Object oriented programming in C# (9)

- Encapsulation
- Inheritance
- Polymorphy

Plus: Abstractions

- A forth pillar
- A class is an user defined type
  - containing attributtes
  - and methods
- CREATES an abstraction

Classes and objects

- An object is an instance of a class
- The methods of the class work on the attributes of the instance

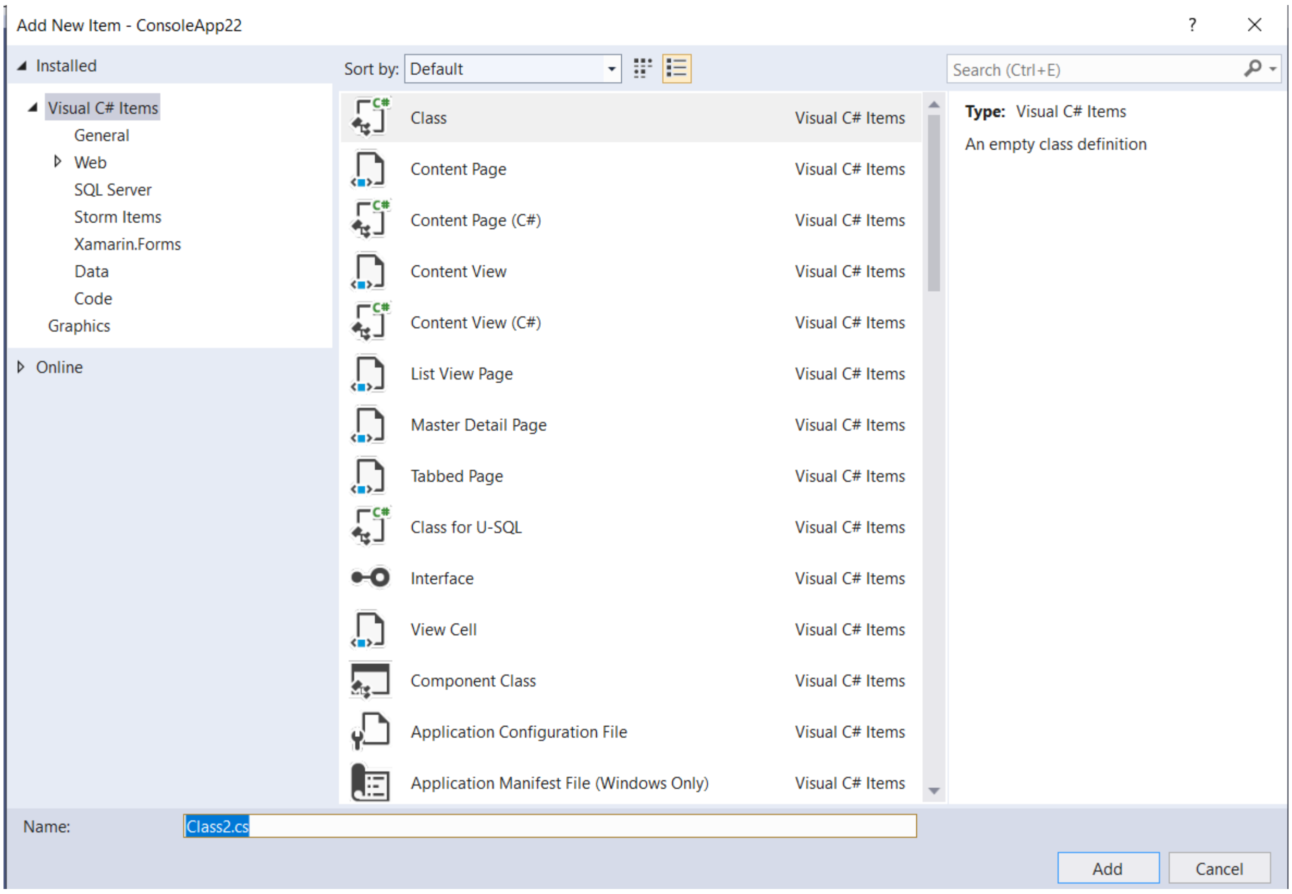
Constructors

- We can have many (with different signature)
- A default constructor exists(zero args)
  - Task: to initialize data with a default value
  - Removed if a specialized constructor is added (logic!)

Classes

- User defined type
- Contents:
  - Field data (member variable)
    - Constructors
    - Properties
    - Methods
    - Events

- Add a new class in VS2022


Add New Item - ConsoleApp22

Installed
- Visual C# Items
  - General
  - Web
  - SQL Server
  - Storm Items
  - Xamarin.Forms
  - Data
  - Code
  - Graphics
- Online

Sort by: Default

Class	Visual C# Items
Content Page	Visual C# Items
Content Page (C#)	Visual C# Items
Content View	Visual C# Items
Content View (C#)	Visual C# Items
List View Page	Visual C# Items
Master Detail Page	Visual C# Items
Tabbed Page	Visual C# Items
Class for U-SQL	Visual C# Items
Interface	Visual C# Items
View Cell	Visual C# Items
Component Class	Visual C# Items
Application Configuration File	Visual C# Items
Application Manifest File (Windows Only)	Visual C# Items

Search (Ctrl+E)

Type: Visual C# Items
An empty class definition

Name: Class2.cs

FIELD DATA IS private or protected

More about constructors

- The constructor should not have too many arguments
- The class will have **to much responsibility**

- Also consider how many constructors you need
- Many constructors also points ar **to much responsibility**

The keyword this

- as in C++ and Java

Example:

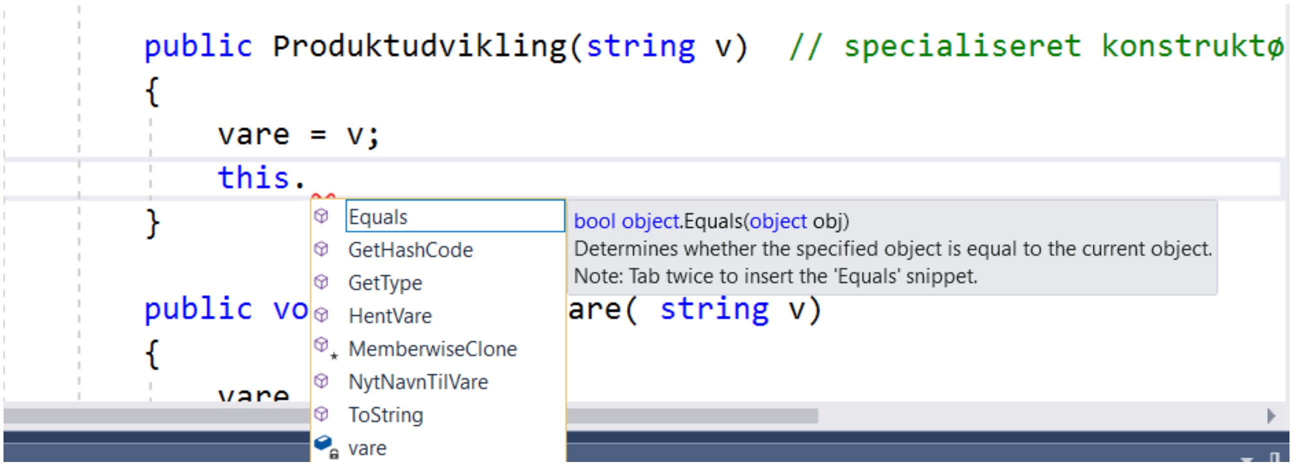
```
...

class Elektronik {

private string compType;
private double pris;


public Elektronik(string compType, double pris)
{
        this.compType = compType;

        this.pris     = pris;


}
...
```

- this refers to the object!

Remember:

- static methods can NOT use the this keyword –
    - static methods operates on class level not on objects

Intellisense:



```
public Produktudvikling(string v)  // specialiseret konstruktø
{
    vare = v;
    this.
}

public vo                        are( string v)
{
    vare
```

bool object.Equals(object obj)
Determines whether the specified object is equal to the current object.
Note: Tab twice to insert the 'Equals' snippet.

this can call other constructors

Example:

```
...

class Elektronik {

public string compType;
public double pris;


public Elektronik(string compType, double pris)
{
        this.compType = compType;

        this.pris     = pris;



}

public Elektronik(string compType)  :this(compType,0.0)
{



}
...
```

Optional & named arguments (constructors)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ObjOriented
{
    class Program
    {
        static void Main(string[] args)
        {
            Bil skoda = new Bil(navn: "Skoda");
            skoda.UdskrivBil();

            Bil lada = new Bil(type:"Lastvogn",navn: "Lada");
            lada.UdskrivBil();


        }
    }


    class Bil
    {

        private string navn;
        private string type;

        public Bil(string navn, string type="Bil")
        {
            this.navn = navn;
            this.type = type;
        }

        public void UdskrivBil()
        {
            Console.WriteLine(navn);
            Console.WriteLine(type);
        }
    }
}
```

- parameter order is changed with purpose

static keyword

- static data

```
...
...
    class Bil
    {

        private string navn;
        private string type;
        private static int tæller;

        public Bil(string navn, string type="Bil")
        {
            this.navn = navn;
            this.type = type;
            tæller++;
        }

        public void UdskrivBil()
        {
            Console.WriteLine(navn);
            Console.WriteLine(type);
            Console.WriteLine("antal køretøjer: {0}",tæller);
        }
    }
```

- static methods

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ObjOriented
{
    class Program
    {
        static void Main(string[] args)
        {
            Bil.UdskrivAntal();

            Bil skoda = new Bil(navn: "Skoda");
            skoda.UdskrivBil();
            Bil.UdskrivAntal();

            Bil lada = new Bil(type:"Lastvogn",navn: "Lada");
            lada.UdskrivBil();
            Bil.UdskrivAntal();

        }
    }


    class Bil
    {

        private string navn;
        private string type;
        private static int tæller;

        public Bil(string navn, string type="Bil")
        {
            this.navn = navn;
            this.type = type;
            tæller++;
        }

        public void UdskrivBil()
        {
            Console.WriteLine(navn);
            Console.WriteLine(type);

        }

        public static void UdskrivAntal()
        {
            Console.WriteLine("antal køretøjer: {0}", tæller);
        }
    }
}
```

Static methods work on static data

Static constructors

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ObjOriented
{
    class Program
    {
        static void Main(string[] args)
        {
            Bil.UdskrivAntal();

            Bil skoda = new Bil(navn: "Skoda");
            skoda.UdskrivBil();
            Bil.UdskrivAntal();

            Bil lada = new Bil(type:"Lastvogn",navn: "Lada");
            lada.UdskrivBil();
            Bil.UdskrivAntal();

        }
    }


    class Bil
    {

        private string navn;
        private string type;
        private static int tæller;

        static Bil()
        {
            tæller = 10;
        }

        public Bil(string navn, string type="Bil")
        {
            this.navn = navn;
            this.type = type;
            tæller++;
        }

        public void UdskrivBil()
        {
            Console.WriteLine(navn);
            Console.WriteLine(type);

        }

        public static void UdskrivAntal()
        {
            Console.WriteLine("antal køretøjer: {0}", tæller);
        }

        public static void ÆndreAntal(int antal)
        {
            tæller = antal;
        }
    }
}
```

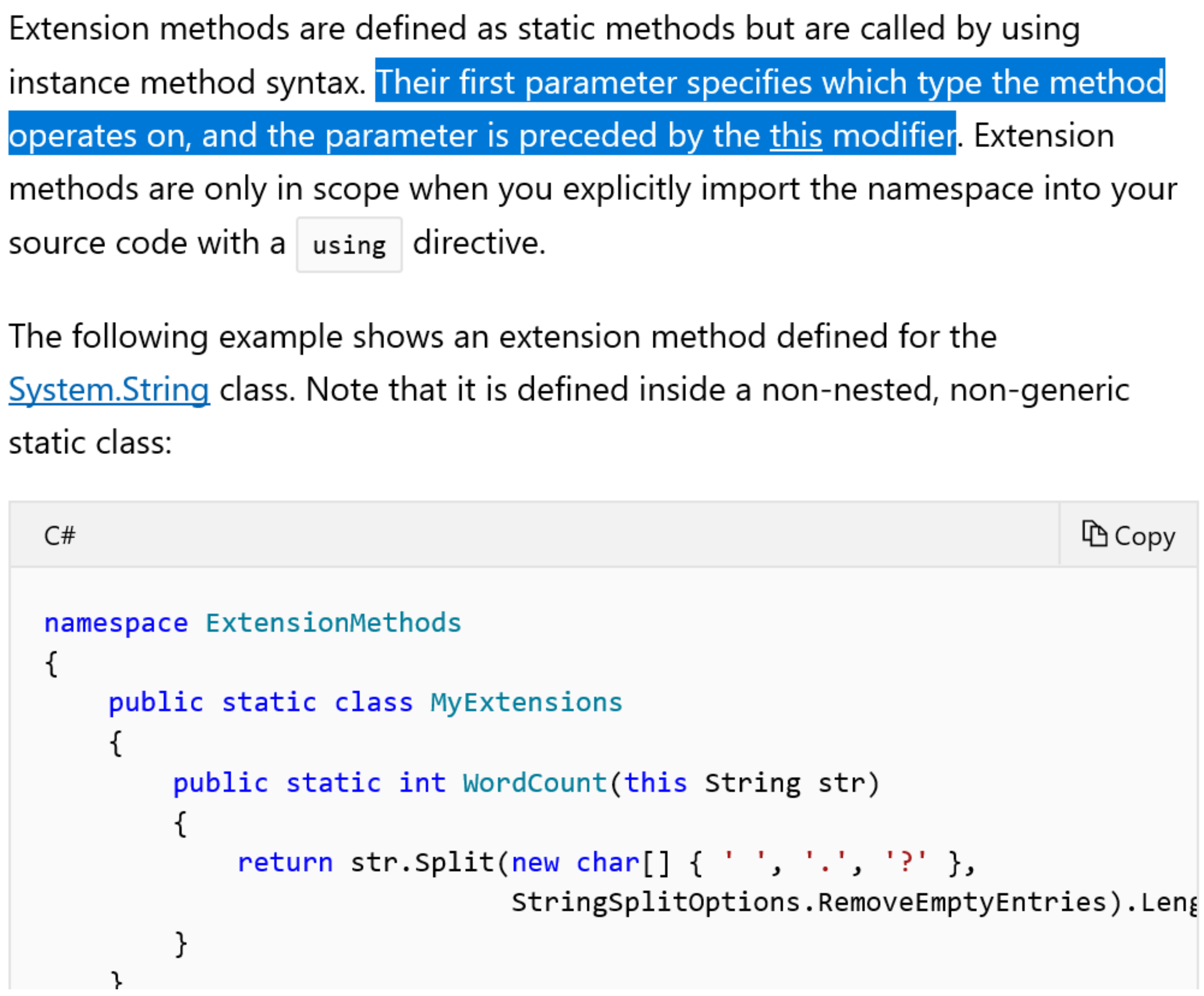
- Work on static data
- A static constructor
    - Takes no arguments
    - Only one static constructor is allowed
    - Is executed ONCE an always before creation of class objects

Static classes

- Data must be static
- Can be used for minor jobs

```
...
static class Dato
   {
        public static void UdskrivDato()
        {
            String dato = DateTime.Now.ToShortDateString();
            Console.WriteLine(dato);
        }
    }
...

Dato.UdskrivDato();  // kald
...
```

Extension methods



Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on, and the parameter is preceded by the this modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the System.String class. Note that it is defined inside a non-nested, non-generic static class:

C# Copy

```csharp
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Leng
        }
    }
```

Ecapsulation, inheritance and polymorphy

- How well is the implementation of the object hidden?
- How easy is it to reuse code? (inheritance)
  - Is the generalization reason?
- How can we treat different object alike (polymorphy)

Encapsulation and data hiding

- A class has a responsibility
  - REFLECTED in the name of the class
  - The methods encapsulates complexity
  - and has private variables to work on (data hiding)

Inheritance: ”is-a” relations

- New class definitions based on existing classes
  - super classes (Java)
  - base classes (C++)
- ”Is a ” relations
  - A circle class is a Shape class

”Has-a” relations

- ”Has a” relations
  - A class uses another class

Polymorphy

Classic polymorphy:

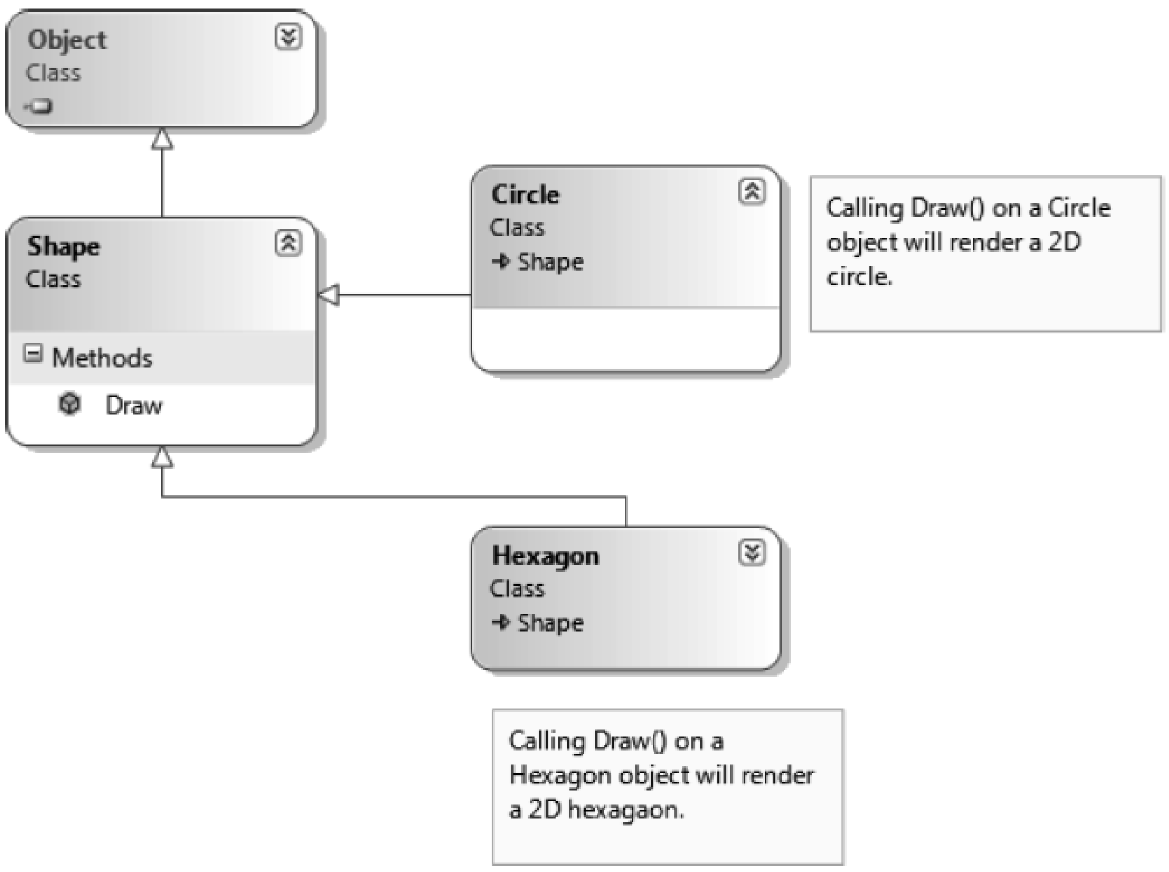

Object
Class
Shape
Class
Methods
Draw
Circle
Class
Shape
Calling Draw() on a Circle object will render a 2D circle.
Hexagon
Class
Shape
Calling Draw() on a Hexagon object will render a 2D hexagaon.

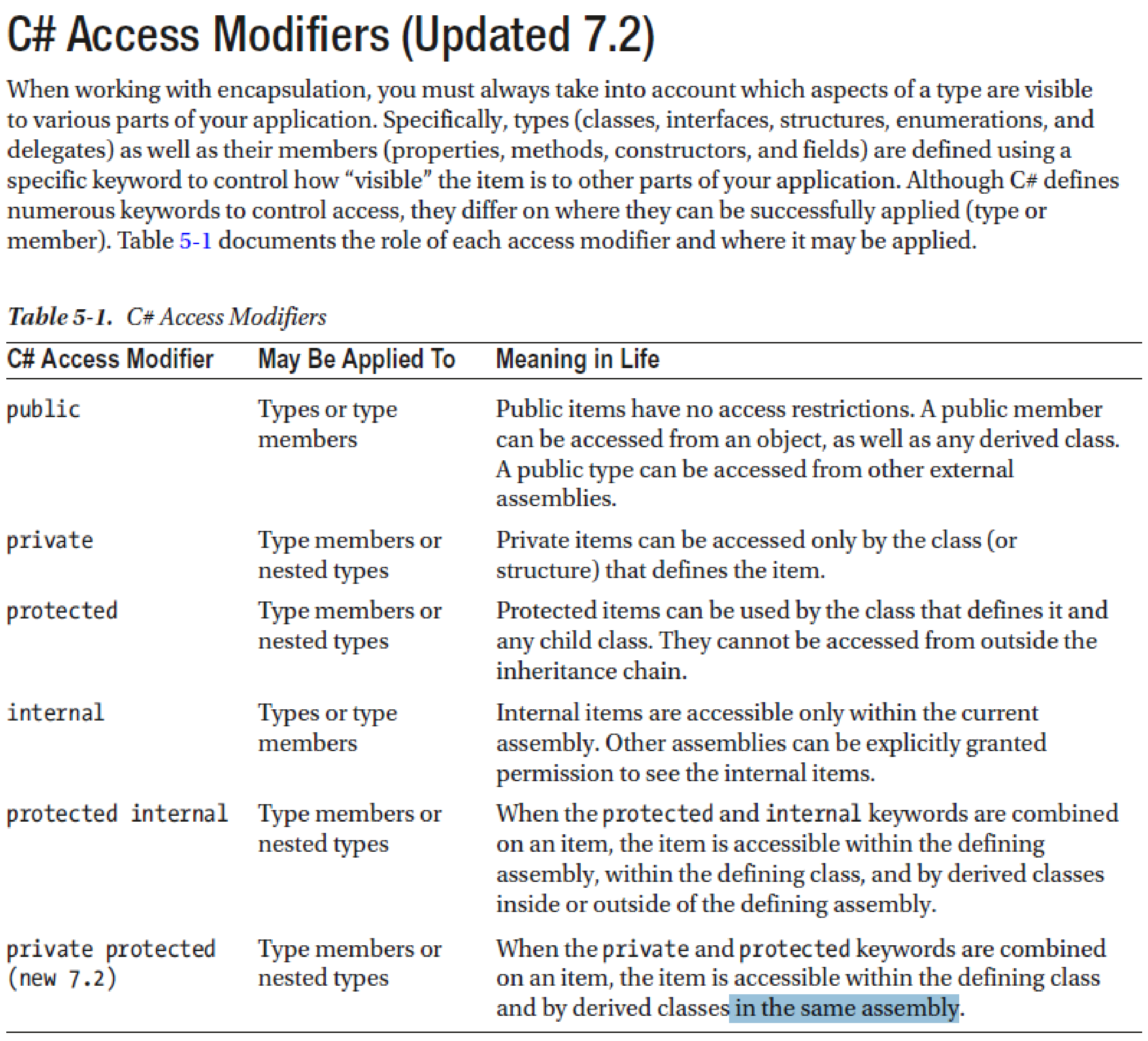
Figure 5-5. Classical polymorphism

- Base class has virtual members
- Override is used by inheriting class
- An abstract method in a base class gives no default implementering but **MUST** be overide
- Override = redefinition

Basic idea is the use of arrays

```
…
class Program
{
static void Main(string[] args)
{
Shape[] myShapes = new Shape[3];
myShapes[0] = new Hexagon();
myShapes[1] = new Circle();
myShapes[2] = new Hexagon();
foreach (Shape s in myShapes)
{
// Use the polymorphic interface!
s.Draw();
}
Console.ReadLine();
}
}
…
```

Access modifiers



C# Access Modifiers (Updated 7.2)

When working with encapsulation, you must always take into account which aspects of a type are visible to various parts of your application. Specifically, types (classes, interfaces, structures, enumerations, and delegates) as well as their members (properties, methods, constructors, and fields) are defined using a specific keyword to control how "visible" the item is to other parts of your application. Although C# defines numerous keywords to control access, they differ on where they can be successfully applied (type or member). Table 5-1 documents the role of each access modifier and where it may be applied.

Table 5-1. *C# Access Modifiers*

C# Access Modifier	May Be Applied To	Meaning in Life
`public`	Types or type members	Public items have no access restrictions. A public member can be accessed from an object, as well as any derived class. A public type can be accessed from other external assemblies.
`private`	Type members or nested types	Private items can be accessed only by the class (or structure) that defines the item.
`protected`	Type members or nested types	Protected items can be used by the class that defines it and any child class. They cannot be accessed from outside the inheritance chain.
`internal`	Types or type members	Internal items are accessible only within the current assembly. Other assemblies can be explicitly granted permission to see the internal items.
`protected internal`	Type members or nested types	When the `protected` and `internal` keywords are combined on an item, the item is accessible within the defining assembly, within the defining class, and by derived classes inside or outside of the defining assembly.
`private protected` (new 7.2)	Type members or nested types	When the `private` and `protected` keywords are combined on an item, the item is accessible within the defining class and by derived classes in the same assembly.

- Type members are default private
- Types are default internal

- A type can contain a private type

- An outer type cannot be private

Nested types

```
public class Motor
{
  private enum MotorType
  {
     Benzin, Diesel, El
   }

   public Motor(){}

}
```

Internal class with private default constructor

```
class Motor
{
  Motor(){} // default: private

}
```

- Can only be accessed in the same assembly
- A private constructor can only be called internally form the class: Use a static method

Encapsulation in C#

- As always in OOP
  - Data of a class should not be accessed directly
  - Uses SetX / GetX methods
- Four levels
  - public
  - private
  - protected
  - protected internal

Encapsulation

Example on why public is dangerous:

```
public class Elektronik {


 public int nummer;   // Oh no



}

public class User {

     public static void Main()
    {

      Elektronik e = new Elektronik();
      e.nummer= 3000;
      Console.WriteLine("No {0}",e.nummer);
      Console.ReadLine();


    }

}
```

Encapsulation in .NET: Properties

- In C# exists
  - Properties
- Example.
  - A System.Int32 has (also) a *value*

```
using System;



public class Elektronik {

 private int nummer;

 public int Nummer
 {
      get{return nummer;}
      set { nummer = value;}

 }



}

public class User {

     public static void Main()
    {

      Elektronik e = new Elektronik();
      e.Nummer= 200;
      Console.WriteLine("No {0}",e.Nummer);
      Console.ReadLine();


    }

}
```

- set and get blocks
- value is not a C# keyword
- but a contextual keyword

- logic can be added:

```
…
set
{
// Here, value is really a string.
if (value.Length > 15)
Console.WriteLine("Error! Name must be less than 16 characters!");
else
empName = value;
}
…
```

Read- only and Write-only

- read-only:
    - Remove the set block
- write-only:
    - Remove the get block

Visiblity and properties

- a **protected set** means that only the class plus inherited classes can access this

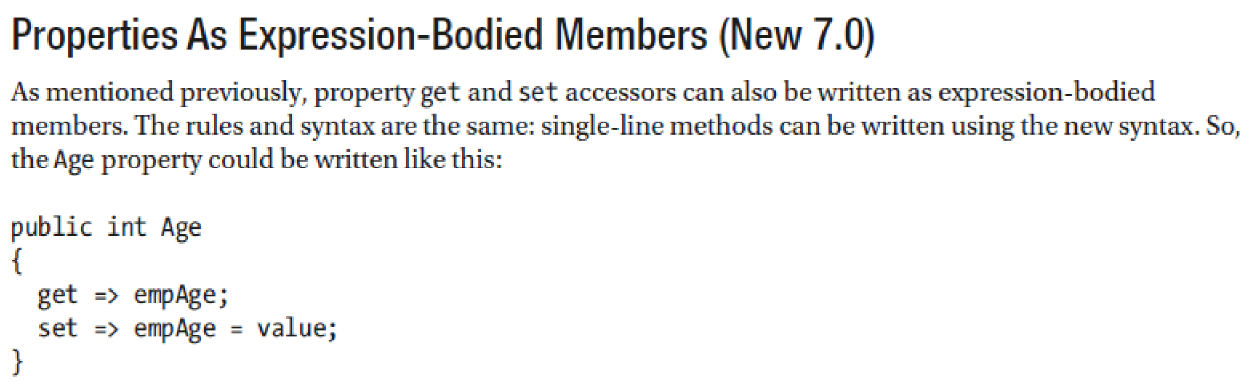
Static properties

- MUST WORK ON STATIC DATA

Automatic properties

- Need a simple data holder: Use automatic properties
- **public string MotorSize { get; set;}**
- read/write can still be used

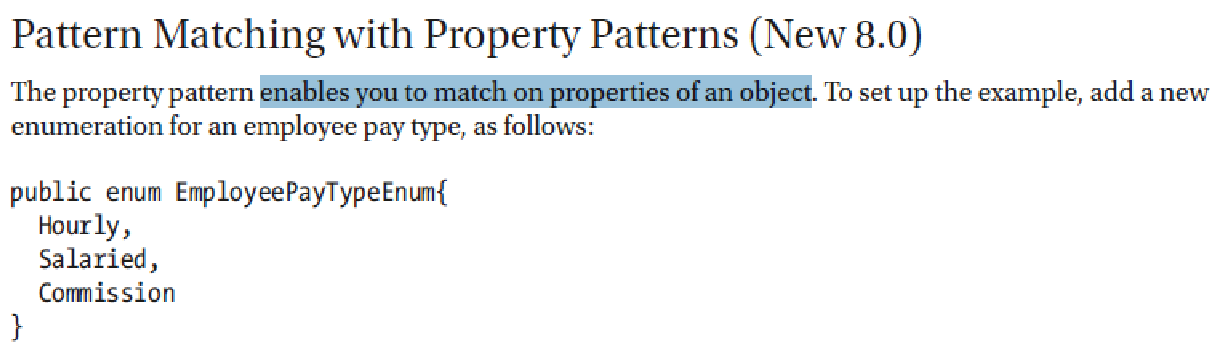
```
…
// Automatic properties!
public string PetName { get; set; }
public int Speed { get; set; }
public string Color { get; set; }
…
```



Properties As Expression-Bodied Members (New 7.0)

As mentioned previously, property get and set accessors can also be written as expression-bodied members. The rules and syntax are the same: single-line methods can be written using the new syntax. So, the Age property could be written like this:

```
public int Age
{
  get => empAge;
  set => empAge = value;
}
```



Pattern Matching with Property Patterns (New 8.0)

The property pattern enables you to match on properties of an object. To set up the example, add a new enumeration for an employee pay type, as follows:

```
public enum EmployeePayTypeEnum{
  Hourly,
  Salaried,
  Commission
}
```

Update the Employee class with a property for the pay type and initialize it from the constructor. The relevant code changes are listed here:

```
private EmployeePayTypeEnum _payType;
public EmployeePayTypeEnum PayType
{
  get => _payType;
  set => _payType = value;
}
public Employee(string name, int id, float pay, string empSsn)
  : this(name,0,id,pay, empSsn, EmployeePayTypeEnum.Salaried)
{
}
public Employee(string name, int age, int id,
  float pay, string empSsn, EmployeePayTypeEnum payType)
{
  Name = name;
  Id = id;
  Age = age;
  Pay = pay;
  SocialSecurityNumber = empSsn;
  PayType = payType;
}
```

```
public void GiveBonus(float amount)
{
  Pay = this switch
  {
    {PayType: EmployeePayTypeEnum.Commission }
      => Pay += .10F * amount,
    {PayType: EmployeePayTypeEnum.Hourly }
      => Pay += 40F * amount/2080F,
    {PayType: EmployeePayTypeEnum.Salaried }
      => Pay += amount,
    _ => Pay+=0
  };
}
```

As with other switch statements that use pattern matching, there must either be a catch-all case statement or the switch statement must throw an exception if none of the case statements are met.

To test this, add the following code to the Main method:

```
Employee emp = new Employee("Marvin",45,123,1000,"111-11-1111",EmployeePayTypeEnum.
Salaried);
Console.WriteLine(emp.Pay);
emp.GiveBonus(100);
Console.WriteLine(emp.Pay);
```

Constructors: Object initialization syntax

- 1. A default constructor can be used together with properties (implicit)
- 2. A default constructor can be used together with properties (explicit)
- 3. Object initialization syntax

```
class Point
{
public int X { get; set; }
public int Y { get; set; }
public Point(int xVal, int yVal)
{
X = xVal;
Y = yVal;
}
public Point() { }
public void DisplayStats()
{
Console.WriteLine("[{0}, {1}]", X, Y);
}
}

Point firstPoint = new Point();
firstPoint.X = 10;
firstPoint.Y = 10;
firstPoint.DisplayStats();

// Or make a Point via a custom constructor.
Point anotherPoint = new Point(20, 20);
anotherPoint.DisplayStats();

// Or make a Point using object init syntax.
Point finalPoint = new Point { X = 30, Y = 30 };
finalPoint.DisplayStats();
Console.ReadLine();
```

- Watch out!

- Point pt = new Point(10, 16) { X = 100, Y = 100 };
- Value will be 100 (not10,16)

Initialization of: “has-a” relations

- Traditional: use **new** in the outer class

```
class Rectangle
{
private Point topLeft = new Point();
private Point bottomRight = new Point();
public Point TopLeft
{
get { return topLeft; }
set { topLeft = value; }
}
public Point BottomRight
{
get { return bottomRight; }
set { bottomRight = value; }
}
public void DisplayStats()
{
Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
topLeft.X, topLeft.Y, topLeft.Color,
bottomRight.X, bottomRight.Y, bottomRight.Color);
}
}
```

- Also object initialization can be used (combined with properties)

```
Rectangle myRect = new Rectangle
{
  TopLeft        = new Point(X =10, Y=20);
  BottomRight  = new Point(X =100, Y=200);
};
```

the readonly keyword

Example:

```
class MyMathClass
{
// Read-only fields can be assigned in ctors,
// but nowhere else.
public readonly double PI;
public MyMathClass ()
{
PI = 3.14;
}
}
```

- Only constructors can place value in a readonly
- A set method is not possible

static readonly variables –close to const

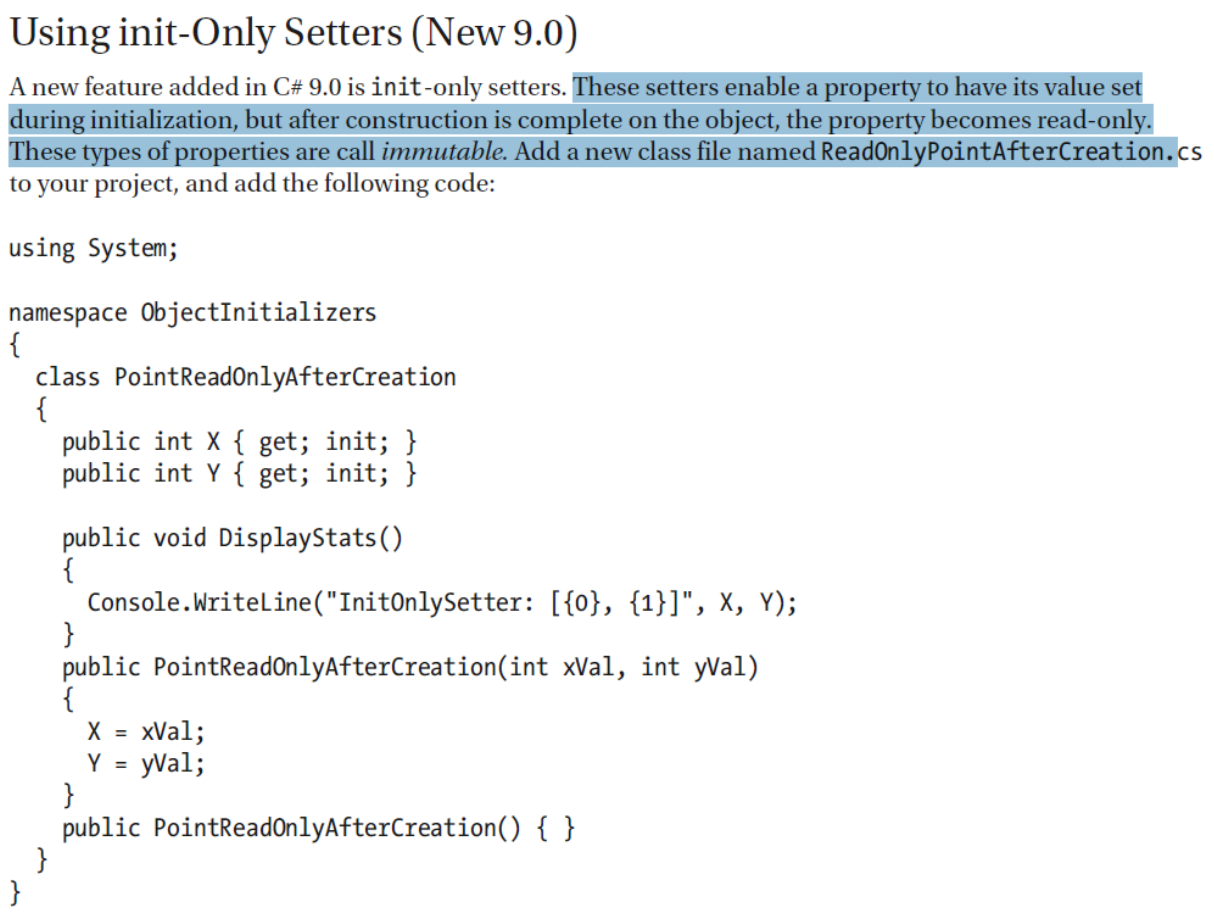
- const is decided on compile-time
- static readonly on run-time

Example:

```
class MyMathClass
{
public static readonly double PI = 3.14;
}
class Program
{
static void Main(string[] args)
{
```

```
Console.WriteLine("***** Fun with Const *****");
Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
Console.ReadLine();
}
}
```

New I C# 9

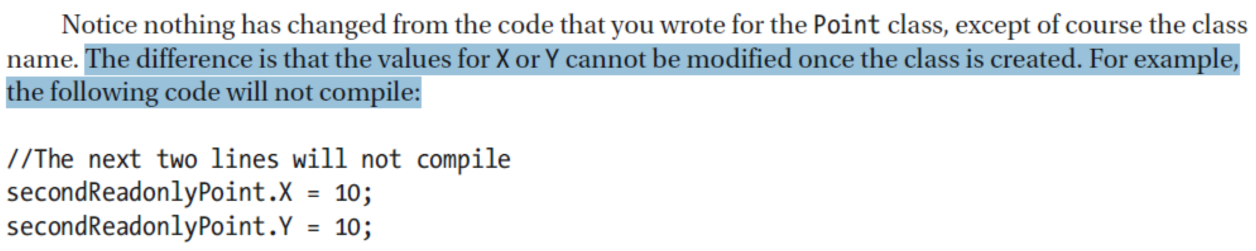


Using init-Only Setters (New 9.0)

A new feature added in C# 9.0 is `init`-only setters. These setters enable a property to have its value set during initialization, but after construction is complete on the object, the property becomes read-only. These types of properties are call *immutable.* Add a new class file named `ReadOnlyPointAfterCreation.cs` to your project, and add the following code:

```
using System;

namespace ObjectInitializers
{
  class PointReadOnlyAfterCreation
  {
    public int X { get; init; }
    public int Y { get; init; }

    public void DisplayStats()
    {
      Console.WriteLine("InitOnlySetter: [{0}, {1}]", X, Y);
    }
    public PointReadOnlyAfterCreation(int xVal, int yVal)
    {
      X = xVal;
      Y = yVal;
    }
    public PointReadOnlyAfterCreation() { }
  }
}
```

```
//Make readonly point after construction
PointReadOnlyAfterCreation firstReadonlyPoint = new PointReadOnlyAfterCreation(20, 20);
firstReadonlyPoint.DisplayStats();

// Or make a Point using object init syntax.
PointReadOnlyAfterCreation secondReadonlyPoint = new PointReadOnlyAfterCreation { X = 30, Y
= 30 };
secondReadonlyPoint.DisplayStats();
```


Notice nothing has changed from the code that you wrote for the `Point` class, except of course the class name. The difference is that the values for `X` or `Y` cannot be modified once the class is created. For example, the following code will not compile:

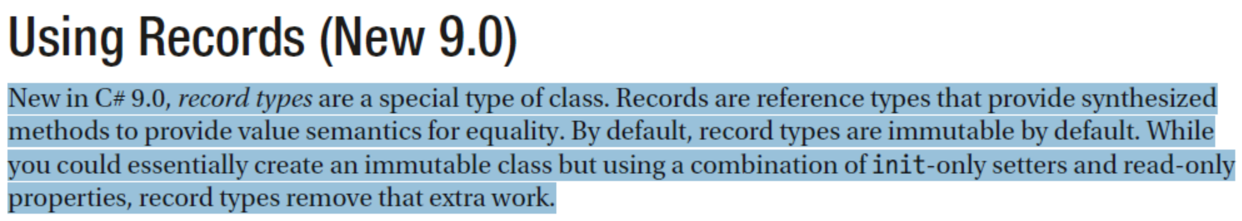
```
//The next two lines will not compile
secondReadonlyPoint.X = 10;
secondReadonlyPoint.Y = 10;
```

Partial types

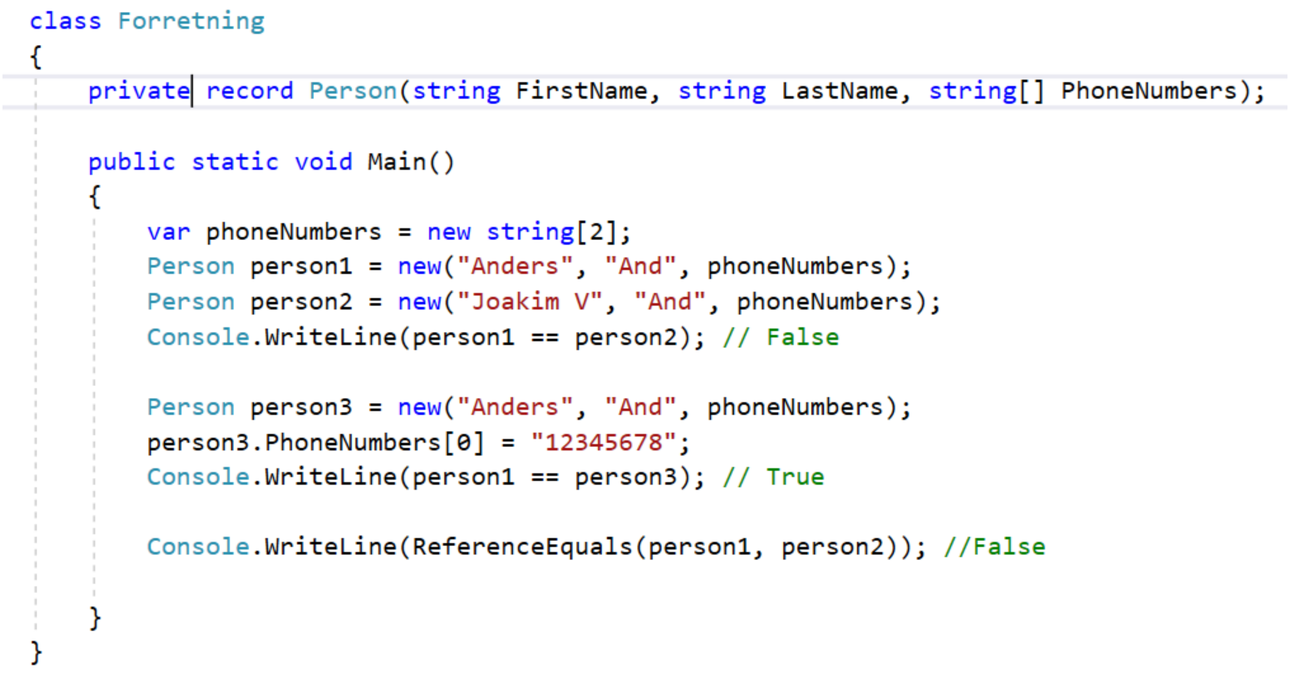
- Splits a class into two files

```
...
public partial class MyClass
{

}
...
```

New in C# 9: Records

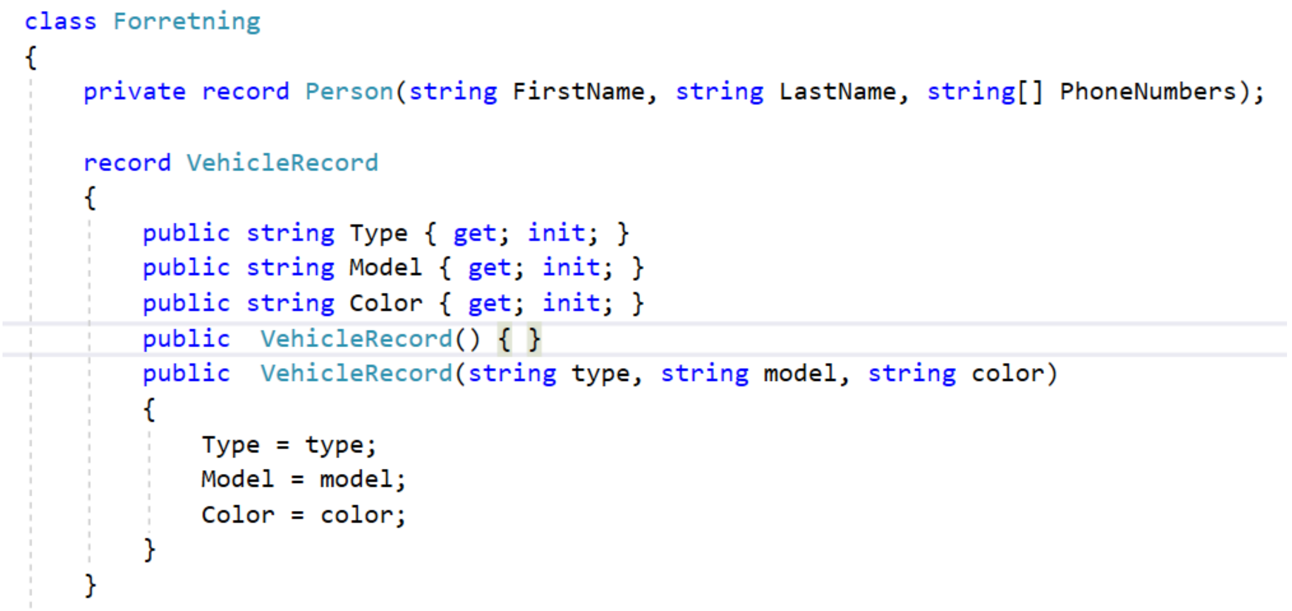

Using Records (New 9.0)

New in C# 9.0, *record types* are a special type of class. Records are reference types that provide synthesized methods to provide value semantics for equality. By default, record types are immutable by default. While you could essentially create an immutable class but using a combination of `init`-only setters and read-only properties, record types remove that extra work.



```
class Forretning
{
    private record Person(string FirstName, string LastName, string[] PhoneNumbers);

    public static void Main()
    {
        var phoneNumbers = new string[2];
        Person person1 = new("Anders", "And", phoneNumbers);
        Person person2 = new("Joakim V", "And", phoneNumbers);
        Console.WriteLine(person1 == person2); // False

        Person person3 = new("Anders", "And", phoneNumbers);
        person3.PhoneNumbers[0] = "12345678";
        Console.WriteLine(person1 == person3); // True

        Console.WriteLine(ReferenceEquals(person1, person2)); //False

    }
}
```

Init syntax (C# 9) can be used:



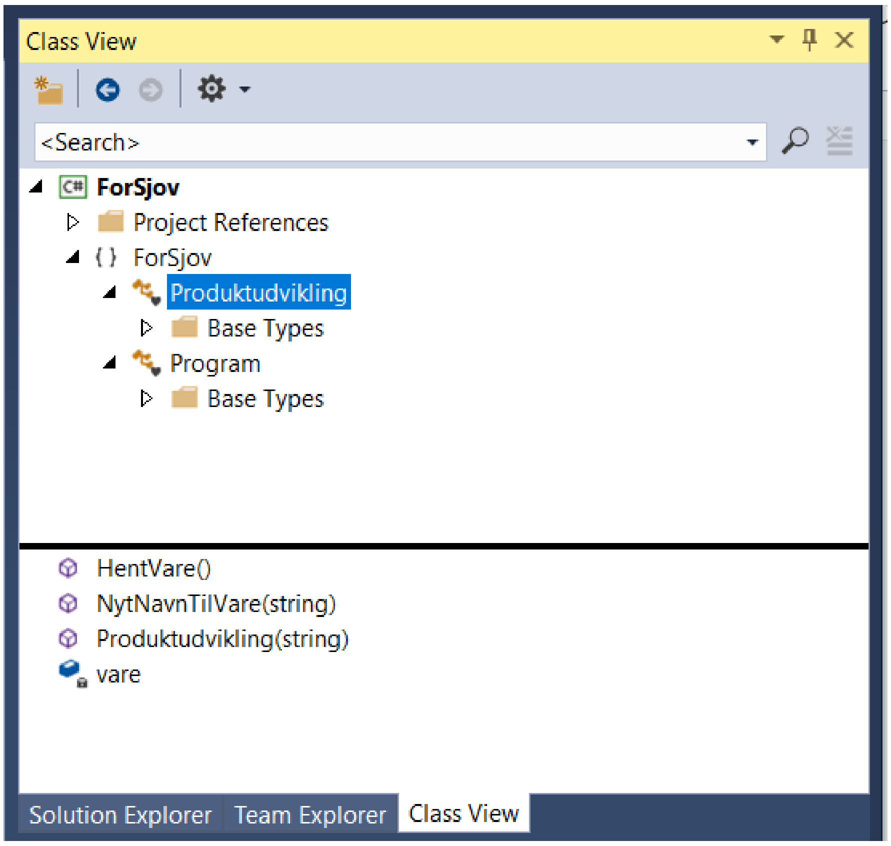
```
class Forretning
{
    private record Person(string FirstName, string LastName, string[] PhoneNumbers);

    record VehicleRecord
    {
        public string Type { get; init; }
        public string Model { get; init; }
        public string Color { get; init; }
        public  VehicleRecord() { }
        public  VehicleRecord(string type, string model, string color)
        {
            Type = type;
            Model = model;
            Color = color;
        }
    }
```

Use:

```
VehicleRecord myVehicle = new VehicleRecord
{
    Type = "Cykel",
    Model = "Fino",
    Color = "Grøn"
};


Console.WriteLine("Type:" + myVehicle.Type);
```

Inheritance – Chp. 6

Visual Studio Class View



```
◢ ForSjov
  ▷ Project References
  ◢ { } ForSjov
    ◢ Produktudvikling
      ▷ Base Types
    ◢ Program
      ▷ Base Types
```

```
HentVare()
NytNavnTilVare(string)
Produktudvikling(string)
vare
```

Relations

- ”is – a ” relations
- ”has – a” relations

- Base classes = generel information
  - Inheriting class = specialization

A small example:

```
using System;


public class Auto
{
    protected int autoType;

    public int AutoType
    {
        set{autoType=value;}
        get{return autoType;}

    }

   public Auto()
   {


   }
}
```

```
public class Skoda: Auto // INHERITANCE
{
   private int color;

   public int Color
   {
       set{color = value;}
       get{return color;}

   }

   public Skoda(int aType, int c)
   {
            autoType = aType;
            color    = c;

   }

}





public class UserAuto
{


   public static void Main()
   {

              Skoda bil = new Skoda(4,256);

              Console.WriteLine("color: {0}",bil.Color);
              Console.WriteLine("Type: {0}",
                                    bil.AutoType);
              Console.ReadLine();
   }
}
```

- If the base class constructor has arguments - call the constructor directly

Example:

```
using System;


public class Auto
{
    private int autoType;

    public int AutoType
    {
        set{autoType=value;}
        get{return autoType;}

    }

   public Auto(int aType)
   {
          autoType = aType;

   }
}


public class Skoda: Auto
{
   private int color;

   public int Color
   {
       set{color = value;}
       get{return color;}

   }

   public Skoda(int aType, int c) :base(aType)
   {

            color    = c;

   }


}
```

```
public class UserAuto
{


   public static void Main()
   {

              Skoda bil = new Skoda(5,122);

              Console.WriteLine("color: {0}",bil.Color);
              Console.WriteLine("Type: {0}",
                              bil.AutoType);

              Console.ReadLine();



   }



}
```

- As in Java and C++
  - The inheriting class calls the base class constructor

- A protected is private outside the inheritance hierarchy

To avoid further inheritance: sealed

```
public sealed class Skoda: Auto
{
   private int color;

   public int Color
   {
       set{color = value;}
       get{return color;}

   }

...
 public Skoda(int aType, int c)  :base(aType)
   {

              color    = c;

   }


}
...
```

- The class Skoda cannot be a base class
- Use it for utility classes, ex. the string class cannot be inherited

Inheritance – Records

```
  record Bus : VehicleRecord
 {
     public int Seating { get; init; }
     public Bus(string type, string model, string color, int seating) : base(type, model, color)
     {
         Seating = seating;
     }
 }
```

Use – as normal:

```
    Bus myBus = new Bus("DAB4D", "1960", "Red", 60);
    Console.WriteLine(myBus.Type);
}
```

Check for type:

```
Console.WriteLine($"Checking myBus is-a VehicleRecord:{myBus is VehicleRecord}");
// true
```

Two different records with same properties?

- Test with Equals(record1,record2)

"Has – a" classes (Containment/delegation)

Example:

```
// This new type will function as a contained class.
class BenefitPackage
{
// Assume we have other members that represent
// dental/health benefits, and so on.
public double ComputePayDeduction()
{
  return 125.0;
}
}


public partial class Employee
{
// Contain a BenefitPackage object.
protected BenefitPackage empBenefits = new BenefitPackage();
// Expose certain benefit behaviors of object.
public double GetBenefitCost()
{ return empBenefits.ComputePayDeduction(); }
// Expose object through a custom property.
public BenefitPackage Benefits
{
get { return empBenefits; }
set { empBenefits = value; }
}
...
}
```

Nested types: inner classes

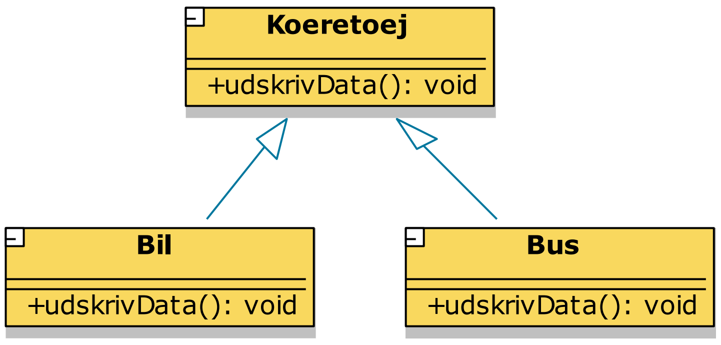
- A class can contain nested types
- Typical: Helper classes

```
using System;


public class Ydre

{
  private int y;
  public Indre indre;
  public int Y
  {
     get{return y;}
     set{y=value;}
  }

  public Ydre(int y,int i) {

     this.y = y;
     indre = new Indre(i);

  }

  public class Indre
  {

    private int i;

    public int I
    {
       get{return i;}
       set{i=value;}
    }

    public Indre( int i)

    {
        this.i = i;
    }

  }


}

public class Bruger
{


  public static void Main()

  {
      Ydre ydre = new Ydre(22,55);

      Console.WriteLine("Y={0}",ydre.Y);
      Console.WriteLine("I={0}",ydre.indre.I);
      Console.ReadLine();

  }


}
```

Polymorphy

- As in C++
    - Use **virtual** on the method of the base class
    - Inheriting class method uses **override**

Example:



Eks:

```
using System;

public class Koeretoej
{
    private int id;

    public Koeretoej( int i)
    {
         id = i;
    }
    public virtual void UdskrivData()
    {


     Console.WriteLine("K_ID {0}",id);
    }
```

```
}
public class Bil: Koeretoej
{
    private int bilId;

    public Bil(int kid, int bid): base(kid)
    {

                bilId = bid;
    }
    public override void UdskrivData()
    {
     base.UdskrivData();
     Console.WriteLine("BIL_ID {0}",bilId);
    }


}

public class Bus: Koeretoej
{
    private int busId;

    public Bus(int kid, int bid): base(kid)
    {

                busId = bid;
    }

    public override void UdskrivData()
    {
     base.UdskrivData();
     Console.WriteLine("BUS_ID {0}",busId);
    }


}

public class Userclass
{


  public static void Main()

   {
    Bil bil = new Bil(1,3);
    Bus bus = new Bus(2,4);
    bil.UdskrivData();
    bus.UdskrivData();

    Console.ReadLine();

  }


}
```

Abstract classes: classes without instantiation

What if we tried as follows?

```
...

public static void Main()

  {
    // ERROR  (GIVES NO MEANING)
    Koeretoej k = new Koeretoej(23);
    k.UdskrivData();

    Bil bil = new Bil(1,3);
    Bus bus = new Bus(2,4);

    bil.UdskrivData();
    bus.UdskrivData();
    Console.ReadLine();

  }
...
```

To avoid this the base class Koeretoej must be abstract:

```
...

abstract public class Koeretoej
{
    private int id;

    public Koeretoej( int i)
    {
         id = i;
    }
    public virtual void UdskrivData()
    {

     Console.WriteLine("K_ID {0}",id);
    }

}

...
```

Now: We got the error:

```
l3_7.cs(63,19): error CS0144: Cannot create an instance of the abstract class
or interface 'Koeretoej'
```

Abstract methods

- As pure virtual in C++
- methods **that must be implemented** by the inheriting class

Example where there **NO DEMAND** about the implementation
(Problem:The Bus class DOES NOT implement the UdskrivData() method)

```
using System;

public abstract class Koeretoej
{
    private int id;

    public Koeretoej( int i)
    {
         id = i;
    }
    public virtual void UdskrivData()
    {

     Console.WriteLine("K_ID {0}",id);
    }



}
public class Bil: Koeretoej
{
    private int bilId;

    public Bil(int kid, int bid): base(kid)
    {

                 bilId = bid;
    }
    public override void UdskrivData()
    {
     base.UdskrivData();
     Console.WriteLine("BIL_ID {0}",bilId);
    }


}

public class Bus: Koeretoej
{
    private int busId;

    public Bus(int kid, int bid): base(kid)
    {

                busId = bid;
    }


}

public class Userclass
{


  public static void Main()

  {

    Bil bil = new Bil(1,3);
    Bus bus = new Bus(2,4);

    bil.UdskrivData();


    Console.ReadLine();

  }


}
```

- Now demanding the implementation:
  - The method must be **abstract** without a body
  - Inheriting classes now **<u>must</u>** implement UdskrivData (Otherwise the class is regarded abstract)

Example:

```
using System;

public abstract class Koeretoej
{
    private int id;

    public Koeretoej( int i)
    {
         id = i;
    }
    // UdskrivData is abstract
    // Notice the semicolon..
    public abstract void UdskrivData();


}
public class Bil: Koeretoej
{
    private int bilId;

    public Bil(int kid, int bid): base(kid)
    {

                bilId = bid;
    }
    public override void UdskrivData()
    {
          Console.WriteLine("BIL_ID {0}",bilId);
    }


}

public class Bus: Koeretoej
{
    private int busId;

    public Bus(int kid, int bid): base(kid)
    {

                busId = bid;
    }


}
```

```
public class Userclass
{


  public static void Main()

  {

    Bil bil = new Bil(1,3);
    Bus bus = new Bus(2,4);

    bil.UdskrivData();


    Console.ReadLine();

  }


}
```

Now we got the following error:

```
l3_8.cs(33,14): error CS0534: 'Bus' does not implement inherited abstract
member  'Koeretoej.UdskrivData()'
```

Abstract methods can only be defined in abstract classes!
Pure protocols!

Another example: Polymorphy

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PolymorfEksempel
{
    class Program
    {
        static void Main(string[] args)
        {

            Shape[] shapes = new Shape[2];
            shapes[0] = new Firkant(2.2, 3.5);
            shapes[1] = new Cirkel(6.5);

            for(int i = 0; i<shapes.Length;i++)
                Console.WriteLine(shapes[i].CalculateArea());

            Console.ReadLine();

        }
    }


    public abstract class Shape
    {

        public abstract double CalculateArea();
        // Force all inheriting classes to implementation

    }

    public class Firkant: Shape
    {
        private double x;
        private double y;
        public Firkant(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public  override double CalculateArea()
        {
            return x * y;
        }

    }

    public class Cirkel : Shape
    {
        private double r;
        public Cirkel(double r)
        {
            this.r = r;
        }

        public  override double CalculateArea()
        {
            return (r * r)*Math.PI;
        }



    }


}
```

Member shadowing

- To prevent use of an inherited method
  - but reuse the same name
  - use **new**

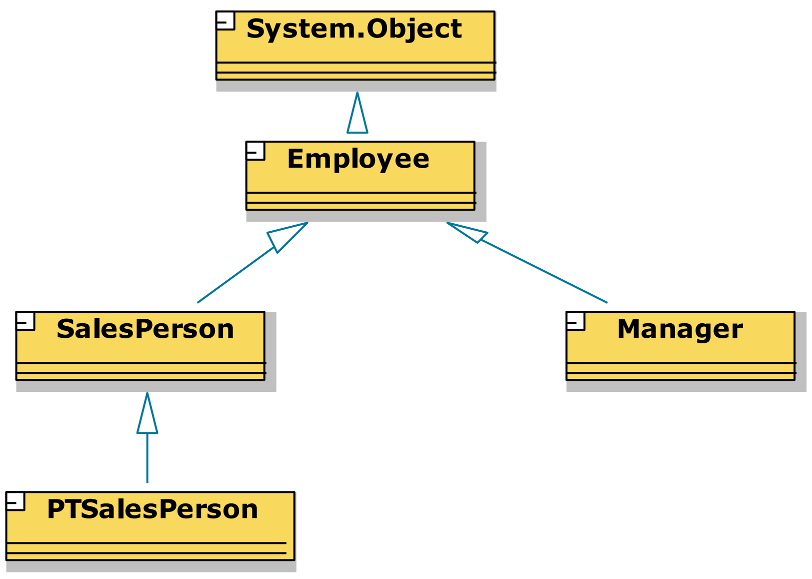
Example:

```
...
public class Oval: Circle
{
  public new void Draw()
   {

   }

}

...
```

Type converting

- A type cast in C# as in Java and C++ (old style)

From the book:



Example:

A Part Time sales person ”is –a” sales person:

SalesPerson jill = new PTSalesPerson(args);

A Manager “is-an” object

object frank = new Manager(args);

It is possible to make a reference to the inheriting class with the base class

Therefore :

```
public class MoveBusinessAbroad {

     public static void FirePerson( Employee e)
    {
          // extra..
    }


}
...
MoveBusinessAbroad.FirePerson(jill);
// BUT not a System.Object
// MoveBusinessAbroad.FirePerson(frank);
```

- You cannot treat a System.Object as an Employee
- Demands explicit type converting

```
...
// explicit type converting
Manager mgr = (Manager) frank;
...
```

The is and as keywords

”is”:

```
public class MoveBusinessAbroad {



     public static void FirePerson( Employee e)
    {
         if(e is SalesPerson)
         {
            // something

         }
         if(e is Manager)
         {
            // something else
         }
     }

}
...
```

”is” UPDATE C# 7: Convert type to a variable (if the cast works)

```
…
public class MoveBusinessAbroad {



     public static void GivePromotion( Employee e)
    {
         if(e is SalesPerson s)
         {
            // assign to variable s
         }

     }

}
...
```

“Is” DISCARDS (new in C# 7)   - CATCH ALL

```
...

         if(e is var _)
         {
            // match everything - so watch out..
         }

...
```

Pattern Matching (new in C# 7)

(switch statements)

...

case SalesPerson s when s.SalesNumber > 5:
...

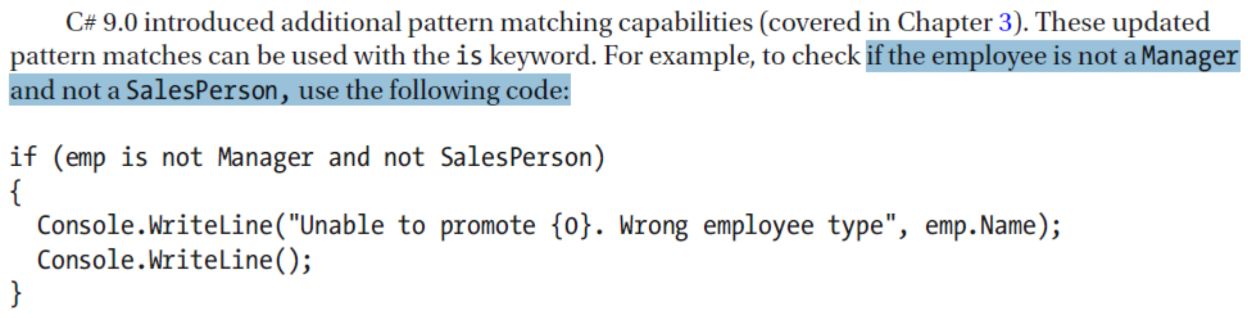
DISCARDS can also used in switch statements (new in C# 7)

. . .

case Intern _:
. . .

”as” :

```
...
SalesPerson p = e as SalesPerson;
if( p!= null)
 Console.WriteLine(“ Salg {0}”,p.NumbSales);
...
```



C# 9.0 introduced additional pattern matching capabilities (covered in Chapter 3). These updated pattern matches can be used with the is keyword. For example, to check if the employee is not a Manager and not a SalesPerson, use the following code:

```
if (emp is not Manager and not SalesPerson)
{
  Console.WriteLine("Unable to promote {0}. Wrong employee type", emp.Name);
  Console.WriteLine();
}
```

Object class

- the highest class in .NET
- The compiler will implicit let a class inherit Object

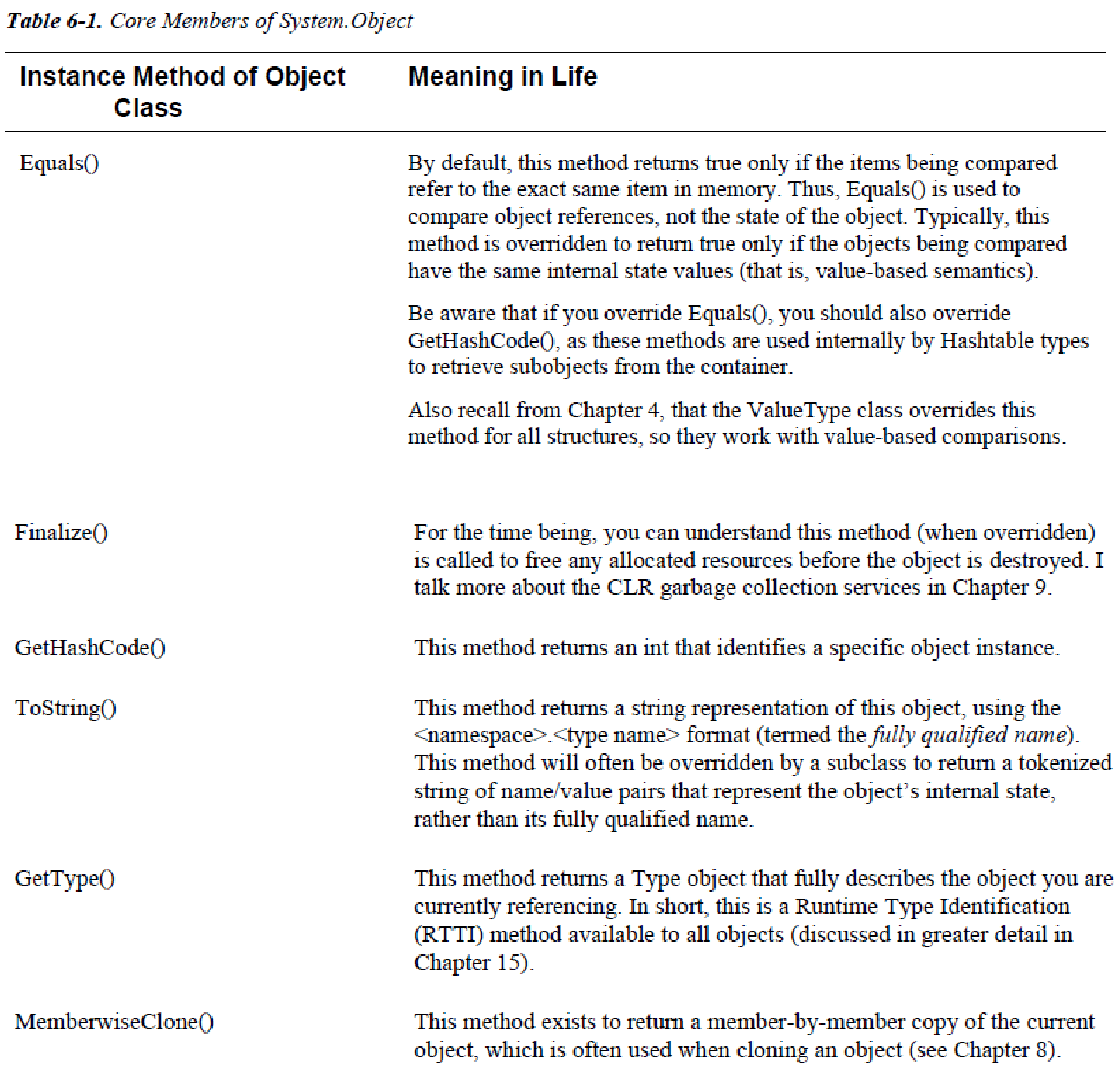


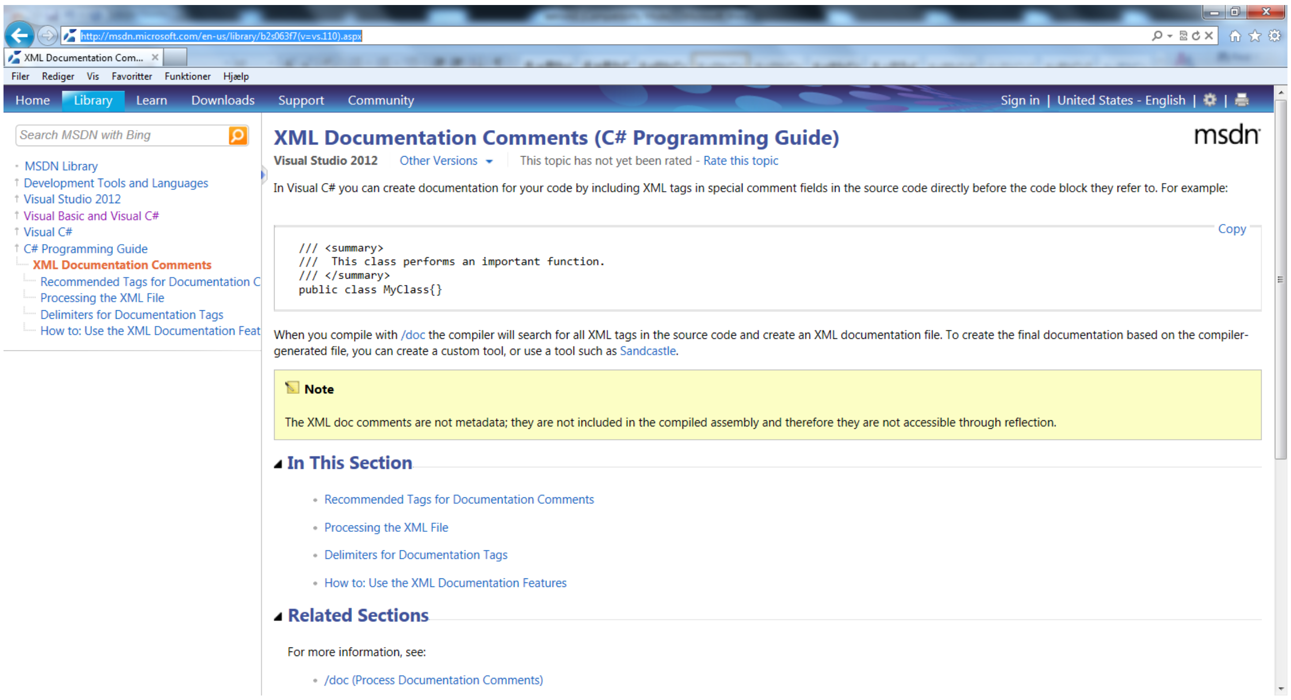
Table 6-1. *Core Members of System.Object*

Instance Method of Object Class	Meaning in Life
Equals()	By default, this method returns true only if the items being compared refer to the exact same item in memory. Thus, Equals() is used to compare object references, not the state of the object. Typically, this method is overridden to return true only if the objects being compared have the same internal state values (that is, value-based semantics). Be aware that if you override Equals(), you should also override GetHashCode(), as these methods are used internally by Hashtable types to retrieve subobjects from the container. Also recall from Chapter 4, that the ValueType class overrides this method for all structures, so they work with value-based comparisons.
Finalize()	For the time being, you can understand this method (when overridden) is called to free any allocated resources before the object is destroyed. I talk more about the CLR garbage collection services in Chapter 9.
GetHashCode()	This method returns an int that identifies a specific object instance.
ToString()	This method returns a string representation of this object, using the <namespace>.<type name> format (termed the *fully qualified name*). This method will often be overridden by a subclass to return a tokenized string of name/value pairs that represent the object's internal state, rather than its fully qualified name.
GetType()	This method returns a Type object that fully describes the object you are currently referencing. In short, this is a Runtime Type Identification (RTTI) method available to all objects (discussed in greater detail in Chapter 15).
MemberwiseClone()	This method exists to return a member-by-member copy of the current object, which is often used when cloning an object (see Chapter 8).

XML documentation

- XML dokumentation
- Looks like javadoc

- See XML Documentation Comments (C# Programming Guide) | Microsoft Docs


http://msdn.microsoft.com/en-us/library/b2s063f7(v=vs.110).aspx
XML Documentation Com...
Filer Rediger Vis Favoritter Funktioner Hjælp
Home Library Learn Downloads Support Community
Sign in | United States - English |
Search MSDN with Bing
msdn

- MSDN Library
- Development Tools and Languages
- Visual Studio 2012
- Visual Basic and Visual C#
- Visual C#
- C# Programming Guide
- XML Documentation Comments
  - Recommended Tags for Documentation C
  - Processing the XML File
  - Delimiters for Documentation Tags
  - How to: Use the XML Documentation Feat

XML Documentation Comments (C# Programming Guide)

Visual Studio 2012   Other Versions   This topic has not yet been rated - Rate this topic

In Visual C# you can create documentation for your code by including XML tags in special comment fields in the source code directly before the code block they refer to. For example:

Copy

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass{}
```

When you compile with /doc the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as Sandcastle.

Note

The XML doc comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In This Section

- Recommended Tags for Documentation Comments
- Processing the XML File
- Delimiters for Documentation Tags
- How to: Use the XML Documentation Features

Related Sections

For more information, see:

- /doc (Process Documentation Comments)