INSTITUTO KHIPU

Semestre III

Sesión 15

CARRERA PROFESIONAL

DESARROLLO DE SISTEMAS DE INFORMACIÓN

HERRAMIENTAS DE PROGRAMACION C#

Tema

HERENCIA DE CLASES

HERENCIA DE CLASES EN C#

La herencia es uno de los pilares fundamentales de la programación orientada a objetos (POO), junto con el encapsulamiento, la abstracción y el polimorfismo. En C#, la herencia permite que una clase pueda adquirir (heredar) las propiedades y métodos de otra clase, promoviendo la reutilización de código y la creación de ¿Qué es la Herencia?

La herencia en C# permite que una clase denominada clase derivada (o subclase) obtenga las características y comportamientos (campos y métodos) de otra clase llamada clase base (o superclase). De este modo, la clase derivada puede:

- Reutilizar los métodos y propiedades de la clase base.
- Sobrescribir los métodos de la clase base si es necesario.
- Extender la funcionalidad añadiendo nuevos métodos o propiedades.

Sintaxis de la Herencia

Para implementar la herencia en C#, se utiliza el símbolo de dos puntos :. La clase derivada hereda de la clase base de la siguiente forma:

```
class ClaseBase
{
    // Propiedades, métodos y campos de la clase base
}

class ClaseDerivada : ClaseBase
{
    // Propiedades, métodos y campos adicionales o modificados de la clase derivada
}
```

Ejemplo Básico de Herencia

Imaginemos una clase base llamada Animal, que tiene propiedades y métodos que son comunes a todos los animales. Luego, creamos una clase derivada Perro, que hereda de Animal y añade características específicas.

```
// Clase base
class Animal
{
    public string Nombre { get; set; }

    public void Comer()
    {
        Console.WriteLine($"{Nombre} está comiendo.");
    }
}

// Clase derivada
class Perro : Animal
{
    public void Ladrar()
    {
        Console.WriteLine($"{Nombre} está ladrando.");
    }
}

class Programa
{
    static void Main(string[] args)
    {
        Perro miPerro = new Perro();
        miPerro.Nombre = "Fido";
        miPerro.Comer();   // Método heredado de Animal
        miPerro.Ladrar();  // Método propio de Perro
    }
}
```

Jerarquía de Herencia

En el ejemplo anterior, Perro hereda de Animal, creando una relación de padre-hijo o superclase-subclase. Cualquier instancia de Perro puede acceder a los métodos y propiedades de Animal, pero Animal no tiene acceso a los miembros de Perro. Este es el principio de la herencia unidireccional en C#.

En C#, una clase solo puede heredar de una clase base, es decir, no admite herencia múltiple directa. Sin embargo, una clase derivada puede heredar de otra clase derivada, formando una cadena de herencia.

```
class Mamifero : Animal
{
    public void Amamantar()
    {
        Console.WriteLine($"{Nombre} está amamantando a su cría.");
    }
}

class Gato : Mamifero
{
    public void Maullar()
    {
        Console.WriteLine($"{Nombre} está maullando.");
    }
}

class Programa
{
    static void Main(string[] args)
    {
        Gato miGato = new Gato();
        miGato.Nombre = "Miau";
        miGato.Comer();         // Método heredado de Animal
        miGato.Amamantar();     // Método heredado de Mamifero
        miGato.Maullar();       // Método propio de Gato
    }
}
```

Modificadores de Acceso en Herencia

El acceso a los miembros de la clase base puede ser controlado mediante modificadores de acceso:

- **public:** Los miembros son accesibles desde cualquier parte, incluida la clase derivada.
- **private:** Los miembros solo son accesibles dentro de la clase base y no pueden ser heredados.
- **protected:** Los miembros son accesibles en la clase base y en cualquier clase derivada.
- **internal:** Los miembros son accesibles dentro del mismo ensamblado (mismo proyecto).

Veamos cómo los modificadores afectan la visibilidad de los miembros en la herencia:

```
class Animal
{
    private int edadPrivada;       // No accesible en clases derivadas
    protected int edadProtegida;   // Accesible en clases derivadas
    public string Nombre { get; set; }  // Accesible en todas partes
}

class Perro : Animal
{
    public void MostrarEdad()
    {
        // No se puede acceder a 'edadPrivada' aquí
        edadProtegida = 5;  // Se puede acceder a 'edadProtegida'
        Console.WriteLine($"El perro tiene {edadProtegida} años.");
    }
}
```

EJERCICIO

Crea una clase base Empleado con las propiedades Nombre, SueldoBase y el método CalcularSalario(), que devuelva el sueldo base. Luego, crea dos clases derivadas: EmpleadoFijo y EmpleadoTemporal. En EmpleadoFijo, añade una propiedad Bonificacion, y en EmpleadoTemporal, añade una propiedad HorasTrabajadas y una TarifaPorHora. Sobrescribe el método CalcularSalario() para que en EmpleadoFijo incluya la bonificación y en EmpleadoTemporal calcule el salario en función de las horas trabajadas y la tarifa

Resolución:

- **Clase empleado**

```
class Empleado
{
   // Propiedades comunes para todos los empleados
   public string Nombre { get; set; }
   public decimal SueldoBase { get; set; }

   // Método que devuelve el sueldo base (será sobrescrito)
   public virtual decimal CalcularSalario()
   {
      return SueldoBase;
   }

   // Constructor de la clase base
   public Empleado(string nombre, decimal sueldoBase)
   {
      Nombre = nombre;
      SueldoBase = sueldoBase;
   }
}
```

- **Clase Empleado fjo**

```
lass EmpleadoFijo : Empleado
{
   // Propiedad específica para los empleados fijos
   public decimal Bonificacion { get; set; }

   // Constructor de EmpleadoFijo
   public EmpleadoFijo(string nombre, decimal sueldoBase, decimal bonificacion)
      : base(nombre, sueldoBase)
   {
      Bonificacion = bonificacion;
   }

   // Sobrescribir el método CalcularSalario para incluir la bonificación
   public override decimal CalcularSalario()
   {
      return SueldoBase + Bonificacion;
   }
}
```

- **Clase Empleado Temporal**

```
class EmpleadoTemporal : Empleado
{
   // Propiedades específicas para los empleados temporales
   public int HorasTrabajadas { get; set; }
   public decimal TarifaPorHora { get; set; }

   // Constructor de EmpleadoTemporal
   public EmpleadoTemporal(string nombre, decimal sueldoBase, int horasTrabajadas,
decimal tarifaPorHora)
      : base(nombre, sueldoBase)
   {
      HorasTrabajadas = horasTrabajadas;
      TarifaPorHora = tarifaPorHora;
   }

   // Sobrescribir el método CalcularSalario para calcular según horas trabajadas
   public override decimal CalcularSalario()
   {
      return HorasTrabajadas * TarifaPorHora;
   }
}
```

- **Programa principal**

```
using System;

class Programa
{
   static void Main(string[] args)
   {
      // Crear un empleado fijo
      EmpleadoFijo empleadoFijo = new EmpleadoFijo("Juan Pérez", 1000m, 200m);
      Console.WriteLine($"{empleadoFijo.Nombre} tiene un salario de:
{empleadoFijo.CalcularSalario()}");

      // Crear un empleado temporal
      EmpleadoTemporal empleadoTemporal = new EmpleadoTemporal("María López",
0m, 160, 15m);
      Console.WriteLine($"{empleadoTemporal.Nombre} tiene un salario de:
{empleadoTemporal.CalcularSalario()}");
```

Ejercicios Propuestos

1. Crea una clase base Vehiculo con una propiedad Marca y un método Encender(). Luego, crea dos clases derivadas: Coche y Motocicleta, que sobrescriban el método Encender() para imprimir "El coche está encendido" y "La motocicleta está encendida".
2. Crea una clase base Persona con las propiedades Nombre y Edad, y un método Presentarse() que imprima "Hola, soy [Nombre] y tengo [Edad] años". Luego, crea una clase derivada Estudiante que herede de Persona y agregue la propiedad Carrera. Sobrescribe el método Presentarse() para que imprima "Hola, soy [Nombre], tengo [Edad] años y estudio [Carrera]".
3. Crea una clase base CuentaBancaria con las propiedades NumeroCuenta y Saldo, y los métodos Depositar() y Retirar(). Luego, crea dos clases derivadas:

CuentaCorriente y CuentaAhorro. La CuentaCorriente permite saldo negativo (hasta un límite) y la CuentaAhorro no lo permite. Implementa estos comportamientos en los métodos Retirar() de ambas clases.

INSTITUTO

KHIPU