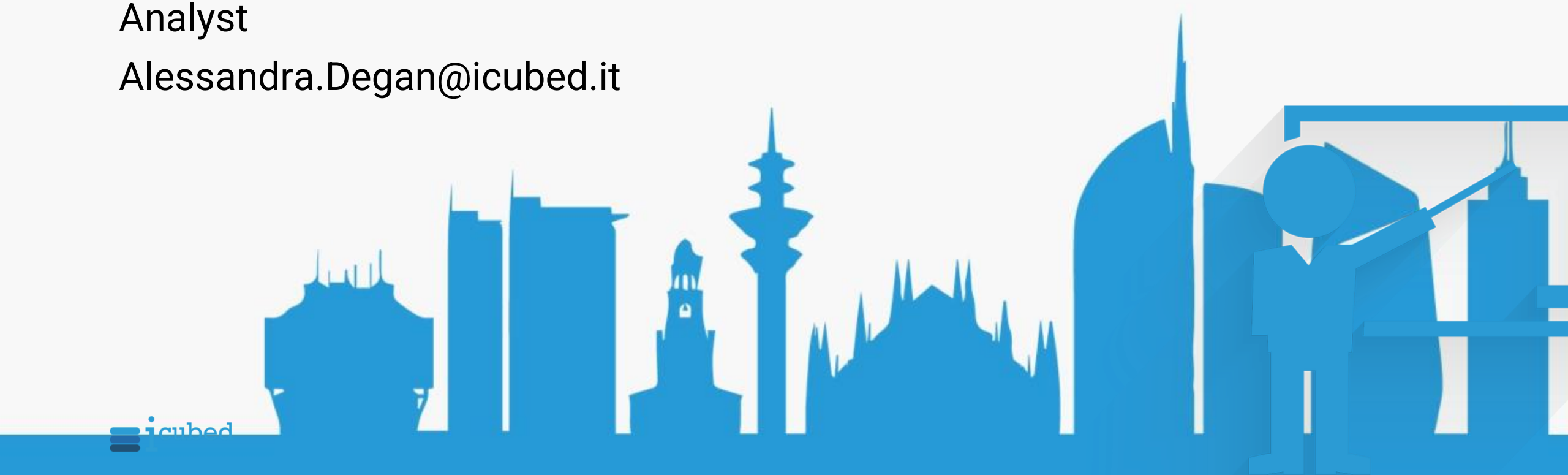
Programmazione a oggetti

Alessandra Degan Di Dieco

Analyst

Alessandra.Degan@icubed.it



Programmazione Procedurale

Organizzazione e suddivisione del codice in funzioni e procedure.

- Un’operazione è corrispondente a una routine, che accetta parametri iniziali e che produce eventualmente un risultato.

- Separazione tra logica applicativa e dati

Object Oriented Programming

OOP – Object Oriented Programming

- È un paradigma di programmazione

- Si basa sulla definizione e uso di diverse entità, collegate e interagenti, caratterizzate da un'insieme di informazioni di stato e di comportamenti
- Tali entità vengono denominate *Oggetti*

Oggetti

Gli oggetti possono contenere:

- Dati
- Funzioni
- Procedure

Funzioni e procedure possono sfruttare lo stato dell’oggetto per ricavare informazioni utili per la rispettiva elaborazione.

Classe

Gli oggetti sono istanze di una classe.

Una classe:
- È un reference type
- È composta da membri

I membri di una classe sono:
- Campi
- Proprietà
- Metodi
- Eventi

```
MyClass c = new MyClass();

public class MyClass {
    //...
}
```

Tipi, classi e oggetti

- Un **tipo** è una rappresentazione concreta di un concetto. Per esempio, il tipo built-in *float* fornisce una rappresentazione concreta di un numero reale. (*)

- Una **classe** è un tipo definito dall'utente. (*)

- Un **oggetto** è l'istanza di una classe caratterizzato da:
  - un'identità (distinto dagli altri);
  - un comportamento (compie elaborazioni tramite i metodi);
  - uno stato (memorizza dati tramite campi e proprietà).

(*) The C++ Programming Language, Third Edition. Bjarne Stroustrup

Istanze delle classi

- La creazione dell’istanza di una classe (ovvero un oggetto) può avvenire utilizzando la *keyword* ***new***

Classi e proprietà

- È il modo migliore per soddisfare uno dei pilastri della programmazione OOP: *incapsulamento*

- Una proprietà può provvedere accessibilità in lettura (**get**) scrittura (**set**) o entrambi.

- Si può usare una proprietà per ritornare valori calcolati o eseguire una validazione.

Classi e proprietà

Proprietà tradizionale

```
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

MyClass c = new MyClass();
c.Name = "C#";
```

ReadOnly / WriteOnly

```
public class MyClass
{
    private string _name = "C#";

    public string Name
    {
        get { return _name; }
    }
}

MyClass c = new MyClass();
c.Name = "C#"; // non si può fare
Console.WriteLine(c.Name); // si può fare
```

Stack e Managed Heap

Value Type

- Dichiarati all’interno di una funzione: Stack
- Parametri di una funzione : Stack
- All’interno di una classe: *Heap*

Reference Type

- Sempre: Heap

Metodi

Definisce un comportamento o un’elaborazione relative all’oggetto.

Si definisce come una routine, quindi ha una firma in cui si definiscono eventuali parametri d’ingresso e valori di ritorno.

```
int MyMethod(string str) {
    int a = int.Parse(str);
    return a;
}
```

Metodi

- Sono **funzioni associate** ad una particolare classe
- Possibilità di associare **modificatori di accesso** (public, private…)
- Definizione di un **metodo**:

```
[modifiers] return_type MethodName([parameters])
{
  // Method body
}
```

- Possibilità di effettuare **overloading sulla chiamata** del metodo

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 (che 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;
}
```

Metodi

- Particolari tipi di metodi: **Costruttori**
- Differenze tra metodi statici e non
- Possibilità di **richiamare costruttori da altri costruttori** (interni alla stessa classe)

```
public Costruttore1(string descrizione, int valore)
public Costruttore2(string descrizione)
public Costruttore3(string descrizione) : this(descrizione, 4)
```

Costruttori

Costruttore: metodo con stesso nome del tipo relativo. La firma di questo metodo include solo il nome del metodo e l'elenco dei parametri, ma NON include un tipo di ritorno.

Se non si specifica un costruttore per la classe, C# ne crea di default uno vuoto che crea un'istanza dell'oggetto e imposta le variabili ai valori di default
→ ***Costruttore senza parametri***

```
public class Person {
    private string _name;
    private int _age;

    //Costruttore CON parametri
    public Person(string name, int age) {
        _name = name;
        _age = age;
    }
}
```

Distruttori o finalizzatori

Per eseguire pulizia finale quando un'istanza di classe viene raccolta dal Garbage Collector.

- Vengono usati solo con le classi.
- Uno per classe.
- Un finalizzatore non accetta modificatori e non ha parametri.
- Il Garbage Collector decide quando chiamarlo → chiama metodo ***Finalize***

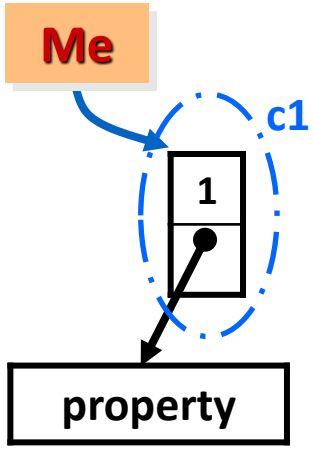
```
class Person {
          ~Person() //finalizzatore
          {
                    // espressioni di cleanup...
          }
}
```

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 property;

    public void MyMethod(int property) {
        this.property  = property;
    }
}
```



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();
```

Esercitazione 1

I prodotti trattati da un'erboristeria sono entità che hanno:

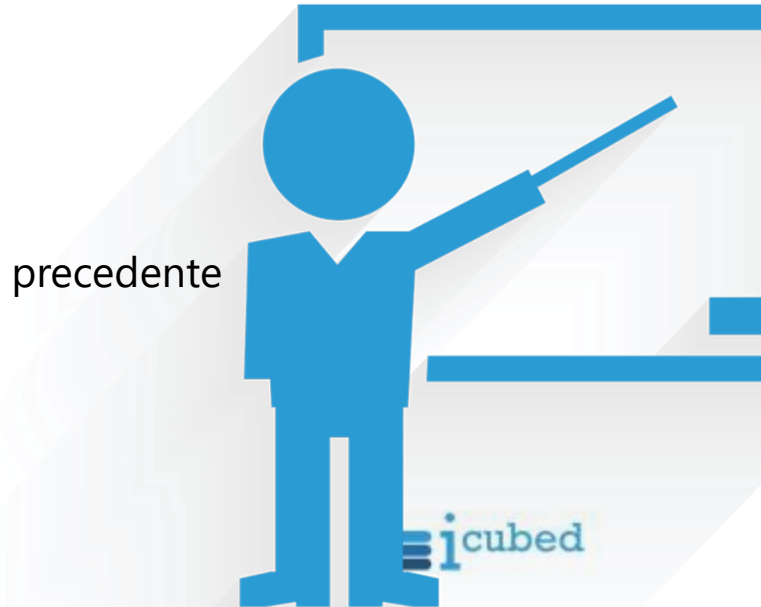
- Id
- Codice
- Nome
- Categoria (cosmetici, integratori o infusi)
- Prezzo

La classe prodotto deve avere un metodo per stampare le informazioni sul prodotto.

Inizializzare una lista con almeno 2 prodotti per categoria.

Sviluppare un programma che, all'accesso, propone:

A. stampare i dati relativi al prodotto di prezzo massimo
B. stampare i dati relativi a un determinato prodotto il cui codice è fornito in input
C. stampare i prodotti di una certa categoria
D. aggiornare il prezzo di un prodotto che ha subito un AUMENTO rispetto al prezzo precedente
E. stampa dei dati relativi ai prodotti con prezzo compreso in una certa fascia, i cui estremi sono forniti in input



Membri statici e classi statiche

- I dati relativi ad una classe sono marcati con la keyword **static** e descrivono le informazioni comuni a tutti gli oggetti dello stesso tipo.
- Ciò che è marcato **static** può essere utilizzato senza la necessità di istanziare oggetti.

```
public class MyClass {
    public static int MyStaticProperty;
    public int MyNotStaticProperty;
}

MyClass c1 = New MyClass();
MyClass c2 = New MyClass();

MyClass. MyStaticProperty  = 3;
c1. MyNotStaticProperty  = 5;
c2. MyNotStaticProperty  = 7;
```

Membri statici e classi statiche

```
public static class MyClass {
    public static int PropOne;
    public static string PropTwo;
}

//MyClass c1 = new MyClass(); //Errore!

MyClass.PropOne = 1;

MyClass. PropTwo = "Two";
```

Classe statica

- Membri devono essere static.
- Non si può istanziare la classe
- È sealed
- Accedere ai membri tramite il nome della classe.

Principi di OOP

- Astrazione
- Incapsulamento
- Ereditarietà
- Polimorfismo

Principi di OOP

- Astrazione
- Incapsulamento
- Ereditarietà
- Polimorfismo

Incapsulamento

Implementato usando i modificatori si accesso.

Una classe può specificare l'accessibilità ai propri membri da codice esterno.

MODIFIER	APPLIES TO	DESCRIPTION
`public`	Any types or members	The item is visible to any other code.
`protected`	Any member of a type, and any nested type	The item is visible only to any derived type.
`internal`	Any types or members	The item is visible only within its containing assembly.
`private`	Any member of a type, and any nested type	The item is visible only inside the type to which it belongs.
`protected internal`	Any member of a type, and any nested type	The item is visible to any code within its containing assembly and to any code inside a derived type.

Accessibilità

- Tutti i tipi e i membri di tipi (es. campi, proprietà e metodi di una classe) possono avere accessibilità diversa (*accessor modifier*):
  - **public** — Accessibili da qualsiasi altro codice
  - **protected** — Accessibile solo dal codice nella stessa classe o in una classe derivata.
  - **private** — Non accessibile dall'esterno
  - **internal** — Accessibile all'interno dell'assembly
  - **internal protected** — Combinazione delle due
- Differenziare l'accessibilità di un membro è fondamentale per realizzare l'*incapsulamento*.
- L'insieme dei membri esposti da un classe rappresenta la sua *interfaccia*.

La classe Object

Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

System.Object

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo

La classe Object

- **ToString:** converte l’oggetto in una stringa
- **GetHashCode:** ottiene il codice hash dell’oggetto
- **Equals:** permette di effettuare la comparazione tra oggetti
- **Finalize:** chiamato in fase di cancellazione da parte del garbage collector
- **GetType:** ottiene il tipo dell’oggetto
- **MemberwiseClone:** effettua la copia dell’oggetto e ritorna una reference alla copia

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.Generics**
- Differenti specializzazioni per **differenti utilizzi**

System.Collection

Contiene le seguenti Collections:

- ArrayList
- Stack
- Queue
- HashTable

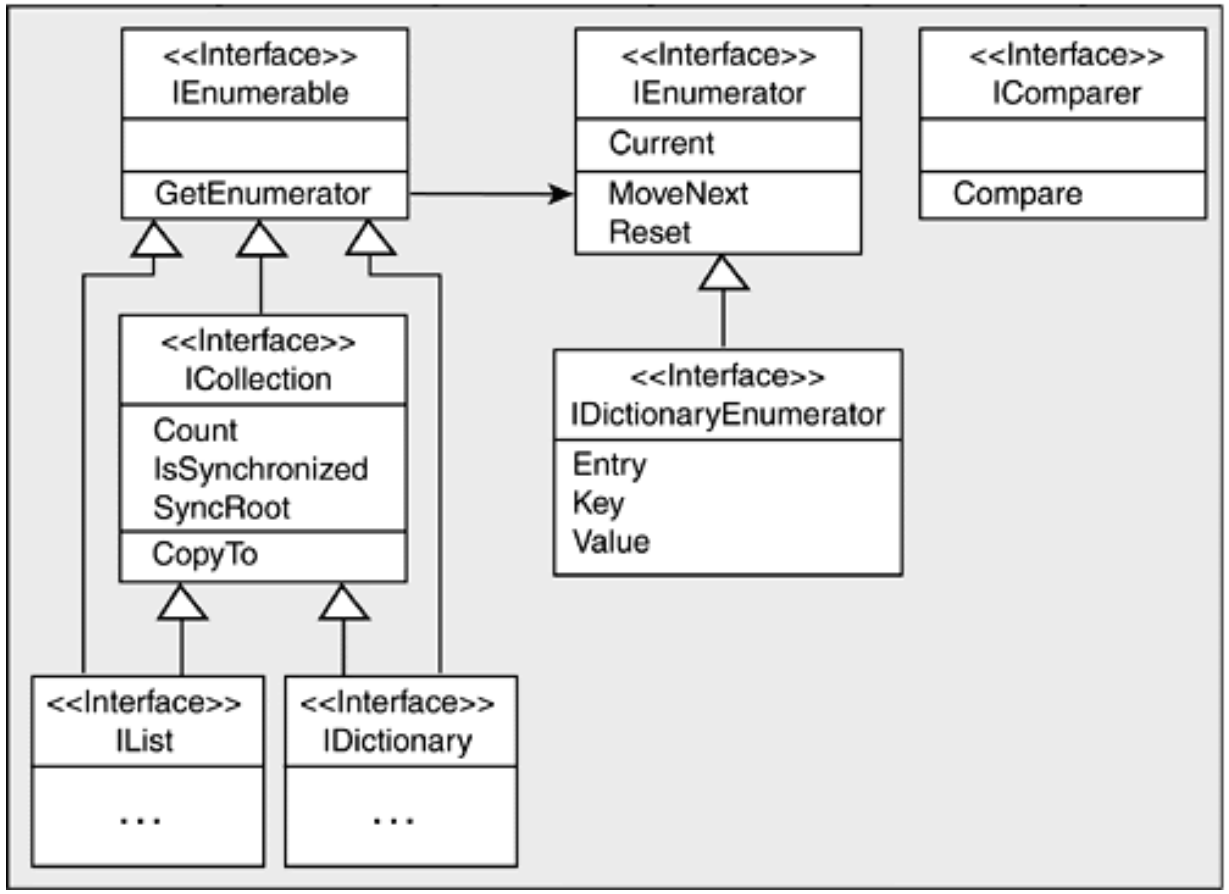
Supportano solo tipizzazione debole

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.Collection



System.Collections.Generic

Contiene le seguenti Collections:

- List<T>
- Dictionary<Tkey, Tvalue>
- HashSet<T>
- Stack<T>
- Queue<T>
- …

Supportano Tipizzazione forte

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

Collections

INTERFACE	DESCRIPTION
`IEnumerable<T>`	The interface `IEnumerable` is required by the `foreach` statement. This interface defines the method `GetEnumerator`, which returns an enumerator that implements the `IEnumerator` interface.
`ICollection<T>`	`ICollection<T>` is implemented by generic collection classes. With this you can get the number of items in the collection (`Count` property), and copy the collection to an array (`CopyTo` method). You can also add and remove items from the collection (`Add`, `Remove`, `Clear`).
`IList<T>`	The `IList<T>` 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 (`Insert`, `RemoveAt` methods). `IList<T>` derives from `ICollection<T>`.
`ISet<T>`	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. `ISet<T>` derives from `ICollection<T>`.
`IDictionary<TKey, TValue>`	The interface `IDictionary<TKey, TValue>` 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 `key`, and items can be added or removed.
`ILookup<TKey, TValue>`	Similar to the `IDictionary<TKey, TValue>` interface, lookups have keys and values. However, with lookups the collection can contain multiple values with one key.
`IComparer<T>`	The interface `IComparer<T>` is implemented by a comparer and used to sort elements inside a collection with the `Compare` method.
`IEqualityComparer<T>`	`IEqualityComparer<T>` 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

icubed

Esercitazione 2

Distributore di snack:
Lo snack ha:
- Nome
- Prezzo
- Id

Creare un metodo per riempire il distributore.

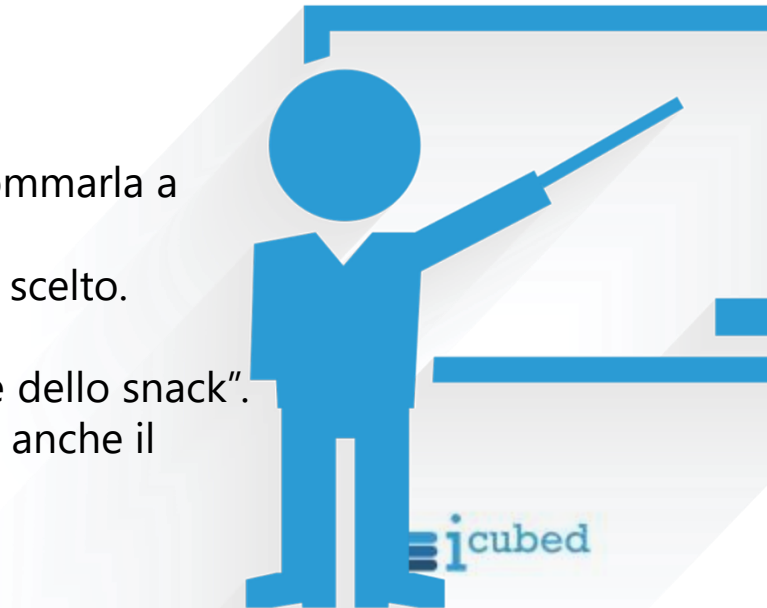
Usare il dictionary come collection → int, Snack.

Mostrare un menu all'utente per far scegliere lo snack desiderato

Una volta scelto lo snack, 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 dello snack scelto.

Se il totale inserito è uguale al prezzo dello snack allora mostrare a video "Erogazione dello snack".
Se il totale supera il prezzo dello snack, mostrare a video "Erogazione dello snack" ed anche il messaggio con il resto "Resto erogato : X.XX €"



© 2019 iCubed Srl

iCubed s.r.l. • Piazza Durante, 8 – 20131, Milano
• Phone: +39 02 57501057 • P.IVA 07284390965