Object Oriented Programming in C# 1:2

Topics

- Exercises from last week. Questions?

- Object Oriented Programming (OOP) 1:2
  - Use classes from the ASP.NET framework
  - Write you own classes
  - Use your own classes and objects in web applications

C# is 100% object oriented

- Ordinary data types like **`int`** or **`string`** is masked objects.

```
int i = 12; // object
string s = i.ToString(); // method
int j = s.Length; // property
```

Class and object

```
string s1 = "Apple Macbook Pro";
string s2 = "HP Compac 2570p";
```

- **`string`** is a class describing what is common for all string object (you can see the class and it's class members by highlighting the type and pressing F12 and the documentation by pressing F1)
- **`s1`** and **`s2`** are two examples of string-objects.
- **`s1`** and **`s2`** has different values: **`"Apple Macbook Pro"`** and **`"HP Compac 2530p"`**

Properties

- **`Length`** is a property of the string-class:

```
string s1 = "Apple Macbook Pro";
int n = s1.Length; // 17
```

  - The **`Length`** property returns the number of characters (Char objects) in the string of the object
  - **`n`** is the number of characters in the **`s1`**-object

Documentation: http://msdn.microsoft.com/en-us/library/system.string.length(v=vs.110).aspx

Properties

- You may set and/or get a property:

```
ViewContext.ClientValidationEnabled = true;
bool isJSValidation = ViewContext.ClientValidationEnabled;
```

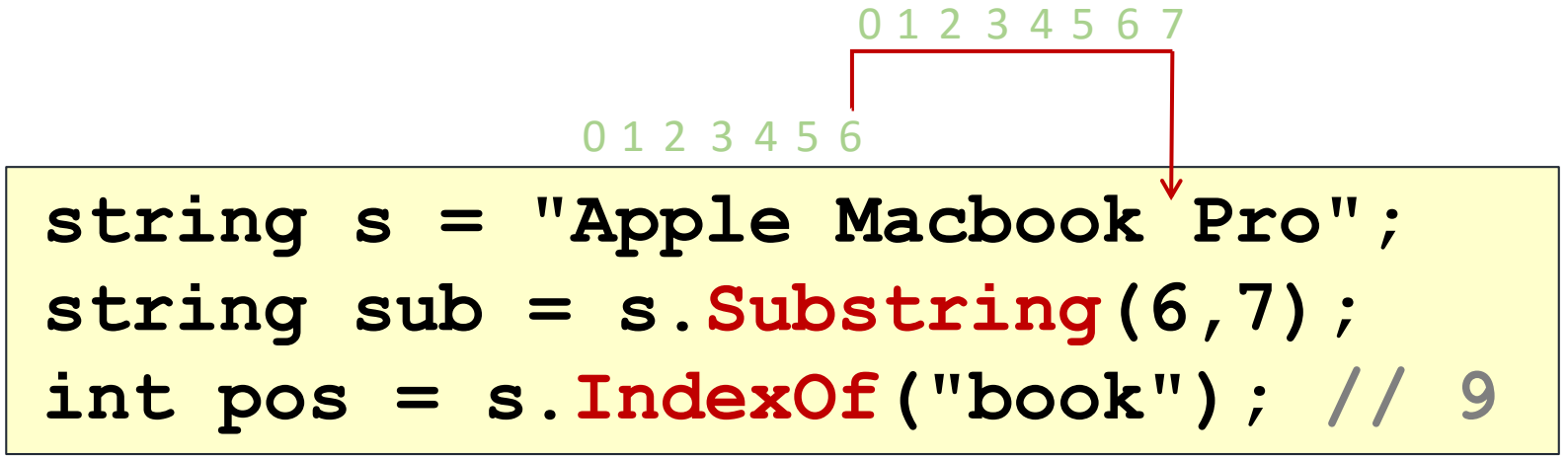
- **Length** is a read-only property. You can only get the value

- Meaning it can only occur on right side of an assignment:

```
int n = s1.Length; // OK
s1.Length = 10; // illegal
```

Methods

- A class can have methods doing ”calculations” = executing code:



- **`Substring`** is a method computing a substring. The result is "**Macbook**".
- **`IndexOf`** is a method computing the position of a substring (**`"book"`**) in the string object (**`"Apple Macbook Pro"`**) The result is 9.

Classes in the Framework

- The C# framework is a huge library of preprogrammed classes
- Some classes are for storing and manipulating data:
  - **`DateTime`** (to register a date and time, although **`DateTime`** is not a class but a **`struct`** which is a “lightweight class”),
  - **`String`** etc.
  - **`Random`** (for creating random numbers),
- Some classes a used for rendering a view
  - **`WebViewPage`**

- See http://msdn.microsoft.com/en-us/library/gg145045.aspx for the official documentation the class libraries

The Random class

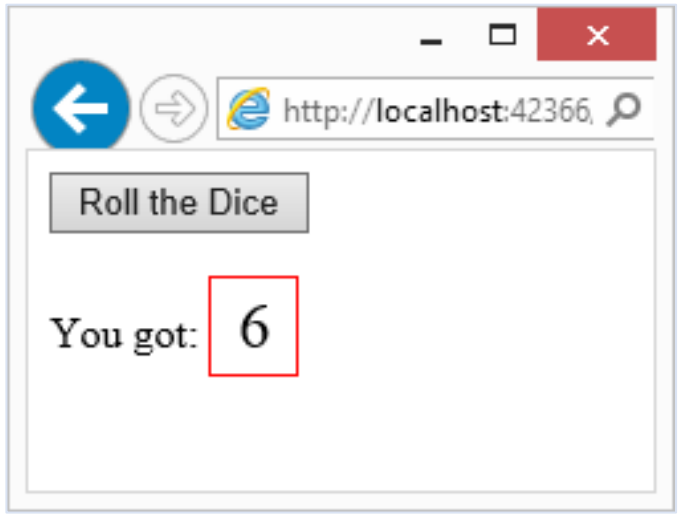
- Use of the **`Random`** class (emulates a dice)
- The first line creates a **`Random`** object. The next two calls methods on the object.

```
// Create a new Random obj.
Random rnd = new Random();

// Returns a random number between 1 and 7
(exclusive)
int i = rnd.Next(1, 7);

// Returns a random number between 0.0 and 1.0
(exclusive).
double d = rnd.NextDouble();
```

Try it!

Also: Explain BeginForm submit, post

1. Build a controller (**`DiceRoller`**) and a simple view with a html submit button
2. The button must emulate a dice roll: Each time the user clicks the button a random number between 1 and 6 must be displayed
3. Write and test the program.
4. Identify the objects used, and the methods and properties.



Write your own classes

A Movie Class example

- We want to be able to work with movies. We want to create **`Movie`** objects in our C#-code.
- The class is a common definition (a "**blueprint**") of **`Movie`** objects.
- To create a **`Movie`** object you must declare a **`Movie`** class
- Each movie has data related to it, such as:
  - **title**
  - **price**
  - **imageUrl**

Writing classes (in Models folder): The **Movie** example

```
public class Movie
{
   private string title;
   private deciaml price;
   private string imageUrl;
}
```

- **title**, **price** and **imageUrl** are called fields or member variables
- They are declared as **private** ("best practice")

Creating objects

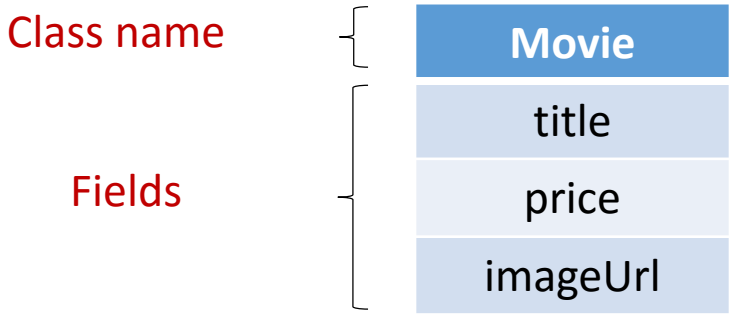
- We can create a new object applying the new keyword:

```
Movie m1 = new Movie();
Movie m2 = new Movie();
Movie m3 = new Movie();
```

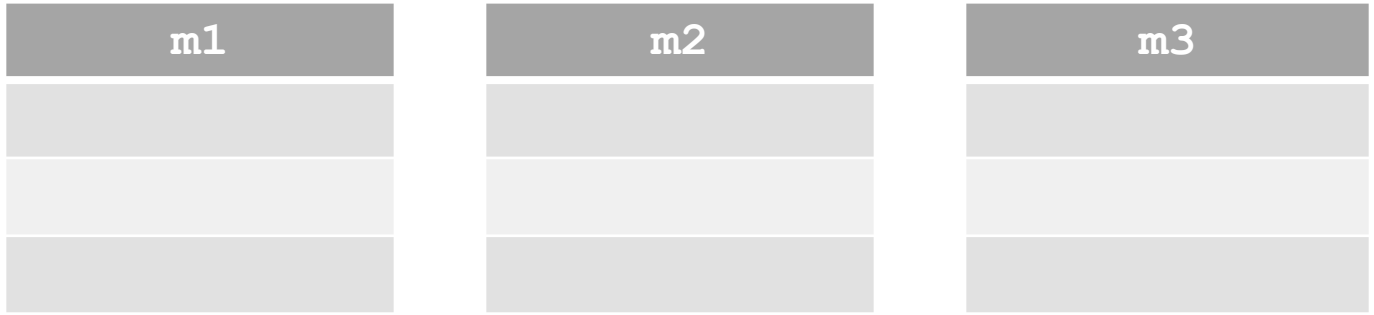
Illustration

```
Movie m1 = new Movie();
Movie m2 = new Movie();
Movie m3 = new Movie();
```

We can illustrate the **`Movie`** class as:



From this class we did create objects with instance names:



Problem

- How do we assign values (data) to the fields?
- Answer: Use a property!

Properties

We want to be able to write:

```
m1.Title = "Schindler's list";
m2.Price = 54.50;
m3.ImageUrl = "schindlers-lst.png";
```

And read values, like:

```
string url = m1.ImageUrl;
```

We cannot write `m1.title`, `m1.price`, and `m1.imageUrl` because the fields are declared private and can only be accessed form inside the class itself.

Movie

- We define a property for the **`title`** field like:

```
private string title;

public string Title
{
  get { return title; } // get accessor method
  set { title = value; } // set accessor method
}
```

- and the same for **`price`** and **`imageUrl`**

Properties opens for access to member variables (fields)

```
Movie m1 = new Movie();
Movie m2 = new Movie();
Movie m3 = new Movie();
```

- We are now able to assign values to the fields by referencing the properties:

```
m1.Title = "Schindler's list";
m1.Price = 54.50;
m1.ImageUrl = "schindlers-lst.png";
```

```
m2.Title = "Godfather";
m2.Price = 120.00;
m2.ImageUrl = "godfather.png";
```

```
m3.Title = "The Matrix";
m3.Price = 49.50;
m3.ImageUrl = "matrix.png";
```

Illustration

We still have the **`Movie`** class:

Movie
title
price
imageUrl

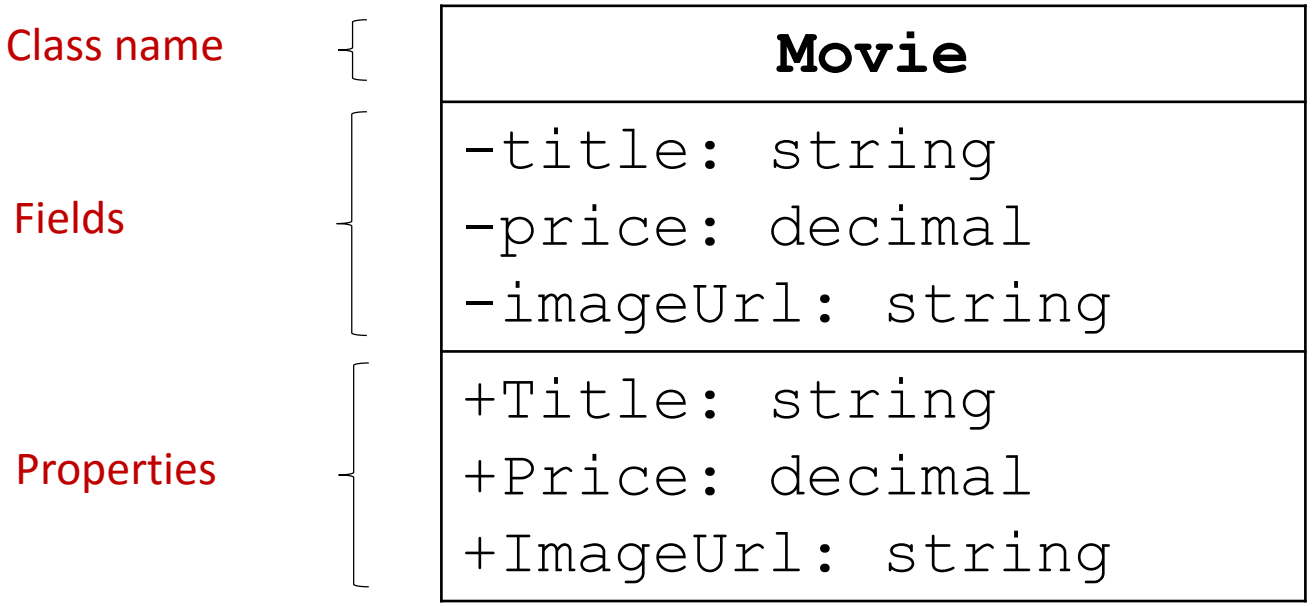
but now we have assigned values to instances of **`Movie`** class (objects) by using its properties:

m1
"Schindler's list"
54,50
"schindlers-lst.png”

m2
"Godfather"
120,00
"godfather.png"

m3
"The Matrix"
49,50
"matrix.png"

UML class diagram



Processing a property value

```
public deciaml Price
{
   set {
        if (value <= 0) // condition for accepting the value
        {
               throw new Exception("Price is not accepted");
        }
        else
        {
               price = value;
        }
   }
   get { return price; }
}
```

Declaring a property as read-only

- We only give the property a get accessor method:

```
private string discount;

public string Discount
{
  get { return discount; }
}
```

Declaring a property as write-only

- We only give the property a set accessor method:

```
private string badWriteOnlyName;

public string BadWriteOnlyName {
  set {badWriteOnlyName = value; }
}
```

CA1044: Properties should not be write-only

Get accessors provide read access to a property and set accessors provide write access. Although it is acceptable and often necessary to have a read-only property, the design guidelines prohibit the use of write-only properties. This is because letting a user set a value and then preventing the user from viewing the value does not provide any security. Also, without read access, the state of shared objects cannot be viewed, which limits their usefulness.

https://msdn.microsoft.com/en-us/library/ms182165.aspx

Reasons why properties are preferred over public fields

- You can have read-only properties, but you can't have these characteristics with a field.
- Since properties are function members, as opposed to data members, they allow you to process the input and output, which you can't do with public fields.

- Gives you fine-grained control over how the outside code (other classes) is allowed to read from and write to the fields

You can change the implementation independently from the public API

Constructors

Problem

- Classes should always be in a valid state which means all the mandatory properties should be set
  - Eg. **`Title`** and **`Price`** for **`Movie`**

- Answer:
  That is possible if we declare a constructor in the class
- A constructor is a method that runs automatically when an object is created

Declaring a constructor

Constructors must have the same name as the class

```
public class Movie
{  ...
   public Movie t,  decimal p)
   {
      title = t;
      price = p;
   }
}
```

Calling the constructor

```
Movie m1 = new Movie("Schildlers list",
54.50m);
```

If the constructor in the above example is the only constructor this will fail because no zero-argument constructor exists in the Movie class:

```
Movie m1 = new Movie();
```

Constructor

- It is common to use same variable name for field and parameter:

```
public class Movie
{
   private string title;
   private string price;

public Movie(string title, decimal price) {

         this.title = title;
          this.price = price;
   }
}
```

Constructor overloading

- As with ordinary methods constructors can be overloaded with multiple versions each with a different set of parameters:

```
public class Movie
{  ...
    public Movie(string title, deciaml price,
                 string imageUrl) {
       this.title = title;
       this.price = price;
       this.imageUrl = imageUrl;
    }
}
```

Default constructor

- A constructor that takes no parameters is called a default constructor.
- Default constructors are invoked whenever an object is instantiated by using the new operator and no arguments are provided to new.
- If no constructors are declared a default constructor is called whenever an object is instantiated.

Object initialization (since C# 3.0)

```
Movie m1 = new Movie {
  Title = "Schindler's list",
  Price = 54.50
};
```

For this to work, there must be:

- only the default constructor (no constructor declarations) or
- a zero-argument constructor.

Automatic Properties

Automatic properties or "Auto-implemented properties"

Allow you to declare properties without declaring backing fields

```
public string ImageUrl
{
    set; get;
}
```

- Auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors.
- The compiler creates a hidden backing field (anonymous & private) that can only be accessed through the property's get and set accessors

Documentation: http://msdn.microsoft.com/en-us/library/bb384054.aspx

Automatic properties: Example

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-Impl Properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerID { get; set; }

    // Constructor
    public Customer(double purchases, string name, int ID)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerID = ID;
    }
    // Methods
    public string GetContactInfo() {return "ContactInfo";}
    public string GetTransactionHistory() {return "History";}

    // .. Additional methods, events, etc.
}
```

Methods

Declaring method **NettoPrice**

```
public class Movie
{  ...
   public decimal NettoPrice()
   {
      return price * 0.8;
   }
```

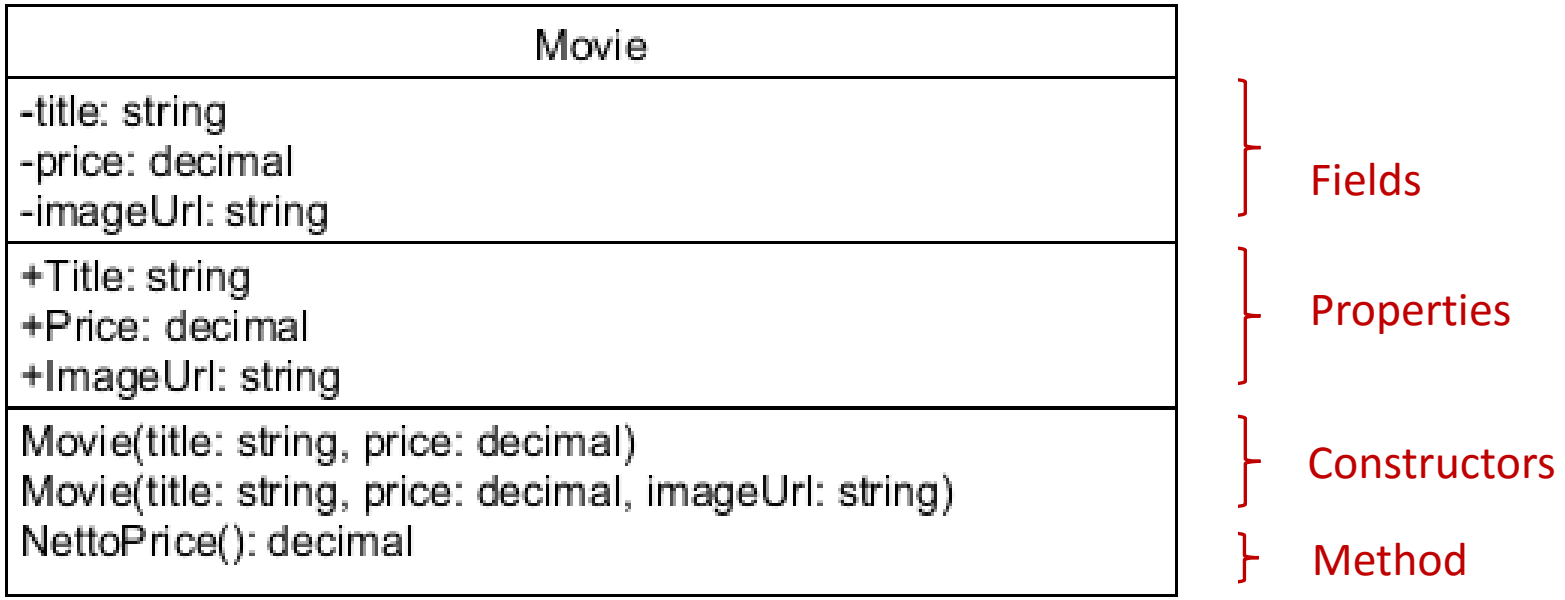
return type

Calling **NettoPrice**

```
decimal nettoPrice = knife.NettoPrice();
```

Refined UML class diagram

- Our UML class diagram has been expanded with properties, a constructor, and a method
- In the diagram below the type of the properties and the return value type of the method is specified. Note the type is specified *after* the name:



Exercise 1

Some of the exercises are part of the first mandatory exercise

Summary of classes

1. Declaration

- The class is declared like:

```
public class Movie {
   ...
}
```

2. Add fields

- Add member variables (or fields) to hold data
- Specify the visibility as private, as we don't want to give other code direct access

```
public class Movie {
  private string title;
  private deciaml price;
  private string imageUrl;
}
```

3. Add properties

- For each field add a get and/or set property
- For example for the price field:

```
public deciaml Price
{
   get { return price; }
   set { price = value; }
}
```

3. Add properties

- For the name field we probably don't want a set property as the name is suppose to be unique and it does not change (read-only):

```
public string Title
{
   get { return title; }
}
```

3. Consider use of auto-implemented properties (or automatic properties)

- Easy to write and read if you do need the bodies of the accessor

```
public string ImageUrl { get; set; }
```

4. Add methods

- A method is doing some "calculation" and may return a result
- In the Movie case we have a method **`NettoPrice()`** which subtracts VAT from **`Price`** and returns it as a deciaml:

```
public class Movie
{  ...
   public decimal NettoPrice ()
   {
      return price * 0.8;
   }
}
```

5. Declare one more constructor

```
public class Movie {
   public Movie(string title,
       deciaml price, string imageUrl) {

      this.title = title;
      this.price = price;
      this.imageUrl = imageUrl;
   }
}
```

Use the constructor

- A constructor is used to set values for (some of) the fields when an object is created:

```
m1 = new Movie(
        "Schindler's List",
        54.50,
        "schindlers-lst.png"
    );
```

You can have more than one constructor as long as the signatures are different

```
public class Movie {
   ...
  public Movie(string title, deciaml price)
  {
    this.title = title;
    this.price = price;
  }
}
```

```
Movie m1 = new Movie("Schindler's List", 54.50,
"schindlers-lst.png");

Movie m2 = new Movie("Godfather", 120.00);
```

Tips & Hints

Special characters which converts literal numeric values

Specialtegn	Datatype
M	decimal
D	double
F	float
L	long

Is interpreted as a double

```
decimal decimalTal = Convert.ToDecimal(0.2);
decimalTal = 0.2M; // interpreted as a decimal
```

List

- If a member variable has more than one value you need to use an array or a collection type like List
- Often it is better (easier) to use **`List`** instead of array types:
  - The size of a list is dynamic and expands as more elements are added to the list
  - The **`List`** class is located in the **`System.Collections.Generic`** namespace

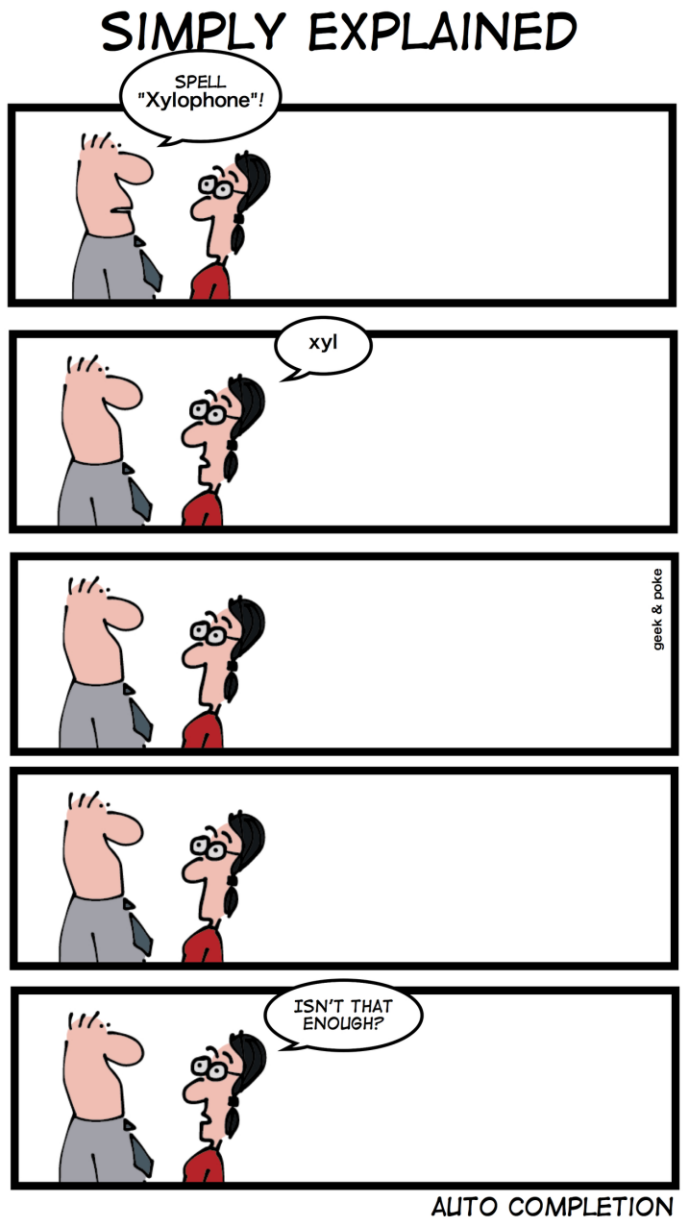
Create a List

```
using System.Collections.Generic;
...
// instantiates a new List object
List<string> myFriends = new List<string>();

// Use the method Add to add new elements to the list
myFriends.Add("Reno");
myFriends.Add("Lisa");
myFriends.Add("Michael");
myFriends.Add("Susan");
```

Run through a list

```
string s = "";

foreach (string friend in myFriends)
{
      @friend + "<br />";
}
```

IntelliSense is great!
- you should always use meaningful variable names



Exercise 2-3

Next week: OOP 2:2

- Object-oriented programming in C#: A Concise Introduction, pp. 28-62
- C# From Scratch (Pluralsight, Jesse Liberty)
  - Object Oriented Programming
  - Arrays and Colllections