

# C# .Net Programming: A graphical approach

## Clase 5

Ing. Hazael Fernando Mojica García

- Object Oriented Programming
- Encapsulation, Inheritance and Polymorphism

# Object Oriented Programming

The Object Oriented Programming (**OOP**), is a computational paradigm based in objects, which are data structures. The **objects** uses classes to **encapsulate** the information (the attributes or **fields**) and the behavior (the **methods**).

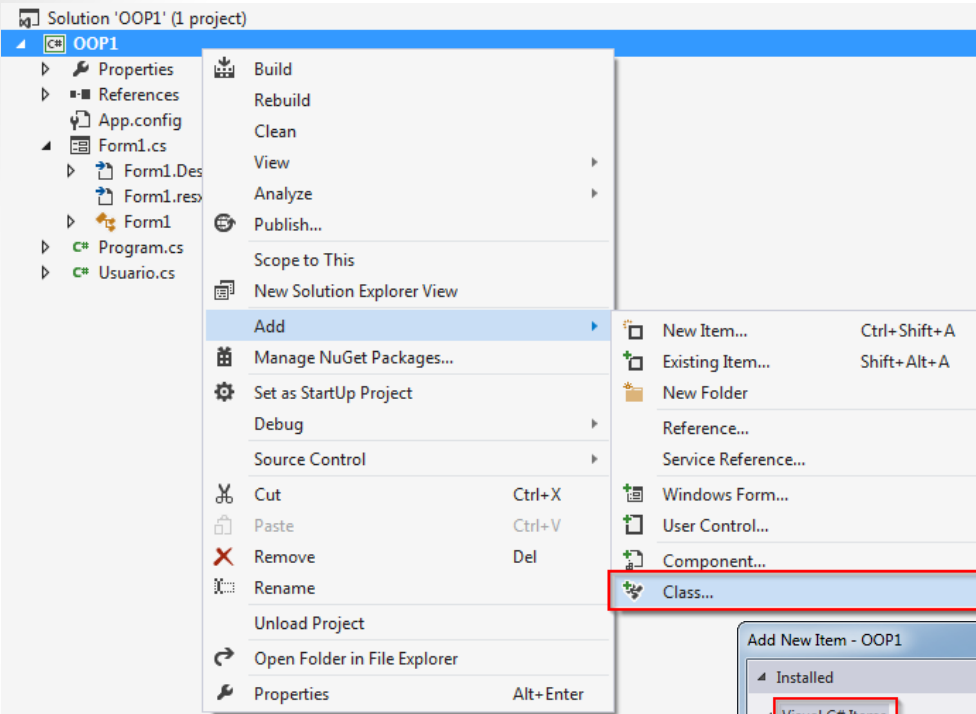
In *functional programming languages* like **C**, the programming is “action oriented”, in **C#** the programming is “object oriented”. In **C** the programming unit is the ***function***, in **C#** the programming unit is the ***class***.

The **C** programmers focus their effort in writing *functions*, the **C#** programmers focus their work in writing their own data structures, called classes.

We can say that in **C#**, *everything is an object* (an **instance of a class**), absolutely everything. Write classes allow the programmers to: *reuse, redistribute and encapsulate* the code in a better way than **C** libraries.

Note: I’m not saying **C#** is better than **C**, I’m just saying it allow us to do some thinks better. In the other hand **C** is the perfect language when you want performance, portability and code optimization.

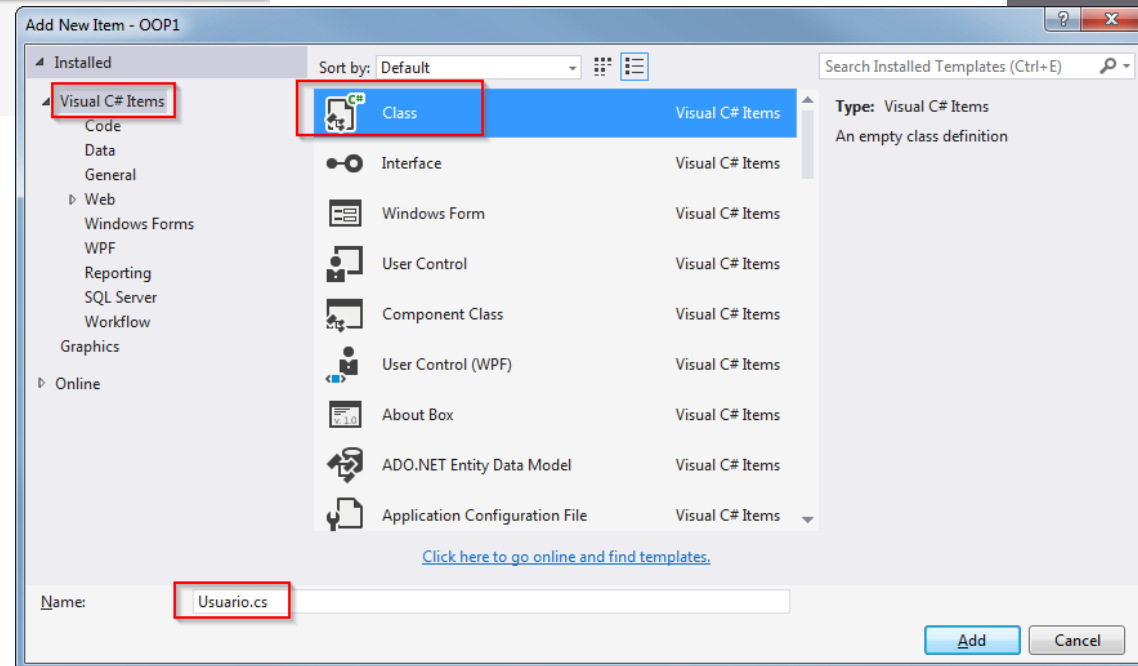
# My first class: User



Create a new Windows Forms  
C# Project and add a new class:

1. Right click in the project (In the Solution Explorer)
2. Add -> Class...

Write the name of the new  
class:  
**User.cs**



# Usuario.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace OOP1
```

```
{
    class Usuario
```

```
{
    string nombre;
    int edad;
```

```
//CONSTRUCTOR DE LA CLASE USUARIO
```

```
public Usuario()
```

```
{
    nombre = "Steve";
    edad = 56;
}
```

```
//PROPIEDADES - ATRIBUTOS - FIELDS
```

```
public string Nombre
```

```
{
    get { return nombre; }
    set { nombre = value; }
}
```

```
public int Edad
```

```
{
    get { return edad; }
    set { edad = value; }
}
```

}

Name of the class

}

Declaring the fields (attributes)

}

Constructor for the class User.

}

Read properties: **Get**  
Write properties: **Set**

}

}

}

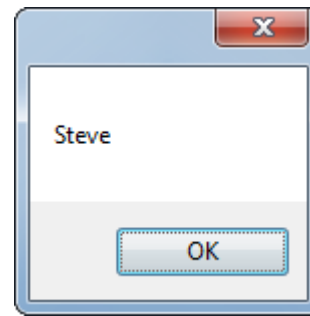
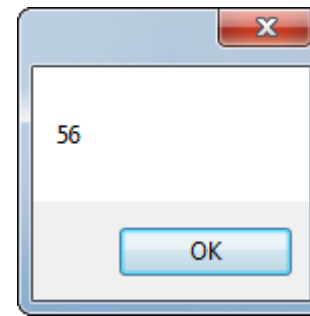
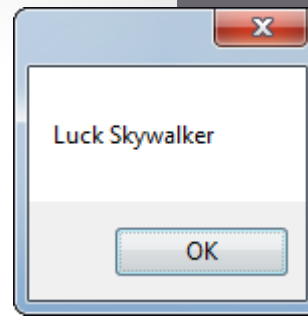
}

## Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace OOP1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

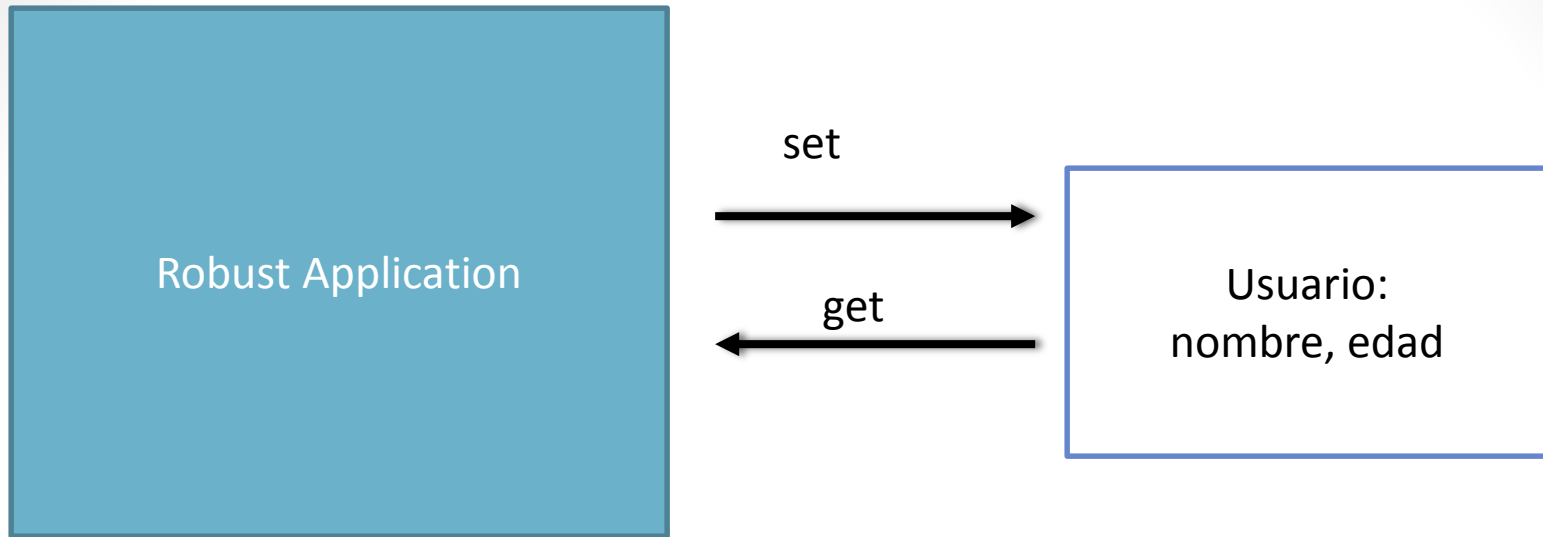
        private void Form1_Load(object sender, EventArgs e)
        {
            Usuario user = new Usuario();
            MessageBox.Show(user.Nombre);
            MessageBox.Show(user.Edad.ToString());
            user.Nombre = "Luck Skywalker";
            MessageBox.Show(user.Nombre);
        }
    }
}
```

A screenshot of a Windows Form window. The title bar is blue with a red 'X' button. The form has a white background with the text 'Steve' in the center. At the bottom, there is a blue 'OK' button.A screenshot of a Windows Form window. The title bar is blue with a red 'X' button. The form has a white background with the text '56' in the center. At the bottom, there is a blue 'OK' button.A screenshot of a Windows Form window. The title bar is blue with a red 'X' button. The form has a white background with the text 'Luck Skywalker' in the center. At the bottom, there is a blue 'OK' button.

We create an object called user, which is an instance of the class **Usuario**.

You can call the methods and properties (fields) using the dot '.' operator.

*Note: Double click in the Form background to create the Form1\_Load event*



We can see the class **Usuario** as an arrange of data which is encapsulated.  
We can access the data inside the class (name and age) using the properties defined by the get set keywords.

We can also access the behavior of the class, the methods.

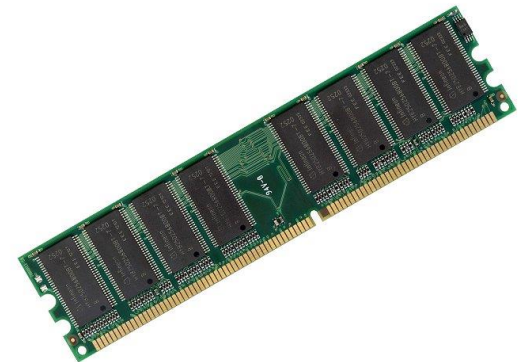
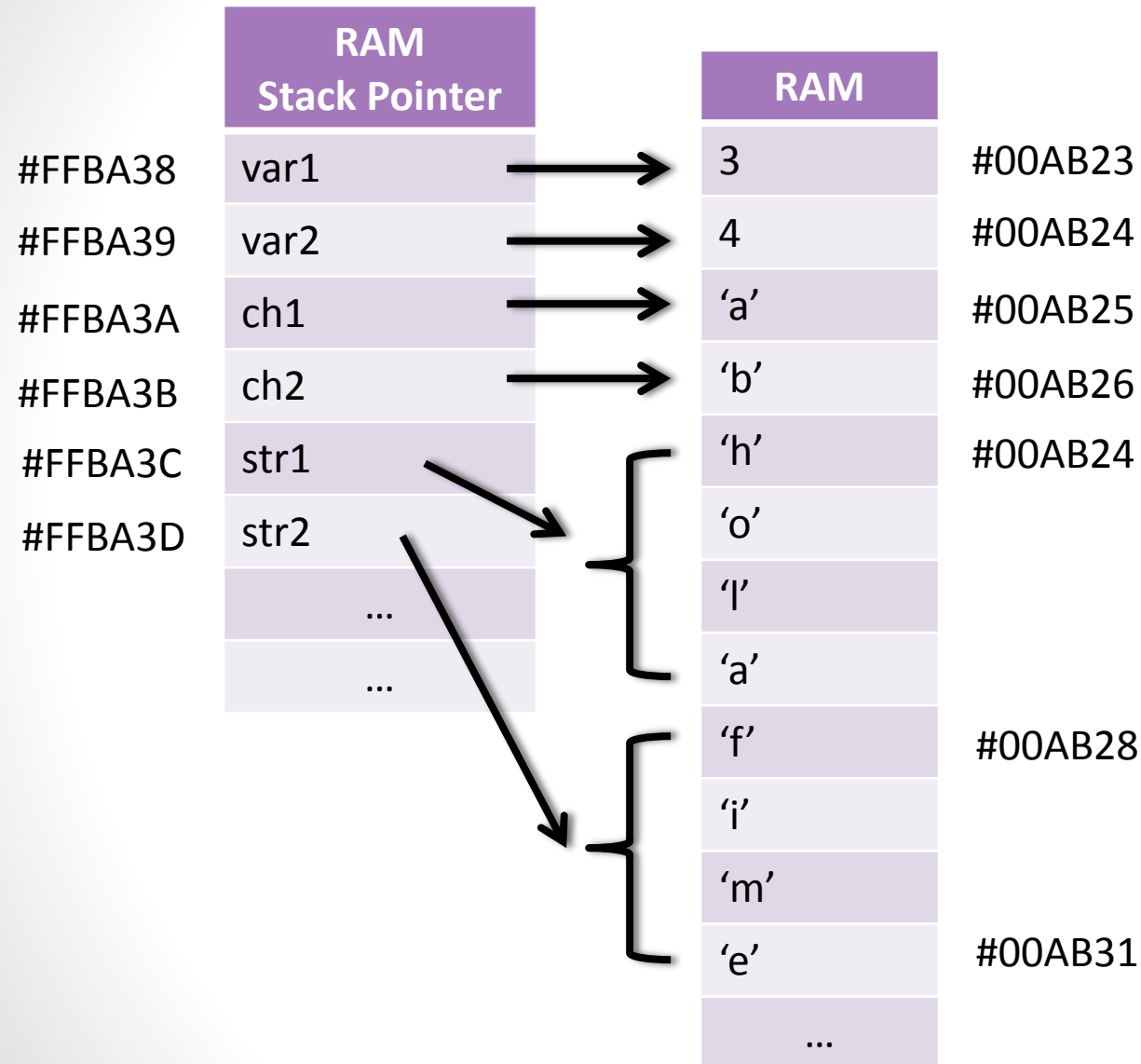
For accessing any of the above we need to use the dot operator ‘.’.

## Data Stack in RAM

```
int var1 = 3;  
int var2 = 4;
```

```
char ch1 = 'a';  
char ch2 = 'b';
```

```
string str1 = "hola";  
string str2 = "fime";
```



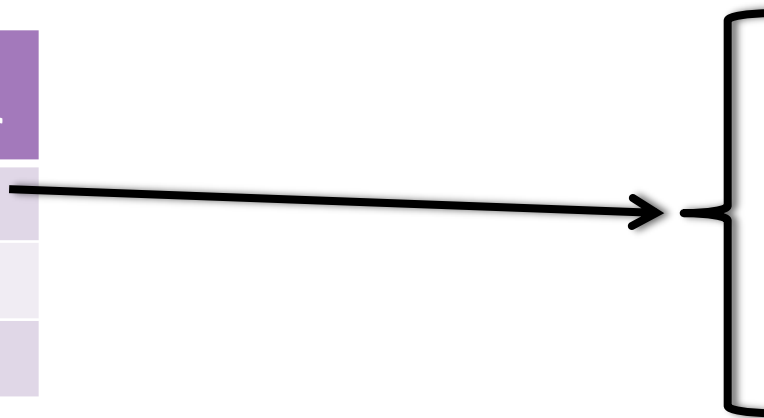
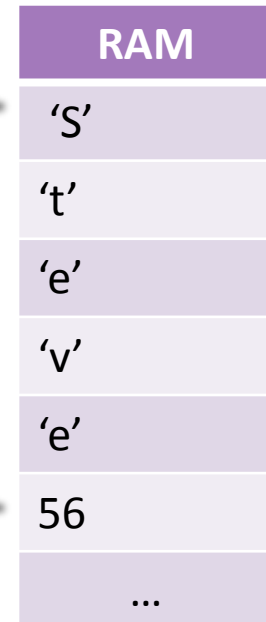
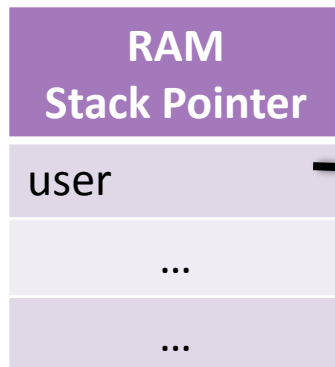
## Data stack in RAM - Classes

Object instance  
of the class  
**Usuario**.

```
Usuario user = new Usuario();
```

```
class Usuario
{
    string nombre;
    int edad;
    public Usuario()
    {
        nombre = "Steve";
        edad = 56;
    }
}
```

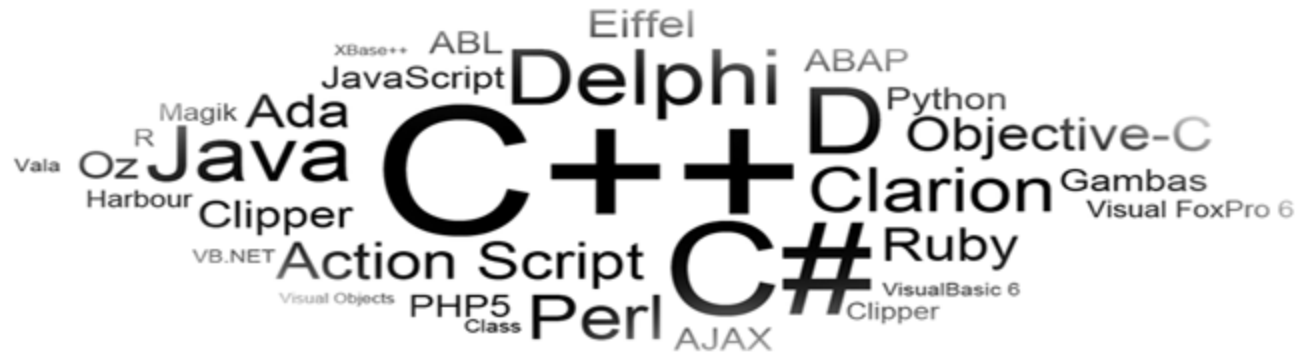
Declaring the class **Usuario**, this class has a constructor (*Usuario()*) and two fields (*nombre*, *edad*)





# OOP usage in the world

Robust applications nowadays demand to know at least the fundamentals of OOP, mainly due to the fact that the most used languages are build using this paradigm.



OOP allow us to model our world into a digital manner in the more natural way possible, this makes programming a lot easier.



```
if (top != self) {
function calcWidth() {
    var wW = 0;
    if (typeof window.innerWidth == 'number') {
        wW = window.innerWidth;
    } else if (document.documentElement.clientWidth) {
        wW = document.documentElement.clientWidth;
    } else if (document.body.clientWidth) {
        wW = document.body.clientWidth;
    }
    if (sH = document.documentElement.scrollHeight) {
        var wH = window.innerHeight || document.body.clientHeight;
        wW = !document.all && (sH > wH) ? wH : wW;
    }
}
```

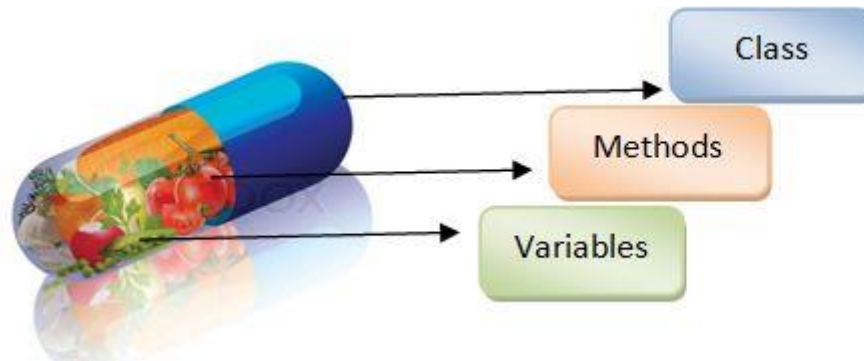


# Encapsulation

In Object Oriented programming **Encapsulation** is the first pace. Encapsulation is the procedure of covering up of data and functions into a single unit (called class).

In the previous example, the class **Usuario** encapsulates the fields age and name un a single unit called class.

We can access all the public attributes and methods of a class using the dot operator '.

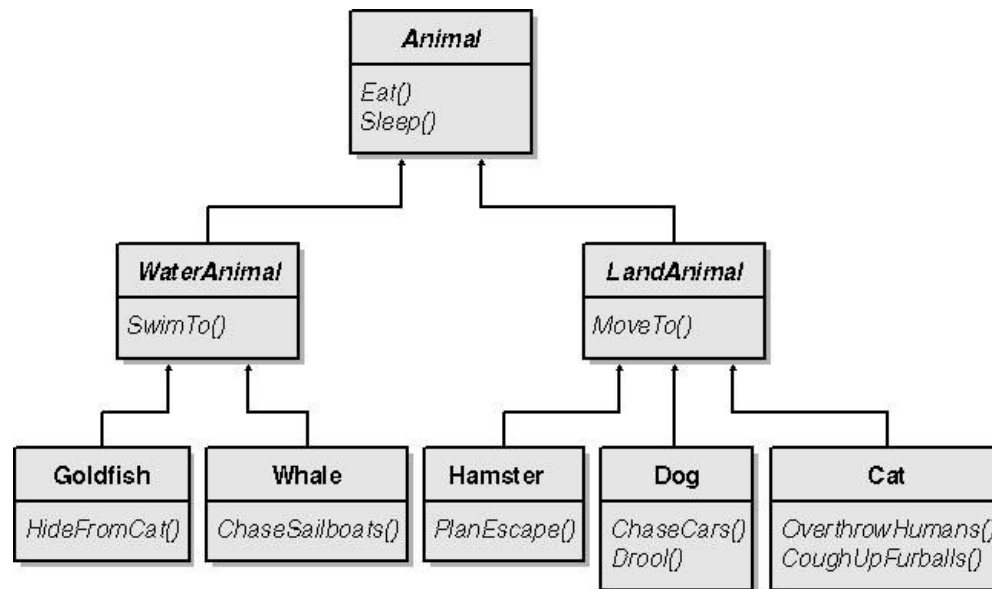


# Inheritance

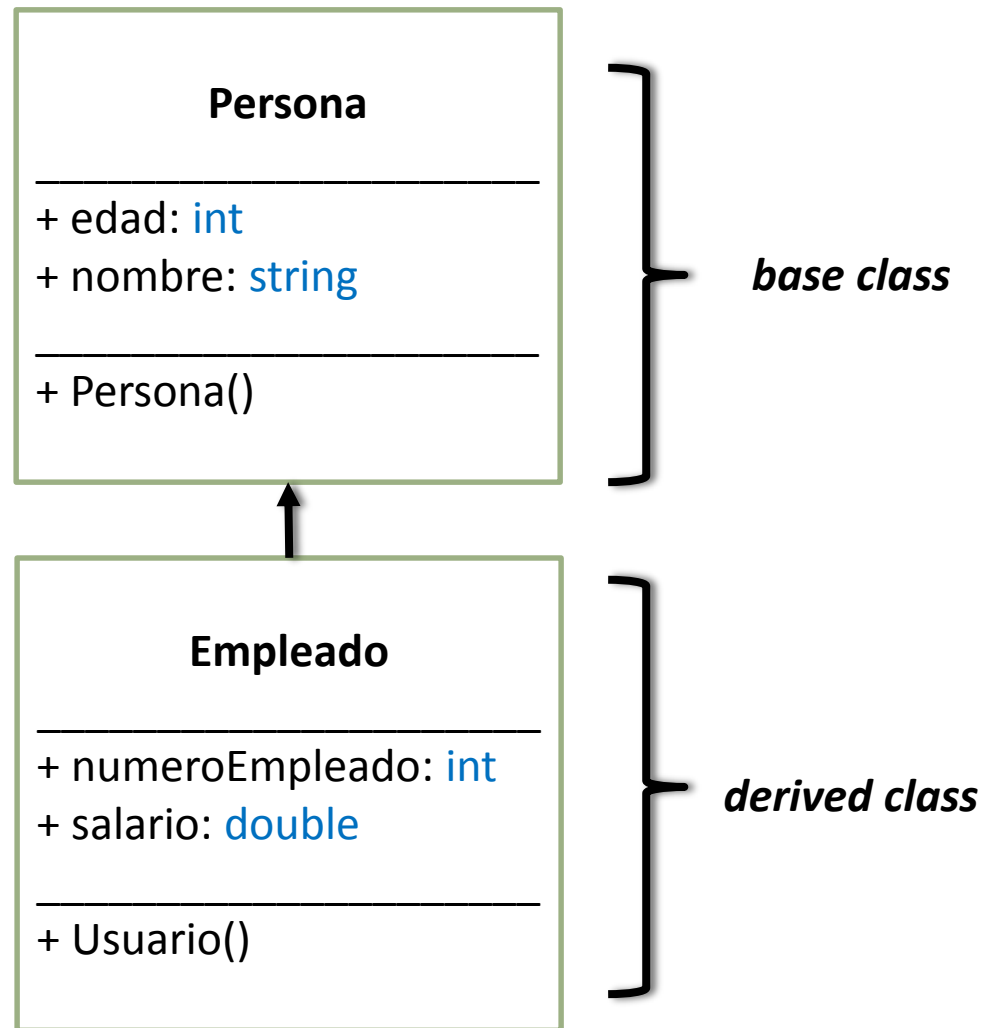
**Inheritance**, together with *encapsulation* and *polymorphism*, is one of the three primary characteristics (or *pillars*) of object-oriented programming.

Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class.

However, inheritance is transitive. If ClassC is derived from ClassB, and ClassB is derived from ClassA, ClassC inherits the members declared in ClassB and ClassA.



## The **PERSONA** class



# Creating a new Windows Forms projects and add the next classes.

## Persona.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Persona
    {
        int edad;
        string nombre;

        //Constructor de la clase Persona
        public Persona()
        {
            this.edad = 25;
            this.nombre = "Steve Jobs";
        }

        //Propiedades de la clase Persona
        public int Edad
        {
            get { return edad; }
            set { edad = value; }
        }
        public string Nombre
        {
            get { return nombre; }
            set { nombre = value; }
        }
    }
}
```

## Empleado.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Empleado : Persona
    {
        int numeroEmpleado;
        double salario;

        //Constructor de la clase Empleado
        public Empleado()
        {
            this.numeroEmpleado = 789456123;
            this.salario = 23500.3;
        }

        public int NumeroEmpleado
        {
            get { return numeroEmpleado; }
            set { numeroEmpleado = value; }
        }

        public double Salario
        {
            get { return salario; }
            set { salario = value; }
        }
    }
}
```

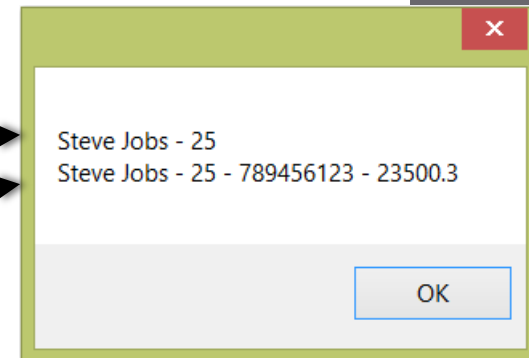
## Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace OOP1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            Persona persona = new Persona();
            Empleado empleado = new Empleado();

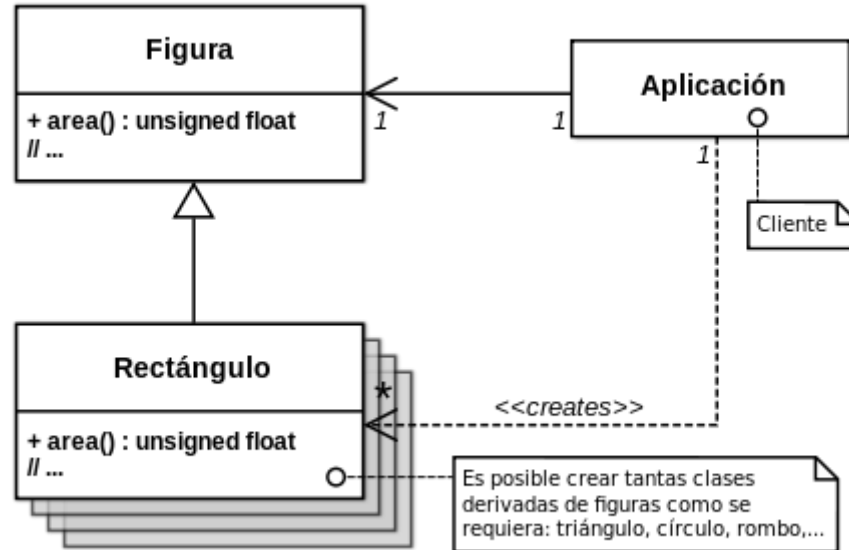
            string resultado = "";
            resultado += persona.Nombre + " - " + persona.Edad + "\n";
            resultado += empleado.Nombre + " - " + empleado.Edad + " - ";
            resultado += empleado.NumeroEmpleado + " - " + empleado.Salario + "\n";
            MessageBox.Show(resultado);
        }
    }
}
```



# Polymorphism

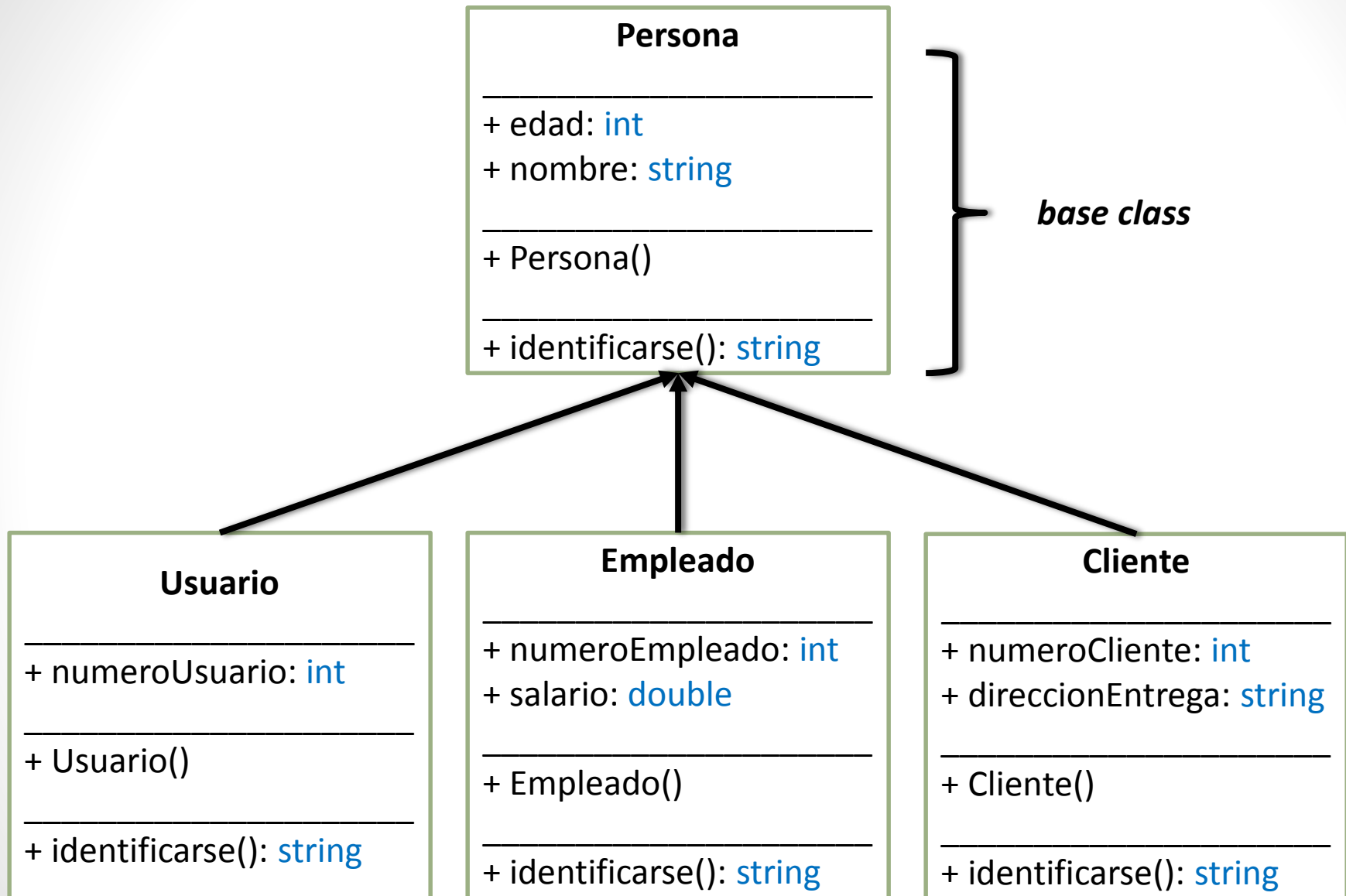
**Polymorphism** is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- The objects of a derived class can be treated as objects of the base class.
- The base class can define and implement **virtual methods** and derived classes can **override** those methods.



This example shows a base class called Figure and a derivate class called Rectangle.

## Extended class inheritance: **Person**





Add the next classes to a Windows Forms project.

## Persona.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Persona
    {
        int edad;
        string nombre;

        //Constructor de la clase Persona
        public Persona()
        {
            this.edad = 25;
            this.nombre = "Steve Jobs";
        }

        //Metodo publico
        public virtual string identificarse()
        {
            string mensaje = "";
            mensaje += this.nombre + " - ";
            mensaje += this.edad.ToString();
            return mensaje;
        }

        //Propiedades de la clase Persona
        public int Edad
        {
            get { return edad; }
            set { edad = value; }
        }
        public string Nombre
        {
            get { return nombre; }
            set { nombre = value; }
        }
    }
}
```

## Usuario.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Usuario : Persona
    {
        int numeroUsuario;

        //Constructor de la clase Usuario
        public Usuario()
        {
            this.numeroUsuario = 123456789;
        }

        //Sobreescritura del Metodo identificarse()
        public override string identificarse()
        {
            string mensaje = "";
            mensaje += base.identificarse();
            mensaje += " - " + this.numeroUsuario.ToString();
            return mensaje;
        }

        //Propiedades
        public int NumeroUsuario
        {
            get { return numeroUsuario; }
            set { numeroUsuario = value; }
        }
    }
}
```

## Cliente.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Cliente : Persona
    {
        int numeroCliente;
        string direccionEntrega;

        public Cliente()
        {
            this.numeroCliente = 456123789;
            this.direccionEntrega = "Ciudad Universitaria s/n";
        }

        //Sobreescritura del Metodo identificarse()
        public override string identificarse()
        {
            string mensaje = "";
            mensaje += base.identificarse();
            mensaje += " - " + this.numeroCliente.ToString();
            mensaje += " - " + this.direccionEntrega;
            return mensaje;
        }

        //Propiedades
        public int NumeroCliente
        {
            get { return numeroCliente; }
            set { numeroCliente = value; }
        }

        public string DireccionEntrega
        {
            get { return direccionEntrega; }
            set { direccionEntrega = value; }
        }
    }
}
```

## Empleado.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP1
{
    class Empleado : Persona
    {
        int numeroEmpleado;
        double salario;

        //Constructor de la clase Empleado
        public Empleado()
        {
            this.numeroEmpleado = 789456123;
            this.salario = 23500.3;
        }

        //Sobreescritura del Metodo identificarse()
        public override string identificarse()
        {
            string mensaje = "";
            mensaje += base.identificarse();
            mensaje += " - " + this.numeroEmpleado.ToString();
            mensaje += " - " + this.salario.ToString();
            return mensaje;
        }

        public int NumeroEmpleado
        {
            get { return numeroEmpleado; }
            set { numeroEmpleado = value; }
        }

        public double Salario
        {
            get { return salario; }
            set { salario = value; }
        }
    }
}
```

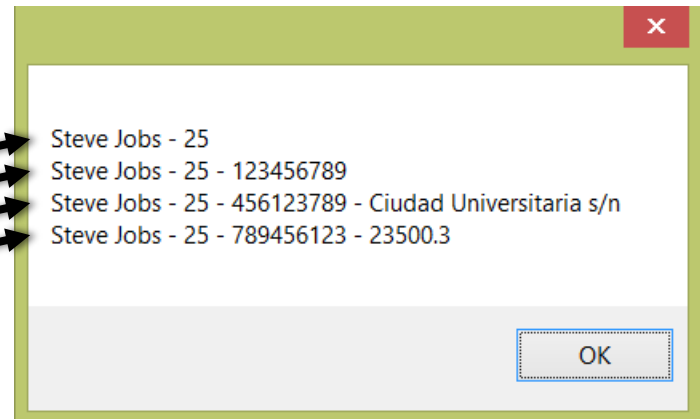
# Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace OOP1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

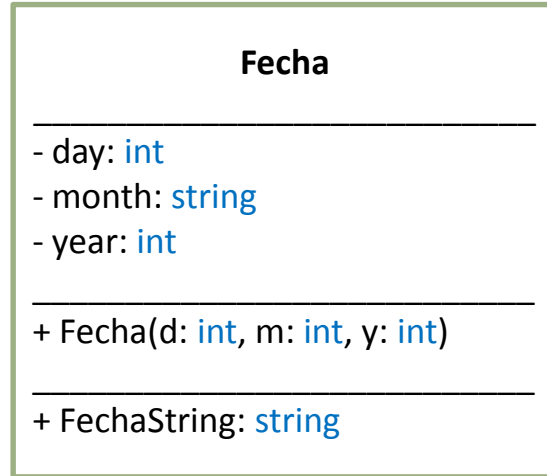
        private void Form1_Load(object sender, EventArgs e)
        {
            Persona persona = new Persona();
            Usuario user = new Usuario();
            Cliente cliente = new Cliente();
            Empleado empleado = new Empleado();

            string resultado = "";
            resultado += persona.identificarse() + "\n";
            resultado += user.identificarse() + "\n";
            resultado += cliente.identificarse() + "\n";
            resultado += empleado.identificarse() + "\n";
            MessageBox.Show(resultado);
        }
    }
}
```



# Homework 1

Implement the code for the next UML diagram.



The constructor takes as arguments: days (dd), months (mm), years (yyy). The string called **FechaString** only return the date in the format “dd-mm-yyyy” or an error message if it’s not correct.

The months January, March, May, July, August, October and December has 31 days. April, June, September and November have 30 days. February has 28 days if not in a leap year and 29 when it’s inside a leap year.

A leap year can be identified if it is divisible by 4. However, an special case is when the year is also divisible by 100 and in that case to be considered as leap year you must check if it is divisible by 400.

**TIP: Usar el código de 5.4-OOP1**

# Homework 2

You are going to implement a Web based system using .Net C# for a metalurgic company in Monterrey. The system must handle the maintenance periods for each machine. We can divide the machines by types depending how often the machine must proceed into maintenance.

- Light Machines: Once a week
- Hard Machines: Once every two weeks
- Heavy Duty Machines: Once every four weeks

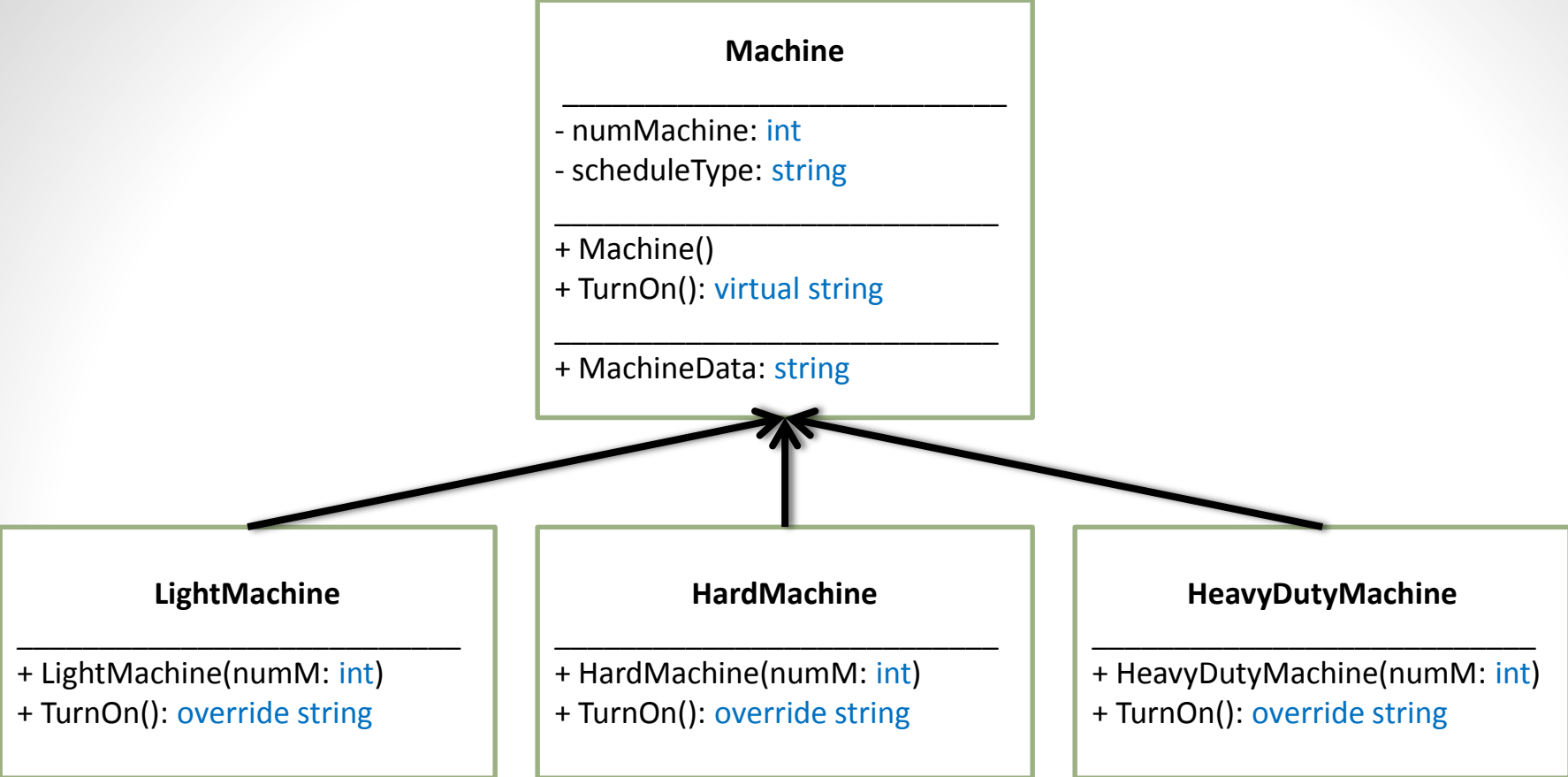
Create a project in C# using WindowsForms, deploy the code for the next UML diagram. Use the next code to test the correct functionality of your implementation.

```
private void Form1_Load(object sender, EventArgs e)
{
    Machine machine = new Machine();
    LightMachine lightMachine = new LightMachine(123);
    HardMachine hardMachine = new HardMachine(456);
    HeavyDutyMachine heavyMachine = new HeavyDutyMachine(789);

    //Mandamos mensaje concatenado de la ejecucion
    //del metodo TurnOn();
    string resultado = "";
    resultado += machine.TurnOn() + "\n";
    resultado += lightMachine.TurnOn() + "\n";
    resultado += hardMachine.TurnOn() + "\n";
    resultado += heavyMachine.TurnOn() + "\n";
    MessageBox.Show(resultado);

    //Mandamos mensaje concatenado de la obtencion
    //de la propiedad MachineData
    resultado = "";
    resultado += machine.MachineData + "\n";
    resultado += lightMachine.MachineData + "\n";
    resultado += hardMachine.MachineData + "\n";
    resultado += heavyMachine.MachineData + "\n";
    MessageBox.Show(resultado);
}
```

*Be free to change this testing code according to your needs.*



The constructors of the derived classes take as input parameters the **id/number(numM)** of the machine.

The method **TurnOn()** needs to be overwritten in each derived class, this method return a string message allusive to each machine type, for example: "Turning on the LightMachine"...

The propertie **MachineData** of the class **Machine** returns a string with format: "*numMachine - scheduleType*".

Remember to initialize with some value the instance variables *numMachine* and *scheduleType* in the constructor of the class **Machine**, is good programming practice to do so.