Guión para Video Explicativo: Sistema de Cálculo de Áreas y Perímetros en C#

Introducción

[Pantalla de Título: "Sistema de Figuras Geométricas - Herencia y Polimorfismo en C#"]

Narrador: ¡Hola! Bienvenidos a este video donde exploraremos la Programación Orientada a Objetos a través de un sistema que calcula áreas y perímetros de diferentes figuras geométricas.

[Mostrar collage de figuras geométricas: círculo, cuadrado, triángulo, etc.]

Narrador: Nuestro sistema trabajará con ocho figuras diferentes:

- **Básicas:** Círculo, Cuadrado y Rectángulo
- **Intermedias:** Triángulo y Paralelogramo
- **Complejas:** Rombo, Cometa y Trapecio

Cada figura calculará su área y perímetro usando sus propias fórmulas matemáticas, pero todas compartirán una estructura común gracias a la **herencia**.

[Mostrar ejemplo de salida: "Circle => Area .....: 78.53982 Perimeter: 31.41593"]

Narrador: El objetivo es crear un sistema que muestre los resultados en un formato estandarizado, aplicando principios sólidos de la Programación Orientada a Objetos.

Desarrollo Parte 1: La Clase Base Abstracta

[Mostrar código de GeometricFigure]

Narrador: El corazón de nuestro sistema es la **clase abstracta GeometricFigure**:

Narrador: Esta clase funciona como un **contrato** que define qué debe hacer toda figura geométrica.

Elementos clave:

- **Name**: almacena el nombre de la figura (Circle, Square, etc.)
- **Constructor protegido**: permite que solo las clases hijas puedan usar este constructor

- **Métodos abstractos**: `GetArea()` y `GetPerimeter()` deben ser implementados por cada figura
- **ToString() sobrescrito**: proporciona el formato de salida estandarizado

Narrador: La palabra **abstract** significa que esta clase sirve como molde, pero no podemos crear objetos directamente de ella. Es como un plano arquitectónico: define la estructura, pero necesitamos construir las casas específicas.

Desarrollo Parte 2: Figuras Básicas

[Mostrar código de Circle]

Narrador: Comencemos con la clase **Circle**, que representa un círculo:

Narrador: Aquí vemos varios conceptos importantes:

Encapsulación:

- **Campo privado** `radius` : solo la clase puede modificarlo directamente
- **Propiedad pública** `Radius` : controla el acceso con validación

Validación:

- El **setter** verifica que el radio sea positivo
- Lanza una **excepción** si el valor es inválido

Herencia:

- `: base("Circle")` llama al constructor de la clase padre
- `override` implementa los métodos abstractos con las fórmulas específicas del círculo

Fórmulas matemáticas:

- **Área**: $\pi \times r^2$
- **Perímetro**: $2 \times \pi \times r$

[Mostrar código de Square y Rectangle brevemente]

Narrador: Las clases **Square** y **Rectangle** siguen el mismo patrón: validación en las propiedades, herencia de la clase base, e implementación de las fórmulas específicas. El cuadrado usa `lado²` para el área, mientras que el rectángulo usa `ancho × alto` .

Desarrollo Parte 3: Figuras con Validaciones Complejas

[Mostrar código de Triangle]

Narrador: La clase **Triangle** introduce validaciones más sofisticadas:

Narrador: Esta clase demuestra **validación de reglas matemáticas**:

Validación geométrica:

- Verifica la **desigualdad triangular**: la suma de dos lados debe ser mayor al tercero
- Esta es una regla fundamental: no todos los conjuntos de tres números pueden formar un triángulo

Fórmula de Herón:

- Calcula el **semiperímetro** (s = perímetro ÷ 2)
- Aplica la fórmula: √[s × (s-a) × (s-b) × (s-c)]
- Es más compleja que área = base × altura, pero funciona con cualquier triángulo

[Mostrar código de Parallelogram]

Narrador: El **Parallelogram** maneja múltiples dimensiones: base, lado y altura. Su área se calcula como `base × altura`, no como `base × lado`, porque la altura es la distancia perpendicular entre las bases paralelas.

Desarrollo Parte 4: Figuras con Múltiples Propiedades (7:00 - 9:00)

[Mostrar código de Rhombus]

Narrador: Las figuras más complejas manejan múltiples propiedades. El **Rhombus** (rombo) necesita tanto el lado como las diagonales:

Narrador: Aquí vemos un patrón interesante:

- **Área**: usa las diagonales → (d1 × d2) ÷ 2
- **Perímetro**: usa los lados → 4 × lado

[Mostrar código de Kite y Trapeze brevemente]

Narrador:

- **Kite (Cometa)**: similar al rombo, pero con lados diferentes → perímetro = 2 × (lado1 + lado2)
- **Trapeze (Trapecio)**: la figura más compleja, con cinco propiedades: dos bases, altura y dos lados laterales

Patrón común:

- Todas mantienen **validaciones** en sus propiedades
- Todas **heredan** de GeometricFigure
- Cada una implementa sus **fórmulas específicas**

Desarrollo Parte 5: El Programa Principal y Polimorfismo (9:00 - 10:30)

[Mostrar código del Main]

Narrador: El programa principal demuestra el poder del **polimorfismo**:

Narrador: Aquí ocurre la **magia del polimorfismo**:

Lista polimórfica:

- `List<GeometricFigure>` puede contener objetos de cualquier clase hija
- Circle, Square, Triangle... todos son GeometricFigure

Polimorfismo en acción:

- `figure.ToString()` llama automáticamente al método de la clase específica
- `GetArea()` y `GetPerimeter()` se ejecutan según el tipo real del objeto
- El **mismo código** maneja diferentes tipos de figuras

Ventajas:

- **Extensibilidad**: agregar una nueva figura no requiere cambiar el bucle
- **Mantenibilidad**: un solo método maneja todas las figuras
- **Reutilización**: el mismo código sirve para cualquier figura geométrica

---

Demostración Práctica

[Mostrar ejecución del programa]

Narrador: Veamos el sistema en funcionamiento:

Circle (radio = 5):

- Área: $\pi \times 5^2 = 78.53982$
- Perímetro: $2 \times \pi \times 5 = 31.41593$

- Salida: "Circle => Area .....: 78.53982 Perimeter: 31.41593"

Square (lado = 10):

- Área: 10^2 = 100.00000
- Perímetro: 4 × 10 = 40.00000
- Salida: "Square => Area .....: 100.00000 Perimeter: 40.00000"

Triangle (lados = 25.33, 29.66, 30):

- Validación: Forma un triángulo válido
- Área: Fórmula de Herón = 280.11400
- Perímetro: 25.33 + 29.66 + 30 = 84.99000

Formato estandarizado:

- Todas las figuras usan el **mismo formato de salida**
- **5 decimales** de precisión
- **Consistencia visual** sin importar la complejidad de la figura

[Mostrar manejo de errores]

Narrador: El sistema también maneja errores elegantemente. Si intentamos crear un triángulo con lados inválidos o un círculo con radio negativo, el programa captura la excepción y muestra un mensaje claro.

Conclusión

[Mostrar diagrama de jerarquía de clases]

Narrador: Este proyecto demuestra los **cuatro pilares** de la Programación Orientada a Objetos:

1. Abstracción:

- GeometricFigure define la **estructura común** sin implementación específica
- Oculta la complejidad y expone solo lo necesario

2. Herencia:

- Todas las figuras **heredan** propiedades y métodos comunes
- **Reutilización de código**: no repetimos Name ni ToString()

3. Encapsulación:

- **Propiedades con validación** protegen la integridad de los datos
- **Campos privados** controlan el acceso interno

4. Polimorfismo:

- **Una lista puede contener diferentes tipos** de figuras
- **El mismo código maneja comportamientos específicos** de cada figura

Beneficios del diseño:

- **Mantenible:** cambios en una figura no afectan las otras
- **Extensible:** agregar nuevas figuras es simple y directo
- **Confiable:** validaciones previenen errores en tiempo de ejecución
- **Legible:** código organizado y fácil de entender

Aplicaciones prácticas:

- Software de **diseño gráfico**
- Calculadoras de **ingeniería y arquitectura**
- **Videojuegos** con detección de colisiones
- **Sistemas educativos** de geometría

Narrador: Este sistema ejemplifica cómo la Programación Orientada a Objetos nos permite modelar conceptos del mundo real de manera elegante y eficiente. La matemática se combina con la programación para crear soluciones robustas y mantenibles.

¡Gracias por acompañarnos en este recorrido por la geometría y la programación!

[Pantalla final con resumen de figuras y sus fórmulas]