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.

Working with DateTime Class in C

Basic File Operations in C

The `System.IO` namespace provides various classes to work with files, such as `File`, `FileInfo`, `StreamReader`, and `StreamWriter`.

Common File Operations:

1. Reading Files:

- **StreamReader**: Used to read text from a file.

```
using (StreamReader reader = new StreamReader("file.txt"))
{
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
```

- **File.ReadAllText()**: Reads the entire content of a file.

```
string content = File.ReadAllText("file.txt");
```

2. Writing Files:

- **StreamWriter**: Used to write text to a file.

```
using (StreamWriter writer = new StreamWriter("file.txt"))
{
    writer.WriteLine("Hello, World!");
}
```

- **File.WriteAllText()**: Writes text to a file, creating the file if it doesn't exist.

```
File.WriteAllText("file.txt", "Hello, World!");
```

3. File Existence:

- **File.Exists()**: Checks if a file exists.

```
bool exists = File.Exists("file.txt");
```

4. Copying Files:

- **File.Copy()**: Copies a file to a new location.

```
File.Copy("source.txt", "destination.txt");
```

5. Deleting Files:

- **File.Delete()**: Deletes a specified file.

```
File.Delete("file.txt");
```

6. Appending Text to a File:

- **File.AppendAllText()**: Appends text to a file.

```
File.AppendAllText("file.txt", "Appended Text");
```

ASP.NET Web Application (.NET Framework) – 5 Types

1. Empty Web Application

Overview:

The Empty Web Application is a minimal project template, providing a basic structure without predefined components. It's ideal when you want complete control over which components you add to your project.

Project Structure:

```
/EmptyWebApp
  ├── /App_Data
  ├── /Content
  ├── /Scripts
  ├── /Views
  ├── Global.asax
  ├── Web.config
```

- **App_Data**: Directory for database files, data, or other data-related resources.
- **Content**: Stores static files like CSS and images.
- **Scripts**: Stores JavaScript files.
- **Views**: The folder where your Razor views reside (if you add MVC).

Key Files:

1. **Global.asax**:
   - Handles application-level events, such as `Application_Start`, `Application_End`, etc.

```
<%@ Application Language="C#" Inherits="System.Web.HttpApplication" %>

<script runat="server">
```

```
void Application_Start(object sender, EventArgs e) {
    // Code that runs on application startup
}
</script>
```

2. **Web.config**:

   - Configuration file for the web application, like database connection strings, routing, and security settings.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <!-- Your app settings -->
  </appSettings>
  <connectionStrings>
    <!-- Your connection strings -->
  </connectionStrings>
</configuration>
```

---

2. Web Forms Application

Overview:

Web Forms is a traditional framework for building web pages in ASP.NET. It uses a drag-and-drop approach with controls like TextBoxes, Buttons, and Grids. This template is commonly used for enterprise-level web apps and forms-based sites.

Project Structure:

```
/WebFormsApp
  ├── /App_Data
  ├── /Content
  ├── /Scripts
  ├── /Pages
  │   └── Default.aspx
  ├── Global.asax
  ├── Web.config
```

Key Files:

1. **Default.aspx**:

   - A typical Web Forms page containing HTML markup and server controls.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="Default.aspx.cs"
Inherits="WebFormsApp._Default" %>

<html>
  <body>
```

```
    <form id="form1" runat="server">
      <div>
        <asp:Label runat="server" ID="Label1" Text="Hello, Web Forms!"
/>
        <asp:Button runat="server" Text="Click Me"
OnClick="Button1_Click" />
      </div>
    </form>
  </body>
</html>
```

2. **Default.aspx.cs**:

   – The code-behind file where you handle server-side logic, such as button clicks.

```
using System;

namespace WebFormsApp
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Button1_Click(object sender, EventArgs e)
        {
            Label1.Text = "Button clicked!";
        }
    }
}
```

---

3. MVC Web Application

Overview:

MVC (Model-View-Controller) is a design pattern that separates application logic into three components: Model (data), View (UI), and Controller (business logic). It's suitable for applications that require more complex and maintainable code.

Project Structure:

```
/MvcApp
  ├── /Controllers
  │   └── HomeController.cs
  ├── /Models
  │   └── WeatherForecast.cs
  ├── /Views
  │   └── /Home
  │       └── Index.cshtml
  ├── Global.asax
  ├── Web.config
```

Key Files:

1. **HomeController.cs**:
   - The controller that handles HTTP requests and returns appropriate views.

```
using System.Web.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            var model = new WeatherForecast { Date = "2024-11-27",
Summary = "Sunny", TemperatureC = 22 };
            return View(model);
        }
    }
}
```

2. **Index.cshtml**:
   - The Razor view that represents the HTML page for the Index action.

```
@model MvcApp.Models.WeatherForecast

<h1>Weather Forecast</h1>
<p>Date: @Model.Date</p>
<p>Temperature: @Model.TemperatureC °C</p>
<p>Summary: @Model.Summary</p>
```

3. **Web.config**:
   - Contains configuration settings for routing, security, etc.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.7.2" />
  </system.web>
</configuration>
```

---

4. Web API Application

Overview:

Web API applications allow you to create RESTful APIs that can be consumed by various clients. It's commonly used for building back-end services.

Project Structure:

```
/WebAPIApp
  ├── /Controllers
  │   └── WeatherController.cs
  ├── /Models
  │   └── WeatherForecast.cs
  ├── Global.asax
  ├── Web.config
```

Key Files:

1. **WeatherController.cs**:
   - The API controller that handles HTTP requests and returns data in JSON format.

```
using System.Collections.Generic;
using System.Web.Http;

namespace WebAPIApp.Controllers
{
    public class WeatherController : ApiController
    {
        public IEnumerable<WeatherForecast> Get()
        {
            return new List<WeatherForecast>
            {
                new WeatherForecast { Date = "2024-11-27", TemperatureC
= 20, Summary = "Sunny" },
                new WeatherForecast { Date = "2024-11-28", TemperatureC
= 15, Summary = "Cloudy" }
            };
        }
    }
}
```

2. **WeatherForecast.cs** (Model):

```
namespace WebAPIApp.Models
{
    public class WeatherForecast
    {
        public string Date { get; set; }
        public int TemperatureC { get; set; }
        public string Summary { get; set; }
    }
}
```

5. Single Page Application (SPA)

Overview:

SPA projects use client-side technologies like Angular, React, or Vue.js to create dynamic, single-page applications. The back-end is usually an API server that handles HTTP requests.

Project Structure:

```
/SPAApp
  ├── /ClientApp (Angular/React/Vue)
  ├── /Controllers
  │   └── ApiController.cs
  ├── Web.config
```

Key Files:

1. **ApiController.cs** (Web API Controller):

```
using System.Collections.Generic;
using System.Web.Http;

namespace SPAApp.Controllers
{
    public class ApiController : ApiController
    {
        public IEnumerable<string> Get()
        {
            return new string[] { "Value1", "Value2" };
        }
    }
}
```

2. **Web.config**:

   – Configuration settings, including API routes and security for the back-end.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.7.2" />
  </system.web>
</configuration>
```