Slide 1 – Introduzione a Reflection e metadati

La **reflection** (riflessione) è la capacità di un programma .NET di **ispezionare i propri metadati a runtime** e interagire con essi. In ogni assembly .NET sono inclusi metadati dettagliati sui tipi (classi, metodi, proprietà, ecc.) definiti. Tramite la reflection possiamo, ad esempio, ottenere il `Type` di un oggetto durante l'esecuzione, elencare i membri di una classe, creare istanze di tipi e invocare metodi in modo dinamico. Questa flessibilità è alla base di molti framework (ORM, serializer, plugin) che adattano il comportamento in base ai tipi definiti dall'utente.

```
int numero = 42;
Type tipo = numero.GetType();  // ottiene il System.Type dell'oggetto
'numero'
Console.WriteLine($"Il tipo dell'oggetto 'numero' \u00e8: {tipo.FullName}");
// Output: Il tipo dell'oggetto 'numero' è: System.Int32
```

Slide 2 – Ottenere informazioni sul tipo e sull'assembly

La classe astratta `Type` è il fulcro della reflection: rappresenta un tipo e fornisce metodi per esaminarne struttura e metadati. Possiamo ottenere un oggetto `Type` in vari modi: tramite l'operatore `typeof` su un tipo noto a compilazione, oppure chiamando `.GetType()` su un'istanza a runtime. La classe `Assembly` permette di esaminare informazioni sul modulo compilato (nome, versione) e i tipi in esso contenuti.

```
Type tipoString = typeof(string);          // ottiene il Type per il tipo
string (nota: operazione compile-time)
Type tipoRuntime = "hello".GetType();      // ottiene il Type dell'istanza
"hello" a runtime
Console.WriteLine(tipoString == tipoRuntime);  // True, entrambi
rappresentano System.String

Assembly asm = Assembly.GetExecutingAssembly();        // ottiene l'assembly
corrente in esecuzione
Console.WriteLine(asm.GetName().Name);                 // Nome dell'assembly
corrente
Console.WriteLine($"Numero di tipi nell'assembly: {asm.GetTypes().Length}");
```

Slide 3 – Ispezionare proprietà e campi di un oggetto

Una volta ottenuto il `Type` di una classe, possiamo esaminarne i membri. La reflection consente di elencare **proprietà** e **campi** (fields) pubblici di un tipo tramite metodi come `GetProperties()` e `GetFields()`. Ciascuna proprietà è rappresentata da un oggetto `PropertyInfo` (con informazioni su nome, tipo, ecc.), analogamente un campo ha un `FieldInfo`. Nell'esempio seguente definiamo una semplice classe con un campo e una proprietà, quindi usiamo la reflection per elencarli. (Nota: per default, `GetProperties` / `GetFields` restituiscono solo i membri pubblici d'istanza.)

```
class Punto { public int X; public int Y { get; set; } }  // classe esempio
con campo X e proprietà Y

Type t = typeof(Punto);
foreach (PropertyInfo prop in t.GetProperties()) {
    Console.WriteLine($"Propriet\u00e0: {prop.Name}, Tipo:
{prop.PropertyType}");
}
foreach (FieldInfo field in t.GetFields()) {
    Console.WriteLine($"Campo: {field.Name}, Tipo: {field.FieldType}");
}
// Output:
// Proprietà: Y, Tipo: System.Int32
// Campo: X, Tipo: System.Int32
```

Slide 4 – Ispezionare metodi e costruttori

Allo stesso modo, la reflection permette di esaminare i **metodi** e i **costruttori** di una classe. Possiamo ottenere tutti i metodi pubblici dichiarati in un `Type` con `GetMethods()`, e per ciascun `MethodInfo` ispezionare il nome, il tipo restituito (`ReturnType`) e i parametri (collezione di `ParameterInfo`). Similmente, `GetConstructors()` restituisce i costruttori della classe. Nell'esempio elenchiamo i metodi (escludendo quelli ereditati da `object`) e i costruttori, inclusi eventuali privati, di una classe di esempio.

```
class Esempio {
    public void Saluta(string nome, int ripeti) { /* ... */ }
    private Esempio() { }  // costruttore privato
}

Type t = typeof(Esempio);
foreach (MethodInfo m in t.GetMethods(BindingFlags.Public |
BindingFlags.Instance | BindingFlags.DeclaredOnly)) {
    Console.WriteLine($"Metodo: {m.Name}, Ritorna: {m.ReturnType}");
    foreach (ParameterInfo p in m.GetParameters()) {
        Console.WriteLine($"   Parametro: {p.Name} : {p.ParameterType}");
    }
}
foreach (ConstructorInfo c in t.GetConstructors(BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.DeclaredOnly))
{
    Console.WriteLine($"Costruttore: {c}");
}
// Output (possibile):
// Metodo: Saluta, Ritorna: System.Void
//    Parametro: nome : System.String
//    Parametro: ripeti : System.Int32
// Costruttore: Void .ctor()    (costruttore privato elencato)
```

Slide 5 – Invocare metodi con Reflection

La reflection non si limita a leggere informazioni: consente anche di **invocare** membri in modo dinamico. Con un `MethodInfo` possiamo chiamare `Invoke(obj, params)` per eseguire il metodo rappresentato. Analogamente, con un `ConstructorInfo` o usando la classe `Activator` (vedi slide successiva) possiamo creare oggetti al volo. Nell'esempio seguente, usiamo reflection per chiamare un metodo conosciuto solo per nome a runtime.

```
class Calcolatore {
    public int Somma(int a, int b) {
        return a + b;
    }
}

object calc = new Calcolatore();
MethodInfo metodo = calc.GetType().GetMethod("Somma");
int risultato = (int) metodo.Invoke(calc, new object[]{ 2, 3 });
Console.WriteLine($"Risultato invocazione: {risultato}");
// Output: Risultato invocazione: 5
```

Slide 6 – Creazione dinamica di istanze di oggetti

Un aspetto potente della reflection è la capacità di **creare istanze** di tipi a runtime, senza conoscerli a compilazione. La classe `Activator` fornisce metodi come `CreateInstance` che, dato un `Type`, invocano il suo costruttore predefinito e restituiscono un nuovo oggetto. Ciò è utile per caricare classi configurate esternamente o plugin. Nell'esempio, creiamo dinamicamente un oggetto di una classe nota (`StringBuilder`), ottenendo prima il suo `Type`.

```
Type tipo = typeof(System.Text.StringBuilder);       // ottiene il Type del
tipo desiderato
object istanza = Activator.CreateInstance(tipo)!;    // crea un'istanza di
StringBuilder
Console.WriteLine($"Oggetto creato: {istanza.GetType().Name}");
// Output: Oggetto creato: StringBuilder
```

Slide 7 – Accesso a membri privati con BindingFlags

Di default la reflection espone solo i membri pubblici. Tuttavia, è possibile accedere a **membri privati** (campi o metodi) specificando opportuni flag. Ad esempio, `Type.GetField(nome, BindingFlags)` consente di ottenere anche campi non pubblici usando `BindingFlags.NonPublic` in combinazione con `Instance` o `Static`. Questa capacità va usata con cautela, in quanto viola l'incapsulamento. Nell'esempio recuperiamo il valore di un campo privato.

```
class Segreto {
    private string messaggioSegreto = "Nascosto";
}
```

```
Segreto obj = new Segreto();
FieldInfo campo = typeof(Segreto).GetField("messaggioSegreto",
                    BindingFlags.NonPublic | BindingFlags.Instance);
string valore = (string) campo!.GetValue(obj)!;
Console.WriteLine($"Valore del campo privato: {valore}");
// Output: Valore del campo privato: Nascosto
```

Slide 8 – Esempio pratico: introspezione generica di un oggetto

Combinando queste tecniche, possiamo creare funzioni generiche che **ispezionano oggetti di qualsiasi tipo**. Ad esempio, una funzione di "dump" che stampa tutte le proprietà pubbliche e i loro valori di un oggetto fornito, senza conoscerne a priori il tipo. Questo è utile per debugging, logging o implementare toString generici. Nell'esempio definiamo una classe `Persona` e una funzione `StampaProprieta` che usa reflection per iterare su tutte le proprietà pubbliche dell'oggetto e mostrarne nome e valore.

```
class Persona { public string Nome { get; set; } public int Eta { get;
set; } }

void StampaProprieta(object obj) {
    Type t = obj.GetType();
    Console.WriteLine($"Tipo: {t.Name}");
    foreach (PropertyInfo prop in t.GetProperties()) {
        object? valore = prop.GetValue(obj);
        Console.WriteLine($"{prop.Name} = {valore}");
    }
}

Persona p = new Persona { Nome = "Alice", Eta = 30 };
StampaProprieta(p);
// Output:
// Tipo: Persona
// Nome = Alice
// Eta = 30
// (Tutte le proprietà pubbliche dell'oggetto sono state elencate)
```

Slide 9 – Introduzione agli attributi (Attributes)

Attributi e **metadati** sono strettamente collegati: un attributo è un **annotazione dichiarativa** che possiamo apporre a definizioni di classi, metodi, proprietà, ecc. per aggiungere informazioni aggiuntive. Gli attributi in .NET vengono compilati negli assembly e possono essere letti tramite reflection a runtime, ma spesso influenzano anche il comportamento a compile-time o di framework. Ad esempio, l'attributo integrato `[Obsolete]` marca un metodo o classe come deprecata. Il compilatore emette un avviso se tale metodo viene usato. Nell'esempio, applichiamo `[Obsolete]` a un metodo.

```
class EsempioAttr {
    [Obsolete("Questo metodo \u00e8 deprecato.")]   // attributo che indica
```

```
obsolescenza
    public void MetodoObsoleto() {
        Console.WriteLine("Met. obsoleto eseguito");
    }
}

var obj = new EsempioAttr();
obj.MetodoObsoleto();  // ⚠ Genera un avviso di compilazione: metodo obsoleto
// Output a runtime: "Met. obsoleto eseguito"
```

Slide 10 – Creare un attributo personalizzato

.NET permette di definire **attributi personalizzati** creando una classe che deriva da `System.Attribute`. Possiamo specificare quali elementi del codice possono riceverlo tramite `[AttributeUsage]`. Gli attributi personalizzati possono includere proprietà per trasportare dati. Nell'esempio creiamo un attributo `InfoAttribute` che aggiunge un'informazione di autore a una classe. Lo contrassegniamo con `AttributeUsage` per consentirne l'uso su classi e struct, e definiamo una proprietà `Autore` valorizzata via costruttore. Poi applichiamo `[Info("Mario Rossi")]` a una classe di esempio.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
class InfoAttribute : Attribute {
    public string Autore { get; }
    public InfoAttribute(string autore) {
        Autore = autore;
    }
}

// Uso dell'attributo personalizzato su una classe
[Info("Mario Rossi")]
class Documento {
    public void Stampa() { Console.WriteLine("Stampa documento..."); }
}
```

Slide 11 – Leggere gli attributi via Reflection

Per utilizzare i dati aggiuntivi forniti dagli attributi, ricorriamo alla reflection. Possiamo ottenere gli attributi applicati a un membro (classe, metodo, proprietà, ecc.) tramite metodi come `Attribute.GetCustomAttribute` o `MemberInfo.GetCustomAttributes`. Questi restituiscono le istanze degli attributi, attraverso cui accedere alle relative proprietà. Riprendendo l'esempio precedente, leggiamo a runtime l'attributo `InfoAttribute` dalla classe `Documento` e ne stampiamo la proprietà `Autore`.

```
Type tipoDoc = typeof(Documento);
InfoAttribute? info = (InfoAttribute?) Attribute.GetCustomAttribute(tipoDoc,
typeof(InfoAttribute));
if (info != null) {
```

```
    Console.WriteLine($"Autore del documento: {info.Autore}");
}
// Output: Autore del documento: Mario Rossi
```

Slide 12 – Esempio: attributi personalizzati in azione

Gli attributi personalizzati sono utili per collegare metadati e logica del programma. Un caso pratico: mappare campi di una classe ai nomi di colonne di un database. Possiamo definire un attributo `[Column]` che specifica il nome della colonna, applicarlo alle proprietà di una classe modello, e poi usare reflection per leggere questi metadati e automatizzare operazioni (es. generazione di query SQL, validazioni). Nell'esempio, applichiamo un attributo `ColumnAttribute` a proprietà della classe `Utente` e poi leggiamo tali mapping.

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

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

// Reflection: leggere il nome di colonna per ogni proprietà di Utente
foreach (PropertyInfo prop in typeof(Utente).GetProperties()) {
    var colAttr = (ColumnAttribute?) Attribute.GetCustomAttribute(prop,
typeof(ColumnAttribute));
    if (colAttr != null) {
        Console.WriteLine($"{prop.Name} -> colonna DB '{colAttr.Name}'");
    }
}
// Output:
// Nome -> colonna DB 'first_name'
// Cognome -> colonna DB 'last_name'
```

Slide 13 – Introduzione alla creazione dinamica di tipi

Oltre a creare istanze di tipi esistenti, .NET permette anche di **definire nuovi tipi a runtime**. Ciò avviene tipicamente usando l'API `System.Reflection.Emit`, che consente di emettere "al volo" assembly, tipi e metodi MSIL. Questo approccio avanzato è usato in scenari come la generazione di **proxy dinamici** (es. per librerie ORM, AOP) o la compilazione di espressioni LINQ in codice eseguibile. La creazione dinamica di tipi è complessa ma estremamente potente: permette al programma di estendere se stesso durante l'esecuzione. Nelle slide seguenti vedremo un esempio che genera un nuovo tipo con un metodo semplice.

```
// 1. Creare un oggetto AssemblyName per l'assembly dinamico
AssemblyName assemblyName = new AssemblyName("RuntimeTypes");
```

```
// 2. Definire un assembly dinamico (solo in esecuzione, non viene salvato su disco)
AssemblyBuilder assemblyBuilder = AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
// 3. Definire un modulo all'interno dell'assembly dinamico
ModuleBuilder moduleBuilder = assemblyBuilder.DefineDynamicModule("MainModule");
// 4. Creare un TypeBuilder per definire un nuovo tipo pubblico chiamato "SalutoDinamico"
TypeBuilder tipoDinamicoBuilder = moduleBuilder.DefineType("SalutoDinamico", TypeAttributes.Public);
```

Slide 14 – Aggiunta di membri al tipo dinamico

Una volta ottenuto il `TypeBuilder`, possiamo aggiungere membri (metodi, proprietà, campi) al tipo in costruzione. Nel nostro esempio, aggiungiamo un **metodo statico** `Hello` al tipo dinamico. Utilizziamo `DefineMethod` per specificare nome, attributi (qui `Public` e `Static`), tipo di ritorno (void) e eventuali parametri (nessuno in questo caso). Otterremo un `ILGenerator` per emettere le istruzioni IL del metodo. Nell'IL, usiamo `EmitWriteLine` per far stampare una stringa e infine `OpCodes.Ret` per terminare il metodo.

```
// 5. Definire un metodo pubblico statico Hello() sul tipo dinamico (ritorno void, nessun parametro)
MethodBuilder metodoBuilder = tipoDinamicoBuilder.DefineMethod(
    "Hello", MethodAttributes.Public | MethodAttributes.Static,
typeof(void), Type.EmptyTypes);
// 6. Ottenere un generatore IL per emettere istruzioni nel metodo Hello
ILGenerator il = metodoBuilder.GetILGenerator();
// 7. Emettere il codice IL: scrivere una linea di testo e quindi effettuare il return
il.EmitWriteLine("Ciao dal tipo dinamico!");
il.Emit(OpCodes.Ret);
```

Slide 15 – Completare e utilizzare il tipo dinamico

Dopo aver definito i membri, occorre "creare" effettivamente il tipo dinamico chiamando `CreateType()` sul `TypeBuilder`. Questo passaggio finalizza la definizione e produce un oggetto `Type` utilizzabile. Possiamo quindi ottenere normalmente il `MethodInfo` del metodo `Hello` appena definito e invocarlo. Nell'esempio, non essendo il metodo statico legato a un'istanza, passiamo `null` come target a `Invoke`. L'effetto sarà l'esecuzione del codice IL che stamperà il messaggio definito.

```
// 8. Creare il tipo runtime (completa la definizione del tipo dinamico)
Type tipoDinamico = tipoDinamicoBuilder.CreateType();
// 9. Ottenere il MethodInfo del metodo Hello definito sul nuovo tipo
MethodInfo metodoHello = tipoDinamico.GetMethod("Hello")!;
```

```
// 10. Invocare il metodo Hello dinamicamente (null perché è metodo statico)
metodoHello.Invoke(null, null);
// Output: Ciao dal tipo dinamico!
```

Slide 16 – Considerazioni su tipi dinamici e alternative

La creazione dinamica di tipi con Reflection.Emit è molto potente ma ha **costi di complessità e manutenzione** elevati. Si usa tipicamente in scenari specializzati (ad esempio, generazione di codice runtime per migliorare prestazioni di riflessione, oppure creazione di oggetti proxy in tool di logging/AOP). In molti casi, esistono alternative più semplici se serve solo flessibilità a runtime. Ad esempio, il **tipo dinamico** `dynamic` di C# permette chiamate late-bound senza dover emettere IL manualmente. Oppure si possono usare `ExpandoObject` per oggetti con proprietà aggiunte a runtime. Nell'esempio, usiamo `ExpandoObject` per creare un oggetto con membri dinamici.

```
dynamic persona = new ExpandoObject();
persona.Nome = "Roberto";
// Aggiungiamo un "metodo" Saluta dinamicamente come delegate
persona.Saluta = (Action)(() => Console.WriteLine($"Ciao, sono
{persona.Nome}"));
persona.Saluta();  // Invoca il metodo dinamico; stampa: "Ciao, sono Roberto"
```

Slide 17 – Introduzione ai generics (tipi generici)

I **generics** introdotti in C# permettono di definire classi, strutture, metodi e interfacce con **parametri di tipo**. Ciò consente di riutilizzare lo stesso codice per diversi tipi, mantenendo la sicurezza di tipo (type safety) e evitando cast e boxing. Ad esempio, `List<T>` è una lista generica che può contenere elementi di qualsiasi tipo specificato in fase di utilizzo, sostituendo le vecchie collezioni non tipizzate (`ArrayList`) e prevenendo errori di runtime. Nell'esempio, confrontiamo l'uso di una lista generica con una non generica.

```
// Collezione non generica (ArrayList) vs generica (List<T>)
ArrayList arr = new ArrayList();
arr.Add(123);                   // 123 viene impacchettato (boxing) come
object
int primo = (int) arr[0];       // necessario cast esplicito da object a int

List<string> nomi = new List<string>();
nomi.Add("Alice");              // aggiunta type-safe, solo stringhe
consentite
string nome = nomi[0];          // accesso tipizzato, nessun cast necessario
// nomi.Add(123);  // Errore di compilazione: 123 non è una string (il tipo è
imposto da <string>)
Console.WriteLine(nome);        // Output: Alice
```

Slide 18 – Metodi generici e classi generiche

I generics si applicano sia a classi/interfacce (definendo tipi "parametrizzati") sia a metodi. Un **metodo generico** è una funzione con parametri di tipo propri, che possono essere dedotti dagli argomenti. Ad esempio, possiamo scrivere un metodo generico di swap che scambia due variabili di qualsiasi tipo. Il vantaggio è che il codice è scritto una sola volta, ma funziona per int, string o qualunque altro tipo senza cast. Nell'esempio il metodo `Scambia<T>` scambia due valori e viene usato con int e string.

```
void Scambia<T>(ref T a, ref T b) {
    T temp = a;
    a = b;
    b = temp;
}

int x = 1, y = 2;
Scambia(ref x, ref y);
Console.WriteLine($"x={x}, y={y}");    // x=2, y=1

string s1 = "uno", s2 = "due";
Scambia(ref s1, ref s2);
Console.WriteLine($"s1={s1}, s2={s2}");  // s1=due, s2=uno
```

Slide 19 – Vincoli generici (constraints)

Di default, un parametro generico `T` può rappresentare **qualsiasi** tipo. Il compilatore, senza maggiori informazioni, consente sul `T` solo operazioni valide per ogni oggetto (metodi di `object`). I **vincoli** (parola chiave `where`) permettono di restringere i tipi ammessi per `T` garantendo determinate funzionalità. Ad esempio `where T : IDisposable` indica che T deve implementare l'interfaccia `IDisposable`. Così, all'interno del metodo generico potremo invocare `Dispose()` su `T` senza errori. Nell'esempio, definiamo un metodo generico che chiude (Dispose) una risorsa se il tipo lo supporta.

```
void ChiudiRisorsa<T>(T risorsa) where T : IDisposable {
    risorsa.Dispose();  // possibile solo se T implementa IDisposable
}

// Esempio d'uso: MemoryStream implementa IDisposable, quindi soddisfa il
vincolo
MemoryStream stream = new MemoryStream();
ChiudiRisorsa(stream);  // chiama Dispose() sul MemoryStream, liberando la
risorsa
```

Slide 20 – Vincoli `class`, `struct` e `new()`

Altri vincoli comuni sono `class` e `struct`, che restringono T rispettivamente a tipi riferimento (classi, interfacce, delegati, array) o tipi valore (struct). Ad esempio, `where T : class` permette `null` come valore di default di T, mentre `where T : struct` assicura che T sia un valore (non nullo,

e implica l'esistenza di un costruttore senza parametri). Il vincolo `new()` richiede che T abbia un costruttore pubblico senza parametri, utile per poter creare istanze di T dentro il codice generico. Nell'esempio, definiamo una fabbrica generica che richiede T sia una classe istanziabile con costruttore vuoto, così da poterne fare `new T()`.

```
class FabbricaOggetti<T> where T : class, new() {
    public T Crea() {
        return new
T();  // valido solo se T ha un costruttore pubblico senza parametri
    }
}

// Esempio: StringBuilder è una classe con costruttore senza parametri,
soddisfa i vincoli
FabbricaOggetti<StringBuilder> fab = new FabbricaOggetti<StringBuilder>();
StringBuilder sb = fab.Crea();   // crea dinamicamente un nuovo StringBuilder
```

(Nota: `where T : struct` *vincola T ai tipi valore non null (implica il costruttore vuoto); non può essere combinato con* `class` *o* `new()`*.)*

Slide 21 – Vincoli di interfaccia e di classe base

Possiamo vincolare un generico a una **classe base specifica** o a un'**interfaccia**. Ad esempio, `where T : Stream` limiterà T a Stream o sue sottoclassi, consentendo di usare i membri di `Stream` su T. Analogamente, `where T : IComparable` richiede che T implementi l'interfaccia IComparable. Si possono anche specificare più interfacce (es. `where T: IDisposable, ICloneable`). Nell'esempio seguente, vincoliamo T alla classe base `Persona`; così possiamo assumere che ogni T abbia la proprietà `Nome` definita in `Persona`.

```
class Persona { public string Nome { get; set; } }
class Studente : Persona { public int Matricola { get; set; } }

void StampaNome<T>(T obj) where T : Persona {
    Console.WriteLine($"Nome: {obj.Nome}");
}

StampaNome(new Studente { Nome = "Marco" });  // Output: Nome: Marco
// (Funziona perch\u00e9 Studente deriva da Persona, soddisfa il vincolo)
```

(È possibile combinare un vincolo di classe base con interfacce aggiuntive, e con `new()` *se necessario. Ad esempio:* `where T : Persona, IComparable<T>, new()`*.)*

Slide 22 – Vincoli avanzati: `notnull`, `unmanaged`, enum, delegate

Versioni recenti di C# hanno introdotto vincoli aggiuntivi:
- `notnull`: T può essere solo un tipo non nullable (esclude reference nullable e `Nullable<T>`). Utile

per prevenire `null` dove non ammesso.
- `unmanaged`: T deve essere un tipo valore "non gestito", cioè senza riferimenti a oggetti (tipicamente struct contenenti solo primitivi o altri unmanaged). Utile per interoperabilità nativa o ottimizzazioni (permette usare `sizeof(T)` e copiare blocchi di memoria).
- È ora possibile vincolare `T` a tipi specifici come **enum** o **delegate** usando rispettivamente `where T : System.Enum` e `where T : System.Delegate` (o `MulticastDelegate`). Questo consente operazioni specifiche su questi tipi.

Nell'esempio, usiamo il vincolo `T : Enum` per accettare solo tipi enum e verifichiamo se un valore enum è definito nelle costanti ammesse.

```
enum Colore { Rosso, Verde, Blu }

bool EValoreDefinito<T>(T valore) where T : Enum {
    return Enum.IsDefined(typeof(T), valore);
}

Console.WriteLine(EValoreDefinito(Colore.Rosso));       // True  (Rosso è
definito)
Console.WriteLine(EValoreDefinito((Colore)999));        // False (999 non è
definito in Colore)
```

(Altri vincoli speciali includono `default` *(per membri generici override) e* `scoped` *per* `ref struct` *in C# 11, usati in contesti avanzati.)*

Slide 23 – Covarianza dei tipi generici

La **covarianza** permette di usare un tipo generico più derivato come se fosse il suo genitore in posizioni **output**. In C#, alcune interfacce e delegate generici sono dichiarati covarianti (con il modificatore `out` sul parametro di tipo). Un esempio classico è `IEnumerable<out T>`: essendo covariante, una `IEnumerable<string>` può essere assegnata a una variabile di tipo `IEnumerable<object>` perché una sequenza di stringhe può essere trattata come una sequenza di object (tutti i tipi derivano da object). Attenzione: la covarianza funziona solo per la lettura (output), non sarebbe sicuro aggiungere un object qualsiasi a una lista di stringhe. Nell'esempio si mostra l'assegnamento covariante e l'iterazione.

```
IEnumerable<string> stringhe = new List<string> { "uno", "due" };
IEnumerable<object> oggetti = stringhe;  // Covarianza: IEnumerable<string> -
> IEnumerable<object>
foreach (object obj in oggetti) {
    Console.WriteLine(obj);
}
// Output:
// uno
// due
```

Slide 24 – Contravarianza dei tipi generici

La **contravarianza** è l'opposto: consente di usare un tipo più generale dove se ne aspetta uno più specifico, in contesti **input**. Delegati come `Action<T>` o interfacce come `IComparer<T>` sono contravarianti (dichiarati con `in T`). Ciò significa che un `Action<object>` (che accetta qualsiasi oggetto) può essere assegnato a una variabile `Action<string>` – che verrà usata solo con stringhe – perché ogni stringa è anche un object e la funzione che gestisce oggetti può gestire stringhe in ingresso. Questo meccanismo è sicuro per parametri in input. Nell'esempio vediamo `Action<object>` assegnato ad `Action<string>` e utilizzato.

```
Action<object> stampaOggetto = obj => Console.WriteLine($"Oggetto: {obj}");
Action<string> stampaStringa = stampaOggetto;   // Contravarianza:
Action<object> -> Action<string>
stampaStringa("Hello");  // in realt\u00e0 chiama stampaOggetto, che accetta
string come object
// Output: Oggetto: Hello
```

Slide 25 – Esempio avanzato: classe generica con vincolo

Per illustrare l'utilità dei generics con vincoli, consideriamo la realizzazione di un **repository** generico che gestisce entità con un identificatore. Definiamo un'interfaccia `IEntity` che richiede una proprietà Id, e una classe generica `Repository<T>` vincolata a `T : IEntity`. La repository può aggiungere elementi di tipo T e cercarli per Id, sfruttando il fatto che T ha sicuramente la proprietà Id. Questo evita di dover scrivere repository separate per Utente, Ordine, ecc., e garantisce a compile-time che solo tipi validi (con Id) vengano usati.

```
interface IEntity { int Id { get; } }

class Repository<T> where T : IEntity {
    private List<T> elementi = new List<T>();
    public void Add(T item) {
        elementi.Add(item);
    }
    public T? GetById(int id) {
        foreach (T item in elementi) {
            if (item.Id == id) return item;
        }
        return default;
    }
}

class Utente : IEntity {
    public int Id { get; set; }
    public string Nome { get; set; }
}

var repo = new Repository<Utente>();
repo.Add(new Utente { Id = 1, Nome = "Alice" });
```

```
Utente? u = repo.GetById(1);
Console.WriteLine(u?.Nome);  // Output: Alice
```

Slide 26 – Generics e membri statici

I generics in .NET sono implementati con **riutilizzo del codice** generato JIT per i tipi reference e specializzazione per i tipi value. Un dettaglio importante è che i membri **statici** di una classe generica esistono separatamente per ogni chiusura di tipo. In altre parole, `ClasseGenerica<int>` avrà le proprie copie di membri statici diverse da `ClasseGenerica<string>`. Questo può essere sfruttato, ad esempio, per tenere contatori separati per ciascun tipo T. Nell'esempio, definiamo una classe generica con un campo statico incrementato e osserviamo che, a seconda di T, il valore statico è distinto.

```
class Contatore<T> {
    public static int Conteggio;
}

Contatore<int>.Conteggio = 5;
Contatore<string>.Conteggio = 10;
Console.WriteLine(Contatore<int>.Conteggio);    // 5  (static separato per
T=int)
Console.WriteLine(Contatore<string>.Conteggio); // 10 (static separato per
T=string)
```

(Ciò avviene perché a runtime `Contatore<int>` *e* `Contatore<string>` *sono tipi distinti; i generics .NET non usano type erasure, ma conservano le informazioni di tipo durante l'esecuzione.)*

Slide 27 – Generics e reflection: creazione di tipi chiusi a runtime

Le funzionalità di generics e reflection possono combinarsi. Ad esempio, possiamo creare istanze di tipi **generici** conoscendo solo il tipo a runtime. Immaginiamo di voler creare una `List<T>` di un tipo determinato a runtime: possiamo partire dal tipo generico aperto `List<>` e usare `MakeGenericType` per ottenere il Type della lista chiusa (es. `List<int>`). Poi, con `Activator.CreateInstance`, instanziarla e perfino chiamare metodi generici riflettivamente. Nell'esempio creiamo una lista di int a runtime e vi aggiungiamo un elemento tramite reflection.

```
Type listaAperta = typeof(List<>);                    // tipo generico
aperto List<T>
Type listaIntType = listaAperta.MakeGenericType(typeof(int));  // costruisce
il tipo List<int>
object listaInt = Activator.CreateInstance(listaIntType)!;     // crea
istanza di List<int>
MethodInfo metodoAdd = listaIntType.GetMethod("Add")!;
metodoAdd.Invoke(listaInt, new object[]{ 42 });         // invoca
List<int>.Add(42)
Console.WriteLine(((List<int>)listaInt)[0]);            // Cast ad List<int>
```

```
per leggere il valore
// Output: 42
```

Slide 28 – Esempio avanzato: operatori numerici generici (C# 11)

Tradizionalmente, ai tipi generici mancava la possibilità di utilizzare operatori aritmetici (+, -, ...) *perché il compilatore non poteva sapere se T li supporta. Con C# 11 (.NET 7) sono state introdotte interfacce come* `INumber<T>` *(in* System.Numerics*) che definiscono operatori statici, permettendo di vincolare T a "comportarsi come un numero". Questo abilita scenari di generic math: algoritmi matematici generici validi per int, float, decimal, ecc. Nell'esempio, il metodo* `Somma<T>` *vincola T a* `INumber<T>` *così da poter usare l'operatore* `+` *su T, e lo usiamo sia con int che con double.* (Richiede .NET 7+)*

```
using System.Numerics;  // contiene INumber<T> e interfacce numeriche

T Somma<T>(T a, T b) where T : INumber<T> {
    return a + b;  // operatore + disponibile grazie al vincolo INumber<T>
}

Console.WriteLine(Somma(10, 20));      // 30   (T dedotto come int)
Console.WriteLine(Somma(1.5, 2.5));    // 4.0  (T dedotto come double)
```

Slide 29 – Applicazioni pratiche di reflection, attributi e generics

Le tecniche descritte trovano impiego in molti contesti reali:
- **ORM e mappatura oggetti-relazionale**: ad es. Entity Framework usa attributi come `[Key]`, `[Column]` per configurare mapping DB, e reflection per leggere queste info ed eseguire query/sql generation.
- **Serializer** (es. Newtonsoft Json, System.Text.Json): ispezionano via reflection le proprietà pubbliche di un oggetto e utilizzano attributi (es. `[JsonProperty]`, `[Ignore]`) per decidere come serializzare/deserializzare i campi.
- **Dependency Injection**: i contenitori DI (Autofac, .NET Core DI) impiegano reflection per trovare costruttori appropriati e creare istanze delle classi richiesta in modo automatico, e spesso usano attributi per configurazioni (es. `[Inject]`).
- **Framework di testing**: NUnit, MSTest, xUnit usano attributi (es. `[Test]`, `[TestMethod]`) per individuare i metodi di test da eseguire, e reflection per invocarli.
- **Librerie generiche**: le collezioni e algoritmi generici in .NET (es. LINQ) consentono di riutilizzare codice efficiente per tipi diversi senza riflessione (ad es. `List<T>` per qualsiasi T, metodi LINQ generici come `.Where()` che funzionano su sequenze di qualsiasi tipo).

```
// (Esempi d'uso reale descritti sopra; vedere spiegazioni nei punti elenco)
```

Slide 30 – Conclusioni e best practice

In conclusione, riflessione, attributi, tipi dinamici e generics sono strumenti avanzati che arricchiscono C# e .NET:
- **Reflection**: offre grande flessibilità (es. plugin, ORM, serializer) ma va usata con moderazione per il costo in performance e complessità. Meglio limitarla a punti centralizzati (cache i risultati di reflection se

usata intensivamente).
- **Attributi**: permettono di aggiungere metadati dichiarativi al codice. Sfruttali per rendere configurazioni e comportamenti estensibili senza hard-code (seguendo le convenzioni dei framework).
- **Creazione dinamica di tipi**: potente ma da riservare a esigenze particolari (es. generazione di codice runtime per ottimizzazione, profiling, AOP). Valuta alternative come il Dynamic proxy pattern o i Source Generators a compile-time per scenari meno dinamici.
- **Generics**: privilegia i generics per scrivere codice riutilizzabile e type-safe invece di ricorrere a `object` + cast o a dynamic. I generics producono codice efficiente (niente overhead di riflessione) e rendono esplicite le aspettative sui tipi. Usa i vincoli per imporre requisiti su T in modo che il codice generico sia il più forte possibile (poter chiamare metodi su T, ecc.).

```
Console.WriteLine("Fine del set di slide – buon apprendimento e buon
coding!");
```
