# Collections

## 21.4.2 Class `Stack`

- It implements a stack data structure.
- Method `Push`
  - It takes an `object` as an argument and inserts it at the top of an `Stack`.
- Method `Pop`
  - It takes no arguments.
  - It removes and returns the `object` currently on a `Stack`'s top.
- Method `Peek`
  - It returns the value of the top stack element but does *not* remove the element from a `Stack`.

## 21.4.2 Class `Stack`

- Method `Contains`
  - It returns `true` if a `Stack` contains the specified `object`, and returns `false` otherwise.
- Property `Count`
  - It obtains the number of elements in a `Stack`.

- *Practice:* 23-1-StackTest-Practice
  - Download and follow the instructions

## 21.4.3 Hashing

- A high-speed scheme for converting keys to unique array indices.
  - i.e. social security numbers
- To store something
  - Convert the key rapidly to an index and the record of information could be stored at that index in the array.
- To retrieve something
  - Apply the conversion to the key, which produces the array index where the data resides in the array and retrieve the data.

## 21.4.3 Class `Hashtable`

- Hash table
  - A data structure to store data in hashing.

- When we convert a key into an array index, we literally *scramble* the bits, making a “hash” of the number.

- The number actually has no real significance beyond its usefulness in storing and retrieving this particular data record.

## 21.4.3 Arrays vs Hash Table

- Sorting and retrieving information with *arrays* is efficient
  - If some aspect of your data directly matches the key value.
  - If those keys are unique and *tightly packed*.
- Example:
  - If you have 100 employees with nine-digit social security numbers and you want to *store and retrieve employee data by using the social security number as a key*.
  - It would nominally require an array with 1,000,000,000 elements, because there are 1,000,000,000 unique nine-digit numbers.
  - If you have an array that large, you could get high performance storing and retrieving employee records by simply using the social security number as the array index, but it would be a *huge waste of memory*.
  - **Solution is to use a *hash table* instead of an array.**

## 21.4.3 Collisions in Hashing

- Collision
  - When for example two different keys “hash into” the same cell, or element, in the array.
  - Since we cannot sort two different data records to the same space, we need to find an alternative home for all records beyond the first that hash to a particular array index.

- The most popular solution
  - Have each cell of the table be a hash “bucket”
    - A linked list of all the key–value pairs that hash to that cell.
    - This is the solution that the .NET Framework’s `Hashtable` class implements.

## 21.4.3 Load Factor

$$\text{Load Factor} = \frac{\text{the number of objects stored in the hash table}}{\text{the total number of cells of the hash table}}$$

- It affects the performance of hashing schemes.
- Load factor
  - A space/time trade-off
  - Increasing: Better memory utilization, but poorer speed due to increased hashing collisions.
  - Decreasing: Better speed, but poorer memory utilization because of a larger portion of the hash table remains empty.

## 21.4.3 Hash Function

- A hash function performs a calculation that determines *where* to place data in the hash table.

  Is applied to the key in a key–value pair of objects.

- Class `Hashtable` can accept any `object` as a key.
- Class `object` defines method `GetHashCode`

  Which all objects inherit.

  – Most classes are candidates to be used as keys in a hash table override this method to provide more efficient hash-code calculations.

    - i.e. `string` has a hash-code calculation that's based on the contents of the `string`.

## 21.4.3 Class `Hashtable`

- Method `ContainsKey(Key)`

  Determines whether the key is in the hash table.

- Method `Add(key, value)`

  Creates a new entry in the has table.

  – `Hashtable` `set` accessor

- Notice:

  Using the `Add` method to add a key that already exists in the hash table causes an `ArgumentExecption`.

## 21.4.3 Class `Hashtable`

- `Hashtable` *Indexer*
  - If  a key is already exist in the hash table, you can use the `Hashtable`’s Indexer to obtain and set the key’s associated value.

    ```
    hashtable[ key ] = value;
    ```
  - `Hashtable` `get` and `set` accessor

## 21.4.3 Class `Hashtable`

- Property `Keys`
  - Read-only
  - Gets an `ICollection` of all the keys.
- Property `Values`
  - Read-only
  - Gets an `ICollection` of all the values.
  - Use this property to iterate through the values without regard for where they’re stored.
- Property `Count`
  - Gets the number of key-value pairs.

## 21.4.3 Class `Hashtable`

- *Example*: 23-2-HashtableTest
  - It uses a `Hashtable` to count the number of occurrences of each word in a `string`.
  - The app should use generic class `Dictionary<K, V>` (we will talk about later) rather than `Hashtable`.
- Note:

```
Regex.Split(string input, string pattern)
```

  - Splits an input string into an array of substrings at the positions defined by a regular expression pattern.
  - Exmple:

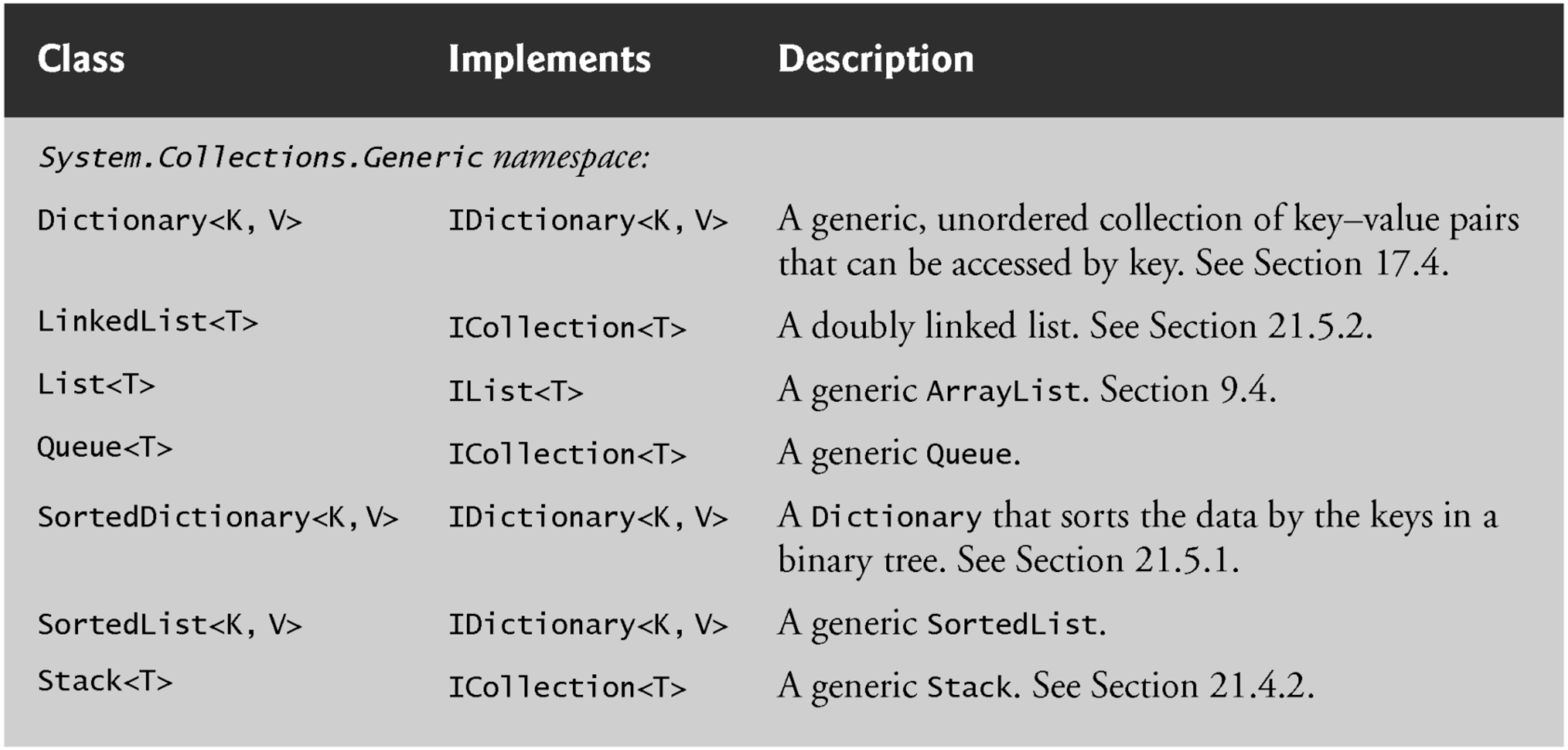
```
string[] words = Regex.Split( input, @"\s+" );
```

Divides the string by its whitespace characters.

## 21.4.3 DictionaryEntry and IDictionary

- DictionaryEntry
  - A structure to store key–value pairs.
  - The enumerator of a Hashtable (or any other class that implements IDictionary)
    - i.e. using a foreach statement with a Hashtable object.

# 21.5 Generic Collections



| Class | Implements | Description |
|---|---|---|
| *`System.Collections.Generic` namespace:* | | |
| `Dictionary<K, V>` | `IDictionary<K, V>` | A generic, unordered collection of key–value pairs that can be accessed by key. See Section 17.4. |
| `LinkedList<T>` | `ICollection<T>` | A doubly linked list. See Section 21.5.2. |
| `List<T>` | `IList<T>` | A generic `ArrayList`. Section 9.4. |
| `Queue<T>` | `ICollection<T>` | A generic `Queue`. |
| `SortedDictionary<K,V>` | `IDictionary<K, V>` | A `Dictionary` that sorts the data by the keys in a binary tree. See Section 21.5.1. |
| `SortedList<K, V>` | `IDictionary<K, V>` | A generic `SortedList`. |
| `Stack<T>` | `ICollection<T>` | A generic `Stack`. See Section 21.4.2. |

## 21.5.1 Generic Class `Dictionary`

- A dictionary is the general term for a collection of *key–value* pairs.
  - A hash table is one way to implement a dictionary.
  - The .NET Framework provides several implementations of dictionaries.
  - Both generic and nongeneric, all of them implement the `IDictionary` interface.

## 21.5.1 Generic Class `Dictionary`

- Class `Dictionary`
  - A generic collection of *key-value* pairs.
  - Method `ContainsKey`
    - To determine if a key exists in the `Dictionary` or not.
  - Method `Add`
    - To insert a new *key-value* pair.
  - Property `Keys`
    - Get a collection of all keys in the `Dictionary`

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

## 21.5.1 Generic Class `SortedDictionary`

- Class `SortedDictionary`
  - A generic class
  - It does not use a hash table, but instead stores its key–value pairs in a *binary search tree.*
  - The entries are sorted in the tree by *key*.
  - The keys implement generic interface `IComparable<T>`.
  - The `SortedDictionary` uses the results of `IComparable<T>` method `CompareTo` to sort the keys.
- Notice

  You can use the same public methods, properties and indexers with classes `Hashtable` and `SortedDictionary` in the same ways.

## 21.5.1 Generic Class `SortedDictionary`

- *Practice*: 23-2-HashtableTest-Practice

  Download and follow the instructions to use `SortedDictionary` instead of `Hashtable`

- Note:
  - Because class `SortedDictionary` keeps its elements in a binary tree, obtaining or inserting a *key*-*value* pair takes O(log n) time, which is fast compared to linear searching, then inserting.
  - Invoking the `get` accessor of a `SortedDictionary` indexer with a key that does not exist in the collection causes a `KeyNotFoundException`.