Attributi personalizzati

🧠 Cos'è un attributo?

Un **attributo** è una classe speciale che consente di **associare metadati dichiarativi** a elementi del codice:

- Classi
- Metodi
- Proprietà
- Campi
- Eventi
- Parametri
- Assembly

Gli attributi possono essere **letti a runtime** tramite Reflection oppure usati dai **compilatori o framework** per modificare il comportamento del codice.

🧰 Esempi di attributi predefiniti

```
[Obsolete("Questo metodo è deprecato.")]
public void MetodoVecchio() { }

[Serializable]
public class Persona { }

[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hWnd, string text, string caption,
int type);
```

✍️ Creare un attributo personalizzato

1. Inherit da `System.Attribute`

2. Specifica i target ammessi con `[AttributeUsage]`

3. Aggiungi proprietà pubbliche (solitamente `get-only`)

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMult
iple = false, Inherited = true)]
public class InfoAutoreAttribute : Attribute
{
    public string Nome { get; }
    public int Anno { get; }

    public InfoAutoreAttribute(string nome, int anno)
    {
        Nome = nome;
        Anno = anno;
    }
}
```

📌 Opzioni di `AttributeUsage`

Opzione	Descrizione
`AttributeTargets`	Dove può essere applicato (Class, Method, Property, ecc.)
`AllowMultiple`	Se può essere applicato più volte sullo stesso elemento (default: false)
`Inherited`	Se gli attributi si ereditano da una classe base (default: false)

✅ Applicare l'attributo personalizzato

```
[InfoAutore("Marius", 2023)]
public class Report { }
```

🔎 Leggere un attributo via Reflection

```
Type tipo = typeof(Report);

// Recupera un attributo specifico
var attributo = (InfoAutoreAttribute?)Attribute.GetCustomAttribute(tipo, type
of(InfoAutoreAttribute));
```

```
if (attributo != null)
{
    Console.WriteLine($"Autore: {attributo.Nome}, Anno: {attributo.Anno}");
}
```

🧪 Esempio completo: Logging automatico

👣 1. Definizione dell'attributo

```
[AttributeUsage(AttributeTargets.Method)]
public class LoggableAttribute : Attribute { }
```

👣 2. Classe con metodi annotati

```
public class Servizio
{
    [Loggable]
    public void Avvia() => Console.WriteLine("Avvio...");

    [Loggable]
    public void Ferma() => Console.WriteLine("Arresto...");
}
```

👣 3. Esecuzione automatica con reflection

```
var tipo = typeof(Servizio);
var istanza = Activator.CreateInstance(tipo);

foreach (var metodo in tipo.GetMethods())
{
    if (Attribute.IsDefined(metodo, typeof(LoggableAttribute)))
    {
        Console.WriteLine($"[LOG] Esecuzione: {metodo.Name}");
        metodo.Invoke(istanza, null);
    }
}
```

🧾 **Output:**

```
[LOG] Esecuzione: Avvia
Avvio...
[LOG] Esecuzione: Ferma
Arresto...
```

🏗️ Parametri posizionali e nominati

- I **parametri posizionali** si impostano nel costruttore
- I **parametri nominati** sono proprietà pubbliche `settable`

```
[AttributeUsage(AttributeTargets.Class)]
public class DocumentazioneAttribute : Attribute
{
    public string Autore { get; set; }
    public bool Ufficiale { get; }

    public DocumentazioneAttribute(bool ufficiale)
    {
        Ufficiale = ufficiale;
    }
}
```

```
[Documentazione(true, Autore = "Marius")]
public class Algoritmo { }
```

🧩 Attributi e ereditarietà

Gli attributi **non vengono ereditati automaticamente**. Per renderli ereditabili su sottoclassi, imposta `Inherited = true` in `AttributeUsage`.

```
[AttributeUsage(AttributeTargets.Class, Inherited = true)]
public class AuditAttribute : Attribute { }
```

💡 Best Practice

- ✅ Usa gli attributi per **descrivere** metadati, **non per implementare logica**
- ✅ Mantieni gli attributi **immutabili** (solo `get`)
- ✅ Dai nomi chiari e sempre terminanti con `Attribute`
- ❌ Non abusarne per evitare metaprogrammazione opaca
- ✅ Crea attributi riutilizzabili (con vincoli precisi di utilizzo)

📚 Usi tipici degli attributi

Caso d'uso	Attributi comuni
Serializzazione JSON	`[JsonProperty]`, `[JsonIgnore]`
Mappatura ORM	`[Key]`, `[Column("nome_colonna")]`
Testing	`[Test]`, `[TestMethod]`
Web API	`[HttpGet]`, `[FromBody]`, `[Route]`
Reflection custom	Attributi personalizzati

🧠 Riassunto finale

- ✅ Gli attributi sono un modo potente e flessibile per **arricchire il codice con metadati**
- ✅ Puoi **crearli facilmente** estendendo `System.Attribute`
- ✅ Sono letti tramite **Reflection**
- ✅ Ottimi per creare **plugin, validazioni, logging, ORM, testing** e molto altro

---

📌 1. **Log automatico di metodi eseguiti**

Scenario: vuoi annotare solo i metodi rilevanti da loggare, e usare reflection per trovarli.

```
[AttributeUsage(AttributeTargets.Method)]
public class LoggableAttribute : Attribute { }

public class Servizio
```

```
{
    [Loggable]
    public void InviaEmail() => Console.WriteLine("Email inviata");
}

// Invocazione automatica via Reflection
var metodi = typeof(Servizio).GetMethods();
foreach (var m in metodi)
{
    if (Attribute.IsDefined(m, typeof(LoggableAttribute)))
        m.Invoke(new Servizio(), null);
}
```

📌 2. **Mappatura proprietà → colonne database (style ORM)**

Scenario: vuoi evitare configurazioni esterne e mappare direttamente nel codice proprietà → colonne SQL.

```
[AttributeUsage(AttributeTargets.Property)]
public class ColumnAttribute : Attribute
{
    public string NomeColonna { get; }
    public ColumnAttribute(string nome) => NomeColonna = nome;
}

public class Utente
{
    [Column("first_name")]
    public string Nome { get; set; }

    [Column("last_name")]
    public string Cognome { get; set; }
}
```

✅ Con Reflection puoi usare `GetCustomAttribute()` per costruire query SQL dinamiche.

📌 3. **Validazione custom dei modelli**

Scenario: vuoi dichiarare vincoli di validazione direttamente sulle proprietà (come `[Required]` o `[Range]`).

```
[AttributeUsage(AttributeTargets.Property)]
public class CampoObbligatorioAttribute : Attribute { }

public class LoginModel
{
    [CampoObbligatorio]
    public string Username { get; set; }

    [CampoObbligatorio]
    public string Password { get; set; }
}
```

🛠️ Un validatore può ispezionare ogni proprietà e restituire errori se il valore è nullo.

📌 4. **Abilitare funzioni solo in modalità debug/test**

Scenario: vuoi eseguire metodi solo se un attributo ne consente l'attivazione.

```
[AttributeUsage(AttributeTargets.Method)]
public class SoloInDebugAttribute : Attribute { }

public class Tester
{
    [SoloInDebug]
    public void MetodoTest() => Console.WriteLine("Test in corso...");
}

// Esecuzione condizionata
if (modalitaDebugAttiva)
{
    var metodo = typeof(Tester).GetMethod("MetodoTest");
    if (Attribute.IsDefined(metodo, typeof(SoloInDebugAttribute)))
        metodo.Invoke(new Tester(), null);
}
```

📌 5. **Documentazione interna del codice (autore, data, note)**

Scenario: vuoi lasciare metadati leggibili via tool o editor per audit interno o strumenti di generazione documentazione.

```
[AttributeUsage(AttributeTargets.Class)]
public class DocumentazioneAttribute : Attribute
{
    public string Autore { get; }
    public string Data { get; }

    public DocumentazioneAttribute(string autore, string data)
    {
        Autore = autore;
        Data = data;
    }
}

[Documentazione("Marius", "2025-11-06")]
public class Calcolatore { }
```

🔍 Questo tipo di attributo può essere letto da tool interni o plugin IDE.

📌 6. **Controllo autorizzazioni su metodi (style Web API / ACL)**

Scenario: vuoi controllare a runtime se un utente ha il permesso per eseguire certi metodi.

```
[AttributeUsage(AttributeTargets.Method)]
public class RichiedeRuoloAttribute : Attribute
{
    public string Ruolo { get; }
    public RichiedeRuoloAttribute(string ruolo) => Ruolo = ruolo;
}

public class ReportService
{
    [RichiedeRuolo("Admin")]
```

```
    public void GeneraReport() => Console.WriteLine("Report generato");
}
```

🔐 A runtime, puoi leggere il ruolo richiesto e confrontarlo con quello dell'utente autenticato.

📌 7. **Inizializzazione automatica di proprietà da configurazione**

Scenario: vuoi contrassegnare quali proprietà vanno valorizzate da un file `.json` o ambiente.

```
[AttributeUsage(AttributeTargets.Property)]
public class DaConfigurazioneAttribute : Attribute
{
    public string Chiave { get; }
    public DaConfigurazioneAttribute(string chiave) => Chiave = chiave;
}

public class ConnessioneDb
{
    [DaConfigurazione("Db:Host")]
    public string Host { get; set; }

    [DaConfigurazione("Db:Port")]
    public int Porta { get; set; }
}
```

🧩 Un loader può leggere queste proprietà e popolarle con i valori della configurazione.

📌 8. **Definizione di comandi in una CLI o chatbot**

Scenario: vuoi mappare metodi come comandi eseguibili da input testuale.

```
[AttributeUsage(AttributeTargets.Method)]
public class ComandoAttribute : Attribute
{
    public string Nome { get; }
```

```
    public ComandoAttribute(string nome) => Nome = nome;
}

public class Bot
{
    [Comando("ciao")]
    public void Saluta() => Console.WriteLine("Ciao!");
}
```

🎛️ A runtime, interpreti la stringa `"ciao"` e invoci il metodo decorato con `[Comando("ciao")]`.

📌 9. **Mapping per esportazione CSV personalizzata**

Scenario: vuoi controllare nomi e ordine delle colonne in un'esportazione CSV.

```
[AttributeUsage(AttributeTargets.Property)]
public class CsvColonnaAttribute : Attribute
{
    public string Nome { get; }
    public int Ordine { get; }
    public CsvColonnaAttribute(string nome, int ordine)
    {
        Nome = nome;
        Ordine = ordine;
    }
}

public class Prodotto
{
    [CsvColonna("Nome prodotto", 1)]
    public string Nome { get; set; }

    [CsvColonna("Prezzo", 2)]
    public decimal Prezzo { get; set; }
}
```

📄 Con reflection puoi costruire righe CSV con intestazioni ordinate correttamente.

📌 10. **Controllo di versioning del codice in ambienti critici**

Scenario: vuoi tracciare versioni di funzioni critiche per compatibilità o rollback.

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]
public class VersioneApiAttribute : Attribute
{
    public string Versione { get; }
    public VersioneApiAttribute(string versione) => Versione = versione;
}

[VersioneApi("v1.0")]
public class CalcoloPrezzi
{
    public decimal Calcola(decimal basePrice) => basePrice * 1.22M;
}
```

📌 Puoi filtrare o loggare solo le versioni `v1.*`, o bloccare versioni obsolete in ambienti di produzione.