

Desarrollo de Software I

Material Completo del Curso

Total de Unidades: 8

Índice de Contenidos

Unidad 1: Desarrollo de Software I 1

Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
Historia y Aplicaciones de C#	26
Aplicaciones de C# en la Vida Real	26
Instalación y Configuración de Visual Studio	26

Recursos Adicionales - Unidad 1 26

Unidad 2: Desarrollo de Software I 51

Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
¿Qué es una Variable?	76
Tipos de Datos en C#	76
Tipos de Datos (Continuación)	76

Recursos Adicionales - Unidad 2 76

Unidad 3: Desarrollo de Software I 101

Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
Estructura Condicional: if	126

Estructura if-else	126
Estructura if-else if-else	126
Recursos Adicionales - Unidad 3	126
Unidad 4: Desarrollo de Software I	151
Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
¿Qué es un Arreglo?	176
Declaración e Inicialización	176
Acceso a Elementos	176
Recursos Adicionales - Unidad 4	176
Unidad 5: Desarrollo de Software I	201
Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
¿Qué es una Función?	226
Declaración de Funciones	226
Funciones con Parámetros	226
Recursos Adicionales - Unidad 5	226
Unidad 6: Desarrollo de Software I	251
Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326

¿Qué es POO?	276
Clases y Objetos	276
Propiedades	276
Recursos Adicionales - Unidad 6	276
Unidad 7: Desarrollo de Software I	301
Competencias y Resultados de Aprendizaje	326
Agenda de la Unidad	326
Encapsulación	326
Modificadores de Acceso	326
Herencia	326
Recursos Adicionales - Unidad 7	326
Unidad 8: Desarrollo de Software I	351
Objetivo del Proyecto Final	376
Ejemplos de Proyectos	376
Más Ejemplos de Proyectos	376
Estructura del Proyecto	376
Criterios de Evaluación	376
Recursos Adicionales - Unidad 8	376

Desarrollo de Software I

Unidad 01: Introducción al Lenguaje C# y Configuración del Entorno

Notas del Presentador:

Bienvenidos a la primera unidad del curso. Esta unidad sienta las bases para todo el curso. Es importante crear un ambiente motivador y explicar la importancia de C# en el desarrollo de software moderno.

Competencias y Resultados de Aprendizaje

Competencia Fundamental:

CF4: Aplicar de forma intensiva y crítica las tecnologías de la información y la comunicación en las actividades profesionales.

Resultados de Aprendizaje:

- Comprender la historia y evolución del lenguaje C#
- Instalar y configurar correctamente Visual Studio
- Crear y ejecutar programas básicos en C#
- Navegar eficientemente por el entorno de desarrollo
- Identificar la estructura básica de un programa en C#

Notas del Presentador:

Estos resultados están alineados con el programa académico. Asegúrate de que los estudiantes comprendan que no solo aprenderán sintaxis, sino también buenas prácticas desde el inicio.

Agenda de la Unidad

1. Historia y aplicaciones de C#
2. Instalación y configuración de Visual Studio
3. Estructura básica de un programa en C#
4. Primer programa: "Hola Mundo"
5. Navegación por Visual Studio
6. Prácticas guiadas
7. Evaluación práctica
8. Mini-Quiz

Notas del Presentador:

Duración estimada: 2 horas. La primera parte será teórica (45 min), luego práctica guiada (45 min), y finalmente evaluación (30 min).

Historia y Aplicaciones de C#

Origen y Evolución

- **2000:** C# fue creado por Microsoft como parte de la plataforma .NET
- **Diseñador principal:** Anders Hejlsberg (también creador de Turbo Pascal y Delphi)
- **Inspiración:** Combina lo mejor de C++, Java y Visual Basic
- **Versión actual:** C# 12.0 (2023) - Lenguaje moderno y en constante evolución

¿Por qué C#? Es un lenguaje multiparadigma, fuertemente tipado, con excelente soporte para programación orientada a objetos y funcional.

Notas del Presentador:

Mencionar que C# es uno de los lenguajes más demandados en el mercado laboral, especialmente en desarrollo empresarial y aplicaciones Windows.

Aplicaciones de C# en la Vida Real

Áreas de Aplicación:

- **Desarrollo Web:** ASP.NET Core para aplicaciones web modernas
- **Aplicaciones de Escritorio:** Windows Forms, WPF, .NET MAUI
- **Desarrollo Móvil:** Xamarin/.NET MAUI para iOS, Android y Windows
- **Videojuegos:** Unity Engine utiliza C# como lenguaje principal
- **Servicios Backend:** APIs REST, microservicios, servicios en la nube
- **IoT:** Desarrollo para dispositivos embebidos

Ejemplo Real: Stack Overflow, Visual Studio, muchos servicios de Azure, y juegos como Pokémon GO fueron desarrollados usando C#.

Instalación y Configuración de Visual Studio

Requisitos del Sistema:

- Windows 10/11 (64-bit) o macOS/Linux
- 4 GB RAM mínimo (8 GB recomendado)
- 20 GB espacio en disco
- Conexión a Internet para descarga

Pasos de Instalación:

1. Descargar Visual Studio Community (gratuito) desde visualstudio.microsoft.com
2. Ejecutar el instalador
3. Seleccionar la carga de trabajo "Desarrollo de escritorio de .NET"
4. Incluir componentes: .NET SDK, C# compiler, Visual Studio Core Editor
5. Instalar y reiniciar

Notas del Presentador:

Si hay estudiantes con problemas de instalación, recomendar Visual Studio Code como alternativa ligera, aunque Visual Studio Community es la mejor opción para principiantes.

Configuración Inicial de Visual Studio

Primera Configuración:

1. **Seleccionar tema:** Oscuro/Claro según preferencia
2. **Configurar fuente:** Consolas o Fira Code (recomendado para código)
3. **Habilitar números de línea:** Herramientas → Opciones → Editor de texto
4. **Configurar formato automático:** Formatear al guardar

Componentes Importantes:

- **Solution Explorer:** Navegación de archivos del proyecto
- **Editor de Código:** Área principal de escritura
- **Output Window:** Mensajes de compilación y errores
- **Error List:** Lista de errores y advertencias

Estructura Básica de un Programa en C#

Componentes Fundamentales:

```
using System;

namespace MiPrimerPrograma
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hola Mundo");
        }
    }
}
```

Explicación:

- `using System;` - Importa la biblioteca estándar
- `namespace` - Organiza el código en espacios lógicos
- `class Program` - Define una clase (POO)
- `Main` - Punto de entrada del programa
- `Console.WriteLine` - Muestra texto en la consola

En la Vida Real: Estructura de Proyectos

Escenario: En un proyecto real de software, el código se organiza en múltiples archivos y carpetas:

```
MiProyecto/
└── Models/          # Clases de datos
└── Views/           # Interfaces de usuario
└── Controllers/     # Lógica de negocio
└── Services/         # Servicios externos
└── Program.cs        # Punto de entrada
```

Buenas Prácticas:

- Usar nombres descriptivos para namespaces
- Un archivo por clase
- Organizar en carpetas lógicas
- Comentar código complejo

Errores Comunes al Iniciar

1. Olvidar punto y coma (;

```
Console.WriteLine("Hola") // ✗ Error  
Console.WriteLine("Hola"); // ✓ Correcto
```

2. Mayúsculas y minúsculas incorrectas

```
console.WriteLine("Hola"); // ✗ Error (C# es case-sensitive)  
Console.WriteLine("Hola"); // ✓ Correcto
```

3. Llaves no balanceadas

```
class Program { // ✗ Falta cerrar la llave  
    static void Main() {  
        Console.WriteLine("Hola");  
    }  
}
```

Buenas Prácticas desde el Inicio

- **Nombres descriptivos:** Usar nombres claros para variables y clases
- **Indentación consistente:** Usar 4 espacios o tabs
- **Comentarios útiles:** Explicar el "por qué", no el "qué"
- **Formato consistente:** Seguir convenciones de C#
- **Versionar código:** Usar Git desde el inicio

Convención de Nombres en C#:

- Clases: PascalCase (ej: `MiClase`)
- Métodos: PascalCase (ej: `CalcularPromedio`)
- Variables: camelCase (ej: `miVariable`)
- Constantes: UPPER_CASE (ej: `PI`)

Práctica Guiada #1: Crear "Hola Mundo"

Objetivo:

Crear y ejecutar tu primer programa en C#

Instrucciones Paso a Paso:

1. Abrir Visual Studio
2. Crear nuevo proyecto: Archivo → Nuevo → Proyecto
3. Seleccionar "Aplicación de consola (.NET)"
4. Nombre del proyecto: "HolaMundo"
5. Ubicación: Elegir carpeta de trabajo
6. Hacer clic en "Crear"
7. Reemplazar el código en Program.cs con:

```
using System;

namespace HolaMundo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("¡Hola Mundo desde C#!");
            Console.WriteLine("Mi nombre es: [Tu Nombre]");
            Console.ReadKey();
        }
    }
}
```

Ejecutar: Presionar F5 o clic en "Iniciar"

Solución Práctica Guiada #1

Resultado Esperado:

```
¡Hola Mundo desde C#!  
Mi nombre es: [Tu Nombre]
```

Explicación del Código:

- `Console.WriteLine()` - Escribe texto y salta línea
- `Console.ReadKey()` - Espera a que presiones una tecla antes de cerrar
- Sin `ReadKey()`, la ventana se cerraría inmediatamente

Variaciones para Probar:

- Cambiar el mensaje
- Agregar más líneas de texto
- Usar `Console.Write()` en lugar de `WriteLine()`

Práctica Guiada #2: Explorar Visual Studio

Objetivo:

Familiarizarse con las herramientas principales de Visual Studio

Actividades:

1. Solution Explorer:

- Ubicar el panel Solution Explorer (lado derecho)
- Expandir el proyecto "HolaMundo"
- Identificar archivos: Program.cs, .csproj

2. Editor de Código:

- Hacer clic en Program.cs
- Observar colores de sintaxis
- Probar autocompletado (IntelliSense): escribir "Conso" y presionar Tab

3. Ventana de Salida:

- Ver → Otras ventanas → Salida
- Ejecutar el programa (F5)

- Observar mensajes de compilación

Solución Práctica Guiada #2

Atajos de Teclado Útiles:

- **F5:** Iniciar depuración
- **Ctrl+F5:** Ejecutar sin depuración
- **Ctrl+K, Ctrl+D:** Formatear documento
- **Ctrl+Space:** Mostrar IntelliSense
- **Ctrl+{/}:** Comentar/descomentar línea
- **F12:** Ir a definición

Panel de Propiedades:

Seleccionar el proyecto en Solution Explorer y observar las propiedades en el panel inferior. Aquí puedes cambiar:

- Versión de .NET
- Nombre del ensamblado
- Configuración de compilación

Consejo: Personaliza Visual Studio según tus preferencias. Ve a Herramientas → Opciones para explorar todas las configuraciones disponibles.

Práctica Evaluada

Enunciado:

Crear un programa de consola que muestre información personal y académica. El programa debe:

1. Mostrar tu nombre completo
2. Mostrar tu carrera universitaria
3. Mostrar el nombre de la asignatura
4. Mostrar un mensaje de motivación sobre aprender C#
5. Usar comentarios explicativos en el código
6. Usar nombres descriptivos para el namespace y la clase

Criterios de Evaluación:

- **Funcionalidad (40%):** El programa compila y ejecuta correctamente
- **Código (30%):** Uso correcto de la estructura básica de C#
- **Comentarios (15%):** Comentarios claros y útiles
- **Nombres (15%):** Nombres descriptivos siguiendo convenciones

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todo funciona perfectamente	Funciona con pequeños errores	Funciona parcialmente	No funciona
Estructura	Estructura perfecta	Estructura correcta	Estructura básica	Estructura incorrecta
Comentarios	Comentarios excelentes	Comentarios adecuados	Pocos comentarios	Sin comentarios
Nombres	Nombres muy descriptivos	Nombres descriptivos	Nombres básicos	Nombres poco claros

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿En qué año fue creado C#?

- a) 1995
- b) 2000
- c) 2005
- d) 2010

2. ¿Cuál es el punto de entrada de un programa en C#?

- a) Start()
- b) Main()
- c) Begin()
- d) Init()

3. ¿Qué comando muestra texto en la consola?

- a) System.Print()
- b) Console.Write()
- c) Output.Show()
- d) Display.Text()

Mini-Quiz (Continuación)

4. ¿C# es case-sensitive (sensible a mayúsculas/minúsculas)?

- a) Verdadero
- b) Falso

5. ¿Qué IDE es recomendado para desarrollo en C#?

- a) Eclipse
- b) Visual Studio
- c) NetBeans
- d) PyCharm

6. ¿Qué palabra clave se usa para importar bibliotecas en C#?

- a) import
- b) include
- c) using
- d) require

Mini-Quiz (Continuación)

7. ¿Cuál es la extensión de archivo para código fuente en C#?

- a) .cpp
- b) .java
- c) .cs
- d) .csharp

8. ¿Qué atajo de teclado ejecuta un programa en Visual Studio?

- a) F1
- b) F5
- c) F9
- d) F12

9. ¿El método Main() debe ser estático (static)?

- a) Verdadero
- b) Falso

10. ¿Qué estructura organiza el código en espacios lógicos en C#?

- a) package
- b) module
- c) namespace
- d) library

Respuestas del Mini-Quiz

- 1. b) 2000** - C# fue creado en el año 2000 por Microsoft.
- 2. b) Main()** - El método Main() es el punto de entrada de cualquier programa en C#.
- 3. b) Console.WriteLine()** - Console.WriteLine() muestra texto en la consola.
- 4. a) Verdadero** - C# distingue entre mayúsculas y minúsculas.
- 5. b) Visual Studio** - Visual Studio es el IDE oficial y más recomendado para C#.
- 6. c) using** - La palabra clave "using" se usa para importar namespaces.
- 7. c) .cs** - Los archivos de código fuente en C# tienen extensión .cs
- 8. b) F5** - F5 inicia la depuración y ejecuta el programa.
- 9. a) Verdadero** - El método Main() debe ser estático para ser el punto de entrada.
- 10. c) namespace** - Los namespaces organizan el código en espacios lógicos.

Resumen de la Unidad

- ✓ C# es un lenguaje moderno creado en 2000, ampliamente usado en desarrollo empresarial
- ✓ Visual Studio es el IDE recomendado para desarrollo en C#
- ✓ La estructura básica incluye: using, namespace, class y método Main()
- ✓ Console.WriteLine() muestra texto en la consola
- ✓ C# es case-sensitive y requiere punto y coma al final de las instrucciones
- ✓ Los nombres siguen convenciones: PascalCase para clases, camelCase para variables
- ✓ Es importante usar comentarios y mantener código organizado desde el inicio
- ✓ Visual Studio ofrece herramientas poderosas como IntelliSense y depuración

Próxima Unidad: Variables, Tipos de Datos y Operadores - Aprenderemos a trabajar con datos y realizar operaciones básicas.

Tarea para Casa

Actividad:

Investigar y documentar los tipos de datos básicos en C# y los operadores principales.

Entregables:

1. **Documento Word/PDF (2-3 páginas)** que incluya:

- Lista de tipos de datos primitivos en C# (int, double, string, bool, char, etc.)
- Rango de valores de cada tipo
- Operadores aritméticos (+, -, *, /, %)
- Operadores de comparación (==, !=, <, >, <=, >=)
- Operadores lógicos (&&, ||, !)
- Ejemplos de código para cada operador

2. **Programa en C#** que demuestre el uso de al menos 5 tipos de datos diferentes

Fecha de Entrega:

Una semana después de esta clase

Criterios de Evaluación:

- Completitud de la información (40%)
- Claridad y organización (30%)

- Ejemplos de código funcionales (30%)

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Enlaces a documentación oficial de C#
- Videos tutoriales sobre Visual Studio
- Guías de instalación y configuración
- Ejercicios adicionales
- Recursos de aprendizaje

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 01

Introducción al Lenguaje C# - Enlaces, videos y material complementario



Documentación Oficial de Microsoft

[Documentación Oficial de C# Oficial](#)

Documentación completa y actualizada de C# directamente de Microsoft. Incluye guías, tutoriales, ejemplos y referencias de API.

[Tour de C# Tutorial](#)

Recorrido completo por las características principales del lenguaje C#. Ideal para entender los fundamentos.

[Estructura de Programas en C# Guía](#)

Explicación detallada sobre la estructura básica de un programa en C#, namespaces, clases y métodos.



Videos Educativos

[C# Tutorial for Beginners Video](#)

Serie completa de tutoriales de C# para principiantes. Cubre desde la instalación hasta conceptos avanzados.

[Instalación de Visual Studio 2022 Tutorial](#)

Guía paso a paso sobre cómo instalar y configurar Visual Studio 2022 para desarrollo en C#.

[Tu Primer Programa en C# Tutorial](#)

Tutorial práctico para crear tu primer programa "Hola Mundo" en C# usando Visual Studio.



Herramientas Recomendadas

[Visual Studio 2022 Gratis](#)

IDE principal para desarrollo en C#. Incluye IntelliSense, depuración avanzada y herramientas de productividad.

[Visual Studio Code Gratis](#)

Editor de código ligero con excelente soporte para C# mediante extensiones. Ideal para principiantes.

[.NET SDK Requisito](#)

SDK necesario para compilar y ejecutar aplicaciones C#. Incluye el compilador y runtime de .NET.



Libros y Tutoriales

[Microsoft Learn - C# Fundamentals Curso Gratis](#)

Curso interactivo gratuito de Microsoft que cubre los fundamentos de C# con ejercicios prácticos.

[W3Schools - Tutorial de C# Tutorial](#)

Tutorial interactivo con ejemplos de código y ejercicios prácticos para aprender C#.



Artículos Especializados

[Blog de .NET - Microsoft Blog](#)

Blog oficial de .NET con artículos sobre nuevas características, mejores prácticas y actualizaciones de C#.

[Stack Overflow - C# Comunidad](#)

Comunidad activa de desarrolladores C# donde puedes hacer preguntas y encontrar soluciones a problemas comunes.



Ejemplos de Código y Repositorios

Especificación de C# GitHub

Repositorio oficial con la especificación del lenguaje C# y propuestas de nuevas características.

Ejemplos de .NET GitHub

Repositorio con ejemplos de código en C# y .NET para diferentes escenarios y casos de uso.



Consejo de Estudio

Para esta unidad introductoria, te recomendamos: 1) Instalar Visual Studio siguiendo la guía oficial, 2) Crear tu primer programa "Hola Mundo" y experimentar con él, 3) Explorar la interfaz de Visual Studio para familiarizarte con las herramientas, 4) Leer la documentación oficial sobre la estructura básica de programas en C#. La práctica es esencial desde el inicio.

Desarrollo de Software I

Unidad 02: Variables, Tipos de Datos y
Operadores

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Declarar y utilizar variables correctamente en C#
- Identificar y usar los tipos de datos primitivos
- Aplicar operadores aritméticos, lógicos y de comparación
- Realizar conversiones entre tipos de datos
- Crear programas que procesen datos numéricos y de texto

Agenda de la Unidad

1. Concepto de variables
2. Tipos de datos en C#
3. Declaración e inicialización
4. Operadores aritméticos
5. Operadores de comparación y lógicos
6. Conversión de tipos
7. Entrada de datos desde consola
8. Prácticas guiadas
9. Evaluación práctica
10. Mini-Quiz

¿Qué es una Variable?

Una **variable** es un espacio en memoria que almacena un valor que puede cambiar durante la ejecución del programa.

Analogía en la Vida Real:

Imagina una **caja** con una **etiqueta**. La etiqueta es el nombre de la variable, y la caja contiene el valor.

Ejemplo: Una caja etiquetada "edad" puede contener el número 20. Más tarde, puede contener 21.

Sintaxis de Declaración:

```
tipoDeDato nombreVariable;  
tipoDeDato nombreVariable = valorInicial;
```

Tipos de Datos en C#

Tipos Numéricos Enteros:

Tipo	Rango	Tamaño
byte	0 a 255	8 bits
int	-2,147,483,648 a 2,147,483,647	32 bits
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	64 bits

Tipos Numéricos Decimales:

Tipo	Precisión	Tamaño
float	~7 dígitos	32 bits
double	~15-16 dígitos	64 bits
decimal	28-29 dígitos	128 bits

Tipos de Datos (Continuación)

Otros Tipos Importantes:

- `bool` : Valores verdadero/falso (true/false)
- `char` : Un solo carácter (ej: 'A', '5', '@')
- `string` : Secuencia de caracteres (texto)

Ejemplos de Declaración:

```
int edad = 25;
double precio = 99.99;
string nombre = "Juan";
bool esEstudiante = true;
char inicial = 'J';
```

Nota: C# es un lenguaje **fuertemente tipado**. Una vez declarada una variable con un tipo, no puede cambiar de tipo.

En la Vida Real: Variables

Escenario: Sistema de gestión de estudiantes

```
// Información de un estudiante
string nombreEstudiante = "María González";
int edad = 20;
double promedio = 85.5;
bool activo = true;
char grado = 'A';

// Información de una clase
string nombreClase = "Desarrollo de Software I";
int numeroEstudiantes = 30;
double precioMatricula = 5000.00;
```

Aplicación: En aplicaciones reales, las variables almacenan información que cambia constantemente: calificaciones, inventarios, precios, estados, etc.

Operadores Aritméticos

Operador	Descripción	Ejemplo	Resultado
+	Suma	5 + 3	8
-	Resta	10 - 4	6
*	Multiplicación	6 * 7	42
/	División	15 / 3	5
%	Módulo (resto)	17 % 5	2

Ejemplo Práctico:

```
int a = 10;
int b = 3;
int suma = a + b;           // 13
int producto = a * b;       // 30
int resto = a % b;          // 1
```

Operadores de Comparación

Operador	Descripción	Ejemplo	Resultado
<code>==</code>	Igual a	<code>5 == 5</code>	true
<code>!=</code>	Diferente de	<code>5 != 3</code>	true
<code><</code>	Menor que	<code>3 < 5</code>	true
<code>></code>	Mayor que	<code>10 > 7</code>	true
<code><=</code>	Menor o igual	<code>5 <= 5</code>	true
<code>>=</code>	Mayor o igual	<code>8 >= 6</code>	true

Importante: Los operadores de comparación siempre devuelven un valor `bool` (true o false).

Operadores Lógicos

Operador	Descripción	Ejemplo
&&	Y lógico (AND)	true && false → false
	O lógico (OR)	true false → true
!	Negación (NOT)	!true → false

Tabla de Verdad:

```
bool a = true;
bool b = false;

bool resultado1 = a && b; // false (ambos deben ser true)
bool resultado2 = a || b; // true (al menos uno es true)
bool resultado3 = !a; // false (niega el valor)
```

Errores Comunes

1. Usar = en lugar de ==

```
if (x = 5) {} // ✗ Error: asigna valor  
if (x == 5) {} // ✓ Correcto: compara
```

2. División entera cuando se necesita decimal

```
int resultado = 10 / 3; // ✗ Resultado: 3 ( pierde decimales  
double resultado = 10.0 / 3; // ✓ Resultado: 3.333...
```

3. No inicializar variables

```
int x;  
Console.WriteLine(x); // ✗ Error: variable no inicializada  
int x = 0;  
Console.WriteLine(x); // ✓ Correcto
```

Buenas Prácticas

- **Nombres descriptivos:** `edad` en lugar de `e`
- **Inicializar variables:** Siempre dar un valor inicial
- **Usar el tipo correcto:** `decimal` para dinero, `int` para conteos
- **Evitar "magic numbers":** Usar constantes con nombres
- **Comentarios cuando sea necesario:** Explicar el propósito

Ejemplo de Código Limpio:

```
// Constantes con nombres descriptivos
const double IVA = 0.18;
const int EDAD_MINIMA = 18;

// Variables bien nombradas
double precioProducto = 100.00;
double precioConIVA = precioProducto * (1 + IVA);
bool puedeComprar = edad >= EDAD_MINIMA;
```

Entrada de Datos desde Consola

Leer Texto:

```
Console.Write("Ingrese su nombre: ");
string nombre = Console.ReadLine();
```

Leer Números:

```
Console.Write("Ingrese su edad: ");
string entrada = Console.ReadLine();
int edad = int.Parse(entrada);

// O más seguro:
int edad = Convert.ToInt32(Console.ReadLine());
```

Ejemplo Completo:

```
Console.Write("Ingrese un número: ");
int numero = Convert.ToInt32(Console.ReadLine());
Console.WriteLine($"El número ingresado es: {numero}");
```

Nota: `Console.ReadLine()` siempre devuelve un `string`, por lo que debemos convertir a números si es necesario.

Práctica Guiada #1: Calcular Promedio

Objetivo:

Crear un programa que calcule el promedio de tres números ingresados por el usuario.

Instrucciones:

1. Crear un nuevo proyecto de consola
2. Pedir al usuario tres números
3. Calcular el promedio
4. Mostrar el resultado

Código Base:

```
using System;

namespace CalculadoraPromedio
{
    class Program
    {
        static void Main(string[] args)
        {
            // Tu código aquí
        }
    }
}
```

Solución Práctica Guiada #1

```
using System;

namespace CalculadoraPromedio
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Ingrese el primer número: ");
            double numero1 = Convert.ToDouble(Console.ReadLine());

            Console.Write("Ingrese el segundo número: ");
            double numero2 = Convert.ToDouble(Console.ReadLine());

            Console.Write("Ingrese el tercer número: ");
            double numero3 = Convert.ToDouble(Console.ReadLine());

            double promedio = (numero1 + numero2 + numero3) / 3;

            Console.WriteLine($"El promedio es: {promedio:F2}");
            Console.ReadKey();
        }
    }
}
```

Explicación:

- Usamos `double` para permitir decimales
- `Convert.ToDouble()` convierte texto a número
- `{promedio:F2}` formatea a 2 decimales

Práctica Guiada #2: Operaciones Básicas

Objetivo:

Implementar operaciones básicas con variables y operadores.

Instrucciones:

1. Declarar dos variables numéricas
2. Realizar suma, resta, multiplicación y división
3. Mostrar todos los resultados
4. Calcular el módulo (resto de la división)

Resultado Esperado:

```
Número 1: 15
Número 2: 4
Suma: 19
Resta: 11
Multiplicación: 60
División: 3.75
Módulo: 3
```

Solución Práctica Guiada #2

```
using System;

namespace OperacionesBasicas
{
    class Program
    {
        static void Main(string[] args)
        {
            int numero1 = 15;
            int numero2 = 4;

            int suma = numero1 + numero2;
            int resta = numero1 - numero2;
            int multiplicacion = numero1 * numero2;
            double division = (double)numero1 / numero2; // Cast
            int modulo = numero1 % numero2;

            Console.WriteLine($"Número 1: {numero1}");
            Console.WriteLine($"Número 2: {numero2}");
            Console.WriteLine($"Suma: {suma}");
            Console.WriteLine($"Resta: {resta}");
            Console.WriteLine($"Multiplicación: {multiplicacion}");
            Console.WriteLine($"División: {division}");
            Console.WriteLine($"Módulo: {modulo}");

            Console.ReadKey();
        }
    }
}
```

Conceptos Clave:

- **Cast:** `(double)numero1` convierte int a double
- División entre enteros da entero, por eso hacemos cast

Práctica Evaluada

Enunciado:

Crear un programa que simule una calculadora básica. El programa debe:

1. Solicitar dos números al usuario
2. Realizar las 4 operaciones básicas (+, -, *, /)
3. Mostrar todos los resultados formateados
4. Validar que el segundo número no sea cero para la división
5. Usar tipos de datos apropiados
6. Incluir comentarios explicativos

Criterios de Evaluación:

- **Funcionalidad (40%):** Todas las operaciones funcionan correctamente
- **Código (25%):** Uso correcto de variables y operadores
- **Validación (20%):** Manejo de división por cero
- **Presentación (15%):** Formato y comentarios

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todas las operaciones correctas	3-4 operaciones correctas	1-2 operaciones correctas	No funciona
Código	Uso perfecto de tipos y operadores	Uso correcto con pequeños errores	Uso básico	Uso incorrecto
Validación	Validación completa	Validación parcial	Validación básica	Sin validación
Presentación	Excelente formato y comentarios	Buen formato	Formato básico	Sin formato

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿Qué tipo de dato usarías para almacenar el precio de un producto?

- a) int
- b) double
- c) string
- d) bool

2. ¿Cuál es el resultado de: 17 % 5?

- a) 3
- b) 2
- c) 3.4
- d) 0

3. ¿Qué operador se usa para comparar si dos valores son iguales?

- a) =
- b) ==
- c) ===
- d) equals

Mini-Quiz (Continuación)

4. ¿Cuál es el resultado de: 10 / 3 cuando ambas variables son int?

- a) 3.33
- b) 3
- c) 3.0
- d) Error

5. ¿Qué devuelve Console.ReadLine()?

- a) int
- b) double
- c) string
- d) char

6. ¿Cuál es el operador lógico "Y" en C#?

- a) &
- b) &&
- c) AND
- d) +

7. ¿Es necesario inicializar una variable antes de usarla en C#?

- a) Verdadero
- b) Falso

Mini-Quiz (Continuación)

8. ¿Qué tipo de dato puede almacenar solo dos valores?

- a) int
- b) bool
- c) char
- d) string

9. ¿Cuál es el rango del tipo int en C#?

- a) 0 a 255
- b) -128 a 127
- c) -2,147,483,648 a 2,147,483,647
- d) Ilimitado

10. ¿Qué operador se usa para concatenar strings?

- a) +
- b) &
- c) concat
- d) join

Respuestas del Mini-Quiz

- 1. b) double** - Para precios con decimales usamos double o decimal.
- 2. b) 2** - El módulo devuelve el resto: $17 \div 5 = 3$ con resto 2.
- 3. b) ==** - == compara igualdad, = asigna valor.
- 4. b) 3** - División entre enteros da resultado entero.
- 5. c) string** - Console.ReadLine() siempre devuelve string.
- 6. b) &&** - && es el operador lógico "Y".
- 7. a) Verdadero** - Las variables deben inicializarse antes de usarse.
- 8. b) bool** - bool solo almacena true o false.
- 9. c)** - El rango de int es -2,147,483,648 a 2,147,483,647.
- 10. a) +** - El operador + concatena strings en C#.

Resumen de la Unidad

- ✓ Las variables almacenan valores que pueden cambiar
- ✓ C# tiene tipos de datos primitivos: int, double, string, bool, char
- ✓ Los operadores aritméticos permiten realizar cálculos
- ✓ Los operadores de comparación devuelven valores bool
- ✓ Los operadores lógicos (&&, ||, !) combinan condiciones
- ✓ Console.ReadLine() lee texto desde la consola
- ✓ Es necesario convertir strings a números para cálculos
- ✓ Usar nombres descriptivos y tipos apropiados es fundamental

Próxima Unidad: Estructuras de Control - Aprenderemos a tomar decisiones y controlar el flujo del programa con if, else, switch y bucles.

Tarea para Casa

Actividad:

Investigar y preparar ejemplos de estructuras de control (if, else, switch).

Entregables:

1. **Documento (2-3 páginas)** con:

- Explicación de if, else, else if
- Explicación de switch
- Ejemplos de código para cada estructura
- Casos de uso en la vida real

2. **Programa en C#** que demuestre:

- Uso de if para validar edad
- Uso de switch para menú de opciones
- Comentarios explicativos

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre tipos de datos y variables
- Documentación de operadores en C#
- Ejercicios adicionales
- Tutoriales interactivos

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 02

Variables, Tipos de Datos y Operadores - Enlaces, videos y material complementario



Documentación Oficial

[Tipos Integrados de C# Oficial](#)

Referencia completa de todos los tipos de datos integrados en C#, incluyendo tipos numéricos, booleanos y de texto.

[Operadores de C# Referencia](#)

Documentación completa sobre todos los operadores disponibles en C#: aritméticos, lógicos, de comparación, etc.

[Conversiones de Tipos Numéricos Guía](#)

Guía detallada sobre cómo funcionan las conversiones entre diferentes tipos numéricos en C#.



Videos Educativos

[Variables y Tipos de Datos en C# Video](#)

Explicación clara sobre cómo declarar variables y usar diferentes tipos de datos en C#.

[Operadores en C# Tutorial](#)

Tutorial sobre el uso de operadores aritméticos, lógicos y de comparación en C#.



Ejercicios Prácticos

[Ejercicios de C# - W3Schools Práctica](#)

Ejercicios interactivos para practicar variables, tipos de datos y operadores.

[Exercism - C# Track Ejercicios](#)

Plataforma con ejercicios progresivos de C# con retroalimentación automática.



Consejo de Estudio

Para dominar variables y operadores: 1) Practica declarando variables de diferentes tipos, 2) Experimenta con operadores aritméticos y lógicos, 3) Realiza conversiones entre tipos y observa los resultados, 4) Crea programas pequeños que usen entrada de datos desde consola. La práctica constante es clave.

Desarrollo de Software I

Unidad 03: Estructuras de Control

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Utilizar estructuras condicionales (if, else, switch) para tomar decisiones
- Implementar bucles (for, while) para repetir código
- Controlar el flujo de ejecución de un programa
- Crear programas interactivos con menús
- Resolver problemas usando estructuras de control

Agenda de la Unidad

1. Estructuras condicionales: if, else, else if
2. Estructura switch
3. Bucles: for
4. Bucles: while
5. Bucles anidados
6. Control de flujo: break y continue
7. Prácticas guiadas
8. Evaluación práctica
9. Mini-Quiz

Estructura Condicional: if

La estructura `if` permite ejecutar código solo si una condición es verdadera.

Sintaxis:

```
if (condición)
{
    // Código a ejecutar si la condición es verdadera
}
```

Ejemplo:

```
int edad = 18;
if (edad >= 18)
{
    Console.WriteLine("Eres mayor de edad");
}
```

Nota: La condición debe ser una expresión que devuelva `bool` (true o false).

Estructura if-else

Permite ejecutar un bloque de código si la condición es verdadera, y otro si es falsa.

Sintaxis:

```
if (condición)
{
    // Código si es verdadero
}
else
{
    // Código si es falso
}
```

Ejemplo:

```
int numero = 7;
if (numero % 2 == 0)
{
    Console.WriteLine("El número es par");
}
else
{
    Console.WriteLine("El número es impar");
}
```

Estructura if-else if-else

Permite evaluar múltiples condiciones en secuencia.

Sintaxis:

```
if (condición1)
{
    // Código 1
}
else if (condición2)
{
    // Código 2
}
else
{
    // Código por defecto
}
```

Ejemplo:

```
int calificacion = 85;
if (calificacion >= 90)
{
    Console.WriteLine("Excelente");
}
else if (calificacion >= 70)
{
    Console.WriteLine("Bueno");
}
else
{
    Console.WriteLine("Necesita mejorar");
}
```

En la Vida Real: Estructuras Condicionales

Escenario: Sistema de validación de acceso

```
// Validar acceso a un sistema
string usuario = "admin";
string password = "12345";
int intentos = 0;

if (usuario == "admin" && password == "12345")
{
    Console.WriteLine("Acceso concedido");
    intentos = 0;
}
else
{
    intentos++;
    Console.WriteLine($"Acceso denegado. Intentos: {intentos}");

    if (intentos >= 3)
    {
        Console.WriteLine("Cuenta bloqueada");
    }
}
```

Aplicación: Las estructuras condicionales se usan en validaciones, autenticación, cálculos condicionales, y toma de decisiones en programas.

Estructura switch

Permite seleccionar entre múltiples opciones basadas en el valor de una variable.

Sintaxis:

```
switch (variable)
{
    case valor1:
        // Código para valor1
        break;
    case valor2:
        // Código para valor2
        break;
    default:
        // Código por defecto
        break;
}
```

Ejemplo:

```
int opcion = 2;
switch (opcion)
{
    case 1:
        Console.WriteLine("Opción 1 seleccionada");
        break;
    case 2:
        Console.WriteLine("Opción 2 seleccionada");
        break;
    default:
        Console.WriteLine("Opción inválida");
        break;
}
```

Bucle for

Ejecuta un bloque de código un número específico de veces.

Sintaxis:

```
for (inicialización; condición; incremento)
{
    // Código a repetir
}
```

Ejemplo:

```
// Imprimir números del 1 al 10
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine(i);
}
```

Componentes:

- **Inicialización:** `int i = 1` - Se ejecuta una vez al inicio
- **Condición:** `i <= 10` - Se evalúa antes de cada iteración
- **Incremento:** `i++` - Se ejecuta después de cada iteración

Bucle while

Ejecuta un bloque de código mientras una condición sea verdadera.

Sintaxis:

```
while (condición)
{
    // Código a repetir
}
```

Ejemplo:

```
int contador = 1;
while (contador <= 5)
{
    Console.WriteLine($"Iteración {contador}");
    contador++;
}
```



Cuidado: Asegúrate de que la condición eventualmente sea falsa, o tendrás un bucle infinito.

Errores Comunes

1. Olvidar llaves en estructuras condicionales

```
if (x > 5)
    Console.WriteLine("Mayor"); // ✗ Solo funciona para una línea
    Console.WriteLine("Siempre"); // ✗ Siempre se ejecuta

if (x > 5)
{
    Console.WriteLine("Mayor"); // ✓ Correcto
    Console.WriteLine("Siempre");
}
```

2. Bucle infinito

```
int i = 0;
while (i < 10) // ✗ Falta incrementar i
{
    Console.WriteLine(i);
    // i++; // Sin esto, el bucle nunca termina
}
```

3. Olvidar break en switch

```
switch (opcion)
{
    case 1:
        Console.WriteLine("Uno");
        // ❌ Falta break - ejecutará el siguiente case
    case 2:
        Console.WriteLine("Dos");
        break;
}
```

Buenas Prácticas

- **Usar llaves siempre:** Incluso para una línea, mejora la legibilidad
- **Nombres descriptivos:** `contador` en lugar de `i` cuando sea posible
- **Evitar bucles anidados profundos:** Máximo 2-3 niveles
- **Usar switch para múltiples valores:** Más legible que múltiples if-else
- **Comentar lógica compleja:** Explicar el propósito del bucle

Ejemplo de Código Limpio:

```
// Calcular suma de números pares del 1 al 100
int suma = 0;
for (int numero = 2; numero <= 100; numero += 2)
{
    suma += numero;
}
Console.WriteLine($"Suma de pares: {suma}");
```

Práctica Guiada #1: Par o Impar

Objetivo:

Escribir un programa que determine si un número ingresado por el usuario es par o impar.

Instrucciones:

1. Pedir al usuario un número
2. Usar el operador módulo (%) para verificar si es divisible por 2
3. Si el resto es 0, es par; si no, es impar
4. Mostrar el resultado

Pista:

Un número es par si `numero % 2 == 0`

Solución Práctica Guiada #1

```
using System;

namespace ParImpar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Ingrese un número: ");
            int numero = Convert.ToInt32(Console.ReadLine());

            if (numero % 2 == 0)
            {
                Console.WriteLine($"El número {numero} es PAR");
            }
            else
            {
                Console.WriteLine($"El número {numero} es IMPAR");
            }

            Console.ReadKey();
        }
    }
}
```

Explicación:

- `numero % 2` devuelve el resto de dividir por 2
- Si el resto es 0, el número es divisible por 2 (par)
- Si el resto es 1, el número no es divisible por 2 (impar)

Práctica Guiada #2: Menú Interactivo

Objetivo:

Crear un menú interactivo utilizando switch.

Instrucciones:

1. Mostrar un menú con opciones (1-4)
2. Pedir al usuario que seleccione una opción
3. Usar switch para ejecutar la acción correspondiente
4. Incluir opción por defecto para opciones inválidas

Menú Sugerido:

- 1. Sumar dos números
- 2. Restar dos números
- 3. Multiplicar dos números
- 4. Salir

Solución Práctica Guiada #2

```
using System;

namespace MenuInteractivo
{
    class Program
    {
        static void Main(string[] args)
        {
            int opcion;
            do
            {
                Console.WriteLine("\n== MENÚ ==");
                Console.WriteLine("1. Sumar dos números");
                Console.WriteLine("2. Restar dos números");
                Console.WriteLine("3. Multiplicar dos números");
                Console.WriteLine("4. Salir");
                Console.Write("Seleccione una opción: ");

                opcion = Convert.ToInt32(Console.ReadLine());

                switch (opcion)
                {
                    case 1:
                        Console.Write("Número 1: ");
                        double a = Convert.ToDouble(Console.ReadLine());
                        Console.Write("Número 2: ");
                        double b = Convert.ToDouble(Console.ReadLine());
                        Console.WriteLine($"Resultado: {a + b}");
                        break;
                    case 2:
                        Console.Write("Número 1: ");
                        a = Convert.ToDouble(Console.ReadLine());
                        Console.Write("Número 2: ");
                        b = Convert.ToDouble(Console.ReadLine());
                        Console.WriteLine($"Resultado: {a - b}");
                        break;
                    case 3:
                        Console.Write("Número 1: ");
                        a = Convert.ToDouble(Console.ReadLine());
                        Console.Write("Número 2: ");
                        b = Convert.ToDouble(Console.ReadLine());
                        Console.WriteLine($"Resultado: {a * b}");
                        break;
                    case 4:
                        Console.WriteLine("¡Hasta luego!");
                        break;
                    default:
                        Console.WriteLine("Opción inválida");
                        break;
                }
            } while (opcion != 4);
        }
    }
}
```

```
    }  
}
```

Práctica Evaluada

Enunciado:

Crear un programa que simule un sistema de calificaciones. El programa debe:

1. Pedir al usuario una calificación (0-100)
2. Usar if-else if-else para determinar la letra:
 - 90-100: A
 - 80-89: B
 - 70-79: C
 - 60-69: D
 - 0-59: F
3. Validar que la calificación esté en el rango válido
4. Usar un bucle para permitir múltiples calificaciones
5. Incluir opción para salir

Criterios de Evaluación:

- **Funcionalidad (40%)**: Todas las funciones trabajan correctamente
- **Estructuras (30%)**: Uso correcto de if-else y bucles
- **Validación (20%)**: Validación de entrada

- **Código (10%):** Claridad y comentarios

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todas las funciones correctas	Mayoría funciona	Funciones básicas	No funciona
Estructuras	Uso perfecto de if/switch/bucles	Uso correcto	Uso básico	Uso incorrecto
Validación	Validación completa	Validación parcial	Validación básica	Sin validación
Código	Excelente organización	Buena organización	Organización básica	Sin organización

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿Qué estructura se usa para múltiples condiciones en secuencia?

- a) if
- b) if-else
- c) if-else if-else
- d) switch

2. ¿Cuál es el resultado de: for(int i=0; i<3; i++) ejecuta el bloque?

- a) 0 veces
- b) 1 vez
- c) 3 veces
- d) Infinitas veces

3. ¿Qué palabra clave se usa para salir de un switch?

- a) exit
- b) break
- c) return
- d) stop

Mini-Quiz (Continuación)

4. ¿Cuál es la diferencia entre for y while?

- a) No hay diferencia
- b) for se usa cuando sabemos las iteraciones, while cuando no
- c) while es más rápido
- d) for solo funciona con números

5. ¿Qué pasa si olvidas incrementar la variable en un while?

- a) El programa no compila
- b) Bucle infinito
- c) El programa se detiene
- d) Nada

6. ¿Puedes usar switch con strings en C#?

- a) Verdadero
- b) Falso

7. ¿Cuántas veces se ejecuta: for(int i=1; i<=5; i++)?

- a) 4
- b) 5
- c) 6
- d) Infinitas

Mini-Quiz (Continuación)

8. ¿Qué estructura es mejor para múltiples valores de una variable?

- a) if-else if
- b) switch
- c) while
- d) for

9. ¿El else es obligatorio en un if?

- a) Verdadero
- b) Falso

10. ¿Qué hace i++ en un bucle for?

- a) Incrementa i en 1
- b) Decrementa i en 1
- c) Multiplica i por 1
- d) Divide i entre 1

Respuestas del Mini-Quiz

- 1. c) if-else if-else** - Permite evaluar múltiples condiciones.
- 2. c) 3 veces** - $i=0,1,2$ (tres iteraciones).
- 3. b) break** - break sale del switch.
- 4. b)** - for cuando sabemos las iteraciones, while cuando no.
- 5. b) Bucle infinito** - La condición nunca se vuelve falsa.
- 6. a) Verdadero** - C# permite switch con strings.
- 7. b) 5** - $i=1,2,3,4,5$ (cinco iteraciones).
- 8. b) switch** - Más legible para múltiples valores.
- 9. b) Falso** - else es opcional.
- 10. a) Incrementa i en 1** - $i++$ es equivalente a $i = i + 1$.

Resumen de la Unidad

- ✓ Las estructuras condicionales (if, else, switch) permiten tomar decisiones
- ✓ Los bucles (for, while) permiten repetir código
- ✓ if-else if-else evalúa múltiples condiciones en secuencia
- ✓ switch es ideal para múltiples valores de una variable
- ✓ for se usa cuando conocemos el número de iteraciones
- ✓ while se usa cuando la condición puede cambiar dinámicamente
- ✓ Siempre usar break en switch para evitar fall-through
- ✓ Evitar bucles infinitos asegurando que la condición eventualmente sea falsa

Próxima Unidad: Arreglos y Bucles - Aprenderemos a trabajar con colecciones de datos y recorrerlas eficientemente.

Tarea para Casa

Actividad:

Investigar sobre arreglos y su uso en C#.

Entregables:

1. **Documento (2-3 páginas)** con:

- Qué son los arreglos
- Cómo declarar e inicializar arreglos en C#
- Cómo acceder a elementos de un arreglo
- Ejemplos de uso práctico

2. **Programa en C#** que demuestre:

- Declaración de un arreglo de números
- Inicialización con valores
- Recorrer el arreglo con un bucle
- Buscar un elemento específico

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre estructuras de control
- Ejemplos de if, else, switch
- Ejercicios prácticos adicionales
- Casos de uso reales

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 03

Estructuras de Control - Enlaces, videos y material complementario



Documentación Oficial

[Declaraciones de Selección \(if, switch\) Oficial](#)

Documentación oficial sobre las estructuras condicionales if, else y switch en C#.

[Declaraciones de Iteración \(bucles\) Oficial](#)

Referencia completa sobre los bucles for, while, do-while y foreach en C#.



Videos Educativos

[Estructuras Condicionales en C# Video](#)

Tutorial sobre el uso de if, else y switch para controlar el flujo de ejecución.

[Bucles en C# Tutorial](#)

Explicación de los diferentes tipos de bucles y cuándo usar cada uno.



Ejercicios Prácticos

[Ejercicios de Estructuras de Control Práctica](#)

Ejercicios interactivos para practicar condicionales y bucles.



Consejo de Estudio

Para dominar estructuras de control: 1) Practica escribiendo programas con múltiples condicionales anidadas, 2) Experimenta con diferentes tipos de bucles para resolver el mismo problema, 3) Crea programas que combinen condicionales y bucles, 4) Resuelve problemas de lógica usando estas estructuras. La práctica es fundamental.

Desarrollo de Software I

Unidad 04: Arreglos y Bucles

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Declarar e inicializar arreglos en C#
- Acceder y modificar elementos de un arreglo
- Recorrer arreglos usando for, foreach y while
- Realizar operaciones comunes con arreglos (buscar, sumar, promedio)
- Aplicar bucles para procesar colecciones de datos

Agenda de la Unidad

1. ¿Qué es un arreglo?
2. Declaración e inicialización de arreglos
3. Acceso a elementos del arreglo
4. Recorrer arreglos con for
5. Recorrer arreglos con foreach
6. Recorrer arreglos con while
7. Operaciones comunes con arreglos
8. Prácticas guiadas
9. Evaluación práctica
10. Mini-Quiz

¿Qué es un Arreglo?

Un **arreglo** es una colección de elementos del mismo tipo almacenados en posiciones contiguas de memoria.

Analogía en la Vida Real:

Imagina un **estante con cajones numerados**. Cada cajón contiene un elemento, y todos los elementos son del mismo tipo.

Ejemplo: Un arreglo de calificaciones: [85, 90, 78, 92, 88]

Características:

- Todos los elementos son del mismo tipo
- Tamaño fijo (se define al crear)
- Acceso por índice (empezando en 0)
- Eficiente para almacenar múltiples valores relacionados

Declaración e Inicialización

Forma 1: Declarar y luego inicializar

```
// Declarar un arreglo de enteros con 5 elementos
int[] numeros = new int[5];

// Asignar valores
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
```

Forma 2: Declarar e inicializar al mismo tiempo

```
// Inicializar con valores directamente
int[] numeros = {10, 20, 30, 40, 50};

// O usando new
int[] numeros = new int[] {10, 20, 30, 40, 50};
```

Nota: Los índices en C# empiezan en 0. El primer elemento está en la posición 0.

Acceso a Elementos

Lectura:

```
int[] calificaciones = {85, 90, 78, 92, 88};

// Acceder al primer elemento (índice 0)
int primera = calificaciones[0]; // 85

// Acceder al tercer elemento (índice 2)
int tercera = calificaciones[2]; // 78

// Acceder al último elemento
int ultima = calificaciones[4]; // 88
```

Escritura:

```
// Modificar un elemento
calificaciones[1] = 95; // Cambia el segundo elemento a 95

// Obtener la longitud del arreglo
int longitud = calificaciones.Length; // 5
```

 **Error común:** Intentar acceder a un índice fuera del rango causa una excepción.

Recorrer Arreglos con for

El bucle `for` es ideal cuando necesitas el índice.

Ejemplo Básico:

```
int[] numeros = {10, 20, 30, 40, 50};

// Recorrer e imprimir todos los elementos
for (int i = 0; i < numeros.Length; i++)
{
    Console.WriteLine($"Elemento {i}: {numeros[i]}");
```

Ejemplo: Modificar elementos

```
// Duplicar todos los valores
for (int i = 0; i < numeros.Length; i++)
{
    numeros[i] = numeros[i] * 2;
```

Ventajas:

- Acceso al índice
- Puedes modificar elementos
- Control total sobre el recorrido

Recorrer Arreglos con `foreach`

El bucle `foreach` es más simple cuando solo necesitas leer los valores.

Sintaxis:

```
foreach (tipo variable in arreglo)
{
    // Código usando variable
}
```

Ejemplo:

```
int[] calificaciones = {85, 90, 78, 92, 88};

// Recorrer e imprimir
foreach (int calificacion in calificaciones)
{
    Console.WriteLine($"Calificación: {calificacion}");
}
```

Ventajas:

- Sintaxis más simple
- No necesitas manejar índices
- Menos propenso a errores

Limitación: No puedes modificar elementos ni acceder al índice directamente.

Recorrer Arreglos con while

El bucle `while` es útil cuando la condición de parada es compleja.

Ejemplo:

```
int[] numeros = {10, 20, 30, 40, 50};  
int i = 0;  
  
while (i < numeros.Length)  
{  
    Console.WriteLine($"Elemento {i}: {numeros[i]}");  
    i++;  
}
```

Ejemplo: Buscar hasta encontrar

```
int[] valores = {5, 8, 3, 12, 7};  
int buscar = 12;  
int i = 0;  
bool encontrado = false;  
  
while (i < valores.Length && !encontrado)  
{  
    if (valores[i] == buscar)  
    {  
        encontrado = true;  
        Console.WriteLine($"Encontrado en posición {i}");  
    }  
    i++;  
}
```

En la Vida Real: Arreglos

Escenario: Sistema de gestión de estudiantes

```
// Almacenar calificaciones de un grupo
double[] calificaciones = new double[30];
string[] nombres = new string[30];

// Ingresar datos
for (int i = 0; i < calificaciones.Length; i++)
{
    Console.Write($"Nombre del estudiante {i + 1}: ");
    nombres[i] = Console.ReadLine();

    Console.Write($"Calificación: ");
    calificaciones[i] = Convert.ToDouble(Console.ReadLine());
}

// Calcular promedio
double suma = 0;
foreach (double cal in calificaciones)
{
    suma += cal;
}
double promedio = suma / calificaciones.Length;
```

Aplicación: Los arreglos se usan para almacenar listas de datos: inventarios, calificaciones, precios, nombres, etc.

Operaciones Comunes con Arreglos

1. Calcular Suma:

```
int[] numeros = {10, 20, 30, 40, 50};  
int suma = 0;  
foreach (int num in numeros)  
{  
    suma += num;  
}
```

2. Encontrar Máximo:

```
int maximo = numeros[0];  
for (int i = 1; i < numeros.Length; i++)  
{  
    if (numeros[i] > maximo)  
    {  
        maximo = numeros[i];  
    }  
}
```

3. Buscar Elemento:

```
int buscar = 30;  
bool encontrado = false;  
for (int i = 0; i < numeros.Length; i++)  
{  
    if (numeros[i] == buscar)  
    {  
        encontrado = true;  
        break;  
    }  
}
```

Errores Comunes

1. Índice fuera de rango

```
int[] arr = {1, 2, 3};  
Console.WriteLine(arr[3]); // ✗ Error: índice 3 no existe  
Console.WriteLine(arr[arr.Length]); // ✗ Error: Length es 3, ú
```

2. Olvidar inicializar elementos

```
int[] arr = new int[5];  
Console.WriteLine(arr[0]); // ✓ OK: valor por defecto es 0  
string[] nombres = new string[5];  
Console.WriteLine(nombres[0]); // ✓ OK: valor por defecto es nu
```

3. Modificar en foreach

```
foreach (int num in numeros)  
{  
    num = num * 2; // ✗ Error: no puedes modificar en foreach  
}  
  
// ✓ Usa for en su lugar  
for (int i = 0; i < numeros.Length; i++)  
{  
    numeros[i] = numeros[i] * 2;  
}
```

Buenas Prácticas

- **Usar Length:** Siempre usa `arreglo.Length` en lugar de números mágicos
- **Validar índices:** Verificar que el índice esté en rango antes de acceder
- **foreach para lectura:** Usa foreach cuando solo necesites leer valores
- **for para modificación:** Usa for cuando necesites modificar elementos
- **Nombres descriptivos:** `calificaciones` en lugar de `arr`

Ejemplo de Código Limpio:

```
// Calcular promedio de calificaciones
double[] calificaciones = {85, 90, 78, 92, 88};
double suma = 0;

foreach (double calificacion in calificaciones)
{
    suma += calificacion;
}

double promedio = suma / calificaciones.Length;
Console.WriteLine($"Promedio: {promedio:F2}");
```

Práctica Guiada #1: Promedio de Calificaciones

Objetivo:

Crear un programa que permita ingresar las calificaciones de un grupo de estudiantes y calcule el promedio.

Instrucciones:

1. Pedir al usuario cuántos estudiantes hay
2. Crear un arreglo del tamaño apropiado
3. Usar un bucle for para ingresar cada calificación
4. Calcular el promedio usando otro bucle
5. Mostrar el resultado

Solución Práctica Guiada #1

```
using System;

namespace PromedioCalificaciones
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("¿Cuántos estudiantes hay? ");
            int cantidad = Convert.ToInt32(Console.ReadLine());

            // Crear arreglo
            double[] calificaciones = new double[cantidad];

            // Ingresar calificaciones
            for (int i = 0; i < calificaciones.Length; i++)
            {
                Console.Write($"Calificación del estudiante {i} ");
                calificaciones[i] = Convert.ToDouble(Console.ReadLine());
            }

            // Calcular promedio
            double suma = 0;
            foreach (double cal in calificaciones)
            {
                suma += cal;
            }
            double promedio = suma / calificaciones.Length;

            Console.WriteLine($"El promedio es: {promedio:F2}");
            Console.ReadKey();
        }
    }
}
```

Explicación:

- Usamos `new double[cantidad]` para crear el arreglo
- El bucle `for` permite ingresar cada calificación
- `foreach` suma todas las calificaciones

- Dividimos la suma entre la cantidad para obtener el promedio

Práctica Guiada #2: Buscar Elemento

Objetivo:

Implementar un programa que busque un elemento en un arreglo.

Instrucciones:

1. Crear un arreglo con valores predefinidos
2. Pedir al usuario qué número buscar
3. Recorrer el arreglo buscando el número
4. Mostrar si se encontró y en qué posición
5. Si no se encuentra, mostrar mensaje apropiado

Solución Práctica Guiada #2

```
using System;

namespace BuscarEnArreglo
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numeros = {10, 25, 8, 42, 15, 30, 7};

            Console.WriteLine("¿Qué número desea buscar? ");
            int buscar = Convert.ToInt32(Console.ReadLine());

            bool encontrado = false;
            int posicion = -1;

            // Buscar en el arreglo
            for (int i = 0; i < numeros.Length; i++)
            {
                if (numeros[i] == buscar)
                {
                    encontrado = true;
                    posicion = i;
                    break; // Salir del bucle cuando se encuentra
                }
            }

            // Mostrar resultado
            if (encontrado)
            {
                Console.WriteLine($"\\n¡Encontrado! El número {busc
            }
            else
            {
                Console.WriteLine($"\\nNo se encontró el número {busc
            }

            Console.ReadKey();
        }
    }
}
```

Conceptos Clave:

- Usamos una variable booleana para rastrear si se encontró
- Guardamos la posición cuando encontramos el elemento
- `break` sale del bucle cuando encontramos el elemento

Práctica Evaluada

Enunciado:

Crear un programa de gestión de calificaciones que:

1. Permita ingresar calificaciones de N estudiantes
2. Calcule y muestre: promedio, calificación máxima, calificación mínima
3. Cuente cuántos estudiantes aprobaron (calificación ≥ 70)
4. Muestre todas las calificaciones ingresadas
5. Permita buscar una calificación específica

Criterios de Evaluación:

- **Funcionalidad (40%):** Todas las funciones trabajan correctamente
- **Arreglos (30%):** Uso correcto de arreglos y bucles
- **Código (20%):** Organización y claridad
- **Validación (10%):** Validación de entrada

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todas las funciones correctas	Mayoría funciona	Funciones básicas	No funciona
Arreglos	Uso perfecto de arreglos	Uso correcto	Uso básico	Uso incorrecto
Bucles	Bucles eficientes y correctos	Bucles correctos	Bucles básicos	Bucles incorrectos
Código	Excelente organización	Buena organización	Organización básica	Sin organización

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿En qué índice empiezan los arreglos en C#?

- a) 1
- b) 0
- c) -1
- d) Depende

2. ¿Cómo obtienes la longitud de un arreglo?

- a) arr.Size
- b) arr.Length
- c) arr.Count
- d) arr.length

3. ¿Puedes modificar elementos en un bucle foreach?

- a) Verdadero
- b) Falso

Mini-Quiz (Continuación)

4. ¿Cuál es el último índice válido de un arreglo de tamaño 10?

- a) 10
- b) 9
- c) 11
- d) 0

5. ¿Qué bucle es mejor para solo leer valores?

- a) for
- b) foreach
- c) while
- d) Todos iguales

6. ¿Qué pasa si accedes a arr[arr.Length]?

- a) Obtienes el último elemento
- b) Error de índice fuera de rango
- c) Obtienes null
- d) Obtienes 0

7. ¿Cómo declaras un arreglo de 5 enteros?

- a) int[] arr = new int(5);
- b) int[] arr = new int[5];
- c) int arr[5];
- d) int arr = new int[5];

Mini-Quiz (Continuación)

8. ¿Qué bucle necesitas si quieres modificar elementos del arreglo?

- a) foreach
- b) for
- c) while
- d) Cualquiera

9. ¿Puedes cambiar el tamaño de un arreglo después de crearlo?

- a) Verdadero
- b) Falso

10. ¿Cuál es la forma correcta de inicializar un arreglo con valores?

- a) int[] arr = {1, 2, 3};
- b) int[] arr = new int[] {1, 2, 3};
- c) Ambas son correctas
- d) Ninguna

Respuestas del Mini-Quiz

- 1. b) 0** - Los arreglos en C# empiezan en el índice 0.
- 2. b) arr.Length** - Length es la propiedad correcta.
- 3. b) Falso** - No puedes modificar elementos en foreach.
- 4. b) 9** - Si el tamaño es 10, los índices van de 0 a 9.
- 5. b) foreach** - foreach es más simple para solo leer.
- 6. b) Error de índice fuera de rango** - El último índice válido es Length-1.
- 7. b) int[] arr = new int[5];** - Sintaxis correcta.
- 8. b) for** - Necesitas acceso al índice para modificar.
- 9. b) Falso** - Los arreglos tienen tamaño fijo.
- 10. c) Ambas son correctas** - Ambas formas son válidas.

Resumen de la Unidad

- ✓ Los arreglos almacenan múltiples valores del mismo tipo
- ✓ Los índices empiezan en 0 y van hasta Length-1
- ✓ Puedes declarar arreglos con `new tipo[tamaño]` o con valores iniciales
- ✓ Usa `for` cuando necesites el índice o modificar elementos
- ✓ Usa `foreach` cuando solo necesites leer valores
- ✓ Usa `while` cuando la condición de parada sea compleja
- ✓ Siempre valida que los índices estén en rango
- ✓ Los arreglos tienen tamaño fijo una vez creados

Próxima Unidad: Funciones y Métodos - Aprenderemos a modularizar el código reutilizando funciones.

Tarea para Casa

Actividad:

Leer sobre funciones y métodos en C#.

Entregables:

1. **Documento (2-3 páginas)** con:

- Qué son las funciones y métodos
- Cómo declarar funciones en C#
- Parámetros y valores de retorno
- Diferencia entre funciones predefinidas y personalizadas
- Ventajas de usar funciones

2. **Programa en C#** que demuestre:

- Una función que calcule el área de un círculo
- Una función que valide si un número es primo
- Uso de funciones predefinidas de C#

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre bucles y arrays
- Ejemplos de for, while, foreach
- Ejercicios con arrays y listas
- Tutoriales avanzados

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 04

Arreglos y Bucles - Enlaces, videos y material complementario



Documentación Oficial

[Arreglos en C# Oficial](#)

Documentación completa sobre arreglos unidimensionales y multidimensionales en C#.

[Guía de Programación - Arreglos Guía](#)

Guía detallada sobre cómo trabajar con arreglos, incluyendo operaciones comunes y mejores prácticas.

[Colecciones en C# Guía](#)

Introducción a las colecciones como List, Dictionary y otras estructuras de datos en C#.



Videos Educativos

[Trabajando con Arreglos en C# Video](#)

Tutorial completo sobre cómo declarar, inicializar y manipular arreglos.

[Recorrer Arreglos con Bucles Tutorial](#)

Ejemplos prácticos de cómo usar for, foreach y while para recorrer arreglos.



Consejo de Estudio

Para dominar arreglos y bucles: 1) Practica creando arreglos de diferentes tipos y tamaños, 2) Implementa operaciones comunes como buscar, ordenar y sumar elementos, 3) Experimenta con bucles anidados para trabajar con arreglos multidimensionales, 4) Resuelve problemas que requieran procesar colecciones de datos. La combinación de arreglos y bucles es fundamental.

Desarrollo de Software I

Unidad 05: Funciones y Métodos

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Declarar y usar funciones en C#
- Pasar parámetros a funciones
- Retornar valores desde funciones
- Diferenciar entre funciones predefinidas y personalizadas
- Modularizar código usando funciones

Agenda de la Unidad

1. ¿Qué es una función?
2. Declaración de funciones
3. Funciones con parámetros
4. Funciones con valor de retorno
5. Funciones void (sin retorno)
6. Funciones predefinidas de C#
7. Ámbito de variables
8. Prácticas guiadas
9. Evaluación práctica
10. Mini-Quiz

¿Qué es una Función?

Una **función** (o método) es un bloque de código que realiza una tarea específica y puede ser reutilizado.

Ventajas de Usar Funciones:

- **Reutilización:** Escribir código una vez y usarlo múltiples veces
- **Modularidad:** Dividir programas complejos en partes más pequeñas
- **Mantenibilidad:** Más fácil de entender y modificar
- **Organización:** Código más limpio y estructurado

Analogía:

Una función es como una **receta**: defines los ingredientes (parámetros), los pasos (código), y obtienes un resultado (valor de retorno).

Declaración de Funciones

Sintaxis Básica:

```
tipoRetorno NombreFuncion()
{
    // Código de la función
    return valor; // Si tiene retorno
}
```

Ejemplo Simple:

```
// Función que saluda
static void Saludar()
{
    Console.WriteLine("¡Hola!");
}

// Llamar a la función
Saludar(); // Imprime: ¡Hola!
```

Nota: En C#, las funciones dentro de una clase deben ser `static` para ser llamadas desde `Main` sin crear una instancia.

Funciones con Parámetros

Los **parámetros** permiten pasar datos a una función.

Sintaxis:

```
tipoRetorno NombreFuncion(tipo1 param1, tipo2 param2)
{
    // Usar param1 y param2
}
```

Ejemplo:

```
// Función que suma dos números
static int Sumar(int a, int b)
{
    return a + b;
}

// Llamar la función
int resultado = Sumar(5, 3); // resultado = 8
```

Ejemplo con Múltiples Parámetros:

```
static void MostrarInfo(string nombre, int edad)
{
    Console.WriteLine($"Nombre: {nombre}, Edad: {edad}");
}

MostrarInfo("Juan", 25); // Nombre: Juan, Edad: 25
```

Funciones con Valor de Retorno

Las funciones pueden devolver un valor usando `return`.

Ejemplo:

```
// Calcular el área de un rectángulo
static double CalcularArea(double largo, double ancho)
{
    double area = largo * ancho;
    return area;
}

// Usar la función
double area = CalcularArea(10, 5); // area = 50
```

Tipos de Retorno:

- `int`, `double`, `string`, `bool`, etc.
- `void` - No retorna nada

Importante: Una función con tipo de retorno diferente a `void` debe tener al menos un `return`.

Funciones void (Sin Retorno)

Las funciones `void` no retornan ningún valor. Se usan para realizar acciones.

Ejemplo:

```
// Función que imprime un mensaje
static void ImprimirMensaje(string mensaje)
{
    Console.WriteLine(mensaje);
}

// Función que muestra un menú
static void MostrarMenu()
{
    Console.WriteLine("1. Opción 1");
    Console.WriteLine("2. Opción 2");
    Console.WriteLine("3. Salir");
}

// Llamar funciones void
ImprimirMensaje("Bienvenido");
MostrarMenu();
```

Nota: Las funciones void pueden usar `return;` sin valor para salir temprano.

En la Vida Real: Funciones

Escenario: Calculadora de áreas de figuras geométricas

```
// Función para área de círculo
static double AreaCirculo(double radio)
{
    return Math.PI * radio * radio;
}

// Función para área de rectángulo
static double AreaRectangulo(double largo, double ancho)
{
    return largo * ancho;
}

// Función para área de triángulo
static double AreaTriangulo(double base, double altura)
{
    return (base * altura) / 2;
}

// Usar las funciones
double area1 = AreaCirculo(5);
double area2 = AreaRectangulo(10, 8);
double area3 = AreaTriangulo(6, 4);
```

Aplicación: Las funciones permiten organizar código en módulos reutilizables, facilitando el mantenimiento y la comprensión.

Funciones Predefinidas de C#

C# incluye muchas funciones útiles ya implementadas.

Matemáticas (Math):

```
double resultado;

resultado = Math.Max(10, 20);      // 20 (el mayor)
resultado = Math.Min(10, 20);      // 10 (el menor)
resultado = Math.Abs(-5);         // 5 (valor absoluto)
resultado = Math.Round(3.7);       // 4 (redondear)
resultado = Math.Sqrt(16);         // 4 (raíz cuadrada)
resultado = Math.Pow(2, 3);        // 8 (2 elevado a 3)
```

Strings:

```
string texto = "Hola Mundo";
int longitud = texto.Length;           // 10
string mayusculas = texto.ToUpper();   // "HOLA MUNDO"
string minusculas = texto.ToLower();   // "hola mundo"
bool contiene = texto.Contains("Mundo"); // true
```

Ámbito de Variables

El **ámbito** (scope) determina dónde una variable es accesible.

Variables Locales:

```
static void MiFuncion()
{
    int x = 10; // Variable local
    Console.WriteLine(x); // ✅ OK
}

static void OtraFuncion()
{
    Console.WriteLine(x); // ❌ Error: x no existe aquí
}
```

Parámetros:

```
static void Procesar(int numero)
{
    // numero es accesible solo dentro de esta función
    Console.WriteLine(numero);
}
```

Regla: Las variables declaradas dentro de una función solo existen dentro de esa función.

Errores Comunes

1. Olvidar return en función con retorno

```
static int Sumar(int a, int b)
{
    int resultado = a + b;
    // ✗ Error: falta return
}

static int Sumar(int a, int b)
{
    return a + b; // ✓ Correcto
}
```

2. Usar variable fuera de su ámbito

```
static void Funcion1()
{
    int x = 5;
}

static void Funcion2()
{
    Console.WriteLine(x); // ✗ Error: x no existe aquí
}
```

3. Olvidar static

```
void MiFuncion() // ✗ Error si se llama desde Main
{
}
```

```
static void MiFuncion() // ✓ Correcto
{
```

Buenas Prácticas

- **Nombres descriptivos:** `CalcularPromedio` en lugar de `Calc`
- **Una función, una responsabilidad:** Cada función debe hacer una cosa
- **Parámetros claros:** Usar nombres que indiquen qué esperan
- **Comentarios cuando sea necesario:** Explicar funciones complejas
- **Mantener funciones pequeñas:** Máximo 20-30 líneas idealmente

Ejemplo de Código Limpio:

```
// Función bien nombrada y con propósito claro
static double CalcularDescuento(double precio, double porcentaje)
{
    return precio * (porcentaje / 100);
}

// Uso claro
double descuento = CalcularDescuento(100, 15); // 15% de descuento
```

Práctica Guiada #1: Áreas de Figuras

Objetivo:

Crear un programa que calcule el área de varias figuras geométricas utilizando funciones.

Instrucciones:

1. Crear función para área de círculo: `AreaCirculo(radio)`
2. Crear función para área de rectángulo:
`AreaRectangulo(largo, ancho)`
3. Crear función para área de triángulo:
`AreaTriangulo(base, altura)`
4. Crear menú para seleccionar figura
5. Pedir datos necesarios y mostrar resultado

Solución Práctica Guiada #1

```
using System;

namespace CalculadoraAreas
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("==> Calculadora de Áreas ==>");
            Console.WriteLine("1. Círculo");
            Console.WriteLine("2. Rectángulo");
            Console.WriteLine("3. Triángulo");
            Console.Write("Seleccione opción: ");

            int opcion = Convert.ToInt32(Console.ReadLine());

            switch (opcion)
            {
                case 1:
                    Console.Write("Radio: ");
                    double radio = Convert.ToDouble(Console.ReadLine());
                    Console.WriteLine($"Área: {AreaCirculo(radio)}");
                    break;
                case 2:
                    Console.Write("Largo: ");
                    double largo = Convert.ToDouble(Console.ReadLine());
                    Console.Write("Ancho: ");
                    double ancho = Convert.ToDouble(Console.ReadLine());
                    Console.WriteLine($"Área: {AreaRectangulo(largo, ancho)}");
                    break;
                case 3:
                    Console.Write("Base: ");
                    double baseTri = Convert.ToDouble(Console.ReadLine());
                    Console.Write("Altura: ");
                    double altura = Convert.ToDouble(Console.ReadLine());
                    Console.WriteLine($"Área: {AreaTriangulo(baseTri, altura)}");
                    break;
            }
        }

        static double AreaCirculo(double radio)
        {
            return Math.PI * radio * radio;
        }

        static double AreaRectangulo(double largo, double ancho)
        {
            return largo * ancho;
        }

        static double AreaTriangulo(double baseTri, double altura)
        {
            return (baseTri * altura) / 2;
        }
    }
}
```

```
    }  
}
```

Práctica Guiada #2: Validación de Datos

Objetivo:

Implementar funciones para validar datos ingresados por el usuario.

Instrucciones:

1. Crear función `ValidarEdad(int edad)` que retorne true si edad está entre 0 y 120
2. Crear función `ValidarEmail(string email)` que verifique si contiene "@"
3. Crear función `ValidarNumeroPositivo(double num)` que retorne true si num > 0
4. Crear programa que use estas funciones para validar entrada del usuario

Solución Práctica Guiada #2

```
using System;

namespace ValidacionDatos
{
    class Program
    {
        static void Main(string[] args)
        {
            // Validar edad
            Console.Write("Ingrese su edad: ");
            int edad = Convert.ToInt32(Console.ReadLine());

            if (ValidarEdad(edad))
            {
                Console.WriteLine("Edad válida");
            }
            else
            {
                Console.WriteLine("Edad inválida (debe estar entre 0 y 120)");
            }

            // Validar email
            Console.Write("Ingrese su email: ");
            string email = Console.ReadLine();

            if (ValidarEmail(email))
            {
                Console.WriteLine("Email válido");
            }
            else
            {
                Console.WriteLine("Email inválido (debe contener '@')");
            }

            // Validar número positivo
            Console.Write("Ingrese un número: ");
            double numero = Convert.ToDouble(Console.ReadLine());

            if (ValidarNumeroPositivo(numero))
            {
                Console.WriteLine("Número válido");
            }
            else
            {
                Console.WriteLine("Número inválido (debe ser positivo)");
            }
        }

        static bool ValidarEdad(int edad)
        {
            return edad >= 0 && edad <= 120;
        }

        static bool ValidarEmail(string email)
        {
            return email.Contains("@");
        }
    }
}
```

```
    }

    static bool ValidarNumeroPositivo(double num)
    {
        return num > 0;
    }
}
```

Práctica Evaluada

Enunciado:

Crear un programa de utilidades matemáticas que incluya:

1. Función para calcular factorial de un número
2. Función para verificar si un número es primo
3. Función para calcular potencia ($\text{base}^{\text{exponente}}$)
4. Función para calcular promedio de un arreglo
5. Menú interactivo para seleccionar operación
6. Validación de entrada usando funciones

Criterios de Evaluación:

- **Funcionalidad (40%)**: Todas las funciones trabajan correctamente
- **Funciones (30%)**: Uso correcto de funciones y parámetros
- **Código (20%)**: Organización y claridad
- **Validación (10%)**: Validación de entrada

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todas las funciones correctas	Mayoría funciona	Funciones básicas	No funciona
Funciones	Uso perfecto de funciones	Uso correcto	Uso básico	Uso incorrecto
Parámetros	Parámetros bien diseñados	Parámetros correctos	Parámetros básicos	Parámetros incorrectos
Código	Excelente organización	Buena organización	Organización básica	Sin organización

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿Qué palabra clave se usa para retornar un valor de una función?

- a) return
- b) break
- c) exit
- d) end

2. ¿Qué significa void en una función?

- a) Retorna un valor
- b) No retorna nada
- c) Tiene parámetros
- d) Es una función especial

3. ¿Las funciones en C# dentro de clases deben ser static para llamarlas desde Main?

- a) Verdadero
- b) Falso

Mini-Quiz (Continuación)

4. ¿Cuántos valores puede retornar una función en C#?

- a) Múltiples
- b) Solo uno
- c) Ninguno
- d) Infinitos

5. ¿Qué es un parámetro?

- a) Valor retorna
- b) Valor pasado a la función
- c) Nombre de la función
- d) Tipo de retorno

6. ¿Puedes tener múltiples return en una función?

- a) Verdadero
- b) Falso

7. ¿Qué función predefinida calcula la raíz cuadrada?

- a) Math.Sqrt
- b) Math.Root
- c) Math.Square
- d) Math.Pow

Mini-Quiz (Continuación)

8. ¿Las variables declaradas dentro de una función son accesibles fuera de ella?

a) Verdadero

b) Falso

9. ¿Cuál es la ventaja principal de usar funciones?

a) Hace el código más rápido

b) Permite reutilizar código

c) Reduce el tamaño del programa

d) Todas las anteriores

10. ¿Qué retorna Math.Max(5, 10)?

a) 5

b) 10

c) 15

d) Error

Respuestas del Mini-Quiz

- 1. a) return** - return retorna un valor de la función.
- 2. b) No retorna nada** - void significa que no retorna valor.
- 3. a) Verdadero** - Las funciones deben ser static para llamarlas desde Main sin instancia.
- 4. b) Solo uno** - Una función retorna un solo valor (o ninguno con void).
- 5. b) Valor pasado a la función** - Los parámetros son valores que se pasan a la función.
- 6. a) Verdadero** - Puedes tener múltiples return, pero solo se ejecutará uno.
- 7. a) Math.Sqrt** - Math.Sqrt calcula la raíz cuadrada.
- 8. b) Falso** - Las variables locales solo existen dentro de la función.
- 9. b) Permite reutilizar código** - La reutilización es la ventaja principal.
- 10. b) 10** - Math.Max retorna el mayor de los dos valores.

Resumen de la Unidad

- ✓ Las funciones permiten modularizar y reutilizar código
- ✓ Las funciones pueden recibir parámetros y retornar valores
- ✓ void significa que la función no retorna nada
- ✓ return se usa para retornar valores de funciones
- ✓ Las funciones deben ser static para llamarlas desde Main
- ✓ C# incluye muchas funciones predefinidas útiles (Math, string, etc.)
- ✓ Las variables locales solo existen dentro de la función
- ✓ Usar funciones mejora la organización y mantenibilidad del código

Próxima Unidad: Programación Orientada a Objetos (POO) -
Aprenderemos a crear clases y objetos para modelar el mundo real.

Tarea para Casa

Actividad:

Investigar sobre Programación Orientada a Objetos (POO).

Entregables:

1. **Documento (3-4 páginas)** con:

- Qué es POO y sus principios básicos
- Qué son las clases y objetos
- Qué son las propiedades y métodos
- Qué son los constructores
- Ejemplos de clases en la vida real

2. **Programa en C#** que demuestre:

- Crear una clase simple (ej: Persona)
- Definir propiedades y métodos
- Crear objetos (instancias) de la clase
- Usar las propiedades y métodos

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre métodos y funciones
- Ejemplos de sobrecarga de métodos
- Parámetros y valores de retorno
- Mejores prácticas

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 05

Funciones y Métodos - Enlaces, videos y material complementario



Documentación Oficial

Métodos en C# Oficial

Documentación completa sobre cómo definir y usar métodos en C#, incluyendo parámetros y valores de retorno.

Sobrecarga de Métodos Guía

Guía sobre cómo crear múltiples versiones de un método con diferentes parámetros.



Videos Educativos

Funciones y Métodos en C# Video

Tutorial completo sobre cómo crear y usar métodos para organizar y reutilizar código.



Consejo de Estudio

Para dominar funciones y métodos: 1) Practica dividiendo programas grandes en métodos más pequeños, 2) Experimenta con diferentes tipos de parámetros (por valor, por referencia), 3) Crea métodos que retornen diferentes tipos de datos, 4) Practica la sobrecarga de métodos. Los métodos son fundamentales para escribir código organizado y reutilizable.

Desarrollo de Software I

Unidad 06: Programación Orientada a
Objetos (POO)

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Comprender los conceptos fundamentales de POO
- Crear clases y objetos en C#
- Definir propiedades y métodos en clases
- Usar constructores para inicializar objetos
- Instanciar y manipular objetos

Agenda de la Unidad

1. ¿Qué es POO?
2. Clases y objetos
3. Propiedades
4. Métodos
5. Constructores
6. Instanciación de objetos
7. Prácticas guiadas
8. Evaluación práctica
9. Mini-Quiz

¿Qué es POO?

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código en objetos que contienen datos y comportamiento.

Principios Fundamentales:

- **Clase:** Plantilla o molde para crear objetos
- **Objeto:** Instancia específica de una clase
- **Encapsulación:** Agrupar datos y métodos relacionados
- **Abstracción:** Ocultar detalles complejos

Analogía:

Una **clase** es como un plano de una casa. Un **objeto** es la casa real construida usando ese plano.

Ejemplo: La clase "Persona" define qué es una persona. El objeto "Juan" es una persona específica.

Clases y Objetos

Clase:

Una **clase** es una plantilla que define las características (propiedades) y comportamientos (métodos) que tendrán los objetos.

Objeto:

Un **objeto** es una instancia específica de una clase. Cada objeto tiene sus propios valores para las propiedades.

Ejemplo:

```
// Definir la clase Persona
class Persona
{
    public string Nombre;
    public int Edad;
}

// Crear objetos (instancias)
Persona persona1 = new Persona();
persona1.Nombre = "Juan";
persona1.Edad = 25;

Persona persona2 = new Persona();
persona2.Nombre = "María";
persona2.Edad = 30;
```

Propiedades

Las **propiedades** son características o atributos de un objeto.

Forma Simple (Campos públicos):

```
class Persona
{
    public string Nombre;      // Propiedad pública
    public int Edad;          // Propiedad pública
}
```

Forma Recomendada (Propiedades con get/set):

```
class Persona
{
    private string nombre;    // Campo privado

    public string Nombre      // Propiedad pública
    {
        get { return nombre; }
        set { nombre = value; }
    }
}
```

Uso:

```
Persona p = new Persona();
p.Nombre = "Juan"; // Usar la propiedad
string n = p.Nombre; // Leer la propiedad
```

Métodos

Los **métodos** son funciones que definen el comportamiento de un objeto.

Ejemplo:

```
class Persona
{
    public string Nombre;
    public int Edad;

    // Método para mostrar información
    public void MostrarInfo()
    {
        Console.WriteLine($"Nombre: {Nombre}, Edad: {Edad}");
    }

    // Método con parámetros
    public void CumplirAnios()
    {
        Edad++;
        Console.WriteLine($"¡Feliz cumpleaños! Ahora tienes {Edad} años");
    }
}
```

Uso:

```
Persona p = new Persona();
p.Nombre = "Juan";
p.Edad = 25;
p.MostrarInfo();      // Llama al método
p.CumplirAnios();    // Llama al método
```

Constructores

Un **constructor** es un método especial que se ejecuta cuando se crea un objeto. Inicializa el objeto.

Sintaxis:

```
class Persona
{
    public string Nombre;
    public int Edad;

    // Constructor sin parámetros
    public Persona()
    {
        Nombre = "Sin nombre";
        Edad = 0;
    }

    // Constructor con parámetros
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
}
```

Uso:

```
Persona p1 = new Persona();           // Usa constructor sin parámetros
Persona p2 = new Persona("Juan", 25); // Usa constructor con parámetros
```

En la Vida Real: POO

Escenario: Sistema de gestión de estudiantes

```
// Clase Estudiante
class Estudiante
{
    public string Nombre;
    public int Edad;
    public double Promedio;

    // Constructor
    public Estudiante(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
        Promedio = 0;
    }

    // Método para actualizar promedio
    public void ActualizarPromedio(double nuevoPromedio)
    {
        Promedio = nuevoPromedio;
    }

    // Método para mostrar información
    public void MostrarInfo()
    {
        Console.WriteLine($"Estudiante: {Nombre}");
        Console.WriteLine($"Edad: {Edad}");
        Console.WriteLine($"Promedio: {Promedio:F2}");
    }
}

// Crear objetos
Estudiante est1 = new Estudiante("María", 20);
est1.ActualizarPromedio(85.5);
est1.MostrarInfo();
```

Errores Comunes

1. Olvidar usar 'new' al crear objeto

```
Persona p;  
p.Nombre = "Juan"; // ✗ Error: objeto no inicializado  
  
Persona p = new Persona();  
p.Nombre = "Juan"; // ✓ Correcto
```

2. Acceder a propiedades antes de inicializar

```
Persona p = new Persona();  
Console.WriteLine(p.Nombre); // ✓ OK: valor por defecto o null  
  
// Pero si Nombre es null:  
string nombre = p.Nombre.ToUpper(); // ✗ Error si Nombre es null
```

3. Olvidar 'public' en propiedades

```
class Persona  
{  
    string Nombre; // ✗ Por defecto es private, no accesible fuera  
    public string Nombre; // ✓ Accesible desde fuera
```

Buenas Prácticas

- **Nombres descriptivos:** Estudiante en lugar de Est
- **Usar propiedades en lugar de campos públicos:** Mejor encapsulación
- **Constructores para inicialización:** Asegurar estado válido
- **Un archivo por clase:** Mejor organización
- **Comentar clases complejas:** Explicar propósito

Ejemplo de Código Limpio:

```
// Clase bien estructurada
class Persona
{
    // Propiedades
    public string Nombre { get; set; }
    public int Edad { get; set; }

    // Constructor
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }

    // Métodos
    public void MostrarInfo()
    {
        Console.WriteLine($"{Nombre}, {Edad} años");
    }
}
```

Práctica Guiada #1: Clase Persona

Objetivo:

Crear una clase Persona con propiedades como Nombre, Edad y un método para mostrar su información.

Instrucciones:

1. Crear clase Persona
2. Agregar propiedades: Nombre (string) y Edad (int)
3. Crear constructor que reciba nombre y edad
4. Crear método MostrarInfo() que muestre nombre y edad
5. En Main, crear objetos y usar el método

Solución Práctica Guiada #1

```
using System;

namespace ClasePersona
{
    class Persona
    {
        public string Nombre;
        public int Edad;

        // Constructor
        public Persona(string nombre, int edad)
        {
            Nombre = nombre;
            Edad = edad;
        }

        // Método para mostrar información
        public void MostrarInfo()
        {
            Console.WriteLine($"Nombre: {Nombre}");
            Console.WriteLine($"Edad: {Edad} años");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Crear objetos
            Persona persona1 = new Persona("Juan", 25);
            Persona persona2 = new Persona("María", 30);

            // Usar métodos
            persona1.MostrarInfo();
            Console.WriteLine();
            persona2.MostrarInfo();

            Console.ReadKey();
        }
    }
}
```

Explicación:

- La clase Persona define la estructura
- El constructor inicializa las propiedades
- Cada objeto tiene sus propios valores
- Los métodos definen el comportamiento

Práctica Guiada #2: Manipular Objetos

Objetivo:

Instanciar objetos y manipular sus propiedades en el programa.

Instrucciones:

1. Crear clase `Producto` con propiedades: Nombre, Precio, Cantidad
2. Agregar método `CalcularTotal()` que retorne `Precio * Cantidad`
3. Agregar método `MostrarDetalles()` que muestre toda la información
4. En Main, crear varios productos y mostrar sus detalles

Solución Práctica Guiada #2

```
using System;

namespace Productos
{
    class Producto
    {
        public string Nombre;
        public double Precio;
        public int Cantidad;

        public Producto(string nombre, double precio, int cantidad)
        {
            Nombre = nombre;
            Precio = precio;
            Cantidad = cantidad;
        }

        public double CalcularTotal()
        {
            return Precio * Cantidad;
        }

        public void MostrarDetalles()
        {
            Console.WriteLine($"Producto: {Nombre}");
            Console.WriteLine($"Precio unitario: ${Precio:F2}");
            Console.WriteLine($"Cantidad: {Cantidad}");
            Console.WriteLine($"Total: ${CalcularTotal():F2}");
            Console.WriteLine();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Producto p1 = new Producto("Laptop", 50000, 2);
            Producto p2 = new Producto("Mouse", 500, 5);

            p1.MostrarDetalles();
            p2.MostrarDetalles();

            Console.ReadKey();
        }
    }
}
```

Práctica Evaluada

Enunciado:

Crear un sistema de gestión de libros que incluya:

1. Clase `Libro` con propiedades: Titulo, Autor, AñoPublicacion, Precio
2. Constructor que inicialice todas las propiedades
3. Método `MostrarInfo()` que muestre toda la información
4. Método `EsAntiguo()` que retorne true si el libro tiene más de 20 años
5. En Main, crear al menos 3 libros y mostrar su información
6. Mostrar cuántos libros son antiguos

Criterios de Evaluación:

- **Clase (40%)**: Estructura correcta de la clase
- **Propiedades (20%)**: Propiedades bien definidas
- **Métodos (20%)**: Métodos funcionan correctamente
- **Código (20%)**: Organización y claridad

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Clase	Estructura perfecta	Estructura correcta	Estructura básica	Estructura incorrecta
Propiedades	Todas correctas	Mayoría correctas	Básicas	Incorrectas
Métodos	Todos funcionan	Mayoría funciona	Básicos	No funcionan
Código	Excelente organización	Buena organización	Organización básica	Sin organización

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿Qué es una clase?

- a) Un objeto
- b) Una plantilla para crear objetos
- c) Un método
- d) Una variable

2. ¿Qué es un objeto?

- a) Una clase
- b) Una instancia de una clase
- c) Un método
- d) Una propiedad

3. ¿Qué palabra clave se usa para crear un objeto?

- a) create
- b) new
- c) make
- d) object

Mini-Quiz (Continuación)

4. ¿Qué es un constructor?

- a) Un método especial que inicializa el objeto
- b) Una propiedad
- c) Un método normal
- d) Una clase

5. ¿Cuántos constructores puede tener una clase?

- a) Solo uno
- b) Múltiples (sobrecarga)
- c) Ninguno
- d) Infinitos

6. ¿Las propiedades deben ser públicas para accederlas desde fuera?

- a) Verdadero
- b) Falso

7. ¿Qué es POO?

- a) Programación Orientada a Objetos
- b) Programación Objetiva
- c) Programación Organizada
- d) Programación Optimizada

Mini-Quiz (Continuación)

8. ¿Puedes tener múltiples objetos de la misma clase?

- a) Verdadero
- b) Falso

9. ¿Qué define el comportamiento de un objeto?

- a) Propiedades
- b) Métodos
- c) Constructores
- d) Clases

10. ¿El constructor tiene el mismo nombre que la clase?

- a) Verdadero
- b) Falso

Respuestas del Mini-Quiz

- 1. b) Una plantilla para crear objetos** - La clase es el molde.
- 2. b) Una instancia de una clase** - El objeto es la instancia específica.
- 3. b) new** - new crea una nueva instancia.
- 4. a) Un método especial que inicializa el objeto** - El constructor inicializa.
- 5. b) Múltiples (sobrecarga)** - Puedes tener varios constructores.
- 6. a) Verdadero** - Deben ser públicas para acceso externo.
- 7. a) Programación Orientada a Objetos** - POO es el paradigma.
- 8. a) Verdadero** - Puedes crear múltiples instancias.
- 9. b) Métodos** - Los métodos definen comportamiento.
- 10. a) Verdadero** - El constructor tiene el mismo nombre.

Resumen de la Unidad

- ✓ POO organiza código en clases y objetos
- ✓ Una clase es una plantilla, un objeto es una instancia
- ✓ Las propiedades almacenan datos del objeto
- ✓ Los métodos definen el comportamiento
- ✓ Los constructores inicializan objetos
- ✓ Usamos `new` para crear objetos
- ✓ Cada objeto tiene sus propios valores de propiedades
- ✓ POO mejora la organización y reutilización del código

Próxima Unidad: Herencia, Encapsulación y Manejo de Excepciones - Aprenderemos conceptos avanzados de POO y cómo manejar errores.

Tarea para Casa

Actividad:

Leer sobre encapsulación, herencia y sobrecarga de métodos.

Entregables:

1. **Documento (3-4 páginas)** con:

- Qué es encapsulación y modificadores de acceso
- Qué es herencia y cómo funciona
- Qué es sobrecarga de métodos
- Ventajas de cada concepto
- Ejemplos prácticos

2. **Programa en C#** que demuestre:

- Una clase base y una clase derivada (herencia)
- Uso de modificadores de acceso (public, private)
- Sobrecarga de métodos

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre Programación Orientada a Objetos
- Ejemplos de clases y objetos
- Encapsulación y propiedades
- Patrones de diseño básicos

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 06

Programación Orientada a Objetos (POO) - Enlaces, videos y material complementario



Documentación Oficial

[Programación Orientada a Objetos en C# Oficial](#)

Guía completa sobre los conceptos fundamentales de POO en C#: clases, objetos, encapsulación, herencia y polimorfismo.

[Clases en C# Referencia](#)

Documentación detallada sobre cómo definir y usar clases en C#.

[Constructores en C# Guía](#)

Explicación sobre constructores, cómo inicializar objetos y diferentes tipos de constructores.



Videos Educativos

[Introducción a POO en C# Video](#)

Tutorial completo sobre los conceptos fundamentales de programación orientada a objetos.

[Clases y Objetos en C# Tutorial](#)

Ejemplos prácticos de cómo crear clases, instanciar objetos y trabajar con propiedades y métodos.

Consejo de Estudio

Para dominar POO: 1) Practica creando clases que representen entidades del mundo real, 2) Experimenta con constructores y diferentes formas de inicializar objetos, 3) Crea programas que usen múltiples objetos interactuando entre sí, 4) Piensa en términos de objetos y sus responsabilidades. POO es un cambio de paradigma importante en la forma de pensar sobre el código.

Desarrollo de Software I

Unidad 07: Herencia, Encapsulación y
Manejo de Excepciones

Competencias y Resultados de Aprendizaje

Resultados de Aprendizaje:

- Comprender y aplicar encapsulación usando modificadores de acceso
- Implementar herencia entre clases
- Manejar excepciones usando try-catch-finally
- Crear jerarquías de clases con herencia
- Validar entrada de datos y manejar errores

Agenda de la Unidad

1. Encapsulación
2. Modificadores de acceso (private, public, protected)
3. Herencia entre clases
4. Clases base y derivadas
5. Manejo de excepciones (try-catch)
6. Bloque finally
7. Prácticas guiadas
8. Evaluación práctica
9. Mini-Quiz

Encapsulación

La **encapsulación** es el principio de ocultar los detalles internos de una clase y exponer solo lo necesario.

Ventajas:

- **Seguridad:** Protege datos de acceso no autorizado
- **Control:** Permite validar datos antes de asignarlos
- **Mantenibilidad:** Cambios internos no afectan código externo
- **Claridad:** Define claramente qué es público y qué es privado

Ejemplo:

```
class Persona
{
    private int edad; // Privado: solo accesible dentro de la clase

    public void SetEdad(int nuevaEdad)
    {
        if (nuevaEdad > 0 && nuevaEdad < 150)
        {
            edad = nuevaEdad; // Validación antes de asignar
        }
    }
}
```

Modificadores de Acceso

Los **modificadores de acceso** controlan desde dónde se puede acceder a miembros de una clase.

Modificador	Acceso	Descripción
<code>public</code>	Público	Accesible desde cualquier lugar
<code>private</code>	Privado	Solo accesible dentro de la clase
<code>protected</code>	Protegido	Accesible en la clase y clases derivadas

Ejemplo:

```
class Persona
{
    public string Nombre;          // Accesible desde cualquier lugar
    private int edad;              // Solo dentro de Persona
    protected string telefono;    // En Persona y clases derivadas

    public void MostrarInfo()
    {
        Console.WriteLine($"Nombre: {Nombre}, Edad: {edad}");
    }
}
```

Herencia

La **herencia** permite crear una nueva clase basada en una clase existente, heredando sus propiedades y métodos.

Sintaxis:

```
class ClaseBase
{
    // Propiedades y métodos
}

class ClaseDerivada : ClaseBase
{
    // Propiedades y métodos adicionales
}
```

Ejemplo:

```
// Clase base
class Persona
{
    public string Nombre;
    public int Edad;

    public void MostrarInfo()
    {
        Console.WriteLine($"Nombre: {Nombre}, Edad: {Edad}");
    }
}

// Clase derivada
class Estudiante : Persona
{
    public string Carrera;

    public void MostrarCarrera()
    {
        Console.WriteLine($"Carrera: {Carrera}");
    }
}
```

Herencia: Ejemplo Completo

```
// Clase base
class Persona
{
    protected string nombre;
    protected int edad;

    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public virtual void MostrarInfo()
    {
        Console.WriteLine($"Nombre: {nombre}, Edad: {edad}");
    }
}

// Clase derivada
class Estudiante : Persona
{
    private string carrera;

    public Estudiante(string nombre, int edad, string carrera)
        : base(nombre, edad)
    {
        this.carrera = carrera;
    }

    public override void MostrarInfo()
    {
        base.MostrarInfo();
        Console.WriteLine($"Carrera: {carrera}");
    }
}

// Uso
Estudiante est = new Estudiante("Juan", 20, "Ingeniería");
est.MostrarInfo(); // Muestra nombre, edad y carrera
```

Manejo de Excepciones: try-catch

Las **excepciones** son errores que ocurren durante la ejecución del programa. El manejo de excepciones permite controlar estos errores.

Sintaxis:

```
try
{
    // Código que puede generar error
}
catch (TipoExcepcion ex)
{
    // Código que maneja el error
}
```

Ejemplo:

```
try
{
    Console.Write("Ingrese un número: ");
    int numero = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine($"Número ingresado: {numero}");
}
catch (FormatException)
{
    Console.WriteLine("Error: Debe ingresar un número válido");
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

Bloque finally

El bloque `finally` se ejecuta siempre, sin importar si hubo una excepción o no.

Sintaxis:

```
try
{
    // Código que puede generar error
}
catch (Exception ex)
{
    // Manejo del error
}
finally
{
    // Código que siempre se ejecuta
}
```

Ejemplo:

```
try
{
    Console.Write("Ingrese un número: ");
    int numero = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine($"Número: {numero}");
}
catch (FormatException)
{
    Console.WriteLine("Error: Número inválido");
}
finally
{
    Console.WriteLine("Operación completada");
    // Este código siempre se ejecuta
}
```

Uso común: finally se usa para limpiar recursos (cerrar archivos, conexiones, etc.)

En la Vida Real: Manejo de Errores

Escenario: Programa de entrada de datos con validación

```
class Validador
{
    public static int LeerEntero(string mensaje)
    {
        int valor = 0;
        bool valido = false;

        while (!valido)
        {
            try
            {
                Console.WriteLine(mensaje);
                valor = Convert.ToInt32(Console.ReadLine());
                valido = true;
            }
            catch (FormatException)
            {
                Console.WriteLine("Error: Debe ingresar un número entero");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error inesperado: {ex.Message}");
            }
        }

        return valor;
    }

    // Uso
    int edad = Validador.LeerEntero("Ingrese su edad: ");
}
```

Errores Comunes

1. Acceder a miembros privados desde fuera

```
class Persona
{
    private int edad;
}

Persona p = new Persona();
p.edad = 25; // ✗ Error: edad es privado

// ✓ Solución: Usar propiedades públicas o métodos
public int Edad { get; set; }
```

2. No manejar excepciones

```
int numero = Convert.ToInt32(Console.ReadLine());
// ✗ Si el usuario ingresa texto, el programa crashea

try
{
    int numero = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException)
{
    Console.WriteLine("Error: número inválido");
}
// ✓ Maneja el error apropiadamente
```

Buenas Prácticas

- **Usar private para datos internos:** Proteger información sensible
- **Usar propiedades públicas:** Controlar acceso a datos
- **Herencia cuando hay relación "es-un":** Estudiante ES-UN Persona
- **Manejar excepciones específicas:** catch específico antes de general
- **Usar finally para limpiar:** Liberar recursos siempre

Ejemplo de Código Limpio:

```
class Persona
{
    private int edad; // Privado para protección

    public int Edad // Propiedad pública con validación
    {
        get { return edad; }
        set
        {
            if (value > 0 && value < 150)
                edad = value;
            else
                throw new ArgumentException("Edad inválida");
        }
    }
}
```

Práctica Guiada #1: Jerarquía con Herencia

Objetivo:

Crear una jerarquía de clases (Persona, Estudiante, Profesor) con herencia.

Instrucciones:

1. Crear clase base `Persona` con: Nombre, Edad
2. Crear clase `Estudiante` que herede de `Persona`, agregar: Carrera
3. Crear clase `Profesor` que herede de `Persona`, agregar: Materia
4. Crear constructores para cada clase
5. Crear método `MostrarInfo()` en cada clase
6. En Main, crear objetos y mostrar información

Solución Práctica Guiada #1

```
using System;

namespace JerarquiaClases
{
    // Clase base
    class Persona
    {
        public string Nombre;
        public int Edad;

        public Persona(string nombre, int edad)
        {
            Nombre = nombre;
            Edad = edad;
        }

        public virtual void MostrarInfo()
        {
            Console.WriteLine($"Nombre: {Nombre}, Edad: {Edad}");
        }
    }

    // Clase derivada
    class Estudiante : Persona
    {
        public string Carrera;

        public Estudiante(string nombre, int edad, string carrera)
            : base(nombre, edad)
        {
            Carrera = carrera;
        }

        public override void MostrarInfo()
        {
            base.MostrarInfo();
            Console.WriteLine($"Carrera: {Carrera}");
        }
    }

    // Clase derivada
    class Profesor : Persona
    {
        public string Materia;

        public Profesor(string nombre, int edad, string materia)
            : base(nombre, edad)
        {
            Materia = materia;
        }

        public override void MostrarInfo()
        {
            base.MostrarInfo();
            Console.WriteLine($"Materia: {Materia}");
        }
    }
}
```

```
}

class Program
{
    static void Main(string[] args)
    {
        Estudiante est = new Estudiante("Juan", 20, "Ingeniería");
        Profesor prof = new Profesor("María", 35, "Matemáticas");

        est.MostrarInfo();
        Console.WriteLine();
        prof.MostrarInfo();

        Console.ReadKey();
    }
}
```

Práctica Guiada #2: Manejo de Errores

Objetivo:

Implementar manejo de errores en un programa de entrada de datos.

Instrucciones:

1. Crear función `LeerEntero()` que valide entrada
2. Usar try-catch para manejar `FormatException`
3. Repetir hasta que se ingrese un número válido
4. Crear función `LeerDouble()` similar
5. Crear programa que use estas funciones

Solución Práctica Guiada #2

```
using System;

namespace ManejoErrores
{
    class Program
    {
        static void Main(string[] args)
        {
            int edad = LeerEntero("Ingrese su edad: ");
            double altura = LeerDouble("Ingrese su altura (m): ");

            Console.WriteLine($"\\nEdad: {edad} años");
            Console.WriteLine($"Altura: {altura:F2} metros");

            Console.ReadKey();
        }

        static int LeerEntero(string mensaje)
        {
            int valor = 0;
            bool valido = false;

            while (!valido)
            {
                try
                {
                    Console.Write(mensaje);
                    valor = Convert.ToInt32(Console.ReadLine());
                    valido = true;
                }
                catch (FormatException)
                {
                    Console.WriteLine("Error: Debe ingresar un n\\umer");
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error inesperado: {ex.Message}");
                }
            }

            return valor;
        }

        static double LeerDouble(string mensaje)
        {
            double valor = 0;
            bool valido = false;

            while (!valido)
            {
                try
                {
                    Console.Write(mensaje);
                    valor = Convert.ToDouble(Console.ReadLine());
                    valido = true;
                }
            }
        }
    }
}
```

```
        catch (FormatException)
        {
            Console.WriteLine("Error: Debe ingresar un numero");
        }
    }
    return valor;
}
}
```

Práctica Evaluada

Enunciado:

Crear un sistema de gestión académica que incluya:

1. Clase base `Persona` con encapsulación (propiedades privadas con `get/set`)
2. Clase `Estudiante` que herede de `Persona`, agregar: Promedio
3. Clase `Profesor` que herede de `Persona`, agregar: Salario
4. Validación en propiedades (edad entre 0-120, promedio entre 0-100)
5. Manejo de excepciones al ingresar datos
6. Métodos para mostrar información en cada clase

Criterios de Evaluación:

- **Encapsulación (30%):** Uso correcto de modificadores de acceso
- **Herencia (30%):** Jerarquía correcta de clases
- **Excepciones (20%):** Manejo adecuado de errores
- **Código (20%):** Organización y claridad

Rúbrica de Evaluación

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Encapsulación	Uso perfecto de private/public	Uso correcto	Uso básico	Uso incorrecto
Herencia	Jerarquía perfecta	Jerarquía correcta	Jerarquía básica	Sin herencia
Excepciones	Manejo completo	Manejo adecuado	Manejo básico	Sin manejo
Código	Excelente organización	Buena organización	Organización básica	Sin organización

Puntuación Total: 16 puntos

Mini-Quiz

1. ¿Qué modificador hace un miembro accesible solo dentro de la clase?

- a) public
- b) private
- c) protected
- d) internal

2. ¿Qué es la herencia?

- a) Crear objetos
- b) Crear una clase basada en otra
- c) Modificar clases
- d) Eliminar clases

3. ¿Qué palabra clave se usa para heredar en C#?

- a) extends
- b) inherits
- c) :
- d) inherits from

Mini-Quiz (Continuación)

4. ¿Qué bloque se ejecuta siempre, incluso si hay excepción?

- a) try
- b) catch
- c) finally
- d) throw

5. ¿Qué excepción se lanza cuando se ingresa texto en lugar de número?

- a) ArgumentException
- b) FormatException
- c) NullReferenceException
- d) IndexOutOfRangeException

6. ¿protected permite acceso en clases derivadas?

- a) Verdadero
- b) Falso

7. ¿Cuántas clases base puede tener una clase en C#?

- a) Una
- b) Múltiples
- c) Ninguna
- d) Infinitas

Mini-Quiz (Continuación)

8. ¿Qué palabra clave se usa para llamar al constructor de la clase base?

- a) super
- b) base
- c) parent
- d) this

9. ¿Puedes tener múltiples bloques catch?

- a) Verdadero
- b) Falso

10. ¿Qué es la encapsulación?

- a) Ocultar detalles internos
- b) Crear objetos
- c) Heredar clases
- d) Manejar errores

Respuestas del Mini-Quiz

- 1. b) private** - private solo permite acceso dentro de la clase.
- 2. b) Crear una clase basada en otra** - La herencia crea clases derivadas.
- 3. c) :** - Se usa : para heredar en C#.
- 4. c) finally** - finally siempre se ejecuta.
- 5. b) FormatException** - FormatException para formato inválido.
- 6. a) Verdadero** - protected permite acceso en derivadas.
- 7. a) Una** - C# solo permite herencia simple.
- 8. b) base** - base llama al constructor base.
- 9. a) Verdadero** - Puedes tener múltiples catch.
- 10. a) Ocultar detalles internos** - Encapsulación oculta detalles.

Resumen de la Unidad

- ✓ La encapsulación protege datos usando modificadores de acceso
- ✓ private: solo dentro de la clase, public: desde cualquier lugar
- ✓ protected: accesible en la clase y clases derivadas
- ✓ La herencia permite crear clases basadas en otras
- ✓ Se usa : para heredar en C#
- ✓ base llama al constructor de la clase base
- ✓ try-catch maneja excepciones
- ✓ finally siempre se ejecuta

Próxima Unidad: Proyecto Final - Integraremos todos los conocimientos adquiridos en un proyecto funcional.

Tarea para Casa

Actividad:

Diseñar el esquema del proyecto final y preparar una presentación preliminar.

Entregables:

1. **Documento de Diseño (3-4 páginas)** con:

- Descripción del proyecto
- Clases que se utilizarán
- Propiedades y métodos de cada clase
- Estructura del programa principal
- Funcionalidades principales

2. **Presentación Preliminar (5-10 diapositivas)** con:

- Objetivo del proyecto
- Funcionalidades principales
- Estructura de clases
- Cronograma de desarrollo

Fecha de Entrega:

Una semana después de esta clase

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre esta unidad, visita nuestra página de recursos con:

- Videos sobre herencia y polimorfismo
- Ejemplos de clases abstractas
- Interfaces en C#
- Manejo de excepciones avanzado

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 07

Herencia, Encapsulación y Manejo de Excepciones - Enlaces, videos y material complementario



Documentación Oficial

[Herencia en C# Oficial](#)

Documentación completa sobre herencia, cómo crear clases derivadas y usar la palabra clave base.

[Encapsulación en C# Oficial](#)

Guía sobre modificadores de acceso (public, private, protected) y cómo implementar encapsulación.

[Manejo de Excepciones Oficial](#)

Documentación sobre try-catch-finally, tipos de excepciones y mejores prácticas para manejo de errores.



Videos Educativos

[Herencia y Polimorfismo en C# Video](#)

Tutorial sobre cómo crear jerarquías de clases y usar herencia para reutilizar código.

[Manejo de Excepciones en C# Tutorial](#)

Guía práctica sobre cómo manejar errores y excepciones en tus programas.

Consejo de Estudio

Para dominar estos conceptos avanzados: 1) Practica creando jerarquías de clases con herencia, 2) Experimenta con diferentes modificadores de acceso para entender la encapsulación, 3) Escribe código que pueda generar errores y aprende a manejarlos correctamente, 4) Crea programas que combinen herencia, encapsulación y manejo de excepciones. Estos son conceptos fundamentales para escribir código robusto y mantenable.

Desarrollo de Software I

Unidad 08: Proyecto Final

Objetivo del Proyecto Final

Integrar todos los conocimientos adquiridos durante el curso en un proyecto funcional y completo.

Competencias a Demostrar:

- Uso correcto de estructuras de control (if, switch, bucles)
- Implementación de funciones y métodos
- Aplicación de Programación Orientada a Objetos
- Uso de herencia y encapsulación
- Manejo de excepciones
- Organización y documentación del código

Ejemplos de Proyectos

1. Sistema de Gestión de Estudiantes

- Registrar estudiantes (nombre, edad, carrera, calificaciones)
- Calcular promedios
- Buscar estudiantes
- Mostrar reportes

2. Simulador de Tienda

- Gestión de productos (agregar, buscar, modificar)
- Carrito de compras
- Cálculo de totales y descuentos
- Generación de facturas

3. Calculadora Avanzada

- Operaciones básicas y avanzadas
- Historial de operaciones
- Funciones matemáticas
- Interfaz de menú

Más Ejemplos de Proyectos

4. Agenda de Contactos

- Agregar, editar, eliminar contactos
- Buscar contactos
- Agrupar por categorías
- Exportar información

5. Sistema de Biblioteca

- Gestión de libros
- Préstamos y devoluciones
- Búsqueda de libros
- Reportes de préstamos

Nota: Puedes proponer tu propio proyecto, pero debe ser aprobado por el profesor.

Estructura del Proyecto

Componentes Mínimos Requeridos:

1. **Clases:** Al menos 3 clases con herencia
2. **Propiedades:** Con encapsulación (private/public)
3. **Métodos:** Funciones para operaciones principales
4. **Constructores:** Para inicializar objetos
5. **Manejo de Excepciones:** try-catch en entrada de datos
6. **Menú Interactivo:** switch para navegación
7. **Bucles:** Para procesar colecciones

Ejemplo de Estructura:

```
Proyecto/
└── Clases/
    ├── ClaseBase.cs
    ├── ClaseDerivada1.cs
    └── ClaseDerivada2.cs
└── Program.cs (Main)
└── README.md (Documentación)
```

Criterios de Evaluación

1. Funcionalidad del Programa (30%)

- El programa ejecuta sin errores
- Todas las funcionalidades principales funcionan
- Manejo adecuado de casos límite

2. Uso Correcto de Conceptos (30%)

- Estructuras de control bien implementadas
- Funciones y métodos utilizados apropiadamente
- POO aplicado correctamente (clases, objetos, herencia)

3. Documentación del Código (20%)

- Comentarios explicativos
- Nombres descriptivos
- README con instrucciones

4. Creatividad e Innovación (20%)

- Funcionalidades adicionales
- Interfaz de usuario mejorada
- Solución creativa a problemas

Rúbrica Detallada

Criterio	Excelente (4)	Bueno (3)	Regular (2)	Insuficiente (1)
Funcionalidad	Todas las funciones perfectas	Mayoría funciona bien	Funciones básicas	No funciona
Estructuras	Uso perfecto	Uso correcto	Uso básico	Uso incorrecto
POO	Excelente diseño	Buen diseño	Diseño básico	Sin POO
Herencia	Jerarquía perfecta	Jerarquía correcta	Herencia básica	Sin herencia
Excepciones	Manejo completo	Manejo adecuado	Manejo básico	Sin manejo
Documentación	Excelente	Buena	Básica	Sin documentación
Creatividad	Muy creativo	Creativo	Básico	Sin creatividad

Puntuación Total: 28 puntos

Plan de Desarrollo

Fase 1: Diseño (Semana 1)

- Definir funcionalidades del proyecto
- Diseñar estructura de clases
- Crear diagrama de clases
- Planificar interfaz de usuario

Fase 2: Implementación Base (Semana 2)

- Crear clases base
- Implementar propiedades y métodos básicos
- Crear menú principal

Fase 3: Funcionalidades (Semana 3)

- Implementar funcionalidades principales
- Agregar manejo de excepciones
- Probar y corregir errores

Fase 4: Pulido (Semana 4)

- Documentar código
- Mejorar interfaz
- Preparar presentación

Guía de Implementación

Paso 1: Crear Estructura Base

```
// Clase base
class Entidad
{
    protected int id;
    protected string nombre;

    public Entidad(int id, string nombre)
    {
        this.id = id;
        this.nombre = nombre;
    }

    public virtual void MostrarInfo()
    {
        Console.WriteLine($"ID: {id}, Nombre: {nombre}");
    }
}
```

Paso 2: Crear Clases Derivadas

```
class Estudiante : Entidad
{
    private double promedio;

    public Estudiante(int id, string nombre, double promedio)
        : base(id, nombre)
    {
        this.promedio = promedio;
    }

    public override void MostrarInfo()
    {
        base.MostrarInfo();
        Console.WriteLine($"Promedio: {promedio}");
    }
}
```

Guía de Implementación (Continuación)

Paso 3: Crear Menú Principal

```
static void MostrarMenu()
{
    Console.WriteLine("==> MENÚ PRINCIPAL ==<");
    Console.WriteLine("1. Agregar");
    Console.WriteLine("2. Buscar");
    Console.WriteLine("3. Mostrar todos");
    Console.WriteLine("4. Salir");
}

static void Main(string[] args)
{
    int opcion;
    do
    {
        MostrarMenu();
        opcion = LeerEntero("Seleccione opción: ");

        switch (opcion)
        {
            case 1: Agregar(); break;
            case 2: Buscar(); break;
            case 3: MostrarTodos(); break;
            case 4: Console.WriteLine("¡Hasta luego!"); break;
            default: Console.WriteLine("Opción inválida"); break;
        }
    } while (opcion != 4);
}
```

Buenas Prácticas para el Proyecto

- **Planificar antes de codificar:** Diseña primero, codifica después
- **Desarrollar incrementalmente:** Una funcionalidad a la vez
- **Probar constantemente:** Prueba cada función antes de continuar
- **Documentar mientras desarrollas:** No dejes la documentación para el final
- **Usar control de versiones:** Git para rastrear cambios
- **Manejar errores apropiadamente:** Validar entrada siempre
- **Mantener código limpio:** Nombres descriptivos, funciones pequeñas

Consejo: Empieza con funcionalidades básicas y luego agrega características avanzadas.

Checklist de Entrega

Antes de Entregar, Verifica:

- ✓ El programa compila sin errores
- ✓ Todas las funcionalidades principales funcionan
- ✓ Hay al menos 3 clases con herencia
- ✓ Se usa encapsulación (private/public)
- ✓ Se manejan excepciones en entrada de datos
- ✓ El código está comentado
- ✓ Los nombres son descriptivos
- ✓ Hay un README con instrucciones
- ✓ El proyecto está organizado en carpetas
- ✓ Se incluye un archivo .sln o proyecto completo

Archivos a Entregar:

- Código fuente (.cs)
- README.md con descripción e instrucciones
- Diagrama de clases (opcional pero recomendado)
- Presentación del proyecto (5-10 diapositivas)

Presentación del Proyecto

Estructura de la Presentación:

1. **Portada:** Nombre del proyecto, autor, fecha
2. **Objetivo:** Qué problema resuelve el proyecto
3. **Funcionalidades:** Lista de características principales
4. **Arquitectura:** Diagrama de clases y estructura
5. **Demostración:** Ejecución en vivo del programa
6. **Desafíos:** Problemas encontrados y cómo se resolvieron
7. **Aprendizajes:** Qué aprendiste durante el desarrollo
8. **Mejoras Futuras:** Qué agregarías si tuvieras más tiempo

Duración:

10-15 minutos de presentación + 5 minutos de preguntas

Recursos y Ayuda

Documentación Oficial:

- Microsoft Learn - C#:
<https://learn.microsoft.com/dotnet/csharp/>
- Documentación de C#:
<https://learn.microsoft.com/dotnet/csharp/language-reference/>

Comunidades:

- Stack Overflow:
<https://stackoverflow.com/questions/tagged/c%23>
- Reddit r/csharp: <https://www.reddit.com/r/csharp/>
- C# Corner: <https://www.c-sharpcorner.com/>

Ayuda del Profesor:

- Horarios de consulta: [Definir con el profesor]
- Email: [Email del profesor]
- Revisión de código: Solicitar con anticipación

Cronograma Sugerido

Semana	Actividad	Entregable
1	Diseño y planificación	Documento de diseño
2	Implementación base	Clases y estructura básica
3	Funcionalidades principales	Programa funcional
4	Pulido y documentación	Proyecto completo
5	Presentación	Demostración en clase

Importante: No dejes todo para el final. Desarrolla de forma constante durante las semanas asignadas.

Ejemplo de README

```
# Sistema de Gestión de Estudiantes

## Descripción
Sistema de consola para gestionar información de estudiantes, calcular promedios y generar reportes.

## Funcionalidades
- Agregar estudiantes
- Buscar estudiantes
- Calcular promedios
- Mostrar reportes

## Requisitos
- .NET 6.0 o superior
- Visual Studio 2022 o VS Code

## Instalación
1. Clonar el repositorio
2. Abrir en Visual Studio
3. Compilar y ejecutar

## Uso
Ejecutar el programa y seguir el menú interactivo.

## Autor
[Tu Nombre]

## Fecha
[Fecha de entrega]
```

Consejos Finales

- ✓ **Empieza temprano:** No esperes hasta la última semana
- ✓ **Prueba frecuentemente:** Verifica que cada parte funciona
- ✓ **Pide ayuda cuando la necesites:** El profesor está para ayudarte
- ✓ **Documenta mientras desarrollas:** Es más fácil que al final
- ✓ **Mantén el código organizado:** Facilita el mantenimiento
- ✓ **Practica la presentación:** Prepárate para demostrar tu trabajo
- ✓ **Sé creativo:** Agrega funcionalidades que te interesen
- ✓ **Disfruta el proceso:** Aprender haciendo es la mejor forma

Resumen del Proyecto Final

Objetivo Principal:

Demostrar dominio de todos los conceptos aprendidos durante el curso mediante un proyecto funcional y completo.

Componentes Clave:

- Estructuras de control bien implementadas
- Funciones y métodos reutilizables
- Programación Orientada a Objetos
- Herencia y encapsulación
- Manejo de excepciones
- Código documentado y organizado

Evaluación:

Funcionalidad (30%) + Conceptos (30%) + Documentación (20%) + Creatividad (20%) = 100%

¡Éxito en tu Proyecto!

¡Has llegado al final del curso!

Ahora es momento de demostrar todo lo que has aprendido.

Confía en tus habilidades y desarrolla un proyecto del que te sientas orgulloso.

¡Mucho éxito en tu proyecto final!

Recursos Adicionales



Material Complementario

Para ampliar tu conocimiento sobre proyectos finales y desarrollo de software, visita nuestra página de recursos con:

- Guías de mejores prácticas
- Ejemplos de proyectos completos
- Recursos de documentación
- Herramientas de desarrollo

[Ver Recursos Adicionales →](#)

Recursos Adicionales - Unidad 08

Proyecto Final - Enlaces, videos y material complementario



Recursos de Documentación

[Documentación Completa de C# Referencia](#)

Referencia completa del lenguaje C# para consultar durante el desarrollo de tu proyecto.

[Convenciones de Código en C# Guía](#)

Guía de estilo y mejores prácticas para escribir código limpio y mantenable en C#.



Ideas de Proyectos

[Ejemplos de Proyectos en C# GitHub](#)

Repositorio con ejemplos de proyectos completos que pueden servir de inspiración.



Herramientas Útiles

Visual Studio IDE

IDE completo con todas las herramientas necesarias para desarrollar tu proyecto.

GitHub Control de Versiones

Plataforma para alojar y versionar tu código del proyecto final.



Consejo para el Proyecto Final

Para tener éxito en tu proyecto final: 1) Planifica bien antes de empezar a codificar, 2) Divide el proyecto en tareas pequeñas y manejables, 3) Usa todas las técnicas aprendidas: POO, herencia, manejo de excepciones, 4) Documenta tu código con comentarios claros, 5) Prueba cada funcionalidad a medida que la desarrollas, 6) No tengas miedo de refactorizar si encuentras una mejor solución. ¡Éxito en tu proyecto!