1. **Exceptions in C#**

   - **Definition**: Exceptions are runtime errors that occur during the execution of a program. They can result from various issues, such as invalid input, resource unavailability, or logic errors.

   - **Purpose**: The primary purpose of exceptions is to provide a mechanism for handling errors gracefully, without crashing the application.

2. **Try & Catch Blocks**

   - **Try Block**: A `try` block contains the code that might throw an exception. It's followed by one or more `catch` blocks.

   - **Catch Block**: A `catch` block is used to handle the exception if it occurs. Each `catch` block can handle a specific type of exception.

   - **Example**:

     ```csharp
     try
     {
         // Code that might throw an exception
     }
     catch (ExceptionType e)
     ```

    {
        // Code to handle the exception
    }

3. **Throw Keyword**
   - **Definition**: The `throw` keyword is used to manually raise an exception. It can be used to throw a built-in exception or a custom one.
   - **Usage**:
     ```csharp
     throw new Exception("This is an error!");
     ```

4. **Finally Keyword**
   - **Definition**: The `finally` block contains code that is always executed, regardless of whether an exception is thrown or not. It's typically used to release resources like file handles or database connections.
   - **Example**:
     ```csharp
     ```

```
try
{
    // Code that might throw an exception
}
catch (ExceptionType e)
{
    // Code to handle the exception
}
finally
{
    // Code that will always execute
}
```

5. **Writing Custom Exceptions**

- **Definition**: In C#, you can create your own exception classes by deriving from the base class `Exception`. This is useful when you want to create more meaningful or specific exceptions for your application.
- **Example**:

```csharp
```

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

6. **Global Exception Handling**

   - **Definition**: Global exception handling is the process of handling exceptions that are not caught by any `try-catch` block in the application. In console applications, this can be done using the `AppDomain.UnhandledException` event.

   - **Purpose**: It ensures that even unexpected errors that aren't specifically handled elsewhere in the code are managed gracefully.

7. **Garbage Collection**

   - **Definition**: Garbage collection is an automatic memory management feature in C#. It frees up memory occupied by objects that are no longer in use.

- **Purpose**: The purpose of garbage collection is to prevent memory leaks and optimize memory usage by reclaiming memory from unused objects.

8. **Mark-Sweep Algorithm**

- **Definition**: The mark-sweep algorithm is one of the fundamental garbage collection techniques. It works in two phases:

  - **Mark Phase**: Identifies all live objects by marking them.

  - **Sweep Phase**: Reclaims memory from unmarked objects (those that are no longer in use).

9. **Finalizers**

- **Definition**: A finalizer is a special method in C# (`~ClassName()`) that is called by the garbage collector when an object is no longer accessible. It is used to clean up unmanaged resources.

- **Note**: Finalizers are non-deterministic, meaning you can't predict exactly when they will be executed.

10. **IDisposable Interface**

   - **Definition**: The `IDisposable` interface is used to release unmanaged resources deterministically. It contains a single method, `Dispose`, which is called to clean up resources manually.

   - **Usage**:

     ```csharp
     public class MyClass : IDisposable
     {
         public void Dispose()
         {
             // Cleanup code here
         }
     }
     ```

11. **Dispose Method**

   - **Definition**: The `Dispose` method is part of the `IDisposable` interface and is used to release unmanaged resources explicitly. Unlike finalizers, `Dispose` is called by the developer, usually when an object is no longer needed.

  - **Best Practice**: Always call `Dispose` on objects that implement `IDisposable` to avoid memory leaks.

12. **Handling Strings in C#**

  - **String Operations**: C# provides various methods to manipulate strings, such as concatenation, substring, replace, and case conversion.

  - **Example**:

    ```csharp
    string str = "Hello World!";
    string newStr = str.Replace("World", "C#");
    ```

13. **StringBuilder for Performance**

  - **Definition**: `StringBuilder` is a class in C# used for efficiently manipulating strings when multiple modifications are required. It is more performance-efficient than using regular strings because it doesn't create a new string object with each modification.

  - **Usage**:

    ```csharp
    ```

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

14. **Builder Design Pattern**
  - **Definition**: The Builder Design Pattern is a creational pattern that allows you to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
  - **Usage**:
    ```csharp
    public class ProductBuilder
    {
        private Product product = new Product();

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

```
        }

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

15. **Regular Expressions (Regex)**

   - **Definition**: Regular expressions are patterns used to match character combinations in strings. They are used for searching, extracting, and modifying text based on defined patterns.

   - **Example**: Validating an email:

    ```csharp
    string pattern =
@"^[^@\s]+@[^@\s]+\.[^@\s]+$";
    bool isValid = Regex.IsMatch(input, pattern);
    ```

16. **Regex Class**

  - **Definition**: The `Regex` class in C# provides a way to work with regular expressions. It contains methods for matching strings against patterns, replacing substrings, splitting strings, and more.

  - **Usage**:

    ```csharp
    Regex regex = new Regex(@"\d+");
    Match match = regex.Match("123 ABC");
    ```

17. **Match Method**

  - **Definition**: The `Match` method of the `Regex` class is used to find a match for a regular expression pattern in a string. It returns a `Match` object, which provides information about the match, such as the index and length of the matched substring.

  - **Usage**:

    ```csharp
    Match match = Regex.Match("The price is $100",
@"\$\d+");
    if (match.Success)
    {
        Console.WriteLine(match.Value);
    ```

```
}
```

These concepts are essential for understanding various aspects of C# programming, from exception handling and memory management to string manipulation and design patterns. By mastering these topics, you'll be able to write more robust, efficient, and maintainable C# code.