In C#, a **Dictionary** is a collection type that represents a **generic, unordered collection of key-value pairs**.

Each **key must be unique** within the Dictionary, and it **provides fast access** to values based on their associated keys.

The Dictionary class is part of the System.Collections.Generic namespace.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creating a Dictionary with string keys and int values
        Dictionary<string, int> myDictionary = new Dictionary<string,
int>();

        // Adding key-value pairs to the dictionary
        myDictionary["Alice"] = 25;
        myDictionary["Bob"] = 30;
        myDictionary["Charlie"] = 22;

        // Accessing values by key
        Console.WriteLine($"Age of Alice: {myDictionary["Alice"]}");

        // Checking if a key exists
        if (myDictionary.ContainsKey("Bob"))
        {
            Console.WriteLine($"Bob's age: {myDictionary["Bob"]}");
        }

        // Iterating over key-value pairs
        foreach (var kvp in myDictionary)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value} years old");
        }
    }
}
```

The **ContainsKey** method is used to check if a key exists in the dictionary, and a foreach loop is used to iterate over all key-value pairs.

Dictionary in C# is **a mutable data structure**, which means you can modify it by adding, removing, or updating key-value pairs after it has been created.

A C# dictionary is a generic collection of key-value pairs. The keys must be unique, but the values can be duplicated. Dictionaries are implemented as hash tables so their keys can quickly access them.

Creating Dictionary:

```
Dictionary<string, int> dictionary = new
Dictionary<string, int>();
```

```
dictionary.Add("one", 1);
dictionary.Add("two", 2);
```

```
int value = dictionary.Get("one");
```

```
must be imported
using System.Collections.Generic;
```

Retrieve Elements from a C# Dictionary

```
public void CreateDictionary()
{
    // Create a dictionary with string key and Int16 value pair
    Dictionary<string, Int16> AuthorList = new Dictionary<string,
Int16>();
    AuthorList.Add("Mahesh Chand", 35);
    AuthorList.Add("Mike Gold", 25);
    AuthorList.Add("Praveen Kumar", 29);
    AuthorList.Add("Raj Beniwal", 21);
    AuthorList.Add("Dinesh Beniwal", 84);
    // Read all data
    Console.WriteLine("Authors List");

    foreach (KeyValuePair<string, Int16> author in AuthorList)
    {
        Console.WriteLine("Key: {0}, Value: {1}", author.Key,
author.Value);
    }
}
```

```
Dictionary<string, Int16>.KeyCollection keys = AuthorList.Keys;
Dictionary<string, Int16>.ValueCollection values = AuthorList.Values;
```

```
Methods:Add(),Remove(),clear()
```

KeyValuePair<TKey, TValue>:

It is a **simple structure representing a key-value pair**.

It is **often used** when you need to work with key-value pairs directly, for example, **when iterating over the entries of a dictionary.**

It is immutable, meaning its values cannot be changed after creation.

KeyValuePair<string, int> pair = new KeyValuePair<string, int>("Alice", 25);

Dictionary<TKey, TValue>:

It is a collection type that stores key-value pairs.

It provides a mapping between keys and values, allowing you to efficiently retrieve a value based on its associated key.

It is mutable, meaning you can add, remove, or modify key-value pairs.

It is part of the System.Collections.Generic namespace.

```
Dictionary<string, int> myDictionary = new Dictionary<string, int>
{
    { "Alice", 25 },
    { "Bob", 30 },
    { "Charlie", 22 }
};
```

TryGetValue

```
var cities = new Dictionary<string, string>(){
    {"UK", "London, Manchester, Birmingham"},
    {"USA", "Chicago, New York, Washington"},
    {"India", "Mumbai, New Delhi, Pune"}
};

Console.WriteLine(cities["UK"]); //prints value of UK key
Console.WriteLine(cities["USA"]);//prints value of USA key
//Console.WriteLine(cities["France"]); // run-time exception: Key does
not exist

//use ContainsKey() to check for an unknown key
if(cities.ContainsKey("France")){
    Console.WriteLine(cities["France"]);
}

//use TryGetValue() to get a value of unknown key
string result;

if(cities.TryGetValue("France", out result))
{
    Console.WriteLine(result);
}

//use ElementAt() to retrieve key-value pair using index
for (int i = 0; i < cities.Count; i++)
{
    Console.WriteLine("Key: {0}, Value: {1}",
                                            cities.ElementAt(i).Key,
                                            cities.ElementAt(i).Value);
}
```

Dictionary<TKey, TValue>[key]:

This syntax allows you to directly access the value associated with a key in the dictionary.

If the key exists, it returns the associated value; otherwise, it throws a KeyNotFoundException.

```
// Assuming 'cities' is a Dictionary<string, string>
string result = cities["UK"];
```

TryGetValue method:

This method is safer when you are unsure whether a key exists in the dictionary.

It attempts to retrieve the value associated with the specified key.

If the key is found, it returns true, and the associated value is stored in an out parameter.

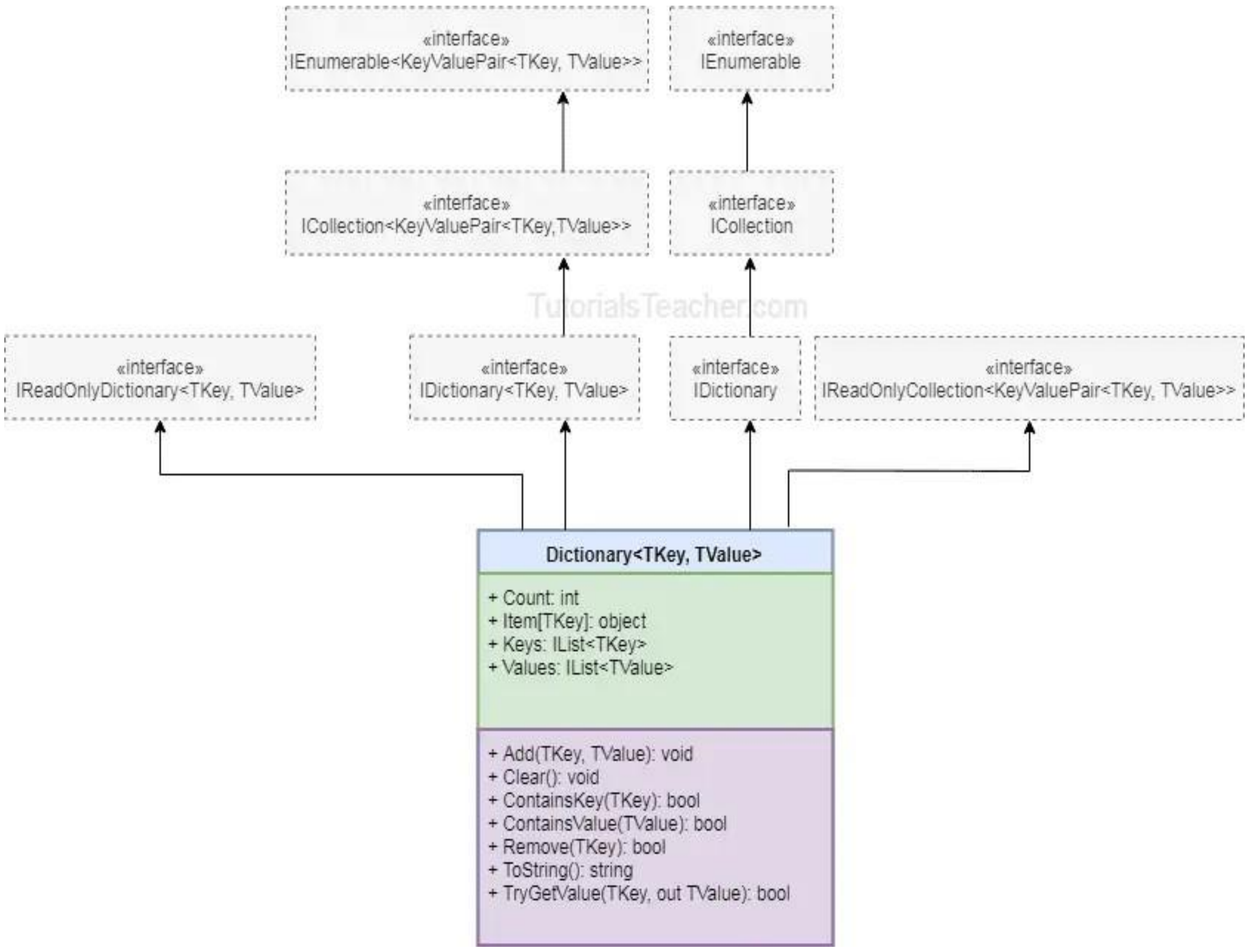
If the key is not found, it returns false, and the out parameter contains the default value for the value type.

```
// Assuming 'cities' is a Dictionary<string, string>
string result;

if (cities.TryGetValue("UK", out result))
{
    // The key "UK" exists, and the associated value is in 'result'
    Console.WriteLine(result);
}
else
{
    // The key "UK" does not exist in the dictionary
    Console.WriteLine("Key not found");
}
```

In summary, Dictionary<TKey, TValue>[key] is a direct access method that assumes the key is present and throws an exception if it's not, while TryGetValue is a safer method that checks for the existence of the key before attempting to retrieve the value. Use TryGetValue when you are uncertain about the existence of the key or want to avoid exceptions.

Dictionary Class Hierarchy


«interface»
IEnumerable<KeyValuePair<TKey, TValue>>
«interface»
IEnumerable
«interface»
ICollection<KeyValuePair<TKey,TValue>>
«interface»
ICollection
TutorialsTeacher.com
«interface»
IReadOnlyDictionary<TKey, TValue>
«interface»
IDictionary<TKey, TValue>
«interface»
IDictionary
«interface»
IReadOnlyCollection<KeyValuePair<TKey, TValue>>
Dictionary<TKey, TValue>
+ Count: int
+ Item[TKey]: object
+ Keys: IList<TKey>
+ Values: IList<TValue>
+ Add(TKey, TValue): void
+ Clear(): void
+ ContainsKey(TKey): bool
+ ContainsValue(TValue): bool
+ Remove(TKey): bool
+ ToString(): string
+ TryGetValue(TKey, out TValue): bool

n C#, the Dictionary<TKey, TValue> class is part of the .NET collections hierarchy. The collections framework in .NET provides a set of interfaces and classes for working with different types of collections. Here's a brief explanation of the hierarchy related to dictionaries:

Extra:

IEnumerable<T> Interface:

The most basic interface representing a collection that can be enumerated.

IEnumerable<T> provides the GetEnumerator method for iterating over the elements of a collection.

ICollection<T> Interface:

Extends IEnumerable<T> and provides additional methods for manipulating collections, such as adding, removing, and checking for the existence of elements.

IDictionary<TKey, TValue> Interface:

Represents a generic collection of key-value pairs.

Inherits from ICollection<KeyValuePair<TKey, TValue>> and extends it with methods specific to dictionaries.

Dictionary<TKey, TValue> directly implements IDictionary<TKey, TValue>.

IReadOnlyCollection<T> Interface:

Represents a read-only view of a collection.

IReadOnlyCollection<T> extends IEnumerable<T> and is implemented by Dictionary<TKey, TValue>.

IReadOnlyDictionary<TKey, TValue> Interface:

Represents a read-only view of a generic dictionary.

Extends IReadOnlyCollection<KeyValuePair<TKey, TValue>> and IEnumerable<KeyValuePair<TKey, TValue>>.

IReadOnlyDictionary<TKey, TValue> is implemented by Dictionary<TKey, TValue> and other read-only dictionary implementations.

Dictionary<TKey, TValue> Class:

The concrete implementation of a generic dictionary.

Implements IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IReadOnlyDictionary<TKey, TValue>, and other related interfaces.

Here's a simplified view of the hierarchy:

```
IEnumerable<T>
   |
ICollection<T>
   |
IDictionary<TKey, TValue>
   |
IReadOnlyCollection<T>
   |
IReadOnlyDictionary<TKey, TValue>
   |
Dictionary<TKey, TValue>
```