

## Object Oriented Programming in C# 2:2

# Topics

- Questions to the lesson 2 exercises
- Static and non-static class members
- Derived classes (inheritance)
  - Class hierarchy
  - Polymorphism
  - Overriding methods
  - Type casting
  - The Object class
- Access modifiers (overview)
- Interfaces
- The Heap and the Stack

# Static and non-static class members

# Static and non-static member in a class

- Class **member** are: **fields**, **properties**, **methods** and **events**
- If a member is **static**, it belongs to the **class**
- If a member is **non-static**, it belongs to a **specific object** of the class
- Static members are defined with the keyword **static**
- Members who's **data** or **behavior** is the same for all instances of the class are suitable candidates for static members of the class
- **Example>>**

```
public class Product {  
    // is the same for all instances of Product  
    public static decimal Moms = 0.25M;  
  
    // Automatic properties  
    public string Name { get; set; }  
    public double Price { get; set; }  
}  
  
// danish consumption tax  
decimal myMoms = Product.Moms; // 0.25  
Product.Moms = 0.2M; // converts to decimal  
myMoms = Product.Moms; // 0.2
```

```
public class Product {  
    // is the same for all instances of Product  
    private static decimal moms = 0.25M;  
  
    // Automatic properties  
    public string Name { get; set; }  
    public double Price { get; set; }  
  
    // Static read-only property  
    public static decimal Moms { get {return  
        moms;} }  
}  
  
// danish consumption tax  
decimal myMoms = Product.Moms;  
Product.Moms = 0.2M; // error
```

# Examples from Framework

```
// Examples of static properties
```

```
Math.PI;
```

```
DateTime.Now;
```

```
// Examples of static methods
```

```
Math.Min(2.22, 4.44);
```

```
DateTime.IsLeapYear(2028);
```

# A user defined **Time** class

```
public class Time {  
    private static int minPerDay=1440; // field  
  
    public int Hour { get; set; } // property  
    public int Min { get; set; } // property  
    public static int MinPerDay { // static property  
        get {return minPerDay; }  
    }  
    // constructor  
    public Time(int min, int hour) {  
        Min = min;  
        Hour = hour;  
    }  
}
```



- As the `int` field, **MinPerDay**, is static it is accessed through the class-name:

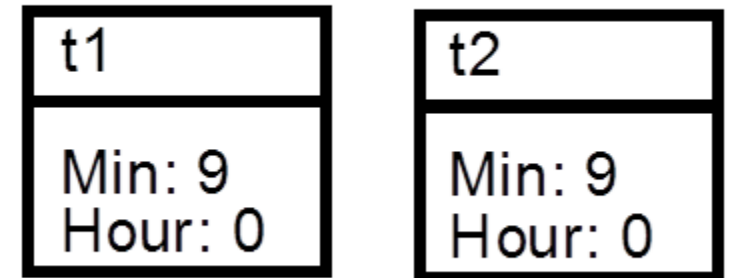
```
int m = Time.MinPerDay;  
Time t1 = new Time(9,0);  
Time t2 = new Time(45, 12);
```

- **MinPerDay** is independent of **t1** and **t2**, it is always 1440, therefore ~~**t1.MinPerDay**~~ and ~~**t2.MinPerDay**~~ are not meaningful, but **Time.MinPerDay** are.

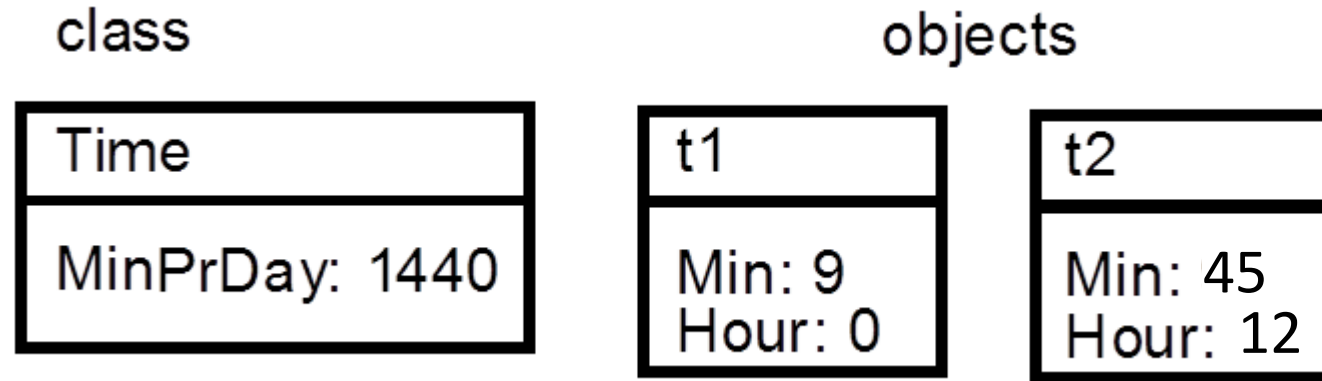
- The properties **Hour** and **Min** are non-static
- They belongs to a specific object of the class

```
int m1 = t1.Min;  
int h1 = t1.Hour;  
  
int m2 = t2.Min;  
int h2 = t2.Hour;
```

objects



# Illustration



- The property **MinPerDay** is static and belongs to the class
- The properties **Min** and **Hour** are non-static and belongs to the object(s)
- 1440 belongs to **Time** class, 9 and 0 to **t1** object, 12 and 45 to **t2** object

# More examples from Framework

- **DateTime** has a **static** method **DaysInMonth**:

```
int n = DateTime.DaysInMonth(2015, 2);
```

The method returns the **number of days** in month 2 in the year 2015, that is 28

- **DateTime** has **non-static** members for **year**, **month** and **day**

```
DateTime dt = new DateTime(2010, 9, 14);
```

# More examples from Framework

- **DateTime** has a non-static method **AddDays**:

```
DateTime d2 = d1.AddDays(20);
```

```
DateTime d1 = DateTime.Now;  
DateTime d2 = d1.AddDays(20);  
int n = DateTime.DaysInMonth(2009, 2);  
int n = d2.Year;
```

## Questions

- Why is **Now** static?
- Why can't **AddDays** be static?
- Why is **DaysInMonth** static?
- Why is **Year** non-static?

# Inheritance

Derived classes

# Inheritance

- The purpose of inheritance is to avoid, to write the same code more than once (avoid redundancy)
- You can declare classes that **inherits members** (fields, properties, methods, and events) from another class: the **base class** (or **super class**).
- This way the **general code**, which applies to all classes, is declared in the **base class** (or **superclass**) while the **special code** is defined in one or more **derived classes** (or **subclasses**).



# Inheritance: **Person** -> **Employee**

```
public class Person { // Person is the base class
    protected string firstname; // fields
    protected string lastname;
    public string Firstname { // properties
        get { return firstname; }
        set { firstname = value; }
    }
    ...
    public Person(string firstname, string lastname) { // constructor
        this.firstname = firstname;
        this.lastname = lastname;
    }
}
```

# Keywords for accessibility

Keyword	Accessibility
<b>public</b>	Can be accessed by any class
<b>private</b>	Can be accessed only by members inside the current class
internal	Can be accessed by members in any of the classes in the current assembly (the compiled code file)
<b>protected</b>	Can be accessed by members in the current class or in any class that inherits from this class
protected internal	Can be accessed by members in the current application (as with internal) and by the members in any class that inherits from this class

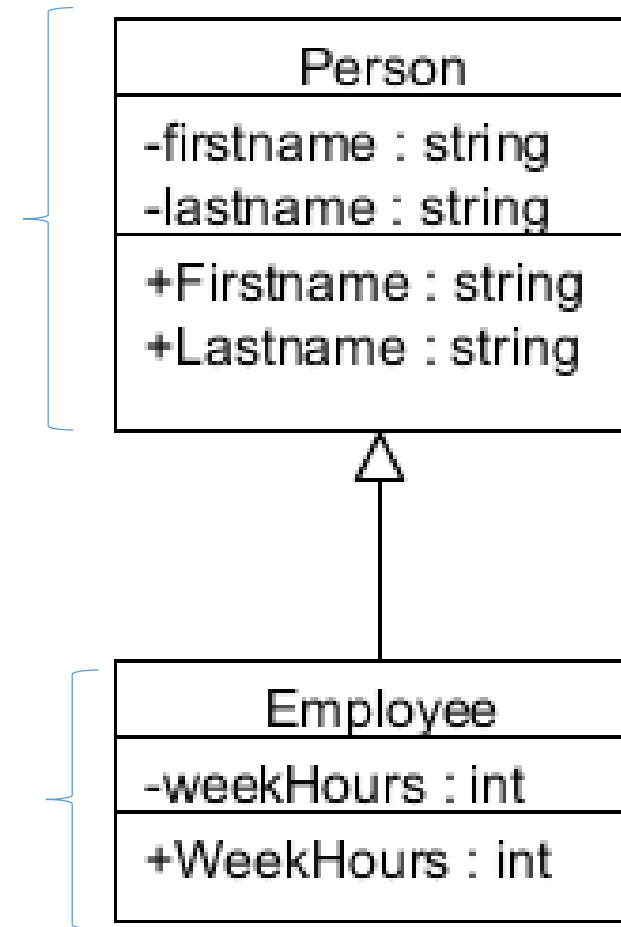
# An example of inheritance

- **Person** (*Person.cs*)
- **Employee** (*Employee.cs*)

The convention is each class in its own file with the class name as file name.

**Base class**

**Derived class**



```
public class Employee : Person
{
    private int weeklyHours;
    public int WeeklyHours {
        get {return weeklyHours;}
        set {weeklyHours = value;}
    }

    public Employee(string firstname, string lastname,
        int weeklyHours) : base(firstname, lastname)
    {
        this.weeklyHours = weeklyHours;
    }
}
```

*Call the base class constructor*



- **Person** is called **base class** and **Employee** is called **derived class**
- **Employee** inherits every member from **Person** and **adds some new**.
  - It **inherits** two fields and two properties:  
**firstname** , **lastname** , **Firstname** and **Lastname**
  - and **adds** a field and a property:  
**weeklyHours** and **WeeklyHours**

Person
-firstname : string
-lastname : string
+Firstname : string
+Lastname : string

Employee
-firstname : string
-lastname : string
-weekHours : int
+Firstname : string
+Lastname : string
+WeekHours : int

An Employee is a Person

A Person **is not necessarily** an Employee

```
Person p1 = new Person("Susan", "Thompson");
Employee e1 = new Employee("Bob", "Simon",
37);
```

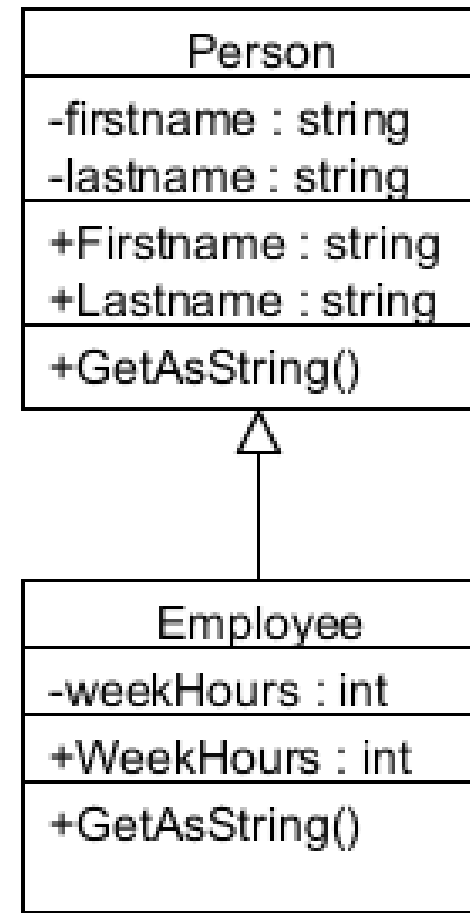
```
string p1Name = p1.Name;
int p1Weekhour = p1.WeeklyHours; // error
string e1Name = e1.Name;
int e1Weekhours = e1.WeeklyHours; // ok
```

# Benefit of inheritance

- **Re-use** of code:
  - In our example we can use everything programmed in **Person** without copying it to **Employee**
  - If we make some **improvements** of **Person**, all **Employee** objects automatically **inherits** it.

# Polymorphisme

- The term polymorphism comes from greek and means **many forms**
- With polymorphism methods of the base class can be **redefined** in the derived class





```
Person p1; Person p2;

p1 = new Person("Susan", "Thompson");
// GetAsString() in Person
string s1 = p1.GetAsString();

// GetAsString() in Employee
p2 = new Employee("Bob", "Hudson", 37);
string s2 = p2.GetAsString();
```

**p1** is a **Person**

**p2** is a **Person** but it is also an **Employee**

- In this example, we want to call the method in the class of the actual object (the one created with new) and not what the reference-variable is declared as:

```
Person p1; Person p2;  
  
p1 = new Person("Susan", "Thompson");  
// GetAsString() in Person  
string s1 = p1.GetAsString(); // GetAsString() in Person  
  
// GetAsString() in Employee  
p2 = new Employee("Bob", "Hudson", 37);  
string s2 = p2.GetAsString(); // GetAsString() in Employee
```

- This is achieved with **the keywords**:
  - **virtual** (in the super class)
  - **override** (in the sub class)

# Overriding methods in **base** class

In the **Person** class

```
public virtual string GetAsString()  
{  
    string s = "Person: " + firstname  
               + " " + lastname;  
    return s;  
}
```

# Overriding methods in the **derived** class

- In the Employee class

```
public override string GetAsString()  
{  
    string s = "Employee: ";  
    s += firstname;  
    s += " " + lastname;  
    s += " [" + weeklyHours + "]" ;  
    return s;  
}
```

# List example 1:2

```
Person p1, p2, p3;  
p1 = new Person("Susan", "Taylor");  
p2 = new Employee("Bob", "Stern", 20);  
p3 = new Employee("Tina", "Raymond", 37);  
  
List<Person> persons = new List<Person>();  
persons.Add(p1);  
persons.Add(p2);  
persons.Add(p3);  
ViewPager.Persons = persons;
```

Example01

# List example 2:2

```
List<Person> persons = ViewBag.persons;
```

```
<ul>
```

```
@foreach (Person p in persons ) {
```

```
    <li>@p.GetAsString()</li>
```

```
}
```

```
</ul>
```

Example01

- Person: Susan Taylor
- Employee: Bob Stern [20]
- Employee: Tina Raymond [37]

```
Person p1 = new Person(...);  
Person p2 = new Employee(...);  
// Error  
// Employee e = new Person(...);  
Employee e = new Employee(...);
```

- **Rule:** A Person-variable can point to an Employee object (because it is also a Person), but an Employee variable *can't* point to a Person (because a Person **might not be** an Employee)

# The **object** class

- **Any** class we define and any class in the Framework **is a derived class** (subclass) of a special class **Object**
- We create new objects without specifying the inheritance from **Object**. **This is implicit.**
- We don't write (although we could):

```
public class Person: Object {  
    ...  
}
```



# **ToString()** method

- **Object** has a few methods, no fields and no properties.
- The most important is **ToString()**
- The **ToString()** method in **object** returns the string:

**System.Object**

- The **ToString()** method is declared **virtual**, meaning it is possible to override in subclasses>>

# Overriding **ToString()**

```
public class Person
{
    . . .
    public override string ToString()
    {
        return firstname + " " + lastname;
    }
}
```

# Example of use 1:2 (the controller)

- In a dropdownlist you might want to add a full name to a **SelectListItem**-object:

```
List<Person> persons = new List<Person>();
persons.Add(new Person(1, "Susan", "Taylor"));
persons.Add(new Employee(2, "Bob", "Stern", 20));

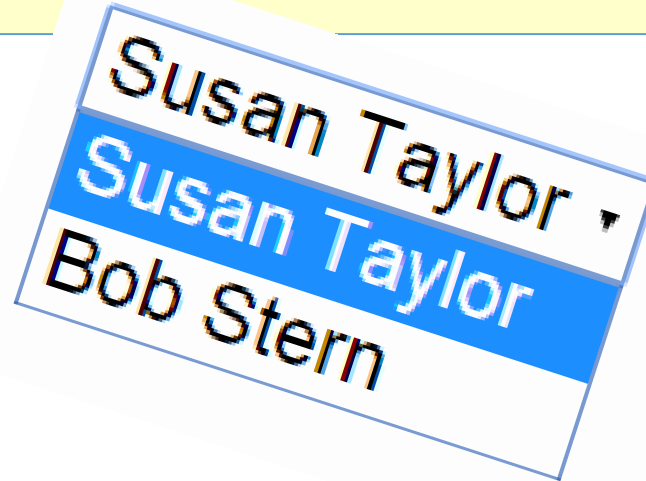
// Dropdown list
List<SelectListItem> items = new List<SelectListItem>();
foreach (Person p in persons) {
    items.Add(new SelectListItem {
        Text = p.ToString(),
        Value = p.PersonId.ToString() });
}
ViewBag.Persons = items;
```

Example02

# Example of use 2:2 (the view)

```
@Html.DropDownList("Persons", ViewBag.Persons  
as IEnumerable<SelectListItem>)
```

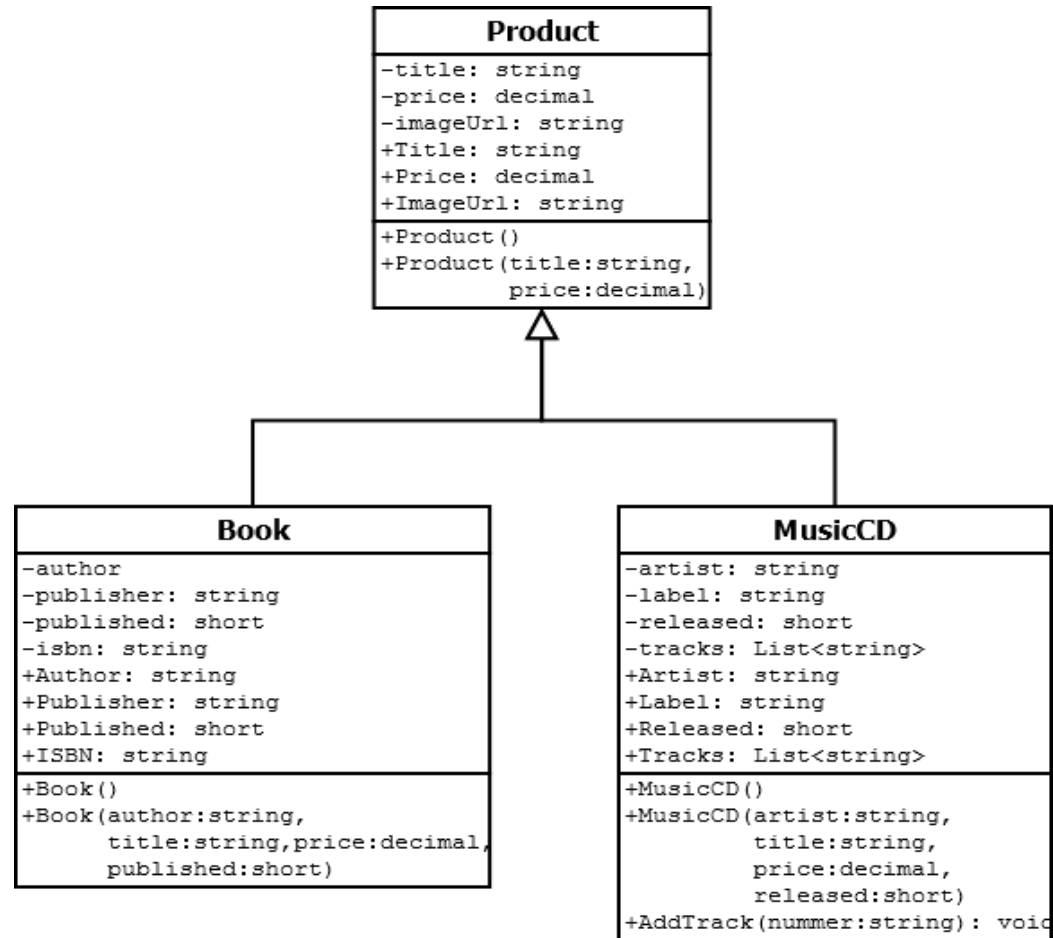
Example02



# Introduction to the exercises

**Base class**

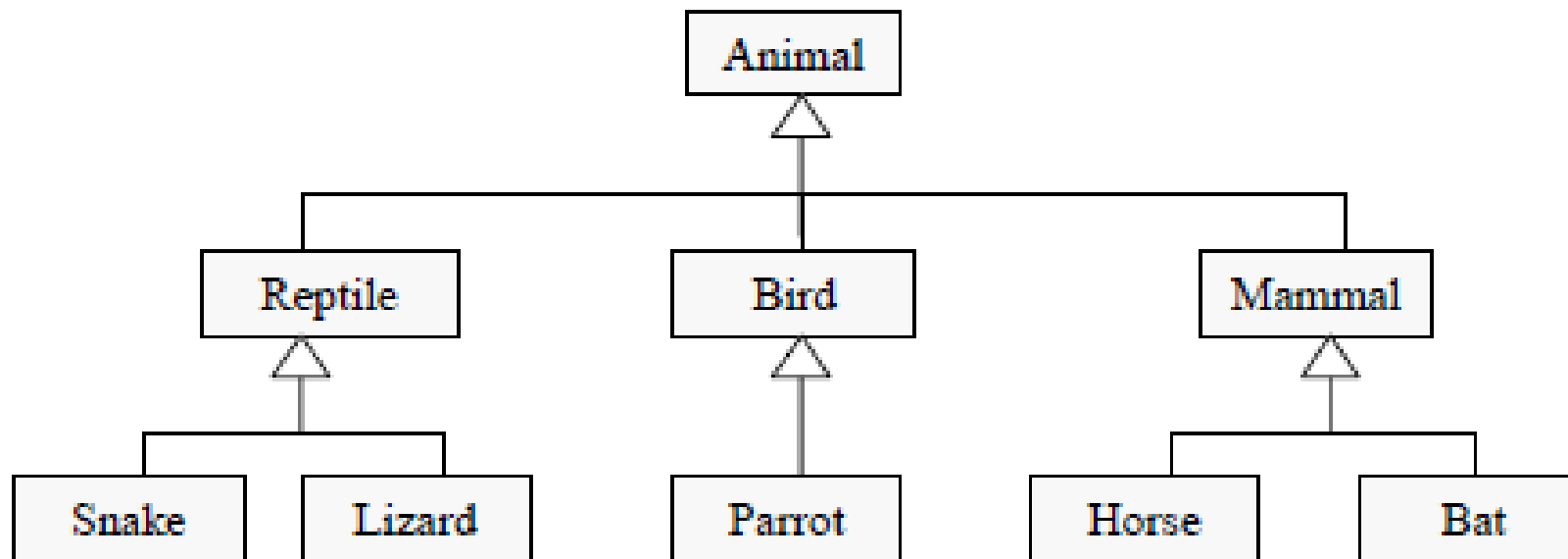
**Derived classes**

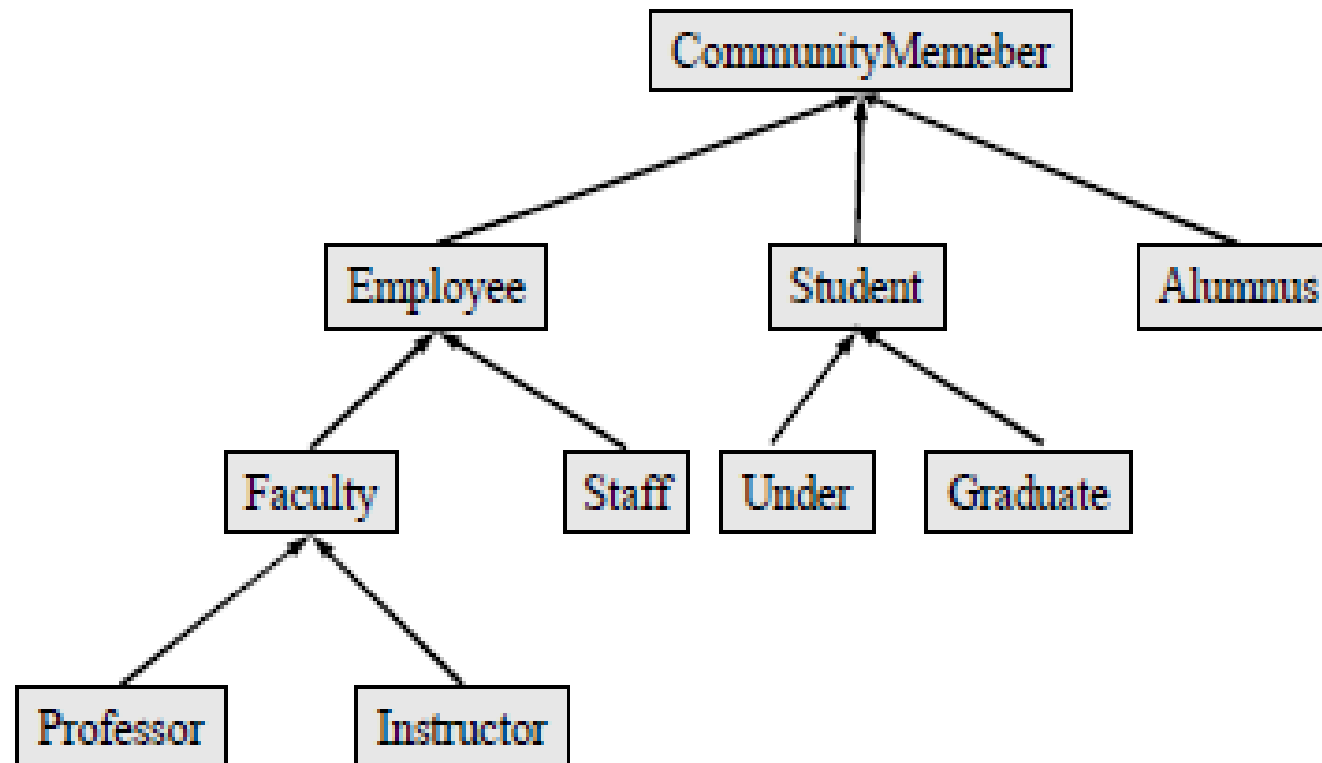


# Exercises 1

# Examples of class hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*







# Inheritance in Framework

- You will properly not need to write many class hierarchies yourself, but the **Framework has many class hierarchies**
- An understanding of class hierarchies is therefore important
- The following pages gives a few examples

# Inheritance in ASP.NET MVC

F12

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
        return View();
    }
}
```

Metadata

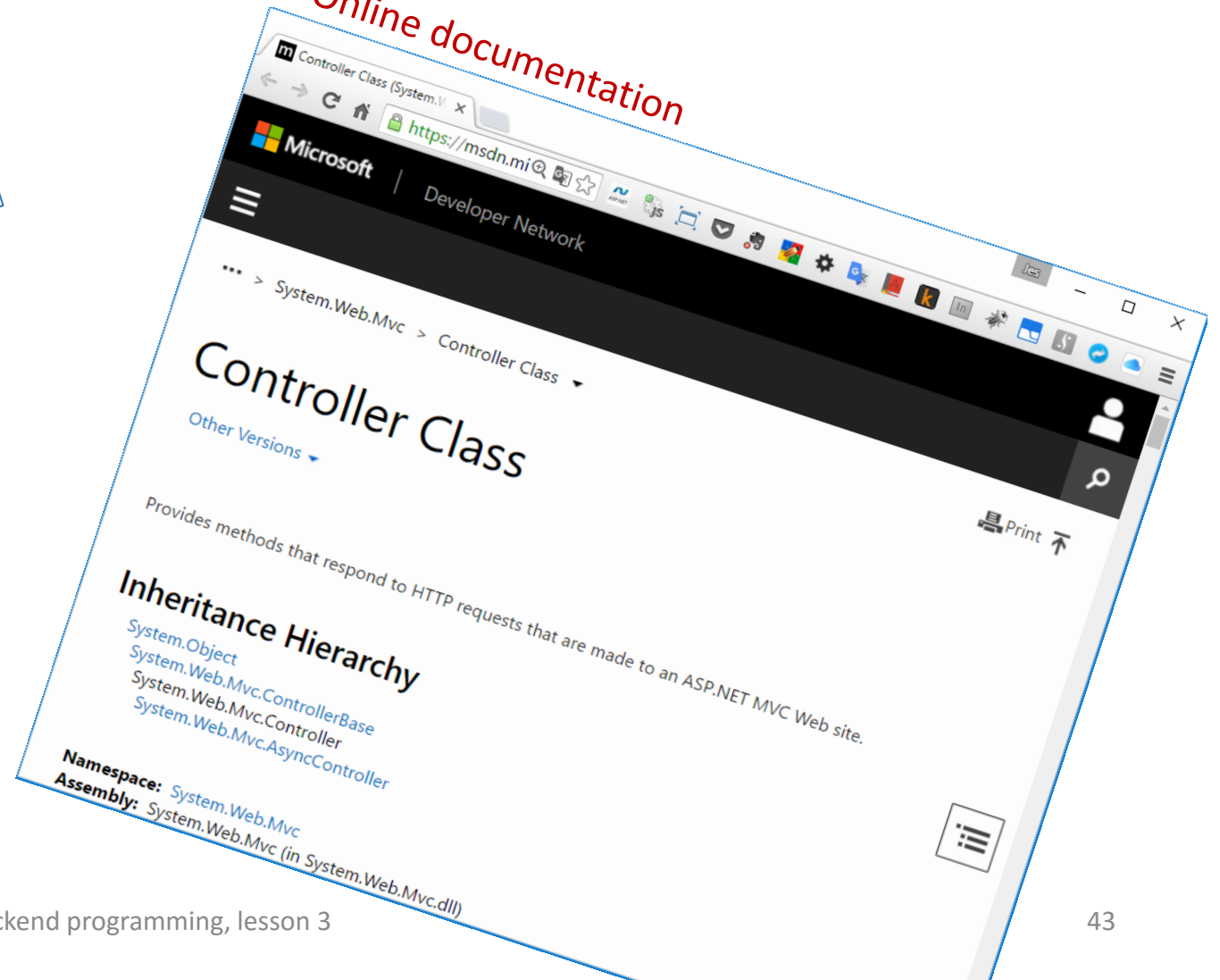
```
15 namespace System.Web.Mvc
16 {
20     ... public abstract class Controller : ControllerBase, IActionFilter, IAuthenticationFilter, IAuthorizationFilter, IDisposable,
23     ... protected Controller()
24     {
30         ... public IActionInvoker ActionInvoker { get; set; }
37         ... public AsyncManager AsyncManager { get; set; }
44         ... protected internal ModelBinderDictionary Binders { get; set; }
51         ... protected virtual bool DisableAsyncSupport { get; set; }
58         ... public HttpContextBase HttpContext { get; set; }
66         ... public ProfileDictionary ModelState { get; set; }
73         ... public IHttpRequestBase Request { get; set; }
80         ... public IResponseResolver Request { get; set; }
85         ... public IRouteData RouteData { get; set; }
92         ... public IRouteData Response { get; set; }
99         ... public IRouteData Response { get; set; }
107         ... public IRouteData Response { get; set; }
114         ... public IRouteData Response { get; set; }
122         ... public IRouteData Response { get; set; }
129         ... public IRouteData Response { get; set; }
136         ... public IRouteData Response { get; set; }
142         ... public IRouteData Response { get; set; }
}
```

# Documentation in ASP.NET MVC

F1

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
        return View();
    }
}
```

Online documentation



# Interface

# Interface

- Is a contract:
  - Defines a set of **properties**, **methods** and **events** that must be declared in all classes that implements that interface.
  - An interface **contains only the signatures** of methods, properties, events or indexers (no implementation).
  - A **class** or **structure** that **implements the interface** must implement the members of the interface that are specified in the interface definition
  - Programming against an interface makes it **easy to change the implementation** because classes that implement the interfaces all have the same members (methods, properties and events).

# Example

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

<http://msdn.microsoft.com/en-us/library/87d83y5b.aspx>

# Using the **IEnumerable** interface with a **strongly typed View** 1:2

```
List<Person> persons = new List<Person>();  
persons.Add(new Person(1, "Susan", "Taylor"));  
persons.Add(new Employee(2, "Bob", "Stern", 20));  
  
// return a strongly typed view  
return View(persons as IEnumerable<Person>);
```

Example03

- The IEnumerable interface is used with many collections such as List and Array, and also with LINQ.

# Using the **IEnumerable** interface with a **strongly typed View** 2:2

If we want a special string if a **Person** in the List is **Employee** it can be done like:

```
@using lesson03_examples.Models;
@model IEnumerable<Person>

@foreach (Person p in Model) {
    if (p is Employee) {
        Employee e = (Employee) p;
        <li>@e.ToString() (@e.WeeklyHours)</li>
    }
    else {
        <li>@p.ToString()</li>
    }
}
```

Example03



# Another Example (Julie Lerman)

```
using System;

namespace NinjaDomain.Classes.Interfaces
{
    public interface IModificationHistory
    {
        DateTime DateModified { get; set; }
        DateTime DateCreated { get; set; }
        bool IsDirty { get; set; }
    }
}
```

# Implementation

```
namespace NinjaDomain.Classes
{
    public class Ninja:IModificationHistory
    {
        public Ninja()
        {
            EquipmentOwned = new List<NinjaEquipment>();
        }
        public int Id { get; set; }
        public string Name { get; set; }
        public bool ServedInOniwaban { get; set; }
        public Clan Clan { get; set; }
        public int ClanId { get; set; }
        public List<NinjaEquipment> EquipmentOwned { get; set; }
        public DateTime DateOfBirth { get; set; }

        public DateTime DateCreated { get; set; }
        public DateTime DateModified { get; set; }
        public bool IsDirty { get; set; }
    }
}
```

# A remark 1

```
Employee e = (Employee) p;
```

Can also be written as:

```
Employee e = p as Employee;
```

**Check** if you're not sure that **Person** p is an **Employee**:

```
if (p is Employee) {  
    Employee e = (Employee) p;  
}
```

## A remark 2 – LINQ example

```
IEnumerable<Movie> movies = new List<Movie>();  
// add music, books and movies to the list  
// get the movies  
movies = Products.Of<Movie>().ToList();  
ViewBag.Movies = movies;
```

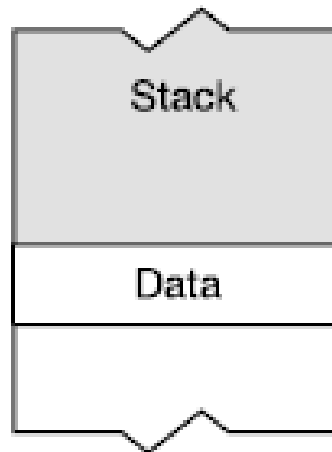
# Value Types and Reference Types

The Stack and the Heap

# Struct (like DateTime for example )

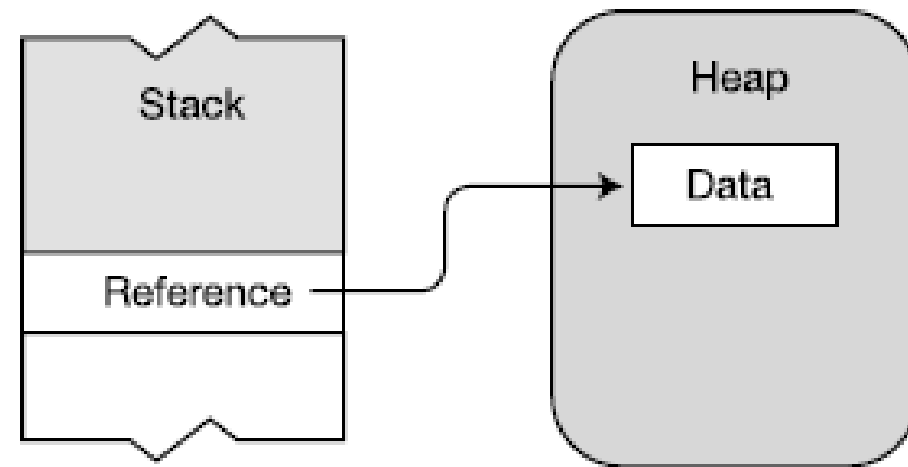
- **Classes** are Reference types and **Structures** are Values types.
- **Classes** support Inheritance while **Structures** don't.
- **Classes** can have explicitly parameterless constructors whereas **Structures** can't.
- Member variable initialization is possible in **Class** whereas in **Structures**, it is not.
- It is not possible to declare destructor in **Structure** but in **Class** it is possible.

# The stack and the heap



## Value Type Data

- The data is stored on the stack.



## Reference Type Data

- The data is stored in the heap.
- The reference is stored on the stack.

Kilde: Daniel Solis: Illustrated C# 2008, s. 41

# Value Types and Reference Types in C#

	Value Types			Reference Types
Predefined Types	sbyte	byte	float	object
	short	ushort	double	string *)
	int	uint	char	
	long	ulong	decimal	
	bool			
User-Defined Types	struct			class
	enum			interface
				delegate
				array

\*) Ligheds- og tildelingsoperatoren fungerer i forhold til variabelindhold, ikke som reference.

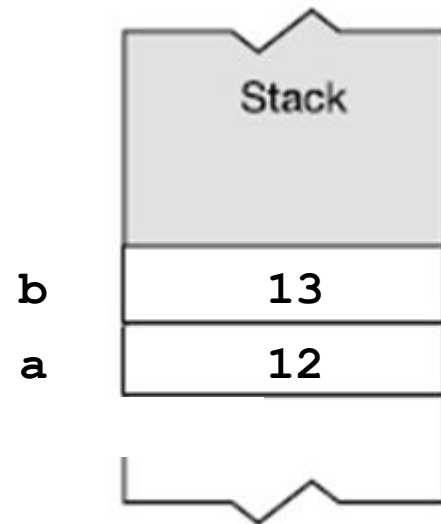
Kilde: Daniel Solis: Illustrated C# 2008, s. 41



# Tildelingsoperationer: Værditype

```
int a = 12;  
int b = a;  
b++; // b is 13 og a is still 12
```

# To referencer til to forskellige



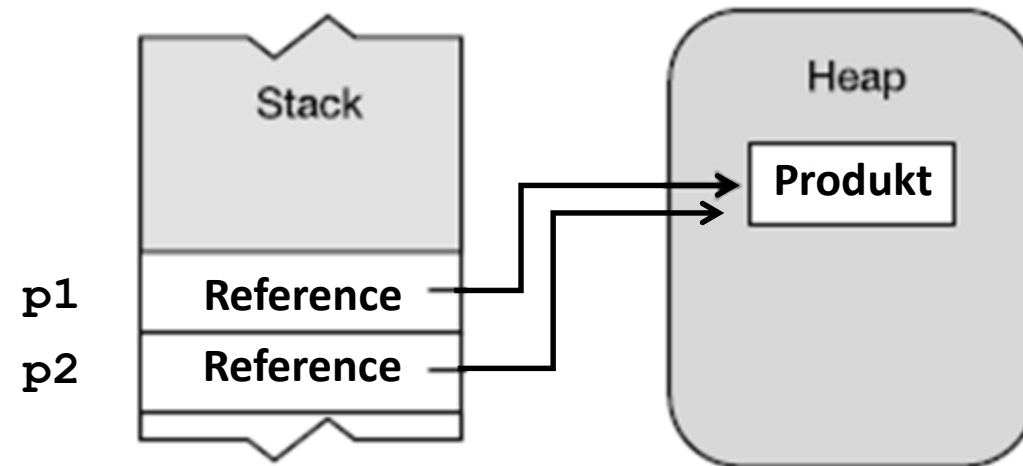
**Value Type Data**  
– The data is stored on the stack.

# Reference types

```
public class Produkt {  
    public decimal Price = 0.00M;  
}
```

```
Product p1 = new Produkt();  
Product p2 = p1;  
p2.Price = 12.50M; // p1.Price is 12,50
```

# To referencer til det samme objekt



# Info

- The 1st mandatory assignment is on Fronter

# Exercises 2-3