1. **Understanding Exceptions in C#**

- **Exercise**: Write a program that intentionally throws a `DivideByZeroException`. Catch this exception and display a user-friendly error message.

- **Solution**:

```csharp
try
{
    int x = 10;
    int y = 0;
    int result = x / y;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: Attempted to divide by zero.");
}
```

2. **Using Try & Catch Blocks**

- **Exercise**: Create a program that reads an integer from the user and catches exceptions related to invalid input.

- **Solution**:

```csharp
try
{
    Console.WriteLine("Enter a number:");
    int number = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine($"You entered: {number}");
}
catch (FormatException ex)
{
    Console.WriteLine("Error: Invalid input, please enter a valid integer.");
}
```

3. **Throwing Exceptions**

- **Exercise**: Write a method that throws an `ArgumentOutOfRangeException` if the input is negative.
- **Solution**:

```csharp
public static void CheckNumber(int number)
{
    if (number < 0)
    {
        throw new ArgumentOutOfRangeException("number", "Number cannot be negative");
    }
}

try
{
    CheckNumber(-5);
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
```

```
}
```

4. **Finally Keyword**

- **Exercise**: Write a program demonstrating the use of a `finally` block that closes a file even if an exception occurs.

- **Solution**:

```csharp
StreamReader reader = null;
try
{
    reader = new StreamReader("test.txt");
    string content = reader.ReadToEnd();
}
catch (Exception ex)
{
    Console.WriteLine("Error: Could not read the file.");
}
finally
```

```
    {
        if (reader != null)
        {
            reader.Close();
            Console.WriteLine("File closed.");
        }
    }
    ```


5. **Writing Custom Exceptions**
  - **Exercise**: Create a custom exception `NegativeNumberException` and use it in a method.
  - **Solution**:
    ```csharp
    public class NegativeNumberException : Exception
    {
        public NegativeNumberException(string message) : base(message) { }
    }
```

```
    public static void ValidateNumber(int number)
    {
        if (number < 0)
        {
            throw new
NegativeNumberException("Negative numbers are
not allowed.");
        }
    }

    try
    {
        ValidateNumber(-10);
    }
    catch (NegativeNumberException ex)
    {
        Console.WriteLine(ex.Message);
    }
    ```

6. **Global Exception Handling**

- **Exercise**: Implement a global exception handler in a console application.
- **Solution**:

```csharp
static void Main(string[] args)
{

AppDomain.CurrentDomain.UnhandledException += GlobalExceptionHandler;


    throw new Exception("Global exception test.");
}


static void GlobalExceptionHandler(object sender, UnhandledExceptionEventArgs e)
{
    Console.WriteLine("A global exception has occurred: " + ((Exception)e.ExceptionObject).Message);
}
```

7. **Garbage Collection Basics**

- **Exercise**: Demonstrate how garbage collection works by forcing a collection using `GC.Collect`.
- **Solution**:

```csharp
class MyClass
{
    ~MyClass()
    {
        Console.WriteLine("Destructor called.");
    }
}

static void Main(string[] args)
{
    MyClass obj = new MyClass();
    obj = null;
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

8. **Mark-Sweep Algorithm**

- **Exercise**: Explain the mark-sweep algorithm with a simple example code.

- **Solution**: (No direct code, but an explanation):

  - The mark-sweep algorithm marks live objects and sweeps away the unreferenced objects. This process helps in garbage collection in managed languages like C#.

9. **Finalizers**

- **Exercise**: Create a class with a finalizer and demonstrate when it is called.

- **Solution**:

  ```csharp
  class MyClass
  {
      ~MyClass()
      {
          Console.WriteLine("Finalizer called.");
      }
  }
  ```

```
    static void Main(string[] args)
    {
        MyClass obj = new MyClass();
        obj = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
    ```

10. **IDisposable Interface**
   - **Exercise**: Implement the `IDisposable` interface in a class and demonstrate proper disposal of resources.
   - **Solution**:
     ```csharp
    class MyResource : IDisposable
    {
        public void Dispose()
        {
            Console.WriteLine("Resources released.");
```

```
    }
}

static void Main(string[] args)
{
    using (MyResource resource = new MyResource())
    {
        // Use resource
    }
}
```

11. **Dispose Method**

- **Exercise**: Write a class that manually disposes of its resources by implementing `Dispose` method.
- **Solution**:

```csharp
class MyResource : IDisposable
{
    private bool disposed = false;
```

```
public void Dispose()
{
   Dispose(true);
   GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
   if (!disposed)
   {
      if (disposing)
      {
         // Dispose managed resources.
      }

      // Dispose unmanaged resources.
      disposed = true;
   }
}
```

```
    ~MyResource()
    {
        Dispose(false);
    }
}
```

12. **Handling Strings in C#**

- **Exercise**: Write a program that concatenates strings efficiently using `StringBuilder`.
- **Solution**:

```csharp
StringBuilder sb = new StringBuilder();
sb.Append("Hello");
sb.Append(" ");
sb.Append("World!");

Console.WriteLine(sb.ToString());
```

13. **String Operations**

- **Exercise**: Perform various string operations like substring, replace, and case conversion.

- **Solution**:

```csharp
string text = "Hello World!";
Console.WriteLine(text.Substring(0, 5)); // Hello
Console.WriteLine(text.Replace("World", "C#")); // Hello C#!
Console.WriteLine(text.ToUpper()); // HELLO WORLD!
```

14. **StringBuilder for Performance**

- **Exercise**: Compare performance between using `String` and `StringBuilder` for multiple concatenations.

- **Solution**:

```csharp
// String Concatenation
string result = "";
for (int i = 0; i < 10000; i++)
{
```

```
    result += "Hello ";
}

// StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.Append("Hello ");
}
```

15. **Builder Design Pattern**

- **Exercise**: Implement the Builder Design Pattern to construct complex objects.
- **Solution**:

```csharp
public class Product
{
    public string Name { get; set; }
    public double Price { get; set; }
}
```

```
public class ProductBuilder
{
    private Product product = new Product();

    public ProductBuilder SetName(string name)
    {
        product.Name = name;
        return this;
    }

    public ProductBuilder SetPrice(double price)
    {
        product.Price = price;
        return this;
    }

    public Product Build()
    {
        return product;
    }
```

```
    }

    static void Main(string[] args)
    {
        Product product = new ProductBuilder()
                        .SetName("Laptop")
                        .SetPrice(1000.00)
                        .Build();
        Console.WriteLine($"Product: {product.Name}, Price: {product.Price}");
    }
    ```

16. **Introduction to Regular Expressions**

  - **Exercise**: Write a program that validates an email address using regular expressions.

  - **Solution**:

    ```csharp
    string pattern = @"^[^@\s]+@[^@\s]+\.[^@\s]+$";
    string input = "test@example.com";
```

```
    bool isValid = Regex.IsMatch(input, pattern);
    Console.WriteLine($"Is valid email: {isValid}");
    ```

17. **Using the Regex Class**
  - **Exercise**: Write a program that finds all occurrences of a pattern in a string using `Regex.Matches`.
  - **Solution**:
    ```csharp
    string input = "One two three two one";
    string pattern = @"\btwo\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match match in matches)
    {
        Console.WriteLine($"Found '{match.Value}' at index {match.Index}");
    }
    ```

18. **Match Method**

  - **Exercise**: Write a program that uses `Regex.Match` to find

 and extract a substring from text.

  - **Solution**:

    ```csharp
    string input = "The price is $100";
    string pattern = @"\$\d+";

    Match match = Regex.Match(input, pattern);
    if (match.Success)
    {
        Console.WriteLine($"Found match: {match.Value}");
    }
    ```

19. **Advanced Exception Handling**

- **Exercise**: Write a program that catches multiple exceptions of different types.
- **Solution**:

```csharp
try
{
    int[] array = new int[5];
    Console.WriteLine(array[10]); // IndexOutOfRangeException
    int x = int.Parse("abc"); // FormatException
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Array index is out of range.");
}
catch (FormatException ex)
{
    Console.WriteLine("Input is not in the correct format.");
}
```

20. **Nested Try-Catch Blocks**

- **Exercise**: Write a program demonstrating nested try-catch blocks.

- **Solution**:

```csharp
try
{
    try
    {
        int[] array = new int[5];
        Console.WriteLine(array[10]); // IndexOutOfRangeException
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine("Array index is out of range.");
        throw; // Re-throwing the exception
    }
}
```

```
    catch (Exception ex)
    {
        Console.WriteLine("Caught a re-thrown exception.");
    }
    ```

21. **Chained Exception Handling**
  - **Exercise**: Write a program that demonstrates chained exceptions (inner exceptions).
  - **Solution**:
    ```csharp
    try
    {
        try
        {
            int x = int.Parse("abc"); // FormatException
        }
        catch (FormatException ex)
        {
```

```
            throw new
InvalidOperationException("Operation failed", ex);
        }
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Inner exception: " +
ex.InnerException.Message);
    }
    ```


22. **Logging Exceptions**
  - **Exercise**: Write a program that logs
exceptions to a file.
  - **Solution**:
    ```csharp
    try
    {
        int x = int.Parse("abc"); // FormatException
    }
    catch (Exception ex)
```

```
    {
        File.WriteAllText("log.txt", ex.ToString());
        Console.WriteLine("Exception logged.");
    }
    ```

23. **IDisposable in Custom Classes**
  - **Exercise**: Implement `IDisposable` in a custom class and use it in a `using` statement.
  - **Solution**:
    ```csharp
    class MyResource : IDisposable
    {
        public void Dispose()
        {
            Console.WriteLine("Resource disposed.");
        }
    }

    static void Main(string[] args)
    {
```

```
        using (MyResource resource = new MyResource())
        {
            // Use resource
        }
    }
    ```

24. **Memory Leaks and Garbage Collection**

- **Exercise**: Write a program that demonstrates a memory leak scenario and how garbage collection handles it.

- **Solution**: Explanation (No direct code):

  - Memory leaks can occur in unmanaged resources if not properly disposed. The garbage collector automatically handles managed objects, but unmanaged resources need explicit disposal.

25. **Weak References**

- **Exercise**: Write a program using weak references to prevent objects from being prematurely garbage collected.

- **Solution**:

```csharp
WeakReference weakRef = new WeakReference(new MyClass());

if (weakRef.IsAlive)
{
    MyClass obj = weakRef.Target as MyClass;
    // Use obj
}
```

26. **Finalizers and Object Resurrection**

- **Exercise**: Write a program that demonstrates object resurrection using finalizers.
- **Solution**:

```csharp
class MyClass
{
    public static MyClass instance;
    ~MyClass()
    {
```

```
            instance = this; // Resurrecting the object
        }
    }

    static void Main(string[] args)
    {
        MyClass obj = new MyClass();
        obj = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();

        if (MyClass.instance != null)
        {
            Console.WriteLine("Object resurrected.");
        }
    }
    ```
```

### 27. **Advanced StringBuilder Operations**

- **Exercise**: Write a program that performs complex string manipulations using `StringBuilder`.
```

- **Solution**:

  ```csharp
  StringBuilder sb = new StringBuilder("Welcome to ");
  sb.Append("C# ");
  sb.Append("Programming!");
  sb.Insert(0, "Hello, ");
  sb.Replace("Programming", "World");

  Console.WriteLine(sb.ToString()); // Hello, Welcome to C# World!
  ```

28. **Regex for Validation**

- **Exercise**: Write a program that validates a phone number using regular expressions.
- **Solution**:

  ```csharp
  string pattern = @"^\d{3}-\d{3}-\d{4}$";
  string input = "123-456-7890";
  ```

```
    bool isValid = Regex.IsMatch(input, pattern);
    Console.WriteLine($"Is valid phone number: {isValid}");
    ```


29. **Regex for Searching**
  - **Exercise**: Write a program that extracts all email addresses from a given text using regular expressions.
  - **Solution**:
    ```csharp
    string input = "Contact us at support@example.com or sales@example.com";
    string pattern = @"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match match in matches)
    {
        Console.WriteLine($"Found email: {match.Value}");
```

```
}
```

30. **StringBuilder for File Operations**

- **Exercise**: Write a program that uses `StringBuilder` to efficiently write a large amount of text to a file.

- **Solution**:

```csharp
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.AppendLine("Line " + i);
}

File.WriteAllText("output.txt", sb.ToString());
```