**Fundamentos de Delegados (delegate) en C#**

1. **¿Qué es un delegado?**

En C#, un delegado es un tipo seguro que hace referencia a un método o a un conjunto de métodos que cumplen con una misma firma (tipo de retorno y parámetros).  
Su función principal es permitir que los métodos sean tratados como objetos, facilitando el paso de funciones como argumentos y la implementación de patrones como callbacks, event handling y programación funcional.

Un delegado actúa como un contrato: solo puede apuntar a métodos cuya firma sea compatible con la que se definió en el delegado.

1. **Características principales**

* Type-safe: garantizan que el método referenciado tenga el mismo tipo de retorno y parámetros.
* Multicast: pueden invocar más de un método a la vez usando el operador +=.
* Permiten desacoplar código, favoreciendo el principio de inversión de dependencias.
* Se encuentran en el namespace System.
* Funcionan con métodos estáticos y de instancia.

Cuando se dice que los delegados funcionan con métodos estáticos y de instancia, se refiere a que un delegado en C# puede apuntar (o referenciar) tanto a:

Métodos estáticos, que pertenecen a la clase, no a un objeto específico.

No requieren una instancia para ser invocados.

Ejemplo:

public class Logger

{

public static void LogMessage(string message)

{

Console.WriteLine($"[LOG]: {message}");

}

}

public delegate void LogHandler(string message);

class Program

{

static void Main()

{

LogHandler handler = Logger.LogMessage; // Apunta a un método estático

handler("Mensaje desde método estático");

}

}

Aquí el delegado handler está referenciando un método estático (Logger.LogMessage).

**Métodos de instancia**. Que pertenecen a un objeto creado.

Necesitan una instancia para ser invocados.

Ejemplo:

public class Printer

{

public void PrintDocument(string document)

{

Console.WriteLine($"Imprimiendo: {document}");

}

}

public delegate void PrintHandler(string document);

class Program

{

static void Main()

{

Printer printer = new Printer();

PrintHandler handler = printer.PrintDocument; // Apunta a un método de instancia

handler("Factura.pdf");

}

}

Aquí el delegado handler hace referencia a un método que necesita un objeto (printer) para ejecutarse.

**En resumen**  
Un delegado es como un "puntero seguro" a un método, y ese método puede ser estático o de instancia.

**Estático** → No requiere un objeto.

**Instancia** → Requiere un objeto.

* Admiten métodos anónimos, expresiones lambda y funciones locales.

1. **Sintaxis básica**

**Declaración de un delegado**

public delegate void MiDelegado(string mensaje);

Este delegado puede referenciar cualquier método que:

* Retorne void
* Reciba un parámetro string

**Ejemplo básico de uso**

using System;

class Program

{

delegate void TypeDelegate(string message);

static void Main(string[] args)

{

TypeDelegate typeDelegate = new TypeDelegate(MessageWelcome.Welcome);

typeDelegate("Llegué"); // Ejecutar el delegado

MessageWelcome.Welcome("Hola hola");

}

// Clases de prueba

class MessageWelcome

{

public static void Welcome(string message)

{

Console.WriteLine($"Hey {message}");

}

}

class MessageBye

{

public static void Bye(string message)

{

Console.WriteLine($"Hey {message}");

}

}

}

1. **Delegados multicast**

Un delegado puede apuntar a varios métodos usando += y eliminarlos con -=.

using System;

class Program

{

delegate void TypeDelegate(string message);

static void Main(string[] args)

{

TypeDelegate typeDelegate = MessageWelcome.Welcome;

typeDelegate += MessageBye.Bye; // Agregamos otro método al delegado

typeDelegate("Hola "); // Ejecutar el delegado

}

// Clases de prueba

class MessageWelcome

{

public static void Welcome(string message)

{

Console.WriteLine($"{message} estoy llegando");

Thread.Sleep(2000);

}

}

class MessageBye

{

public static void Bye(string message)

{

Console.WriteLine($"{message} estoy saliendo");

}

}

}

Otro ejemplo:

public delegate void Operacion();

static void Metodo1() => Console.WriteLine("Método 1");

static void Metodo2() => Console.WriteLine("Método 2");

Operacion op = Metodo1;

op += Metodo2;

op(); // Ejecuta Método 1 y Método 2

El orden de ejecución de métodos en un delegado multicast es el orden en que fueron agregados.

1. **Uso de delegados para procesar listas de forma flexible y reutilizable**

### ****Qué hace el programa****

1. Tiene una lista inicial de nombres.
2. Usa **distintas funciones** (ToUpperCase, AddPrefix, CapitalizeInitial) para procesar los nombres.
3. Aplica esas funciones mediante el método genérico ProcessAndOrderList:

* Recorre la lista.
* Procesa cada elemento usando el delegado.
* Ordena alfabéticamente el resultado.

1. Imprime la lista final.

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

delegate string StringOperation(string text);

static void Main(string[] args)

{

List<string> names = new List<string> { "sonia", "juan", "alba", "pedro", "carlos" };

// --- Escenario 1: Convertir a mayúsculas y ordenar ---

Console.WriteLine("\n--- Procesando: Convertir a Mayúsculas y Ordenar ---");

StringOperation toUpperOp = new StringOperation(StringProcessor.ToUpperCase);

List<string> processedAndOrderedNamesUpper = ProcessAndOrderList(names, toUpperOp);

Console.WriteLine("Lista en mayúsculas y ordenada:");

PrintList(processedAndOrderedNamesUpper);

Console.WriteLine("\n--- Procesando: Añadir Prefijo 'Hola ' y Ordenar ---");

StringOperation addPrefixOp = new StringOperation(StringProcessor.AddPrefix);

List<string> processedAndOrderedNamesPrefixed = ProcessAndOrderList(names, addPrefixOp);

Console.WriteLine("Lista con prefijo y ordenada:");

PrintList(processedAndOrderedNamesPrefixed);

StringOperation capitalizeOp = new StringOperation(StringProcessor.CapitalizeInitial);

List<string> namesCapitalize = ProcessAndOrderList(names, capitalizeOp);

Console.WriteLine("Lista capitalizada:");

PrintList(namesCapitalize);

}

// Método que realiza las operaciones sobre los strings

class StringProcessor

{

public static string ToUpperCase(string text)

{

return text.ToUpper(); // Convierte el string a mayúsculas

}

public static string AddPrefix(string text)

{

return "Hola " + text; // Añade un prefijo al string

}

public static string CapitalizeInitial(string text) // Capitaliza la primera letra

{

if (string.IsNullOrEmpty(text))

{

return string.Empty;

}

return char.ToUpper(text[0]) + text.Substring(1).ToLower();

}

}

// Nuestro método genérico que toma el delegado

// Este método no sabe \*cómo\* se procesará cada string, solo que se procesará.

static List<string> ProcessAndOrderList(List<string> inputList, StringOperation operation)

{

List<string> processedList = new List<string>();

foreach (string item in inputList)

{

processedList.Add(operation(item));

}

// Finalmente, ordenamos la lista resultante

return processedList.OrderBy(s => s).ToList();

}

// Método auxiliar para imprimir listas

static void PrintList(List<string> list)

{

foreach (string item in list)

{

Console.WriteLine($"- {item}");

}

}

}

1. **Delegados genéricos: Action, Func, Predicate**
   1. **Action**

Representa un delegado que no devuelve valor (void).

Ejemplo:

using System;

class Program

{

static void Main()

{

// Action sin parámetros: registra inicio

Action registrarInicio = () =>

{

Console.WriteLine($"[{DateTime.Now}] - Inicio del proceso");

Thread.Sleep(1500);

};

// Action<string>: registra mensajes personalizados

Action<string> registrarMensaje = mensaje =>

{

Console.WriteLine($"[{DateTime.Now}] - {mensaje}");

Thread.Sleep(1500);

};

registrarInicio();

registrarMensaje("Conectando a la base de datos...");

registrarMensaje("Consulta ejecutada con éxito");

registrarMensaje("Proceso finalizado");

}

}

Otro ejemplo:

using System;

using System.Collections.Generic;

public class Notificador

{

// Lista de acciones que se ejecutarán cuando haya una notificación

private List<Action<string>> accionesNotificacion = new List<Action<string>>();

public void Suscribir(Action<string> accion)

{

accionesNotificacion.Add(accion);

}

public void Notificar(string mensaje)

{

foreach (var accion in accionesNotificacion)

{

accion(mensaje); // Ejecuta cada acción suscrita

}

}

}

class Program

{

static void Main()

{

Notificador notificador = new Notificador();

// Suscribir diferentes acciones

notificador.Suscribir(m => Console.WriteLine($"[LOG] {m}"));

notificador.Suscribir(m => EnviarCorreo(m));

notificador.Suscribir(m => MostrarEnPantalla(m));

// Ejecutar notificación

notificador.Notificar("Factura #125 generada y lista para envío.");

}

static void EnviarCorreo(string mensaje)

{

Console.WriteLine($"Correo enviado: {mensaje}");

}

static void MostrarEnPantalla(string mensaje)

{

Console.WriteLine($"Pantalla: {mensaje}");

}

}

* Action<string> se usa para registrar métodos diferentes que reciben un string.
* Se agregan a una lista de acciones (accionesNotificacion) y se ejecutan todas cuando hay un evento de notificación.
* En este ejemplo, una misma notificación genera tres acciones: log en consola, envío de correo y visualización en pantalla.
  1. **Func**

Representa un delegado que devuelve un valor.  
El último parámetro genérico es el tipo de retorno.

using System;

using System.Globalization;

public class Program

{

public static void Main()

{

// Definir cultura colombiana

CultureInfo culturaCOP = new CultureInfo("es-CO");

// Un Func que recibe un decimal (precio) y retorna un decimal (precio con IVA)

Func<decimal, decimal> calcularIVA = precio => precio \* 1.19m;

// Un Func que recibe precio y descuento y retorna el precio final

Func<decimal, decimal, decimal> aplicarDescuento = (precio, descuento) => precio - (precio \* descuento / 100);

decimal precioBase = 100000m;

// Aplicar IVA

decimal precioConIVA = calcularIVA(precioBase);

Console.WriteLine($"Precio con IVA: {precioConIVA.ToString("C", culturaCOP)}");

// Aplicar descuento del 10% al precio con IVA

decimal precioFinal = aplicarDescuento(precioConIVA, 10);

Console.WriteLine($"Precio final con descuento: {precioFinal.ToString("C", culturaCOP)}");

}

}

* Func<decimal, decimal> → recibe 1 decimal y devuelve 1 decimal.
* Func<decimal, decimal, decimal> → recibe 2 decimales y devuelve 1 decimal.
* Los delegados calcularIVA y aplicarDescuento se ejecutan como si fueran funciones normales.
  1. **Predicate**

Representa un delegado que devuelve bool.

using System.Globalization;

public class Program

{

public static void Main()

{

CultureInfo copCulture = new CultureInfo("es-CO");

// Lista de product prices

List<decimal> productPrices = new List<decimal> { 50000m, 150000m, 300000m, 80000m, 120000m };

// Predicate: check if the price > 100.000

Predicate<decimal> isExpensive = price => price > 100000m;

Console.WriteLine("=== Lista de precios ===");

foreach (var price in productPrices)

{

Console.WriteLine(price.ToString("C0", copCulture));

}

Console.WriteLine("\n=== Filtrando precios caros (mayores a $100.000) ===");

List<decimal> expensivePrices = productPrices.FindAll(isExpensive);

foreach (var price in expensivePrices)

{

Console.WriteLine(price.ToString("C0", copCulture));

}

}

}

* Predicate es un delegado genérico
* Recibe un valor de tipo T y devuelve un bool (true o false).
* Predicate<T> sirve para encapsular una condición lógica (sí/no) que se aplica a valores u objetos, y se usa principalmente para filtrar, buscar o validar en colecciones.

Otro ejemplo:

using \_08\_Delegate08;

using System.Globalization;

public class Program

{

public static void Main()

{

CultureInfo copCulture = new CultureInfo("es-CO");

const decimal TRANSPORT\_ALLOWANCE\_LIMIT = 2847000m;

// Lista de empleados

List<Employee> employees = new List<Employee>

{

new Employee { Id = 1, Name = "Carlos Pérez", Salary = 2500000m },

new Employee { Id = 2, Name = "Ana Gómez", Salary = 3200000m },

new Employee { Id = 3, Name = "Luis Torres", Salary = 2800000m },

new Employee { Id = 4, Name = "María Díaz", Salary = 4000000m }

};

// Predicate para verificar derecho al auxilio de transporte

Predicate<Employee> hasTransportAllowance = emp => emp.Salary <= TRANSPORT\_ALLOWANCE\_LIMIT;

Console.WriteLine("=== Empleados con derecho a auxilio de transporte ===");

List<Employee> eligibleEmployees = employees.FindAll(hasTransportAllowance);

foreach (Employee emp in eligibleEmployees)

{

Console.WriteLine($"Nombre: {emp.Name}, Salario: {emp.Salary.ToString("C0", copCulture)}");

}

}

}

1. **Delegados y eventos**

Los eventos en C# se basan en delegados.

using System.Diagnostics;

class Program

{

public delegate void Notificacion();

public class Proceso

{

public event Notificacion? ProcesoFinalizado;

public void Ejecutar()

{

Console.WriteLine("Ejecutando...");

ProcesoFinalizado?.Invoke();

}

}

static void Main(string[] args)

{

Proceso p = new Proceso();

p.ProcesoFinalizado += () => Console.WriteLine("Proceso terminado");

p.ProcesoFinalizado += () => Console.WriteLine("=== FIN ===");

p.Ejecutar();

}

}

1. **Comparativa de tipos de delegados**

**Tabla 1**

*Comparativa de tipos de delegados*

| **Tipo** | **Retorno** | **Uso principal** |
| --- | --- | --- |
| delegate personalizado | Cualquiera | Firmas específicas y control total |
| Action | void | Ejecución de lógica sin valor de retorno |
| Func | Genérico | Cálculos o transformaciones que devuelven valor |
| Predicate | bool | Filtrado y validaciones |

1. **Buenas prácticas**

* Usar Action, Func o Predicate para evitar declarar delegados innecesarios.
* Usar expresiones lambda para simplificar código.
* Usar ?.Invoke() para invocar delegados de forma segura, evitando excepciones cuando no hay suscriptores.
* No abusar del multicast si el orden de ejecución es crítico.
* En sistemas con eventos, preferir la firma (object sender, EventArgs e) para compatibilidad.

1. **Ejemplo práctico: sistema de notificaciones**

using System;

public class Notificador

{

public delegate void EnviarNotificacion(string mensaje);

public EnviarNotificacion Notificar;

public void EjecutarProceso()

{

Console.WriteLine("Proceso en ejecución...");

Notificar?.Invoke("El proceso ha finalizado.");

}

}

class Program

{

static void Main()

{

Notificador n = new Notificador();

n.Notificar += m => Console.WriteLine($"[Email] {m}");

n.Notificar += m => Console.WriteLine($"[Log] {m}");

n.EjecutarProceso();

}

}

**Salida:**

Proceso en ejecución...

[Email] El proceso ha finalizado.

[Log] El proceso ha finalizado.

1. **Ejemplo adicional: delegado personalizado con múltiples métodos**

Este ejemplo muestra cómo crear un delegado, asignarlo a un método y luego convertirlo en multicast con varios métodos.

using System;

class Program

{

// Definición del delegado

delegate void TypeDelegate(string message);

static void Main(string[] args)

{

// Instanciar delegado apuntando a un método

TypeDelegate typeDelegate = MessageWelcome.Welcome;

// Invocar delegado

typeDelegate("Llegué");

// Convertirlo en multicast agregando otro método

typeDelegate += MessageBye.Bye;

typeDelegate("Nos vemos");

}

class MessageWelcome

{

public static void Welcome(string message)

{

Console.WriteLine($"Hey {message}");

}

}

class MessageBye

{

public static void Bye(string message)

{

Console.WriteLine($"Adiós {message}");

}

}

}

**Salida:**

Hey Llegué

Hey Nos vemos

Adiós Nos vemos

Este patrón es útil cuando quieres ejecutar una cadena de métodos en respuesta a una sola invocación de delegado.