POLIMORFISMO Y REUTILIZACIÓN

PARADIGMAS DE PROGRAMACIÓN II

Carlos Rojas Sánchez

Licenciatura en Informática

Universidad del Mar

Contenido

1. Concepto del polimorfismo
2. Sobrecarga
3. Sobreescritura
4. Variables Polimórficas
5. Tipos de Clases en C-Sharp
6. Creación y uso de paquetes / librerías en C-Sharp

Concepto del polimorfismo

Concepto del polimorfismo

El polimorfismo es el modo en que los lenguajes OO implementan el concepto de polisemia del mundo real: Un único nombre para muchos significados, según el contexto.

Concepto del polimorfismo

- Capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo.
- El polimorfismo nos permite programar de manera general en lugar de programar de manera específica.

Concepto del polimorfismo

Hay cuatro técnicas, cada una de las cuales permite una forma distinta de reutilización de software, que facilita a su vez el desarrollo rápido, la confianza y la facilidad de uso y mantenimiento.

- Sobrecarga
- Sobreescritura
- Variables polimórficas
- Genericidad

Tipos de polimorfismo

Sobrecarga (Overloading, Polimorfismo ad-hoc): un solo nombre de método y muchas implementaciones distintas. Las funciones sobrecargadas normalmente se distinguen en tiempo de compilación por tener distintos parámetros de entrada y/o salida.

Tipos de polimorfismo

Sobreescritura (Overriding, Polimorfismo de inclusión): Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia. En este caso la signatura es la misma (refinamiento o reemplazo del método del padre) pero los métodos se encuentran en dos clases distintas relacionadas mediante herencia.

Tipos de polimorfismo

Variables polimórficas (Polimorfismo de asignación): variable que se declara como de un tipo pero que referencia en realidad un valor de un tipo distinto. Cuando una variable polimórfica se utiliza como argumento, la función resultante se dice que exhibe un polimorfismo puro.

Tipos de polimorfismo

Genericidad (plantillas o templates): forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.

Tipos de polimorfismo

- Sobrecarga
  Factura::imprimir()
  Factura::imprimir(int numCopias)
  ListaCompra::imprimir()
- Sobreescritura
  Cuenta::abonarInteres()
  CuentaJoven::abonarInteres()
- Variables polimórficas
  Cuenta *pc=new CuentaJoven();
- Genericidad
  Lista<Cliente> Lista<Articulo> Lista<Alumno>
  Lista<Habitacion>

Sobrecarga

Sobrecarga (Overloading, polimorfismo ad-hoc)

- Un mismo nombre de mensaje asociado a varias implementaciones.
- La sobrecarga se realiza en tiempo de compilación (enlace estático) en función de la signatura completa del mensaje.

Sobrecarga (Overloading, polimorfismo ad-hoc)

Dos tipos de sobrecarga:

1. Basada en ámbito: Métodos con diferentes ámbitos de definición, independientemente de sus signaturas de tipo. Permitido en todos los lenguajes OO.
   - Un mismo método puede ser utilizado en dos o más clases.
   - P. ej. Sobrecarga de operadores como funciones miembro.

Sobrecarga (Overloading, polimorfismo ad-hoc)

Dos tipos de sobrecarga:

2. Basada en signatura: Métodos con diferentes signaturas de tipo en el mismo ámbito de definición. No permitido en todos los lenguajes OO.
   - P. ej. Cualquier conjunto de funciones no miembro (en el ámbito de definición global) que comparten nombre.
   - Dos o más métodos en la misma clase pueden tener el mismo nombre siempre que tengan distinta signatura de tipos.

Sobrecarga basada en ámbito

- Distintos ámbitos implican que el mismo nombre de método puede aparecer en ellos sin ambigüedad ni pérdida de precisión.
- La sobrecarga por ámbito no requiere que las funciones asociadas con un nombre sobrecargado tengan ninguna similitud semántica, ni la misma signatura de tipo.

Sobrecarga basada en signaturas de tipo

- Métodos en el mismo ámbito pueden compartir el mismo nombre siempre que difieran en número, orden y, en lenguajes con tipado estático, el tipo de los argumentos que requieren.
- No todos los lenguajes OO permiten la sobrecarga:
  - Permiten sobrecarga de métodos y operadores: C++
  - Permiten sobrecarga de métodos pero no de operadores: Java, Python, Perl
  - Permiten sobrecarga de operadores pero no de métodos: Eiffel

Polimorfismo en jerarquías de herencia

Sobrecarga en jerarquías de herencia

- Métodos con el mismo nombre, la misma signatura de tipo y enlace estático: Shadowing (refinamiento/reemplazo): las signaturas de tipo son las misma en clases padre e hijas, pero el método a invocar se decide en tiempo de compilación
- Métodos con el mismo nombre y distinta signatura de tipo y enlace estático: Redefinición: clase hija define un método con el mismo nombre que el padre pero con distinta signatura de tipos. Modelo MERGE: SOBRECARGA y Modelo JERÁRQUICO: NO HAY SOBRECARGA

Sobreescritura

Polimorfismo en jerarquías de herencia

Sobreescritura (Overriding, polimorfismo de inclusión)

- Decimos que un método en una clase derivada sobreescribe un método en la clase base si los dos métodos tienen el mismo nombre, la misma signatura de tipos y enlace dinámico.
- La sobreescritura es importante cuando se combina con el principio de sustitución.
- En algunos lenguajes (Java, Smalltalk) la simple existencia de un método con el mismo nombre y signatura de tipos en clase base y derivada indica sobreescritura.

Polimorfismo: Sobreescritura, Shadowing y Redefinición

- Sobreescritura: la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza con la llamada en tiempo de ejecución.
- Shadowing: la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza en tiempo de compilación (en función del tipo de la variable receptora).
- Redefinición: La clase derivada define un método con el mismo nombre que en la calse base y con distinta signatura de tipos.

Variables Polimórficas

Variables Polimórficas (Polimorfismo de asignación)

- Una variable polimórfica es aquélla que puede referenciar más de un tipo de objeto. Puede mantener valores de distintos tipos en distintos momentos de ejecución del programa.
- En un lenguaje débilmente tipado todas las variables son potencialmente polimórficas.
- En un lenguaje fuertemente tipado la variable polimórfica es la materialización del principio de sustitución.

Polimorfismo

Ventajas

- El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código original.
- Permite programar a nivel de clase base utilizando objetos de clases derivadas (posiblemente no definidas aún): Técnica base de las librerías/frameworks.

Polimorfismo

- Estableces clase base.
- Defines nuevas variables y funciones.
- Ensamblas con el código objeto que tenías.
- Los métodos de la clase base pueden ser reutilizados con variables y parámetros de la clase derivada.

Tipos de Clases en C-Sharp

Clase estándar

Una clase común que puede ser instanciada.

```
public class Persona
{
    public string Nombre { get; set; }

    public void Saludar()
    {
        Console.WriteLine($"Hola, soy {Nombre}");
    }
}
```

Clase estática

No puede ser instanciada. Solo contiene miembros estáticos.

```
public static class Calculadora
{
    public static int Sumar(int a, int b)
    {
        return a + b;
    }
}
```

Clase parcial

Divide la implementación de una clase en múltiples archivos.

```
// Archivo 1
public partial class Producto
{
    public string Nombre { get; set; }
}

// Archivo 2
public partial class Producto
{
    public decimal Precio { get; set; }
}
```

Clase abstracta

Sirve como base. No puede ser instanciada.

```
public abstract class Animal
{
    public abstract void HacerSonido();
}
```

Clase sellada

No puede ser heredada.

```
public sealed class ConexionBD
{
    public void Conectar()
    {
        // Código de conexión
    }
}
```

Clase genérica

Permite trabajar con distintos tipos.

```
public class Caja<T>
{
    public T Contenido { get; set; }
}
```

Clase anidada

Definida dentro de otra clase.

```
public class Externa
{
    public class Interna
    {
        public void Mostrar()
        {
            Console.WriteLine("Clase interna");
        }
    }
}
```

¿Qué es una clase abstracta?

- Una clase abstracta no puede ser instanciada directamente.
- Puede contener:
  - Métodos abstractos (sin implementación).
  - Métodos concretos (con implementación).
- Se utiliza como una plantilla base para otras clases.

Componentes del ejemplo

- Clase base abstracta: `Animal`
  - Contiene métodos abstractos: `HacerSonido()`
  - Métodos concretos: `Comer()`, `Dormir()`
- Clases hijas: `Perro`, `Gato`, `Vaca`
  - Implementan `HacerSonido()`
  - Algunas sobreescriben `Dormir()`

Ejemplo de uso

- Se crea una lista de objetos tipo `Animal`, utilizando polimorfismo.
- Cada animal ejecuta su propia versión de `HacerSonido()` y `Dormir()`.
- Se demuestra reutilización de código y flexibilidad de diseño.

Resumen Clases Abstractas

- Las clases abstractas son útiles cuando quieres definir una estructura general, pero permitir que las clases hijas personalicen ciertos comportamientos.
- Fomentan el uso de polimorfismo, encapsulamiento y diseño orientado a objetos limpio.

¿Qué es una interfaz?

- Una interfaz en C# define un contrato que una clase debe cumplir.
- Contiene solo la definición de métodos y propiedades, sin implementación.
- Una clase puede implementar múltiples interfaces.

Componentes del ejemplo

- **IVehiculo:** define comportamiento común de vehículos.
- **IElectrico:** define comportamiento específico de dispositivos eléctricos.
- **Automovil:** implementa solo `IVehiculo`.
- **ScooterElectrico:** implementa `IVehiculo` y `IElectrico`.
- **PanelSolar:** implementa solo `IElectrico`.

Uso del polimorfismo

- Se crea una lista de `IVehiculo` para manejar diferentes tipos de vehículos.
- Cada objeto responde de forma diferente a los mismos métodos.
- También se usa `IElectrico` para abstraer dispositivos eléctricos.

Resumen Interfaz

- Las interfaces permiten definir comportamientos que múltiples clases pueden compartir.
- Fomentan la reutilización, modularidad y el polimorfismo.
- Son esenciales para el diseño limpio y la programación orientada a interfaces.

Creación y uso de paquetes / librerías en C-Sharp

Creación y uso de paquetes / librerías

En C#, crear y usar paquetes o librerías se hace principalmente con librerías de clases (Class Library) que puedes compilar como .dll y luego reutilizar en otros proyectos. Esto es ideal para separar lógica de negocio, utilidades, controladores, etc.

¿Qué es una librería en C#?

- Una librería (o biblioteca) es un conjunto de clases y métodos compilados en un archivo `.dll`.
- Se puede reutilizar en múltiples proyectos.
- Se crea con un proyecto de tipo `Class Library`.

Paso 1: Crear la librería

- En Visual Studio: Nuevo Proyecto → `Class Library (.NET)`.
- Agrega una clase, por ejemplo: `Calculadora.cs`.
- Escribe métodos reutilizables:

Paso 1: Crear la librería

- En Visual Studio: Nuevo Proyecto → `Class Library (.NET)`.
- Agrega una clase, por ejemplo: `Calculadora.cs`.
- Escribe métodos reutilizables:

Ejemplo

```
public int Sumar(int a, int b) => a + b;
```

Paso 2: Compilar la librería

- Compila la librería con `Ctrl + Shift + B`.
- Se genera un archivo `.dll` en la carpeta `bin/Debug/netX.X/`.
- Este archivo puede referenciarse desde otros proyectos.

Paso 3: Crear el proyecto consumidor

- Nuevo Proyecto → `Console App (.NET)`.
- Agrega una referencia a la librería:
  - Si están en la misma solución: **Add → Project Reference**.
  - Si tienes el .dll: **Add → Reference → Browse...**.
- Usa `using MiLibreria;` en el código.

Paso 4: Usar la clase desde el programa principal

- Puedes crear instancias de las clases de la librería:

Ejemplo

```
Calculadora calc = new Calculadora();
Console.WriteLine(calc.Sumar(5, 3));
```

Ventajas de usar librerías

- Reutilización de código entre proyectos.
- Organización y separación de responsabilidades.
- Facilita el mantenimiento y pruebas.
- Permite empaquetar lógica como paquetes NuGet.