Programación Orientada a Objetos

Por Jesús Alberto Zamarripa,
Northware Project Manager

Febrero 2012

Northware®
Software Development

Northware®
Software Development

La Programación Orientada a Objetos (POO) es el paradigma de programación más utilizado en la actualidad.

Su consistente base teórica y la amplia gama de herramientas que permiten crear código a través de diseños orientados a objetos la convierten en la alternativa más adecuada para el desarrollo de aplicaciones. Se expondrán los conceptos de este paradigma y las características de este tipo de programación aplicadas al lenguaje C#.

La POO se refiere a transformar el mundo real en código.

En las imágenes a continuación vemos hay 2 objetos: un automóvil y un león. Como objetos ambos tienen rasgos que los caracterizan (Propiedades) y pueden realizar acciones (métodos). Una vez identificados, debemos proceder a transportar este concepto a nuestro código. Generando primeramente nuestras Clases, que nos servirán para la creación de nuestros objetos.

Características
- Marca
- Color
- Tipo
- Precio
- No. de Puertas
- Tipo Combustible
- Cilindros
- Transmisión

Acciones
- Encender
- Avanzar
- Retroceder
- Detener
- Apagar

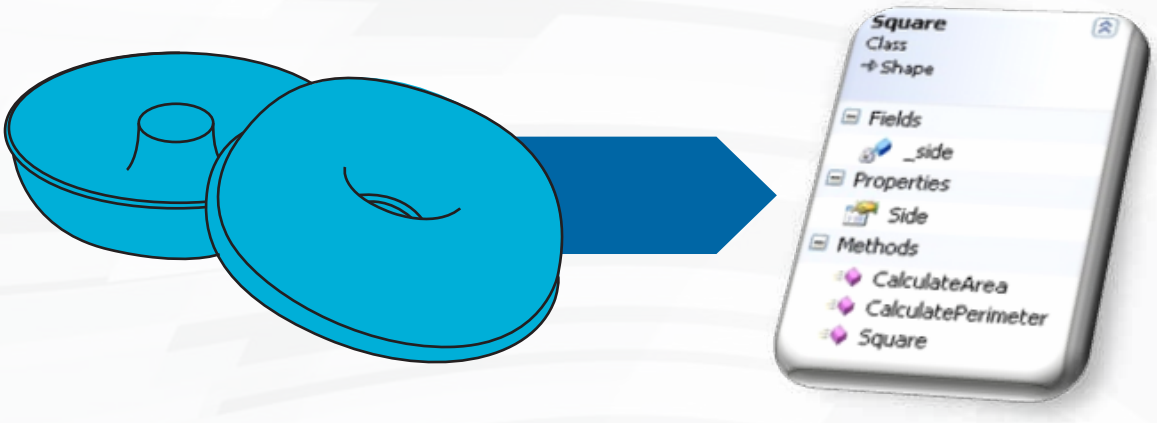
Características
- Nombre
- Especie
- Color
- Edad

Acciones
- Comer
- Dormir
- Correr

¿Qué es una Clase?

Al igual que un diseñador crea prototipos de dispositivos que podrían utilizarse en repetidas ocasiones para construir los dispositivos reales, una clase es un prototipo de software que puede utilizarse para instanciar (es decir crear) muchos objetos iguales.

También lo podemos visualizar como un molde, sí, por ejemplo uno para hacer gelatinas, con ese molde podemos hacer muchas gelatinas con la misma forma y tamaño. Pues ese molde sería nuestra clase y la instancia de una clase sería nuestro objeto.



Sintaxis para generar una clase

```
<modificador>class<nombre_clase>
{
        <declaración_atributo>
        <declaración_constructor>
        <declaración_método>
}
```

```
public class Vehiculo
{
      private double cargaMax;
      public Vehiculo()
      {
            cargaMax = 0;
      }
      public void SetCargaMax (int valor)
      {
            cargaMax= valor;
      }
}
```

¿Qué contiene una clase?

Los miembros de una clase son un conjunto de elementos que definen a los objetos (atributos ó propiedades), así como los comportamientos o funciones (métodos) que maneja el objeto.

Entonces tenemos que una clase es la estructura de un objeto, es decir, la definición de todos los elementos de que está hecho un objeto.

Los atributos se declaran de la siguiente forma:

<modificador><tipo><nombre> [= <valor_inicial>];

```
public class Hotel
{
      private float x;
      public int y = 1000;
      private string name = "Hotel Central";
}
```

Los métodos de una clase se declaran de la siguiente forma:

<modificador><tipo_retorno><nombre> (<argumentos>) { <sentencia> }

```
public class Perro
{
      private int Peso;
      public int GetPeso()
      {
            return Peso;
      }
      public void SetPeso(int newPeso)
      {
            if (newPeso > 0)
            {
                  Peso = newPeso;
            }
      }
}
```

¿Qué es el constructor de una clase?

El constructor es un miembro que implementa las acciones requeridas para inicializar la instancia de una clase. El constructor es invocado cuando se usa el operador new.

El constructor es parecido a los métodos, pero no tiene un tipo de retorno y su nombre es el mismo que el de la clase y también puede o no recibir parámetro como los métodos. Sin embargo hay que tener mucho cuidado al nombrar los parámetros tanto del constructor como de los métodos para no confundirlos con las variables propias de la clase. Por ejemplo:

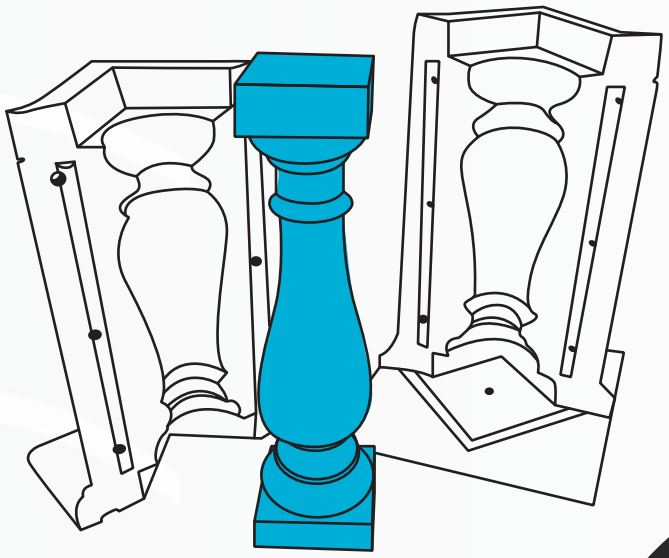
```
public class Persona
{
      public Persona() {}
      {
      this.Nombres = string.Empty;
      this.Apellidoss = string.Empty;
      this.Edad = 0;
      }

      #region Properties
      private string Nombres { get; set; }
      private string Apellidos { get; set; }
      private int Edad { get; set; }
      private string NombreCompleto [...]
      #endregion

}
```

Analogía de una clase y un objeto

En esta analogía quise que gráficamente vieran a la clase como el molde, y una vez que hacemos una instancia de una clase, se crea el objeto, en este caso el barrote.



Tipos de variables que puede tener una clase

De instancia

Son aquellas que se declaran dentro de una clase y que no existen hasta que se hace una instancia de esa clase (se crea un objeto)

<modificador><tipo> nombre [= <valor_inicial>];

```
public int Edad = 25;
```

De referencia

Son aquellas que hace referencia a otra clase. Esto se puede definir mejor diciendo que es una variable del tipo de otra clase, y cuando se haga la instancia tendremos: un objeto dentro de otro objeto.

<modificador><Clase> nombre [= <instancia>];

```
public Persona P = new Persona();
```

De clase

Estas variables se declaran antemoniendo la palabra static a su declaración, y a diferencia de las variables de instancia, éstas no necesitan que se haga una instancia (que se cree un objeto), existen desde que se crea la clase.

<modificador>static<tipo> nombre [= <valor_inicial>];

```
public static Boolean Flag;
```

Tipos de variables que puede tener una clase (cont.)

Una vez que tenemos definidos todos los miembros de una clase, ¿Cómo accedemos a ellos?

Para acceder a los miembros de un objeto, se utiliza la notación de pinto. **El operador punto (.)** permite acceder a los atributos y métodos que componen los miembros no privados de una clase.

Dentro de la definición de los métodos no es necesario usar la notación de punto para acceder a un atributo de la propia clase.

Modificadores de Acceso

Es la forma en cómo clasificamos nuestras propiedades o métodos para hacerlos visibles o invisibles a quienes usen un objeto de una clase que creamos. Los modificadores de acceso son: **Public, Private, Protected**.

En seguida muestro en una tabla la visibilidad de los miembros de una clase dependiendo del modificador de acceso que tengan.

Visibilidad	Public	Private	Protected	Default*
Desde la misma clase	✓	✓	✓	✓
Desde una subclase	✓	✓	✓	⊘
Desde otra clase (no subclase)	✓	⊘	⊘	⊘

**Con default, me refiero a omitir el modificador de acceso.*

Encapsulación

Es un mecanismo que permite a los diseñadores de tipos de datos determinar qué miembros de los tipos pueden ser utilizados por otros programadores y cuáles no. Las principales ventajas que ello aporta son:

Encapsulación (cont.)

- Se facilita a los programadores que vaya a usar el tipo de dato (programadores clientes) el aprendizaje de cómo trabajar con él, pues se le pueden ocultar todos los detalles relativos a su implementación interna y sólo dejarle visibles aquellos que puedan usar con seguridad. Además, así se les evita que cometan errores por manipular inadecuadamente miembros que no deberían tocar.

- Se facilita al creador del tipo la posterior modificación del mismo, pues si los programadores clientes no pueden acceder a los miembros no visibles, sus aplicaciones no se verán afectadas si éstos cambian o se eliminan. Gracias a esto es posible crear inicialmente tipos de datos con un diseño sencillo aunque poco eficiente, y si posteriormente es necesario modificarlos para aumentar su eficiencia, ello puede hacerse sin afectar al código escrito.

- La encapsulación se consigue añadiendo modificadores de acceso en las definiciones de miembros y tipos de datos. Estos modificadores son partículas que se les colocan delante para indicar desde qué códigos puede accederse a ellos, entendiéndose por acceder el hecho de usar su nombre para cualquier cosa que no sea definirlo, como llamarlo si es una función, leer o escribir su valor si es un campo, crear objetos o heredar de él si es una clase, etc.

Ejemplo:

```
//Definición de métodos públicos (interfaz pública de la clase).
class clase_cliente
{
     //Definición de variables de instancia.
     private string is_nombre;
     private string is_ape1;
     private string is_ape2;

     public void nombre (string nombre, string ape1, string ape2){
          is_nombre = nombre;
          is_ape1 = ape2;
          is_ape2 = ape2;
     }

     public string nombre(){
          return nombre_completo();
     }

     //Definición de métodos de soporte.
     private string nombre_completo(){
          return "Nombre: " + is_nombre + " | Primer ape.: " +
          is_ape1 + " | Segundo ape.: " + is_ape2 ;
     }
}
```

¿Qué es la Herencia?

La herencia se da cuando una clase hereda sus propiedades y métodos a otra clase, ésta segunda se convierte en la subclase y la primera en clase superior o clase padre. Es decir, una clase base o clase padre le hereda sus propiedades a la clase hija o subclase. **Ejemplo:**

```
public class Persona
{
     public Persona()[...]
     public Persona (string N, string A, int E)[...]

     #region Properties
     private string Nombres { get; set; }
     private string Apellidos { get; set; }
     private int Edad { get; set; }
     private string NombreCompleto[...]
     #endregion

     #region Methods
     private string GetFullName()[...]
     private int GetAge()[...]
     private void SetName(string N)[...]
     private void SetLastName(string A)[...]
     private void SetAge(int E)[...]
     #endregion

}

public class Empleado:Persona
{
     [constructor]

     private long Nomina { get; set; }
     private string Imss{ get; set; }

     public long GetNomina[...]
     public string GetImss[...]
}
```

Después de instanciar un objeto de tipo Empleado, como Empleado hereda de Persona, ya podemos hacer uso de las propiedades de la clase Persona. Nombre completo es una propiedad de la clase persona que ya la podemos usar en nuestro objeto empleado gracias a la herencia.

```
Empleado E = New Empleado();
```

Método Override

Este método proporciona una nueva implementación de un miembro heredado de una clase base. El método reemplazado por una declaración override se conoce como método base reemplazado. El método base reemplazado debe tener la misma firma que el método **override**.

Una declaración de propiedad de reemplazo debe especificar el mismo modificador de acceso, tipo y nombre que la propiedad heredada. **Ejemplo:**

```
public class Animal {
     public void Hablar(){
          Console.WriteLine("Estoy comunicándome...");
     }
}
public class Perro : Animal {
     public new void Hablar(){
          Console.WriteLine("¡Guau!");
     }
}
public class Gato : Animal {
     public new void Hablar(){
          Console.WriteLine("Miauuu");
     }
}


public class Ejemplo {
     public static void Main(){
          //Primero creamos un animal de cada tipo
          Perro miPerro = new Perro();
          Gato miGato = new Gato();
          Animal miAnimal = new Animal();

          miPerro.Hablar();
          miGato.Hablar();
          miAnimal.Habler();

          //Línea en blanco, por legibilidad
          Console.WriteLine();

          //Ahora los creamos desde un array
          Animal[] misAnimales = new Animal[3];

          misAnimales[0] = new Perro();
          misAnimales[1] = new Gato();
          misAnimales[2] = new Animal();

          misAnimales[0].Hablar();
          misAnimales[1].Hablar();
          misAnimales[2].Hablar();
     }
}
```

Resultado

```
Guau!
Miauuu
Estoy comunicándome…

Estoy comunicándome…
Estoy comunicándome…
Estoy comunicándome…
```

Método Override (cont.)

El mismo Ejemplo con Override:

```
public class Animal {
     public virtual void Hablar(){
       Console.WriteLine("Estoy comunicándome...");
     }
}
public class Perro : Animal {
     public override void Hablar(){
          Console.WriteLine("¡Guau!");
     }
}
public class Gato : Animal {
     public override void Hablar(){
          Console.WriteLine("Miauuu");
     }
}
```

Resultado

Guau!
Miauuu
Estoy comunicándome…

Guau!
Miauuu
Estoy comunicándome…

¿Qué es el Polimorfismo?

En este ultimo ejemplo pudimos observar este concepto. A través de la herencia, una clase puede utilizarse como más de un tipo; puede utilizarse como su propio tipo, o como el tipo de su clase base.

En el ejemplo mencionado, tanto la clase Perro como la clase Gato, heredaban de la calase animal y gracias a la herencia podemos hacer que Perro y Gato se comporten como tal ó como animales (clase base). Es por eso que necesitamos indicar qué comportamiento tomarán en un caso u otro.

Si se dieron cuenta, el arreglo que declaramos fue un arreglo de Animales, en el cual agregamos un objeto Animal, uno objeto Perro y un objeto Gato, a pesar de ser objetos de clases diferentes, pues al final, las 3 clases son Animales.

Northware®
Software Development

Estoy seguro que siguiendo estas líneas, tendrás un mejor resultado al final de tu proyecto de desarrollo.

Si prefieres recibir **ayuda profesional** y evitar errores en la planeación financiera y de calidad en tu proyecto de desarrollo, te invito a que nos contactes. Somos una empresa especialista en desarrollo de aplicaciones, base de datos y aplicaciones para Iphone/Ipad.

Desarrollamos software basado en Microsoft .net, java y iOS; y para aquellas empresas que sólo requieren la contratación directa de especialistas, proveemos consultores por proyecto, temporales o fijos con experiencia en las tecnologías más avanzadas para apoyar tú estrategia en sistemas de información y desarrollo de software.

Contáctanos

Interior de la República Mexicana
01 800 288 OPEN (6736)

Ciudad e México (D.F.)
(55) 5536 2968

Monterrey, Nuevo León
(81) 8262 1111

Desde Estados Unidos (U.S.A.)
(512) 853 9472

Síguenos en
facebook.com/northware **twitter.com/northwaremx**

Nuestro correo electrónico
info@northware.mx

Northware®
Software Development

Febrero 2012