C# Topics by Skill Level

Basic Topics

1. **Introduction to C#**
   - Overview of .NET Framework/.NET Core
   - Setting up the development environment (Visual Studio/VS Code)
   - Hello World program
2. **C# Syntax**
   - Data types and variables
   - Constants and literals
   - Type conversion and casting
3. **Operators**
   - Arithmetic, comparison, logical, and bitwise operators
   - Assignment operators
   - Null-coalescing and ternary operators
4. **Control Statements**
   - `if`, `else`, `else if`
   - `switch` statements
   - Loops: `for`, `while`, `do-while`, `foreach`
5. **Functions and Methods**
   - Defining and calling methods
   - Method parameters: value, reference, and `out`
   - Method overloading
   - `Main()` method structure
6. **Arrays and Strings**
   - Single-dimensional and multi-dimensional arrays
   - Array methods (`Length`, `Sort`, etc.)
   - String manipulation (`Substring`, `Split`, `Replace`, etc.)
7. **Exception Handling**
   - `try`, `catch`, `finally`
   - Common exceptions
   - Throwing exceptions
8. **Basic Input/Output**
   - `Console.ReadLine()` and `Console.WriteLine()`
   - File operations (basic `StreamReader` and `StreamWriter`)

Intermediate Topics

1. **Object-Oriented Programming (OOP)**
   - Classes and objects
   - Constructors and destructors
   - Properties and fields
   - `this` keyword
2. **Inheritance and Polymorphism**
   - Base and derived classes
   - Overriding methods
   - Abstract classes and interfaces
3. **Collections and Generics**
   - Collections (`List`, `Dictionary`, `Queue`, `Stack`)
   - Generics (`List<T>`, `Dictionary<TKey, TValue>`)
   - Iterators and `yield`
4. **Delegates and Events**
   - Introduction to delegates
   - Multicast delegates
   - Events and event handlers
5. **LINQ (Language Integrated Query)**
   - LINQ basics (`Select`, `Where`, `OrderBy`)
   - Query and method syntax
   - LINQ to Objects, XML, and SQL
6. **Asynchronous Programming**
   - `async` and `await`
   - Task-based asynchronous pattern (TAP)
   - `Task` and `Task<T>`
7. **File Handling**
   - File and directory operations (`File`, `Directory`, `Path`)
   - Reading and writing files (`StreamReader`, `StreamWriter`, `FileStream`)
   - Serialization and deserialization (JSON and XML)
8. **Custom Exceptions**
   - Creating custom exception classes
   - Throwing and catching custom exceptions

Advanced Topics

1. **Advanced OOP Concepts**
    - Partial classes and methods
    - Extension methods
    - Static classes and methods
    - Sealed classes
2. **Memory Management**
    - Garbage collection
    - Finalizers and `Dispose` method
    - `IDisposable` and `using` statement
3. **Threading and Parallel Programming**
    - Multithreading with `Thread` class
    - Thread synchronization (`lock`, `Monitor`, `Mutex`)
    - `Task Parallel Library` (TPL) and PLINQ
4. **Reflection and Attributes**
    - Introduction to reflection
    - Working with metadata
    - Creating and using custom attributes
5. **Dependency Injection**
    - Principles of DI
    - Using DI frameworks like `Microsoft.Extensions.DependencyInjection`
6. **Networking**
    - Working with `HttpClient`
    - Sockets and TCP/IP programming
    - REST API consumption
7. **Advanced LINQ**
    - LINQ expressions and custom providers
    - Deferred execution
    - Query optimization
8. **Design Patterns**
    - Creational patterns (Singleton, Factory, Builder)
    - Structural patterns (Adapter, Proxy)
    - Behavioral patterns (Observer, Strategy)
9. **Performance Optimization**
    - Benchmarking and profiling
    - Using Span<T> and Memory<T>
    - Avoiding common performance pitfalls
10. **Advanced Asynchronous Programming**
    - Cancellation tokens
    - Progress reporting
    - Synchronization contexts
11. **Interoperability**
    - COM Interop
    - Platform Invocation Services (P/Invoke)
    - Using unmanaged code in C#
12. **Testing and Debugging**
    - Unit testing frameworks (`NUnit`, `xUnit`)
    - Mocking and stubs
    - Advanced debugging techniques (Visual Studio diagnostics, Debugger attributes)

C#- 01

Introduction to C#: A Beginner's Guide

C# is a modern, object-oriented programming language developed by Microsoft as part of its .NET initiative. It is widely used for building desktop, web, and mobile applications.

Overview of .NET Framework and .NET Core

What is .NET Framework?

The **.NET Framework** is a Windows-only development framework designed for building desktop and web applications. It includes:

- A large class library called **Framework Class Library (FCL)**.
- **Common Language Runtime (CLR)** for runtime services like memory management and security.
- Compatibility with Windows Forms, WPF, and ASP.NET applications.

What is .NET Core?

.NET Core is a cross-platform, open-source successor to the .NET Framework. It offers:

- **Cross-platform compatibility** (Windows, macOS, Linux).
- High performance and scalability.
- Unified development for cloud, IoT, and web applications.

Setting Up the Development Environment

To start coding in C#, you need the following:

1. **Install Visual Studio or Visual Studio Code**
    - Download Visual Studio for a complete IDE.
    - Download Visual Studio Code for a lightweight editor.
2. **Install .NET SDK**
    - Download .NET SDK for building and running .NET applications.

Verify Installation Open a terminal or command prompt and run:
bash
Copy code
```
dotnet --version
```

3. This should display the installed .NET version.

Hello World Program in C#

Let's write and run a simple **Hello World** program to get started.

Code Example

Using Visual Studio

1. Create a new project:
    - Open Visual Studio, select **Create a new project**.
    - Choose **Console App (.NET Core)** and click **Next**.
    - Name the project `HelloWorld` and click **Create**.

Replace the default code with the following:
csharp
Copy code
```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

2.
3. Run the program:

Press `F5` or click the **Run** button to see the output:
Copy code
```
Hello, World!
```

    ○

Using Visual Studio Code

1. Create a new folder and open it in VS Code.

Open the terminal and initialize a new project:
bash
Copy code
```
dotnet new console -n HelloWorld
cd HelloWorld
```

2.

Open `Program.cs` and edit the code if necessary:
csharp
Copy code
```csharp
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

3.

Run the program:
bash
Copy code
```bash
dotnet run
```
Output:
Copy code
```
Hello, World!
```

C#-02

Understanding C# Syntax: A Comprehensive Guide

C# is a statically-typed, object-oriented programming language. This guide covers the basics of C# syntax, including data types, variables, constants, literals, type conversion, and casting.

C# Syntax

C# syntax is similar to other C-style languages like Java or C++. The structure of a C# program typically includes:

1. **Namespaces**: Provide a way to organize classes and prevent name conflicts.
2. **Classes**: Define the structure and behavior of objects.
3. **Methods**: Contain executable code.

Basic Program Structure

csharp
Copy code
```csharp
using System;

namespace MyNamespace
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to C# Programming!");
        }
    }
}
```

Data Types and Variables

Data Types

C# has two types of data: **value types** and **reference types**.

1. **Value Types**: Store data directly (e.g., `int`, `float`, `bool`).
2. **Reference Types**: Store references to memory locations (e.g., `string`, arrays, classes).

Common Data Types

Data Type	Description	Example
`int`	Integer values	`42`
`float`	Floating-point numbers	`3.14f`
`double`	Double-precision float	`3.14159`
`char`	Single character	`'A'`
`bool`	Boolean values (`true/false`)	`true`
`string`	Sequence of characters	`"Hello"`

Example

csharp
Copy code

```
int age = 25;
float height = 5.9f;
bool isStudent = true;
string name = "Alice";

Console.WriteLine($"Name: {name}, Age: {age}, Height: {height}, Student: {isStudent}");
```

Variables

A **variable** is a named storage location for data.

Declaring Variables

csharp
Copy code

```
<datatype> <variableName> = <value>;
```

Example

csharp
Copy code

```
int number = 10;    // Integer variable
string greeting = "Hello, World!"; // String variable
```

Constants and Literals

Constants

A constant is a value that cannot be changed during the program's execution. Use the `const` keyword to declare constants.

Example

csharp
Copy code

```
const double Pi = 3.14159;
Console.WriteLine($"The value of Pi is {Pi}");
```

Literals

Literals represent fixed values in code.

Examples

- Integer literal: `42`
- Floating-point literal: `3.14`
- String literal: `"Hello"`
- Character literal: `'A'`

Type Conversion and Casting

Type Conversion

Changing a value from one data type to another.

Implicit Conversion

Automatically performed when there is no loss of data.

csharp
Copy code
```csharp
int num = 10;
double result = num;  // Implicit conversion from int to double
Console.WriteLine(result); // Outputs: 10
```

Explicit Conversion (Casting)

Requires the `cast` operator when there is potential data loss.

csharp
Copy code
```csharp
double price = 9.99;
int discountedPrice = (int)price;  // Explicit casting
Console.WriteLine(discountedPrice); // Outputs: 9
```

Using Convert Class

The `Convert` class provides methods for type conversion.

csharp
Copy code
```csharp
string str = "123";
int number = Convert.ToInt32(str);
Console.WriteLine(number); // Outputs: 123
```

Code Example: Combining It All

csharp
Copy code
```csharp
using System;

namespace CSharpBasics
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables and Data Types
            int age = 30;
            float salary = 45000.75f;
            char grade = 'A';
            bool isEmployed = true;

            // Constants
            const double Pi = 3.14159;

            // Type Conversion
            string ageText = age.ToString(); // Int to String
            double bonus = Convert.ToDouble(salary); // Float to Double

            // Explicit Casting
            int truncatedSalary = (int)salary;

            // Display
            Console.WriteLine($"Age: {age}, Salary: {salary}, Grade: {grade}, Employed: {isEmployed}");
            Console.WriteLine($"Pi Value: {Pi}");
            Console.WriteLine($"Age as Text: {ageText}, Bonus: {bonus}, Truncated Salary: {truncatedSalary}");
        }
    }
}
```

C#-03

Understanding C# Operators: A Comprehensive Guide

Operators in C# are special symbols or keywords used to perform operations on variables and values. These operators can be categorized into arithmetic, comparison, logical, bitwise, assignment, and conditional operators, each serving a unique purpose in programming.

---

1. Arithmetic Operators

Arithmetic operators perform basic mathematical operations.

Operator	Description	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus (Remainder)	`x % y`

Example

csharp
Copy code

```
int a = 10, b = 3;
Console.WriteLine(a + b);  // 13
Console.WriteLine(a % b); // 1
```

---

2. Comparison Operators

Comparison operators compare two values and return a boolean result.

Operator	Description	Example
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`>`	Greater than	`x > y`
`<`	Less than	`x < y`
`>=`	Greater than or equal to	`x >= y`
`<=`	Less than or equal to	`x <= y`

Example

csharp
Copy code

```
int x = 5, y = 10;
Console.WriteLine(x > y);  // False
Console.WriteLine(x != y); // True
```

---

3. Logical Operators

Logical operators are used to perform logical operations on boolean expressions.

Operator	Description	Example
`&&`	Logical AND	`x && y`
`	`	`
`!`	Logical NOT	`!x`

Example

csharp
Copy code

```
bool a = true, b = false;
Console.WriteLine(a && b);  // False
Console.WriteLine(a || b);  // True
Console.WriteLine(!a);      // False
```

---

4. Bitwise Operators

Bitwise operators work on binary representations of integers.

Operator	Description	Example
`&`	Bitwise AND	`x & y`
`	`	Bitwise OR
`^`	Bitwise XOR	`x ^ y`
`~`	Bitwise Complement	`~x`
`<<`	Left Shift	`x << y`
`>>`	Right Shift	`x >> y`

Example

csharp
Copy code

```
int x = 5, y = 3; // Binary: 0101 and 0011
Console.WriteLine(x & y);  // 1 (0001)
Console.WriteLine(x | y);  // 7 (0111)
```

---

5. Assignment Operators

Assignment operators assign values to variables.

Operator	Description	Example
`=`	Assign	`x = y`
`+=`	Add and assign	`x += y`
`-=`	Subtract and assign	`x -= y`
`*=`	Multiply and assign	`x *= y`
`/=`	Divide and assign	`x /= y`
`%=`	Modulus and assign	`x %= y`

Example

csharp
Copy code

```
int x = 5;
```

```
x += 3;  // x = x + 3
Console.WriteLine(x); // 8
```

---

6. Null-Coalescing and Ternary Operators

Null-Coalescing Operator (`??`)

Provides a default value when the variable is `null`.

csharp
Copy code
```
string name = null;
string result = name ?? "Default Name";
Console.WriteLine(result); // Default Name
```

Ternary Operator (`?:`)

A shorthand for `if-else` conditions.

csharp
Copy code
```
int age = 20;
string eligibility = (age >= 18) ? "Eligible" : "Not Eligible";
Console.WriteLine(eligibility); // Eligible
```

Code Example: Operators in Action

csharp
Copy code
```
using System;

namespace OperatorDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Arithmetic Operators
            int a = 10, b = 3;
            Console.WriteLine($"Addition: {a + b}");
            Console.WriteLine($"Modulus: {a % b}");

            // Comparison Operators
            Console.WriteLine($"Is {a} greater than {b}? {a > b}");

            // Logical Operators
            bool x = true, y = false;
            Console.WriteLine($"x AND y: {x && y}");

            // Assignment Operators
            int num = 5;
            num += 3;
            Console.WriteLine($"After +=, num = {num}");

            // Null-Coalescing and Ternary Operators
            string name = null;
            string output = name ?? "Anonymous";
            Console.WriteLine(output);

            int age = 20;
            Console.WriteLine(age >= 18 ? "Adult" : "Minor");
        }
    }
}
```

C#-04

Control Statements in C#: A Comprehensive Guide

Control statements in C# allow developers to direct the flow of a program based on conditions and repetitive tasks. These include conditional statements (`if`, `else if`, `else`, and `switch`) and looping constructs (`for`, `while`, `do-while`, and `foreach`).

---

1. Conditional Statements

If Statement

The `if` statement executes a block of code only if a specified condition evaluates to `true`.

Syntax
csharp
Copy code
```
if (condition)
{
    // Code to execute if the condition is true
}
```

Example
csharp
Copy code
```
int age = 18;
if (age >= 18)
{
    Console.WriteLine("You are eligible to vote.");
}
```

If-Else Statement

The `if-else` statement specifies an alternative block of code to execute if the condition is `false`.

Example
csharp
Copy code
```
int number = 10;
if (number % 2 == 0)
{
    Console.WriteLine("Even Number");
}
else
{
    Console.WriteLine("Odd Number");
}
```

Else-If Ladder

The `else if` statement allows for multiple conditions to be checked sequentially.

Example
csharp
Copy code
```
int marks = 75;
if (marks >= 90)
{
    Console.WriteLine("Grade: A");
}
else if (marks >= 75)
{
    Console.WriteLine("Grade: B");
}
else
{
    Console.WriteLine("Grade: C");
```

```
}
```

Switch Statement

The `switch` statement tests a variable or expression against multiple values (called cases) and executes a block corresponding to the first match.

Syntax
csharp
Copy code

```
switch (expression)
{
    case value1:
        // Code block
        break;
    case value2:
        // Code block
        break;
    default:
        // Default block
        break;
}
```

Example
csharp
Copy code

```
char grade = 'B';
switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent");
        break;
    case 'B':
        Console.WriteLine("Good");
        break;
    default:
        Console.WriteLine("Needs Improvement");
        break;
}
```

---

2. Loops

For Loop

A `for` loop repeats a block of code a specified number of times.

Syntax
csharp
Copy code

```
for (initialization; condition; increment/decrement)
{
    // Code block
}
```

Example
csharp
Copy code

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Iteration {i}");
}
```

While Loop

The `while` loop continues executing as long as its condition evaluates to `true`.

Syntax
csharp
Copy code

```
while (condition)
{
    // Code block
}
```

Example

csharp

Copy code

```
int counter = 0;
while (counter < 5)
{
    Console.WriteLine($"Counter: {counter}");
    counter++;
}
```

Do-While Loop

The `do-while` loop executes the code block at least once before evaluating the condition.

Syntax

csharp

Copy code

```
do
{
    // Code block
} while (condition);
```

Example

csharp

Copy code

```
int num = 0;
do
{
    Console.WriteLine($"Number: {num}");
    num++;
} while (num < 5);
```

Foreach Loop

The `foreach` loop iterates over elements in a collection.

Syntax

csharp

Copy code

```
foreach (var item in collection)
{
    // Code block
}
```

Example

csharp

Copy code

```
string[] fruits = { "Apple", "Banana", "Cherry" };
foreach (string fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

Comprehensive Code Example

csharp

Copy code

```
using System;

namespace ControlStatements
{
    class Program
```

```
{
    static void Main(string[] args)
    {
        // If-Else Example
        int age = 20;
        if (age < 18)
            Console.WriteLine("Minor");
        else
            Console.WriteLine("Adult");

        // Switch Example
        int day = 3;
        switch (day)
        {
            case 1: Console.WriteLine("Monday"); break;
            case 2: Console.WriteLine("Tuesday"); break;
            case 3: Console.WriteLine("Wednesday"); break;
            default: Console.WriteLine("Other day"); break;
        }

        // For Loop Example
        for (int i = 0; i < 3; i++)
            Console.WriteLine($"For Loop Iteration: {i}");

        // While Loop Example
        int count = 0;
        while (count < 3)
        {
            Console.WriteLine($"While Loop Count: {count}");
            count++;
        }

        // Do-While Loop Example
        int num = 0;
        do
        {
            Console.WriteLine($"Do-While Loop Number: {num}");
            num++;
        } while (num < 3);

        // Foreach Example
        int[] numbers = { 1, 2, 3 };
        foreach (int n in numbers)
            Console.WriteLine($"Foreach Loop: {n}");
    }
}
```

C#-05

Functions and Methods in C#

Functions and methods form the backbone of programming in C#. They allow you to organize, reuse, and structure your code effectively. In this article, we will cover the basics of defining and calling methods, method parameters, method overloading, and the structure of the `Main()` method.

Defining and Calling Methods

A method in C# is a block of code that performs a specific task. It can be called or invoked whenever needed. Methods are defined inside a class.

Syntax
csharp
Copy code

```
returnType MethodName(parameters)
{
    // Code to execute
    return value; // If returnType is not void
}
```

Example: A Simple Method
csharp
Copy code

```csharp
using System;

class Program
{
    static void Greet()
    {
        Console.WriteLine("Hello, welcome to C#!");
    }

    static void Main(string[] args)
    {
        Greet(); // Calling the method
    }
}
```

Method Parameters

Parameters allow methods to accept input values, making them more dynamic.

1. Value Parameters

Value parameters pass the value of arguments to the method. Changes made to parameters inside the method do not affect the original variable.

Example

csharp
Copy code

```csharp
static void Add(int a, int b)
{
    Console.WriteLine($"Sum: {a + b}");
}

static void Main(string[] args)
{
    Add(5, 10); // Output: Sum: 15
}
```

2. Reference Parameters

Reference parameters allow methods to modify the original variable by using the `ref` keyword.

Example

csharp
Copy code

```csharp
static void DoubleValue(ref int number)
{
    number *= 2;
}

static void Main(string[] args)
{
    int value = 5;
    DoubleValue(ref value);
    Console.WriteLine(value); // Output: 10
}
```

3. Out Parameters

The `out` keyword allows a method to return multiple values.

Example

csharp
Copy code

```csharp
static void Divide(int a, int b, out int quotient, out int remainder)
{
    quotient = a / b;
    remainder = a % b;
}

static void Main(string[] args)
{
    int q, r;
    Divide(10, 3, out q, out r);
```

```
    Console.WriteLine($"Quotient: {q}, Remainder: {r}");
}
```

Method Overloading

Method overloading allows multiple methods with the same name but different parameters. The compiler determines the method to call based on the argument list.

Example
csharp
Copy code
```
static void Display(int number)
{
    Console.WriteLine($"Number: {number}");
}

static void Display(string text)
{
    Console.WriteLine($"Text: {text}");
}

static void Main(string[] args)
{
    Display(10);       // Calls the first method
    Display("Hello");  // Calls the second method
}
```

The Main() Method Structure

The `Main()` method serves as the entry point of a C# program. It can take an array of strings as parameters for command-line arguments.

Example
csharp
Copy code
```
class Program
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            Console.WriteLine($"Hello, {args[0]}!");
        }
        else
        {
            Console.WriteLine("No arguments provided.");
        }
    }
}
```

C#-06

Arrays and Strings in C#

C# is a versatile programming language that provides robust support for managing arrays and strings. Arrays allow us to store collections of data, while strings handle sequences of characters. This article delves into single-dimensional and multi-dimensional arrays, array methods, and string manipulation techniques with detailed explanations and examples.

Arrays in C#

An array is a collection of elements of the same type, stored in contiguous memory locations. Arrays in C# are zero-indexed and can be single-dimensional, multi-dimensional, or jagged.

Single-Dimensional Arrays

Single-dimensional arrays store elements in a linear sequence.

Syntax:
csharp
Copy code
```
int[] numbers = new int[5]; // Declaring an array of size 5
```

```
numbers[0] = 10;            // Assigning values
numbers[1] = 20;
// ... add values for other indices
```

Example:

csharp

Copy code

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = { 10, 20, 30, 40, 50 };

        Console.WriteLine("Elements in the array:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

Multi-Dimensional Arrays

Multi-dimensional arrays are used to represent data in a matrix format.

Syntax:

csharp

Copy code

```
int[,] matrix = new int[2, 3]; // Declaring a 2x3 matrix
matrix[0, 0] = 1;              // Assigning values
matrix[1, 2] = 6;
```

Example:

csharp

Copy code

```
using System;

class Program
{
    static void Main()
    {
        int[,] matrix =
        {
            { 1, 2, 3 },
            { 4, 5, 6 }
        };

        Console.WriteLine("Matrix elements:");
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(1); j++)
            {
                Console.Write(matrix[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Array Methods

C# arrays come with several built-in methods to make operations easier.

Length

Returns the number of elements in an array.

csharp

Copy code

```
int[] arr = { 1, 2, 3, 4, 5 };
Console.WriteLine("Array Length: " + arr.Length);
```

Sort

Sorts the elements of an array.

csharp
Copy code

```
int[] arr = { 5, 3, 1, 4, 2 };
Array.Sort(arr);

Console.WriteLine("Sorted Array:");
foreach (int num in arr)
{
    Console.Write(num + " ");
}
```

Reverse

Reverses the order of elements in an array.

csharp
Copy code

```
Array.Reverse(arr);
Console.WriteLine("\nReversed Array:");
foreach (int num in arr)
{
    Console.Write(num + " ");
}
```

Find and FindAll

Finds elements matching a condition.

csharp
Copy code

```
int[] arr = { 1, 2, 3, 4, 5 };
int result = Array.Find(arr, x => x > 3);
Console.WriteLine("\nFirst element greater than 3: " + result);
```

Strings in C#

Strings in C# are immutable, meaning their content cannot be changed after creation. However, various methods allow us to manipulate string data efficiently.

String Manipulation Methods

Substring

Extracts a portion of the string.

csharp
Copy code

```
string text = "Hello, World!";
string sub = text.Substring(7, 5); // Start at index 7, take 5 characters
Console.WriteLine("Substring: " + sub);
```

Split

Splits a string into an array based on a delimiter.

csharp
Copy code

```
string sentence = "C#, Java, Python";
string[] languages = sentence.Split(", ");

Console.WriteLine("Languages:");
foreach (string lang in languages)
{
    Console.WriteLine(lang);
```

```
}
```

Replace

Replaces occurrences of a substring with another string.

csharp
Copy code
```
string text = "I love Java.";
string updatedText = text.Replace("Java", "C#");
Console.WriteLine(updatedText);
```

Trim

Removes leading and trailing whitespace.

csharp
Copy code
```
string rawText = "   Hello C#   ";
string trimmedText = rawText.Trim();
Console.WriteLine("Trimmed Text: '" + trimmedText + "'");
```

ToUpper and ToLower

Converts a string to upper or lower case.

csharp
Copy code
```
string name = "John Doe";
Console.WriteLine("Upper Case: " + name.ToUpper());
Console.WriteLine("Lower Case: " + name.ToLower());
```

Example: Combining String Manipulation

csharp
Copy code
```
using System;

class Program
{
    static void Main()
    {
        string sentence = "   Learning C# is fun and powerful.   ";
        string trimmed = sentence.Trim();
        string replaced = trimmed.Replace("fun", "exciting");
        string[] words = replaced.Split(' ');

        Console.WriteLine("Processed Sentence: " + replaced);
        Console.WriteLine("Words in the sentence:");
        foreach (string word in words)
        {
            Console.WriteLine(word);
        }
    }
}
```

Performance Tip: StringBuilder

For extensive string manipulation, use `StringBuilder` to optimize performance.

csharp
Copy code
```
using System.Text;

class Program
{
    static void Main()
    {
        StringBuilder sb = new StringBuilder("C# ");
        sb.Append("is ");
        sb.Append("awesome!");
```

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

C#-07

Comprehensive Guide to Exception Handling in C#

Exception handling is a fundamental concept in programming that helps developers anticipate and manage errors in a controlled manner. In C#, exception handling is achieved using the `try`, `catch`, and `finally` blocks. This mechanism provides a way to handle runtime errors without crashing the program. In this article, we'll explore exception handling in C#, the use of `try`, `catch`, `finally`, common exceptions, and how to throw exceptions.

1. Introduction to Exception Handling in C#

An **exception** is an error that occurs at runtime, disrupting the normal flow of execution in a program. When an exception is thrown, the normal flow of the program is interrupted, and control is transferred to a special block of code known as the **exception handler**. In C#, this is managed using the `try`, `catch`, and `finally` blocks.

The core idea behind exception handling is to handle errors gracefully without crashing the application. C# provides a structured way to catch exceptions and to clean up resources, ensuring that your program runs smoothly even in the presence of errors.

2. The Structure of Exception Handling: `try`, `catch`, `finally`

`try` Block

The `try` block is used to define a section of code that might throw an exception. Code within the `try` block is executed, and if an exception occurs, it is transferred to the appropriate `catch` block.

csharp
Copy code
```
try
{
    // Code that might throw an exception
    int result = 10 / 0;  // Division by zero
}
```

`catch` Block

The `catch` block is used to handle exceptions. If an exception occurs in the `try` block, the program will jump to the `catch` block. You can have multiple `catch` blocks to handle different types of exceptions.

csharp
Copy code
```
catch (DivideByZeroException ex)
{
    // Handle the specific exception
    Console.WriteLine("Error: Division by zero occurred.");
}
```

You can also catch a general exception if you are unsure about the type of exception that might occur.

csharp
Copy code
```
catch (Exception ex)
{
    // Handle any exception
    Console.WriteLine($"An unexpected error occurred: {ex.Message}");
}
```

`finally` Block

The `finally` block is optional, but it's highly recommended when you need to execute code regardless of whether an exception occurs or not. This block is typically used for cleanup operations, such as closing database connections or releasing resources.

csharp
Copy code
```
finally
{
    // Cleanup code, executed whether or not an exception occurred
    Console.WriteLine("This will always run.");
}
```

The `finally` block is executed after the `try` and `catch` blocks, even if an exception is not thrown. This guarantees that necessary cleanup tasks are always performed.

**

3. Common Exceptions in C#

C# provides a rich set of predefined exception types that you can use to handle common runtime errors. Below are some of the most commonly encountered exceptions:

1. `DivideByZeroException`

This exception is thrown when a division by zero operation is attempted.

csharp
Copy code
```
try
{
    int result = 10 / 0; // This will throw a DivideByZeroException
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: Cannot divide by zero.");
}
```

2. `NullReferenceException`

This exception occurs when you try to dereference a null object, such as calling a method on a null reference.

csharp
Copy code
```
try
{
    string name = null;
    int length = name.Length; // This will throw a NullReferenceException
}
catch (NullReferenceException ex)
{
    Console.WriteLine("Error: Attempted to access a null object.");
}
```

3. `IndexOutOfRangeException`

Thrown when an invalid index is accessed in an array or collection.

csharp
Copy code
```
try
{
    int[] arr = { 1, 2, 3 };
    int number = arr[5]; // This will throw an IndexOutOfRangeException
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Error: Index is out of bounds.");
}
```

4. `ArgumentException`

This exception is thrown when an argument passed to a method is not valid.

csharp
Copy code
```
try
{
    int.Parse("invalid"); // This will throw a FormatException
}
catch (ArgumentException ex)
{
    Console.WriteLine("Error: Invalid argument passed to method.");
}
```

5. `FileNotFoundException`

This exception occurs when an attempt to access a file fails because the file does not exist.

csharp

Copy code

```
try
{
    System.IO.File.ReadAllText("nonexistentFile.txt"); // This will throw a FileNotFoundException
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Error: The file was not found.");
}
```

6. `InvalidOperationException`

This exception is thrown when a method call is invalid for the current state of the object.

csharp
Copy code

```
try
{
    var list = new List<int>();
    list.RemoveAt(0); // This will throw an InvalidOperationException if the list is empty
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Error: Invalid operation.");
}
```

4. Throwing Exceptions in C#

In addition to catching exceptions, you may also want to throw exceptions manually in your code. This is typically done to signal that something unexpected or erroneous has occurred.

Syntax for Throwing an Exception

To throw an exception in C#, you can use the `throw` keyword followed by an instance of the exception class.

csharp
Copy code

```
throw new ArgumentException("Invalid input parameter.");
```

Example of Throwing an Exception

Here’s an example where we manually throw an exception when a method receives invalid input:

csharp
Copy code

```
public void ValidateAge(int age)
{
    if (age < 0)
    {
        throw new ArgumentException("Age cannot be negative.");
    }
    Console.WriteLine("Valid age.");
}

try
{
    ValidateAge(-5); // This will throw an ArgumentException
}
catch (ArgumentException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

In this example, the `ValidateAge` method throws an `ArgumentException` if the `age` is less than zero. The `catch` block handles the exception and displays the error message.

Custom Exception Classes

Sometimes, you may need to define your own exception types to represent specific errors in your application. You can create custom exceptions by inheriting from the `Exception` class.

csharp
Copy code

```
public class AgeNotValidException : Exception
```

```
{
    public AgeNotValidException(string message) : base(message) { }
}
```

Now you can throw this custom exception in your code:

csharp
Copy code
```
public void ValidateAge(int age)
{
    if (age < 0)
    {
        throw new AgeNotValidException("Age cannot be negative.");
    }
    Console.WriteLine("Valid age.");
}
```

5. Best Practices for Exception Handling

While exception handling is powerful, it’s essential to use it correctly to avoid issues like performance degradation and difficult-to-maintain code. Here are some best practices for exception handling in C#:

1. Catch Specific Exceptions

Rather than catching a generic `Exception`, try to catch specific exceptions. This allows you to handle different error types appropriately.

csharp
Copy code
```
try
{
    // Code that might throw an exception
}
catch (DivideByZeroException ex)
{
    // Handle division by zero error
}
catch (FileNotFoundException ex)
{
    // Handle file not found error
}
catch (Exception ex)
{
    // Handle any other unexpected error
}
```

2. Don’t Use Exceptions for Control Flow

Exceptions should not be used as a regular part of the program flow. They should be used for exceptional conditions that cannot be handled with regular logic.

csharp
Copy code
```
// Bad practice - using exceptions for control flow
try
{
    int index = 10;
    Console.WriteLine(arr[index]);
}
catch (IndexOutOfRangeException)
{
    // Handle as a special case rather than using exception for control flow
}
```

3. Avoid Empty Catch Blocks

An empty `catch` block might hide errors and make debugging harder. Always log or handle the exception meaningfully.

csharp
Copy code
```
catch (Exception ex)
{
    // Bad practice: Swallowing exception without any action
}
```

Instead, log the error or handle it appropriately:

csharp
Copy code
```
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
    // Log the error to a file or monitoring system
}
```

4. Use `finally` for Cleanup

Always use the `finally` block for releasing resources (such as database connections, file handles, etc.) that need to be cleaned up regardless of whether an exception occurred.

csharp
Copy code
```
finally
{
    // Cleanup resources, e.g., closing file streams or database connections
}
```

6. Conclusion

Exception handling is a vital aspect of writing robust and maintainable C# programs. By using `try`, `catch`, and `finally` blocks effectively, you can gracefully handle errors, ensuring that your application continues running smoothly even in the face of unexpected events. Additionally, understanding common exceptions and how to throw exceptions manually will enable you to write better, more predictable code. Always keep best practices in mind to ensure that your exception handling enhances, rather than hinders, your application.

C#-08

A Comprehensive Guide to Basic Input/Output in C#

Input and output (I/O) are fundamental aspects of any application. In C#, handling input from the user and outputting data is crucial for building interactive console applications. This article will cover basic input and output operations, including how to read and write to the console using `Console.ReadLine()` and `Console.WriteLine()`, and how to work with files using basic file operations like `StreamReader` and `StreamWriter`.

Console Input and Output

C# provides a set of methods for interacting with the user via the console. The two primary methods for console I/O are `Console.ReadLine()` and `Console.WriteLine()`. These methods allow for reading from and writing to the console, respectively.

`Console.ReadLine()`

The `Console.ReadLine()` method reads a line of text input by the user from the console. It returns the input as a string. If the user presses Enter without typing anything, `Console.ReadLine()` will return an empty string.

Example of `Console.ReadLine()`:

csharp
Copy code
```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Please enter your name:");
        string name = Console.ReadLine();
        Console.WriteLine("Hello, " + name + "!");
    }
}
```

Output:

yaml
Copy code
```
Please enter your name:
John
Hello, John!
```

In the above example, the program prompts the user to enter their name. The `Console.ReadLine()` method captures the input and stores it in the `name` variable, which is then used to display a personalized greeting.

`Console.WriteLine()`

The `Console.WriteLine()` method writes a line of text to the console. It automatically appends a newline character (`\n`) after the text, meaning each subsequent output is printed on a new line.

Example of `Console.WriteLine()`:

csharp
Copy code

```csharp
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("This is a message.");
        Console.WriteLine("Another message.");
    }
}
```

Output:

csharp
Copy code

```
This is a message.
Another message.
```

In the above example, both strings are printed on separate lines. `Console.WriteLine()` is the most commonly used method for displaying messages in C# console applications.

You can also use `Console.Write()` if you want to print text without adding a new line after it. `Console.Write()` behaves similarly to `Console.WriteLine()`, but without the newline character at the end.

Combining `Console.ReadLine()` and `Console.WriteLine()`

You can combine both methods to create interactive applications that prompt users for input and then display results based on that input.

Example:

csharp
Copy code

```csharp
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int num1 = int.Parse(Console.ReadLine()); // Read user input and convert it to an integer
        Console.Write("Enter another number: ");
        int num2 = int.Parse(Console.ReadLine()); // Read another number
        int sum = num1 + num2;
        Console.WriteLine("The sum of " + num1 + " and " + num2 + " is: " + sum);
    }
}
```

Output:

yaml
Copy code

```
Enter a number: 10
Enter another number: 20
The sum of 10 and 20 is: 30
```

In this example, `Console.ReadLine()` is used to read user input as strings, and `int.Parse()` converts those strings into integers for calculation. The result is then printed using `Console.WriteLine()`.

File Operations: `StreamReader` and `StreamWriter`

In addition to basic console I/O, C# also provides ways to read from and write to files. The `StreamReader` and `StreamWriter` classes are commonly used for reading from and writing to text files. These classes are part of the `System.IO` namespace and provide an easy way to handle file operations in C#.

Writing to a File with `StreamWriter`

The `StreamWriter` class is used to write text to a file. If the file does not already exist, `StreamWriter` will create it. If the file exists, it will overwrite the existing content unless the `append` parameter is set to `true`.

Example of Writing to a File with `StreamWriter`:

csharp
Copy code

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "example.txt";
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            writer.WriteLine("Hello, this is a test file.");
            writer.WriteLine("This file was created using StreamWriter.");
        }

        Console.WriteLine("File written successfully.");
    }
}
```

In this example, `StreamWriter` is used to create and write to a text file named `example.txt`. The `WriteLine()` method writes text to the file, and the `using` block ensures that the file is properly closed after writing.

Explanation:

- The `StreamWriter` is wrapped in a `using` statement to automatically close the file and release resources once the block is executed.
- The `WriteLine()` method writes a line of text followed by a newline character. You can also use `Write()` to write text without a newline.

Reading from a File with `StreamReader`

The `StreamReader` class is used to read text from a file. It allows you to read one line at a time or the entire file in a more efficient way.

Example of Reading from a File with `StreamReader`:

csharp
Copy code

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "example.txt";
        if (File.Exists(filePath))
        {
            using (StreamReader reader = new StreamReader(filePath))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
        }
        else
        {
            Console.WriteLine("The file does not exist.");
        }
    }
}
```

Explanation:

- The `StreamReader` reads the file line by line using the `ReadLine()` method, which returns each line as a string. The loop continues until there are no more lines to read (when `ReadLine()` returns `null`).
- The `File.Exists()` method checks if the file exists before attempting to read from it. If the file is missing, an appropriate message is displayed.

Handling File Not Found Exceptions

It's essential to handle exceptions when dealing with file operations to avoid runtime errors. The most common exception is `FileNotFoundException`, which occurs when the program attempts to read from a file that doesn't exist.

Example of Exception Handling in File I/O:

csharp

Copy code

```csharp
using System;
using System.IO;

class Program
{
    static void Main()
    {
        try
        {
            string filePath = "nonexistentfile.txt";
            using (StreamReader reader = new StreamReader(filePath))
            {
                string content = reader.ReadToEnd();
                Console.WriteLine(content);
            }
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine("Error: The file could not be found.");
            Console.WriteLine(ex.Message);
        }
    }
}
```

In this example, a `FileNotFoundException` is caught if the specified file doesn't exist, and an error message is displayed.

Conclusion

In C#, basic input/output operations are essential for building interactive console applications. The `Console.ReadLine()` and `Console.WriteLine()` methods provide an easy way to handle user input and output. For working with files, `StreamReader` and `StreamWriter` are convenient tools to read from and write to text files. By understanding and mastering these basic I/O operations, you can easily handle console and file interactions in your C# applications.

Here’s an outline of Angular topics categorized into **Basic**, **Intermediate**, and **Advanced** levels:

Basic Topics

These topics focus on fundamental concepts and getting started with Angular.

1. **Introduction to Angular**
   - What is Angular?
   - Angular vs. AngularJS
   - Features of Angular
   - Setting up the Development Environment
2. **Angular CLI**
   - Installing Angular CLI
   - Creating a new Angular project
   - CLI commands for development and testing
3. **Components**
   - What are components?
   - Creating and using components
   - Component template, styles, and metadata
4. **Templates and Data Binding**
   - String Interpolation
   - Property Binding
   - Event Binding
   - Two-way Data Binding (`[(ngModel)]`)
5. **Directives**
   - Structural Directives: `*ngIf`, `*ngFor`
   - Attribute Directives: `ngClass`, `ngStyle`
6. **Modules**
   - NgModule structure
   - Root module (AppModule)
   - Feature modules
7. **Dependency Injection (DI) and Services**

    - What is DI?
    - Creating and using services
    - Injecting services into components
8. **Routing**
    - Setting up routing in Angular
    - `RouterModule` and Routes
    - Navigation between routes
9. **Forms**
    - Template-driven forms
    - Form validation (required, pattern, etc.)
10. **Pipes**
    - Built-in pipes (`date`, `uppercase`, `currency`, etc.)
    - Creating custom pipes

Intermediate Topics

These topics delve deeper into Angular features and best practices.

1. **Advanced Components**
    - Component lifecycle hooks
    - Input and Output properties
    - ViewChild and ContentChild decorators
2. **Reactive Forms**
    - Introduction to Reactive Forms
    - FormBuilder and FormGroup
    - Custom validators
3. **Advanced Routing**
    - Lazy loading modules
    - Route guards (`CanActivate`, `CanDeactivate`, etc.)
    - Passing data via routes (`ActivatedRoute`)
4. **HTTP Client**
    - Making HTTP requests with `HttpClient`
    - Handling responses and errors
    - Interceptors
5. **Angular Animations**
    - Introduction to Angular Animations
    - Trigger and State
    - Animating transitions
6. **Component Communication**
    - Parent-child communication
    - Event emitters
    - Shared services
7. **Change Detection**
    - How Angular detects changes
    - OnPush strategy
    - Zones and `NgZone`
8. **Angular Material**
    - Installing Angular Material
    - Using Material components (e.g., Buttons, Dialogs, Forms)
9. **Observables and RxJS**
    - Introduction to Observables
    - Operators (`map`, `filter`, `mergeMap`, etc.)
    - Subjects and BehaviorSubjects

Advanced Topics

These topics focus on optimizing, securing, and architecting complex Angular applications.

1. **State Management**
    - Introduction to state management
    - Using NgRx for state management
    - Store, Actions, Reducers, and Effects
2. **Dynamic Components**
    - Creating and loading dynamic components
    - Using `ComponentFactoryResolver`
3. **Unit Testing**
    - Testing components, services, and directives
    - Writing test cases using Jasmine and Karma
4. **Performance Optimization**
    - Lazy loading and preloading strategies
    - Optimizing change detection
    - Using Ahead-of-Time (AOT) compilation
    - Tree-shaking and bundle optimization
5. **Angular Universal**
    - Server-side rendering (SSR) with Angular Universal
    - Benefits and use cases
    - Setting up SSR

6. **Progressive Web Apps (PWA)**
    - What is a PWA?
    - Adding PWA features to Angular applications
    - Service workers
7. **Advanced Dependency Injection**
    - Hierarchical injectors
    - Multi-providers
    - Injection tokens
8. **Custom Libraries**
    - Creating reusable Angular libraries
    - Publishing libraries to npm
9. **Security**
    - Protecting applications from XSS and CSRF
    - Securing routes
    - Using Content Security Policy (CSP)
10. **Internationalization (i18n)**
    - Adding language translations
    - Using Angular i18n tools
    - Formatting dates and numbers for localization
11. **Monorepos**
    - Managing Angular projects in a monorepo
    - Using Nx or Lerna for monorepo management
12. **Micro Frontends**
    - What are micro frontends?
    - Implementing micro frontends with Angular

Introduction to Angular -01

Angular is a popular **TypeScript-based open-source front-end framework** developed and maintained by Google. It is widely used to build dynamic, single-page web applications (SPAs) with a focus on modularity, scalability, and maintainability. By employing declarative templates, dependency injection, and a component-based architecture, Angular simplifies the development of robust web applications.

What is Angular?

Angular is the successor to AngularJS and represents a complete rewrite of the original framework. It addresses the challenges posed by AngularJS, including performance bottlenecks and the lack of modularity, while introducing new concepts such as components and reactive programming with RxJS.

Key Features of Angular:

1. **Component-based Architecture:** Simplifies UI development by breaking it into reusable components.
2. **Two-way Data Binding:** Syncs the UI and business logic seamlessly.
3. **Dependency Injection:** Provides services to various components efficiently.
4. **Directives:** Enable the creation of custom HTML elements and dynamic templates.
5. **Routing:** Supports navigation between different views or components.
6. **Reactive Programming:** Uses RxJS for managing asynchronous data streams.

Angular vs. AngularJS

Feature	AngularJS	Angular
Language	JavaScript	TypeScript
Architecture	MVC (Model-View-Controller)	Component-based
Data Binding	Two-way with `$scope`	Two-way with `ngModel`
Performance	Comparatively slower	Faster, with Ahead-of-Time (AOT) compilation
Mobile Support	Limited	Robust

Angular was designed to address the limitations of AngularJS and support modern web development needs.

Features of Angular

Angular's rich set of features includes:

1. **Cross-Platform Development:**
    - Build web, mobile, and desktop applications using the same framework.
2. **High Performance:**

    - Faster rendering with AOT compilation and optimized change detection.
3. **Modular Development:**
    - Organize code into feature modules for better maintainability.
4. **Tooling and Ecosystem:**
    - Seamless integration with tools like Angular CLI, Angular Material, and RxJS.
5. **Testability:**
    - Built-in support for unit testing and end-to-end testing.

Setting up the Development Environment

To begin developing with Angular, follow these steps:

1. **Install Node.js:**
    - Download and install Node.js, which includes npm (Node Package Manager).
2. **Install Angular CLI:**

Use the command:
bash
Copy code
```
npm install -g @angular/cli
```

    -
3. **Create a New Angular Project:**

Run:
bash
Copy code
```
ng new my-angular-app
cd my-angular-app
```

    -
4. **Run the Development Server:**

Start the server:
bash
Copy code
```
ng serve
```

    -
    - Access the application at `http://localhost:4200`.

Angular CLI: An Essential Tool for Angular Development

The Angular CLI (Command Line Interface) is a powerful tool that streamlines the Angular development process. It automates common tasks such as project setup, component creation, and code generation, making it an indispensable part of the Angular ecosystem.

Installing Angular CLI

Before using Angular CLI, you need to install it globally on your system. Follow these steps:

1. Prerequisites:

○ Ensure that Node.js and npm are installed on your system.

○ Download Node.js from nodejs.org, which includes npm.

2. Install Angular CLI:

Open a terminal and run:

bash

Copy code

npm install -g @angular/cli

○

Verify the installation:

bash

Copy code

ng version

○

○ This command displays the installed Angular CLI version along with its dependencies.

Creating a New Angular Project

Once Angular CLI is installed, you can create a new Angular project:

1. Create a Project:

Run the following command:

bash

Copy code

ng new my-angular-app

○

○ Replace my-angular-app with your desired project name.

2. Interactive Prompts:

○ The CLI may ask for additional configuration options, such as:

■ Add Angular routing? (Yes or No)

■ CSS preprocessor? (Choose from CSS, SCSS, SASS, or LESS)

3. Navigate to the Project Folder:

Change directory into the newly created project:

bash

Copy code

cd my-angular-app

○

4. Run the Development Server:

Start the Angular development server:

bash

Copy code

ng serve

○

○ Open a browser and go to http://localhost:4200 to see the default Angular application.

CLI Commands for Development and Testing

The Angular CLI includes a suite of commands for development and testing. Here are some commonly used commands:02

1. Development Commands

Generate Components, Services, and More:

bash

Copy code

ng generate component component-name

ng generate service service-name

ng generate module module-name

●

Run the Development Server:

bash

Copy code

ng serve

●

○ Options:

■ --port <number>: Specify a different port.

- --open: Automatically opens the application in the browser.

Add Features or Libraries:

bash

Copy code

ng add <library-name>

-

Example:

bash

Copy code

ng add @angular/material

-

2. Build and Deployment

Build for Production:

bash

Copy code

ng build --prod

-
  - Produces optimized files for deployment.

Serve Production Build Locally:

bash

Copy code

npm install -g http-server

http-server ./dist/my-angular-app

-

3. Testing Commands

Run Unit Tests:

bash

Copy code

ng test

-
  - Launches the Karma test runner.

Run End-to-End Tests:

bash

Copy code

ng e2e

-
  - Executes tests using Protractor.

4. Linting and Formatting

Lint Code for Issues:

bash

Copy code

ng lint

-

Format Code (if a formatter is configured):

bash

Copy code

npm run format

●

5. Other Useful Commands

Check Angular Version:

bash

Copy code

ng version

●

Update Angular:

bash

Copy code

ng update @angular/cli @angular/core

●

Help with Commands:

bash

Copy code

ng help

●

Advantages of Angular CLI

- Efficiency: Automates repetitive tasks, saving time and effort.
- Consistency: Ensures consistent project structure and coding standards.
- Ease of Use: Simple commands for building, serving, and testing applications.
- Customizability: Offers options to tailor projects to specific requirements.

Conclusion

Angular CLI is a cornerstone for Angular developers, offering tools to simplify and enhance productivity during application development. Whether you're starting a new project or managing an existing one, mastering the CLI commands is crucial for effective Angular development.

Comprehensive Guide to Angular Components 03

What Are Components in Angular?

Components are the building blocks of Angular applications. Each component controls a part of the screen, known as a view, and is responsible for rendering the user interface and handling logic associated with that view.

A component in Angular consists of three key parts:

1. Template: Defines the HTML structure of the component.
2. Styles: Specifies the CSS styles that apply to the template.
3. Class: Contains the logic and data associated with the component.

Key Characteristics of Components:

- Encapsulated: Each component is self-contained, managing its own view and behavior.
- Reusable: Components can be reused across the application.
- Declarative: Angular uses declarative HTML templates bound to data.

Creating and Using Components

1. Creating a Component Using Angular CLI

Angular CLI simplifies the creation of components. Use the following command:

bash

Copy code

ng generate component component-name

or its shorthand:

bash

Copy code

ng g c component-name

This command:

- Creates a new folder with the component's name.
- Generates the following files:
  - component-name.component.ts: Component logic and metadata.
  - component-name.component.html: HTML template.
  - component-name.component.css: CSS styles (or your chosen preprocessor).
  - component-name.component.spec.ts: Unit test file.
- Updates the module file to declare the new component.

2. Using a Component

To use a component in another component’s template:

Include the selector of the component in the HTML.

html

Copy code

<app-component-name></app-component-name>

1.

2. Ensure the component is declared in the same Angular module or imported if declared elsewhere.

Component Template, Styles, and Metadata

1. Component Metadata

Metadata is defined using the @Component decorator in the TypeScript file:

typescript

Copy code

```
import { Component } from '@angular/core';


@Component({
  selector: 'app-component-name',
  templateUrl: './component-name.component.html',
  styleUrls: ['./component-name.component.css']
})
export class ComponentNameComponent {
  // Component logic goes here
}
```

- Selector: Specifies the HTML tag used to include the component.

- TemplateUrl: Points to the HTML file.
- StyleUrls: Points to the CSS or styling file.

2. Component Template

The template defines the structure and layout of the view. It can:

- Be defined in an external HTML file (templateUrl) or inline using the template property.
- Include Angular directives, bindings, and event handlers.

Example:

html

Copy code

```
<div>
  <h1>{{ title }}</h1>
  <button (click)="onClick()">Click Me</button>
</div>
```

3. Component Styles

Styles allow you to define how the component's template is rendered visually. They can be:

- Scoped to the component, preventing interference with styles in other components.
- Declared in external CSS files or inline.

Component Lifecycle

Angular components have a well-defined lifecycle. The lifecycle includes hooks like:

1. ngOnInit: Called once after the component is initialized.
2. ngOnChanges: Responds to input property changes.
3. ngOnDestroy: Cleans up before the component is destroyed.

Best Practices for Using Components

- Break down large applications into smaller, reusable components.
- Use clear and meaningful selectors.
- Avoid placing logic in templates; keep it in the component class.
- Utilize lifecycle hooks to manage resources and performance.

Comprehensive Guide to Angular Templates and Data Binding 04

Introduction to Templates and Data Binding

In Angular, templates are the HTML structures that define how the view (UI) is rendered. Templates can dynamically update based on changes in the component's data, thanks to **data binding**. Data binding is a mechanism that connects the template to the component's logic and allows data to flow between them.

Types of Data Binding

1. **String Interpolation**: Binding component data to the template.
2. **Property Binding**: Dynamically setting properties of HTML elements or directives.
3. **Event Binding**: Capturing and responding to user actions.
4. **Two-Way Binding**: Synchronizing data between the component and the view.

String Interpolation

String interpolation allows you to embed component properties or expressions into the template using double curly braces (`{{ }}`).

Example

In the component class:

typescript
Copy code
```
export class AppComponent {
  title: string = 'Angular Basics';
}
```

In the template:

html
Copy code
```
<h1>{{ title }}</h1>
```

Result: The `h1` element will display `Angular Basics`.

Key Points

- Can evaluate expressions within the braces, e.g., `{{ 2 + 2 }}`.
- Only reads data; it cannot modify component properties.

Property Binding

Property binding is used to set properties of an HTML element or directive dynamically. It uses square brackets `[ ]` around the property name.

Example
html
Copy code
```
<img [src]="imageUrl" [alt]="imageDescription">
```

In the component class:

typescript
Copy code
```
imageUrl: string = 'https://example.com/image.jpg';
imageDescription: string = 'An example image';
```

Result: The `src` and `alt` attributes of the `img` tag will be dynamically set.

Key Points

- One-way binding: From component to template.
- Cannot use property binding for attributes like `class` and `style` (use `[ngClass]` or `[ngStyle]` instead).

Event Binding

Event binding is used to listen for and handle user events such as clicks, key presses, or mouse movements. It uses parentheses `( )` around the event name.

Example
html
Copy code
```
<button (click)="onClick()">Click Me</button>
```

In the component class:

typescript
Copy code
```
onClick() {
  alert('Button clicked!');
}
```

Result: Clicking the button triggers the `onClick` method and displays an alert.

Key Points

- One-way binding: From template to component.
- Can pass event data to the handler, e.g., `(input)="onInput($event)"`.

Two-Way Data Binding

Two-way data binding allows synchronization of data between the component and the template. It uses the syntax `[( )]`, a combination of property and event binding.

Example

html
Copy code

```
<input [(ngModel)]="name">
<p>Hello, {{ name }}!</p>
```

In the component class:

typescript
Copy code

```
name: string = 'Angular';
```

Result: Typing into the input box updates the `name` variable, and changes in `name` update the input box.

Key Points

- Requires the `FormsModule` to be imported into the application module.
- Ideal for form controls like text inputs.

Advantages of Data Binding

- Simplifies dynamic UI updates.
- Reduces boilerplate code for DOM manipulation.
- Enables better separation of concerns between the view and logic.

Comprehensive Guide to Angular Directives - 05

Introduction to Angular Directives

In Angular, directives are instructions that help you manipulate the DOM. They enhance the functionality of your templates by providing a way to apply behavior to elements dynamically. Angular offers three types of directives:

1. Structural Directives: Modify the structure of the DOM (e.g., adding or removing elements).
2. Attribute Directives: Change the appearance or behavior of an element.
3. Custom Directives: User-defined directives for specific needs.

This guide focuses on structural directives like *ngIf and *ngFor and attribute directives like ngClass and ngStyle.

Structural Directives

Structural directives change the structure of the DOM by adding or removing elements. These directives are prefixed with *.

1. *ngIf

The *ngIf directive conditionally includes or excludes elements from the DOM based on an expression's truthiness.

Example:

html

Copy code

<div *ngIf="isLoggedIn">Welcome, User!

In the component:

typescript

Copy code

isLoggedIn: boolean = true;

Result: The will appear only when isLoggedIn is true.

Key Points:

- Elements managed by *ngIf are completely removed from the DOM if the condition is false.

- It can be combined with else using <ng-template> for alternate views.

---

2. *ngFor

The *ngFor directive creates a template for each item in a collection, allowing you to loop through arrays.

Example:

html

Copy code

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

In the component:

typescript

Copy code

```
items: string[] = ['Apple', 'Banana', 'Cherry'];
```

Result: The list displays:

- Apple
- Banana
- Cherry

Key Points:

- Syntax: *ngFor="let variable of collection".
- Provides local variables like index (current item's index) and first, last, even, odd.

Attribute Directives

Attribute directives change the behavior or appearance of an existing DOM element.

1. ngClass

The ngClass directive dynamically applies or removes a set of CSS classes based on an expression.

Example:

html

Copy code

```
<div [ngClass]="{'active': isActive, 'highlight': isHighlighted}">
  Dynamic Classes
</div>
```

In the component:

typescript

Copy code

```
isActive: boolean = true;
isHighlighted: boolean = false;
```

Result: The active class is applied, while highlight is not.

Key Points:

- Accepts strings, arrays, or objects.
- Can conditionally apply multiple classes.

---

2. ngStyle

The ngStyle directive dynamically sets inline styles on an element.

Example:

html

Copy code

```
<div [ngStyle]="{'color': textColor, 'font-size': fontSize}">
  Dynamic Styles
</div>
```

In the component:

typescript

Copy code

```
textColor: string = 'blue';
fontSize: string = '20px';
```

Result: The text appears in blue with a font size of 20px.

Key Points:

- Accepts an object where keys are style properties, and values are their respective values.
- Styles are applied inline.

Combining Directives

You can combine structural and attribute directives to create more dynamic and powerful templates.

Example:

html

Copy code

```
<ul>
  <li *ngFor="let item of items" [ngClass]="{'selected': selectedItem === item}">
    {{ item }}
  </li>
</ul>
```

This code loops through items and highlights the selected item using ngClass.

Advantages of Directives

- Simplify dynamic DOM manipulation.
- Encourage reusable and declarative code.
- Minimize direct DOM handling in components.

Comprehensive Guide to Angular Modules - 06

Introduction to Angular Modules

In Angular, a **module** is a mechanism to group components, directives, pipes, and services that are related to a specific application feature or functionality. This modular architecture promotes better organization, reusability, and maintainability of code in large applications.

Modules are defined using the `@NgModule` decorator, which provides metadata about the module.

NgModule Structure

An Angular module is defined in a TypeScript file using the `@NgModule` decorator. The structure typically includes:

- **Declarations**: Declares the components, directives, and pipes belonging to this module.
- **Imports**: Specifies other modules that are required for this module to function.
- **Exports**: Specifies the components, directives, or pipes that should be available to other modules.
- **Providers**: Registers services that this module uses.
- **Bootstrap**: Lists the root component(s) to bootstrap the application (only in the root module).

Example of a Module:

typescript
Copy code
```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';

@NgModule({
  declarations: [ // Declare components, directives, and pipes
    AppComponent,
    HeaderComponent,
    FooterComponent
  ],
  imports: [ // Import required modules
    BrowserModule
  ],
  providers: [], // Register services
  bootstrap: [AppComponent] // Root component to bootstrap
})
export class AppModule { }
```

Root Module (AppModule)

The **root module**, typically named `AppModule`, is the starting point of any Angular application. It is responsible for bootstrapping the application and usually imports core modules like `BrowserModule`.

Example:

typescript
Copy code
```typescript
@NgModule({
  declarations: [
    AppComponent // Root component
  ],
  imports: [
    BrowserModule // Core module required for browser-based applications
  ],
  bootstrap: [AppComponent] // Entry point for the app
})
export class AppModule { }
```

Key Points:

- Every Angular app must have at least one root module.
- It is declared in the `main.ts` file, which bootstraps the module:

typescript

Copy code

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Feature Modules

Feature modules are used to encapsulate a specific feature or functionality within an application. They help in keeping the application modular and easy to scale.

Why Use Feature Modules?

- To organize large applications.
- To allow lazy loading of specific features for performance optimization.
- To reuse feature-specific components across the app.

Example of a Feature Module:

typescript
Copy code

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductDetailComponent } from './product-detail/product-detail.component';

@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent
  ],
  imports: [
    CommonModule // Provides Angular directives like *ngIf and *ngFor
  ],
  exports: [ // Makes these components available to other modules
    ProductListComponent
  ]
})
export class ProductModule { }
```

Using the Feature Module in AppModule:

typescript
Copy code

```
import { ProductModule } from './product/product.module';

@NgModule({
  imports: [
    BrowserModule,
    ProductModule // Import the feature module
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Shared Modules

A **shared module** is used to declare and export common components, directives, and pipes that are used across multiple feature modules. This avoids code duplication.

Example:

typescript
Copy code

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ButtonComponent } from './button/button.component';

@NgModule({
  declarations: [
    ButtonComponent
  ],
```

```
  imports: [
    CommonModule
  ],
  exports: [
    ButtonComponent // Make it available for other modules
  ]
})
export class SharedModule { }
```

Lazy Loading with Feature Modules

Lazy loading is a technique to load feature modules only when they are needed, reducing the initial load time of the application. This is achieved using Angular's routing module.

Example:

typescript
Copy code

```
const routes: Routes = [
  { path: 'products', loadChildren: () => import('./product/product.module').then(m => m.ProductModule) }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Comprehensive Guide to Dependency Injection (DI) and Services in Angular 07

Introduction to Dependency Injection (DI)

Dependency Injection (DI) is a design pattern that allows a class (such as a component or service) to receive its dependencies from an external source rather than creating them itself. In Angular, DI is a core concept that facilitates the development of modular, testable, and maintainable applications. It allows the components and services to be loosely coupled and easily reusable.

With DI, Angular can manage the creation and lifecycle of services and inject them into components, directives, or other services that need them. This reduces the need for the components to be responsible for creating and managing these dependencies, making the application more modular.

How Dependency Injection Works in Angular

In Angular, DI is powered by the **Angular Injector**, which is responsible for creating service instances and providing them to the classes that request them. Services are typically singleton objects that are created once and shared across components, ensuring efficiency and better resource management.

Key Concepts:

- **Injector**: A container that holds and manages services.
- **Providers**: Configuration objects used to define how Angular should create and inject a service.
- **Tokens**: Identifiers used by the injector to locate the service.

Creating and Using Services

In Angular, services are used to provide business logic, data storage, or any functionality that needs to be shared across multiple components. Angular services are typically classes that use the `@Injectable` decorator, which allows them to be injected into components or other services.

Steps to Create a Service:
Generate a Service: Use Angular CLI to generate a service.
bash
Copy code

```
ng generate service my-service
```

1.
2. **Inject the Service into a Component**: Once the service is created, it can be injected into a component to provide its functionality.

Service Example:

Here's an example of creating a service that fetches data from an API:

typescript
Copy code

```
import { Injectable } from '@angular/core';
```

```typescript
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root', // Makes the service available globally
})
export class DataService {
  private apiUrl = 'https://api.example.com/data';

  constructor(private http: HttpClient) {}

  fetchData(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

In the above code:

- The `@Injectable` decorator makes the service injectable.
- The `providedIn: 'root'` syntax means the service is available throughout the application, as it is provided in the root injector.

Injecting Services into Components

Once a service is created, it can be injected into a component using the constructor. Here's how you can inject the `DataService` into a component:

typescript
Copy code

```typescript
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
})
export class MyComponent implements OnInit {
  data: any;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.fetchData().subscribe(response => {
      this.data = response;
    });
  }
}
```

In this example:

- The `DataService` is injected into the component through the constructor.
- The `ngOnInit()` lifecycle hook is used to fetch data when the component is initialized.

Scope of Services:

Services can have different scopes based on how they are provided:

- **Root Scope**: If a service is provided in the root injector (`providedIn: 'root'`), it's a singleton and can be accessed globally across the application.
- **Module Scope**: A service can be provided at the module level if it's registered in the `providers` array of a module.
- **Component Scope**: Services can also be provided at the component level, making them available only to that component and its descendants.

Providers and Tokens

Providers: Angular uses providers to define how to create instances of a service. The most common provider is `useClass`, which tells Angular to instantiate the service using a particular class.
Example:
typescript
Copy code

```typescript
providers: [{ provide: DataService, useClass: MockDataService }]
```

- This configuration allows Angular to provide a mock service instead of the real one for testing purposes.
- **Tokens**: Tokens are used to identify services in DI. By default, Angular uses class types as tokens, but you can use strings, injection tokens, or other symbols for advanced DI scenarios.

Comprehensive Guide to Angular Routing - 08

Introduction to Routing in Angular

Routing in Angular is a mechanism that allows navigation between different views or pages within a single-page application (SPA). It enables dynamic loading of content without the need to reload the entire page. This is accomplished through Angular's **RouterModule**, which provides powerful tools for routing and navigation.

In a typical Angular application, the Router allows developers to manage URLs and map them to components, effectively enabling different views of the app to be displayed when the user navigates between them. The routing system is crucial in organizing large applications and providing a seamless user experience.

Setting Up Routing in Angular

Setting up routing in Angular involves configuring routes and linking them to components. Let's walk through the essential steps to set up routing in an Angular application.

1. Importing the RouterModule

The first step in setting up routing is to import **RouterModule** from `@angular/router` in the application module.

Example:

typescript
Copy code

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

In the above code:

- **RouterModule** and **Routes** are imported.
- The `RouterModule.forRoot(routes)` method is used to configure routes.
- `routes` defines the route configuration where each route is mapped to a specific component.

2. Configuring Routes

Routes are configured using an array of route objects. Each route object contains:

- **path**: The URL or path associated with the route.
- **component**: The component to display when the path is matched.

For example, if the user navigates to `/about`, the `AboutComponent` will be displayed.

3. Adding RouterOutlet

To display the routed components, a **<router-outlet>** directive is added to the `AppComponent` template. This placeholder will be replaced by the appropriate component based on the current route.

Example:

html
Copy code

```
<div>
  <h1>Welcome to Angular Routing!</h1>
  <nav>
    <a routerLink="/">Home</a>
    <a routerLink="/about">About</a>
  </nav>
  <router-outlet></router-outlet>
</div>
```

The **<router-outlet>** tag acts as a placeholder where the content of routed components will be injected.

RouterModule and Routes

The **RouterModule** is the primary module used to manage routing in Angular. It provides functionalities such as defining routes, navigation, and lifecycle hooks related to route navigation.

RouterModule

- **forRoot()**: This method configures the root routes of the application. It should be used in the root module only.
- **forChild()**: Used to configure routes for feature modules. It is used in child modules to set up routes in a modular way.

Routes

Routes are a collection of route objects, each representing a specific path and its corresponding component. A route object can contain more advanced configuration options, such as route guards, lazy loading, and parameterized paths.

Example of a more advanced route configuration:

typescript
Copy code

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'user/:id', component: UserComponent }
];
```

In this example:

- The `:id` is a route parameter that allows passing dynamic values to the component.

Navigation Between Routes

Once routes are configured, navigating between them is simple. Angular provides a set of directives and services to handle routing and navigation:

1. **routerLink**: This directive is used in templates to navigate between routes.
2. **routerLinkActive**: This directive is used to add a class to the active link.
3. **Router Service**: The `Router` service provides methods for navigating programmatically.

Using routerLink in Templates

The `routerLink` directive is used to bind a route path to an HTML element (typically an anchor tag):

Example:

html
Copy code

```
<a routerLink="/home">Go to Home</a>
<a routerLink="/about">Go to About</a>
```

Using routerLinkActive

The `routerLinkActive` directive can be used to apply a CSS class to the active link. This is useful for styling the currently active route:

html
Copy code

```
<a routerLink="/home" routerLinkActive="active-link">Home</a>
<a routerLink="/about" routerLinkActive="active-link">About</a>
```

In this example, the class `active-link` will be added to the link that matches the current route.

Programmatic Navigation with the Router Service

Angular's **Router** service allows navigation to be triggered programmatically. This can be useful in scenarios like after form submission or conditional navigation.

Example:

typescript
Copy code

```typescript
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html'
})
export class LoginComponent {

  constructor(private router: Router) {}

  onSubmit() {
    // Navigate to the dashboard after successful login
    this.router.navigate(['/dashboard']);
  }
}
```

In the above code, the `this.router.navigate(['/dashboard'])` method programmatically navigates to the dashboard route.

Comprehensive Guide to Angular Forms - 09

Introduction to Angular Forms

In Angular, forms are a key component for collecting user input. Angular provides two types of forms for handling user input: **Template-driven forms** and **Reactive forms**. In this article, we will focus on **Template-driven forms**, exploring how to create them, apply form validation, and manage form data.

Forms in Angular are powerful tools for creating user-friendly interfaces, ensuring data validation, and simplifying complex form operations. Template-driven forms are a declarative approach to forms where the form logic is primarily defined in the template (HTML) rather than in the component class. This makes them easy to use and ideal for simple forms.

What Are Template-driven Forms?

Template-driven forms are built using Angular's directives in the template. These forms are easy to use and do not require much code in the component class. They rely on the **FormsModule** for their functionality, and the form data is usually bound to the HTML elements using Angular's two-way data binding with `[(ngModel)]`.

Setting Up Template-driven Forms

To set up a template-driven form, we need to:

1. Import the `FormsModule` in the application module.
2. Create the form in the template using form elements and Angular's directives like `ngModel`.
3. Handle form submission and validation in the component.

Example:

typescript
Copy code

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule   // Add FormsModule to imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Template-driven form in the component's HTML:

html
Copy code

```
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)">
  <label for="username">Username</label>
  <input type="text" id="username" name="username" [(ngModel)]="user.username" required>

  <label for="email">Email</label>
  <input type="email" id="email" name="email" [(ngModel)]="user.email" required email>

  <button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>
```

In this example:

- The form is bound to the component's `user` object using `ngModel`.
- `required`, `email`, and `ngSubmit` are used for basic validation and form submission handling.

Form Data Binding

Angular provides two-way data binding with `[(ngModel)]` to bind form input fields to component properties. Any changes made in the form are reflected in the component, and vice versa.

Form Validation in Angular

Validation is an essential part of handling user input in forms. Angular provides multiple built-in validators that can be applied to form fields. Template-driven forms also offer a declarative way to perform form validation, such as required fields, pattern matching, and custom validations.

Built-in Validators

1. **Required Validator (`required`)**: Ensures that the field is not left empty.
2. **Pattern Validator (`pattern`)**: Validates input based on a regular expression pattern.
3. **Email Validator (`email`)**: Ensures the entered value is a valid email address.
4. **Minlength and Maxlength Validators (`minlength`, `maxlength`)**: Specifies minimum and maximum lengths for input fields.

Example of Validation in Template-driven Forms

html
Copy code

```
<form #form="ngForm" (ngSubmit)="onSubmit(form)">
  <div>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" [(ngModel)]="username" required>
    <div *ngIf="form.submitted && !username">
      <small class="error">Username is required.</small>
    </div>
  </div>

  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="email" required email>
    <div *ngIf="form.submitted && !email">
      <small class="error">Email is required and must be valid.</small>
    </div>
  </div>

  <button type="submit" [disabled]="!form.valid">Submit</button>
</form>
```

In this example:

- The `required` and `email` validators ensure that both fields are filled and that the email address is in the correct format.
- The error message is displayed if the form has been submitted and the field is invalid.

Displaying Validation Errors

It is important to provide feedback to users when they fill out forms incorrectly. Angular provides a simple mechanism to show error messages when a form control is invalid.

You can use `*ngIf` to conditionally display messages when a form control is invalid.

Example:

html
Copy code

```
<input type="text" name="username" [(ngModel)]="username" required />
<div *ngIf="username?.invalid && username?.touched">
  <small class="error">Username is required.</small>
</div>
```

In this case:

- The error message will only appear if the `username` field is invalid and has been touched (focused and then blurred).

Submitting Template-driven Forms

Once the form is validated, the next step is handling the form submission. This is typically done in the component's `onSubmit` method, where you can access the form data.

Example:

typescript
Copy code

```
export class AppComponent {
  username: string = '';
  email: string = '';

  onSubmit(form: NgForm) {
    if (form.valid) {
      console.log('Form Submitted:', form.value);
    }
  }
}
```

In this example:

- The `onSubmit` method receives the form object as an argument and logs the form data if the form is valid.

Comprehensive Guide to Angular Pipes - 10

In Angular, **pipes** are a powerful feature that allows you to transform data directly within the template. This transformation can be anything from formatting dates to transforming text into uppercase. Angular provides several built-in pipes, and you can also create custom pipes to meet your specific needs.

What are Pipes in Angular?

Pipes are simple functions that accept an input value and return a transformed value. They can be used directly in the template to display data in a formatted way without modifying the underlying data itself. Angular provides both **built-in pipes** (such as `date`, `uppercase`, `currency`, etc.) and the ability to create **custom pipes**.

Built-in Pipes in Angular

Angular comes with several built-in pipes to handle common data transformation tasks:

1. **DatePipe**: Formats a date value according to a specified format.
   - Example: `{{ today | date:'shortDate' }}`
2. **UppercasePipe**: Transforms the input string into uppercase.
   - Example: `{{ 'hello' | uppercase }}`
3. **LowercasePipe**: Converts the input string to lowercase.
   - Example: `{{ 'HELLO' | lowercase }}`
4. **CurrencyPipe**: Transforms a number into a currency format.
   - Example: `{{ 1000 | currency:'USD':true:'1.2-2' }}`
5. **DecimalPipe**: Formats a number according to the specified number of decimal places.
   - Example: `{{ 3.14159 | number:'1.2-2' }}`
6. **PercentPipe**: Converts a number to a percentage.
   - Example: `{{ 0.25 | percent }}`
7. **JsonPipe**: Converts an object to a JSON-formatted string.
   - Example: `{{ user | json }}`
8. **I18nSelectPipe**: Selects a string based on the current locale.
   - Example: `{{ 'post.comment' | i18nSelect:comments }}`
9. **I18nPluralPipe**: Converts numbers to plural or singular form based on the number value.
   - Example: `{{ count | i18nPlural: pluralMap }}`
10. **AsyncPipe**: Subscribes to an Observable or Promise and returns the latest value.
    - Example: `{{ dataObservable | async }}`

Creating Custom Pipes

In Angular, you can also create your custom pipes to handle specific transformations. This is particularly useful when you need to implement a functionality that is not covered by built-in pipes.

Steps to Create a Custom Pipe

1. **Define a Pipe**: You define a custom pipe by creating a class that implements the `PipeTransform` interface and adding the `@Pipe` decorator.
2. **Implement the transform Method**: The `transform()` method in the pipe class takes an input value, processes it, and returns the transformed value.
3. **Register the Pipe**: Once the pipe is defined, it needs to be declared in the module's `declarations` array.

Example: Custom Pipe to Reverse a String

typescript
Copy code

```typescript
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'reverse'
})
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

Usage in the template:

html
Copy code

```html
{{ 'hello' | reverse }}
```

This would output `"olleh"`.