Collections

.NET

Collections *are ready-made classes that provide a more flexible way to work with groups of objects. The group of objects can grow and shrink dynamically as the needs of the application change.*

HTTPS://DOCS.MICROSOFT.COM/EN-US/DOTNET/CSHARP/PROGRAMMING-GUIDE/CONCEPTS/COLLECTIONS

Collections

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections

There are two ways to group similar objects in C#:

1. ***arrays*** of objects, and
2. ***collections*** of objects.

Collections provide a more flexible way to work with groups of objects. The group of objects can grow and shrink dynamically.

A ***collection*** is a ***class***, so an instance of the ***class*** must be created before elements can be added to that ***collection***.

If the collection contains elements of only one data type, the ***System.Collections.Generic*** namespace ca be used. A ***generic collection*** enforces type safety so that no other data type can be added to it.

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

C# Array Class

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/
https://docs.microsoft.com/en-us/dotnet/standard/generics/?view=net5.0

- The ***Array*** class is considered a collection because it is based on the IList ***interface***.
- It has methods for creating, manipulating, searching, and sorting arrays,
- length, and data type are set when the array instance is created and cannot be changed.
- An array can be Single-Dimensional, Multidimensional or Jagged.
- Numeric default values are zero (0).
- Reference default values are 'null'.
- Arrays are 'zero indexed' (They start at 0).

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Generic Collections (1/2)

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections#systemcollectionsgeneric-classes

---

The .NET class library provides a number of ***strongly-typed generic*** collection classes in the ***System.Collections.Generic*** and ***System.Collections.ObjectModel*** namespaces.

Many generic collection types are direct analogs of non-generic types.		
Dictionary<TKey,TValue>	Hashtable	Creates a Dictionary of any datatype key and value.
List<T>	ArrayList	Creates a list of any object type you need.
Queue<T> and Stack<T>	Queue and Stack	Creates a Queue or Stack of any object type.
SortedList<TKey,TValue>	SortedList	Both versions are hybrids of a dictionary and a list.

Generic Collections (1/2)

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2?view=net-5.0
https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-5.0

The .NET class library provides a number of ***strongly-typed generic*** collection classes in the ***System.Collections.Generic*** and ***System.Collections.ObjectModel*** namespaces.

Generic Structures Unique to .NET	
SortedDictionary<TKey,TValue>	This is a pure dictionary of key/value pairs sorted based on the value of the key.
LinkedList<T>	A linked list of any object type.

Generics

https://docs.microsoft.com/en-us/dotnet/standard/generics/?view=net5.0

Generics let you tailor a method, class, structure, or interface to the precise data ***type*** it acts upon.
When you create an instance of a ***generic*** class, you specify the ***types*** to substitute for the ***type*** parameters. This establishes a new ***generic*** class, referred to as a constructed ***generic*** class, with your chosen ***types*** substituted everywhere that the ***type*** parameters appear. The result is a ***type-safe*** class that is tailored to your choice of ***types***.

```
public static void Main()
{
    Generic<string> g = new Generic<string>();
    g.Field = "A string";
    //...
    Console.WriteLine("Generic.Field           = \"{0}\"", g.Field);
    Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName);
}
```

```
public class Generic<T>
{
    public T Field;
}
```

```
T Generic<T>(T arg)
{
    T temp = arg;
    //...
    return temp;
}
```

Generics - Terminology

https://docs.microsoft.com/en-us/dotnet/standard/generics/?view=net5.0

- generic type definition - a class, structure, or interface declaration that functions as a template with placeholders for its types.
- generic type parameters - type parameters. Placeholders in a generic type or method definition. Conventionally named <T>.
- constructed generic type - constructed type. The result of specifying types for the generic type parameters of a generic type definition.
- generic type argument - any type that is substituted for a generic type parameter.
- generic type - constructed types and generic type definitions.
- constraints - limits placed on generic type parameters. Arguments that do not satisfy the constraints cannot be used.
- generic method definition - a method with two parameter lists: a list of generic type parameters and a list of formal parameters. Type parameters can appear as the return type or as the types of the formal parameters.

Generic Methods

https://docs.microsoft.com/en-us/dotnet/standard/generics/?view=net5.0

- A method is ***generic*** only if it has its own list of type parameters.
- ***Generic*** methods can appear on ***generic*** or nongeneric types.
- A method is not ***generic*** just because it belongs to a ***generic*** type, or even because it has formal parameters whose types are the ***generic*** parameters of the enclosing type.
- In the following code, only method G is ***generic***.

```
class A
{
    T G<T>(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
class Generic<T>
{
    T M(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
```

Generics - Instantiation

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/
https://docs.microsoft.com/en-us/dotnet/standard/generics/?view=net5.0

On instantiation of a ***generic*** class, specify the actual ***types*** to substitute for the ***type*** parameters. This establishes a 'constructed ***generic*** class', with your chosen ***types*** substituted everywhere that the ***type*** parameters appear. The result is a ***type***-safe class that is tailored to your choice of ***types.***

```
public static void Main()
{
    Generic<string> g = new Generic<string>();
    g.Field = "A string";
    //...
    Console.WriteLine("Generic.Field            = \"{0}\"", g.Field);
    Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName);
}
```

List<T>

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections
https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-5.0

- List<T> represents a strongly typed list of objects
- Elements can be accessed by (zero-based) index[].
- Provides methods to search, sort, and manipulate lists.
- The List<T> class is the generic equivalent of the (Deprecated) ArrayList class.
- It implements the IList<T> generic interface by using an array whose size is dynamically increased as required.
- The List<T> is not guaranteed to be sorted.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "   " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

List<T>

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections
https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-5.0

- Add content using .Add().
- Use foreach() loop to iterate through the List.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

- Use a 'collection initializer' to add content of the specified type.
- Use a foreach() loop to iterate through the List.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

Dictionary<TKey,TValue>

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netcore-5.0

- Represents a collection of ***key***/***value*** pairs.
- The Dictionary<TKey,TValue> generic class provides a mapping from a set of ***keys*** to a set of ***values***. A ***key*** and its ***value*** must be added at the same time.
- A ***key*** cannot be null, but a ***value*** can be, if its type ***TValue*** is a reference type.
- As elements are added to a Dictionary<TKey,TValue>, the capacity is automatically increased as required by reallocating the internal array.

Dictionary<TKey,TValue> – Usage

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netcore-5.0

```
// Create a new dictionary of strings, with string keys.
//
Dictionary<string, string> openWith =
    new Dictionary<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.");
}
```

```
// 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"]);
}
```

```
// 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.");
}
```

Dictionary<TKey,TValue>

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netcore-5.0

The foreach statement returns an ***object*** representing the ***key/value*** pair in the ***collection***. Since the Dictionary<TKey,TValue> is a collection of ***keys*** and ***values***, the element ***type*** is not the ***type*** of the ***key*** or the ***type*** of the ***value***. Instead, the element ***type*** is a KeyValuePair<TKey,TValue> of the ***key*** type and the ***value*** type. The foreach statement is a wrapper around the enumerator, which allows only reading from the ***collection***, not writing to it.

```
foreach( KeyValuePair<string, string> kvp in myDictionary )
{
    Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
}
```

SortedList<TKey,TValue>

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1?view=net-5.0

- Represents a collection of ***key/value*** pairs that are sorted by ***key*** based on the associated IComparer<T> implementation.
- SortedList<TKey,TValue> is implemented as an array of ***key/value*** pairs, sorted by the ***key***.
- SortedList<TKey, TValue> is similar to the SortedDictionary<TKey,TValue> ***generic class***
  - SortedList<TKey,TValue> uses less memory than SortedDictionary<TKey,TValue>.
  - SortedDictionary<TKey,TValue> has faster insertion and removal operations for unsorted data.
  - If the list is populated all at once from sorted data, SortedList<TKey,TValue> is faster than SortedDictionary<TKey,TValue>.
  - SortedList<TKey,TValue> supports efficient indexed retrieval of keys and values
- The capacity can be decreased by calling .TrimExcess() or by setting the Capacity property explicitly.
- The foreach() statement is a wrapper around the enumerator and is readonly.

SortedList<TKey,TValue>

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2?view=net-5.0

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

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

        // Add some elements to the list. 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 list.
        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 list.
try
{
    Console.WriteLine("For key = \"tif\", value = {0}.",
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Key = \"tif\" is not found.");
}
```

```
// When you use foreach to enumerate list 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.
IList<string> ilistValues = openWith.Values;

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

SortedList<TKey,TValue>

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2?view=net-5.0

```
// The Values property is an efficient way to retrieve
// values by index.
Console.WriteLine("\nIndexed retrieval using the Values " +
    "property: Values[2] = {0}", openWith.Values[2]);

// To get the keys alone, use the Keys property.
IList<string> ilistKeys = openWith.Keys;

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

```
// The Keys property is an efficient way to retrieve
// keys by index.
Console.WriteLine("\nIndexed retrieval using the Keys " +
    "property: Keys[2] = {0}", openWith.Keys[2]);

// 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.");
}
```

Queue<T> (not used much)

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-5.0

- FIFO - Objects stored in a Queue<T> are inserted at one end and removed from the other.
- Use Queue<T> if you need to access the information in the same order that it is stored in the collection.
- Use ConcurrentQueue<T> if you need to access the collection from multiple threads concurrently.
- Queue<T> accepts null as a valid value for reference types and allows duplicate elements.
- Queues and stacks are useful when you need temporary storage for information
- Three main operations can be performed on a Queue<T> and its elements:
  - Enqueue() adds an element to the end of the Queue<T>.
  - Dequeue() removes the oldest element from the start of the Queue<T>.
  - Peek() returns the oldest element that is at the start of the Queue<T> but does not remove it from the Queue<T>.

Queue<T> (not used much)

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-5.0

```
class Example
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");

        // A queue can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
        Console.WriteLine("Peek at next item to dequeue: {0}",
            numbers.Peek());
        Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());
```

```
// Create a copy of the queue, using the ToArray method and the
// constructor that accepts an IEnumerable<T>.
Queue<string> queueCopy = new Queue<string>(numbers.ToArray());

Console.WriteLine("\nContents of the first copy:");
foreach( string number in queueCopy )
{
    Console.WriteLine(number);
}

// Create an array twice the size of the queue and copy the
// elements of the queue, starting at the middle of the
// array.
string[] array2 = new string[numbers.Count * 2];
numbers.CopyTo(array2, numbers.Count);
```

```
// Create a second queue, using the constructor that accepts an
// IEnumerable(Of T).
Queue<string> queueCopy2 = new Queue<string>(array2);

Console.WriteLine("\nContents of the second copy, with duplicates and nulls:");
foreach( string number in queueCopy2 )
{
    Console.WriteLine(number);
}

Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
    queueCopy.Contains("four"));

Console.WriteLine("\nqueueCopy.Clear()");
queueCopy.Clear();
Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
```

Stack<T> (not used much)

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1.push?view=net-5.0

- A (LIFO) collection of instances of the same specified type.
- Stack<T> is implemented as an array.
- Stacks (and Queues) are useful when you need temporary storage for information
- Use Stack<T> if you need to access the information in reverse order.
- Use System.Collections.Concurrent.ConcurrentQueue<T> when access is needed from multiple threads concurrently.
- System.Collections.Generic.Stack<T> preserves variable states during calls to other procedures.
- The capacity can be decreased by calling .TrimExcess().
- Stack<T> accepts null as a valid value for reference types and allows duplicate elements.
- Three main operations can be performed on a System.Collections.Generic.Stack<T> and its elements:
  - .Push() inserts an element at the top of the Stack<T>.
  - .Pop() removes an element from the top of the Stack<T>.
  - .Peek() returns an element that is at the top of the Stack<T> but does not remove it from the Stack<T>.

Stack<T> (not used much)

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1.push?view=net-5.0

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

class Example
{
    public static void Main()
    {
        Stack<string> numbers = new Stack<string>();
        numbers.Push("one");
        numbers.Push("two");
        numbers.Push("three");
        numbers.Push("four");
        numbers.Push("five");

        // A stack can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nPopping '{0}'", numbers.Pop());
        Console.WriteLine("Peek at next item to destack: {0}",
            numbers.Peek());
        Console.WriteLine("Popping '{0}'", numbers.Pop());
```

```
// Create a copy of the stack, using the ToArray method and the
// constructor that accepts an IEnumerable<T>.
Stack<string> stack2 = new Stack<string>(numbers.ToArray());

Console.WriteLine("\nContents of the first copy:");
foreach( string number in stack2 )
{
    Console.WriteLine(number);
}

// Create an array twice the size of the stack and copy the
// elements of the stack, starting at the middle of the
// array.
string[] array2 = new string[numbers.Count * 2];
numbers.CopyTo(array2, numbers.Count);
```

```
// Create a second stack, using the constructor that accepts an
// IEnumerable(Of T).
Stack<string> stack3 = new Stack<string>(array2);

Console.WriteLine("\nContents of the second copy, with duplicates and nulls:");
foreach( string number in stack3 )
{
    Console.WriteLine(number);
}

Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
    stack2.Contains("four"));

Console.WriteLine("\nstack2.Clear()");
stack2.Clear();
Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
```

Task

1. Create a List<>.
2. Create an object, Person, that contains a persons name, age, and telephone number.
3. add 5 Person objects to the List<>.
4. Sort the list by the age of the people.
5. Print the Name and Age of the sorted list.