Working with Collections

What are Collections in C#?

- Collections in C# are classes that provide a way to store and manage a group of related objects.
- Collections are used to handle objects that are logically related, such as lists, queues, or dictionaries.
- C# provides several built-in collection classes under the `System.Collections` and `System.Collections.Generic` namespaces.

Types of Collections

1. **Array**:

   - Fixed-size collection of elements of the same type.
   - Syntax for declaring an array:

   ```
   int[] numbers = new int[5];
   numbers[0] = 10;
   numbers[1] = 20;
   ```

2. **List**:

   - A generic collection that can grow or shrink dynamically.
   - Provides methods for adding, removing, and accessing elements.
   - Syntax:

   ```
   List<int> list = new List<int>();
   list.Add(10);
   list.Add(20);
   ```

3. **Dictionary<TKey, TValue>**:

   - A collection of key-value pairs.
   - Allows fast lookups by key.
   - Syntax:

   ```
   Dictionary<int, string> dict = new Dictionary<int, string>();
   dict.Add(1, "One");
   dict.Add(2, "Two");
   ```

4. **Queue**:

- A collection representing a first-in, first-out (FIFO) list of objects.
- Syntax:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("First");
queue.Enqueue("Second");
```

5. **Stack**:
   - A collection representing a last-in, first-out (LIFO) list of objects.
   - Syntax:

```
Stack<string> stack = new Stack<string>();
stack.Push("First");
stack.Push("Second");
```

Collection Methods

- Common methods used with collections include:
  - `Add()`: Adds an element.
  - `Remove()`: Removes an element.
  - `Contains()`: Checks if an element exists.
  - `Clear()`: Removes all elements.
  - `Count`: Returns the number of elements.

Data Table

What is a Data Table?

- A **DataTable** is an in-memory representation of a single table of data.
- It is part of the `System.Data` namespace and is used in ADO.NET to store data retrieved from a database.

Creating a Data Table

- You can create a DataTable by defining columns and adding rows.
- Syntax:

```
DataTable dt = new DataTable();
dt.Columns.Add("ID", typeof(int));
dt.Columns.Add("Name", typeof(string));

dt.Rows.Add(1, "John");
dt.Rows.Add(2, "Jane");
```

Working with DataTable

- You can perform various operations on a DataTable, like filtering, sorting, and accessing individual rows.
- Example:

```
foreach (DataRow row in dt.Rows)
{
    Console.WriteLine($"ID: {row["ID"]}, Name: {row["Name"]}");
}
```

Using DataTable with DataAdapter

- A `DataAdapter` is used to fill a DataTable with data from a database.

```
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Users",
connection);
DataTable dt = new DataTable();
adapter.Fill(dt);
```

Accessing and Modifying Data in DataTable

- You can access a specific row or column in a DataTable using indexers.

```
DataRow row = dt.Rows[0]; // Access the first row
Console.WriteLine(row["Name"]); // Access the "Name" column of the
first row
```

Exception Handling

What is Exception Handling?

- Exception handling in C# provides a way to handle runtime errors and ensure that the program can continue to execute after an error occurs.
- It uses `try`, `catch`, `finally` blocks to manage exceptions.

Syntax of Exception Handling

```
try
{
    // Code that might throw an exception
}
catch (ExceptionType ex)
{
    // Code to handle the exception
}
finally
```

```
{
    // Code that runs regardless of whether an exception was thrown
}
```

Exception Types

- `Exception`: The base class for all exceptions.
- Common derived classes include:
  - `System.NullReferenceException`: Thrown when you try to access a null object.
  - `System.IO.IOException`: Thrown when an I/O error occurs (file not found, etc.).
  - `System.DivideByZeroException`: Thrown when attempting to divide by zero.

Throwing Exceptions

- You can manually throw exceptions using the `throw` keyword:

```
if (age < 0)
{
    throw new ArgumentOutOfRangeException("Age cannot be negative.");
}
```

Handling Multiple Exceptions

- You can catch different types of exceptions using multiple `catch` blocks:

```
try
{
    int result = 10 / 0;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Cannot divide by zero.");
}
catch (Exception ex)
{
    Console.WriteLine("An error occurred: " + ex.Message);
}
```

Finally Block

- The `finally` block is optional and runs after the `try` and `catch` blocks, regardless of whether an exception was thrown.

```
try
{
    // Code
}
```

```
catch (Exception ex)
{
    // Handle exception
}
finally
{
    // Code that always runs (e.g., cleanup code)
}
```

Custom Exceptions

- You can create custom exceptions by inheriting from the `Exception` class.

```
public class InvalidAgeException : Exception
{
    public InvalidAgeException(string message) : base(message) { }
}
```

- Example usage:

```
throw new InvalidAgeException("Age must be between 1 and 100.");
```

Best Practices for Exception Handling

- Use exceptions to handle exceptional, unforeseen errors, not for regular control flow.
- Catch specific exceptions rather than a general `Exception` class.
- Avoid empty `catch` blocks; log the exception or rethrow it.
- Always clean up resources in the `finally` block.

Different Project Types in C

In C#, there are several types of projects that you can create depending on the application you are developing. These projects vary in functionality and target environments. Here are some common types:

1. Console Application

- **Description**: A console application is a simple application that runs in a command-line environment. It's a text-based interface where the user interacts with the application through the console window.
- **Uses**: Suitable for utilities, learning programming basics, or backend processing.
- **Example**: Simple calculators, command-line tools.

2. Windows Forms Application

- **Description**: Windows Forms applications are used to create graphical user interfaces (GUIs) on Windows operating systems. It uses controls like buttons, textboxes, and labels to build the interface.
- **Uses**: Desktop applications like media players, text editors.
- **Example**: A simple text editor or a calculator with GUI.

3. WPF (Windows Presentation Foundation) Application

- **Description**: WPF is used for building modern Windows desktop applications with rich graphical interfaces. It supports more advanced graphics, animations, and data binding.
- **Uses**: Desktop applications with complex UIs, advanced graphics.
- **Example**: Complex desktop applications like accounting software or graphical design tools.

4. ASP.NET Core Application

- **Description**: ASP.NET Core is used for creating web applications. It is a modern, cross-platform framework for building web applications and APIs.
- **Uses**: Websites, web services, and web APIs.
- **Example**: E-commerce sites, RESTful APIs.

5. Class Library

- **Description**: A class library project is a collection of classes and functions that can be used by other applications.
- **Uses**: Creating reusable libraries that can be shared across different applications.
- **Example**: Utility libraries, frameworks, or custom class libraries for an application.

6. Xamarin Application

- **Description**: Xamarin is used for building mobile applications for Android, iOS, and Windows using a single C# codebase.
- **Uses**: Cross-platform mobile applications.
- **Example**: Mobile apps like social media clients, task management apps.

7. Azure Functions

- **Description**: Azure Functions allows you to run small pieces of code (functions) in the cloud without having to manage the underlying infrastructure.
- **Uses**: Serverless applications, cloud-triggered functions.
- **Example**: Event-driven applications that respond to cloud events.

8. Blazor Application

- **Description**: Blazor is a framework for building interactive web UIs using C# instead of JavaScript. It can run on the client-side via WebAssembly or server-side.

- **Uses**: Interactive web applications with C# on both server and client sides.
- **Example**: Web-based dashboards, e-commerce platforms.