Parte 1: Guía de Estudio Teórica

Aquí tienes los temas fundamentales, con un énfasis especial en la modularidad.

Tema 1: Fundamentos de C# y Programación Estructurada

Para empezar, es crucial entender la estructura básica de un programa en C# y cómo se manejan los datos.

- **Estructura de una Aplicación de Consola:** Toda aplicación en C# consiste en al menos una declaración de clase. La ejecución comienza en el método Main. Una estructura mínima en C# se ve así:
- using System; // Directiva para usar clases del espacio de nombres System, como Console.
-
- public class MiPrograma // Declaración de una clase.
- {
- // El método Main es el punto de inicio de la aplicación.
- public static void Main(string[] args)
- {
- // Aquí va el código que se ejecutará.
- Console.WriteLine("¡Hola, Mundo!"); // Muestra texto en la consola.
- }
- }
- **Sintaxis Básica y Variables:**
    - **Instrucciones:** Cada acción que define el programa es una instrucción y debe terminar con un punto y coma (;). Omitirlo es un error de sintaxis.
    - **Comentarios:** Se usan para documentar el código y son ignorados por el compilador. Pueden ser de una sola línea (//) o multilínea (/* ... */).
    - **Variables:** Son ubicaciones en la memoria para almacenar datos. Se deben declarar especificando su tipo y nombre (ej. int numero;). Es una buena práctica declarar cada variable en una línea separada y usar nombres significativos. Los nombres de variables suelen seguir el estilo *camelCase*

(la primera letra en minúscula y las siguientes palabras con mayúscula inicial).

    - **Tipos de Datos Simples:** int para enteros, double o decimal para números con punto decimal (siendo decimal más preciso para cálculos monetarios), y char para caracteres individuales.

- **Entrada y Salida de Datos:**
    - Para mostrar datos en la consola se usan Console.Write() (no agrega un salto de línea) y Console.WriteLine() (agrega un salto de línea al final).
    - Para leer datos introducidos por el usuario desde el teclado, se utiliza Console.ReadLine(), que devuelve una cadena de caracteres (string).
    - Para convertir esa cadena a un tipo numérico, se usa la clase Convert, por ejemplo: numero = Convert.ToInt32(Console.ReadLine());.
- **Operadores Aritméticos y de Decisión:**
    - **Aritméticos:** Incluyen suma (+), resta (-), multiplicación (*), división (/) y residuo (%). C# respeta una **precedencia de operadores** (primero *, /, %, y luego +, -) que se puede alterar con paréntesis ().
    - **Relacionales y de Igualdad:** Se usan para formar condiciones que pueden ser verdaderas o falsas. Incluyen == (igual a), != (no igual a), < (menor que), > (mayor que), <= (menor o igual que) y >= (mayor o igual que).

Tema 2: Control de Flujo (Selección e Iteración)

El control de flujo permite que tu programa tome decisiones y repita acciones.

- **Instrucciones de Selección:**
    - **if:** Ejecuta un bloque de código solo si una condición es verdadera (true).
    - **if-else:** Proporciona un bloque de código alternativo que se ejecuta si la condición es falsa (false).
    - **if-else if-else:** Permite evaluar múltiples condiciones en secuencia.
    - **switch:** Es ideal cuando se compara una sola variable contra múltiples valores constantes. Es más legible y, a veces, más eficiente que una cadena larga de if-else if. La expresión en un switch puede ser de tipo int, char, string, entre otros.

- **Instrucciones de Iteración (Bucles):** Permiten ejecutar un bloque de código repetidamente.
    - **while:** Repite un bloque de código mientras una condición sea verdadera. La condición se evalúa *antes* de cada iteración.
    - **do-while:** Similar a while, pero la condición se evalúa *después* de cada iteración. Esto garantiza que el bloque de código se ejecute al menos una vez.
    - **for:** Es perfecto cuando se conoce de antemano el número de iteraciones. Consta de tres partes: un inicializador, una condición y un iterador.
    - **foreach:** Se utiliza para recorrer todos los elementos de una colección (como un arreglo o una lista) sin necesidad de manejar índices.
- **Instrucciones de Salto:**
    - **break:** Termina inmediatamente la ejecución de un bucle (for, while, etc.) o de una instrucción switch.
    - **continue:** Salta el resto de la iteración actual y pasa a la siguiente.

Tema 3: Programación Modular (¡Tu Enfoque Principal!)

La modularidad consiste en **descomponer un problema grande en trozos más pequeños y manejables**, llamados funciones o métodos. Esto hace el código más fácil de leer, reutilizar y depurar.

- **Funciones y Procedimientos:**
    - **Función:** Es un bloque de código que realiza una tarea específica y **devuelve un valor**. Se declara especificando el tipo de dato que retornará (ej. int, string, double).
    - **Procedimiento (o método void):** Es similar a una función, pero **no devuelve ningún valor**. Se declara usando la palabra clave void como tipo de retorno.
- **Sintaxis de un Método:**
- <modificadores> <tipo_retorno> <NombreMetodo>(<lista_parametros>)
- {
- // Cuerpo del método
- return valor; // Si no es void

- }
    - public: El método es accesible desde cualquier otra clase.
    - static: El método pertenece a la clase misma, no a una instancia. Se puede llamar directamente usando el nombre de la clase (ej. MiClase.MiMetodo()).
- **Parámetros de un Método:** Son variables que permiten pasar información al método.
    - **Parámetros por Valor (predeterminado):** Se pasa una *copia* del valor de la variable. Cualquier cambio dentro del método no afecta a la variable original fuera de él.
    - **Parámetros por Referencia (ref y out):**
        - ref: Se pasa la *referencia* de memoria de la variable. Los cambios dentro del método **sí afectan** a la variable original. La variable debe estar inicializada antes de ser pasada.
        - out: Similar a ref, pero se usa cuando el propósito principal es que el método devuelva uno o más valores a través de sus parámetros. El método está obligado a asignar un valor al parámetro out antes de terminar. La variable no necesita estar inicializada previamente.
    - **Parámetros params:** Permiten pasar un número variable de argumentos del mismo tipo, que se agrupan en un arreglo dentro del método.
    - **Parámetros Opcionales:** Se les asigna un valor por defecto en la declaración del método. Si no se proporciona un argumento al llamar al método, se usará el valor predeterminado.
- **Ámbito de las Variables:** Define dónde una variable es accesible.
    - **Variables Locales:** Se declaran dentro de un método o bloque de código ({}). Solo existen y son accesibles dentro de ese bloque.
    - **Variables "Globales" (Campos static):** En C#, no existen variables globales como en otros lenguajes. El comportamiento similar se logra declarando una variable como static dentro de una clase. Esta variable es compartida por todas las instancias de la clase y mantiene su valor durante toda la ejecución del programa. Su uso debe ser cuidadoso para evitar efectos secundarios inesperados.

Tema 4: Manejo de Errores y Excepciones

Un programa robusto debe ser capaz de manejar situaciones inesperadas sin colapsar.

- **Tipos de Errores:**
    - **Sintácticos:** Errores en la gramática del lenguaje que impiden la compilación (ej. falta un ;).
    - **Lógicos:** El programa compila y se ejecuta, pero produce resultados incorrectos porque la lógica es defectuosa (ej. usar + en lugar de -).
    - **De Ejecución (Excepciones):** Errores que ocurren mientras el programa está en funcionamiento.
- **Excepciones:** Son objetos que representan un error en tiempo de ejecución. Interrumpen el flujo normal del programa.
- **Estructura try-catch-finally:**
    - **try:** Se coloca el código que podría generar una excepción.
    - **catch:** Si ocurre una excepción en el bloque try, el control pasa al bloque catch correspondiente, donde se maneja el error. Puedes tener varios catch para distintos tipos de excepciones.
    - **finally:** Este bloque (opcional) se ejecuta *siempre*, ya sea que haya ocurrido una excepción o no. Es ideal para liberar recursos, como cerrar archivos.
- **Excepciones Comunes en C#:**
    - DivideByZeroException: Intentar dividir un número entre cero.
    - FormatException: Intentar convertir una cadena a un número con un formato incorrecto (ej. Convert.ToInt32("hola")).
    - NullReferenceException: Intentar usar un objeto que tiene el valor null.
    - IndexOutOfRangeException: Intentar acceder a un elemento de un arreglo con un índice fuera de los límites válidos.

Parte 2: Ejercicios Resueltos

Aquí tienes ejemplos prácticos que combinan los temas teóricos.

Ejercicio Resuelto 1: Calculadora Básica (Tema 1 y 2)

Este programa lee dos números y realiza una operación básica seleccionada por el usuario.

```
// Figura 3.18 y 3.26 adaptadas con switch
using System;


public class CalculadoraBasica
{
   public static void Main(string[] args)
   {
      int numero1, numero2; // Declara variables para los operandos.

      // Pide y lee el primer número.
      Console.Write("Escriba el primer entero: ");
      numero1 = Convert.ToInt32(Console.ReadLine());

      // Pide y lee el segundo número.
      Console.Write("Escriba el segundo entero: ");
      numero2 = Convert.ToInt32(Console.ReadLine());

      // Muestra un menú de opciones.
      Console.WriteLine("\nSeleccione una operación:");
      Console.WriteLine("1 - Suma");
      Console.WriteLine("2 - Resta");
      Console.WriteLine("3 - Multiplicación");
      Console.WriteLine("4 - División");
      Console.Write("Opción: ");
```

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


// Usa switch para seleccionar la operación.
switch (opcion)
{
    case 1:
        int suma = numero1 + numero2; // Operador de suma.
        Console.WriteLine($"La suma es: {suma}"); // Muestra el resultado.
        break;
    case 2:
        int resta = numero1 - numero2; // Operador de resta.
        Console.WriteLine($"La resta es: {resta}");
        break;
    case 3:
        int multiplicacion = numero1 * numero2; // Operador de multiplicación.
        Console.WriteLine($"La multiplicación es: {multiplicacion}");
        break;
    case 4:
        // Usa 'if' para validar la división por cero.
        if (numero2 != 0)
        {
            double division = (double)numero1 / numero2; // División de enteros puede truncar, convertimos a double.
            Console.WriteLine($"La división es: {division}");
        }
        else
```

```
            {
                Console.WriteLine("Error: No se puede dividir entre cero.");
            }
            break;
        default: // Se ejecuta si ninguna opción coincide.
            Console.WriteLine("Opción no válida.");
            break;
    }
  }
}
```

Ejercicio Resuelto 2: Calculadora Modular con Manejo de Errores (Tema 3 y 4)

Este es el enfoque principal de tu examen. Refactorizamos la calculadora anterior usando funciones y añadiendo try-catch para un manejo de errores robusto.

```
using System;


public class CalculadoraModular
{
  // Método principal que controla el flujo del programa.
  public static void Main(string[] args)
  {
    bool continuar = true;
    // Bucle 'while' para permitir múltiples cálculos.
    while (continuar)
    {
      MostrarMenu();
      int opcion = LeerOpcion();
```

```
        // Usamos un 'if' para la opción de salir.
        if (opcion >= 1 && opcion <= 4)
        {
            EjecutarOperacion(opcion);
        }
        else if (opcion == 5)
        {
            continuar = false;
            Console.WriteLine("Saliendo de la aplicación...");
        }
        else
        {
            Console.WriteLine("Error: Opción no válida. Intente de nuevo.");
        }
        Console.WriteLine("---------------------------------");
    }
}

// Procedimiento para mostrar el menú. No devuelve valor (void).
public static void MostrarMenu()
{
    Console.WriteLine("\n--- Calculadora Modular ---");
    Console.WriteLine("1. Sumar");
    Console.WriteLine("2. Restar");
    Console.WriteLine("3. Multiplicar");
```

```
    Console.WriteLine("4. Dividir (con cociente y residuo)");
    Console.WriteLine("5. Salir");
}

// Función para leer y validar la entrada del usuario.
public static int LeerNumero(string mensaje)
{
    while (true) // Bucle para reintentar si la entrada es inválida.
    {
        try
        {
            Console.Write(mensaje);
            return Convert.ToInt32(Console.ReadLine());
        }
        catch (FormatException) // Captura error de formato.
        {
            Console.WriteLine("Error: Por favor, ingrese solo números enteros.");
        }
    }
}

// Función para leer la opción del menú.
public static int LeerOpcion()
{
    return LeerNumero("Seleccione una opción: ");
}
```

```
// Procedimiento que coordina la operación a realizar.
public static void EjecutarOperacion(int opcion)
{
   int a = LeerNumero("Ingrese el primer número: ");
   int b = LeerNumero("Ingrese el segundo número: ");

   switch (opcion)
   {
      case 1:
         Console.WriteLine($"Resultado: {a} + {b} = {Sumar(a, b)}");
         break;
      case 2:
         Console.WriteLine($"Resultado: {a} - {b} = {Restar(a, b)}");
         break;
      case 3:
         Console.WriteLine($"Resultado: {a} * {b} = {Multiplicar(a, b)}");
         break;
      case 4:
         // Llamamos a la función Dividir, que usa parámetros 'out'.
         if (Dividir(a, b, out int cociente, out int residuo))
         {
            Console.WriteLine($"Resultado: {a} / {b} = {cociente} (Residuo: {residuo})");
         }
         break;
   }
```

```
    }

    // Funciones modulares para cada operación aritmética.

    public static int Sumar(int num1, int num2) => num1 + num2; // Sintaxis de cuerpo de
expresión.

    public static int Restar(int num1, int num2) => num1 - num2;

    public static int Multiplicar(int num1, int num2) => num1 * num2;

    // Función que devuelve un booleano y usa parámetros 'out' para múltiples resultados.
    public static bool Dividir(int num1, int num2, out int cociente, out int residuo)
    {
        // Inicializamos los parámetros 'out'.
        cociente = 0;
        residuo = 0;

        try
        {
            // El bloque 'try' protege el código propenso a errores.
            cociente = num1 / num2; // Posible DivideByZeroException.
            residuo = num1 % num2; // Operador residuo.
            return true; // La operación fue exitosa.
        }
        catch (DivideByZeroException) // Captura específica.
```

```
    {
        Console.WriteLine("Error: No se puede dividir entre cero.");
        return false; // La operación falló.
    }
  }
}
```

Parte 3: Ejercicios Propuestos (Tarea)

Ahora te toca a ti. Intenta resolver estos ejercicios aplicando los conceptos de modularidad, control de flujo y manejo de excepciones.

Ejercicio de Tarea 1: Verificador de Números Primos (Modular)

1. Crea una **función bool EsPrimo(int numero)** que reciba un número entero y devuelva true si es primo y false si no lo es.
    - *Pista: Un número es primo si solo es divisible por 1 y por sí mismo. Puedes usar un bucle for para comprobar sus divisores desde 2 hasta la raíz cuadrada del número.*
2. En el método Main, pide al usuario que ingrese un número.
3. Llama a la función EsPrimo para verificar el número.
4. Muestra un mensaje indicando si el número es primo o no.
5. **Extra:** Usa un bucle do-while para permitir que el usuario verifique varios números hasta que decida salir.

Ejercicio de Tarea 2: Gestión de Calificaciones (Modular con params y out)

1. Crea una función void CalcularEstadisticas(out double promedio, out int max, out int min, params int[] calificaciones):
    - Debe recibir un número variable de calificaciones usando params.
    - Debe calcular el promedio, la calificación máxima y la mínima.
    - Debe devolver estos tres valores usando parámetros out.
2. En el método Main:

- Define un arreglo de calificaciones (ej. int[] notas = { 85, 92, 78, 65, 95, 88 };).
- Llama a CalcularEstadisticas pasándole las notas.
- Muestra en consola el promedio, la nota más alta y la más baja.
- **Manejo de errores:** Modifica la función para que, si no se pasan calificaciones, el promedio, máximo y mínimo se establezcan en 0 y se muestre un mensaje de advertencia.

Ejercicio de Tarea 3 (Desafío Final): Simulador de Cajero Automático (ATM)

Este ejercicio combina todos los temas y se basa en el caso de estudio de tus fuentes.

1. Usa una variable static decimal saldo = 1000.00m; para simular el saldo de la cuenta.
2. Crea un **menú principal** en un bucle while que ofrezca las siguientes opciones:
    - 1. Ver Saldo
    - 2. Depositar Dinero
    - 3. Retirar Dinero
    - 4. Salir
3. Crea **métodos separados** para cada funcionalidad:
    - void VerSaldo(): Muestra el saldo actual.
    - void Depositar(): Pide al usuario un monto, lo valida (debe ser positivo) y lo suma al saldo.
    - void Retirar(): Pide un monto, valida que sea positivo y que no exceda el saldo disponible. Si es válido, lo resta del saldo.
4. **Manejo de Errores:**
    - Dentro de los métodos Depositar y Retirar, usa try-catch para manejar FormatException si el usuario introduce texto en lugar de un número.

- Usa if para validar las reglas de negocio (ej. no retirar más dinero del que hay, no depositar montos negativos).

¡Mucha suerte con tu estudio! Si tienes alguna duda sobre un concepto o un ejercicio, no dudes en preguntar.