Exception Handling and Collections(C#)

Exceptions are unexpected events that occur at runtime and cause the program tot halt, preventing further execution.

Handling exceptions helps in graceful termination of program execution, prevents crashes, and helps identify the root cause of the issue.

There are two types of exceptions in C# :- System exceptions and Application Exceptions.

1. System Exceptions :- The .NET framework throws System exceptions internally when something goes wrong in your code , and these exceptions can be caught within your code.

- They usually come from the .NET runtime, OS, or system-level issues.
- `System.NullReferenceException` : Thrown when you try to dereference a null object reference.
- `System.OutOfMemoryException` : Thrown when the system runs out of memory.
- `System.IO.IOException` : Thrown when there's an I/O error, like file access issues.
- `System.DivideByZeroException` : Thrown when attempting to divide by zero.

```
using System;

class Program
{
    static void Main()
    {
        try
        {
            // Example of a system exception: Divide by zero
            int result = 10 / 0;
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("System Exception Caught: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Execution continues after exception handling.");
        }
    }
}
```

2. Application Exceptions:- These are exceptions that are specifically thrown by the application itself when there are issues or business logic errors that need to be handled within the application's context. These exceptions come from the logic or intentional error conditions within the application code.

```
using System;

class Program
{
    // Custom ApplicationException
    public class MyApplicationException : ApplicationException
    {
        public MyApplicationException(string message) : base(message) { }
    }

    static void Main()
    {
        try
        {
            // Throwing a custom application exception
            throw new MyApplicationException("Something went wrong in the app
        }
        catch (MyApplicationException ex)
        {
            Console.WriteLine("Caught Application Exception: " + ex.Message);
            // Re-throwing the exception
            throw;
        }
        finally {
           Console.Writeline("Completed");

        }
    }
}
```

#Collections

Collections are objects that store and manipulate groups of data in C#. They provide efficient ways to manage and access data.

Collections offer features like dynamic size , type safety and built-in methods for adding , removing and searching elements.

Collections are widely used for data storage , retrieval , sorting and iteration in various programming scenarios.

1. Non-generic Collections (Store any type of data , no need to specify the datatype)

(i). ArrayList :- Flexible and dynamic array-like collection that can store objects if any type.

(ii). Hashtable :- Associative array that maps keys to values for efficient key-based data retrieval.

(iii). SortedList:- Collection that maintains elements in a sorted order based on the keys.

(iv). Stack

(v). Queue

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

```
class Program
{
    static void Main()
    {
        // ArrayList example
        ArrayList arrayList = new ArrayList();
        arrayList.Add(10);
        arrayList.Add("Hello");
        arrayList.Add(3.14);
        Console.WriteLine("ArrayList:");
        foreach (var item in arrayList)
            Console.WriteLine(item);

        // Hashtable example
        Hashtable hashtable = new Hashtable();
        hashtable.Add("Key1", "Value1");
        hashtable.Add("Key2", "Value2");
        Console.WriteLine("\nHashtable:");
        foreach (DictionaryEntry entry in hashtable)
            Console.WriteLine($"{entry.Key}: {entry.Value}");

        // SortedList example
        SortedList sortedList = new SortedList();
        sortedList.Add(2, "Second");
        sortedList.Add(1, "First");
        sortedList.Add(3, "Third");
        Console.WriteLine("\nSortedList:");
        foreach (DictionaryEntry entry in sortedList)
            Console.WriteLine($"{entry.Key}: {entry.Value}");

        // Stack example
        Stack stack = new Stack();
        stack.Push(10);
        stack.Push("Hello");
        stack.Push(3.14);
        Console.WriteLine("\nStack:");
        while (stack.Count > 0)
```

```
            Console.WriteLine(stack.Pop());

        // Queue example
        Queue queue = new Queue();
        queue.Enqueue(10);
        queue.Enqueue("Hello");
        queue.Enqueue(3.14);
        Console.WriteLine("\nQueue:");
        while (queue.Count > 0)
            Console.WriteLine(queue.Dequeue());
    }
}
```

2. Generic (DataType needs to be specified)

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

class Program
{
    static void Main()
    {
        // List<T> example
        List<int> list = new List<int>();
        list.Add(10);
        list.Add(20);
        list.Add(30);
        Console.WriteLine("List<T>:");
        foreach (var item in list)
            Console.WriteLine(item);

        // Dictionary<TKey, TValue> example
        Dictionary<string, string> dictionary = new Dictionary<string, string>();
        dictionary.Add("Key1", "Value1");
        dictionary.Add("Key2", "Value2");
        Console.WriteLine("\nDictionary<TKey, TValue>:");
        foreach (var entry in dictionary)
```

```
            Console.WriteLine($"{entry.Key}: {entry.Value}");

        // SortedList<TKey, TValue> example
        SortedList<int, string> sortedList = new SortedList<int, string>();
        sortedList.Add(2, "Second");
        sortedList.Add(1, "First");
        sortedList.Add(3, "Third");
        Console.WriteLine("\nSortedList<TKey, TValue>:");
        foreach (var entry in sortedList)
            Console.WriteLine($"{entry.Key}: {entry.Value}");

        // Stack<T> example
        Stack<string> stack = new Stack<string>();
        stack.Push("First");
        stack.Push("Second");
        stack.Push("Third");
        Console.WriteLine("\nStack<T>:");
        while (stack.Count > 0)
            Console.WriteLine(stack.Pop());

        // Queue<T> example
        Queue<double> queue = new Queue<double>();
        queue.Enqueue(10.5);
        queue.Enqueue(20.7);
        queue.Enqueue(30.9);
        Console.WriteLine("\nQueue<T>:");
        while (queue.Count > 0)
            Console.WriteLine(queue.Dequeue());
    }
}
```