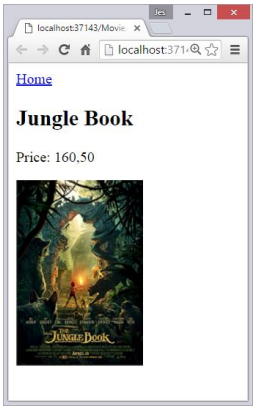
Lesson 2

Some of the exercises in this and upcoming lessons are part of the mandatory assignments. These lessons are marked with *Mandatory*.

Exercise 1, extending Movie demo (Mandatory)

Download `MbmStore.zip` from the Exercises folder. Extract and open the project in Visual Studio.

Open the website in a browser from inside Visual Studio (Ctrl+F5) and click at the "Exercise "1 link. This should give you a browser window similar to this:



Open `Models/Movie.cs` and inspect the Movie class. Which fields and properties does it have? Which controllers? Open the `Controllers/MovieController.cs` controller. What happens inside the index controller? Explain how data is sent to the view? What is the purpose of the view (`Views/Movie/Index.cshtml`)?

Make the following modifications of the code:

1. Open the `Controllers/ MovieController.cs` file and create two new instances of the Movie class with holds information about movies of you own choice.
2. Open `Views/Movie/Index.cshtml` and update the view to display the two new products
3. Open the `Models/Movie.cs` class file and change the `Title` property to a read-only property
4. Add a new private field of type string and with the name *director*
5. Add a property for this new field
6. Add *director* as a new parameter to the second constructor
7. When you now select Debug -> Build the website has a compile error, why?
8. Correct this error by calling the constructor inside the controller with an extra argument.
9. Modify the view to display all data from the new instances of Movie.
10. Run the application and check that the new information is displayed correctly.
11. As the final step, you must change the `ImageUrl` property to an auto-implemented property.

Exercise 2, *a Customer class* (Mandatory)

Add a Customer-class to the Models folder, which holds values of *firstname*, *lastname*, *address*, *zip*, *city*, and *phone* number.

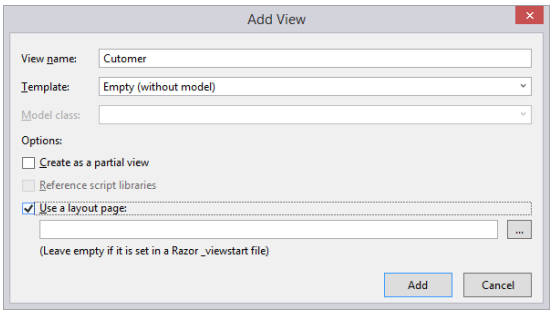
1. Create the necessary properties as automatic properties (all data types should be strings)

2. Add a constructor with this parameter list:
   `Cutomer(string firstnavn, string lastnavn, string address, string zip, string city)`

   Tip.: In Visual Studio, there are many shortcuts for inserting code snippets. They all start with a keyword followed by double tapping the tab key. The shortcut for inserting **automatic properties** is *prop* followed by double tapping *tab* and for creating a **constructor** it is *ctor* followed by *tab tab*. For a full list of Visual Studio code snippets, see https://msdn.microsoft.com/en-us/library/z41h7fat.aspx.

3. Add a new empty `Customer` controller and create a couple of `Customer` objects and save each of them as a `ViewBack` property.

4. Create a view for the Index action method of the Customer controller. It must be a view *without* model and *with* a layout page:



5. Insert *ViewBack* data and html and display information about customers as address labels:
   [*firstname*] [*lastname*]
   [*address*]
   [*zip*] [*city*]

   To write efficient and easily maintainable code, you must make it DRY (Don't Repeat Yourself). It is problem in this case because you must repeat the mark-up for each customer. Since ASP.NET MVC 3, you can solve this issue by declaring you own methods in Razor Views with the @helper syntax.

   Take a close look at this article: ASP.NET MVC 3 and the @helper syntax within Razor, and write a `RenderCustomer` helper method that renders a customer in accordance with the required layout.

Tip: Use `Customer` as parameter type in the `RenderCustomer` method:

```
@helper RenderCustomer (Customer customer) {
     ...
}
```

For the view to know of the `Customer` class, you must include a reference to that class at the beginning of the view file:

```
@using [YourProjectName].Models;
```

Exercise 3, *a list of phone numbers* (Mandatory)

1. Assign a phone number to the *phone* property of each customer in the controller class and update the view to display a customers list with phone numbers beneath the address labels:

   [*firstname*] [*lastname*] ([*phone*])

2. Open the `Customer.cs` file and delete the `Phone` propery. Add `phoneNumbers` and `PhoneNumbers` as respectively *field* and *property*. Instead of `sting`, the type must be a `List` collection of type `string`.

   The `PhoneNumbers` property must be read-only:

   ```
   // read only property
   public List<string> PhoneNumbers {
        get { return phoneNumbers; }
   }
   ```

   With this change, it is now possible to store more than one phone number for each customer.

   Tip:
   To make it possible to add new phone numbers to the list you must first instantiate the List as an object. You can do it as you declare `phone` as a field:

   ```
   List<string> phone = new List<string>();
   ```

1. Add a new method `AddPhone` with the header:

   ```
   public void addPhone(string phone)
   ```

2. Change the view to display a list of phone numbers for each customer:
   `<p>[firstname] [lastname] ([comma separated list of phone numbers])</p>`

   Tip:
   To display a comma-separated list of strings you can use the `join` method of the `string` object:

   ```
   string.Join (", ", theList);
   ```

   The first parameter is the string that separates the items in the `List`, and the second parameter is the `List` you want to join as a single string.

Exercise 4, *calculate age for birthday* (Mandatory)

1. Open the Customer class file and add a field `birthDate` and a property `BirthDate` of type `DateTime`.

2. Write a read-only property, `Age` that returns the age of the Customer.

   Tip: To calculate the age of a customer from birth date you can use this algorithm:

   ```
   DateTime birthDate; // field

   // code inside the get block of the Age property
   DateTime now = DateTime.Now;
   int age;


   age = now.Year - birthDate.Year;

   // calculate to see if the customer hasn't had birthday yet
   // subtract one year if that is so
   if (now.Month < birthDate.Month || (now.Month == birthDate.Month &&
   now.Day < birthDate.Day))

   {
       age--;

   }
   ```

3. Modify the `BirthDate` property to ensure that only dates that calculate an age between 0 and 120 are accepted.

   Tip: Use the `set` block of the `BirthDate` property to validate for a realistic age:

```
public DateTime BirthDate
{
   set {
        if (expression)
        {
           throw new Exception("Age not accepted");
        }
        else
        {
            birthDate = value;
        }
    }
    get { return birthDate; }
}
```

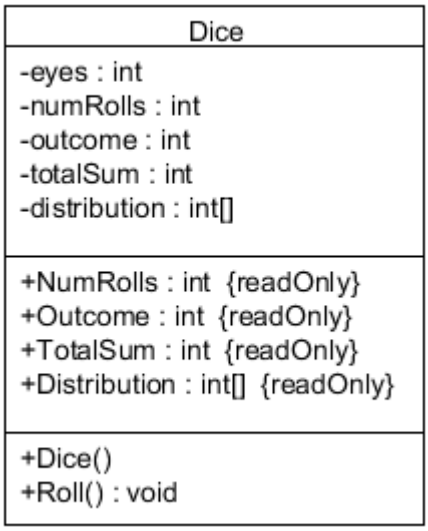
4. Test the code by assigning a birth date to one or more `Customer` objects.

5. Add Razor code to the view and display the age of one of the `Customer` objects, like: Peter Thompson is 24 years old

Exercise 5, Rolling Dices (Optimal)

In this exercise, you must simulate dice rolls, and keep track of the result of a sequence of rolls.

1. Create a new empty ASP.NET MVC project for this exercise and name it *lesson02* or *diceroll* as you prefer.
2. In the Models folder, create a new class `Dice` as an implementation of this UML-diagram:



Inside the constructor you must set the right initial values:

```
public Dice() {

        eyes = 6;
        numRolls = 0;
        totalSum = 0;
        distribution = new int[ojne];
  }
```

The `numRolls` is the total number of rolls of the dice. The `outcome` is what the dice landed at, and `totalSum` is the total sum of eyes in all dice rolls. The `distribution` field is used to keep track of how many times the dice landed on each of the possible outcomes (1, 2, 3, 4, 5, and 6):

Index = Eye minus 1	Outcome
0	Number of ones
1	Number of twos
2	Number of threes
3	Number of fours
4	Number of fives
5	Number of sixes

Write the Roll() method. Inside that method you must update the `outcome`, `numRolls`, `totalSum`, and `distribution` fields.

3. Create a new controller for the DiceRoll program.

   You must have two `Index` action methods. One for the initial request and one for post requests:

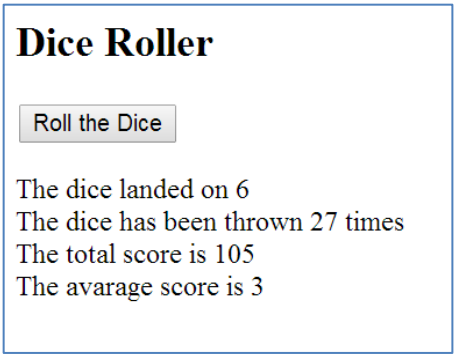
```
public ActionResult Index()
{
    return View();
}

[HttpPost]
public ActionResult Index(FormCollection fc) {
    return View();
}
```

   The overall result of dice rolls is stored in as a Dice object. To store that result between dice rolls you can use a session variable by inserting this code into the second action method reacting on HTTP POST requests:

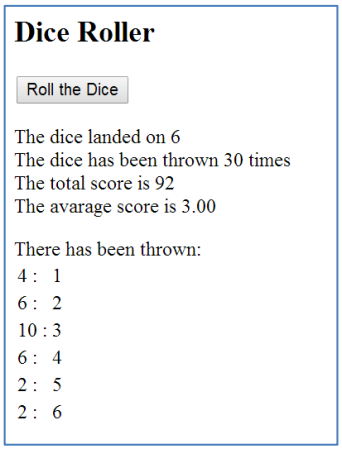
```
Dice dice;
if (Session["dice"] == null) {
   dice = new Dice();
   dice.Roll();
   Session["dice"] = dice;
}
else {
   dice = (Dice)Session["dice"];
   dice.Roll();
}
```

   If the session variable is not set, you create a new instance of `Dice`, roll the dice and store the result in the session variable named "`dice`". Otherwise, you read the `Dice` object from the session variable and roll the dice.

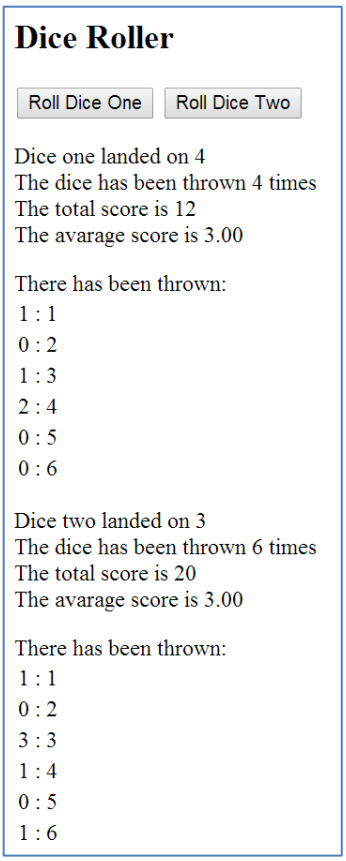
3. Create a view with a button that calls the controller, which rolls the Dice. Return a view whoch displays the result:



4. Update the `distribution` field, and display the distribution of eyes:



5. Create an new controller with a view and add one more button and a dice, and present the result of the two independent rolls:



/ Jes Arbov, 5 September 2016