POLITÉCNICO DE LEIRIA
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

www.dei.estg.ipleiria.pt

Systems Integration

Review of Object-Oriented Programming (OOP) and LINQ

Goals/Topics:

– Experiment with the Visual Studio (VS) IDE

– Create libraries (DLL) and executable applications

– Review and understand the concepts of OOP using C# programming language

– Explore the concepts of namespaces, classes, fields, properties, methods, indexers and constructors using C# language

– Use LINQ queries

Duration: 1 class

PART 1 – OOP REVIEW

Create a C# Visual Studio solution named ***SolutionOOP.***

- The project should be created using the templates available for the C# language.
- Add a *Console Application* project named ***ConsoleAppSheet1*** and a *Class Library* (.Net Framework) project named ***OrganizationLib*** to that solution.
- The classes should be created inside the *Class Library* project;
- The *Console Application* project will be used to test the developed Library.

The interaction between projects is possible by adding a reference between them. Therefore, add a **reference** to the ***OrganizationLib*** *Class Library* in the *ConsoleAppSheet1 project.*

1. Figure 1 shows the hierarchy of classes to be implemented, which represents a software organization. This organization is composed by a group of persons, who can either be customers or employees. There are two types of roles for the employees: project manager and programmer. The project manager contains a team of several programmers under his supervision.

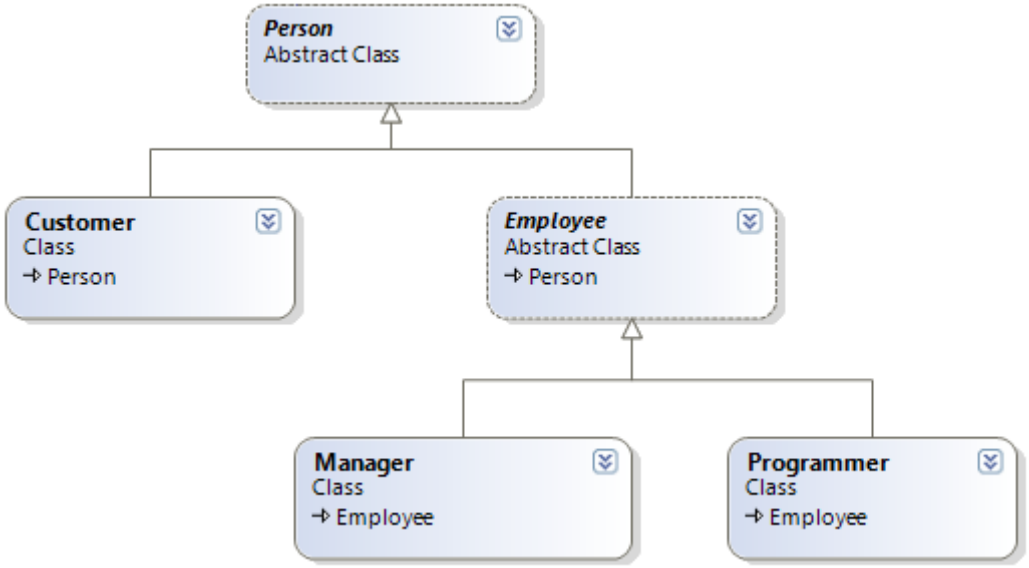


Figure 1 – Class hierarchy

Create all the classes in a namespace called ***OrganizationLib***. Each class should be implemented in its own file. The implementation should be performed according to the following guidelines.

Note: At the end of the worksheet, you can find the complete diagram of the proposed exercise containing all the classes and their behavior.

2. Define the fields, properties, and methods of the ***Person*** class. Implement the *Person* class with the following behavior:

2.1. It is an abstract class.

2.2. The fields *_firstName*, *_lastName* and *_birthDate* represent the first name, last name and the birthdate of a person.

2.3. The constructor receives, as parameters, the first name, the last name and the birthdate of a person.

2.4. The properties ***FullName*** and ***Age*** are two read-only properties. The *FullName* property returns the full name of the person, and the property *Age* gives (returns) a person's age.

2.5. Add a new property named **Birthdate**. It is a read and write properties, therefore you need to implement the *get* and *set*.

2.6. Add an auto-implemented property named **Gender**. Auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. You can also add an **enum** type (named GenderType) that allows the values "*Feminine*", "*Masculine*" or "*Undifined*".

2.7. The ***Print*** method outputs the full name and the age of a person to a *String*. Derived classes should be allowed to reimplement this method (therefore you must use the ***virtual*** modifier when defining this method).

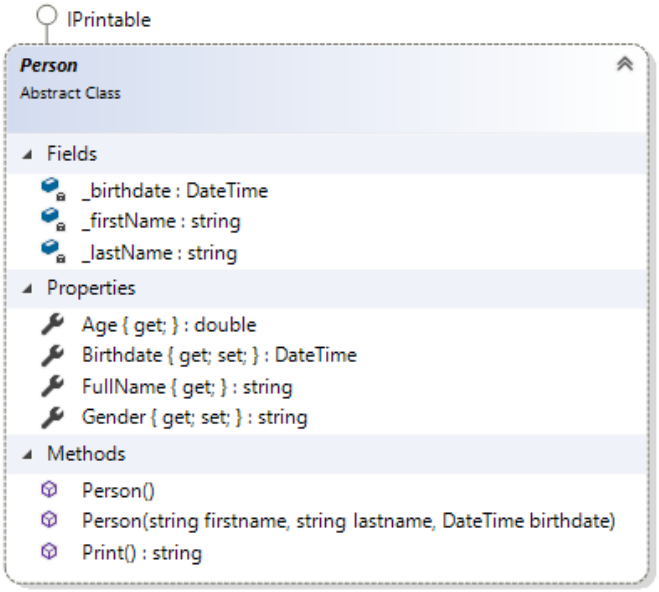


Figure 2 – Person class.

3. Define the private attributes, properties and methods of the ***Customer*** class. Implement the *Customer* class with the following behavior:

3.1. It extends from the Person class.

3.2. The constructor receives, as parameters, the first name, the last name, the birthdate, and the address of a costumer. (Note: this constructor should invoke its base class's constructor).

3.3. The ***Address*** is a property (read and write property) that stores the customer address. (Note: You may implement it as an auto-implemented property).

3.4. The ***Print*** method outputs the full name, the age, and the address of a customer to the console. (Note: it should override the Print method of its base class).

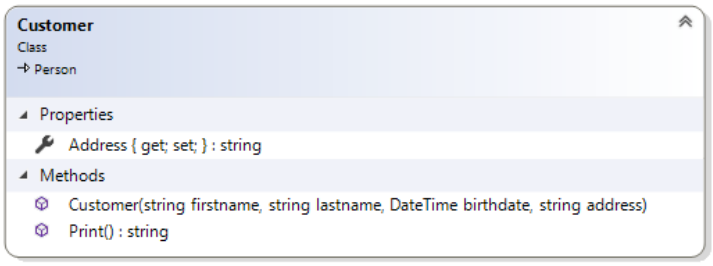


Figure 3 – Customer class.

4. Define the fields, properties and methods of the ***Employee*** class. Implement the *Employee* class with the following behavior:

4.1. It is an **abstract** class and extends the Person class.

4.2. The fields *_salary* stores the employee salary (note: this is optional since you may choose to implement an auto-implemented property to store this data).

4.3. The constructor receives, as parameters, the first name, the last name, the birth date, and salary of the employee. (Note: this constructor should invoke its **base** class's constructor).

4.4. The ***Salary*** property is a read-only property that can be overridden in the derived classes.

4.5. The ***Print*** method outputs the full name, the age, and the salary of the employee to the console. (Note: it should override the Print method of its base class).

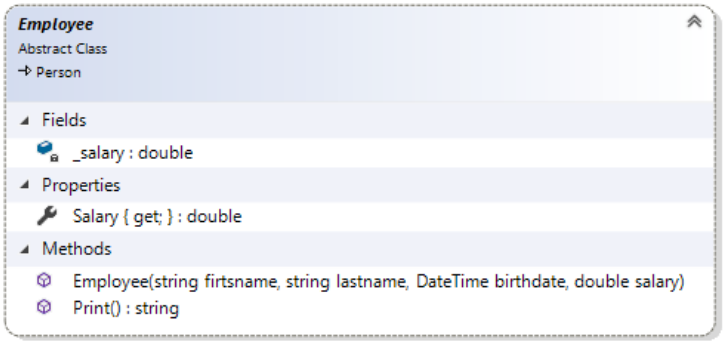


Figure 4 – Employee class.

5. Define the ***Programmer*** class. Implement the *Programmer* class with the following behavior:

5.1. It extends the **Employee** class.

5.2. Auto-generate the constructor that receives, as parameters, the first name, the last name, the birth date, and the salary of the Programmer. (Note: this constructor should invoke its base class's constructor).

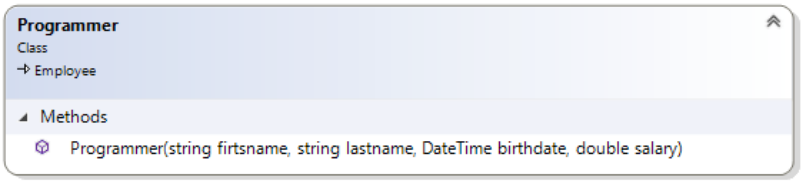


Figure 5 – Programmer class.

6. Define the ***Manager*** class. The ***Manager*** class implementation should have the following behavior:

6.1. It extends the **Employee** class.

6.2. A manager supervises several programmers. That information is stored in the field *_programmers.* This field contains the list of programmers (i.e., list of objects of type *Programmer*) supervised by the manager.

6.3. The Manager class contains an indexer, which allows accessing a programmer in a specific position in the programmers list. Indexers allow your class to be used just like an array. Its implementation is identified by the ***this*** keyword and square brackets, ***this****[int pos]*. It accepts a single position parameter, *pos*. The implementation of an Indexer is the same as a **Property**.

6.4. The read-only property ***NumberOfProgrammers*** returns the number of programmers associated with the manager.

6.5. The ***AddProgrammer*** method allows adding a programmer to the list of programmers belonging to the manager's team.

6.6. The ***Print*** method outputs manager's full name, age and salary to the console. It also outputs the number of programmers who belong to the manager's team, and lists the programmers info. (Note: it should override the Print method of its base class).

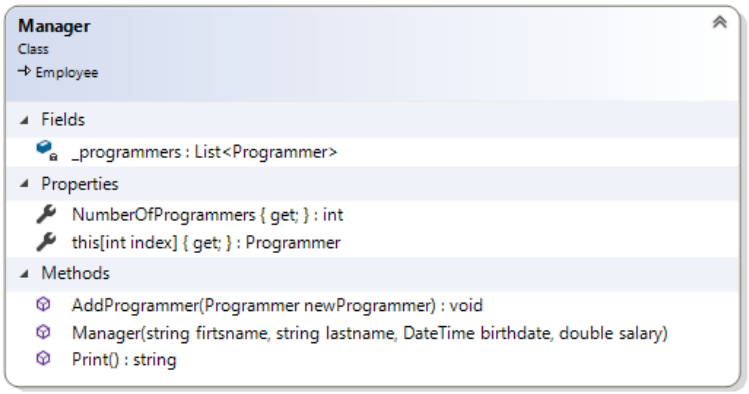
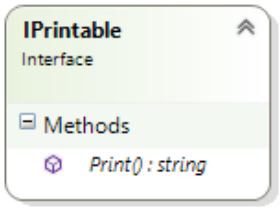


Figure 6 – Manager class.

7. To test the classes implemented so far in the library, use the console application project to create: a list containing 3 customers and 1 manager with a team of 5 programmers. Output the information to the console.

8. Make the necessary changes so that the Person class implements the **IPrintable interface**, as shown in the figure. The interface meets the requirements of the Person class, in which is defined the `Print()` method.



PART 2 - LINQ AND QUERY EXPRESSIONS [EXTRA]

The LINQ standard query operators make possible to query various data sources, such as arrays, collections, even XML and relational data. The operator forms a pattern that operates on a sequence (an object that implements the `IEnumerable<T>` or `IQueryable<T>` interface).

For Example, considering:

```
string[] firstnames = { "Scott", "Steve", "Ken", "Joe",
                        "John", "Alex", "Chuck", "Sarah"};

IEnumerable<string> val = from fn in firstnames
                          where fn.StartsWith("S")
                          select fn;

foreach (string item in val)
{
    Console.WriteLine(item);
}
```

In this example, several operators are utilized from different operation types. The *select* query operator enumerates the source sequence of first names. The *where* query operator is a restriction operator that filters the sequence. In the example, it filters the sequence by limiting the results to only those whose names begin with the letter 'S'.

Instead of `IEnumerable<T>` it can be used an anonymous type (`var`). The `var` keyword is called *Implicit Type Variable*. During runtime it automatically assigns its type of the variable based on the object it finds.

Note: The order of operators is: `from`, `where`, `orderby (thenby)`, `select` (only `from` and `select` are mandatory).

It is also possible to use the **dot notation (lambda expression)**.

Example: select all the names that have more than 6 letters.

```
IEnumerable<string> firstnames = names.Where(s => s.Length < 6);
```

9. Using the above concepts, filter the list of customers to show only the customers whose name starts with the letter's "A". Output the person information to the console.

10. Change the previous filter to show only the customers whose name starts with the letters "A" and live in Leiria. Output the person information to the console.

11. Define a filter that allows you to show how many persons have more than 40 years old. Output the person information to the console.

PART 3 – CREATE TYPES AND PERSONALIZE EXCEPTIONS [EXTRA]

12. Define the **Persons** class. It represents a collection of Person type objects. The aim is to define a new data types which represents itself a collection. Implement the **Persons** class with the following behavior:

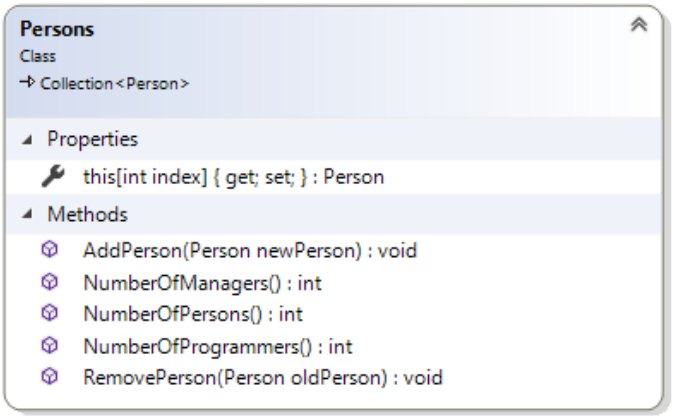


Figure 7 – Persons class

12.1. It represents a collection of Person (Person type).

12.2. It has a read and write default indexer property. It allows the access to the Person in a specific position of the collection. Note: implemented with a *this* property.

12.3. The *AddPerson* method adds one Person to the collection. If that person already exists in the collection, it must throw an exception. The exception message must notify the user that the person already exists in the collection. That exception should be named **ExPersonAlreadyExists**.

12.4. The *RemovePerson* method allows removing a Person from the collection. If the Person does not exist in the collection, it must throw an exception. The exception message must inform that this Person does not exist in the collection. The exception should be named **ExPersonDoNotExists**.

12.5. The *NumberOfProgrammers* method returns the number of programmers inside the collection.

12.6. The *NumberOfManagers* method returns the number of managers inside the collection.

12.7. The *NumberOfPersons method* returns the number of elements (Person) in the collection.

13. To test the *Persons* class, use the console application to create a **Persons** instance containing 10 clients, 2 managers with a team of 5 programmers each. Output the information to the console.

Worksheet final class diagram: