# Transforming Python to C# – demo explained

## Contents

# Classes

The demo retakes one of the (slightly modified) Python exercises of the Device Programming 1 course, namely the Beer exercise (course week 10). All the transformations that were applied are descripted below.

```python
class Beer:
```

```
public class Beer
{
```

- The **public** keyword is an access modifier with the most permissive access level. There are no restrictions on accessing public classes.

## Properties

### Simple (default) property – prop

```python
@property
def name(self):
    return self.__name

@name.setter
def name(self, newname):
    self.__name = newname
```

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

- A property always has the **public** access specifier
- The Name property provides read (**get**) and write (**set**) possibilities
- It was easily created using the shortcut **prop** (type the shortcut followed by tab + tab)
- The property is of type **string**, which means it *must* contain text
- We use **CamelCasing**: every property name starts with a capital, and so does each other new word in the variable name. (Eg.: **F**irst**N**ame).

### Simple (default) property – propfull

```python
@property
def name(self):
    return self.__name

@name.setter
def name(self, newname):
    self.__name = newname
```

```
private string _name;

public string Name
{
    get { return _name; }
    set { _name = value; }
}
```

- This is **exactly the same property** as the one above that was created with 'prop', but fully written out
  - ✓ It was created using the **propfull** shortcut
  - ✓ In this manner, you see the actual **field** (_name) that is connected to the property. A similar field is also present in the shorter 'prop', but you cannot see it since it's automatically created behind the scenes.
  - ✓ You can always **choose which notation you prefer**, both are correct. In general, you would be advised to use 'prop' for default properties that only need a type, and 'propfull' for properties that need extra logic in get or set (see further for an example with 'color').

## Readonly property

```python
@property
def brewery(self):
    return self.__brewery
```

```
public string Brewery { get; private set; }
```

- By adding the '**private**' keyword before set, the property becomes readonly the outside the class
- You can do the same thing in a full property, just add private before the set keyword

## Extra logic in get or set

### *Check for types*

```python
@alcoholperc.setter
def alcoholperc(self, newalcoholperc):
    if (isinstance(newalcoholperc, float)):
        self.__alcoholperc = newalcoholperc
    else:
        self.__alcoholperc = -1
```

```
public double Alcohol { get; set; }
```

- The Python example checks if the value of the new alcohol percentage that is provided for the property is of type float. As you can see, checking the type is **not** necessary in C#, since it is automatically enforced by the **type of the property**.
- Providing any other value to this property in C# would result in an error.

### *Check for values*

```python
@property
def color(self):
    return self.__color

@color.setter
def color(self, newcolor):
    if (newcolor != ""):
        self.__color = newcolor
    else:
        self.__color = "unknown"
```

```
private string _color;

public string Color
{
    get { return _color; }
    set
    {
        if (value != "")
        {
            this._color = value;
        }
        else
        {
            this._color = "unknown";
        }
    }
}
```

- While in Python you can choose the variable name of the new value (eg.: **newcolor**), in C# this is always accessed by the default keyword **value**.
- The keyword **self** in Python is similar to **this** in C#

# Constructors

```python
def __init__(self, name, brewery, alcoholperc, color):
    self.name = name
    self.brewery = brewery
    self.alcoholperc = alcoholperc
    self.color = color
```

```csharp
public Beer(string name, string brewery, double alcoholPerc, string color)
{
    this.Name = name;
    this.Color = color;
    this.Alcohol = alcoholPerc;
    this.Brewery = brewery;

}
```

- An empty constructor can easily be generated automatically by using the shortcut **ctor** (followed by 2x the tab key). You can then add the required parameters to it.
- The keyword 'self' (or 'this' in C#) is not present in a C# constructor
- All parameters must have a **type** (preferably the same type as the property they will be applied to).

# Methods

## Object to string

Determine the translation of the object to a String.

```python
def __str__(self):
    return self.name + " (" + self.brewery + ") "
```

```csharp
public override string ToString()
{
    return Name + " (" + Brewery + ")";
}
```

- Every object has a default ToString function in C#, determining how an object is displayed as a string (eg. when writing to a debug window, or showing it in a list). You can **override** the default behavior.
- 'Name' could also be written as 'this.Name' ; both would be correct.

## Return a list of objects

For now, we will not read from a csv file. Instead, we return a fixed array of Beer objects.

```python
@staticmethod
def get_beers():
    beers = []
    beers.append(Beer("Rodenbach Grand Cru", "Rodenbach", 6, "brown"))
    beers.append(Beer("Jupiler", "InBev", 5.2, "blond"))
    beers.append(Beer("Omer","Bockor", 8, "blond"))
    beers.append(Beer("Duvel", "Duvel Moortgat", 8.5, ""))
    return beers;
```

```python
public static List<Beer> GetBeers()
{
    return new List<Beer>
    {
        new Beer("Rodenbach Grand Cru", "Rodenbach", 6, "brown"),
        new Beer("Jupiler", "InBev", 5.2, "blond"),
        new Beer("Omer","Bockor", 8, "blond"),
        new Beer("Duvel", "Duvel Moortgat", 8.5, "")
    };
}
```

- We use the datatype **List<T>**, where **T** is the type of objects the list contains (so in case of the example, only Beer objects can be added to the list).
- The @staticmethod keyword in Python is replaced by the **static** keyword in the function declaration.
- We must declare the **return type** in the function:
    - ✓ A function of type **void** does not return anything, it only executes something.
    - A function of type **T** (eg. in this case: List<Beer>) *must* return an object of this type (or null).
- The **new** keyword creates in instance of the class, using (one of the) constructor(s).
- We make use of the **Add** function to add something to a List<T>, instead of append in Python.

## Search in a list of objects

```python
@staticmethod
def search_beers_by_name(beers, searchterm):
    results = []
    for beer in beers:
        if (beer.name.lower().find(searchterm.lower()) >= 0):
            results.append(beer)
    return results

@staticmethod
def search_beers_by_alcoholperc(beers, min_percentage, max_percentage):
    results = []
    for beer in beers:
        if (beer.alcoholperc >= min_percentage) and (beer.alcoholperc <= max_percentage):
            results.append(beer)
    return results
```

```csharp
public static List<Beer> SearchByAlcohol(List<Beer> beers, double min, double max)
{
    List<Beer> results = new List<Beer>();
    foreach(Beer beer in beers)
    {
        if (beer.Alcohol >= min && beer.Alcohol <= max)
        {
            results.Add(beer);
        }
    }
    return results;
}

public static List<Beer> SearchByName(List<Beer> beers, string searchTerm)
{
    List<Beer> results = new List<Beer>();
    foreach (Beer beer in beers)
    {
        if (beer.Name.ToLower().Contains(searchTerm.ToLower()))
        {
            results.Add(beer);
        }
    }
    return results;
}
```

- We use the datatype **List<T>**, where **T** is the type of objects the list contains (so in case of the example, only Beer objects can be added to the list).
- All parameters must have a **type**.
- We must declare the **return type** in the function:
    - ✓ A function of type **void** does not return anything, it only executes something.
- A function of type **T** (eg. in this case: List<Beer>) *must* return an object of this type (or null).
- The lower() function is replaced by **ToLower()** in C#. Also, the find function is replaced by **Contains**, but you could also make use of **IndexOf**, which shows similar more behavior as Pythons find function (returning an index).
- We make use of the **Add** function to add something to a List<T>, instead of append in Python.

# Executable program

After creating our class, we will test it by creating objects and display them. In Xamarin, for now, we will always do this in the **MainPage.xaml.cs** file.

## Namespaces

To provide access to the class created, we must tell him what to import / use.

```
from model.Bier import Bier
from model.ScoresStudent import ScoresStudent
```

```
using LAB01_DEMO.Model;
```

- You can find the **namespace** of a class by looking at the class file.
  - ✓ Would the namespace be the same as the one your MainPage.xaml.cs file is in, then a using statement is not necessary.
- All classes in the namespace are automatically imported, while in Python, you have to add all of them separately.

## Create/display a single object

```
def create_beer():
    omer = Beer("Omer","Bockor", 8, "blond")
    print(omer);
```

```
private void CreateBeer()
{
    Beer omer = new Beer("Omer", "Bockor", 8, "blond");
    Debug.WriteLine(omer);
}
```

- The **new** keyword creates a new instance of the object.
- Every variable must have a type, where the most specific type is preferred. However, you can make use of the keyword 'var'.
- To write to the Debug window (output window that is by default in the lower part of your vs.net environment), you have to add a **using** statement to the **System.Diagnostics** namespace.
- By printing the entire Beer object, we automatically call the **ToString()** function.

## Get/display a list of objects

```
def load_all_beers():
    list_beers = Beer.get_beers()
    print_beers(list_beers)
```

```
private void LoadAllBeers()
{
    List<Beer> allBeers = Beer.GetBeers();
    PrintBeers(allBeers);
}
```

- The static function is called in the same manner as it would be in Python
- The results are passed via the function's arguments

```python
def print_beers(list_beers):
    for beer in list_beers:
        print("%s -> alcoholpercentage: %s" % (beer, beer.alcoholpercentage))
```

```
public void PrintBeers(List<Beer> beerList)
{
    foreach (Beer beer in beerList)
    {
        Debug.WriteLine("* {0} -> alcohol percentage: {1}", beer, beer.Alcohol);
    }
}
```

- The 'for … in … :' loop in Python is replaced by a **foreach** in C#. similar to Python, a variable named 'beer' is created to loop through all the values in the array.
  https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in
- The **string concatenation** is also similar to the one in Python:
  - ✓ Python: "%s is replaced, as well as %s." % ("this", "that")
  - ✓ C#: "{1} is replaced, as well as {0}", "that", "this"
  - ✓ Both the statements above result in "this is replaced, as well as that"
- Since an entire beer object is passed as the first value, automatically ToString will be called.

# Xamarin specific – use layout to display data

We will display the details of the single object we created, as well as the hard coded list of beers.

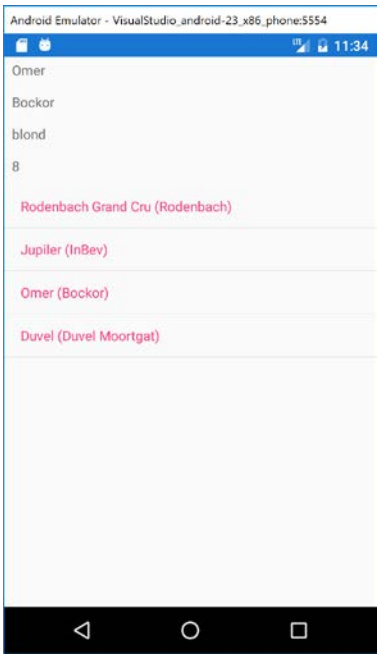
```
<StackLayout>

    <Label x:Name="lblName" Margin="8,4" Text="(name)" />
    <Label x:Name="lblBrewery" Margin="8,4" />
    <Label x:Name="lblColor" Margin="8,4" />
    <Label x:Name="lblAlcohol" Margin="8,4" />

    <ListView x:Name="lstBeers" />

</StackLayout>
```

- **MainWindow.xaml** contains 4 Label controls in a stacklayout.
  - ✓ The **StackLayout** container by default displays all controls underneath one another.
    Note: you can change the direction from up/down to left/right by changing the *orientation* property to *horizontal*
  - ✓ The **ListView** control is used to display a list of objects, represented as a list of strings.
  - ✓ The **Label** control is used to display a single string.
  - ✓ Each control gets a name using **x:Name**, to make it accessible from the code behind.
    - The **Text** property of the label is not required, but if it contains a value, this is the default text that will be displayed. *Should you start this xaml layout without code, only the first label would show text.*
    - The **Margin** property sets the margin of the Label to 8 on the left/right, and 4 above/below the control.

```
private void ShowDetails(Beer beer)
{
    lblName.Text = beer.Name;
    lblBrewery.Text = beer.Brewery;
    lblColor.Text = beer.Color;
    lblAlcohol.Text = beer.Alcohol.ToString();
}
```

```
List<Beer> allBeers = Beer.GetBeers();
lstBeers.ItemsSource = allBeers;
```

- **MainWindow.xaml.cs** accesses the:
  - ✓ **Label** controls by their **name** to change their **Text** property to the values of the associated beer object properties.
  - ✓ **ListView** control by its **name** to change the **ItemsSource** property. This property contains the list of items it contains, which can be any type of object. Even though it stores the **full** objects, by default, it will only display the **ToString** value of the objects.




This is a screenshot of the demo running in an Android emulator. The details of the visualization (such as colors) will vary among the different operation systems and are based on the settings of the device!