.NET and C#

Dag 3: Objektorientering

Indhold

- Klasser
- Normale OO-begreber
- Functions/Methods
- Properties
- Interfaces
- Generics (selvstændig læsning)
- Læse Filer
- Splitting af strings
- IComparer<> interface and sorting Lists

C# class definition

Generel skabelon:

```
class classname
: superclass, interfaces
{
        fields/properties
        constructors
        methods
}
```

```
public class Teacher : Person
    {
        int Salary { get; set; }
        public string Name { get; set; }

        public Teacher(string InitialName, int InitialSalary)
        {
            Name = InitialName;
            Salery = InitialSalery;
        }

        public int getSalary()
        {
            return Salery;
        }
    }
```

Visibility modifiers på members af en klasse – de mest almindelige

- private member: synlig i klassen selv **(default modifier)**
- protected member: synlig i klassen selv og alle subklasser til klassen
- public member: synlig i alle klasser
- internal: synlig i samme assembly

Generelt bruger man mest public/private og i nogle tilfælde protected.

Visibility modifiers for klasser

- Typer i yderste niveau i et namespace (class, struct, interface, enum eller delegate) kan være
  - internal: (default hvis modifier mangler) kun synlig i samme assembly
  - public: synlig i alle assemblies
- En indre (nested klasse) kan have forskellige modifiers:
  - private: (default hvis modifier mangler)
  - public
  - protected
  - internal
- Det er nok ikke så almindeligt at bruge internal.

Metode modifiers

- private protected og protected internal: lidt mere obskure, man kan læse om dem her:
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/access-modifiers

Constants fields

- Erklæret med keyword const

```
public class Car {
  private const int maxSpeed = 100;
  ..
}
```

- En const er implicit static
- En const **skal** initialiseres i erklæringen.
- En const evalueres på **compile time**

Nedarvnings-hierarki

- Relaterede klasser kan organiseres i klassehierarkier
- En subklasse arver alle members (bortset fra constructors) fra sin superklasse
- En subklasse kan kun arve fra én klasse (ingen multipel nedarvning) (men kan implementere multiple interfaces – ligesom Java)
- Syntaks:

```
class SubKlasse: SuperKlasse { … }
```

Superklasse: Shape

```
public class Shape {
   private double x,y;

   public Shape(double x, double y)
   {
         this.x = x; this.y = y;
   }
   public double X {
         get {return x;}
         set {x = value;}
   }
   public double Y {
         get {return y;}
         set {y = value;}
   }
}
```

Subklasse: Circle

```
public class Circle: Shape {
   private double radius;

   public Circle(double x, double y,
         double radius): base(x,y) {
         this.radius = radius;
   }
   public double Radius {
         get {return radius;}
         set {radius = value;}
   }
   public Point getCenter() {
         return new Point(X+radius,
                  Y+radius);
     /* OBS: de private felter x og
     y kan ikke ses her! Så vi
   bruger properties X og Y (stort!)
   */
   }
}
```

Eksempler på klassehierarkier

- TextBox nedarver fra
  TextBoxBase
  Control
  Component
  Object
- Form nedarver fra
  ContainerControl
  ScrollableControl
  Control

Constructors

- Har samme navn som klassen – altså Person(string: name), hvis klassen hedder Person
- C# laver en tom default constructor, hvis ingen constructor er lavet
- C# tillader overloadededede constructors definitioner – altså forskellige constructors med forskellige parametre. **Der er kan ikke være 2 constructors med de samme parametre.**

Constructors i subklasser

- En constructor i en subklasse *kalder altid først* superklassens constructor (hvis man vil bruge superklassens constructor selvfølgelig) eller en anden constructor fra samme klasse
- Kaldet kan være implicit eller explicit
- Implicit kald:

```
public SubKlasse () {…}
```

Superklassens ***parameterløse*** constructor kaldes implicit som første instruktion i subklassens constructor, hvis ingen eksplicit constructor er angivet.

Constructors i subklasser

- Explicit kald:

```
public Circle(double x, double y,…) :base(x,y) {…}
```

Superklassens constructor

```
public Shape(double x, double y) {…}
```

kaldes som første linie i Circle-constructoren

Eksempel

```
public class Super
{
   public Super()
   { }
   public Super(int age)
   { }
}
```

```
public class Sub : Super
{
   public Sub()
   { }
   public Sub(int age):
                    base(age)
   { }
}
```

```
Sub obj = new Sub()
```

Kalder:
1) Super()
2) Sub()

```
Sub obj = new Sub(25);
```

Kalder:
1) Super(25)
2) Sub(25)

Constructor chaining (ikke inheritance)

```
public Circle()
{
  x = 0; y = 0; r = 1;
}

public Circle(int x, int y): this() // bruger egen constructor
{
  this.x=x; this.y=y;
}

public Circle(int x, int y, int r): this(x,y) // bruger egen constructor
{
  this.r = r;
}
```

Virtuelle og ikke-virtuelle metoder

- En virtuel metode kendes på nøgleordene virtual *og* override

  Superklassen:

```
public virtual void
    move(double dist) {…}
```

  Subklassen:

```
public override void
    move(double dist) {…}
```

- **Metoder uden virtual er ikke-virtuelle og kan således ikke overrides (**men kan faktisk gen-erklæres i en subklasse med new foran metodenavn**)**

Polymorfi-eksempel

```
Shape[ ] shapes = new Shape[10];
// … her tilføjes Shape-objekter og
// … Circle-objekter til arrayet shapes
foreach (Shape s in shapes) {
   s.Move();
}
```

Afhængigt af typen af objektet s, kaldes enten Area() fra Shape eller Area() fra Circle. Præcis som i JAVA

Abstrakte metoder

- En abstrakt metode (eller property eller indexer) indeholder ingen kode (ingen krop)
- Syntaks:

```
abstract double Move();
//obs: metoden har ingen krop
```

- En metode erklæret med **abstract** er implicit **virtual** og *skal* derfor override's i subklasser
- En abstrakt metode kan ikke være private eller static

Abstrakte klasser

- En klasse indeholdende en abstrakt metode (eller property eller indexer) skal selv være abstrakt
- Syntaks abstrakt klasse:

```
public abstract class Shape {…}
```

- Abstrakt property:

```
public abstract int N{
  get;
  set;    }
```

Abstrakt klasse

- Der kan ikke skabes objekter tilhørende en abstrakt klasse
- En abstrakt klasse behøver ikke have nogen abstrakte metoder (men vil ofte have det)
- En abstrakt klasse med abstrakte metoder er tænkt som superklasse i et klassehierarki, hvor polymorfi anvendes – dvs. subklasser vil implementere de abstrakter metoder.

Sealed metoder

- En sealed (forseglet) metode kan ikke override's i en subklasse (svarer til Java's final)
- Syntaks:

```
public sealed override void move() {…}
```

- Statiske og private metoder og properties er implicit sealed.
- Forhindrer mod override i subklasser

Sealed klasser

- En sealed klasse kan ikke have subklasser
- Syntaks:

```
public sealed class Math {…}
```

- Sikrer mod nedarvning og override af virtuelle metoder

struct, en letvægts klasse

- struct minder meget om class
- programmøren kan **ikke** erklære en parameterløs constructor
- struct indeholder både felter og metoder (i modsætning til C++, hvor en struct kun indeholder felter)
- en struct kan **ikke** nedarves
- en struct er en **value** *type*
- et objekt af en class er en reference type

Operatorerne is og as

```
object o1=”Hans Nielsen";
object o2=new ListBox();

if (o1 is string) Console.WriteLine("o1 is string");
if (o2 is string) Console.WriteLine("o2 is string");
if (o1 is ListBox) Console.WriteLine("o1 is ListBox");
if (o2 is ListBox) Console.WriteLine("o2 is ListBox");

(o2 as ListBox).Items.Add("element");
```

Interface erklæring

Syntaks:

```
interface IMoveable {
  void move();
}
```

- Metoder, properties og indexers i et interface er **virtuelle**.
- Metoden move() er implicit public og abstract (og dermed virtuel). Altså *ingen ekstra* modifiers på metode i interface

Interface eksempel

```
static void Main(string[] args)
{
    Person me = new Teacher("Morten",
                                    10000);
    Console.WriteLine(me.getSalery());
    Console.ReadLine();
}
```

- Interface er en samling af abstrakte public metoder
- Interface indeholder ikke felter
- Interface indeholder ikke metode implementation

```
interface Person
{
   UInt64 getSalary();
}
public class Teacher : Person
{
    UInt64 Salary { get; set; }
    public String Name { get; set; }

    public Teacher(String InitialName, UInt64 InitialSalery)
    {
        Name = InitialName;
        Salary = InitialSalery;
    }

    public UInt64 getSalary()
    {
        return Salary;
    }
}
```

Interface

- Compileren kontrollerer, at en klasse som implementerer et interface, indeholder alle members fra interfacet – ellers får man fejl!

Interface

- En klasse kan nedarve fra én superklasse og samtidig implementere et eller flere interfaces.
- Syntaks:

```
public class Drawing: Shape, IMoveAble
{…}
```

Superklassen ***skal*** stå først efter kolon.

Interface – nedarvning fra andet interface

- Et interface kan nedarve fra et andet interface
- En klasse som implemeterer subinterfacet, skal implementere alle metoder fra superinterfacet og subinterfacet

Classes/Inheritance/Interfaces - eksempel

Indexers

- List har ikke en get-metode som i Java
- I stedet har den en Item-metode, der virker både som getter og setter.
- Den kan **ikke** kaldes som liste.Item(i) i C#.
- Man skal skrive liste[i] i stedet.
- Det er fordi Item er **indexeren** for List.
- Man kan lave sin egen indexer. Slå det op, det er ret let. Eks.: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/using-indexers

OPGAVER!