# Esercitazione di Fine Settimana – Week 2

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Nome | Debora |
|  |  | Cognome | Spano |
|  |  | Data | 17/09/2021 |

Leggete attentamente ogni domanda e argomentare quanto più possibile fornendo anche degli esempi.

1. Descrivere le modalità per ritornare più valori da un metodo in C#

In c# esistono 3 metodi per restituire più valori:

- restituire una struttura con all’interno tutti i valori che ci servono. Questo metodo è chiamato request/response pattern ed è rappresentato come segue:

public Response (Request request)

{

//codice

return response;

}

In questo caso incampuliamo tutti i dati che ci servono nella struttura Response e nascondiamo al cliente la vera forma del valore restituito.

- avere un solo valore di ritorno ma utilizzare la keyword out per ottenere in output dei parametri modificati all’interno della funzione. La sua rappresentazione è:

public bool Method(int param1, out int param2, out int param3)

{

// codice

return true;

}

In questo caso I parametri preceduti da “out” non sono semplici copie del dato (come accadrebbe di solito) ma vengono trattati come parametri passati per riferimento.

La chiamata ad una funzione con parametri out è:

bool b = a.Method(f, out int c, out int d);

I parametri “out” devono sempre stare alla fine della lista dei parametri.

- restiruire più valori tramite una tupla. La tupla è un insieme di dati (non per forza dello stesso tipo) ordinato.

public (bool, int, string) Method ()

{

// codice

return (false, 4, “stringa”);

}

Per estrarre I valori dalla tupla ottenuta possiamo usare il seguente modo

(bool success, int num, string str) = a.Method();

1. Descrivere le due tipologie di casting tra tipi in C#

Esistono due tipi di castingo: il casting implicito e il casting esplicito.

La principale differenza tra i due si vede nella modalità con cui vengono effettuati.

Il casting implicito viene fatto in automatico, quindi si può scrivere:

double a = 5;

in questo caso viene fatto un casting implicito da int a double e possiamo farlo senza nessun problema perchè non abbiamo alcuna perdita di dato (infatti otterremo 5.0 nell’esempio)

Il casting esplicito invece non è automatico è per poterlo fare abbiamo bisogno di scrivere il tipo che vogliamo ottenere tra parentesi, davanti al tipo da castare, in questo modo:

int a = (int)5.6;

In questo caso abbiamo una perdita di dati (infatti otteremo 5 nell’esempio).

I cast sono considerati degli operatori e ne possiamo fare l’overloadin. Per definire un cast implicit dobbiamo scrivere:

public static implicit operator Val1(Val2 v)

{

//codice

}

Dove Val1 è il tipo che si vuole ottenere castando Val2. In questo modo

Val1 val1 = val2; //val2 è di tipo Val2

Per definire un cast esplicito invece dobbiamo scrivere:

public static explicit operator Val1 (Val2 v)

{

//codice

}

E otteniamo il cast in questo modo

Val1 val1 = (Val1) val2;

Per riconosce se un cast è possibile o no si possono usare due diversi modi:

Il primo consiste nel tentare un cast dentro un blocco try/catch e gestire l’eventuale eccezione lanciata nel caso il cast non sia possibile;

try

{

Val1 val = (Val1) val2;

}

Catch(Exception ex)

{

//Errore

}

il secondo usa la keyword “is” che controlla se una determinata variabile è di un tipo (o di una sua sottoclasse)

if(val2 is Val1)

val1 = (Val1) val2;

Un’altra keyword utilizzata con i cast è “as” che converte in modo sicuro un dato e se il cast non è possibile assegna nell al dato stesso

Val1 val1 = val2 as Val1;

If(val2 == null)

//errore

1. Quali sono gli utilizzi della keyword static?

La keyword static può essere associata a membri e classi.

Se viene associata ad una classe otteniamo un tipo che non può essere istanzata (non si può fare la new() di quella classe) perchè di fatto esiste una sola istanza. Una classe statica può contenere solo membri statici.

Se associamo static ad un metodo quello che viene creato prenderà il nome di metodo di classe. Non si può richiamare un metodo statico dal nome di un istanza ma si utilizzerà il nome della classe:

MyClass.StaticMethod();

Questo perchè il metodo non “appartiene” alle istanze, ma alla classe. Infatti il metodo sarà lo stesso per ogni istanza creata da quel tipo

Se associamo invece static ad un campo o proprietà otterremo, come per il metodo, un campo/proprietà di classe. Quindi essa non sarà chiamabile dall’istanza ma solo dalla classe

MyClass.StaticProp;

1. Descrivere le modalità di implementazione di un evento in C#

Un evento è un membro di una classe è viene usato per notificare tutti i suoi “iscritti” quando esso viene lanciato

Vengono gestiti tramite dei delegate (puntatori a un metodo);

Per creare un evento bisogna quindi dichiararlo come membro di una classe e dargli come tipo un delegate

Public event DelegateType EventName;

A questo punto per lanciarlo basta richiamarlo nel punto in cui si vogliono notificare gli iscritti (stando attenti a controllare che l’evento non sia nullo, ossia che abbia almeno un iscritto).

Ogni iscritto può registrarsi ad un evento tramite l’operatore += in questo modo:

subscribe.EventName += EventHandle;

l’EventHandle è il metodo che avrà il compito di gestira la notifica che riceverà dall’evento.

In parole povere: l’evento viene lanciato, l’event-handle riceve la notifica e esegue una serie di azioni. L’event handle inoltre deve sempre rispettare la firma del delegate con cui è stato creato l’evento

Esistono due classi che permettono di gestire gli eventi senza creare il delegate: EventHandler e EventHandle<EventArgs>.

La prima classe non prende argomenti, quindi è utile per tutti gli eventi che non devono passare niente agli iscritti.

La seconda prende un EventArgs. EventArgs è inutile perchè non contiene nessun dato utilizzabile, ma si può creare una propria classe Args (che erediterà da EventArgs) per poter passare che parametro i dati che ci servono.

1. A cosa serve l’interfaccia IEnumerable<T>? Come si implementa in una nostra

classe?

L’interfaccia Ienumerable<T> è un interfaccia generica. Il simbolo <T> rapresenta il fatto che l’interfaccia può prendere qualunque T tipo. Essa permette di creare una classe che è enumerabile, cioè che può contenere (e essere) una collezione di oggetti (di tipo T).

Se una classe implementa Ienumerable<T> è costretto a implementare due metodi:

Public Ienumerator<T> GetEnumerator(){}

public Ienumerator Ienumerable.GetEnumerator(){}

Il primo metodo restituisce una alla volta gli elementi della collezione senza dimenticare il punto in cui è arrivato (grazie alla keyword yield). Viene usato, per esempio, dal foreach.

La sua implementazione base è la seguente

Public Ienumerator<T> GetEnumerator()

{

Foreach(var v in lista)

Yield return v;

}

Il secondo metodo fa la stessa cosa ma non restituisce oggetti di tipo T ma di tipo object.

Di solito viene implementata solo per rimandare la computazione alla GetEnumerator() che restituisce il tipo T, per fare in modo che se in alcune parti del framework viene chiamata questa essa in ogni caso restisuisce la computazione della GetEnumerator “ufficiale”

Quindi la sua implementazione è

public Ienumerator Ienumerable.GetEnumerator()

{

Return this.GetEnumerato();

}

Si preferisce implementare questa interfaccia piuttosta che creare semplicemente una lista interna perchè in questo modo si fa un lavoro di incapsulamento sulla collazione. Si può rendere privata la variabile che rappresenta la collezione e nascondere all’utente la sua vera natura. In questo modo si possono anche controllare le aggiunte e le rimozioni in modo più accurato.

**Esercitazione Pratica**

* Realizzare una classe Warehouse per gestire un Magazzino Merci, con le seguenti proprietà:
  + *Id Magazzino (GUID)*
  + *Indirizzo*
  + *Importo Totale Merci in giacenza*
  + *Data Ultima Operazione*
  + *Lista delle Merci in giacenza*
* Realizzare **l’overload degli operatori + e –** in modo che sia possibile aggiungere e rimuovere Merci dalla lista (l’overload dovrà anche occuparsi di aggiornare l’Importo Totale e la Data di Ultima Operazione)
* Realizzare un metodo StockList() che stampi i dati del Magazzino, inclusa la lista delle Merci in giacenza
* Realizzare una gerarchia di classi per rappresentare le Merci (Good). Tutte le classi avranno le proprietà
  + *Codice Merce*
  + *Descrizione*
  + *Prezzo*
  + *Data di Ricevimento*
  + *Quantità in Giacenza*
  + Realizzare le classi che rappresentano:
    - ElectronicGood, con la proprietà aggiuntiva *Produttore*
    - PerishableGood, con le proprietà aggiuntive *Data di Scadenza* e *Modalità di Conservazione* (enum con i valori FREEZER, FRIDGE e SHELF)
    - SpiritDrinkGood, con le proprietà aggiuntive *Tipo* (enum con i valori WHISKY, WODKA, GRAPPA, GIN e OTHER) e *Gradazione Alcoolica*
  + Tutte le classi saranno dotate di costruttore che accetti tutti i parametri necessari per popolare le proprietà
  + Tutte le classi dovranno implementare la propria versione del metodo ToString() e visualizzare tutti i dati
  + ***OPZIONALE***: realizzare una procedura di caricamento dati da un file della lista delle Merci in giacenza (l’implementazione di eventi per notificare le fasi di caricamento dati costituisce un bonus)
  + ***OPZIONALE 2***: utilizzare una (o più) eccezione custom per gestire tutte le tipologie di errori che dovessero verificarsi durante l’utilizzo della classe Warehouse e delle classi Good
  + ***OPZIONALE 3:*** implementare IEnumerable
* Realizzare una Console app che
  + Crei un nuovo Magazzino
  + Permetta di ricevere diverse tipologie di Merci (gestire l’input dall'utente)
  + Stampi i dati del Magazzino e le Merci in giacenza