IT2154 Practical 3: OOP and Polymorphism

Learning Outcomes

By the end of this lesson, you should be able to

- Develop simple OOP application using C#
- Develop simple MVC web application using DotNet

Exercise 1

In lecture we have seen a C# application using OOP to capture characters in an RPG program.

The complete code of the RPG game program is as follows,

```
using System;
using System.Collections.Generic;

namespace RPGAttempt6

{

    class Character {
        private string name;
        private int hp;
        public Character(string name, int hp)
        {
            this.name = name;
            this.hp = hp;
        }
        public string Name
        {
            get { return this.name; }
            set { this.name = value; }
        }
        public int Hp
        {
            get { return this.hp; }
            set { this.hp = value; }
        }
        public virtual string Battle() { return ""; }
    }

    interface Healer {
        public void Heal(Character target);
    }
```

```
    class Knight : Character, Healer
    {
        public Knight(string name, int hp) : base(name, hp) { }
        public override string Battle() { return "For glory!"; }

        public void Heal(Character target) {
            if (target.Hp != 0) {
                target.Hp += 10;
            }
        }
    }

    class Ogre : Character
    {
        public Ogre(string name, int hp) : base(name, hp) { }
        public override string Battle() { return "Roooaaarrr!"; }

    }

    class Program
    {
        static void Main(string[] args)
        {
            var knight = new Knight("Arthur", 100);
            var ogre_1 = new Ogre("Growl", 200);
            var ogre_2 = new Ogre("Dumb", 200);
            List<Character> chars = new List<Character>();
            chars.Add(knight);
            chars.Add(ogre_1);
            chars.Add(ogre_2);
            foreach (var c in chars) {
                Console.WriteLine(c.Battle());
            }
        }
    }
}
```

The code is also available in a project folder `RPGAttempt6` given in the student resource file. Compile and run the code and observe the output.

Exercise 2

You are tasked to develop a chat module for an online ordering system. a). Create a folder Message b). Open using Visual Studio Code (VSC) c). In terminal of VSC, run dotnet new console.

You are considering the Message model. There are two types of messages, normal `Message` and `ImageMessage`.

- A `Message` contains `Id` (string) , `CreatedAt` (DateTime) and `Text` (string). It has a `Display` operation, which display its `Text` and `CreatedAt`, separated by a character "|".

- An `ImageMessage` contains all the attributes that a `Message` has. In addition, an `ImageMessage` contains a `ImageURL` (string) field. The `Display` operation for an `ImageMessage` should display `Text`, `CreatedAt` and `ImageURL`, separated by a character "|".

Using the Exercise 1 as a reference, develop a C# OOP program to implement the above mentioned message model.

You may use the following for a simple testing.

```
using System;
using System.Collections.Generic;
namespace Message
{
    class Program
    {
        static void Main(string[] args)
        {
            Message m = new Message("1", "Hello");
            ImageMessage im = new ImageMessage("2", "Hello World",

"https://miro.medium.com/max/1400/1*OohqW5DGh9CQS4hLY5FXzA.png");
            List<Message> messages = new List<Message>();
            messages.Add(m);
            messages.Add(im);
            foreach (var message in messages) {
                Console.WriteLine(message.Display());
            }
        }
    }
}
```

Expected output

```
$ dotnet run
Hello|07/05/2020 11:02:19
https://miro.medium.com/max/1400/1*OohqW5DGh9CQS4hLY5FXzA.png|Hello
World|07/05/2020 11:02:19
```

Exercise 3

In this exercise, we are using C# .NET MVC Core to develop a web application.

1. Create folder `AgeGuru`,
2. Open the folder using Visual Studio Code
3. In the terminal of VSC, run

```
$ dotnet new mvc
```

which will populate the `AgeGuru` folder with the following files and sub folders.

```
AgeGuru
|-Controllers
|  |- HomeControllers.cs
|-Models
|  |- ErrorViewModel.cs
|-Properties
|  |- launchSettings.json
|-Views
|  |- Home
|     |- Index.cshtml
|     |- Privacy.cshtml
|  |- Shared
|     |- _Layouet.cshtml
|     |- _ValidationScriptPartial.cshtml
|     |- Error.cshtml
|  |- _ViewImports.cshtml
|  |- _ViewStart.cshtml
|-wwwroot
|  |- css
|     |- site.css
|  |- js
|     |- site.js
|  |- lib
|     |- bootstrap
|     |- jquery
|     |- jquery-validation
|     |- jquery-validation-unobstrusive
|  |- favicon.ico
|-appsettings.Development.json
|-appsettings.json
|-AgeGuru.csproject
|-Program.cs
```

4. In the terminal of VSC, run

```
$ dotnet build
```

then

```
$ dotnet run
```

5. Open the link http://localhost:<port_number> from the terminal in your browser, you should be able to see a template website. What we are using is the MVC .NET Core framework to quick start a web site project. MVC stands for Model View Controller design pattern for web development. **Model** corresponds to the component that interact with the databases. **View** coresponds to the User Inferface and Layout. **Controller** takes care of the user browser HTTP requests, invokes the right models to generate data for display, and uses the view to render the result to the UI. In this practical, we will build a simple controller.

6. Press Ctrl-C to stop the web app from running.

7. Go back to the Visual Studio Code. In the Controller folder, add a new file with name `AgeController.cs`. Edit the file with the following content.

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace AgeGuru.Controllers
{
    public class AgeController : Controller
    {
        //
        // GET: /Age/

        public string Index()
        {
            return "Enter http://localhost:<port_number>/age/compute/?name=<name>&
            dob= <dob>"  + " to find out your age.";

        }

        //
        // GET: /Age/Compute/

        public string Compute(string name, string dob)
        {
            int age = 0;
            var dateOfBirth = DateTime.Parse(dob);
            var now = DateTime.Now;
            var span = now.Subtract(dateOfBirth);
            age = Convert.ToInt32(span.TotalDays / 365);
            return $"Dear {name}, you are {age} years old this year.";
        }
    }
}
```

8. Save the file and compile the project

```
$ dotnet build
```

9. Restart the web app.

```
$ dotnet run
```

10. Visit `http://localhost:<port_number>/age/`, what do you see? visit `http://localhost:<port_number>/age/index/`, what do you see? Visit `http://localhost:<port_number>/age/compute/?name=kenny&dob=2000-01-01`, what do you see?

11. In the last few steps, we build a new controller called `AgeController`. The dotnet core MVC automatically maps the urls `http://localhost:<port_number>/age/index/` to the `AgeController` method `Index()`, and `http://localhost:<port_number>/age/compute/` to the method `Compute()`. Note that the `Compute()` method expects two arguments. They are passed in as the URL query string, i.e. `?name=kenny&dob=2000-01-01`.

12. In summary, we developed a simple web application using DotNet MVC Core, with a standalone controller.

Exercise 4

In Exercise 2, we designed and develop a two-class model for messages. In this exercise, we are going to further develop it by building an MVC core web app around it.

1. Create folder `ChatApp`
2. Open the folder using Visual Studio Code
3. In the terminal of VSC, run

```
$ dotnet new mvc
```

which will create the folder structure for this web app.

4. In the terminal of VSC, run

```
$ dotnet build
$ dotnet run
```

Make sure the web app is running fine by checking http://localhost:<port_number> with your browser

5. Press Ctrl-C to stop the web app from running.

6. Before we start writing down our codes, let's think through what we are planning to do.

7. Go back to Visual Studio Code. In the Model folder, add a new file with name `Message.cs`. Edit the file and include the following content.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ChatApp
{
    public class Message
    {
        public int Id { get; set; }
        public string? Text { get; set; }

        [DataType(DataType.Date)]
        public DateTime CreatedAt { get; set; }
    }
}
```

Similar to the solution to Exercise 2. the `Message` class contains an `Id` field, which is required by the database for the primary key.

The `DataType` attribute on `CreatedAt` specifies the type of the data (Date). With this attribute: The user is not required to enter time information in the date field. Only the date is displayed, not time information. For the time being, we ignore the sub-class `ImageMessage`.

8. We need to add the needed libraries for us to sync the model with the database. In order to do that we need more libraries. We can install the needed libraries with some `dotnet` commands. Go to VSC terminal and make sure we are in the root folder of our project, that is, `ChatApp`. To install the tools based on your current .NET version (e.g. 7.0.xxx), add **`--version="7.0"`** behind each command bellow. Run:

```
$ dotnet tool uninstall --global dotnet-aspnet-codegenerator
$ dotnet tool install --global dotnet-aspnet-codegenerator
$ dotnet tool uninstall --global dotnet-ef
$ dotnet tool install --global dotnet-ef
$ dotnet add package Microsoft.EntityFrameworkCore.Design
$ dotnet add package Microsoft.EntityFrameworkCore.SQLite
$ dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
$ dotnet add package Microsoft.EntityFrameworkCore.SqlServer
$ dotnet add package Microsoft.EntityFrameworkCore.Tools
```

The commands above added the following

(a) The aspnet-codegenerator scaffolding tool. (b) The Entity Framework Core Tools for the .NET Core CLI. (c) The EF Core SQLite provider, which installs the EF Core package as a dependency. (d) Packages needed for scaffolding: `Microsoft.VisualStudio.Web.CodeGeneration.Design` and `Microsoft.EntityFrameworkCore.SqlServer`.

9. Next, in order to access the model class as database tables, we need to add a database context class. In the project root folder, create a new folder named `Data`. In this newly created folder, add a file named, `ChatAppContext.cs` with the following content.

```
using Microsoft.EntityFrameworkCore;
using ChatApp.Models;
```

```
namespace ChatApp.Data
{
    public class ChatAppContext : DbContext
    {
        public ChatAppContext (DbContextOptions<ChatAppContext> options)
            : base(options)
        {
        }

        public DbSet<Message> Message { get; set; }
    }
}
```

With this class, we can perform basic CRUD operations on the database without writing SQL queries.

10. Next we need to register the data context with the `Program.cs` module in the web app, so that the data context is loaded when the web server starts up. Add the following to `Program.cs`, after `var builder = WebApplication.CreateBuilder(args);`

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    builder.Services.AddDbContext<ChatAppContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("ChatAppContext")));
}
else
{
    builder.Services.AddDbContext<ChatAppContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("ProductionChatAppContext")));
}
```

The string ChatAppContext is our database connection string.

Make sure Program.cs import the following two libraries.

```
using ChatApp.Data;
using Microsoft.EntityFrameworkCore;
```

11. Edit `appsettings.json` file with the following connection string

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
```

```
  "ConnectionStrings": {
    "ChatAppContext": "Data Source=ChatApp.db"
  }

}
```

12. Now we are ready to scaffold the controller and view codes for the `Message` model. Run the following in the terminal, assuming we are at the root folder of the project.

If you are using a windows machine, ignore the following, otherwise,

```
$ export PATH=$HOME/.dotnet/tools:$PATH # only needed for Mac and Linux
```

run

```
$ dotnet aspnet-codegenerator controller -name MessageController -m Message
   -dc ChatAppContext --relativeFolderPath Controllers --useDefaultLayout
   --referenceScriptLibraries
```

where

- `-name` defines the name of the controller to be generated
- `-m` defines the name of the model class to be referenced
- `-dc` defines the data context
- `--relativeFolderPath` defines the folder path of the Controller
- `--useDefaultLayout` defines the default layout to be used for the views
- `--referenceScriptLibraries` adds _ValidationScriptsPartial to Edit and Create pages

After this command, we should see the new files added in the `controller` folder and the `view/Message` folder.

13. We need to sync up the changes with the database, so that the tables will created/modified automatically. Run the following in the terminal at the root folder of the project.

```
$ dotnet ef migrations add InitialCreate
$ dotnet ef database update
```

- `ef migrations add InitialCreate`: Generates an `Migrations/{timestamp}_InitialCreate.cs` migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `ChatAppContext` class (in the `Data/ChatAppContext.cs` file).

- `ef database update`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the `Migrations/{time-stamp}_InitialCreate.cs` file, which creates the database.

For simplicity, we are using a sqlite file based database, `ChatApp.db`. You can install SQLite extension from VSC to view the ChatApp.db. Press **Ctrl+Shift+P** to launch the Command Pallete, then select SQLite: Open Database and navigate to SQLite Explorer. For real world application, we should probably use MSSQL or MySQL.

14. Build and run the web app. Visit `http://localhost:<port_number>/Message/` to test out the CRUD functions.
15. Note that we have not incoporated the `ImageMessage` into this web app yet. We will look into that in the next practical.

References

- https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-6.0&tabs=visual-studio-code