Rami Wail Shoula: Collections

I- Collections Selected

A. **HashSet<T>**

B. **List<T>**

C. **SortedDictionary<TKey, TValue>**

All technical information was gathered from Microsoft documentation at

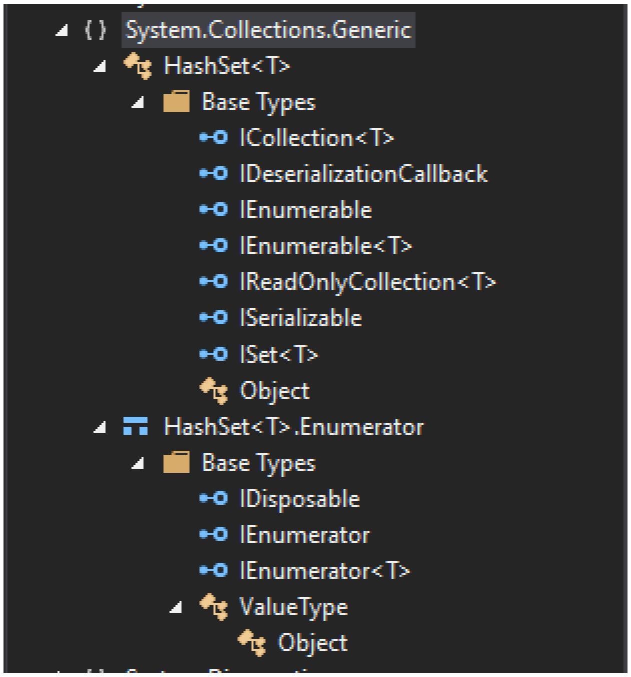
Ref: https://docs.microsoft.com/en-us/dotnet/api/system.collections?view=net-6.0

II- Visual Studio (Practical Data)

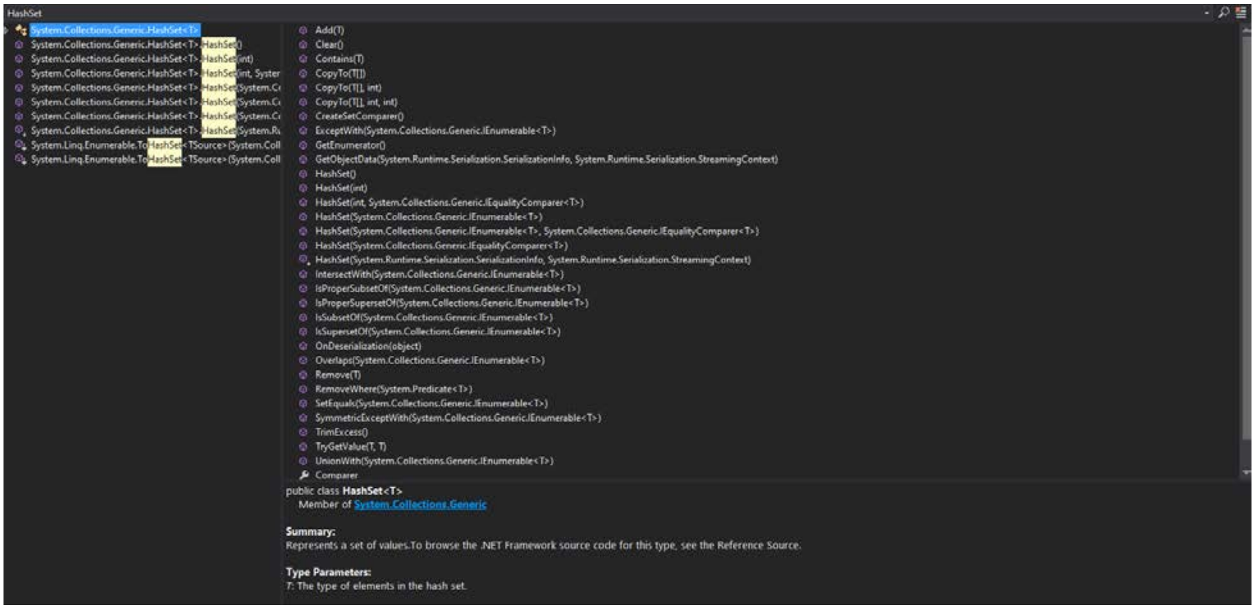
As instructed in the Lec. To get any collection data from within Visual Studio the procedure is:

View -> Object Browser -> Search //or shortcut Ctrrl + Alt + J

A. HashSet<T>



```
{ } System.Collections.Generic
    HashSet<T>
        Base Types
            ICollection<T>
            IDeserializationCallback
            IEnumerable
            IEnumerable<T>
            IReadOnlyCollection<T>
            ISerializable
            ISet<T>
            Object
    HashSet<T>.Enumerator
        Base Types
            IDisposable
            IEnumerator
            IEnumerator<T>
            ValueType
                Object
```



```
HashSet
System.Collections.Generic.HashSet<T>
System.Collections.Generic.HashSet<T>.HashSet()
System.Collections.Generic.HashSet<T>.HashSet(int)
System.Collections.Generic.HashSet<T>.HashSet(int, Syster
System.Collections.Generic.HashSet<T>.HashSet(System.C
System.Collections.Generic.HashSet<T>.HashSet(System.C
System.Collections.Generic.HashSet<T>.HashSet(System.C
System.Collections.Generic.HashSet<T>.HashSet(System.R
System.Linq.Enumerable.ToHashSet<TSource>(System.Coll
System.Linq.Enumerable.ToHashSet<TSource>(System.Coll
```

```
Add(T)
Clear()
Contains(T)
CopyTo(T[])
CopyTo(T[], int)
CopyTo(T[], int, int)
CreateSetComparer()
ExceptWith(System.Collections.Generic.IEnumerable<T>)
GetEnumerator()
GetObjectData(System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext)
HashSet()
HashSet(int)
HashSet(int, System.Collections.Generic.IEqualityComparer<T>)
HashSet(System.Collections.Generic.IEnumerable<T>)
HashSet(System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IEqualityComparer<T>)
HashSet(System.Collections.Generic.IEqualityComparer<T>)
HashSet(System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext)
IntersectWith(System.Collections.Generic.IEnumerable<T>)
IsProperSubsetOf(System.Collections.Generic.IEnumerable<T>)
IsProperSupersetOf(System.Collections.Generic.IEnumerable<T>)
IsSubsetOf(System.Collections.Generic.IEnumerable<T>)
IsSupersetOf(System.Collections.Generic.IEnumerable<T>)
OnDeserialization(object)
Overlaps(System.Collections.Generic.IEnumerable<T>)
Remove(T)
RemoveWhere(System.Predicate<T>)
SetEquals(System.Collections.Generic.IEnumerable<T>)
SymmetricExceptWith(System.Collections.Generic.IEnumerable<T>)
TrimExcess()
TryGetValue(T, T)
UnionWith(System.Collections.Generic.IEnumerable<T>)
Comparer
```

public class **HashSet<T>**
Member of System.Collections.Generic

Summary:
Represents a set of values.To browse the .NET Framework source code for this type, see the Reference Source.

Type Parameters:
T: The type of elements in the hash set.

public class HashSet<T>

Member of System.Collections.Generic

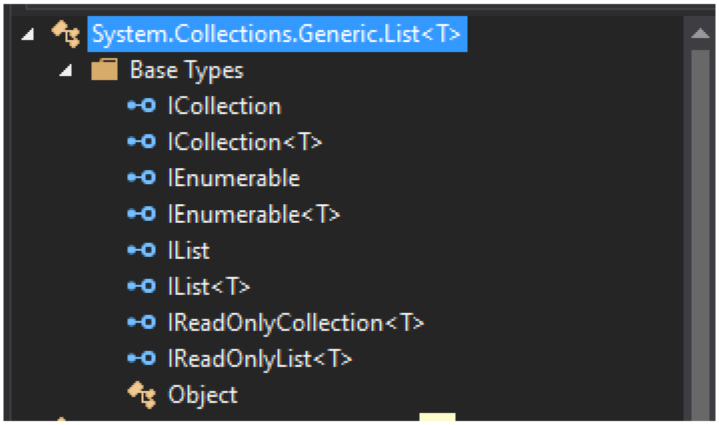
Summary:

Represents a set of values.

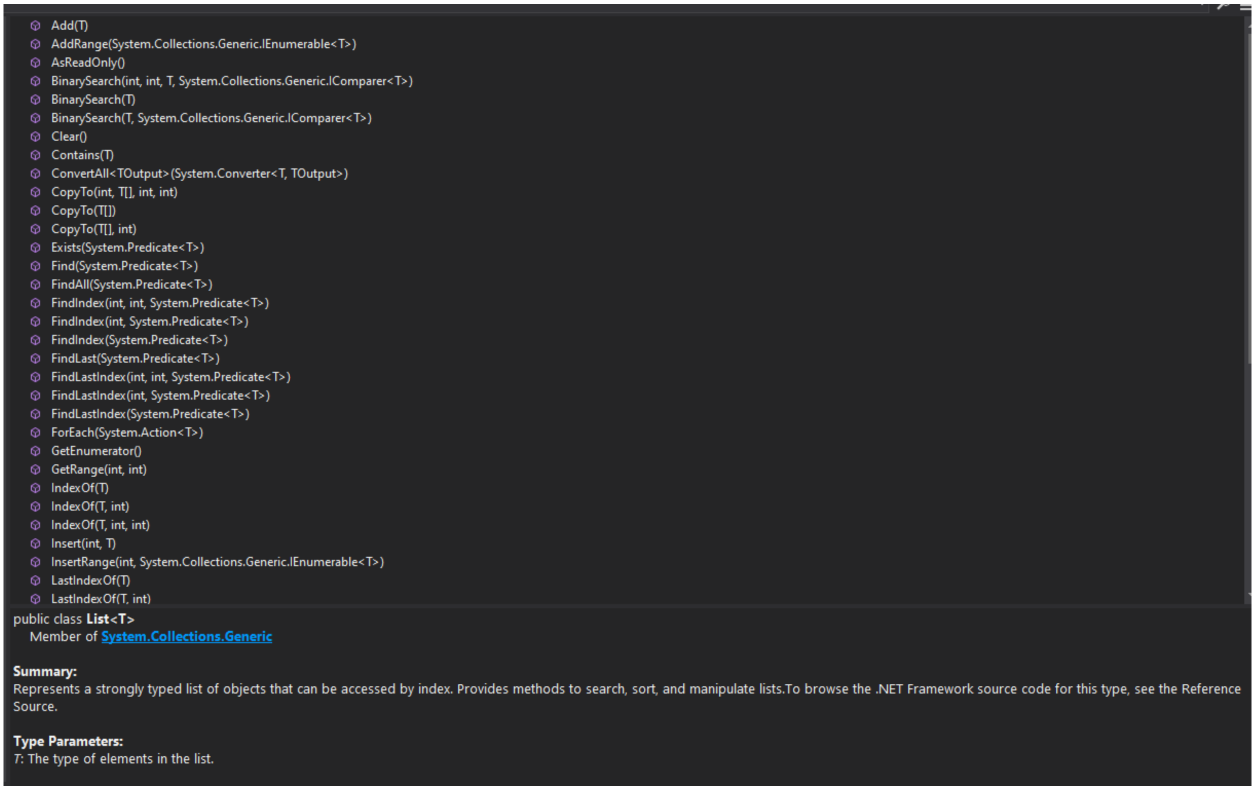
Type Parameters:

T: The type of elements in the hash set.

B. List<T>



```
System.Collections.Generic.List<T>
    Base Types
        ICollection
        ICollection<T>
        IEnumerable
        IEnumerable<T>
        IList
        IList<T>
        IReadOnlyCollection<T>
        IReadOnlyList<T>
        Object
```



```
Add(T)
AddRange(System.Collections.Generic.IEnumerable<T>)
AsReadOnly()
BinarySearch(int, int, T, System.Collections.Generic.IComparer<T>)
BinarySearch(T)
BinarySearch(T, System.Collections.Generic.IComparer<T>)
Clear()
Contains(T)
ConvertAll<TOutput>(System.Converter<T, TOutput>)
CopyTo(int, T[], int, int)
CopyTo(T[])
CopyTo(T[], int)
Exists(System.Predicate<T>)
Find(System.Predicate<T>)
FindAll(System.Predicate<T>)
FindIndex(int, int, System.Predicate<T>)
FindIndex(int, System.Predicate<T>)
FindIndex(System.Predicate<T>)
FindLast(System.Predicate<T>)
FindLastIndex(int, int, System.Predicate<T>)
FindLastIndex(int, System.Predicate<T>)
FindLastIndex(System.Predicate<T>)
ForEach(System.Action<T>)
GetEnumerator()
GetRange(int, int)
IndexOf(T)
IndexOf(T, int)
IndexOf(T, int, int)
Insert(int, T)
InsertRange(int, System.Collections.Generic.IEnumerable<T>)
LastIndexOf(T)
LastIndexOf(T, int)
```

public class **List<T>**
  Member of System.Collections.Generic

Summary:
Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.To browse the .NET Framework source code for this type, see the Reference Source.

Type Parameters:
T: The type of elements in the list.

public class List<T>

Member of System.Collections.Generic

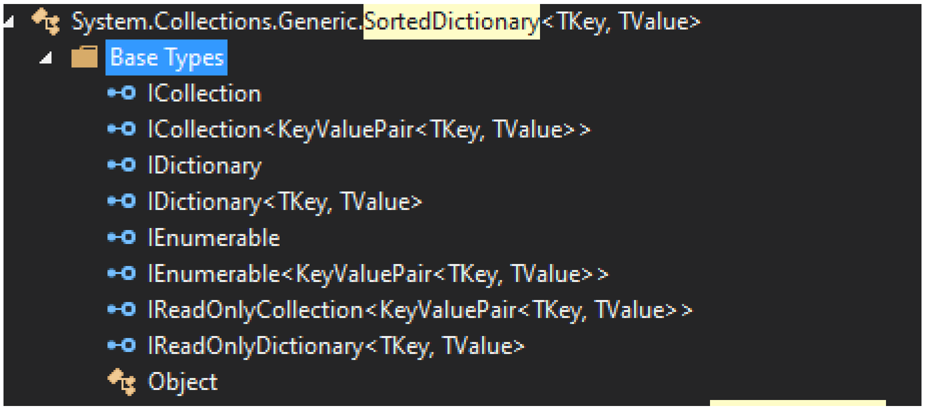
Summary:

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

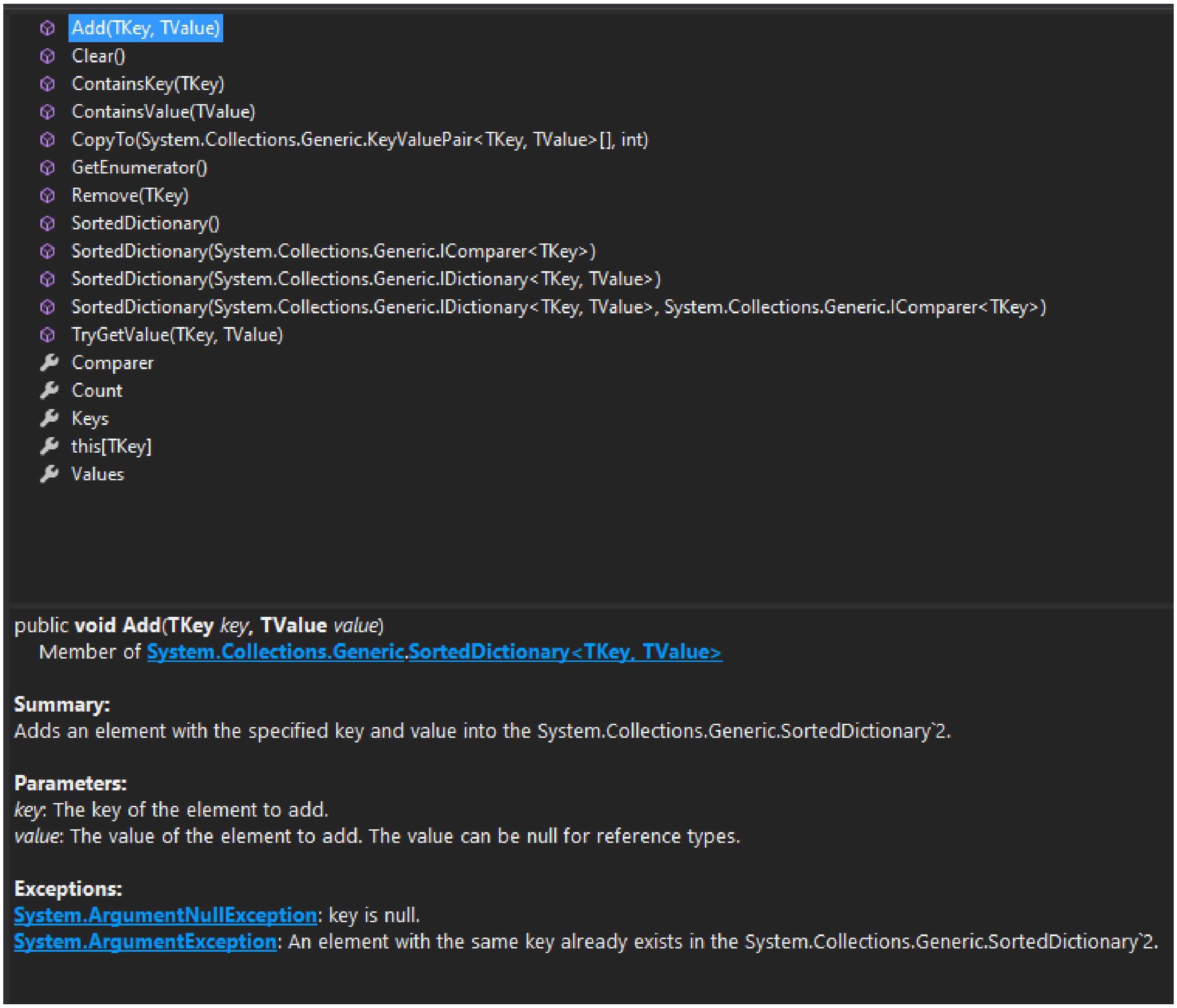
Type Parameters:

T: The type of elements in the list.

C. SortedDictionary<TKey, TValue>



```
System.Collections.Generic.SortedDictionary<TKey, TValue>
  Base Types
    ICollection
    ICollection<KeyValuePair<TKey, TValue>>
    IDictionary
    IDictionary<TKey, TValue>
    IEnumerable
    IEnumerable<KeyValuePair<TKey, TValue>>
    IReadOnlyCollection<KeyValuePair<TKey, TValue>>
    IReadOnlyDictionary<TKey, TValue>
    Object
```



Add(TKey, TValue)
Clear()
ContainsKey(TKey)
ContainsValue(TValue)
CopyTo(System.Collections.Generic.KeyValuePair<TKey, TValue>[], int)
GetEnumerator()
Remove(TKey)
SortedDictionary()
SortedDictionary(System.Collections.Generic.IComparer<TKey>)
SortedDictionary(System.Collections.Generic.IDictionary<TKey, TValue>)
SortedDictionary(System.Collections.Generic.IDictionary<TKey, TValue>, System.Collections.Generic.IComparer<TKey>)
TryGetValue(TKey, TValue)
Comparer
Count
Keys
this[TKey]
Values

public **void Add**(**TKey** *key*, **TValue** *value*)
Member of System.Collections.Generic.SortedDictionary<TKey, TValue>

Summary:
Adds an element with the specified key and value into the System.Collections.Generic.SortedDictionary`2.

Parameters:
key: The key of the element to add.
value: The value of the element to add. The value can be null for reference types.

Exceptions:
System.ArgumentNullException: key is null.
System.ArgumentException: An element with the same key already exists in the System.Collections.Generic.SortedDictionary`2.

public class SortedDictionary<TKey, TValue>

Member of System.Collections.Generic

Summary:

Represents a collection of key/value pairs that are sorted on the key.

Type Parameters:

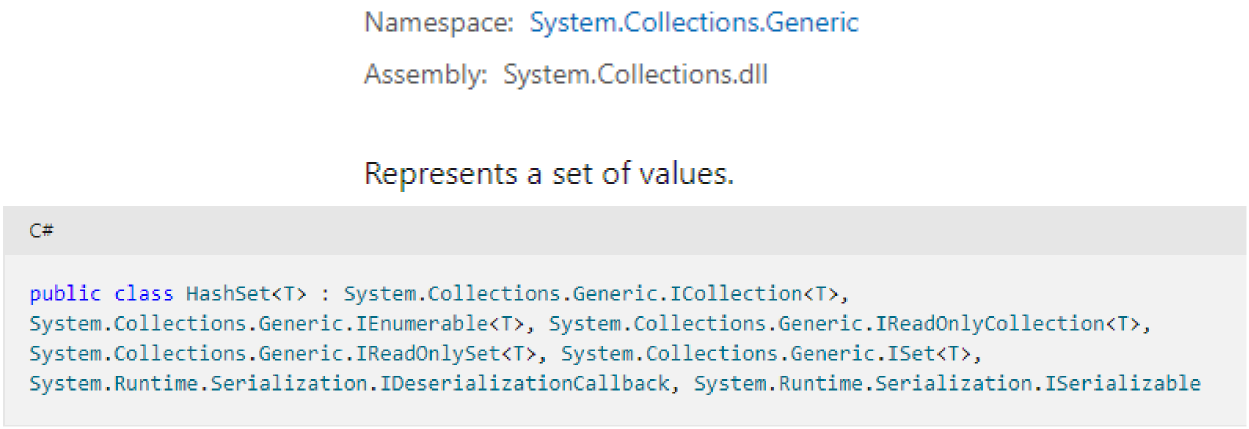
TKey: The type of the keys in the dictionary.

TValue: The type of the values in the dictionary.

III- Technical Information: HashSet<T>

Ref: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-6.0

A. Definition



Namespace: System.Collections.Generic

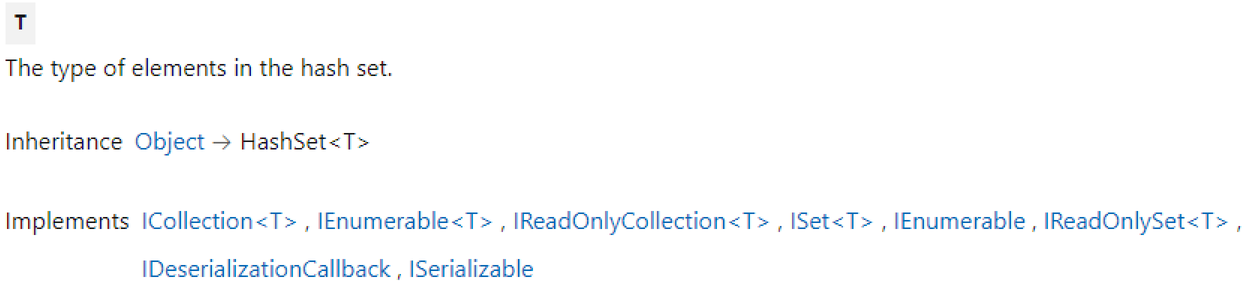
Assembly: System.Collections.dll

Represents a set of values.

C#

```
public class HashSet<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.Generic.IReadOnlySet<T>, System.Collections.Generic.ISet<T>,
System.Runtime.Serialization.IDeserializationCallback, System.Runtime.Serialization.ISerializable
```

B. Type Parameters



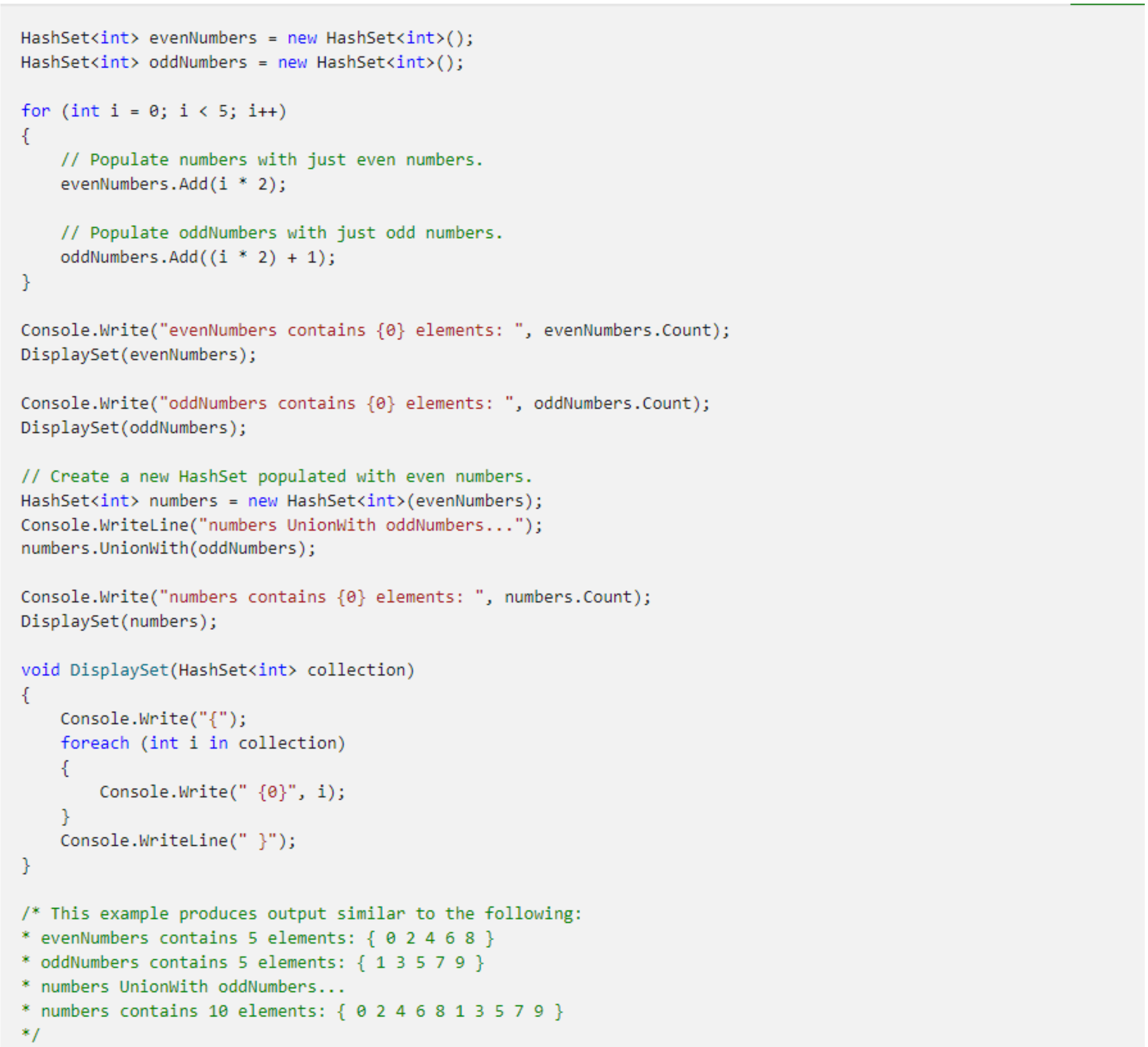
T

The type of elements in the hash set.

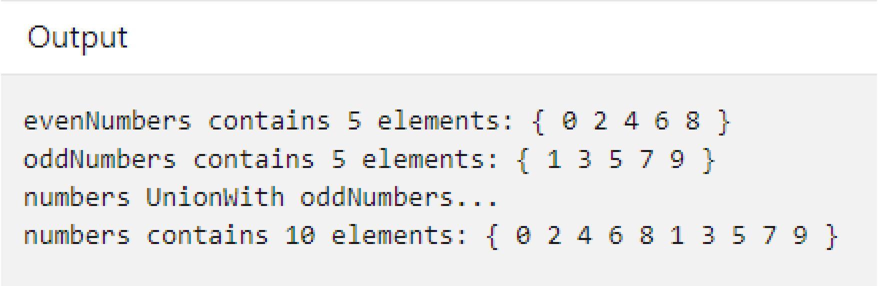
Inheritance Object → HashSet<T>

Implements ICollection<T> , IEnumerable<T> , IReadOnlyCollection<T> , ISet<T> , IEnumerable , IReadOnlySet<T> , IDeserializationCallback , ISerializable

C. Examples



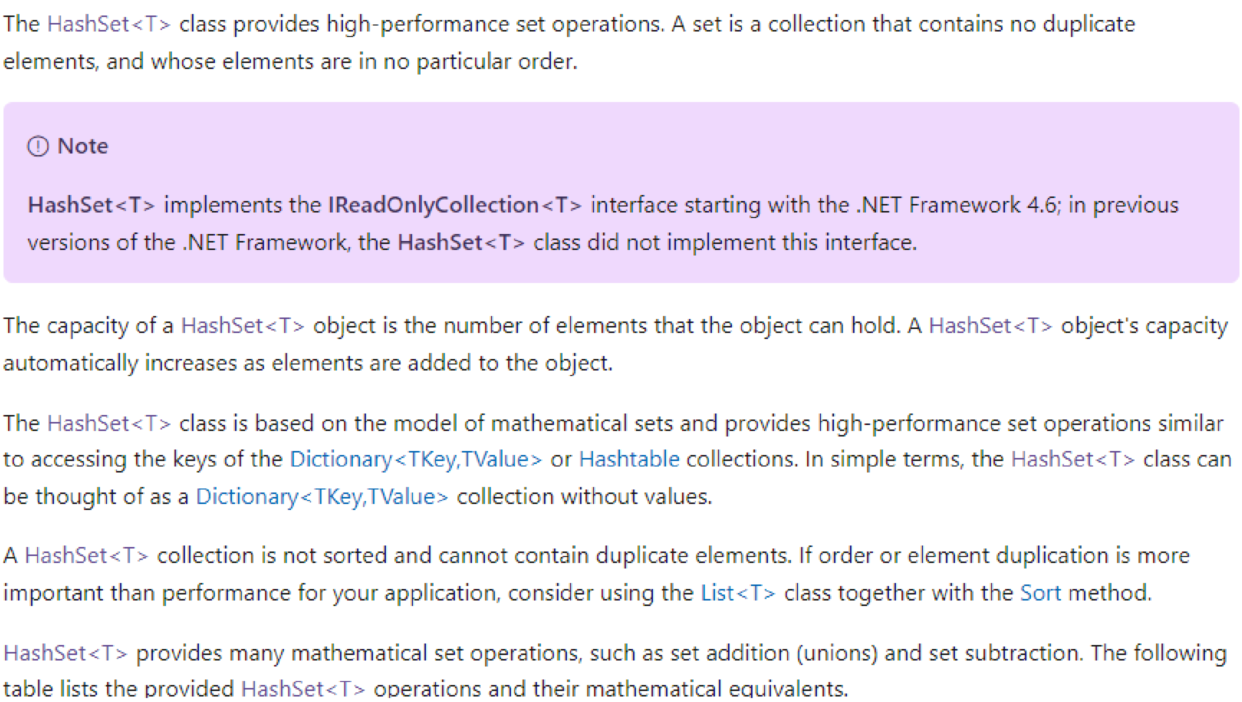
```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}

Console.Write("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);

Console.Write("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);

// Create a new HashSet populated with even numbers.
HashSet<int> numbers = new HashSet<int>(evenNumbers);
Console.WriteLine("numbers UnionWith oddNumbers...");
numbers.UnionWith(oddNumbers);

Console.Write("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

void DisplaySet(HashSet<int> collection)
{
    Console.Write("{");
    foreach (int i in collection)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example produces output similar to the following:
* evenNumbers contains 5 elements: { 0 2 4 6 8 }
* oddNumbers contains 5 elements: { 1 3 5 7 9 }
* numbers UnionWith oddNumbers...
* numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
*/
```



Output

```
evenNumbers contains 5 elements: { 0 2 4 6 8 }
oddNumbers contains 5 elements: { 1 3 5 7 9 }
numbers UnionWith oddNumbers...
numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
```

D. Remarks



The HashSet<T> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.

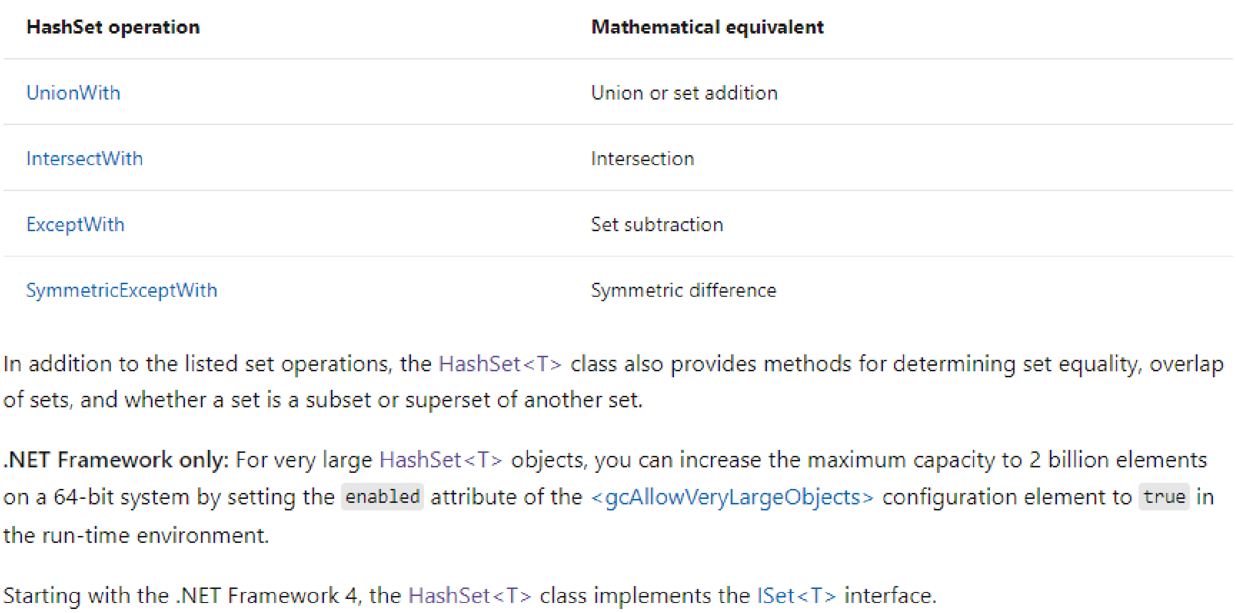
> ⓘ Note
>
> **HashSet<T>** implements the **IReadOnlyCollection<T>** interface starting with the .NET Framework 4.6; in previous versions of the .NET Framework, the **HashSet<T>** class did not implement this interface.

The capacity of a HashSet<T> object is the number of elements that the object can hold. A HashSet<T> object's capacity automatically increases as elements are added to the object.

The HashSet<T> class is based on the model of mathematical sets and provides high-performance set operations similar to accessing the keys of the Dictionary<TKey,TValue> or Hashtable collections. In simple terms, the HashSet<T> class can be thought of as a Dictionary<TKey,TValue> collection without values.

A HashSet<T> collection is not sorted and cannot contain duplicate elements. If order or element duplication is more important than performance for your application, consider using the List<T> class together with the Sort method.

HashSet<T> provides many mathematical set operations, such as set addition (unions) and set subtraction. The following table lists the provided HashSet<T> operations and their mathematical equivalents.



HashSet operation	Mathematical equivalent
UnionWith	Union or set addition
IntersectWith	Intersection
ExceptWith	Set subtraction
SymmetricExceptWith	Symmetric difference

In addition to the listed set operations, the HashSet<T> class also provides methods for determining set equality, overlap of sets, and whether a set is a subset or superset of another set.

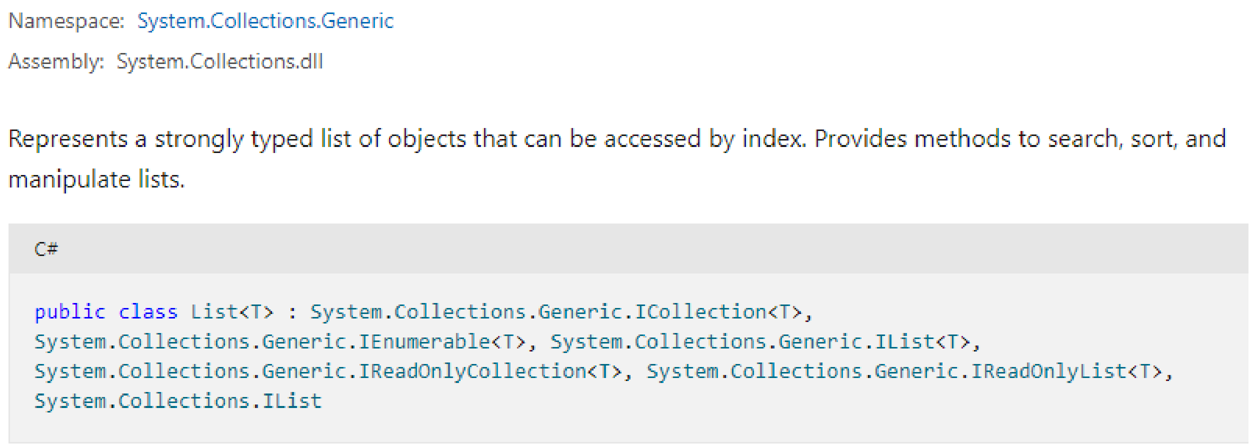
.NET Framework only: For very large HashSet<T> objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the `enabled` attribute of the <gcAllowVeryLargeObjects> configuration element to `true` in the run-time environment.

Starting with the .NET Framework 4, the HashSet<T> class implements the ISet<T> interface.

IV- Technical Information: List<T>

Ref: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0

A. Definition



Namespace: System.Collections.Generic

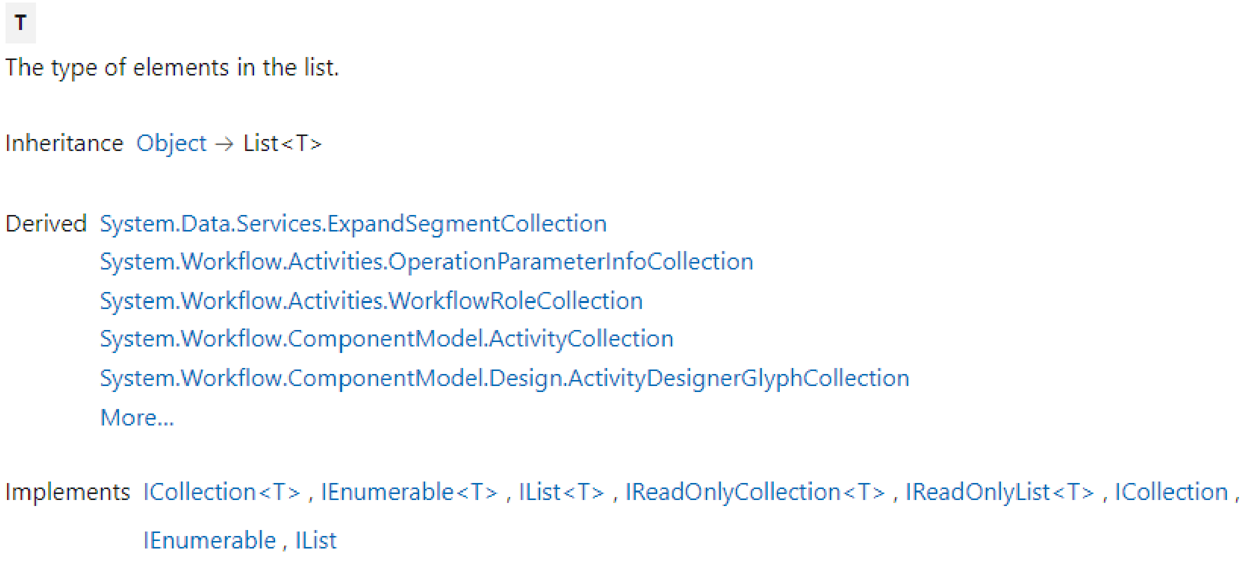
Assembly: System.Collections.dll

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

C#

```
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>,
System.Collections.IList
```

B. Type Parameters



T

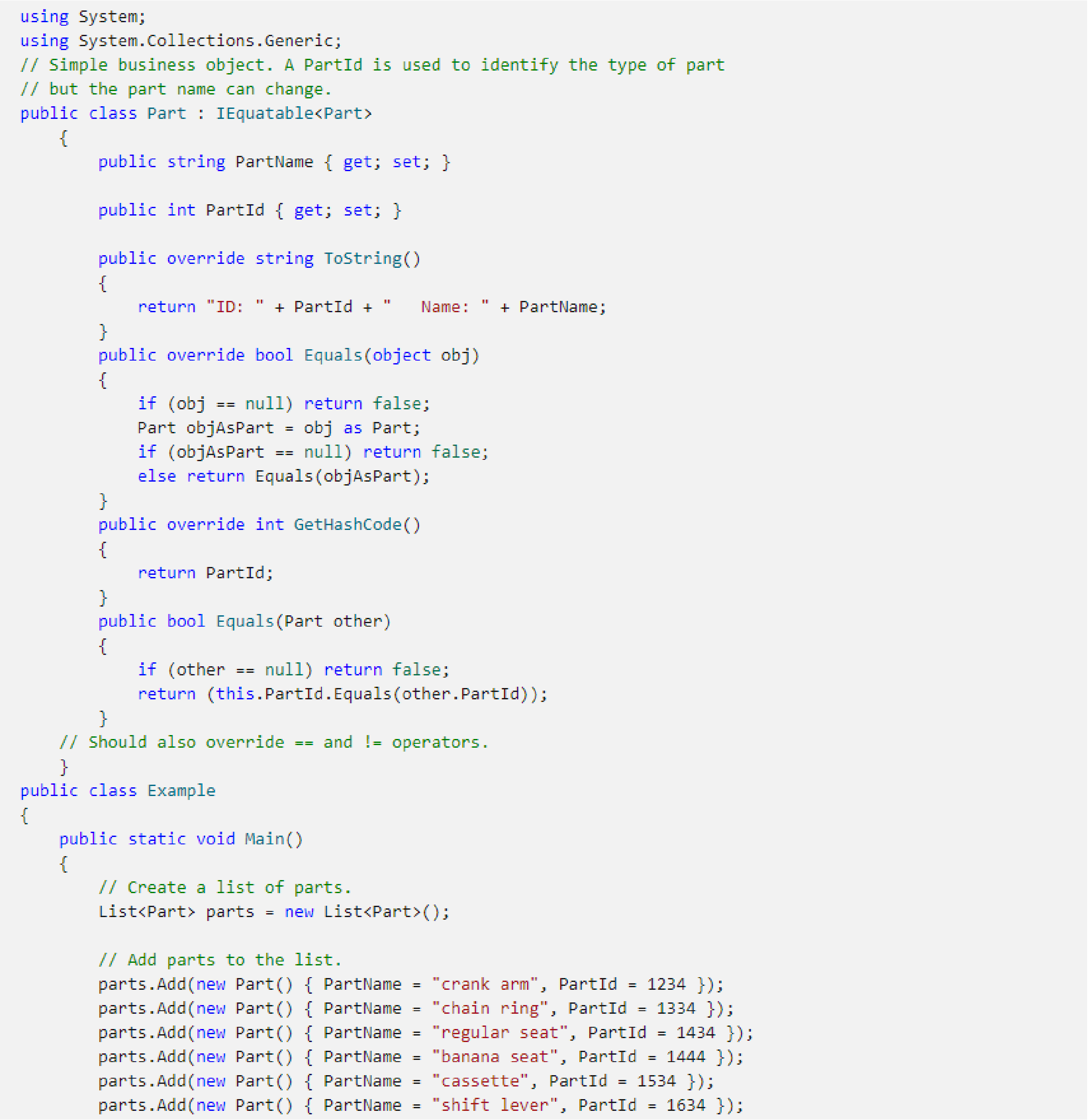
The type of elements in the list.

Inheritance Object → List<T>

Derived System.Data.Services.ExpandSegmentCollection
System.Workflow.Activities.OperationParameterInfoCollection
System.Workflow.Activities.WorkflowRoleCollection
System.Workflow.ComponentModel.ActivityCollection
System.Workflow.ComponentModel.Design.ActivityDesignerGlyphCollection
More...

Implements ICollection<T> , IEnumerable<T> , IList<T> , IReadOnlyCollection<T> , IReadOnlyList<T> , ICollection , IEnumerable , IList

C. Examples



```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>
    {
        public string PartName { get; set; }

        public int PartId { get; set; }

        public override string ToString()
        {
            return "ID: " + PartId + "   Name: " + PartName;
        }
        public override bool Equals(object obj)
        {
            if (obj == null) return false;
            Part objAsPart = obj as Part;
            if (objAsPart == null) return false;
            else return Equals(objAsPart);
        }
        public override int GetHashCode()
        {
            return PartId;
        }
        public bool Equals(Part other)
        {
            if (other == null) return false;
            return (this.PartId.Equals(other.PartId));
        }
    // Should also override == and != operators.
    }
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });
        // Write out the parts in the list. This will call the overridden ToString method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains(\"1734\"): {0}",
        parts.Contains(new Part { PartId = 1734, PartName = "" }));

        // Insert a new item at position 2.
        Console.WriteLine("\nInsert(2, \"1834\")");
        parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834 });

        //Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        Console.WriteLine("\nParts[3]: {0}", parts[3]);

        Console.WriteLine("\nRemove(\"1534\")");

        // This will remove part 1534 even though the PartName is different,
        // because the Equals method only checks PartId for equality.
        parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });

        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }
        Console.WriteLine("\nRemoveAt(3)");
        // This will remove the part at index 3.
        parts.RemoveAt(3);

        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }
```

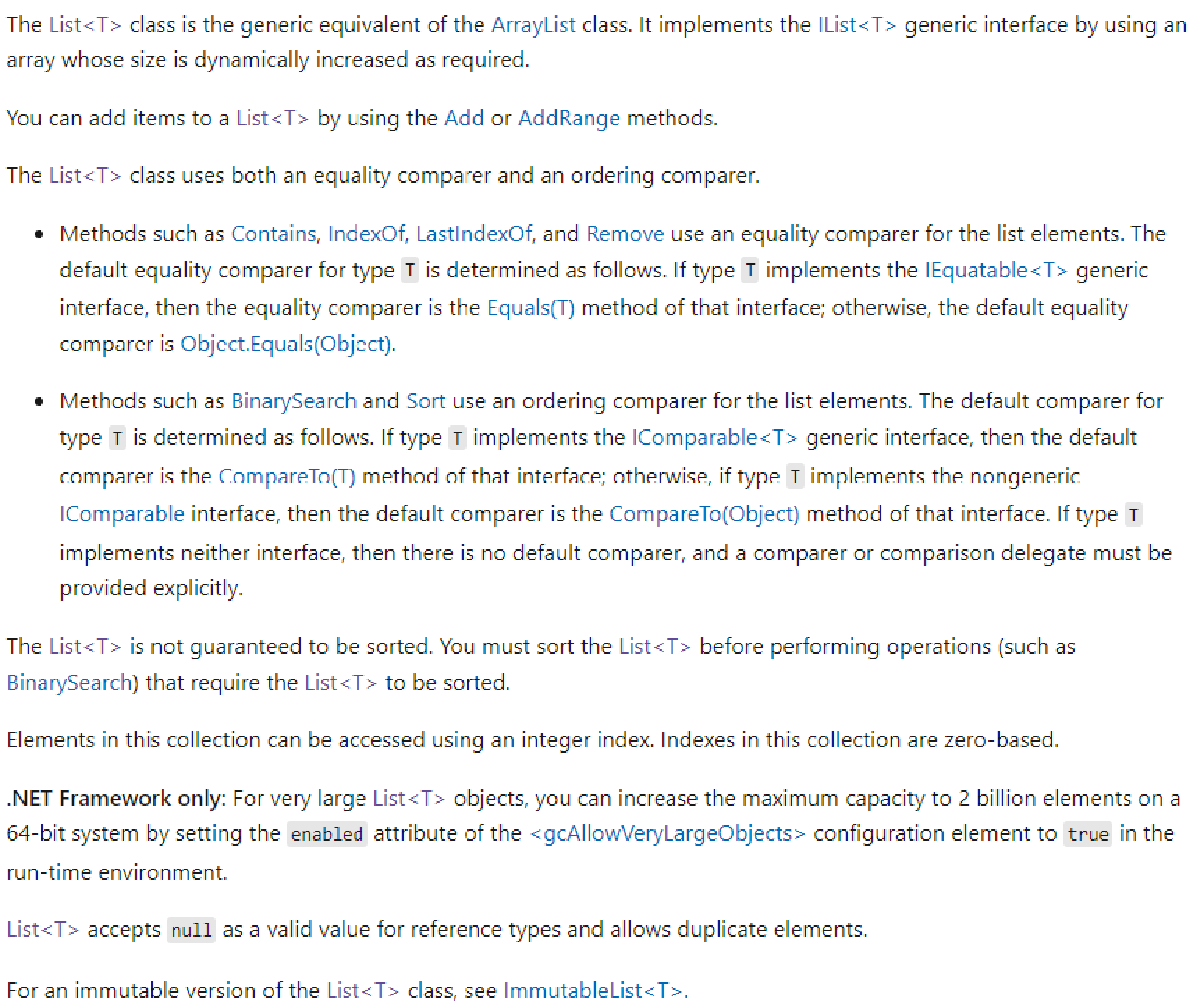
```
    }
}
```

Output:

```
D. ID: 1234   Name: crank arm
E. ID: 1334   Name: chain ring
F. ID: 1434   Name: regular seat
G. ID: 1444   Name: banana seat
H. ID: 1534   Name: cassette
I. ID: 1634   Name: shift lever
J.
K. Contains("1734"): False
```

```
L.
M.  Insert(2, "1834")
N.  ID: 1234   Name: crank arm
O.  ID: 1334   Name: chain ring
P.  ID: 1834   Name: brake lever
Q.  ID: 1434   Name: regular seat
R.  ID: 1444   Name: banana seat
S.  ID: 1534   Name: cassette
T.  ID: 1634   Name: shift lever
U.
V.  Parts[3]: ID: 1434   Name: regular seat
W.
X.  Remove("1534")
Y.
Z.  ID: 1234   Name: crank arm
AA. ID: 1334   Name: chain ring
BB. ID: 1834   Name: brake lever
CC. ID: 1434   Name: regular seat
DD. ID: 1444   Name: banana seat
EE. ID: 1634   Name: shift lever
FF.
GG. RemoveAt(3)
HH.
II. ID: 1234   Name: crank arm
JJ. ID: 1334   Name: chain ring
KK. ID: 1834   Name: brake lever
LL. ID: 1444   Name: banana seat
MM. ID: 1634   Name: shift lever
```

D. Remarks



The List<T> class is the generic equivalent of the ArrayList class. It implements the IList<T> generic interface by using an array whose size is dynamically increased as required.

You can add items to a List<T> by using the Add or AddRange methods.

The List<T> class uses both an equality comparer and an ordering comparer.

- Methods such as Contains, IndexOf, LastIndexOf, and Remove use an equality comparer for the list elements. The default equality comparer for type `T` is determined as follows. If type `T` implements the IEquatable<T> generic interface, then the equality comparer is the Equals(T) method of that interface; otherwise, the default equality comparer is Object.Equals(Object).
- Methods such as BinarySearch and Sort use an ordering comparer for the list elements. The default comparer for type `T` is determined as follows. If type `T` implements the IComparable<T> generic interface, then the default comparer is the CompareTo(T) method of that interface; otherwise, if type `T` implements the nongeneric IComparable interface, then the default comparer is the CompareTo(Object) method of that interface. If type `T` implements neither interface, then there is no default comparer, and a comparer or comparison delegate must be provided explicitly.

The List<T> is not guaranteed to be sorted. You must sort the List<T> before performing operations (such as BinarySearch) that require the List<T> to be sorted.

Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

.NET Framework only: For very large List<T> objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the `enabled` attribute of the <gcAllowVeryLargeObjects> configuration element to `true` in the run-time environment.

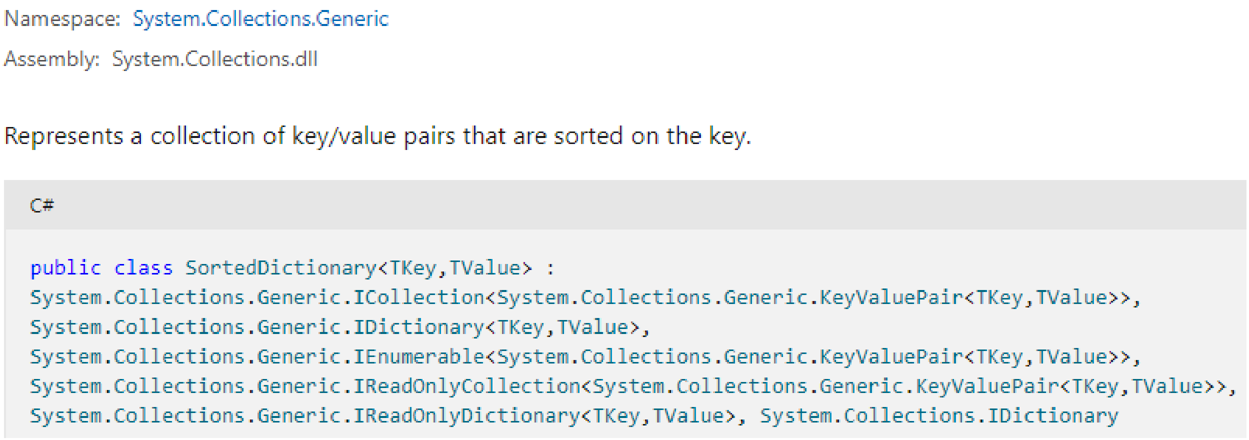
List<T> accepts `null` as a valid value for reference types and allows duplicate elements.

For an immutable version of the List<T> class, see ImmutableList<T>.

V- Technical Information: SortedDictionary<TKey, TValue>

Ref: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2?view=net-6.0

A. Definition



Namespace: System.Collections.Generic

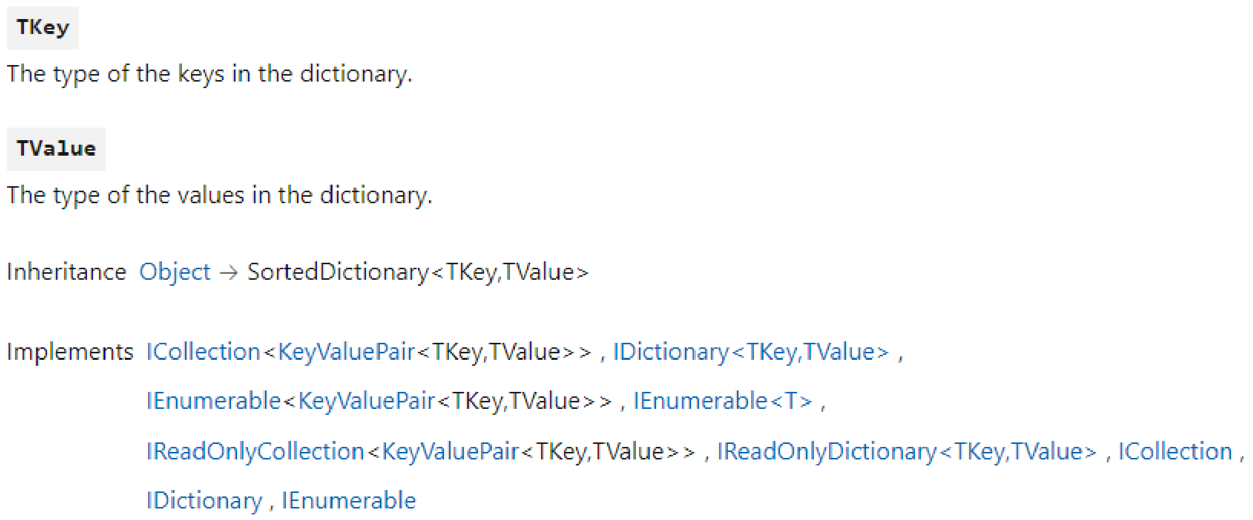
Assembly: System.Collections.dll

Represents a collection of key/value pairs that are sorted on the key.

C#

```csharp
public class SortedDictionary<TKey,TValue> :
System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
System.Collections.Generic.IDictionary<TKey,TValue>,
System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>, System.Collections.IDictionary
```

B. Type Parameters



`TKey`

The type of the keys in the dictionary.

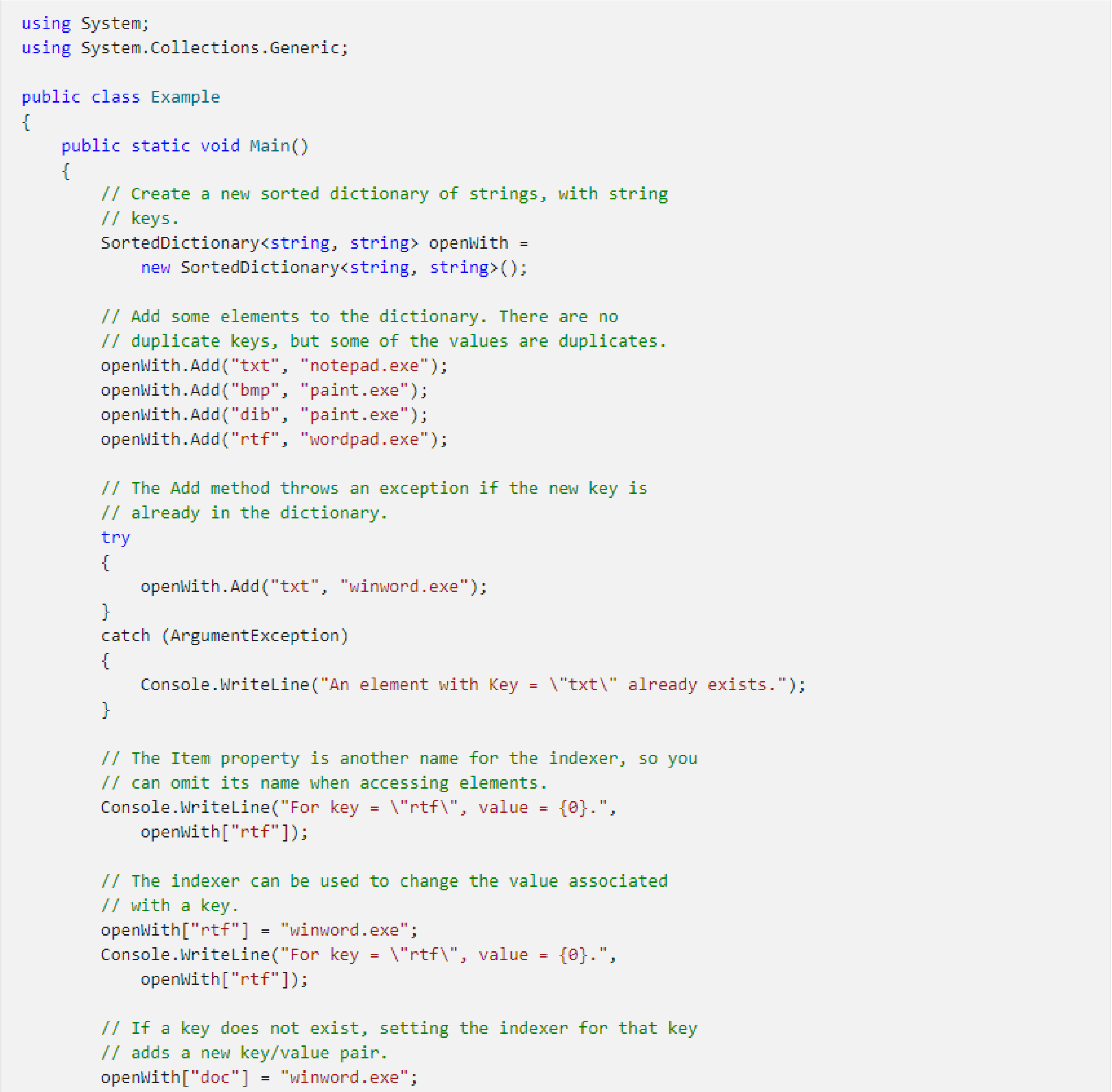
`TValue`

The type of the values in the dictionary.

Inheritance Object → SortedDictionary<TKey,TValue>

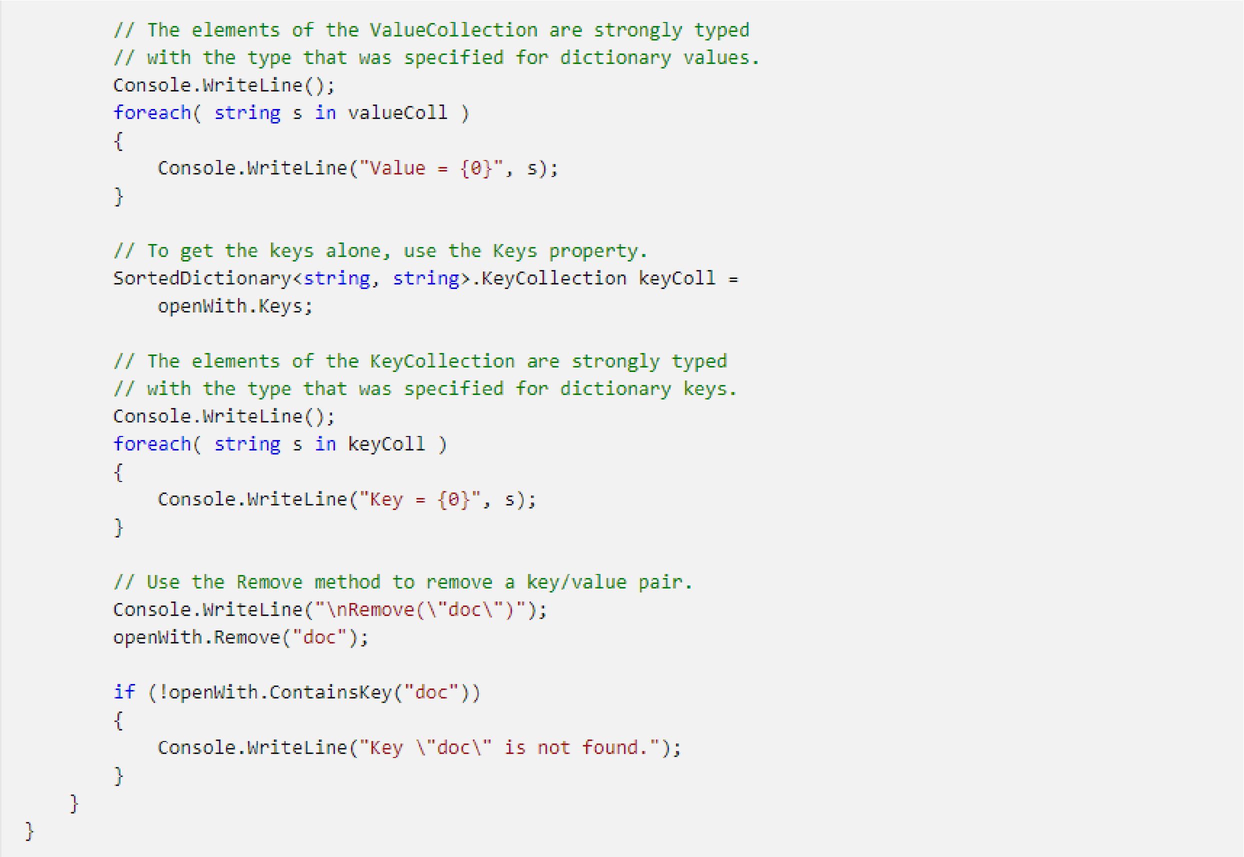
Implements ICollection<KeyValuePair<TKey,TValue>> , IDictionary<TKey,TValue> , IEnumerable<KeyValuePair<TKey,TValue>> , IEnumerable<T> , IReadOnlyCollection<KeyValuePair<TKey,TValue>> , IReadOnlyDictionary<TKey,TValue> , ICollection , IDictionary , IEnumerable

C. Examples

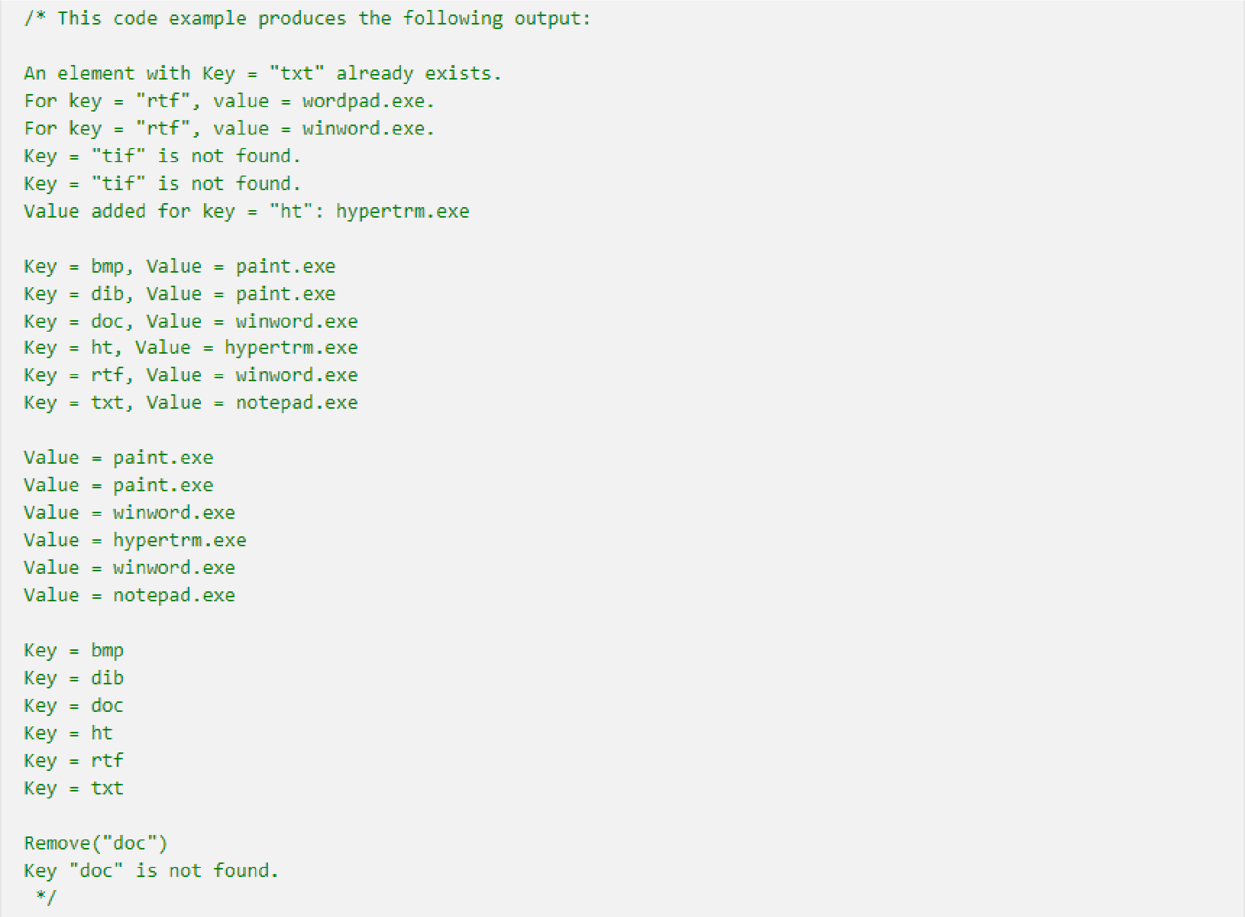


```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string
        // keys.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The Add method throws an exception if the new key is
        // already in the dictionary.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("An element with Key = \"txt\" already exists.");
        }

        // The Item property is another name for the indexer, so you
        // can omit its name when accessing elements.
        Console.WriteLine("For key = \"rtf\", value = {0}.",
            openWith["rtf"]);

        // The indexer can be used to change the value associated
        // with a key.
        openWith["rtf"] = "winword.exe";
        Console.WriteLine("For key = \"rtf\", value = {0}.",
            openWith["rtf"]);

        // If a key does not exist, setting the indexer for that key
        // adds a new key/value pair.
        openWith["doc"] = "winword.exe";
```

```
// The indexer throws an exception if the requested key is
// not in the dictionary.
try
{
    Console.WriteLine("For key = \"tif\", value = {0}.",
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Key = \"tif\" is not found.");
}

// When a program often has to try keys that turn out not to
// be in the dictionary, TryGetValue can be a more efficient
// way to retrieve values.
string value = "";
if (openWith.TryGetValue("tif", out value))
{
    Console.WriteLine("For key = \"tif\", value = {0}.", value);
}
else
{
    Console.WriteLine("Key = \"tif\" is not found.");
}

// ContainsKey can be used to test keys before inserting
// them.
if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Value added for key = \"ht\": {0}",
        openWith["ht"]);
}

// When you use foreach to enumerate dictionary elements,
// the elements are retrieved as KeyValuePair objects.
Console.WriteLine();
foreach( KeyValuePair<string, string> kvp in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        kvp.Key, kvp.Value);
}

// To get the values alone, use the Values property.
SortedDictionary<string, string>.ValueCollection valueColl =
    openWith.Values;
```



```
        // The elements of the ValueCollection are strongly typed
        // with the type that was specified for dictionary values.
        Console.WriteLine();
        foreach( string s in valueColl )
        {
            Console.WriteLine("Value = {0}", s);
        }

        // To get the keys alone, use the Keys property.
        SortedDictionary<string, string>.KeyCollection keyColl =
            openWith.Keys;

        // The elements of the KeyCollection are strongly typed
        // with the type that was specified for dictionary keys.
        Console.WriteLine();
        foreach( string s in keyColl )
        {
            Console.WriteLine("Key = {0}", s);
        }

        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"doc\")");
        openWith.Remove("doc");

        if (!openWith.ContainsKey("doc"))
        {
            Console.WriteLine("Key \"doc\" is not found.");
        }
    }
}
```



```
/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Key = "tif" is not found.
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe

Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = doc, Value = winword.exe
Key = ht, Value = hypertrm.exe
Key = rtf, Value = winword.exe
Key = txt, Value = notepad.exe

Value = paint.exe
Value = paint.exe
Value = winword.exe
Value = hypertrm.exe
Value = winword.exe
Value = notepad.exe

Key = bmp
Key = dib
Key = doc
Key = ht
Key = rtf
Key = txt

Remove("doc")
Key "doc" is not found.
 */
```

D. Remarks

The SortedDictionary<TKey,TValue> generic class is a binary search tree with O(log n) retrieval, where n is the number of elements in the dictionary. In this respect, it is similar to the SortedList<TKey,TValue> generic class. The two classes have similar object models, and both have O(log n) retrieval. Where the two classes differ is in memory use and speed of insertion and removal:

- SortedList<TKey,TValue> uses less memory than SortedDictionary<TKey,TValue>.
- SortedDictionary<TKey,TValue> has faster insertion and removal operations for unsorted data: O(log n) as opposed to O(n) for SortedList<TKey,TValue>.
- If the list is populated all at once from sorted data, SortedList<TKey,TValue> is faster than SortedDictionary<TKey,TValue>.

Each key/value pair can be retrieved as a KeyValuePair<TKey,TValue> structure, or as a DictionaryEntry through the nongeneric IDictionary interface.

Keys must be immutable as long as they are used as keys in the SortedDictionary<TKey,TValue>. Every key in a SortedDictionary<TKey,TValue> must be unique. A key cannot be `null`, but a value can be, if the value type `TValue` is a reference type.

SortedDictionary<TKey,TValue> requires a comparer implementation to perform key comparisons. You can specify an implementation of the IComparer<T> generic interface by using a constructor that accepts a `comparer` parameter; if you do not specify an implementation, the default genericcomparer Comparer<T>.Default is used. If type `TKey` implements the System.IComparable<T> generic interface, the default comparer uses that implementation.