UNIVERSIDAD DE ORIENTE (UNIVO)

Programación Orientada a Eventos (POE)

Ing. Karla Cecilia Cruz de González

Ariel Neftali Argueta Rodríguez

06 de septiembre de 2025

Ejercicio 1

Explique la diferencia entre int.Parse( ), int.TryParse( ) y TryParse( ) ¿Cuál es más recomendable usar para validar entrada del usuario y por qué? Proporcione un ejemplo de cada uno y explique qué sucede cuando se ingresa un valor inválido.

int.Parse()

- Convierte un string en un int.
- No valida: si el valor no se puede convertir, lanza una excepción (FormatException).

```
string input = "123";
int numero = int.Parse(input);
Console.WriteLine(numero); // 123

string inputbad = "abc";
int numerobad = int.Parse(inputbad); // <-- Lanza FormatException
```

int.TryParse()

- Intenta convertir un string a int de forma segura.
- No lanza excepción.
- Devuelve true si pudo convertir, false si no.
- Usa un parámetro out para guardar el resultado.

```
string input = "456";
int numero;

bool exito = int.TryParse(input, out numero);
if (exito)
    Console.WriteLine("Número válido: " + numero);
else
    Console.WriteLine("Entrada inválida");

string inputbad = "abc";
int numerobad;
bool exitobad = int.TryParse(inputbad, out numerobad);
Console.WriteLine(exitobad); // false
Console.WriteLine(numerobad); // 0 (valor por defecto)
```

TryParse en general

- TryParse es un patrón usado en muchos tipos, no solo int.
- Ejemplos: double.TryParse(), DateTime.TryParse(), decimal.TryParse().
- Funciona igual: devuelve true si pudo convertir y false si no, sin lanzar excepciones.

```
string input = "7,5";
double nota;
if (double.TryParse(input, out nota))
    Console.WriteLine("Nota válida: " + nota);
else
    Console.WriteLine("Entrada inválida");
```

¿Cual usar?

Para entrada de usuario, siempre es recomendable usar TryParse().

- Razones:
    1. Evita que el programa se caiga con datos inválidos.
    2. Permite mostrar mensajes amigables y repetir la entrada hasta que sea correcta.
    3. Compatible con todo tipo de datos (int, double, decimal, DateTime…).

Ejercicio 2

Explique: En este ejercicio del inventario, se utiliza una clase Producto con propiedades y un método calculado. Explique:

• ¿Qué son las propiedades automáticas en C#?

Propiedades automáticas en C#

- Son una forma rápida de definir propiedades sin tener que escribir un campo privado explícito.
- Sintaxis:

```
public class Producto
{
    // Propiedades automáticas
    4 references
    public string Nombre { get; set; }
    4 references
    public double Precio { get; set; }
    4 references
    public int Cantidad { get; set; }

    // Propiedad calculada
    3 references
    public double ValorTotal => Precio * Cantidad;
}
```

- El compilador crea un campo privado oculto para almacenar el valor de la propiedad.
- Permite acceder a la propiedad como si fuera un campo, pero con control adicional si se desea.

• ¿Cuál es la diferencia entre un campo y una propiedad?

Concepto	Campo	Propiedad
Definición	Variable dentro de la clase	"Envoltorio" que controla acceso a un campo
Ejemplo	private double precio;	public double Precio { get; set; }
Acceso	Directo (producto.precio = 10;)	A través de get/set (producto.Precio = 10;)
Control	Ninguno, acceso directo	Puedes controlar lectura/escritura, validaciones, cálculos
Seguridad	Baja, no hay validación	Alta, puedes limitar escritura o lectura

• ¿Qué ventajas tiene usar get y set en las propiedades?

1. Control de acceso: Puedes hacer que algunas propiedades sean de solo lectura (get) o de solo escritura (set).
2. Validación de datos: Antes de asignar un valor, puedes chequearlo. Ejemplo:

```
private double precio;
0 references
public double Precio
{
    get { return precio; }
    set
    {
        if (value >= 0) precio = value;
        else throw new Exception("El precio no puede ser negativo");
    }
}
```

3. Encapsulamiento: Oculta la implementación interna del campo.
4. Compatibilidad futura: Si quieres cambiar cómo se almacena el dato, no afecta al código que usa la propiedad.

• Explique cómo funciona la propiedad calculada ValorTotal => Precio * Cantidad

Una propiedad calculada no almacena un valor, sino que lo devuelve en base a otros campos o propiedades.

- Ejemplo:

```
public double ValorTotal => Precio * Cantidad;
```

- Cada vez que accedes a ValorTotal, se hace la multiplicación de Precio por Cantidad en tiempo real.
- Ventaja: no hay que actualizar manualmente el valor cuando cambian Precio o Cantidad.

```
Producto p = new Producto { Precio = 5, Cantidad = 10 };
Console.WriteLine(p.ValorTotal); // 50
p.Cantidad = 12;
Console.WriteLine(p.ValorTotal); // 60 automáticamente
```

Ejercicio 3

Explique:

• ¿Cómo funciona el operador módulo en C#?

En C#, el operador módulo **%** devuelve el resto de la división entre dos números.

```
int a = 17;
int b = 5;
int restomodular = a % b; // resto = 2
```

- Aquí 17 % 5 devuelve 2 porque 17 dividido entre 5 da 3 como cociente y 2 como resto.

Regla clave:

- a % b siempre es menor que b y mayor o igual a 0 si a es positivo.

• ¿Por qué es útil para separar unidades, decenas y centenas?

El módulo permite extraer cifras específicas de un número:

- Unidades: numero % 10 → da la última cifra.
- Decenas: (numero / 10) % 10 → divide entre 10 para eliminar la unidad y luego módulo 10 para obtener la decena.
- Centenas: (numero / 100) % 10 → divide entre 100 para eliminar unidades y decenas, luego módulo 10 para obtener la centena.

```
int numero = 157;
int unidades = numero % 10;         // 7
int decenas = (numero / 10) % 10;   // 5
int centenas = (numero / 100) % 10; // 1
```

Esto permite descomponer cualquier número en sus cifras fácilmente.

• Describa el algoritmo usado para convertir 157 a "ciento cincuenta y siete"

Se puede hacer usando **división y módulo** para separar centenas, decenas y unidades:

1.

```
int centenas = numero / 100;      // 1
int decenas = (numero / 10) % 10; // 5
int unidades = numero % 10;       // 7
```

2.

```
string[] palabrasCentenas = { "", "ciento", "doscientos", "trescientos", "cuatrocientos", "quinientos", "seiscientos", "setecientos", "ochocientos", "novecientos" };
string[] palabrasDecenas = { "", "diez", "veinte", "treinta", "cuarenta", "cincuenta", "sesenta", "setenta", "ochenta", "noventa" };
string[] palabrasUnidades = { "", "uno", "dos", "tres", "cuatro", "cinco", "seis", "siete", "ocho", "nueve" };
```

3. Combinar las palabras según las reglas del idioma:

- Centenas: "ciento"
- Decenas y unidades: "cincuenta y siete"

Resultado: "ciento cincuenta y siete"

Nota: Para números exactos como 100, 200, etc., se aplican excepciones (cien, doscientos, etc.).

• ¿Qué otros problemas se pueden resolver usando aritmética modular?

El operador % es muy útil en muchos contextos:

1. Días de la semana:

```
int dia = (hoy + 3) % 7; // Para calcular el día de la semana dentro de 0 a 6
```

2. Relojes y horas:

```
int hora = (horaActual + 5) % 24; // Mantener hora en formato 0-23
```

3. Detección de números pares o impares:

```
if (numero % 2 == 0) { /* par */ } else { /* impar */ }
```

4. Rotación circular de arrays o buffers circulares.

5. Criptografía básica y códigos de verificación (checksum, contraseñas, etc.).

En general, modular es clave siempre que se necesite trabajar con ciclos o restos de división.

Ejercicio 4

Explique:

• ¿Cuál es la diferencia entre tipos por valor y tipos por referencia?

Tipos por valor

- Ejemplos: int, double, bool, struct.
- Se almacenan directamente en la pila (stack).
- Cuando asignas o pasas una variable por valor, se copia el contenido, no la referencia.
- Cambiar la copia no afecta al original.

```
int a = 5;
int b = a; // se copia el valor
b = 10;
Console.WriteLine(a); // imprime 5, a no cambia
```

Tipos por referencia

- Ejemplos: class, string, array.
- Se almacenan en el heap, y la variable contiene una referencia al objeto.
- Cambiar el objeto a través de una referencia afecta al original.

```
int[] numeros = { 1, 2, 3 };
int[] copia = numeros; // referencia al mismo array
copia[0] = 99;
Console.WriteLine(numeros[0]); // imprime 99
```

• ¿Por qué usar decimal para precios en lugar de double?

- double y float son imprecisos con números decimales debido a su representación binaria.
- decimal tiene mayor precisión decimal y es más exacto para cálculos financieros, evitando errores de redondeo.

```
double precioDouble = 0.1 + 0.2;   // 0.30000000000000004
decimal precioDecimal = 0.1m + 0.2m; // 0.3 exacto
```

• ¿Qué sucede en memoria cuando se crea un array de 1000 elementos?

```
int[] numeros = new int[1000];
```

- Se crea un objeto en el heap que puede almacenar 1000 int.
- La variable numeros es una referencia que se guarda en la pila (stack).
- Cada elemento se inicializa con el valor por defecto (0 para int).

Visualmente seria algo así.

```
// Stack: numeros->referencia a heap
// Heap: [0, 0, 0, ..., 0](1000 elementos)
```

- El heap permite almacenar grandes cantidades de datos que no caben en la pila.

• ¿Cuándo se libera la memoria ocupada por las variables locales?

- Las variables locales por valor (p. ej. int, double) se liberan automáticamente al salir del método porque estaban en la pila.
- Las variables por referencia (objetos en heap) se liberan cuando el recolector de basura (Garbage Collector) detecta que no hay referencias apuntando a ellas.

Basicamente:

- Stack: memoria liberada automáticamente al terminar el método.
- Heap: liberada por Garbage Collector cuando el objeto ya no es accesible.

Ejercicio 5

Responda:

• ¿Qué son las enumeraciones en C# y cuándo usarlas?

Enumeraciones (enum) son tipos de valor que permiten definir un conjunto de constantes con nombre para un tipo de dato, generalmente int.
Se usan para mejorar la legibilidad y seguridad del código, evitando usar números "mágicos".

```
enum DiasSemana
    {
        Lunes,
        Martes,
        Miercoles,
        Jueves,
        Viernes,
        Sabado,
        Domingo
    }

    DiasSemana hoy = DiasSemana.Miercoles;
```

- Aquí Lunes = 0, Martes = 1, etc., pero usamos nombres claros.
- Cuándo usar:
    - Cuando tienes un conjunto limitado de valores posibles.
    - Para mejorar legibilidad y evitar errores con valores fuera de rango.
    - Ejemplos: días de la semana, estados de un pedido (Pendiente, Enviado, Entregado), niveles de acceso (Admin, Usuario, Invitado).

• ¿Cuál es la diferencia entre switch statement tradicional y switch expression?

Switch tradicional:

- Sintaxis clásica, con case y break.
- Se usa para ejecutar bloques de código según el valor.

```
int dia = 3;
string nombreDia;

switch (dia)
{
    case 1: nombreDia = "Lunes"; break;
    case 2: nombreDia = "Martes"; break;
    case 3: nombreDia = "Miércoles"; break;
    default: nombreDia = "Desconocido"; break;
}
```

Switch expression (C# 8.0+):

- Más concisa y retorna un valor directamente.
- Usa => y switch como expresión.

```
int dia = 3;

string nombreDia = dia switch
{
    1 => "Lunes",
    2 => "Martes",
    3 => "Miércoles",
    _ => "Desconocido"
};
```

Característica	Switch tradicional	Switch expression
Sintaxis	Verbosa con case y break	Concisa con =>
Retorno de valor	Necesita variable externa	Retorna directamente
Uso	Ejecutar múltiples líneas por caso	Generalmente expresiones simples
Compatibilidad	Cualquier versión de C#	C# 8.0 en adelante

Preguntas finales:

• ¿Cuál es la diferencia entre tipos por valor y tipos por referencia?

Tipos por valor

- Ejemplos: int, double, bool, struct.
- Se almacenan directamente en la pila (stack).
- Cuando asignas o pasas una variable por valor, se copia el contenido, no la referencia.
- Cambiar la copia no afecta al original.

```
int a = 5;
int b = a; // se copia el valor
b = 10;
Console.WriteLine(a); // imprime 5, a no cambia
```

Tipos por referencia

- Ejemplos: class, string, array.
- Se almacenan en el heap, y la variable contiene una referencia al objeto.
- Cambiar el objeto a través de una referencia afecta al original.

```
int[] numeros = { 1, 2, 3 };
int[] copia = numeros; // referencia al mismo array
copia[0] = 99;
Console.WriteLine(numeros[0]); // imprime 99
```

• ¿Por qué usar decimal para precios en lugar de double?

- double y float son imprecisos con números decimales debido a su representación binaria.
- decimal tiene mayor precisión decimal y es más exacto para cálculos financieros, evitando errores de redondeo.

```
double precioDouble = 0.1 + 0.2;   // 0.30000000000000004
decimal precioDecimal = 0.1m + 0.2m; // 0.3 exacto
```

• ¿Qué sucede en memoria cuando se crea un array de 1000 elementos?

```
int[] numeros = new int[1000];
```

- Se crea un objeto en el heap que puede almacenar 1000 int.
- La variable numeros es una referencia que se guarda en la pila (stack).
- Cada elemento se inicializa con el valor por defecto (0 para int).

Visualmente seria algo así.

```
// Stack: numeros->referencia a heap
// Heap: [0, 0, 0, ..., 0](1000 elementos)
```

- El heap permite almacenar grandes cantidades de datos que no caben en la pila.

• ¿Cuándo se libera la memoria ocupada por las variables locales?

- Las variables locales por valor (p. ej. int, double) se liberan automáticamente al salir del método porque estaban en la pila.
- Las variables por referencia (objetos en heap) se liberan cuando el recolector de basura (Garbage Collector) detecta que no hay referencias apuntando a ellas.

Basicamente:

- Stack: memoria liberada automáticamente al terminar el método.
- Heap: liberada por Garbage Collector cuando el objeto ya no es accesible.

• ¿Qué significa que un método sea estático?

- Un método estático pertenece a la clase, no a una instancia.
- Se puede llamar sin crear un objeto.

```
class Matematica
{

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

int resultado = Matematica.Sumar(3, 5);
```

• ¿Cuándo es apropiado usar métodos estáticos vs. métodos de instancia?

- Métodos estáticos:
    - Operaciones que no dependen de datos de un objeto.
    - Utiles para funciones auxiliares o utilidades (Math.Sqrt, Console.WriteLine).
- Métodos de instancia:
    - Dependen del estado interno del objeto.
    - Se necesita una instancia para llamarlos.

• ¿Por qué Main() debe ser estático?

- Main() es el punto de entrada del programa.
- No hay objetos creados al inicio, por eso debe ser estático para que la ejecución pueda comenzar directamente desde la clase.

• ¿Qué limitaciones tienen los métodos estáticos?

1. No pueden acceder a miembros de instancia directamente.
2. No pueden usar this porque no hay objeto asociado.
3. Solo pueden llamar a otros métodos estáticos a menos que creen una instancia.