

C#ADV - Week 3




Antonia Sacchitella

Analyst@icubedsrl

antonia.sacchitella@icubed.it



Week 3 - Agenda

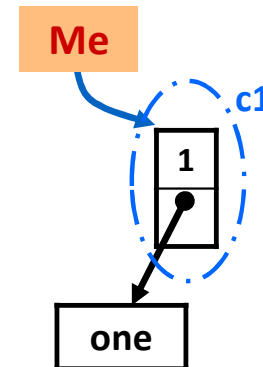
- Approfondimenti di classi
- Polimorfismo
- Overloading e Overriding dei metodi
- System.Collections
- Tipi Generici
- Nullable
-  Esercitazione



Keyword this

this è un riferimento che punta all'istanza della classe stessa.
È usabile solo in relazione ai membri non statici.

```
public class MyClass {  
    int one;  
  
    public void MyMethod(int one) {  
        this.one = one;  
    }  
}
```



Costruttore

Il costruttore è fondamentale perché permette di dare all'oggetto uno stato iniziale stabile e congruo.

Il costruttore di default non ha argomenti.

```
public class Person {  
    private string _name;  
    private int _age;  
  
    public Person(string name, int age) {  
        _name = name;  
        _age = age;  
    }  
}
```

Overloading di metodi e proprietà

Possono esistere metodi e proprietà con lo stesso nome.

È possibile perché il vero “nome” è rappresentato dalla **firma**: nome, numero e tipi dei parametri, ivi inclusi i modificatori come **ref** o **out**.

Non possono esistere due metodi che differiscono del solo parametro di ritorno (non fa parte della firma).

```
public int Sum(int a, int b) {  
    return a + b;  
}
```

```
public decimal Sum(decimal a, decimal b) {  
    return a + b;  
}
```

```
public int Sum(int a, int b, int c) {  
    return a + b + c;  
}
```

```
public decimal Sum(Decimal a, Decimal b, Decimal c) {  
    return a + b + c;  
}
```

Classi annidate

Le classe annidata è semplicemente un tipo definito all'interno di un'altra classe.

```
public class MyClass {  
    public class MyNestedClass {  
        //...  
    }  
}  
  
// occorre specificare il nome della classe container  
MyClass.MyNestedClass nested = New MyClass.MyNestedClass();
```

Demo

Classi

Costruttori, this e Overloading



Strutture

Quanto visto finora per le classi è tipicamente valido anche per la definizione di strutture (keyword **struct**).

Il compilatore genera sempre un costruttore di default che inizializza tutti i membri della struttura al valore di default.

È possibile dichiarare unicamente costruttori con parametri.

```
struct MyStructure {  
    public MyStructure(string a) {  
        ValueOne = int.Parse(a);  
        ValueTwo = false;  
    }  
  
    public int ValueOne;  
    public bool ValueTwo;  
}
```


Classi vs Strutture

Classi	Strutture
Possono definire data member, proprietà, metodi.	Possono definire data member, proprietà, metodi.
Supportano costruttori e l'inizializzazione dei membri.	Non supportano costruttori di default e l'inizializzazione dei membri.
Supportano il metodo Finalize .	Non supportano il metodo Finalize .
Supportano l'ereditarietà.	Non supportano l'ereditarietà.
È un Reference Type.	È un Value Type.

Demo

Classi e Strutture



Ereditarietà

Si applica quando tra due classi esiste una relazione “è un tipo di”. Esempio: **Customer** è un tipo di **Person**.

Consente di specializzare e/o estendere una classe.

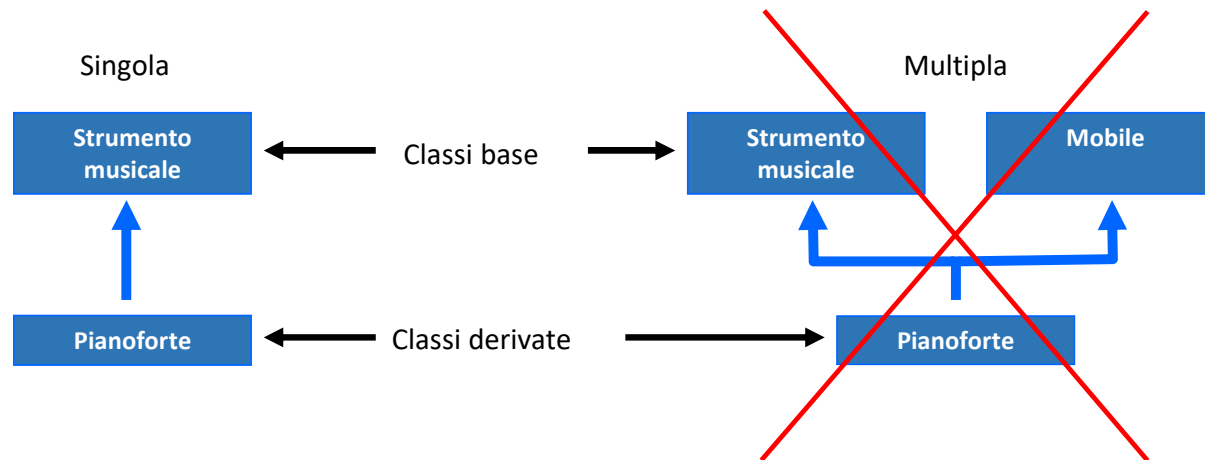
Si chiama *ereditarietà* perché la classe che deriva (classe derivata) può usare tutti i membri della classe ereditata (classe base – keyword **base**) come se fossero propri, ad eccezione di quelli dichiarati privati.

```
public class Person {  
    protected string name;  
}  
  
public class Customer : Person {  
  
    public void ChangeName(string newName) {  
        base.name = newName;  
    }  
}
```

Ereditarietà singola e multipla

Non è ammessa l'ereditarietà multipla.

Una classe può derivare unicamente da una sola altra classe.



Polimorfismo

Il *polimorfismo* è la possibilità di trattare un'istanza di un tipo come se fosse un'istanza di un altro tipo.

Il polimorfismo è subordinato all'esistenza di una relazione di derivazione tra i due tipi.

Affinchè un metodo possa essere polimorfico, deve essere marcato come **virtual** o **abstract**.



```
public class Strumento
{
    public virtual void Accorda() { }
}

public class Violino : Strumento
{
    public override void Accorda()
    {
        base.Accorda();
    }
}

public class Orchestra
{
    public Strumento violino, chitarra, pianoforte;

    public Orchestra()
    {
        violino = new Violino();
        violino.Accorda();
    }
}
```

Classi astratte e metodi virtuali

Una classe è astratta (**abstract**) se contiene almeno un metodo astratto.

Un metodo è astratto se dichiara la sua firma, ma non fornisce alcuna implementazione.

Se una classe derivata deve poter provvedere una nuova implementazione di un metodo, nella classe base questo deve essere contrassegnato come virtuale (**virtual**).

La classe derivata che voglia (o debba) fornire una implementazione sostitutiva di una classe base deve marcare con **override** il metodo.

Classi astratte e metodi virtuali

```
public abstract class Strumento
```

```
    public abstract void Accorda();
```

```
    public virtual void PausaTempo() {  
        // ...  
    }
```

```
}
```

```
public class Piano : Strumento
```

```
    public override void Accorda() {  
        // ...  
    }
```

```
    public override void PausaTempo() {  
        // ...  
    }
```

```
}
```

Modifier predefiniti

Per tipi

- **abstract**
- **sealed**

Il tipo deve essere derivato.

Il tipo non può essere derivato.

Per membri

- **static**

Non è un membro dell'istanza, ma del tipo.

Per metodi

- **static**
- **virtual**
- **new**
- **override**
- **abstract**

Il metodo è associato al tipo non all'istanza.

Il tipo derivato può eseguire l'override.

Maschera il metodo del tipo base.

Ridefinisce il metodo del tipo base.

Il tipo derivato deve eseguire l'override.

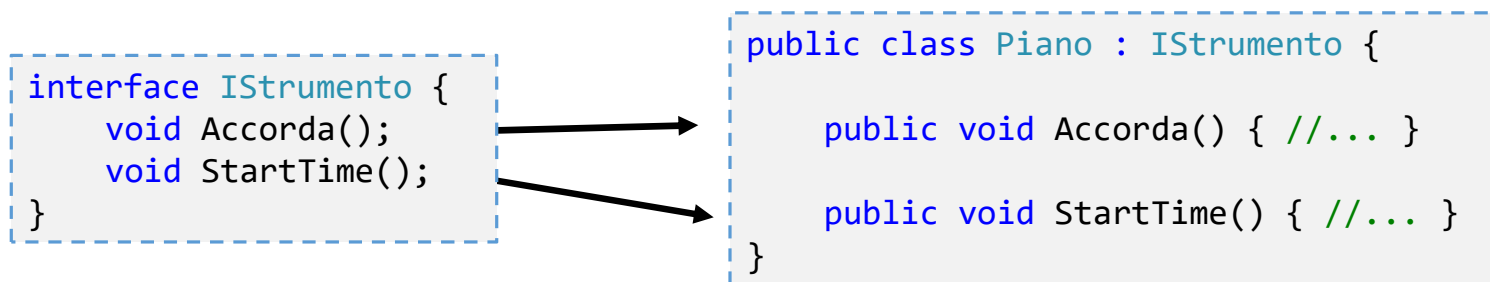
Interfacce

Un'**interfaccia** è simile ad una classe astratta pura, ossia con soli metodi e/o proprietà astratte.

Un'interfaccia è priva di qualsiasi implementazione e di modificatore di accessibilità (**public**, **private**, ecc.).

Un'interfaccia definisce un contratto che la classe che la implementa deve rispettare.

Una classe può implementare più interfacce contemporaneamente.



Perché usare le interfacce?

Perché rappresentano tipi astratti che permettono di limitare l'accoppiamento.

Perché in .NET l'ereditarietà multipla non è ammessa.

Per creare dipendenze tra i tipi senza ricorrere all'ereditarietà.

Per caricare assembly in modo dinamico e realizzare un sistema pluggabile.

Esercitazione n.1

Aggiungere come sottoclassi dell'esercitazione precedente la classe moto e la classe automobile

Le moto hanno un'altra informazione che riguarda il numero di cilindri, mentre le auto hanno il continente in cui è stata costruita.

Rivedere la gerarchia inserendo una classe astratta.

Dato un veicolo, il sistema calcola la sua valutazione. La valutazione sarà calcolata come:

"prezzo di acquisto" – ("prezzo di acquisto" * "**coefficiente di svalutazione**").

Il coefficiente di svalutazione si calcola in modo differente a seconda che la vettura sia un'auto o una moto:

- Moto: $CS = (((2021 - \text{"anno di immatricolazione"}) + 1) / 30) * (\text{"numero di cilindri"}) / 10$
- Auto: $CS = (((2021 - \text{"anno di immatricolazione"}) + 1) / 30) * \text{"fattore continente"}$. Il fattore continente vale 0,8 se l'auto è prodotta in europa o america del nord, altrimenti vale 1



Esercizio Aggiuntivo

Scrivere la classe Dipendente che ha i seguenti attributi:

matricola (string), **stipendio** (double), **straordinario** (double).

- Specificare un costruttore con parametri per istanziare completamente l'oggetto.
- Scrivere il metodo **Paga(int oreStraordinario)** che ha come parametro un numero intero indicante le ore di straordinario effettuate dal dipendente, il metodo deve poter restituire il calcolo dello stipendio comprensivo del pagamento delle ore di straordinario.
- Specificare il metodo **ToString()** per la conversione da istanza a stringa.

Specificare una sottoclasse DipendenteInterno, sottoclasse della classe Dipendente. La sottoclasse deve prevedere una proprietà aggiuntiva **malattia** (int) che specifica i giorni di malattia richiesti.

- Inserire il metodo **PrendiMalattia(int giorniMalattia)**, questo metodo ha il solo scopo di modificare la proprietà definita su.
- Ridefinire il metodo della superclasse Paga(int oreStraordinario): alla paga di base verrà sottratto il valore ottenuto da (giorniMalattia * 15)
- Ridefinire opportunamente il ToString() della classe appena creata.



Nullable

Le variabili di tipo Type Reference **possono essere** null

Le variabili di tipo Value **non possono essere** null

Problema tipico: mappatura tra classi e tabelle del database

- I valori nel db possono essere **null**
- I valori dei campi delle classi non lo possono essere (es. Int, double)

Nullable

Con i reference types è necessario il Garbage Collector per **liberare la memoria**, mentre con i tipi a valore questa operazione non è necessaria

I tipi a valore vengono liberati dalla memoria non appena **non sono più all'interno del loro scope**

Nullable

In C# possiamo utilizzare i **tipi Nullable**

Come dice il nome, sono tipi che **possono essere null**

E' sufficiente utilizzare il **carattere “?”** dopo un tipo per renderlo nullable:

```
int x1 = 1; // intero "tradizionale"  
int? x2 = null; // intero "nullabile"
```

L'assegnazione x1 a x2 non genera problemi

L'assegnazione da x2 a x1 non è possibile. E' necessario il cast:

```
x1 = (int) x2
```

Nullable

Il tipo nullable ha anche associato il **metodo** **.HasValue()** che ritorna true o false a seconda che il tipo abbiamo un valore o meno.

Nella **proprietà** **.Value** possiamo recuperare il valore

Nullable

Il tipo nullable ha anche associato il **metodo** **.HasValue()** che ritorna true o false a seconda che il tipo abbiamo un valore o meno.

Nella **proprietà** **.Value** possiamo recuperare il valore:

```
int x5 = x3.HasValue ? x3.Value : -1;
```

Ma anche:

```
int x6 = x3 ?? -1;
```

Demo

Nullable



Collections

Necessità di raggruppare oggetti omogenei.

- Array, oggetti della classe System.Array
- ArrayList – Collezione che più si avvicina ad un array

Metodo	Descrizione
Add(item) / Insert(index, item)	Aggiunta di un elemento nell'array list
AddRange(items) / Insert(index, items)	Aggiungi un insieme di elementi
Clear() / Remove(item) / RemoveAt(index)	Rimozione di un elemento nella lista
Contains(item)	Verifica che un elemento sia contenuto o meno
Count	Restituisce il numero di elementi contenuti
ToArray()	Genera un array a partire dal contenuto
LastIndexOf(item)	Ritorna l'indice dell'ultima occorrenza

Collections

Definite in **System.Collections**

Definite in **System.Collections.Generic**

Differenti specializzazioni per **differenti utilizzi**

System.Collection

Contiene le seguenti Collections:

ArrayList

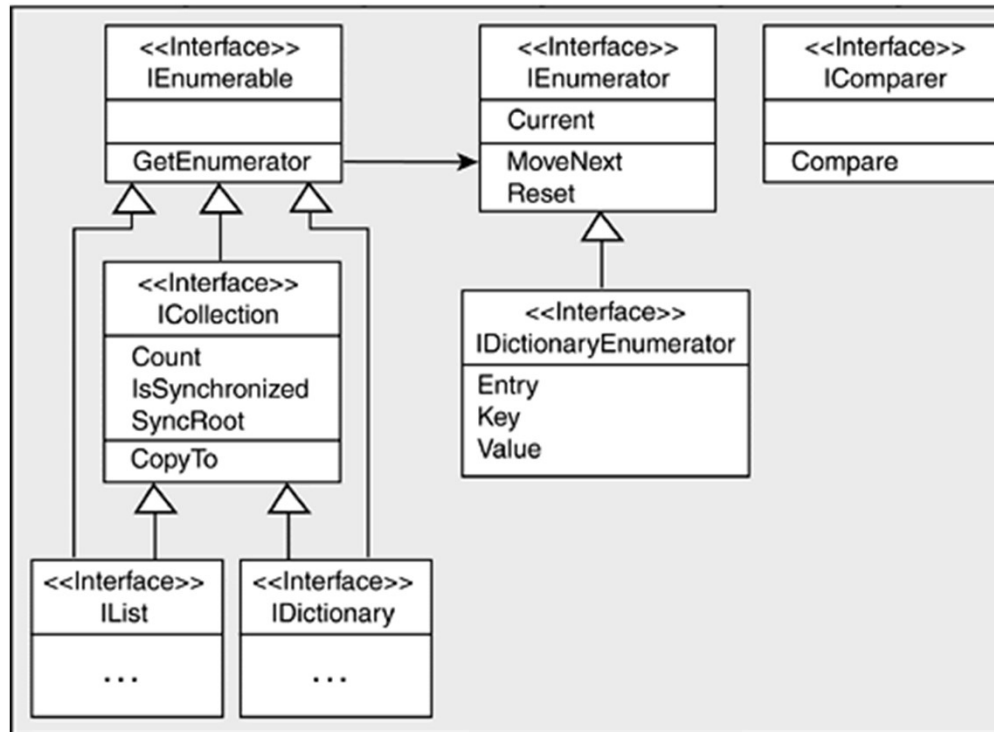
Stack

Queue

HashTable

Supportano solo tipizzazione debole

System.Collection



Demo

Collections



Tipizzazione debole di System.Collections

Le collezioni di oggetti definiti fino ad ora consentono di utilizzare collezioni di qualsiasi tipo di oggetti.

Questo implica la necessità di eseguire il cast per utilizzare gli oggetti opportunamente. Può però provare *InvalidCastException*

System.Collections.Generic

Contiene le seguenti Collections:

List<T>

Dictionary<Tkey, Tvalue>

HashSet<T>

Stack<T>

Queue<T>

...

Supportano Tipizzazione forte

Collections

INTERFACE	DESCRIPTION
<code>IEnumerable<T></code>	The interface <code>IEnumerable</code> is required by the <code>foreach</code> statement. This interface defines the method <code>GetEnumerator</code> , which returns an enumerator that implements the <code>IEnumerator</code> interface.
<code>ICollection<T></code>	<code>ICollection<T></code> is implemented by generic collection classes. With this you can get the number of items in the collection (<code>Count</code> property), and copy the collection to an array (<code>CopyTo</code> method). You can also add and remove items from the collection (<code>Add</code> , <code>Remove</code> , <code>Clear</code>).
<code> IList<T></code>	The <code>IList<T></code> interface is for lists where elements can be accessed from their position. This interface defines an indexer, as well as ways to insert or remove items from specific positions (<code>Insert</code> , <code>RemoveAt</code> methods). <code>IList<T></code> derives from <code>ICollection<T></code> .
<code>ISet<T></code>	This interface is implemented by sets. Sets allow combining different sets into a union, getting the intersection of two sets, and checking whether two sets overlap. <code>ISet<T></code> derives from <code>ICollection<T></code> .
<code>IDictionary<TKey, TValue></code>	The interface <code>IDictionary<TKey, TValue></code> is implemented by generic collection classes that have a key and a value. With this interface all the keys and values can be accessed, items can be accessed with an indexer of type <code>key</code> , and items can be added or removed.
<code>ILookup<TKey, TValue></code>	Similar to the <code>IDictionary<TKey, TValue></code> interface, lookups have keys and values. However, with lookups the collection can contain multiple values with one key.
<code>IComparer<T></code>	The interface <code>IComparer<T></code> is implemented by a comparer and used to sort elements inside a collection with the <code>Compare</code> method.
<code>IEqualityComparer<T></code>	<code>IEqualityComparer<T></code> is implemented by a comparer that can be used for keys in a dictionary. With this interface the objects can be compared for equality.

Collections

COLLECTION	ADD	INSERT	REMOVE	ITEM	SORT	FIND
List<T>	O(1) or O(n) if the collection must be resized	O(n)	O(n)	O(1)	O (n log n), worst case O(n ^ 2)	O(n)
Stack<T>	Push, O(1), or O(n) if the stack must be resized	n/a	Pop, O(1)	n/a	n/a	n/a
Queue<T>	Enqueue, O(1), or O(n) if the queue must be resized	n/a	Dequeue, O(1)	n/a	n/a	n/a
HashSet<T>	O(1) or O(n) if the set must be resized	Add O(1) or O(n)	O(1)	n/a	n/a	n/a
SortedSet<T>	O(1) or O(n) if the set must be resized	Add O(1) or O(n)	O(1)	n/a	n/a	n/a
LinkedList<T>	AddLast O(1)	Add After O(1)	O(1)	n/a	n/a	O(n)
Dictionary <TKey, TValue>	O(1) or O(n)	n/a	O(1)	O(1)	n/a	n/a
SortedDictionary <TKey, TValue>	O(log n)	n/a	O(log n)	O(log n)	n/a	n/a
SortedList <TKey, TValue>	O(n) for unsorted data, O(log n) for end of list, O(n) if resize is needed	n/a	O(n)	O(log n) to read/ write, O(log n) if the key is in the list, O(n) if the key is not in the list	n/a	n/a

Generics

Consentono di scrivere classi e metodi **indipendenti dal tipo**

Anzichè scrivere metodi differenti per tipi differenti, è possibile scrivere **un unico metodo**

Anche per le **classi, metodi ed interfacce**

Generics

Performance

- Possiamo utilizzare **System.Collection** e **System.Collections.Generic**
- Utilizzando i tipi a valore con classi non generiche sono necessarie operazioni di **boxing** e **unboxing**. Il tipo a valore è convertito in un tipo a referenza e viceversa
- Utilizzando List<T> con il tipo int viene effettuata la **traduzione solo in fase di compilazione**. Non vengono eseguite operazioni boxing / unboxing

Generics

Type safety

Se fossero utilizzati degli object come parametri potremmo trovarci in **situazioni non sicure** in termini di esecuzione

Possiamo aggiungere stringhe, interi, ecc... **senza generare errori** in compilazione

Con i Generics il compilatore **si accorge del tipo** che stiamo inserendo

Demo

Collections Generics



Esercitazione n.2

Scrivere un'applicazione console che gestisca una squadra di calcio.

La gestione della squadra passa dalla definizione delle seguenti entità:

Atleta con: **nome, cognome, età**

Calciatore con in aggiunta: **ruolo** (ruoli = centrocampista, difensore, portiere e attaccante) e **numero maglia**

Tipi particolari di calciatori sono:

- I portieri che hanno di default il **numero maglia = 1** ed il **numero gol subiti**
- Gli attaccanti che hanno il numero gol segnati a partita
- I centrocampisti che hanno il numero di assist
- I difensori con il numero di avversari placcati

Per svolgere una partita, serve anche un arbitro che è sempre un atleta ma con in aggiunta il **numero di anni di carriera**.

Una squadra di calcio è formata da 11 calciatori di cui:

- 1 portiere
- 4 difensori
- 3 centrocampisti
- 3 attaccanti

Definire opportunamente la gerarchia delle classi. E la Collection che può contenere le entità descritte su.



Esercitazione n.3

Scrivere un'applicazione per la gestione di conti correnti bancari.

La classe **BankAccount** rappresenta un conto bancario.

In particolare il conto bancario è caratterizzato da:

- Un numero di 10 cifre che identifica in modo univoco il conto.
- Una stringa in cui viene archiviato il nome del titolare.
- Un valore double che specifica il saldo del conto corrente

Esiste inoltre la possibilità di richiedere altre due tipologie di conti correnti che aggiungono la funzionalità di TransazioneFineMese:

- **Conto Credito**, che implementa il metodo di TransazioneFineMese decurtando una tassa 2% del saldo dal conto corrente qualora il saldo del conto sia negativo;
- **Conto Risparmio** che implementa il metodo TransazioneFineMese accreditando un bonus del 2% sul conto corrente qualora il saldo sia maggiore di 500€



Esercitazione n.3

Le funzionalità legate alle varie tipologie di conto bancario devono consentire di:

- Recuperare il saldo.
- Tenere traccia delle **transazioni** eseguite sul conto. Ciascuna **transazione** sarà caratterizzata da: **data**, **importo e causale**.
- Apertura di un conto bancario. Quando un cliente apre un conto, deve specificare il saldo iniziale e le informazioni sul titolare del conto. (Il numero di conto deve essere assegnato al momento della costruzione dell'oggetto, ma la sua assegnazione non deve essere responsabilità del chiamante).
- Realizzare dei metodi che consentono di eseguire versamenti e i prelievi sul conto.
- Scrivere un metodo che consenta di mostrare a video le informazioni delle transazioni eseguite

Bisogna inoltre garantire che:

- Il saldo iniziale sia positivo.
- I prelievi non possono portare il saldo negativo.



Generics

Riuso del codice

Meno codice da **scrivere**

Meno codice da **mantenere!**

Scriviamo un metodo/classe che può essere utilizzato con **tipi differenti**

Generics

Crescita del codice

Quanto viene compilata una Generics Class il compilatore crea una classe **per ogni tipo che deve gestire**

Una classe Generics viene definita direttamente all'interno dell'assembly, istanziare una classe Generics con tipi specifici **non duplica** la classe nell'IL

Generics

Creare una classe generica

Non è possibile assegnare null
ad una classe generica

```
public class LinkedListNode<T>
{
    public LinkedListNode(T value)
    {
        Value = value;
    }

    public T value { get; private set; }
    public LinkedListNode<T> Next { get; internal set; }
    public LinkedListNode<T> Prev { get; internal set; }
}
```

Utilizzo della keyword **T**

Generics

Non è possibile associare un **valore null** ad un Generics perché:

- Un tipo generics può essere istanziato come Value Types ed i Value Types non accettano valori null
- Un tipo generics **non** è un Reference Types!

Utilizzando **default(T)**

- Viene associato **null** a **Reference Types**
- Viene associato **0** a **Values Types**

Generics

E' possibile definire **alcune regole** relativamente al tipo di Generics che deve essere utilizzato:

CONSTRAINT	DESCRIPTION
<code>where T: struct</code>	With a struct constraint, type T must be a value type.
<code>where T: class</code>	The class constraint indicates that type T must be a reference type.
<code>where T: IFoo</code>	Specifies that type T is required to implement interface IFoo.
<code>where T: Foo</code>	Specifies that type T is required to derive from base class Foo.
<code>where T: new()</code>	A constructor constraint; specifies that type T must have a default constructor.
<code>where T1: T2</code>	With constraints it is also possible to specify that type T1 derives from a generic type T2.

Generics

E' possibile **combinare** multipli constraint:

```
public class MyClass<T>  
    where T : IFoo, new()  
{  
    //...
```

E' possibile utilizzare **l'ereditarietà**.

Un tipo Generics può **implementare un'interfaccia Generics**:

```
public class LinkedList<T> : IEnumerable<T>  
{  
    //...
```


Generics

Metodi Generici

Come per le classi è possibile **definire metodi Generics**

Anche in questo caso è possibile **aggiungere Constraints**

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)  
    where TAccount : IAccount
```

Demo

Generics



Esercitazione n.4

Erogatore Merendine (si richiede l'uso del Dictionary):

Mostrare un menu all'utente per far scegliere la merendina desiderata (scelta tra 4 merendine):

Esempio:

Scegli:

1 -> per Buondì: prezzo 1 €

2 -> per Patatine: prezzo 0.5 €

ecc

Una volta scelta la merendina, chiedere all'utente di inserire il denaro necessario.

Se la quota inserita non è sufficiente richiedere nuovamente l'aggiunta di denaro e sommarla a quella già inserita. Rieffettuare il controllo fino al raggiungimento o superamento del prezzo della merendina scelta.

Se il totale inserito è uguale al prezzo della merendina allora mostrare a video "Erogazione merendina".

Se il totale supera il prezzo della merendina, mostrare a video "Erogazione merendina" ed anche il messaggio con il resto "Resto erogato : X.XX €".



Esercitazione n.4

Progettare la struttura ad oggetti per gestire un carrello di un e-commerce.

Il sito ha degli utenti iscritti. (Inserire almeno già un utente iscritto).

L'**utente** è caratterizzato da **Username, Password, Nome e Cognome**.

Per ogni utente è previsto un solo carrello contenente il riepilogo degli ordini effettuati e del prezzo totale da pagare.

(nota: è importante risalire dall'utente al carrello e non il viceversa).

Ogni **prodotto** che il sito mette in vendita è caratterizzato da un **codice**, una **descrizione**, un **prezzo**, una **percentuale di sconto**.

(Nota: creare un elenco di prodotti che il sito metterà a disposizione. Almeno 2.)

Ogni carrello può contenere diversi **ordini**; ogni ordine è costituito dalle seguenti informazioni: **prodotto**, la **quantità**, il **prezzo totale** (rispetto alla quantità e al prezzo "pieno" del prodotto), il **prezzo totale scontato** (calcolato rispetto alla percentuale di sconto del singolo prodotto).



Esercitazione n.4

All'accesso, viene chiesta username e password. Se sono corrette si procede a far visualizzare il menu, altrimenti si chiede all'utente se vuole registrarsi quindi si richiedono i dati necessari alla registrazione e si aggiunge l'utente agli utenti iscritti al sito. Completata la registrazione, l'utente resta quindi "loggato" e accede al menu.

1. Aggiungi prodotto al carrello*
2. Elimina prodotto
3. Modifica quantità di un prodotto già inserito
4. Stampa a video riepilogo del carrello (formato a piacere. Deve contenere i dati dell'utente "loggato" e il Totale da pagare in base agli ordini presenti nel carrello)
5. Esci

*Si noti che nel caso sia inserito un prodotto che già esiste nel carrello questo va a modificare la quantità del prodotto precedentemente inserito.

Requisito: all'avvio si richiede di avere almeno un utente inserito ed una serie di prodotti differenti.

Si richiede di utilizzare la Collection adeguata per la gestione dei prodotti

Generare una gerarchia opportuna per la rappresentazione dei prodotti

