II. Creating Types

Contents

1. Enums

The **enum** keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

Assignment

- Add the following enumeration

```
internal enum OccupationEnum
{
      Child = 0,
      Student,
      Employee
}
```

- In the Main method try to cast from
  - UserAccountTypeEnum. SoftwareDeveloper to System.Int32
  - System.Int32 to UserAccountTypeEnum

2. Structures

A Structure (struct in C#) type is a value type that is typically used to encapsulate small groups of related variables.

Assignment

1. Add the PersonStruct defined bellow

```
internal struct PersonStruct
{
      #region Attributes
```

```
	public int Age;
	public string Name;
	public OccupationEnum Occupation;
	#endregion

	public PersonStruct(int age, string name, OccupationEnum 
occupation)
	{
		Age = age;
		Name = name;
		Occupation = occupation;
	}

	public override string ToString()
	{
		return string.Format("Name: {0}, Age: {1},  Occupation: {2}", 
Name, Age, Occupation);
	}
}
```

2. Add the ValueTypeAssignment method in Program.cs and call it from the Main() method.

```
 private static void ValueTypeAssignment()
{
	Console.WriteLine("###Assigning value types\n");
	var personStruct1 = new PersonStruct(1, "name1", 
OccupationEnum.Student);
	var personStruct2 = personStruct1;

	Console.WriteLine(personStruct1); // automatically calls 
.ToString(). The method is defined in System.Object and overridden in 
PersonStruct
	Console.WriteLine(personStruct2);

	// Change personStruct1.Name and print again. personStruct2.Name is 
not changed.
	personStruct1.Name = "NewName";
	Console.WriteLine(personStruct1);
	Console.WriteLine(personStruct2);
}
```

Questions

- Why is it possible to override the ToString method?

3. Classes

Objectives

- Encapsulation using properties;
- Multiple constructors

Assignment

1. Add the PersonClass class defined bellow.

```
internal class PersonClass
{
	#region Properties
```

```
      #region Age - Without using Properties
      private int _age;
      public int GetAge()
      {
            return _age; // "this._age" is implicit
      }
      public void SetAge(int value)
      {
            _age = value; // "this._age" is implicit
      }
      #endregion

      #region Name - Using properties
      private string _name;
      //Read/Write property
      public string Name
      {
            get { return _name; }
            set { _name = value; }
      }

      //Readonly property
      public string Name2
      {
            get { return _name; }
      }
      #endregion

      #region Occupation - Using auto-property
      public OccupationEnum Occupation { get; set; }
      #endregion
      #endregion

      public PersonClass(int age)
      {
            Console.WriteLine("Constructor(default)");
            _age = age; //equivalent with this._age = age;
      }

      public PersonClass(int age, string name, OccupationEnum
occupation):this(age)
      {
            Console.WriteLine("Constructor(parameters)");
            Name = name;
            Occupation = occupation;
      }

      //Copy constructor - https://msdn.microsoft.com/en-
us/library/ms173116.aspx
      public PersonClass(PersonClass previousPerson) :
this(previousPerson.GetAge(), previousPerson.Name,
previousPerson.Occupation)
      {
            Console.WriteLine("Copy Constructor");
      }

      //Destructor
      ~PersonClass()
      {
            Console.WriteLine("Destructor");
      }

      public override string ToString()
      {
            return string.Format("Name: {0}, Age: {1},  Occupation: {2}",
Name, _age, Occupation);
      }
}
```

2. Add the ReferenceTypeAssignment method in Program.cs and call it from the Main() method.

```
private static void ReferenceTypeAssignment()
{
      Console.WriteLine("Assigning reference types\n");
      var personClass1 = new PersonClass(1, "name1",
OccupationEnum.Student);
      var personClass2 = personClass1;

      Console.WriteLine(personClass1); // automatically calls
.ToString(). The method is defined in System.Object and overridden in
PersonClass
      Console.WriteLine(personClass2);

      // Change personClass1.Name and _age and print again.
personClass2.Name and _age have changed.
      personClass1.Name = "NewUserName";
      personClass1.SetAge(22);
      Console.WriteLine("\n=> Changed personClass1.Name and
personClass1._age\n");
      Console.WriteLine(personClass1);
      Console.WriteLine(personClass2);
}
```

Question

- Can the PersonClass () constructor be made private? (can we have private constructors?)

4. Standard Interfaces

4.1. IComparable<T>

Assignment

1. Create a new project with the name “StandardInterfaces”
2. Add the following “Person” class

```
internal class Person
{
      #region Properties
      public string Name { get; set; }
      public int Age { get; set; }
      #endregion

      public Person(string name, int age)
      {
            Name = name;
            Age = age;
      }
}
```

3. Add the following method in the “Program” class and call it from the “Main()” method (Note: an exception will be thown when you run the project)

```
private static void ReferenceTypeArray()
{
	var p1 = new Person("Name3", 42);
	var p2 = new Person("Name1", 23);
	var p3 = new Person("Name2", 32);

	var pArray = new Person[] { p1, p2, p3 };

	Array.Sort(pArray);

	//IComparable implementation is called automatically by methods such as 
Array..::.Sort

	foreach (var person in pArray)
	{
		Console.WriteLine(person);
	}
}
```

4. Implement the IComparable<Person> interface for the “Person” class.

```
internal class Person : IComparable<Person>
{
	#region Properties
	public string Name { get; set; }
	public int Age { get; set; }
	#endregion

	public Person(string name, int age)
	{
		Name = name;
		Age = age;
	}

	public int CompareTo(Person other)
	{
		//Note: string.CompareTo is culture-specific
		return Name.CompareTo(other.Name);
	}
}
```

5. Change the IComparable<Person> implementation in order to use the Age of the persons

4.2. IClonable

1. Based on the “Person” class, derive the “PersonLuckyNumbers” class.

```
internal class PersonLuckyNumbers : Person
{
	public int[] LuckyNumbers { get; set; }

	public PersonLuckyNumbers(string name, int age, int[] luckyNumbers) : 
base(name, age)
	{
		LuckyNumbers = luckyNumbers;
	}
}
```

2. Add the following method in the “Program” class and call it from the Main method

```
private static void ReferenceTypeClone()
{
        var p1 = new PersonLuckyNumbers("Name 1", 21, new []{13, 26, 39});
        var p2 = p1;

        p1.Age = 12;
        p1.LuckyNumbers[0] = 1;

        Console.WriteLine(p1);
        Console.WriteLine(p2);
}
```

3. Run the application and notice the values in the two objects
4. Implement IClonable interface for the “PersonLuckyNumbers” class as follows (shallow copy only)

```
internal class PersonLuckyNumbers : Person, ICloneable
{
        public int[] LuckyNumbers { get; set; }

        public PersonLuckyNumbers(string name, int age, int[] luckyNumbers) :
base(name, age)
        {
                LuckyNumbers = luckyNumbers;
        }

        public object Clone()
        {
                // First get a shallow copy.
                var newPerson = (PersonLuckyNumbers)MemberwiseClone();
        }
}
```

5. Run the application and notice the values in the two objects
6. Change the implementation of the “Clone()” method in order to perform a **deep copy**

```
public object Clone()
{
        // First get a shallow copy.
        var newPerson = (PersonLuckyNumbers)MemberwiseClone();

        // Then fill in the gaps.
        newPerson.LuckyNumbers = new int[LuckyNumbers.Length];
        for (var i=0; i< LuckyNumbers.Length; i++)
        {
                newPerson.LuckyNumbers[i] = LuckyNumbers[i];
        }

        return newPerson;
}
```

5. Operators

- can be overload by defining static member functions using the operator keyword.
- not all operators can be overloaded and others have restrictions
- further reading: link

Operators	Overloadability

+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using & and |, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
(T)x	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be overloaded, but +=, for example, is evaluated using +, which can be overloaded.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	These operators cannot be overloaded.

Activity

1. For the standard Person class overload the >, < and explicit (int) operators

```
#region Operators
// operator de conversie explicita la int
public static explicit operator int(Person p)
{
	return p.Age;
}

public static bool operator <(Person p1, Person p2)
{
	return p1.Age < p2.Age;
}

public static bool operator >(Person p1, Person p2)
{
	return p1.Age > p2.Age;
}
#endregion
```

2. Use the operators in the Main method

```
private static void Main(string[] args)
{
	var p = new Person("Name1", 21);
	var p2 = new Person("Name2", 22);

	//int age = p; //error
	int age = (int)p;
	Console.WriteLine("Age: {0}", age);

	if(p<p2)
		Console.WriteLine("p.Age < p2.Age");
}
```

3. Implement the implicit (int) cast

6. Class Inheritance

6.1. Abstract classes

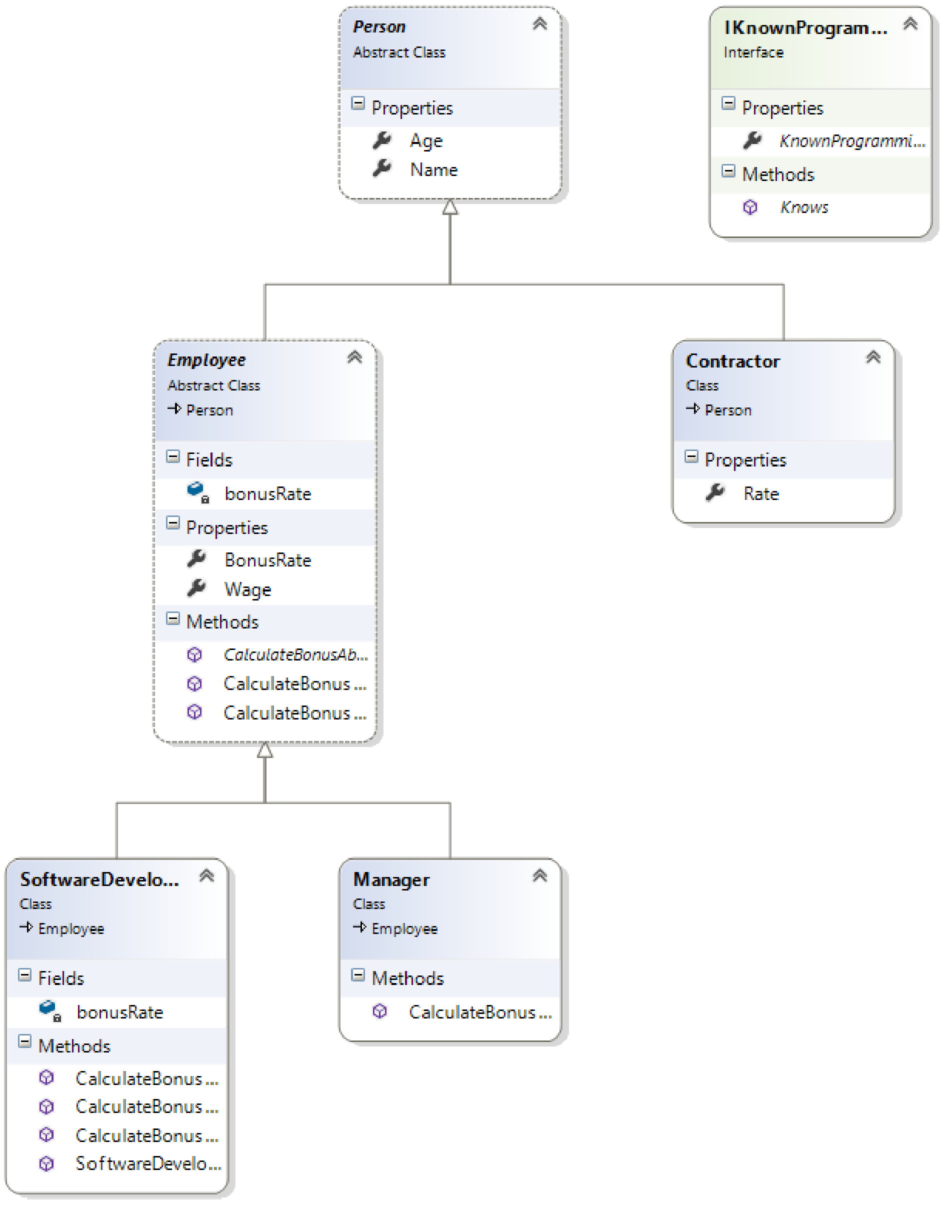

Figure 1 Class hierarchy

Activity

Let's imagine that you are asked to develop an application that handles the wage and bonus calculations for the persons that work in a certain software development company. The categories of persons are: Software Developer, Managers.

1. Create a new project that will include the classes shown in Figure 1.
2. Add an abstract Person class (keep in mind that we only consider the three types of person categories mentioned above)

```
internal abstract class Person
{
      public string FirstName { get; set; }
```

```
	public string LastName { get; set; }
	public int Age { get; set; }
}
```

3. Add an abstract Employee class

```
internal abstract class Employee : Person
{
	// A static point of data.
	private static double bonusRate = 1.1;
	// A static property.
	public static double BonusRate
	{
		get { return bonusRate; }
		set { bonusRate = value; }
	}

	public double Wage { get; set; }


	//Abstract method
	public abstract double CalculateBonusAbstract();

	public double CalculateBonusNormal()
	{
		Console.WriteLine("Employee - CalculateBonusNormal");
		return bonusRate * Wage;
	}

	public virtual double CalculateBonusVirtual()
	{
		Console.WriteLine("Employee - CalculateBonusVirtual");
		return bonusRate * Wage;
	}
}
```

4. Add a SoftwareDeveloper class

```
 internal class SoftwareDeveloper : Employee
{
	private static double bonusRate = 1.2;

	public SoftwareDeveloper(double wage)
	{
		Wage = wage;
	}

	public override double CalculateBonusAbstract()
	{
		Console.WriteLine("SoftwareDeveloper - 
CalculateBonusAbstract");
		return bonusRate * Wage;
	}

	public new double CalculateBonusNormal()
	{
		Console.WriteLine("SoftwareDeveloper - 
CalculateBonusNormal");
		return bonusRate * Wage;
	}
```

```
        public override double CalculateBonusVirtual()
        {
                Console.WriteLine("Employee - CalculateBonusVirtual");
                return bonusRate * Wage;
        }
}
```

5. Add the following method to the “Program” class and call it from the “Main()” method. Inside the method declare a `SoftwareDeveloper` object and call the `CalculateBonusAbstract`, `CalculateBonusNormal` and `CalculateBonusVirtual`

```
private static void AbstractNormalVirtualMethods()
{
        var softwareDeveloper = new SoftwareDeveloper(2000);

        //Abstract method
        Console.WriteLine("\n###Abstract");
        Console.WriteLine(softwareDeveloper.CalculateBonusAbstract());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusAbstract());

        //Normal method
        Console.Write("\n###Hide");
        Console.WriteLine(softwareDeveloper.CalculateBonusNormal());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusNormal());

        //Virtual method
        Console.Write("\n###Override");
        Console.WriteLine(softwareDeveloper.CalculateBonusVirtual());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusVirtual());
}
```

6. In the previous method declare an array of Employee[] and call the previously mentioned methods

6.2.Custom Interfaces

Activity

Let’s imagine that the company starts to work with external contractors. You are required to add this category of persons to the previously developed application. Moreover, the management is interested to quickly find what programming languages a Software Developers or Contractor knows.

1. Add the “Contractor” class

```
internal class Contractor : Person
      {

      }
```

2. Add the “IKnownProgrammingLanguages” interface

```
internal interface IKnownProgrammingLanguages
{
        string[] KnownProgrammingLanguages { get; set; }
        bool Knows(string language);
}
```

3. Derive the “SoftwareDevloper” and “Contractor” classes from the “IKnownProgrammingLanguages” interface

4. Add a new method to the "Program" class and call it from the "Main()" method. Inside the method define an array of persons and populate it with the three categories of persons in the company (SofwareDeveloper, Manager, Contractor).
5. In the previous method iterate the list of persons and display the known programming languages for each person
6. Search for all the persons that know "C#".