk type C  
  
    ((B)a2).MethodB(); //erklæret type B, faktisk type B  
  
    ((C)a3).MethodC(); //erklæret type C, faktisk type C  
  
    //InvalidCastException!!!  
    ((B)a3).MethodB(); //erklæret type B, faktisk type C  
}
```



# Typetjek og typekonvertering

- Typetjek inden en eksplicit typekonvertering
  - for at undgå `InvalidCastException`
- Man kan foretage et typetjek på forskellige måder:
  1. **is**: Tjekker om objekt er typekompatibelt med target-type.

```
static void Main(string[] args) {  
  
    A a = new B(); //erklæret type A, faktisk B  
  
    if (a is B)    // Er a af typen B eller subklasse til B ?  
        ((B)a).MethodB();  
  
}
```

# Typetjek og typekonvertering

- Typetjek inden en eksplicit typekonvertering
  - for at undgå `InvalidCastException`
- Man kan foretage et typetjek på forskellige måder:
  1. **is**: Tjekker om objekt er typekompatibelt med target-type.

```
static void Main(string[] args) {  
  
    A a = new B(); //erklæret type A, faktisk B  
  
    if (a is B b)  
        b.MethodB();  
  
}
```

# Typetjek og typekonvertering

- Typetjek inden en eksplicit typekonvertering
  - for at undgå `InvalidCastException`

2. **GetType()**: Tjekker om objektets faktiske type svarer til target-typen. (**undgå**)

```
static void Main(string[] args) {  
    A b = new B(); //erklæret type A, faktisk B  
  
    if (b.GetType() == typeof(B)) //gælder kun hvis b er af typen B  
        ((B)b).MethodB();  
  
    Console.WriteLine(b is A); //TRUE  
    Console.WriteLine(b.GetType() == typeof(A)); //FALSE  
}
```

# Typetjek og typekonvertering

- Typetjek inden en eksplicit typekonvertering
  - for at undgå `InvalidCastException`

3. **as**: Som **is** – men laver typetjek og typekonvertering i ét hug.

```
static void Main(string[] args) {  
    A b = new B(); //erklæret type A, faktisk B  
  
    Console.WriteLine(b is A); //TRUE  
    Console.WriteLine(b.GetType() == typeof(A)); //FALSE  
  
    C c = b as C; //typetjek + konvertering eller null hvis ugyldig  
    if (c != null)  
        c.MethodC();  
}
```

# this og base

- Et objekts faktiske type afgør også metodekald inden for nedarvningskæden
- **this**: Søgning startes i objektets faktiske type
- **base**: Søgning startes fra objektets baseklasse.

```
class Program {
    static void Main(string[] args) {
        B b = new B();
        Console.WriteLine(b.CallThisVirtualMethod()); //b.VirtualMethod
        Console.WriteLine(b.CallBaseVirtualMethod()); //a.VirtualMethod
        Console.ReadLine();
    }
}

class A {
    public virtual string virtualMethod() { return "a.VirtualMethod"; }
    public string CallThisVirtualMethod() { return this.virtualMethod(); }
}

class B : A {
    public override string virtualMethod() { return "b.virtualMethod"; }
    public string CallBaseVirtualMethod() { return base.virtualMethod(); }
}
```

# Polymorfi og properties

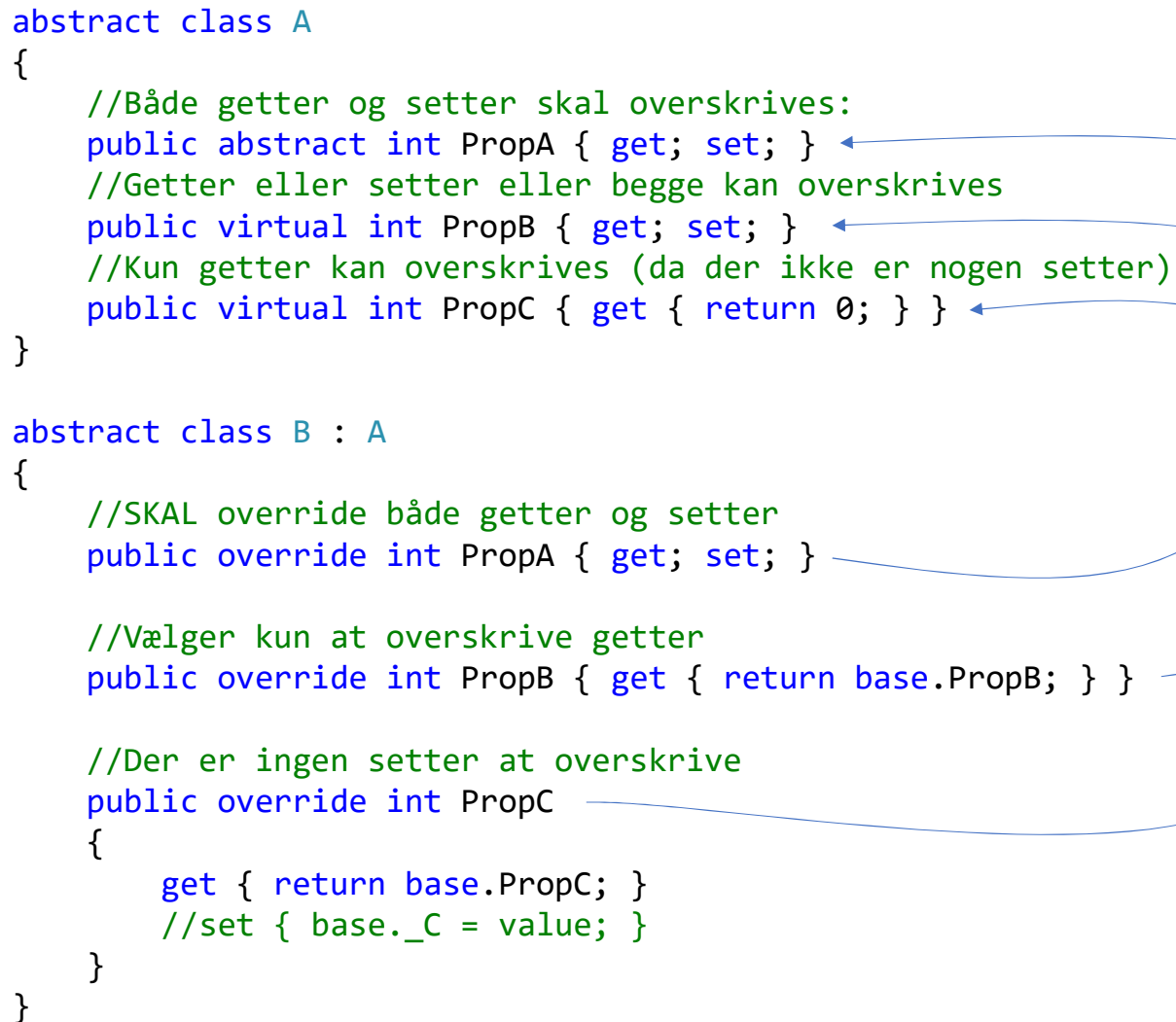
- En virtual/abstract property er "en samlet pakke" (tænk Get- og Set- metoder:
- Hvis property er abstract og definerer både getter og setter, skal *både* getter og setter redefineres i subklasser.
- Hvis property er virtual og definerer både getter og setter kan *enten* getter *eller* setter *eller* begge redefineres i subklasser.
- Man kan kun redefinere getters og setters såfremt de er erklærede. Eks: Hvis ikke der findes en virtual/abstract setter, så kan man ikke introducere en setter i subklasser.

```
abstract class A
{
    //Både getter og setter skal overskrives:
    public abstract int PropA { get; set; }
    //Getter eller setter eller begge kan overskrives
    public virtual int PropB { get; set; }
    //Kun getter kan overskrives (da der ikke er nogen setter)
    public virtual int PropC { get { return 0; } }
}

abstract class B : A
{
    //SKAL override både getter og setter
    public override int PropA { get; set; }

    //Vælger kun at overskrive getter
    public override int PropB { get { return base.PropB; } }


    //Der er ingen setter at overskrive
    public override int PropC
    {
        get { return base.PropC; }
        //set { base._C = value; }
    }
}
```



# Object – alt og alle arver fra **object**

- Er roden i type hierakiet, og udstiller:

Når vi vil tjekke objekters værdi-lighed (senere i kurset)

- [Equals\(Object\)](#)
  - [Object.Equals\(Object a, Object b\)](#)
  - [Object.ReferenceEquals\(Object a, Object b\)](#)
  - [GetHashCode](#)
  - [GetType\(\)](#) **Tommelfingerregel: Brug den ikke til type-tjek**
  - [ToString\(\)](#)
- 



## Specialiser ToString() for fornuftig string repræsentation

```
abstract class Student : Human
{
    public int StudieNummer { get; set; }
    public override string ToString()
    {
        return base.ToString() + $", studienummer = {StudieNummer}";
    }
}
class MatStudent : Student
{
    public override string ToString()
    {
        return base.ToString() + ...
    }
}
```

Kalder base klassens implementation  
vha. base keywordet.

# Stop for specialisering/nedarvning

- Klassens instans-metoder og properties er implicit forseglede per default.
  - Godt/skidt? (Java defaultter til **virtual**)
- Man kan desuden angive at et specialiseret medlem ikke må kunne specialiseres yderligere vha. **sealed** keyword.
- Selve klassen kan også forsegles = ingen subtyper

## Forsegling med sealed

```
class A2
{
    public virtual void Foo() {}
}

class B2 : A2
{
    public override sealed void Foo() {} //Foo() kan nu ikke overrides

    //FEJL: Kan kun forsegle virtuelle metoder
    public sealed string Bar() { return null; }
}

class C2 : B2
{
    //FEJL: Kan ikke override forseglet metode
    public override void Foo() {}
}

sealed class D2 : A2 {}

//FEJL: Kan ikke arve fra forseglet klasse
class E2 : D2 {}
```

# Opsummering

- Nedarvning
  - Giver kodegenbrug i klassehierarkier
  - Constructor chaining i klassehierarki giver også genbrug.
  - Nedarvet indkapsling med protected
  - Abstrakte klasser er klasser der ikke kan instantieres
- SubType-Polymorfi
  - Beror på nedarvning samt virtuelle/abstrakte medlemmer der kan/skal redefineres i subklasser.
- Typekompatibilitet
  - Følger nedarvningskæden
  - Typetjek og typekonvertering med is og as.