

**POLIMORFISMO**

**I.- Ficha técnica**

|  |  |
| --- | --- |
| CARRERA | Técnico de nivel superior en Análisis de sistemas |
| CURSO | Lenguaje de Programación II |
| ASOCIADA AL APRENDIZAJE ESPERADO | Aplicar principios fundamentales de la programación orientada a objetos |
| ASOCIADA A LOS CRITERIOS DE EVALUACIÓN | Define clases, objetos, herencia, polimorfismo |
| CLASES (n°) | Clases 10-11 |
| TIPO | Personal |
| DESCRIPCIÓN | Implementación de polimorfismo entre métodos de clases |
| EVIDENCIA | Un programa que implemente polimorfismo en una clase |
| ESPACIO | Laboratorio de computadores |
| INSUMOS | Visual Studio 2010, 2013 ó 2015 |

**Materiales/Equipos/Herramientas (M/E/H)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Nombre** | **Tipo** | **Marca** | **M/E/H** | **Unidad de medida** | **Cantidad** | **Cada N alumnos** | **Observaciones** |
| Un computador |  |  |  |  | 1 | 1 | Operativo |
| Visual Studio 2010, 2013 ó 2015 |  |  |  |  | 1 | 1 | Instalado |

**II.- MATERIAL PARA EL ESTUDIANTE**

Se adjunta guía POLIMORFISMO

III.- **PROCEDIMIENTO DE LA EXPERIENCIA**

1. Lea atentamente la guía, si tiene dudas consulte con el docente
2. La experiencia es individual. No requiere del uso de material ajeno a ella. Siga las instrucciones expuestas en el documento
3. La experiencia es formativa
4. El producto obtenido en el desarrollo de la experiencia debe ser enviado al docente quien indicará, oportunamente, la forma más apropiada de hacerlo.

**POLIMORFISMO**

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

* En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices. Cuando ocurre, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.
* Las clases base pueden definir e implementar *métodos* virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esa invalidación del método virtual. Por lo tanto, en el código fuente puede llamar a un método en una clase base y hacer que se ejecute una versión del método de la clase derivada.

Los métodos virtuales permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, supongamos que tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie de dibujo. En tiempo de compilación, no sabe qué tipos específicos de formas creará el usuario. Sin embargo, la aplicación tiene que realizar el seguimiento de los distintos tipos de formas que se crean, y tiene que actualizarlos en respuesta a las acciones del mouse del usuario. Para solucionar este problema en dos pasos básicos, puede usar el polimorfismo:

1. Crear una jerarquía de clases en la que cada clase de forma específica deriva de una clase base común.
2. Usar un método virtual para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

Primero, cree una clase base llamada Shape y clases derivadas como Rectangle, Circle y Triangle.Dé a la clase Shape un método virtual llamado Draw e invalídelo en cada clase derivada para dibujar la forma determinada que la clase representa. Cree un objeto List<Shape> y agregue Circle, Triangle y Rectangle a él.Para actualizar la superficie de dibujo, use un bucle [foreach](https://msdn.microsoft.com/es-cl/library/ttw7t8t6.aspx) para iterar por la lista y llamar al método Draw en cada objeto Shape de la lista.Aunque cada objeto de la lista tenga un tipo declarado de Shape, se invocará el tipo en tiempo de ejecución (la versión invalidada del método en cada clase derivada).

C#

public class Shape

{

// A few example members

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

// Virtual method

public virtual void Draw()

{

Console.WriteLine("Performing base class drawing tasks");

}

}

class Circle : Shape

{

public override void Draw()

{

// Code to draw a circle...

Console.WriteLine("Drawing a circle");

base.Draw();

}

}

class Rectangle : Shape

{

public override void Draw()

{

// Code to draw a rectangle...

Console.WriteLine("Drawing a rectangle");

base.Draw();

}

}

class Triangle : Shape

{

public override void Draw()

{

// Code to draw a triangle...

Console.WriteLine("Drawing a triangle");

base.Draw();

}

}

class Program

{

static void Main(string[] args)

{

// Polymorphism at work #1: a Rectangle, Triangle and Circle

// can all be used whereever a Shape is expected. No cast is

// required because an implicit conversion exists from a derived

// class to its base class.

System.Collections.Generic.List<Shape> shapes = new System.Collections.Generic.List<Shape>();

shapes.Add(new Rectangle());

shapes.Add(new Triangle());

shapes.Add(new Circle());

// Polymorphism at work #2: the virtual method Draw is

// invoked on each of the derived classes, not the base class.

foreach (Shape s in shapes)

{

s.Draw();

}

// Keep the console open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

Drawing a rectangle

Performing base class drawing tasks

Drawing a triangle

Performing base class drawing tasks

Drawing a circle

Performing base class drawing tasks

\*/

En C#, cada tipo es polimórfico porque todos los tipos, incluidos los definidos por el usuario, heredan de [Object](https://msdn.microsoft.com/es-cl/library/system.object.aspx).

[Introducción al polimorfismo](javascript:void(0))

[Miembros virtuales](javascript:void(0))

Cuando una clase derivada hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la clase base. El diseñador de la clase derivada tiene las siguientes opciones:

* Invalidar los miembros virtuales de la clase base.
* Heredar el método de la clase base más próximo sin invalidarlo.
* Definir una nueva implementación no virtual de esos miembros que oculte las implementaciones de la clase base.

Una clase derivada puede invalidar un miembro de la clase base si este se declara como [virtual](https://msdn.microsoft.com/es-cl/library/9fkccyh4.aspx) o [abstracto](https://msdn.microsoft.com/es-cl/library/sf985hc5.aspx).El miembro derivado debe usar la palabra clave [override](https://msdn.microsoft.com/es-cl/library/ebca9ah3.aspx) para indicar explícitamente que el propósito del método es participar en una invocación virtual. El siguiente fragmento de código muestra un ejemplo:

C#

public class BaseClass

{

public virtual void DoWork() { }

public virtual int WorkProperty

{

get { return 0; }

}

}

public class DerivedClass : BaseClass

{

public override void DoWork() { }

public override int WorkProperty

{

get { return 0; }

}

}

Los campos no pueden ser virtuales; solo los métodos, propiedades, eventos e indizadores pueden ser virtuales. Cuando una clase derivada invalida un miembro virtual, ese miembro es llamado aun cuando una instancia de esa clase está siendo accedida como una instancia de la clase base. El siguiente fragmento de código muestra un ejemplo:

C#

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Also calls the new method.

Los métodos y propiedades virtuales permiten a las clases derivadas extender una clase base sin necesidad de usar la implementación de clase base de un método.

Una interfaz proporciona otra manera de definir un método o conjunto de métodos cuya implementación se deja a las clases derivadas.

[**Ocultar miembros de clase base con nuevos miembros**](javascript:void(0))

Si quiere que el miembro derivado tenga el mismo nombre que un miembro de una clase base, pero no quiere que participe en la invocación virtual, puede usar la palabra clave [new](https://msdn.microsoft.com/es-cl/library/51y09td4.aspx).La palabra clave **new** se coloca antes que el tipo devuelto del miembro de la clase que se está reemplazando.El siguiente fragmento de código muestra un ejemplo:

C#

public class BaseClass

{

public void DoWork() { WorkField++; }

public int WorkField;

public int WorkProperty

{

get { return 0; }

}

}

public class DerivedClass : BaseClass

{

public new void DoWork() { WorkField++; }

public new int WorkField;

public new int WorkProperty

{

get { return 0; }

}

}

Aún se puede acceder a los miembros de la clase base ocultos desde el código de cliente convirtiendo la instancia de la clase derivada en una instancia de la clase base.Por ejemplo:

C#

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Calls the old method.

[Evitar que las clases derivadas invaliden los miembros virtuales](javascript:void(0))

Los miembros virtuales permanecen virtuales indefinidamente, independientemente de cuántas clases se hayan declarado entre el miembro virtual y la clase que originalmente la declaró. Si la clase A declara un miembro virtual y la clase B deriva de A, y la clase C deriva de B, la clase C hereda el miembro virtual y tiene la opción de invalidarlo, independientemente de que la clase B declarara una invalidación para ese miembro. El siguiente fragmento de código muestra un ejemplo:

C#

public class A

{

public virtual void DoWork() { }

}

public class B : A

{

public override void DoWork() { }

}

Una clase derivada puede detener la herencia virtual declarando una invalidación como [sealed](https://msdn.microsoft.com/es-cl/library/88c54tsw.aspx). Para ello, es necesario colocar la palabra clave **sealed** antes de la palabra clave **override** en la declaración del miembro de la clase. El siguiente fragmento de código muestra un ejemplo:

C#

public class C : B

{

public sealed override void DoWork() { }

}

En el ejemplo anterior, el método DoWork ya no es virtual para ninguna clase que derive de C.Sigue siendo virtual para las instancias de C, aunque se conviertan al tipo B o al tipo A. Los métodos sellados se pueden reemplazar por clases derivadas usando la palabra clave **new**, como muestra el ejemplo siguiente:

C#

public class D : C

{

public new void DoWork() { }

}

En este caso, si se llama a DoWork en D usando una variable de tipo D, se llama al nuevo DoWork. Si se usa una variable de tipo C, B o A para acceder a una instancia de D, la llamada a DoWork seguirá las reglas de herencia virtual y enrutará dichas llamadas a la implementación de DoWork en la clase C.

[Acceder a miembros virtuales de clases base desde clases derivadas](javascript:void(0))

Una clase derivada que ha reemplazado o invalidado un método o propiedad puede seguir accediendo al método o propiedad en la clase base usando la siguiente palabra clave base.El siguiente fragmento de código muestra un ejemplo:

C#

public class Base

{

public virtual void DoWork() {/\*...\*/ }

}

public class Derived : Base

{

public override void DoWork()

{

//Perform Derived's work here

//...

// Call DoWork on base class

base.DoWork();

}

}