Section 1: Course Fundamentals

Welcome to your first step into the world of programming with C#. This section will introduce you to the core ideas and the tools of the trade. Our goal here is to understand the "what" and "why" behind C# development, which will make learning the "how" much easier.

Introduction to .NET

When you decide to write a C# program, you are not just using the C# language by itself. You are using it within a powerful ecosystem called **.NET**. For a developer, .NET is the most important asset you have.

Think of .NET as a contract between you and the computer. You agree to write code according to the rules of C#, and in return, the .NET platform provides a staggering amount of power and safety, handling the most complex tasks for you. This power comes in two main forms:

1. The Base Class Library (BCL): Your Giant Toolbox of Pre-built Code
   Imagine being asked to build an application that needs to display the current date. This seemingly simple task involves asking the computer's operating system for its internal clock, getting the raw data, and then formatting it correctly. This is complex.
   Instead of you having to write this difficult code, the .NET BCL provides a pre-built "tool" for it. You simply write one line of C# to ask the BCL's `DateTime` tool for the current time, and it does all the hard work for you.
   The BCL is a gigantic library containing thousands of such tools, all organized neatly into namespaces. It contains tested, reliable code for:
   - **Mathematics:** You don't need to write logic for finding a square root. You just use the tools in the `System.Math` namespace.
   - **Text Manipulation:** Joining, splitting, or searching through text is made simple by the `string` and `System.Text` tools.
   - **Data Structures:** The `System.Collections.Generic` namespace gives you powerful tools like `List` for managing collections of items that can grow and shrink automatically.
   - **File and Network Operations:** The BCL provides robust tools to read files or download content from a website, handling the complexity behind the scenes.

2. This lets you focus on solving your unique problem, rather than re-solving common problems that Microsoft's engineers have already perfected.

3. The Runtime: Your Code's Guardian and Manager
   The other half of .NET is the runtime environment that executes your code, called the Common Language Runtime (CLR). It acts as the engine of your application.

CLR (Common Language Runtime)

The CLR is the heart of the .NET platform. It's a sophisticated program that manages your C# application while it's running. It acts as a protective bubble, a translator, and a manager. A computer's processor only understands its own native machine code; it does not understand C#. The CLR's primary job is to bridge this gap.

The Translation Process (JIT Compilation)

1. **The Intermediate Step (CIL):** Your C# source code is first compiled into **Common Intermediate Language (CIL)**. CIL is not tied to any specific operating system or processor. It's a universal, platform-agnostic language. Your compiled program file (a .dll or .exe) is a container for this CIL code.

2. **The "Just-In-Time" (JIT) Compilation:** When you run your program and a method is called for the very first time, the CLR's JIT Compiler translates the CIL for that specific method into the native machine code for the processor it's currently running on.

3. **Efficiency and Caching:** This "Just-In-Time" approach is highly efficient because it only translates code that is actually needed. After translating a method once, the CLR caches the resulting native code. The next time the method is called, the CLR skips the translation and executes the super-fast native code directly.

The Other Jobs of the CLR: Your Free Safety Net

Besides translation, the CLR provides critical services that make C# development safer and more productive:

- **Memory Management:** The CLR automatically handles allocating memory for your objects and, crucially, cleaning it up when you're done (a process called Garbage Collection). This prevents a major source of bugs common in older languages.

- **Security Enforcement:** The CLR acts like a security guard, preventing your code from doing inherently dangerous things, like accessing another program's memory.
- **Exception Handling:** It provides a structured way to handle errors that occur while your program is running, preventing your application from simply crashing.

.NET Framework Architecture

To understand modern .NET, it helps to know the structure of its predecessor, the classic **.NET Framework**. Think of its architecture as a layered cake, where each layer builds upon the one below it.

- Layer 1: The Foundation - Common Language Runtime (CLR)
  At the very bottom is the CLR, sitting directly on top of the Windows Operating System and acting as the fundamental execution engine.
- Layer 2: The Core Structure - Base Class Library (BCL)
  The next layer is the BCL, providing the fundamental "building materials" like file I/O and data collections that all applications use.
- Layer 3: The Specialized Frameworks
  On top of the BCL, the .NET Framework provided several powerful, specialized frameworks for building specific kinds of applications on Windows:
  - **ASP.NET:** For building web applications and websites.
  - **Windows Presentation Foundation (WPF) & Windows Forms:** For building desktop applications with graphical user interfaces.
  - **ADO.NET:** For connecting to and working with databases.

The key takeaway is that the .NET Framework was a monolithic, tightly integrated system designed specifically for Windows. Modern .NET has evolved from this architecture to be more modular and cross-platform.

Versions of .NET Framework and .NET Core

The history of .NET is a tale of two platforms: the original, Windows-only framework and the modern, cross-platform successor.

- **.NET Framework (The Legacy Platform)**
    - **Versions:** Ranged from 1.0 (2002) to the final version, **4.8** (2019).
    - **Platform:** Windows only.
    - **Status:** It is now in maintenance mode. It receives security updates, but no new features are being added. You will still find it in many established enterprise applications.
- **.NET Core and modern .NET (The Successor)**
    - **The Rewrite:** Microsoft created **.NET Core** from the ground up to be open-source, high-performance, and **cross-platform** (working on Windows, macOS, and Linux). Its versions ran from 1.0 to 3.1.
    - **The Unification:** Starting with **.NET 5** (released in 2020), Microsoft dropped the "Core" and "Framework" branding to create a single, unified platform going forward. All new development happens on this platform.
    - **Current Versions:** Releases are now annual: .NET 6, .NET 7, .NET 8, and so on. Even-numbered versions (like .NET 8) are designated **LTS (Long-Term Support)**, meaning they are guaranteed support for three years, making them the best choice for new projects.

Introduction to Visual Studio

Visual Studio is a world-class **Integrated Development Environment (IDE)**. It is your command center for writing, managing, and debugging C# code. Getting comfortable in this environment is key to your productivity.

- **The Code Editor:** This is the "smart" text editor where you'll write your code. Its most powerful feature is **IntelliSense**, which actively helps you as you type by providing auto-completion, parameter info, and lists of available actions (methods) on your objects. It's a discovery tool as much as a writing tool.
- **Solution Explorer:** This window (typically on the right) is your project's table of contents. It shows you every file and folder that makes up your application. This is how you navigate your project.
- **Error List:** This window (typically at the bottom) is your personal quality inspector. If you make a mistake, a detailed error message will appear in this list instantly, often telling you the exact file and line number where the problem exists.

- The Debugger: Your Most Powerful Tool
  The Debugger allows you to pause your program at any point and inspect what's happening.
    - **Breakpoints:** You can click in the margin next to a line of code to set a **breakpoint**. When you run your program, it will pause execution completely when it hits that line.
    - **Inspection:** While paused, you can hover your mouse over any variable to see the value it contains at that exact moment. This is how you find logical errors. You can then step through your code line-by-line to see exactly how values change and pinpoint where things went wrong. Mastering the debugger is the fastest way to solve problems in your code.

Introduction to C#

C# (pronounced "C Sharp") is the modern, powerful, and versatile programming language you will be learning. Its single most important characteristic is that it is **statically-typed**.

This means that when you create a variable (a named storage location), you must declare the type of data it will hold. If you create a variable to hold a number, you can only ever put numbers in it. If you try to put text in it, the C# compiler will immediately show you an error before you even run the program. This compile-time checking is like having a vigilant assistant who proofreads your work, catching a massive category of bugs for you automatically.

The Building Blocks: Class, Object, Fields, and Methods

Object-Oriented Programming (OOP) is the paradigm C# is built on. It's a way of structuring your code around the concepts of "blueprints" and "objects."

1. Creating the Blueprint (The Class)
   You, the programmer, write a class. A class is a text file that acts as the blueprint. Inside the class, you define its components:
    - **Fields:** These are variables that hold the state or data of each individual object created from the class (e.g., color, model, currentSpeed).

    - **Methods:** These are blocks of code that define the actions an object can perform (e.g., Accelerate(), Brake()).

2. Building Real Things from the Blueprint (Objects)
   In your main program, you use the new keyword to create an object, which is a real instance of the class constructed in the computer's memory. You can create as many objects as you want from a single class blueprint.

3. Interacting with Objects
   Each object you create is independent. If you create car1 and car2 from the Car class, setting the color field of car1 to "Red" has no effect on car2. Calling the Accelerate() method on car2 only changes the currentSpeed field of car2. This allows you to model complex, real-world systems in a clean and organized way.

Introduction to Namespaces

A namespace is a way to **organize code and prevent naming conflicts**. In a large application, you might have hundreds of classes, and some could have the same name. Namespaces group related classes together, like putting files into folders.

Imagine you have two classes named Logger, one for writing to a file and one for writing to a database. You could place them in different namespaces:

- MyProject.FileLogging.Logger
- MyProject.DatabaseLogging.Logger

Their full names are now unique. At the top of a code file, you can add a using directive (e.g., using MyProject.FileLogging;) to tell the compiler which "folder" of classes you intend to use.

The Grammar of Code: C# Language Tokens

Every C# instruction is composed of smaller pieces called tokens. Let's analyze a single line of code:

int userAge = 30;

This line contains five distinct tokens:

1. int: A **Keyword** that specifies a data type for an integer number.

2. userAge: An **Identifier**, the name you chose for your variable.
3. =: An **Operator** used to assign a value.
4. 30: A **Literal**, a fixed value written directly in the code.
5. ;: A **Punctuator** used to mark the end of the statement.

Recognizing these helps you read code and decipher compiler errors.

The Evolution of a Language: Versions of C#

The C# language is constantly evolving. Each new version gives developers better tools to write more expressive, concise, and safer code.

- **C# 1.0:** The basic, object-oriented language.
- **C# 2.0:** Introduced **Generics**, a cornerstone feature for creating type-safe, reusable collections.
- **C# 3.0:** A landmark release that brought us **LINQ**, a feature that dramatically simplified querying data.
- **C# 5.0:** Another landmark that introduced async and await, which made writing responsive, non-blocking applications far easier.
- **C# 6.0 and beyond:** More recent versions have focused on developer productivity, adding dozens of features to let you achieve the same results with less code. As a new developer, you get the benefit of two decades of language refinement from day one.

Professional Habits: C# Naming Conventions

Following a consistent naming style is not required by the compiler, but it is **essential** for writing professional, readable code.

1. **PascalCase**: Capitalize the first letter of every word. Use for "big" or "public" things.
    - **Class names:** ClientManager, UserPermissions
    - **Method names:** GetData, CalculateFinalScore
2. **camelCase**: The first letter is lowercase; the first letter of every following word is capitalized. Use for "local" or "private" things.

- **Local variables (inside a method):** int itemCounter = 0;
- **Method parameters (inputs to a method):** public void SetAge(int newAge)

Adopting these conventions from the very beginning is one of the best habits you can form.

Section 2: C# Language Basics

Lecture 1: Installing Visual Studio

Introduction

Visual Studio is the professional Integrated Development Environment (IDE) that you'll use for C# development. Think of it as your complete workshop, containing not just a text editor, but all the specialized tools you need to build, debug, and manage your applications efficiently.

How it Works & Best Practices

When you install Visual Studio, you select Workloads, which are bundles of tools for specific development types. The ".NET desktop development" workload is essential for our purposes, as it includes the C# compiler, the .NET Framework, and project templates.

Interview Perspective

If an interviewer asks about your favorite Visual Studio features, you should highlight the Debugger and IntelliSense.

- **The Debugger**: Explain that it's your primary tool for troubleshooting. It allows you to set **breakpoints** to pause execution, **step through code** line-by-line, and **inspect the state of variables**.
- **IntelliSense**: Describe it as intelligent code completion that suggests code, checks for errors as you type, and helps you discover the features of the C# language.

Lecture 2: Creating Your First C# Application

Introduction

The "Hello, World!" application is a fundamental first step. It confirms your environment is set up correctly and introduces you to the basic structure of a C# program. When creating your project, be sure to select the "Console App (.NET Framework)" template.

How it Works: The Structure of a Program

The code Visual Studio generates resides in a file named Program.cs.

C#

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

- namespace: An organizer for your code, like a folder.
- class Program: A container for your methods and data.
- static void Main(string[] args): This is the **Entry Point**. When you run your program, the .NET runtime looks for this specific method to begin execution.

Interview Perspective

- **"Explain this line: static void Main(string[] args)"**: This is a very common interview question.
    - static: The method is called on the Program class itself, not on an object created from it.
    - void: The method does not return any value.
    - Main: The conventional name for the entry point.
    - string[] args: The parameter that accepts command-line arguments as an array of strings.

Lecture 3: The System.Console Class

Introduction

The Console class is your primary tool for interacting with the text-based console window. It provides fundamental methods for input and output.

Common Methods

- Console.WriteLine(string value): Prints the specified text to the console, followed by a new line character.
- Console.Write(string value): Prints the text but leaves the cursor on the same line.
- Console.ReadLine(): Reads a full line of text entered by the user and returns it as a string.
- Console.ReadKey(): Waits for a single keypress.
- Console.Clear(): Clears all text from the console window.

Interview Perspective

- **Write vs. WriteLine**: WriteLine moves to the next line after printing; Write does not.
- **Input Handling**: A frequent question is, "If you ask a user for their age, how do you store it?" The key is knowing that Console.ReadLine() always returns a string. You must explicitly convert this string to a number using a method like int.Parse().

Lecture 4: Variables

Introduction

A variable is a named placeholder in memory used to store a value that your program can manipulate.

How it Works: Scope and Memory

When you declare a variable of a primitive type inside a method, its value is stored on the stack, a very fast region of memory. The variable's lifetime is tied to its scope, which is the code block {...} where it is declared.

Best Practices & Interview Perspective

- **Naming**: Always use **camelCase** (userAge, firstName) for local variables.
- **Scope**: Be prepared for questions like, "If I declare a variable inside an if block, can I use it in the else block?" The answer is no, because they are separate scopes.

Lecture 5: Primitive Types

Introduction

Primitive types are the fundamental data building blocks. C# is a statically-typed language, so you must declare the type of a variable before you use it.

Integer Types (Whole Numbers)

These types store whole numbers without decimal points. The main difference is their size and whether they can store negative values.

- sbyte: 8-bit signed integer (-128 to 127).
- byte: 8-bit unsigned integer (0 to 255). Use when a value can't be negative, like for raw data from a file or network stream.
- short: 16-bit signed integer (-32,768 to 32,767).
- ushort: 16-bit unsigned integer (0 to 65,535).
- int: 32-bit signed integer (-2.1 billion to 2.1 billion). This is the **default and most common integer type**. Use it unless you have a specific reason not to.
- uint: 32-bit unsigned integer (0 to 4.2 billion).
- long: 64-bit signed integer (a very large range). Use when int is not large enough. To specify a long literal, use the L suffix: long bigNumber = 9000000000L;.
- ulong: 64-bit unsigned integer.

Floating-Point Types (Decimal Numbers)

These types store numbers with decimal points. The choice involves a trade-off between range, precision, and performance.

- float: 32-bit single-precision (approx. 6-9 digits of precision). Use when you need a decimal but don't require high precision, like in graphics programming. Use the F suffix: float height = 1.88F;.
- double: 64-bit double-precision (approx. 15-17 digits of precision). This is the **default and most common floating-point type**, used for scientific and general calculations.
- decimal: 128-bit high-precision (28-29 digits of precision). This type is slower than double but eliminates the small rounding errors inherent in binary floating-point types. **You must use decimal for any financial or monetary calculations.** Use the M suffix: decimal accountBalance = 100.05M;.

Other Primitive Types

- char: Represents a single 16-bit Unicode character. The value is enclosed in single quotes: char grade = 'A';.
- bool: Represents a logical value, either true or false.

Lecture 6: Operators

Introduction

Operators are special symbols that perform operations on variables and values (operands).

Arithmetic Operators

- + (Addition): Adds two operands. int sum = 10 + 5; // 15
- - (Subtraction): Subtracts the right operand from the left. int diff = 10 - 5; // 5
- * (Multiplication): Multiplies two operands. int prod = 10 * 5; // 50
- / (Division): Divides the left operand by the right. Its behavior depends on the operand types.
    - **Integer Division**: If both operands are integers, the result is an integer, and the decimal part is **truncated (discarded)**. int result = 10 / 4; // result is 2
    - **Floating-Point Division**: If at least one operand is a floating-point type, the result is a floating-point type. double result = 10.0 / 4; // result is 2.5
- % (Modulus): Returns the remainder of an integer division. int remainder = 10 % 3; // remainder is 1. Useful for checking if a number is even (num % 2 == 0).

Assignment Operators

- = (Simple Assignment): Assigns the value of the right operand to the left operand. int x = 10;
- +=, -=, *=, /=, %= (Compound Assignment): Performs an operation and an assignment in one step. x += 5; is syntactic sugar for x = x + 5;.

Comparison Operators

These compare two operands and always return a bool (true or false).

- == (Equal to): bool areEqual = (name == "John");
- != (Not equal to): bool areNotEqual = (age != 30);
- >, <, >=, <= (Greater than, Less than, etc.): bool isAdult = (age >= 18);

Logical Operators

These are used to combine bool expressions.

- && (Logical AND): Returns true only if both operands are true.
- || (Logical OR): Returns true if at least one operand is true.
- ! (Logical NOT): Inverts the value of a bool. !true is false.

Ternary Operator

- ?: (Conditional Operator): A shortcut for a simple if-else statement. string message = (score >= 50) ? "Pass" : "Fail";

Interview Perspective: Data Types and Assignments

- **Literals**: You can assign values in different formats. Interviewers might check this knowledge.
    - Hexadecimal: int hex = 0x2A; // 42 in decimal
    - Binary: int bin = 0b00101010; // 42 in decimal
- **Type Casting and Data Loss**: This is a critical interview topic.
    - **Implicit Casting (Safe)**: A conversion from a smaller type to a larger type where no data can be lost. C# does this automatically. int i = 100; double d = i; // d is 100.0
    - **Explicit Casting (Unsafe)**: A conversion from a larger type to a smaller type where data might be lost. You must tell the compiler you're aware of the risk by using a cast (type). double d = 9.78; int i = (int)d; // i is 9. The decimal part is lost. An interviewer will expect you to explain this concept of potential data loss.

Lecture 7: if and its forms

Simple if Statement

When to Use: When you want to execute a block of code only if a single condition is true, and do nothing otherwise.

How it Works: The boolean expression inside the parentheses is evaluated. If it's true, the code block inside the curly braces is executed. If it's false, the block is skipped.

Example:

C#

```
int currentStock = 12;
if (currentStock < 10)
{
    Console.WriteLine("Warning: Stock is low. Time to re-order.");
}
```

if-else Statement

When to Use: When you have a single condition but need to execute one block of code if it's true and a different block if it's false.

How it Works: If the condition is true, the first block (if) is executed. If it's false, the second block (else) is executed. One of the two blocks will always execute.

Example:

C#

```
int userAge = 17;
if (userAge >= 18)
{
    Console.WriteLine("Access granted to the restricted area.");
}
else
{
    Console.WriteLine("Access denied. You must be 18 or older.");
}
```

if-else if Ladder

When to Use: When you have multiple related conditions to check in sequence.

How it Works: The conditions are evaluated from top to bottom. The code block for the first condition that evaluates to true is executed, and all subsequent else if and else blocks are skipped. The final else is optional and acts as a catch-all if none of the preceding conditions are true.

Example:

C#

```
double score = 85.5;
if (score >= 90)
{
    Console.WriteLine("Grade: A");
}
else if (score >= 80)
{
    Console.WriteLine("Grade: B");
}
else if (score >= 70)
{
```

```
    Console.WriteLine("Grade: C");
}
else
{
    Console.WriteLine("Grade: F");
}
```

Nested if Statements

When to Use: When a condition can only be checked if another, outer condition is already true.

How it Works: The inner if statement is only evaluated if the outer if statement's condition is true.

Example:

C#

```
bool hasTicket = true;
bool isVip = false;
if (hasTicket)
{
    Console.WriteLine("Welcome! Please proceed.");
    if (isVip)
    {
        Console.WriteLine("Please enjoy the VIP lounge.");
    }
}
else
{
    Console.WriteLine("You need a ticket to enter.");
}
```

Best Practice: Avoid nesting if statements too deeply (more than 2 or 3 levels) as it can make the code very difficult to read and understand.

Lecture 8: switch-case

When to Use It: A switch is a cleaner, often more readable, alternative to a long if-else if ladder when you are checking a single variable against a series of specific, constant values.

How it Works: The value of the variable in the switch() is compared to the value of each case. When a match is found, the code in that case block is executed until a break statement is hit. The break causes execution to jump out of the switch block. The default case is executed if no other cases match.

Example:

C#

```
string userRole = "admin";
switch (userRole.ToLower()) // Good practice to normalize string input
{
    case "admin":
        Console.WriteLine("You have full access.");
        break;
    case "editor":
        Console.WriteLine("You can create and edit content.");
        break;
    case "viewer":
        Console.WriteLine("You can only view content.");
        break;
    default:
        Console.WriteLine("Unknown role. Access denied.");
        break;
}
```

Lecture 9: while Loop

When to Use It: When you need to loop based on a condition, and you do not know the exact number of iterations beforehand.

How it Works: The boolean condition is evaluated before each potential iteration. If true, the loop body executes. This process repeats until the condition becomes false. If the condition is false to begin with, the loop body never executes.

Examples:

- **Counter-Controlled Loop**:
  C#
  int ticketsLeft = 5;

- while (ticketsLeft > 0)
- {
- Console.WriteLine($"Selling a ticket... {ticketsLeft} tickets remaining.");

```
        ticketsLeft--; // This change is crucial to prevent an infinite loop
    }
    Console.WriteLine("All tickets sold!");
```

- **Sentinel-Controlled Loop (waiting for user input)**:
  C#
  ```
  string command = "";

  while (command != "quit")
  {
      Console.Write("Enter a command ('help', 'run', 'quit'): ");
      command = Console.ReadLine();
      // ... process the command ...
  }
  ```

Lecture 10: do-while Loop

When to Use It: In the specific scenario where you need to guarantee the loop body executes at least once.

How it Works: The loop body executes first, and then the boolean condition is evaluated. If true, the loop repeats.

Example (the classic use case: input validation):

C#

```
int age;
do
{
    Console.Write("Please enter your age (must be 18 or older): ");
    // This code must run at least once to get the input
    string input = Console.ReadLine();
    age = int.Parse(input);
    if (age < 18)
    {
        Console.WriteLine("Invalid age. Please try again.");
    }
} while (age < 18); // Check the condition after the first run
Console.WriteLine("Age successfully validated.");
```

Lecture 11: for Loop

When to Use It: When you know the number of iterations before the loop starts.

How it Works: for (initializer; condition; iterator)

1. The initializer runs once at the beginning.
2. The condition is checked before each iteration.
3. The loop body executes if the condition is true.
4. The iterator runs after each iteration.

Common Examples:

- **Incrementing from 0 to n-1 (for arrays):**
  C#
  string[] shoppingList = { "Apples", "Bananas", "Carrots" };
- for (int i = 0; i < shoppingList.Length; i++)
- {
- Console.WriteLine($"Item {i + 1}: {shoppingList[i]}");
- }
-
- **Decrementing Loop (rocket launch):**
  C#
  for (int i = 10; i > 0; i--)
- {
- Console.WriteLine($"{i}...");
- }
- Console.WriteLine("Liftoff!");
-
- **Loop that skips numbers (even numbers):**
  C#
  for (int i = 2; i <= 10; i += 2)
- {
- Console.WriteLine(i);
- }
-

Lecture 12: break and continue

When to Use Them: To manually alter the flow of a loop from within its body.

- **break**: Use when you need to **exit the loop immediately**, regardless of the loop's condition. For example, if you find the item you're searching for.
- **continue**: Use when you want to **skip the current item** and move directly to the next iteration. For example, to ignore invalid data in a list.

Example:

C#

```
// Find the first multiple of 7, but ignore numbers less than 20.
for (int i = 1; i <= 100; i++)
{
    if (i < 20)
    {
        continue; // Skip this number and go to the next i
    }
    if (i % 7 == 0)
    {
        Console.WriteLine($"Found the first multiple of 7 greater than 20: {i}");
        break; // Found it, exit the loop completely
    }
}
```

Lecture 13: Nested for Loops

When to Use It: Whenever you need to work with a two-dimensional structure, like a grid, a multiplication table, or a game board.

How it Works: The inner loop completes its entire run for every single iteration of the outer loop.

Example (Multiplication Table):

C#

```
// Outer loop for the first number in the multiplication
for (int i = 1; i <= 5; i++)
{
    // Inner loop for the second number
    for (int j = 1; j <= 5; j++)
    {
        // Use Write to keep everything on the same line for this row
        Console.Write($"{i * j}\t"); // \t adds a tab for spacing
    }
```

```
    // After the inner loop finishes, move to the next line for the next row
    Console.WriteLine();
}
```

Lecture 14: goto

Introduction and Warning: The goto statement provides an unconditional jump to a label. While it is part of the language, its use is **extremely discouraged** in modern programming. It creates "spaghetti code" that is difficult to read and maintain. Structured control flow (if, switch, loops) is always the superior choice. An interviewer will expect you to know this and to advocate against using goto.

Lecture 15: Pattern Printing using Nested for Loops

Introduction: This is a classic exercise to demonstrate your command of nested loops and algorithmic thinking.

Example 1: Right-Angled Triangle

C#

```
int size = 5;
for (int row = 1; row <= size; row++)
{
    for (int col = 1; col <= row; col++)
    {
        Console.Write("*");
    }
    Console.WriteLine();
}
```

How it Works: The outer loop controls the rows. The inner loop's condition col <= row is the key; it prints a number of stars equal to the current row number.

Example 2: Inverted Right-Angled Triangle

C#

```
int size = 5;
for (int row = size; row >= 1; row--)
{
    for (int col = 1; col <= row; col++)
    {
```

```
        Console.Write("*");
    }
    Console.WriteLine();
}
```

How it Works: The logic is reversed. The outer loop counts down from size. The inner loop still prints a number of stars equal to the current row value, which is now decreasing.

Example 3: Pyramid

This is a more advanced pattern that requires a third loop for spacing.

C#

```
int size = 5;
for (int row = 1; row <= size; row++)
{
    // Loop for leading spaces
    for (int space = 1; space <= size - row; space++)
    {
        Console.Write(" ");
    }
    // Loop for the stars
    for (int star = 1; star <= (2 * row) - 1; star++)
    {
        Console.Write("*");
    }
    Console.WriteLine();
}
```

How it Works:

1. **Outer Loop:** Controls the row number.
2. **Space Loop:** For each row, it prints size - row spaces. On row 1, it prints 4 spaces; on row 5, it prints 0 spaces. This centers the pattern.
3. **Star Loop:** Prints (2 * row) - 1 stars. On row 1, it prints 1 star; on row 2, it prints 3; on row 3, it prints 5, creating the pyramid shape.

Section 4: C# OOP - Basics

Lecture 1: Understanding OOP Fundamentals

Introduction

Object-Oriented Programming (OOP) is a fundamental shift in how we think about and structure our programs. Before OOP, the dominant style was procedural programming, where a program was essentially a long list of instructions or functions that acted upon data. This works for small programs, but as applications grow in size and complexity, it becomes incredibly difficult to manage. Changes in one part of the code can unpredictably break other parts, leading to what is often called "spaghetti code."

OOP was created to solve this problem of complexity. The core idea is simple yet powerful: instead of separating data and the operations that act on that data, we bundle them together into conceptual "objects." We stop thinking about our program as a sequence of steps and start thinking about it as a collection of interacting objects, where each object models a distinct entity from the problem we're trying to solve.

The Core Idea: Modeling the World

An "object" in programming represents a person, place, thing, or even an abstract concept. Think about a simple e-commerce application. A procedural approach might have separate variables for product names, prices, and inventory counts, and separate functions to update inventory or calculate totals.

An OOP approach would model this differently:

- We would create a Product **object**. This single object would contain its own data: its name, its price, and its stockLevel.
- It would also contain its own behavior: methods like Purchase(quantity) or IsInStock().
- Similarly, we would have a Customer object with its own data (name, address) and behavior (PlaceOrder()).
- We could have a ShoppingCart object with its own data (a list of products) and behavior (AddItem(), CalculateTotal().

The program then becomes a simulation of these objects interacting: a Customer object PlaceOrder(), which might involve a ShoppingCart object that checks if each Product object IsInStock() before finalizing the purchase.

The Three Core Concepts of an Object

Every object in OOP has three key characteristics:

1. **Identity:** Each object is a unique entity, distinct from all other objects, even if they are of the same type and have the same data. When we create two different Product objects, they occupy different spaces in memory.
2. **State:** This is the data or attributes that describe an object at any given moment. A Car object's state would include its current color, currentSpeed, and fuelLevel. The state is stored in **fields** (variables) within the object.
3. **Behavior:** This is what an object can do. The behavior of a Car object would be defined by its **methods**, such as Accelerate(), Brake(), and Refuel(). These methods are typically used to modify the object's internal state.

Interview Perspective

- **Question:** "Can you explain what Object-Oriented Programming is at a high level?"
- **Ideal Answer:** "OOP is a programming paradigm that organizes software design around data, or 'objects,' rather than functions and logic. It's a way of managing complexity by bundling data (state) and the methods that operate on that data (behavior) into a single unit. Instead of a top-down procedural approach, an OOP application is modeled as a collection of interacting objects, which often represent real-world entities. This makes the code more modular, reusable, and easier to maintain as it grows."

Lecture 2: Creating Classes & Objects

Introduction

Now that we understand the concept of an "object," we need a way to define what our objects will look like and how they will behave. This is the role of a class. A class is the blueprint, the template, or the design for an object. An object is the actual concrete instance that is built from that blueprint.

You can have one Car class (the blueprint), but you can use it to build thousands of individual Car objects, each with its own unique state (one might be a red Ferrari, another a blue Ford).

Defining a Class

A class is defined using the class keyword, followed by the name of the class (which should be in PascalCase). Inside the curly braces {}, we define its members: fields and methods.

C#

```
// This is the blueprint for all Car objects.
public class Car
{
    // 1. Fields: These are variables that store the state of an object.
    // By convention, we make fields private to protect them (Encapsulation).
    // The underscore prefix (_) is a common naming convention for private fields.
    private string _color;
    private string _model;
    private int _currentSpeed;

    // 2. Methods: These are functions that define the object's behavior.
    // We make them public so other parts of our code can use them.
    public void Accelerate()
    {
        // This method modifies the object's internal state.
        _currentSpeed += 10;
        Console.WriteLine($"Accelerating. Current speed: {_currentSpeed} km/h");
    }

    public void Brake()
    {
        if (_currentSpeed > 0)
        {
            _currentSpeed -= 5;
            Console.WriteLine($"Braking. Current speed: {_currentSpeed} km/h");
        }
    }

    // A method to provide controlled access to a private field's value.
    public string GetColor()
    {
        return _color;
    }

    // A method to provide controlled modification of a private field's value.
```

```
    public void SetColor(string newColor)
    {
        // We could add validation here if we wanted.
        _color = newColor;
    }
}
```

Creating an Object (Instantiation)

Once you have the class (blueprint), you can create objects from it. This process is called instantiation. We use the new keyword to do this.

C#

```
// The Main method where our program runs
static void Main(string[] args)
{
    // 1. Declaration: Declare a variable that can hold a reference to a Car object.
    Car myFirstCar;

    // 2. Instantiation: Use the 'new' keyword to create a new Car object in memory
    //    and assign its memory address to our variable.
    myFirstCar = new Car();

    // Or, more commonly, do it all in one line:
    Car mySecondCar = new Car();

    // Now we have two distinct Car objects in memory.
}
```

Accessing Members

You interact with an object's public members using the dot operator (.).

C#

```
// Continuing in the Main method...

// Use the public methods to set the state of the first car.
myFirstCar.SetColor("Red");

// Call methods to change its behavior.
myFirstCar.Accelerate(); // Output: Accelerating. Current speed: 10 km/h
myFirstCar.Accelerate(); // Output: Accelerating. Current speed: 20 km/h
myFirstCar.Brake();      // Output: Braking. Current speed: 15 km/h

// Now, interact with the second car. It is completely separate.
```

```
mySecondCar.SetColor("Blue");
mySecondCar.Accelerate(); // Output: Accelerating. Current speed: 10 km/h

// Getting the color of the first car does not affect the second.
Console.WriteLine($"My first car's color is: {myFirstCar.GetColor()}");   //
Output: Red
Console.WriteLine($"My second car's color is: {mySecondCar.GetColor()}"); //
Output: Blue

// IMPORTANT: You cannot access private fields directly from outside the
class.
// The following line would cause a compiler error:
// myFirstCar._currentSpeed = 100; // ERROR! '_currentSpeed' is
inaccessible.
```

Interview Perspective

- **Question:** "What is the difference between a class and an object?"
- **Ideal Answer:** "A class is a blueprint or template that defines the properties and behaviors for a type of thing. It exists at compile-time and doesn't occupy memory itself. An object is a concrete instance of a class, created at run-time. It's the actual thing that occupies memory and has its own state. You can create many objects from a single class, just like you can build many houses from one blueprint."

Lecture 3: IMP Points to remember about Classes and Objects

Introduction

Understanding a few crucial, behind-the-scenes concepts about classes and objects is vital. These points are frequently the subject of interview questions because they test for a deeper understanding beyond just the basic syntax.

1. A Class is a Reference Type

This is the single most important concept to understand at this stage. When you create a variable of a class type, the variable does **not** hold the object itself. Instead, it holds a **reference**—which is essentially the memory address—to where the object is stored. The actual object lives on a region of memory called the **heap**.

How It Works & Why It Matters:

Because the variable holds a reference, not the object, you can have multiple variables referencing the same object.

C#

```
// We create ONE car object in memory.
// carA holds the memory address of this object.
Car carA = new Car();
carA.SetColor("Green");

// We declare a new variable, carB.
// We then copy the VALUE of carA into carB.
// The value of carA is the memory address, NOT the object itself.
// Now, both carA and carB point to the EXACT SAME object in memory.
Car carB = carA;

Console.WriteLine($"Car A color: {carA.GetColor()}"); // Output: Green
Console.WriteLine($"Car B color: {carB.GetColor()}"); // Output: Green

// Now, let's change the color using the carB reference.
carB.SetColor("Yellow");
Console.WriteLine("--- Color changed via carB ---");

// Because both variables point to the same object, the change is
// reflected when we access the object through carA as well.
Console.WriteLine($"Car A color: {carA.GetColor()}"); // Output: Yellow
Console.WriteLine($"Car B color: {carB.GetColor()}"); // Output: Yellow
```

Interview Perspective:

- **Question:** "What happens when you assign one object variable to another? Does it create a copy of the object?"
- **Ideal Answer:** "No, it does not create a copy of the object. Since a class is a reference type, the variable holds a reference to the object's location in memory. When you assign one variable to another, you are only copying the reference. Both variables will then point to the same single object on the heap. Any changes made to the object through one variable will be visible through the other."

2. The new Keyword Deconstructed

The new keyword is more than just a piece of syntax; it performs three distinct operations in order:

1. **Memory Allocation:** It allocates a block of memory on the **heap** large enough to hold all the fields defined in the class.
2. **Constructor Execution:** It calls a special method on the class called a **constructor** (more on this in a later section). The constructor's job is to initialize the object's fields to a valid starting state.

3. **Return Reference:** After the object is initialized, the `new` operator returns the memory address (the reference) of the newly created object, which is then stored in your variable.

3. Members of a Class

So far we have seen two types of members: fields and methods. However, a class can contain several other kinds of members, which you will learn about later. It's good to be aware of the terminology:

- **Fields:** Variables that store the object's state.
- **Methods:** Functions that define the object's behavior.
- **Properties:** A more advanced way to expose data that combines the simplicity of a field with the control of a method (often used instead of `Get...` and `Set...` methods).
- **Constructors:** Special methods used to initialize a new object when it's created.
- **Events:** A mechanism for an object to provide notifications to other objects.
- **And more...** (like finalizers, indexers, and nested types).

Section 5: Fields

Lecture 1: Introduction to Fields

Introduction

In Object-Oriented Programming, fields are variables declared directly inside a class. They are the fundamental building blocks for storing data and represent the state, or attributes, of an object. If a class is a blueprint for a Student, the fields would be the lines on that blueprint for name, studentId, and dateOfBirth. Every object created from the class gets its own copy of these fields to store its unique state.

How It Works: Memory Allocation and Object Isolation

This is a critical concept. When you create an object using the new keyword, the .NET runtime allocates a block of memory on a region called the heap. This block is sized to hold all of the instance fields defined in that object's class.

The most important takeaway is that **each object gets its own, separate block of memory**. This ensures that objects are isolated from one another. Modifying the state (the field values) of one object has no effect on the state of another object of the same class.

Example 1: Demonstrating Object Isolation

This example proves that two objects have separate memory for their fields.

C#

```
public class Student
{
    // Fields to store the state of a single student.
    // We use public here only for a clear demonstration.
    public int studentId;
    public string name;
}

public class Program
{
    public static void Main(string[] args)
    {
```

```
        // Create the first Student object. A block of memory is allocated on
the heap for it.
        Student student1 = new Student();
        student1.studentId = 101;
        student1.name = "Alice";

        // Create a second, completely separate Student object.
        // A NEW block of memory is allocated on the heap for this one.
        Student student2 = new Student();
        student2.studentId = 102;
        student2.name = "Bob";

        Console.WriteLine($"Student 1 Info: ID={student1.studentId},
Name={student1.name}");
        Console.WriteLine($"Student 2 Info: ID={student2.studentId},
Name={student2.name}");

        // Now, let's change the name of the first student.
        student1.name = "Alicia";
        Console.WriteLine("\n--- Changed student1's name ---");

        // The change only affects the first object because they have separate
memory.
        Console.WriteLine($"Student 1 Info: ID={student1.studentId},
Name={student1.name}"); // Name is now Alicia
        Console.WriteLine($"Student 2 Info: ID={student2.studentId},
Name={student2.name}"); // Name is still Bob
    }
}
```

How It Works: Copying Object References

A class is a reference type. This means the variable (student1) does not hold the object itself, but rather the memory address (a reference) to the object on the heap. When you assign one object variable to another, you are only copying this memory address.

Example 2: Demonstrating Reference Sharing

C#

```
public static void Main(string[] args)
{
    // studentA holds a reference to a new Student object in memory.
    Student studentA = new Student();
    studentA.name = "Charles";
```

```
    // studentB is created, and the REFERENCE from studentA is copied into it.
    // Both variables now point to the EXACT SAME object on the heap.
    Student studentB = studentA;

    Console.WriteLine($"Student A's Name: {studentA.name}"); // Output: Charles
    Console.WriteLine($"Student B's Name: {studentB.name}"); // Output: Charles

    // Let's change the name using the studentB reference.
    studentB.name = "Charlie";
    Console.WriteLine("\n--- Changed name via studentB ---");

    // Because they both point to the same object, the change is visible through studentA.
    Console.WriteLine($"Student A's Name: {studentA.name}"); // Output: Charlie
}
```

Interview Perspective

You must be able to clearly explain the difference between the two scenarios above.

- **Question:** "What is the difference between creating two new objects versus assigning one object variable to another?"
- **Ideal Answer:** "Creating two objects with new allocates two separate blocks of memory on the heap, making them independent. Assigning one object variable to another does not create a new object; it only copies the memory reference. As a result, both variables point to the same single object, and a change made through one variable is visible through the other."

Lecture 2: Access Modifiers for Fields

Introduction

Access modifiers are keywords that define the visibility of a class member. They are the core mechanism for enforcing encapsulation, which dictates that an object's internal data should be protected from direct, uncontrolled external access. The golden rule is to make your fields as private as possible.

List and Explanation of Access Modifiers

- **public**
  - **Accessibility:** Can be accessed from anywhere.
  - **When to Use: Almost never for fields.** Making a field public breaks encapsulation entirely. It's only used for constants or in simple Data Transfer Objects (DTOs).
- **private**
  - **Accessibility:** Can only be accessed by code within the **same class**. This is the **default modifier** if you don't specify one.
  - **When to Use: This should be your default choice for all fields.** It is the cornerstone of encapsulation. It forces all interaction with the data to go through public methods or properties.
- **protected**
  - **Accessibility:** Can be accessed within the containing class and by any **derived (child) classes**, even if those child classes are in a different project.
  - **When to Use:** When designing a class to be inherited, and you want to give child classes direct access to certain helper fields or data.
- **internal**
  - **Accessibility:** Can be accessed from any code within the **same project (assembly)**.
  - **When to Use:** For utility classes or helper fields that need to be shared within a project but should not be part of its public API.
- **protected internal**
  - **Accessibility:** A combination of protected **OR** internal. Can be accessed by any code in the same project, OR by derived classes in any other project.
- **private protected**
  - **Accessibility:** A combination of private **AND** protected. The field can be accessed only by derived classes that are declared in the **same project (assembly)**.

Interview Perspective

- **Question:** "Explain the difference between protected and private protected."

- **Ideal Answer:** "protected allows access by any derived class, regardless of which project it's in. private protected is more restrictive; it allows access only by derived classes that are located within the same project as the base class."

Lecture 3: Static Fields

Introduction

A static field belongs to the class itself, not to any individual object instance. No matter how many objects of the class you create, there is only one copy of the static field in memory, and it is shared by all of them.

How It Works: Memory

While instance fields are stored on the heap with their respective objects, a static field is stored in a special area of memory associated with the type's metadata (often on a special section of the heap called the High-Frequency Heap). It is initialized only once, before the type is first used. You access it using the ClassName.FieldName syntax.

When to Use & Real-World Examples

- **Counters:** To keep track of how many objects of a class have been created.
- **Shared Resources:** Storing a single database connection string or a logger instance that all objects of the class should share.
- **Singleton Pattern:** A design pattern where a static field holds the single instance of a class.

Example

C#

```
public class User
{
    public string name;
    public static int userCount = 0; // One counter shared by ALL User
objects.

    public User(string name)
    {
        this.name = name;
```

```
        userCount++; // Increment the shared static counter.
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine($"Initial user count: {User.userCount}"); // Access
via class name

        User user1 = new User("Alice");
        User user2 = new User("Bob");

        // The count is shared across all instances.
        Console.WriteLine($"User count after creating users:
{User.userCount}"); // Output: 2
    }
}
```

Lecture 4: Constant Fields (const)

Introduction

A const (constant) field declares a value that can never be changed. Its value is fixed at compile-time.

How It Works: Compilation

The C# compiler replaces every usage of the const field in your code with its literal value directly in the Intermediate Language (IL). It's essentially a compile-time "find and replace" operation. For this reason, a const field is implicitly static and must be initialized at the time of declaration.

When to Use & Real-World Examples

- **Mathematical Constants:** When a value is universally and eternally fixed.
  public const double PI = 3.14159;
- **Physical Constants:** public const int SpeedOfLight = 299792458;
- **Fixed Conversion Factors:** public const int SecondsInHour = 3600;

- **Application-wide Keys:** When you have a key for a dictionary or configuration that should never change. public const string ApiKeyName = "MyApp:ApiKey";

Example

C#

```
public class Configuration
{
    public const int DefaultTimeoutSeconds = 30;
}

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine($"The default timeout is
{Configuration.DefaultTimeoutSeconds} seconds.");
    }
}
```

Lecture 5: Readonly Fields (readonly)

Introduction

A readonly field's value can only be assigned once. This assignment can happen either at its declaration or within the constructor of the same class. After the constructor finishes, the field becomes "read-only" and cannot be changed.

How It Works: Initialization

Unlike const, a readonly field's value is fixed at run-time. This is a crucial difference. It allows each instance of an object to have its own unique, immutable value that is determined when the object is created.

When to Use & Real-World Examples

This is perfect for per-object values that should not change after creation.

- **Unique IDs:** public readonly Guid transactionId;
- **Creation Timestamps:** public readonly DateTime creationDate;

- **Dependency Injection:** When a service is injected via the constructor and should be used for the lifetime of the object without being replaced. private readonly ILogger _logger;

Example

C#

```
public class LogEntry
{
    public readonly Guid entryId;
    public readonly DateTime entryTime;
    public string message;

    public LogEntry(string message)
    {
        // The readonly fields are assigned here, in the constructor, at run-time.
        this.entryId = Guid.NewGuid();
        this.entryTime = DateTime.UtcNow;
        this.message = message;
    }
}
```

Lecture 6: Local Constants

Introduction

You can also declare constants that are local to a method. The const keyword, when used inside a method, creates a local variable whose value is fixed for the duration of that method call.

When to Use It

To improve readability and maintainability by giving a descriptive name to a "magic number" or "magic string" that is used within a method. It also prevents accidental modification of a value that should remain fixed.

Example

C#

```
public decimal CalculateSalesTax(decimal amount)
{
```

```
    // Instead of using a magic number like '0.0875', we declare a local
constant.
    const decimal SalesTaxRate = 0.0875M;

    return amount * SalesTaxRate;
}
```

Lecture 7: IMP Points to remember about Fields

Introduction

This lecture summarizes the most critical, practical, and interview-worthy points about working with fields in C#.

1. Fields Represent State

Remember that fields are the "nouns" or "adjectives" of your class. They hold the data that defines the current state of any given object.

2. Default to private

This is the golden rule of encapsulation. Always declare your fields as private. Expose them in a controlled manner using public methods or properties.

3. Instance vs. Static Memory

- **Instance Fields** (readonly, public, private, etc.): A copy of this field exists for **every object** you create. They live on the **heap** with their object.
- **Static Fields** (static): Only **one copy** of this field exists, and it's associated with the **class itself**. It lives in a separate memory area and is shared by all objects.

4. const vs. readonly: The Core Interview Question

You will almost certainly be asked to compare these. Be ready with a clear, multi-point answer.

When Set

Const: At **compile-time**. Must be initialized at declaration.

Readonly: At **run-time**. Can be set at declaration or in the constructor.

Scope

Const: Implicitly static. Belongs to the class.

Readonly: Can be either instance (belongs to the object) or static.

Per-Object

Const: No. One value for the entire application.

Readonly: Yes (if instance). Each object can have a different readonly value.

Types

Const: Limited to built-in value types, enums, and strings.

Readonly: Can be any type, including complex objects.

Use Case

Const: Universal, unchanging constants (e.g., Math.PI).

Readonly: Per-object immutable values set at creation (e.g., a unique TransactionID).

Section 7: Type Conversion

Lecture 1: Overview of Type Conversion

Introduction

Type conversion, also known as type casting, is the process of converting a variable from one data type to another. This is a fundamental and frequent requirement in programming. You often receive data in one format but need it in another to perform operations. For instance, you might get a user's age as string text from a console input, but you need to convert it to an int to perform mathematical calculations with it. C# is a statically-typed language, meaning the compiler must know the type of every variable. Type conversion is the mechanism that allows us to work within this system flexibly.

Categories of Conversion

There are two fundamental categories of conversion in C#, defined by whether the conversion is considered "safe" or "unsafe."

1. **Implicit Conversions (Safe Conversions):** These are conversions that are guaranteed to succeed without any loss of data. The C# compiler knows they are safe and performs them automatically for you without requiring any special syntax. This happens when you convert from a "smaller," less precise type to a "larger," more precise type.

2. **Explicit Conversions (Unsafe Conversions):** These are conversions where there is a potential risk of losing information or throwing an exception. Because the conversion is not guaranteed to succeed, the C# compiler requires you to explicitly state your intent to perform the conversion. This is you telling the compiler, "I understand the risk, but I want to do this anyway." This is achieved through three primary methods:

    - **Casting:** Using the (type) operator.
    - **Parsing:** Using helper methods like int.Parse() to convert strings.
    - **Helper Classes:** Using the System.Convert class for more controlled conversions.

We will explore each of these in detail in the following lectures.

Lecture 2: Implicit Casting

Introduction

Implicit casting is the simplest form of type conversion. It is performed automatically by the C# compiler when it can verify that the conversion is guaranteed to be safe and that no data will be lost. It's often called a widening conversion because you are moving a value into a data type that has a wider range or greater precision.

How It Works

The compiler allows this to happen automatically because the destination type can fully represent every possible value of the source type. For example, the long data type has a much larger range than the int data type. Therefore, any possible int value can fit comfortably inside a long without any risk of overflow. The compiler recognizes this and performs the conversion without requiring any special syntax from you.

When to Use It

You don't "choose" to use it; it simply happens whenever you assign a variable of a smaller compatible type to a variable of a larger type.

Common Examples

- **int to long:**
  C#

```
int numberOfUsers = 50000;

long bigNumberOfUsers = numberOfUsers; // Implicitly cast. No data loss possible.
Console.WriteLine(bigNumberOfUsers);
```

- **int to double:** An int is a whole number, while a double can hold decimals. This is safe because every whole number can be represented as a number with a .0 decimal part.

s int itemCount = 12;

double itemCountAsDouble = itemCount; // Implicitly cast. The value becomes 12.0.

Console.WriteLine(itemCountAsDouble);

```

- **float to double:** A double has greater precision (more decimal places) than a float, so this is also a safe, widening conversion.
  C#
  float price = 49.95F;

double morePrecisePrice = price; // Implicitly cast.
Console.WriteLine(morePrecisePrice);

**Interview Perspective**

- **Question:** "Why can you assign an int to a double variable directly, but not the other way around?"
- **Ideal Answer:** "You can assign an int to a double because it's an implicit, widening conversion. Every possible integer value can be safely represented as a double without any loss of data, so the compiler performs it automatically for convenience and safety. The other way around requires an explicit cast because you could lose the fractional part of the double, which is a form of data loss that the programmer must explicitly acknowledge."

**Lecture 3: Explicit Casting**

Introduction

Explicit casting is a conversion that you must manually request from the compiler. You do this when you are converting from a "larger" type to a "smaller" type, a process known as a narrowing conversion. This action is potentially unsafe because it can result in data loss or, in more complex scenarios, throw an exception at run-time.
```

How It Works: The Cast Operator (type)

You perform an explicit cast by placing the destination data type in parentheses before the variable you want to convert. By doing this, you are effectively telling the compiler: "I am aware that this conversion might lose data, but I accept the risk and I am taking responsibility for the result."

Common Scenarios and Data Loss

1. **Floating-Point to Integer (Truncation):** When you cast a double or float to an integer type like int, the entire fractional part is **truncated** (chopped off). It is **not rounded**.
   C#
   ```
   double myDouble = 9.99;

   int myInt = (int)myDouble; // You MUST use the (int) cast operator.
   Console.WriteLine(myInt);  // Output: 9. The .99 is completely lost.
   ```

2. **Larger Integer to Smaller Integer (Overflow Risk):** When you cast a larger integer type (like long) to a smaller one (like int), if the value is within the smaller type's range, it works fine. However, if the value is too large, an **overflow** will occur, resulting in a completely different and seemingly random number as the bits are reinterpreted.
   C#
   ```
   long myLong = 100;

   int myIntFromLong = (int)myLong; // This is safe.
   Console.WriteLine(myIntFromLong); // Output: 100

   long veryLargeLong = 3000000000L; // This is larger than int.MaxValue
   // The following line would result in an overflow and an incorrect value.
   // int overflowedInt = (int)veryLargeLong;
   ```

InvalidCastException

In the context of object-oriented programming, if you try to cast an object to a type that it is not compatible with, the program will compile, but it will throw an InvalidCastException at run-time.

Interview Perspective

- **Question:** "What will be the result of the following code: `double d = 9.99; int i = (int)d;`?"
- **Ideal Answer:** "The value of i will be 9. Explicitly casting a double to an int performs truncation, not rounding. The decimal part is completely discarded." This is a very common question to check if you understand the data loss mechanism.

Lecture 4: Parse

Introduction

Parsing is the specific process of converting the string representation of a value into its actual data type. This is extremely common because data from external sources—user input, text files, web responses—almost always arrives as a string. Every primitive type in C# (int, double, bool, etc.) provides a static Parse method to handle this.

How It Works

The Parse method takes a string as input and attempts to interpret it as a value of its type. For example, int.Parse() expects a string containing only digits (and possibly a leading minus sign).

The Risk: FormatException

The Parse method is "optimistic." It assumes the string will be in a valid format. If it is not, the method will fail by throwing a FormatException. If this exception is not handled (with a try-catch block, a more advanced topic), your entire application will crash.

When to Use It

You should only use Parse when you are absolutely certain that the string source is reliable and will always contain a valid value. For example, reading from a configuration file that you control, or after you have already validated the string through other means. You should never use Parse on direct user input.

Common Examples

C#

```
string numberString = "123";
int parsedInt = int.Parse(numberString); // Works perfectly. parsedInt is 123.

string boolString = "true";
bool parsedBool = bool.Parse(boolString); // Works perfectly. parsedBool is true.

string invalidNumberString = "123a";
// The following line will THROW a FormatException and crash the program if not handled.
// int errorInt = int.Parse(invalidNumberString);
```

Lecture 5: TryParse

Introduction

The TryParse method is the safe, robust, and preferred alternative to Parse for converting strings, especially when dealing with unreliable data like user input. Instead of throwing an exception upon failure, it gracefully returns a bool indicating whether the conversion succeeded or not.

How It Works: The out Parameter

The TryParse method has a very distinct signature, typically bool TryParse(string s, out T result), where T is the type.

1. It attempts to parse the input string s.
2. If the conversion is **successful**, it returns true and places the converted value into the result variable (which is passed using the out keyword).
3. If the conversion **fails**, it returns false and places the default value for the type (e.g., 0 for an int) into the result variable.
4. Crucially, **no exception is ever thrown**, allowing you to handle failures elegantly with a simple if statement.

When to Use It

Almost always. This should be your default choice for converting strings from any external source (user input, files, network data). It allows you to build robust applications that don't crash due to invalid input.

Common Example: Handling User Input

C#

```
Console.Write("Please enter your age: ");
string userInput = Console.ReadLine();

// We use the modern 'out variable' declaration here.
if (int.TryParse(userInput, out int age))
{
    // This block only executes if the conversion was successful.
    Console.WriteLine($"Success! You will be {age + 1} on your next
birthday.");
}
else
{
    // This block executes if the user typed something invalid, like "abc".
    Console.WriteLine("That is not a valid age. Please enter numbers only.");
}
```

Lecture 6: Conversion Methods (The Convert Class)

Introduction

C# provides a dedicated static helper class, System.Convert, which contains a rich set of methods for converting between the base data types. It offers more flexibility and different behaviors compared to casting or parsing.

How It Works

The Convert class provides a consistent set of methods, such as Convert.ToInt32(), Convert.ToDouble(), Convert.ToString(), and Convert.ToBoolean(). ToInt32 is the method for converting to a standard int.

Key Differences and Behaviors

1. **Rounding vs. Truncation (Critical Difference):**
    - Explicit casting (int) **truncates** (chops off) the decimal. (int)9.7 is 9.
    - The Convert class performs **rounding** to the nearest even integer (also known as "Banker's Rounding"). Convert.ToInt32(9.7) is 10. Convert.ToInt32(8.5) is 8, while Convert.ToInt32(9.5) is 10.
2. C#

```
double d = 9.7;

int castedInt = (int)d;             // Result: 9
int convertedInt = Convert.ToInt32(d); // Result: 10
```

3. **Null Handling:**
    - int.Parse(null) will throw an exception.
    - Convert.ToInt32(null) will gracefully return 0.

When to Use It

- When you are converting between different non-string types (like double to int) and you **prefer rounding behavior** over truncation.
- When you are converting from an object type and need to handle null values gracefully without a try-catch block.

Interview Perspective

- **Question:** "What are the main differences between casting a double to an int using (int) versus using Convert.ToInt32()?"
- **Ideal Answer:** "There are two key differences. First is the conversion behavior: (int) truncates the decimal portion, while Convert.ToInt32() performs rounding to the nearest even integer. Second is null handling: Convert.ToInt32(null) returns 0, whereas an explicit cast from a null object would throw an exception. Convert is generally more robust for handling varied data types."

Lecture 7: IMP Points to remember

- **Safety First: Always Prefer TryParse:** When converting strings, especially from external sources like user input, TryParse should be your default choice. It prevents your application from crashing due to FormatException and leads to cleaner code than using try-catch blocks for validation.
- **Data Loss is The Developer's Responsibility:** When you use an explicit cast (type), you are telling the compiler you are aware of and accept the risk of data loss. You must understand how that data is lost (e.g., truncation vs. overflow).
- **Know The Difference: Cast, Parse, Convert:**
    - **Cast (int)**: Fastest. Truncates decimals. Throws InvalidCastException. Use for converting between compatible numeric types when performance is critical and you want truncation.
    - **Parse()**: Converts a valid string to a type. Throws FormatException on failure. Use only when the input string is guaranteed to be valid.
    - **Convert class**: Most flexible. Rounds decimals. Handles null. Slightly slower. Use when you need rounding or need to handle a wider variety of source types gracefully.
- **Know Your Exceptions:** Understanding which action causes which exception is crucial for debugging and interviews.
    - **InvalidCastException**: Trying to cast between incompatible reference types or a null object.
    - **FormatException**: Using Parse() on a string with an invalid format.
    - **OverflowException**: Casting a value that is too large to fit into the destination type's range.
    - **ArgumentNullException**: Passing null to a method that doesn't accept it, like int.Parse(null).

Section 7: Constructors

Lecture 1: Instance Constructors

Introduction

An instance constructor is a special method within a class that is automatically called whenever an object of that class is created using the new keyword. Its primary and most critical purpose is to initialize the object's fields and put the object into a valid, consistent, and usable initial state. A constructor ensures that no object starts its life with random or null data where a specific value is required. It's the first piece of code that runs for a new object instance.

How It Works: Syntax and Execution Flow

A constructor is defined like a method but with two strict rules:

1. It must have the **exact same name as the class**.
2. It must **not have a return type**, not even void.

The process when new MyClass() is called is as follows:

1. The .NET runtime allocates a block of memory on the heap for the new object. All fields in this block are initially set to their default values (0 for numbers, false for bools, null for reference types).
2. The appropriate instance constructor is called.
3. The code inside the constructor executes, typically assigning initial values to the instance fields.
4. After the constructor completes, the reference to the newly created and initialized object is returned.

The Default Constructor

This is a critical concept often tested in interviews. If you do not define any constructor in your class, the C# compiler will automatically provide one for you behind the scenes. This is called the default constructor. It is always public and has no parameters. Its only job is to create the object and leave the fields with their default values.

However, the moment you define **any** constructor yourself (e.g., a constructor that takes parameters), the compiler **no longer provides the automatic default constructor**. If you still need a parameterless constructor in that case, you must write one explicitly.

Example: A Class With a Parameterized Constructor

C#

```
public class Car
{
    // Private instance fields
    private string _model;
    private string _color;
    private int _year;

    // This is a public, parameterized instance constructor.
    // Its job is to force every new Car object to have a model, color, and
year.
    public Car(string model, string color, int year)
    {
        Console.WriteLine("Car object is being created...");
        // Use 'this' to assign the parameter values to the private fields.
        this._model = model;
        this._color = color;
        this._year = year;
    }

    public void DisplayInfo()
    {
        Console.WriteLine($"Car Info: {_year} {_color} {_model}");
    }
}

// --- In your Main method ---
// Create an object using the constructor. We MUST provide the required
arguments.
```

```
Car myCar = new Car("Ford Mustang", "Red", 2023);
myCar.DisplayInfo(); // Output: Car Info: 2023 Red Ford Mustang

// The following line would now cause a compiler error because the default
// parameterless constructor is no longer available.
// Car anotherCar = new Car(); // ERROR! No constructor takes 0 arguments.
```

Lecture 2: Static Constructor

Introduction

A static constructor is a special constructor used to initialize static members of a class or to perform a one-time action required for the class itself, before any instances are created or any other static members are accessed.

How It Works & Rules

The behavior of a static constructor is very specific and is a common source of interview questions.

- **Syntax:** It is declared with the `static` keyword, has the same name as the class, has **no access modifiers** (it's effectively private), and **cannot have any parameters**.
- **Execution:** It is called **automatically** by the .NET runtime. You cannot call it directly from your code.
- **Timing:** It is guaranteed to be called **only once** during the lifetime of your application. This call happens at some point before the first instance of the class is created or before any static members (fields or methods) of the class are referenced. The exact timing is determined by the runtime, but the "before first use" guarantee is what matters.

When to Use & Real-World Examples

- **Initializing Complex Static Fields:** If a static field needs some logic to get its initial value (e.g., reading from a configuration file, getting a value from the operating system), you do this in the static constructor.

- **Setting up Singleton Instances:** In the Singleton design pattern, the static constructor is the perfect place to create the single, shared instance of the class.

Example

C#

```
public class AppConfig
{
    // A static field to hold a configuration value.
    public static readonly string LogFilePath;

    // A static constructor to initialize the static field.
    static AppConfig()
    {
        // This code runs only ONCE in the entire application.
        Console.WriteLine("--- Static Constructor Called: Initializing App
Settings ---");
        // In a real app, this might read from a file or the environment.
        LogFilePath = "C:\\logs\\application.log";
    }

    public static void LogMessage(string message)
    {
        Console.WriteLine($"Logging to {LogFilePath}: {message}");
    }
}

// --- In your Main method ---
Console.WriteLine("Program starting...");
AppConfig.LogMessage("First log message."); // The static constructor is
called automatically before this line.
AppConfig.LogMessage("Second log message."); // The static constructor is
NOT called again.
Console.WriteLine("Program finished.");
```

Output:

```
Program starting...
--- Static Constructor Called: Initializing App Settings ---
Logging to C:\logs\application.log: First log message.
Logging to C:\logs\application.log: Second log message.
Program finished.
```

Lecture 3: Types of Constructor

Introduction

While we've seen instance and static constructors, it's useful to categorize the different forms and patterns of constructors you will encounter and write.

- **Default Constructor**
    - **What it is:** The parameterless constructor that the C# compiler automatically generates for a class **if and only if you do not declare any other constructors**.
    - **Its Job:** To initialize all instance fields to their default system values (0, false, null, etc.).
    - **Key Point:** As soon as you write even one constructor, the automatic default constructor is gone.

- **Parameterless Constructor**
    - **What it is:** A constructor that you explicitly write yourself which takes no parameters.
    - **When to Use:** When you want to allow the creation of an object in a default, empty, or pre-defined state. If you have other parameterized constructors but still want to allow `new MyClass()`, you must provide an explicit public parameterless constructor.

- **Parameterized Constructor**
    - **What it is:** A constructor that accepts one or more parameters.
    - **When to Use:** This is the most common and useful type of constructor. It allows you to enforce that an object is created with all the necessary data to be in a valid state right from the beginning.

- **Copy Constructor**
  - **What it is:** This is a common programming **pattern**, not a special C# feature. It's a parameterized constructor that takes another object of the same class as its parameter.
  - **Its Job:** To create a new object by copying the values from the existing object, creating a duplicate.

Example:

C#

```
public class Car

{
    public string Model;
    // ... other fields

    // Parameterized constructor
    public Car(string model) { this.Model = model; }

    // Copy Constructor
    public Car(Car otherCar)
    {
        this.Model = otherCar.Model;
    }
}
Car originalCar = new Car("Mustang");
Car copiedCar = new Car(originalCar); // Creates a new object with the same data.
```

- **Static Constructor**
  - **What it is:** A special constructor used for one-time initialization of a class's static members. It's parameterless, has no access modifier, and is called automatically by the runtime.

- **Private Constructor**
  - **What it is:** A constructor declared with the private access modifier.
  - **Its Job:** Making the constructor private prevents anyone outside the class from creating an instance of it using the new keyword.

- **When to Use:** Primarily used in specific design patterns, most notably the **Singleton Pattern** (where you want to ensure only one instance of a class ever exists) or in utility classes that only contain static members (like System.Math) to prevent them from being instantiated.

Lecture 4: Constructor Overloading

Introduction

Constructor overloading is the practice of having multiple constructors in the same class, each with a different signature (i.e., a different number or type of parameters). This provides flexibility, allowing an object to be created in several different ways depending on the information the caller has available.

Why and When to Use It

Use constructor overloading to provide convenient ways to create an object. For example, you might create a User object:

- With just an email address (and generate a default username).
- With an email and a username.
- With an email, username, and a specific ID.

Instead of forcing the user to pass null or default values, you provide a dedicated constructor for each common scenario.

Constructor Chaining with : this(...)

A critical best practice when overloading constructors is to avoid duplicating initialization logic. If you have three constructors that all set the name field, and you later need to change how name is processed, you'd have to update three places.

To solve this, you use **constructor chaining**. The : this(...) syntax allows one constructor to call another constructor in the same class. The best practice is to have one "master" constructor that does all the real work, and have the simpler constructors call it, providing default values for the parameters they don't have.

Example

C#

```
public class User
{
    private string _username;
    private string _email;
    private bool _isActive;

    // 1. The simplest constructor.
    // It provides a default value for 'isActive' and calls the next constructor.
    public User(string username, string email) : this(username, email, true)
    {
        // This body is empty because the work is done by the constructor it
calls.
        Console.WriteLine("Two-parameter constructor called.");
    }

    // 2. The "master" constructor. This is where all the initialization logic
lives.
    public User(string username, string email, bool isActive)
    {
        // This is the single point of logic.
        Console.WriteLine("Master three-parameter constructor called.");
        this._username = username;
        this._email = email;
        this._isActive = isActive;
    }
}

// --- In your Main method ---
// This call matches the first constructor. It will execute, then immediately
// chain to the master constructor before its own body runs.
User user1 = new User("alice", "alice@example.com");
```

Output:

```
Master three-parameter constructor called.
Two-parameter constructor called.
```

Lecture 5: Object Initializer

Introduction

Object initializers are a feature introduced in C# 3.0 that provides a concise, readable, and flexible syntax for creating an object and setting its public properties or fields at the same time.

How It Works: Syntactic Sugar

The object initializer syntax is syntactic sugar. This means it doesn't introduce new functionality but provides a cleaner way to write code. When you write:

```
Book myBook = new Book { Title = "The Hobbit", Author = "J.R.R. Tolkien" };
```

The C# compiler translates this behind the scenes into the following equivalent code:

C#

```
Book myBook = new Book(); // 1. First, the parameterless constructor is called.
myBook.Title = "The Hobbit";   // 2. Then, a separate assignment for each property.
myBook.Author = "J.R.R. Tolkien";
```

Because the compiler must call the parameterless constructor first, a class **must have an accessible parameterless constructor** (either the automatic default one or one you've written explicitly) to be used with object initializer syntax.

Why and When to Use It

- **Flexibility:** It's perfect for creating objects when you don't want to create a specific constructor for every possible combination of properties you might want to set.
- **Readability:** It creates highly readable, declarative code, especially when creating Data Transfer Objects (DTOs) or model objects.

Example

C#

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PublicationYear { get; set; }
}

// --- In your Main method ---

// The "old" way without object initializers
Book book1 = new Book();
book1.Title = "1984";
book1.Author = "George Orwell";
book1.PublicationYear = 1949;

// The modern, clean way with an object initializer
Book book2 = new Book
{
    Title = "1984",
    Author = "George Orwell",
    PublicationYear = 1949
};
```

Lecture 6: IMP Points to Remember

- **Primary Role of Constructors:** The main purpose of a constructor is to **enforce invariants** and ensure an object is created in a **valid initial state**. A well-designed constructor guarantees that the object is usable immediately after it's created.
- **The Default Constructor Gotcha:** This is a frequent source of bugs for beginners and a good interview topic. The C# compiler only provides a public, parameterless default constructor **if you have written no other constructors**. As soon as you define even one constructor, the automatic one disappears.
- **Use Constructor Chaining for Maintainability:** Always use `: this(...)` to chain overloaded constructors. This avoids duplicating initialization logic,

reduces the chance of bugs, and makes your code easier to maintain by creating a single "master" constructor responsible for the core logic.

- **Static vs. Instance Constructors:**
    - **Instance Constructor:** Runs for **every new object** created with `new`. Used to initialize instance fields.
    - **Static Constructor:** Runs **only once** for the entire application lifetime, before the class is first used. Used to initialize `static` fields.
- **Object Initializers Require a Parameterless Constructor:** Remember that object initializer syntax (`new Book { ... }`) is syntactic sugar that first calls the parameterless constructor and then sets the properties. If your class does not have an accessible parameterless constructor, you cannot use this syntax.

Section 9: Properties & Indexers

Lecture 1: Introduction to Properties & Working with Properties

Introduction

In modern C# programming, properties are the preferred way to expose data from a class. A property is a class member that provides a flexible and controlled mechanism to read, write, or compute the value of a private field. To the outside world, a property looks and feels like a public field (myObject.Name = "Test"), but internally, it is actually a special pair of methods called accessors (get and set). This combination gives you the ease of use of a field with the power and safety of a method, perfectly embodying the principle of encapsulation.

The Problem Properties Solve: The Old Way

Before properties, if you wanted to protect a private field, you had to write explicit "getter" and "setter" methods.

C#

```
public class Student
{
    private string _name; // The private data

    // "Getter" method
    public string GetName()
    {
        return _name;
    }

    // "Setter" method
    public void SetName(string name)
    {
        if (!string.IsNullOrEmpty(name))
        {
            _name = name;
        }
    }
}
```

// Usage: student.SetName("John"); string name = student.GetName();
This works, but it's verbose and not very intuitive. Properties provide a much cleaner syntax for this exact pattern.

How They Work: Backing Fields and Accessors

A fully implemented property consists of two parts:

1. A **private backing field**: This is the actual variable that stores the data on the heap.
2. A **public property**: This contains the get and/or set accessors that control access to the backing field.

- The **get accessor** is a method with no parameters that is executed when the property's value is read. It must return a value of the property's type.
- The **set accessor** is a method that is executed when a value is assigned to the property. It has an implicit parameter named **value**, which contains the value from the right side of the assignment operator.

Example: The Modern Property Syntax

Let's refactor the Student class to use a property.

C#

```
public class Student
{
    // 1. A private "backing field" to hold the data.
    private string _name;

    // 2. A public property to control access to the backing field.
    public string Name
    {
        // The 'get' accessor is executed when you read the property.
        get
        {
            // It simply returns the value of the backing field.
            return _name;
        }

        // The 'set' accessor is executed when you assign a value to the
property.
        set
```

```
        {
            // 'value' is an implicit keyword holding the assigned value.
            // We can add validation logic here to protect our data.
            if (string.IsNullOrEmpty(value))
            {
                Console.WriteLine("Error: Name cannot be null or empty.");
            }
            else
            {
                // If valid, update the backing field.
                _name = value;
            }
        }
    }
}

// --- In your Main method ---
Student student = new Student();

// This assignment calls the 'set' accessor. The string "John" becomes the
'value' parameter.
student.Name = "John";

// This read operation calls the 'get' accessor.
string currentName = student.Name;
Console.WriteLine(currentName); // Output: John

// This assignment will trigger our validation logic.
student.Name = ""; // Output: Error: Name cannot be null or empty.
```

Lecture 2: Read-Only Properties

Introduction

A read-only property is one that can be read from outside the class but cannot be changed. This is essential for exposing data that should be immutable to the consumer of your object.

How It Works

You can create a read-only property in several ways. The most common is by simply omitting the set accessor. This means there is no mechanism for external code to assign a value to the property.

Another modern and common way is to use an **expression-bodied member**, which is a concise syntax using => for properties that have only a get accessor and return the result of a single expression.

When to Use & Real-World Examples

- **Computed Values:** When a property's value is calculated from other data in the object. A FullName property that combines FirstName and LastName is a classic example.
- **Exposing Internal State:** To allow external code to see the value of a private field without being able to change it.

Example

C#

```
public class Person
{
    private string _firstName;
    private string _lastName;
    public readonly DateTime DateOfBirth;

    public Person(string firstName, string lastName, DateTime dob)
    {
        _firstName = firstName;
        _lastName = lastName;
        DateOfBirth = dob;
    }

    // Read-only property for a computed value using expression-bodied
syntax.
    // There is no backing field for this property; its value is computed on
the fly.
    public string FullName => $"{_firstName} {_lastName}";

    // Read-only property to calculate age.
    public int Age
    {
        get
```

```
        {
            int age = DateTime.Today.Year - DateOfBirth.Year;
            if (DateTime.Today.DayOfYear < DateOfBirth.DayOfYear)
            {
                age--;
            }
            return age;
        }
    }
}

// --- In your Main method ---
Person p = new Person("Jane", "Doe", new DateTime(1990, 5, 15));
Console.WriteLine(p.FullName); // Output: Jane Doe
Console.WriteLine($"Age: {p.Age}");

// The following line would cause a compiler error because FullName is read-only.
// p.FullName = "John Smith"; // ERROR! Property or indexer 'Person.FullName' cannot be assigned to -- it is read only.
```

Lecture 3: Write-Only Properties

Introduction

A write-only property is the opposite of a read-only property: it can be written to from outside the class, but its value cannot be read back.

How It Works

You create a write-only property by providing only a set accessor and omitting the get accessor.

Why and When to Use It

Write-only properties are extremely rare and often considered a sign of questionable design because they are counter-intuitive. An object should generally be able to report its own state.

The most common (and often only valid) use case is for setting **sensitive data that should never be read back**, like a password. The set accessor would take the incoming password, hash it, and store the secure hash in a private field. There would be no get accessor to prevent anyone from ever reading the raw password.

Example

C#

```
public class PasswordManager
{
    private string _hashedPassword;

    // This is a write-only property.
    public string Password
    {
        set
        {
            if (string.IsNullOrEmpty(value) || value.Length < 8)
            {
                Console.WriteLine("Error: Password must be at least 8
characters.");
            }
            else
            {
                // In a real app, you would use a strong hashing algorithm.
                _hashedPassword = $"HASHED({value})";
                Console.WriteLine("Password has been set successfully.");
            }
        }
    }
}

// --- In your Main method ---
PasswordManager pm = new PasswordManager();
pm.Password = "MySecret123!"; // This calls the 'set' accessor.

// The following line would cause a compiler error because there is no 'get'
accessor.
// string myPass = pm.Password; // ERROR! The property
'PasswordManager.Password' cannot be used in this context because it
lacks the get accessor.
```

Lecture 4: Auto-Implemented Properties

Introduction

Auto-Implemented Properties, often called "auto-properties," are a concise C# feature that dramatically simplifies property declaration. They are used in the common scenario where a property's get and set accessors do not need any custom logic; they simply read from and write to a backing field. This feature allows you to declare the property in a single line without having to explicitly declare the private backing field yourself.

How It Works: Compiler Magic

When you use the auto-property syntax, you are giving an instruction to the C# compiler.

When you write this:

C#

```
public string Name { get; set; }
```

The compiler automatically generates this for you behind the scenes:

C#

```
private string _name_backingField; // A hidden, private backing field with a "mangled" name.

public string Name
{
    get
    {
        return this._name_backingField;
    }
    set
    {
        this._name_backingField = value;
    }
}
```

You never see or interact with this hidden backing field directly, but it is there, created by the compiler to store the property's value. This gives you the best of both worlds: a clean, one-line declaration in your code, with the proper encapsulated implementation generated for you.

When to Use It

This is now the standard and default way to create properties in C#. You should always start by creating an auto-property. Only if you later discover that you need to add validation or computation logic to the get or set accessor should you expand it to a fully implemented property with an explicit backing field.

Example

A Data Transfer Object (DTO) or a simple model class is a perfect use case for auto-properties, as they are typically just containers for data.

C#

```
public class Product
{
    // These are all auto-properties.
    // The compiler will create hidden backing fields for each one.
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Sku { get; set; }
}

// --- In your Main method ---
Product laptop = new Product();
laptop.Id = 101;
laptop.Name = "SuperBook Pro";
laptop.Price = 1299.99M;
laptop.Sku = "SBP-2025-BL";

Console.WriteLine($"Product: {laptop.Name} (ID: {laptop.Id}) costs
{laptop.Price:C}");
```

Lecture 5: Auto-Implemented Property Accessor Accessibility

Introduction

Even when using the concise syntax of auto-properties, C# allows you to specify different access modifiers for the get and set accessors individually. This gives you fine-grained control over how a property can be read and written, allowing you to create powerful and safe data patterns.

How It Works & Rules

You can add an access modifier (private, protected, internal) directly to the get or set keyword. The key rule is that the modifier on an individual accessor must be **more restrictive** than the accessibility of the property itself. For example, you can have a public property with a private set, but you cannot have a private property with a public set.

When to Use It: The private set Pattern

The most common and powerful use case for this feature is creating a property with a public getter and a private setter.

C#

```
public int Id { get; private set; }
```

This pattern creates a property that is **read-only to the outside world**, but can still be set from code **within the class itself**. This is the modern, preferred way to create fields that should be set once (typically in the constructor) and then remain immutable from an external perspective for the lifetime of the object.

Example

Let's refactor our Product class. A product's ID should never change after it has been created.

C#

```
public class Product
{
    // This property can be read by anyone...
    public int Id { get; private set; } // ...but can only be set by code inside this class.
```

```
    public string Name { get; set; }
    public decimal Price { get; set; }

    // We can set the private setter in the constructor.
    public Product(int id)
    {
        this.Id = id;
    }

    public void ChangeName(string newName)
    {
        // We can change a regular property.
        this.Name = newName;
    }
}

// --- In your Main method ---
Product phone = new Product(202);
phone.Name = "SuperPhone 12";

// You can READ the Id from outside.
Console.WriteLine($"Created product with ID: {phone.Id}");

// But you CANNOT WRITE to the Id from outside.
// The following line would cause a compiler error.
// phone.Id = 203; // ERROR! The property or indexer 'Product.Id' cannot be used in this context because the set accessor is inaccessible.
```

Interview Perspective

- **Question:** "How would you create a property that is read-only to the public but can be set internally by the class?"
- **Ideal Answer:** "I would use an auto-property with a public getter and a private setter, like `public string Name { get; private set; }`. This allows the property to be set within the class, typically in the constructor, while still appearing as read-only to any external code, which is excellent for encapsulation."

Lecture 6: Auto-Implemented Property Initializers

Introduction

This is a modern C# feature (introduced in C# 6.0) that allows you to provide a **default value** for an auto-implemented property directly at the point of its declaration. This makes code more concise and readable by removing the need to set default values inside the constructor.

How It Works

It's syntactic sugar. When you write `public string Status { get; set; } = "Active";`, the compiler ensures that when a new object of this class is created, the hidden backing field for the `Status` property will be initialized with the value `"Active"`. This happens before the constructor runs, making it a reliable way to set default states.

When to Use It

Use auto-property initializers anytime you want a property to have a sensible default value other than the system defaults (`0`, `false`, `null`). This makes your objects more robust and predictable from the moment they are created.

Example

Consider a `Customer` class where new customers should have a default status and join date.

C#

```
public class Customer
{
    public int Id { get; private set; }

    // Auto-property initializer sets a default value.
    public string Status { get; set; } = "Active";

    // This initializer calls a static method to get its default value.
    public DateTime MemberSince { get; private set; } = DateTime.UtcNow;

    public string Name { get; set; }
```

```
    public Customer(int id, string name)
    {
        this.Id = id;
        this.Name = name;
        // We no longer need to set Status or MemberSince here.
        // They are already initialized by the time the constructor runs.
    }
}

// --- In your Main method ---
Customer newCustomer = new Customer(1, "Jane Doe");

Console.WriteLine($"Customer: {newCustomer.Name}");
Console.WriteLine($"Status: {newCustomer.Status}"); // Output: Active
Console.WriteLine($"Member Since: {newCustomer.MemberSince}"); //
Output: The current date/time
```

Lecture 7: IMP Points to Remember about Properties

Introduction

This lecture consolidates the most critical, practical, and interview-worthy points about working with properties.

1. Properties are Methods in Disguise

This is the most fundamental concept to understand. A property is not a direct variable; it is **syntactic sugar** over one or two methods (get and/or set). Understanding this explains why you can add logic to them and control their accessibility. An interviewer might ask "What is the difference between a public field and a public property?" The answer is that a property provides a layer of abstraction and control through its accessors, while a public field offers no protection or logic whatsoever.

2. Auto-Properties Are The Standard

In modern C#, you should **always start with an auto-property**.

C#

public string Name { get; set; }
Only expand it to a full property with an explicit backing field if and when you need to add custom logic to the get or set accessor. This keeps your code clean and concise.

3. The public { get; private set; } Pattern is a Cornerstone of Encapsulation

This pattern is extremely common and powerful. It allows you to create properties that are publicly readable but can only be modified from within the class. This is the ideal way to handle properties like IDs, creation dates, or any state that should be immutable to the outside world after an object is created.

4. Properties vs. Methods: A Guideline

An interview question might be, "When do you use a property versus a method?" Here's a good guideline:

- **Use a Property if:**
    - The member represents a piece of data or an attribute of the object (e.g., Name, Color, Age).
    - Getting the value is a fast, simple operation that does not have significant side effects.
- **Use a Method if:**
    - The member performs an action or a verb (e.g., Save(), Calculate(), Convert()).
    - The operation is complex, slow, or has noticeable side effects.
    - The operation takes parameters. (Properties do not, but indexers do).

Lecture 8: Intro to Indexers & Working with Indexers

Introduction

An **indexer** is a special type of class member that allows an object of your class to be accessed using the familiar square bracket [] notation, just like an array or a dictionary. Its purpose is to provide a more natural and intuitive syntax for accessing elements when your class logically represents a collection or list of items. Instead of writing myTeam.GetPlayer(0), an indexer allows you to write the much cleaner myTeam[0].

How It Works: Syntax and Mechanics

An indexer looks very similar to a property, but with two key differences:

1. It is declared using the this keyword as its name.
2. It takes one or more parameters inside the square brackets, which become the "index" or "key".

Like a property, an indexer has get and set accessors.

- The **get accessor** is executed when you read from the indexer (e.g., var player = myTeam[0];). The index value (0) is passed into the indexer's parameter.
- The **set accessor** is executed when you assign a value to the indexer (e.g., myTeam[0] = "New Player";). It receives both the index value (0) and the assigned value (in the implicit value keyword).

When to Use It

You should implement an indexer when the primary purpose of your class is to encapsulate and manage an internal collection (like an array, list, or dictionary). It provides a simplified "front-end" for a more complex "back-end" storage mechanism.

Example

Let's create a Team class that internally stores player names in a List<string>. We will expose access to this list through an indexer, adding our own bounds-checking logic for safety.

C#

```
using System;
using System.Collections.Generic;

public class Team
{
    // The private backing collection.
    private List<string> _players = new List<string>();

    // The indexer declaration. It takes an integer 'index'.
    public string this[int index]
    {
        // The 'get' accessor for reading a player's name.
        get
        {
            // Add validation to prevent errors.
            if (index >= 0 && index < _players.Count)
            {
                return _players[index];
            }
            // Return null or throw an exception if the index is out of bounds.
            return null;
        }

        // The 'set' accessor for adding or updating a player's name.
        set
        {
            if (index >= 0 && index < _players.Count)
            {
                // If the index is valid, update the existing player.
                _players[index] = value;
            }
            else if (index == _players.Count)
            {
                // A special rule: if the index is at the end, add a new player.
                _players.Add(value);
            }
        }
    }

    public int PlayerCount => _players.Count;
}

// --- In your Main method ---
```

```
Team myTeam = new Team();

// Use the 'set' accessor of the indexer to add players.
myTeam[0] = "Alice";
myTeam[1] = "Bob";
myTeam[2] = "Charlie";

// Use the 'get' accessor of the indexer to read player names.
Console.WriteLine($"Player at index 1 is: {myTeam[1]}"); // Output: Bob

// Update a player using the setter.
myTeam[0] = "Alicia";
Console.WriteLine($"Player at index 0 is now: {myTeam[0]}"); // Output:
Alicia

// Access an invalid index.
Console.WriteLine($"Player at index 5 is: {myTeam[5] ?? "Not Found"}"); //
Output: Not Found
```

Lecture 9: Indexer Overloading

Introduction

Just like methods, indexers can be **overloaded**. This means you can declare multiple indexers in the same class, as long as their signatures are different. The signature of an indexer is determined by the **number and/or type of its parameters** (the values inside the square brackets).

How It Works

The C# compiler determines which indexer to call based on the data type of the index you provide in the square brackets. This allows you to provide multiple, intuitive ways to look up data in your encapsulated collection. For example, you can have one indexer to look up an item by its integer position and another to look up the same item by its string name or key.

When to Use It

Indexer overloading is perfect for classes that represent more complex collections where items can be identified in multiple ways. A common pattern is to have an `int` indexer for positional access and a `string` indexer for key-based lookup, making the class behave like both an array and a dictionary.

Example

Let's enhance our `Team` class. Instead of just storing names, it will now store `Player` objects. We will provide an `int` indexer to get a player by their position and a `string` indexer to look up a player by their name.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Player
{
    public string Name { get; set; }
    public int JerseyNumber { get; set; }
}

public class Team
{
    private List<Player> _players = new List<Player>();

    // Method to add players to the team
    public void AddPlayer(Player player)
    {
        _players.Add(player);
    }

    // --- Indexer Overload 1: Takes an integer index ---
    public Player this[int index]
    {
        get
        {
            if (index >= 0 && index < _players.Count)
            {
                return _players[index];
            }
            return null;
```

```
        }
    }

    // --- Indexer Overload 2: Takes a string name ---
    public Player this[string name]
    {
        get
        {
            // Use LINQ to find the first player with a matching name.
            return _players.FirstOrDefault(p => p.Name.Equals(name,
StringComparison.OrdinalIgnoreCase));
        }
    }
}

// --- In your Main method ---
Team myTeam = new Team();
myTeam.AddPlayer(new Player { Name = "Alice", JerseyNumber = 10 });
myTeam.AddPlayer(new Player { Name = "Bob", JerseyNumber = 7 });

// Using the INT indexer
Player firstPlayer = myTeam[0];
Console.WriteLine($"Player at index 0 is {firstPlayer.Name}, Jersey
#{firstPlayer.JerseyNumber}");

// Using the STRING indexer
Player bob = myTeam["Bob"];
Console.WriteLine($"Looked up Bob by name. Jersey #{bob.JerseyNumber}");

Player charlie = myTeam["Charlie"];
Console.WriteLine($"Is Charlie on the team? {charlie == null}");
```

Lecture 10: IMP Points to Remember about Properties & Indexers

Introduction

This lecture consolidates the most critical, practical, and interview-worthy points about both properties and indexers.

1. Properties are Methods in Disguise

This is the most fundamental concept. A property is not a direct variable; it is **syntactic sugar** over one or two methods (get and/or set). This explains why you can add logic to them, control their accessibility, and why they are central to encapsulation.

2. Auto-Properties Are The Standard

In modern C#, you should **always start with an auto-property** (public string Name { get; set; }). Only expand it to a full property with an explicit backing field if and when you need to add custom logic to the get or set accessor.

3. The public { get; private set; } Pattern is a Cornerstone of Encapsulation

This pattern is extremely common and powerful. It allows you to create properties that are publicly readable but can only be modified from within the class, making it ideal for IDs or state that should be immutable to the outside world after an object is created.

4. Indexers Provide "Array-Like" Syntax

The entire purpose of an indexer is to make an object that represents a collection more intuitive to use. It's syntactic sugar that makes your class feel like a built-in array or dictionary.

5. Properties vs. Indexers: A Key Interview Topic

Be prepared to clearly articulate the differences.

- **Access:** Properties are accessed by a **static name** known at compile time (myObject.Name). Indexers are accessed by a **dynamic index or key** whose value is often not known until run-time (myObject[key]).
- **Static Members:** Properties can be declared as static (e.g., DateTime.Now). Indexers must always be **instance members**; they operate on a specific object.
- **Parameters:** Properties do not take parameters in their call. Indexers **always** take at least one parameter (the index).

Section 10: Inheritance

Lecture 1: Intro to Inheritance

Introduction

Inheritance is a fundamental pillar of Object-Oriented Programming (OOP) that allows us to create a new class (known as the **derived** or **child** class) that acquires the properties and behaviors of an existing class (known as the **base** or **parent** class). This mechanism models a hierarchical **"is-a" relationship**. For example, a Manager "is an" Employee, a Car "is a" Vehicle, and a Dog "is an" Animal.

The primary purpose and benefit of inheritance is **code reuse**. Instead of duplicating common logic and data fields across multiple similar classes, we can place all the shared functionality into a single base class. The derived classes then automatically inherit this functionality and can add their own unique features or modify the inherited behaviors. This leads to a more organized, maintainable, and logical code structure.

How It Works: Syntax and Member Access

You establish an inheritance relationship in C# using a colon (:) in the class declaration.

Syntax:

C#

```
public class BaseClass
{
    // ... members ...
}

public class DerivedClass : BaseClass
{
    // ... additional members ...
}
```

When DerivedClass inherits from BaseClass, it automatically gains access to all of BaseClass's public and protected members. It's crucial to remember that

private members of the base class are **not** inherited; they remain completely hidden and inaccessible to the derived class, preserving the base class's encapsulation.

Example

Let's model a simple hierarchy. All employees in a company have a name and an ID. A manager is a specific type of employee who also has a team size.

C#

```
// The base class with common fields and methods.
public class Employee
{
    public string Name { get; set; }
    public int EmployeeId { get; set; }

    public void Work()
    {
        Console.WriteLine($"{Name} is working.");
    }
}

// The derived class inherits from Employee.
// A Manager "is-an" Employee.
public class Manager : Employee
{
    // A property specific only to the Manager class.
    public int TeamSize { get; set; }
}

// --- In your Main method ---
Manager manager = new Manager();

// Accessing members inherited from the Employee class.
manager.Name = "Jane Doe";
manager.EmployeeId = 101;
manager.Work(); // This method was defined in the Employee class.

// Accessing members specific to the Manager class.
manager.TeamSize = 12;
Console.WriteLine($"{manager.Name} manages a team of
{manager.TeamSize}.");
```

Lecture 2: Types of Inheritance

Introduction

While inheritance is a broad concept, C# has specific rules about how it can be implemented. Understanding these supported (and unsupported) types is crucial for designing correct class hierarchies.

Supported Types of Inheritance in C#

- **Single Inheritance**
    - **What it is:** A class can only inherit directly from **one** other class. This is a strict rule in C#.
    - **Why:** This rule exists to prevent the **"Diamond Problem"**—a situation that can occur in languages that allow multiple inheritance. If a ClassD inherits from both ClassB and ClassC, and both B and C inherit from a common ClassA that has a method DoWork(), which version of DoWork() should ClassD inherit? C#'s single inheritance model completely avoids this ambiguity.
- **Multi-level Inheritance**
    - **What it is:** A class can be derived from another class, which is itself derived from another class. This creates a chain of inheritance.
    - **Example:**
      C#
      public class Animal { }

      public class Mammal : Animal { }
      public class Dog : Mammal { } // Dog inherits from Mammal and Animal.
    -
- **Hierarchical Inheritance**
    - **What it is:** Multiple classes inherit from a single base class. This is a very common structure.
    - **Example:**
      C#
      public class Vehicle { }

```
public class Car : Vehicle { }
public class Motorcycle : Vehicle { }
public class Bus : Vehicle { }
```

Unsupported Types of Inheritance in C#

- **Multiple Inheritance (for classes)**
    - **What it is:** The ability for a single class to inherit from more than one base class directly.
    - **C# Status:** As mentioned, C# **does not** support this for classes.
    - **The Alternative:** This functionality is achieved in C# through the use of **Interfaces**. A class can inherit from only one base class, but it can implement multiple interfaces.

Interview Perspective

- **Question:** "Does C# support multiple inheritance?"
- **Ideal Answer:** "No, C# does not support multiple inheritance for classes, primarily to avoid issues like the Diamond Problem. It enforces a single inheritance model. However, similar functionality can be achieved by a class implementing multiple interfaces."

Lecture 3: `base` Keyword

Introduction

The `base` keyword is a special reference, available only within a derived class, that allows you to access members of its direct **base (parent) class**. It's your way of reaching "up" the inheritance chain to use functionality from the parent that might have been hidden or needs to be augmented in the child class.

Why and When to Use It

1. **To call a method in the base class that has been hidden or overridden** in the derived class.
2. **To call a specific constructor of the base class** from a derived class's constructor (covered in the next lecture).

Example: Calling a Base Class Method

Let's imagine a Report base class and a more specific FinancialReport class. Both will have a Print() method. The financial report needs to print the standard header first, then its own specific content.

C#

```
public class Report
{
    public string Title { get; set; }

    public virtual void Print() // Marked as virtual for potential overriding
    {
        Console.WriteLine("--- Report Header ---");
        Console.WriteLine($"Title: {Title}");
        Console.WriteLine("----------------------");
    }
}

public class FinancialReport : Report
{
    public decimal Revenue { get; set; }

    public override void Print() // Overriding the base method
    {
        // 1. Use 'base' to call the Print() method from the Report class first.
        base.Print();

        // 2. Then, add the specific functionality of the derived class.
        Console.WriteLine($"Revenue: {Revenue:C}");
        Console.WriteLine("--- End of Financials ---");
    }
}

// --- In your Main method ---
FinancialReport report = new FinancialReport();
```

```
report.Title = "Q2 Financials";
report.Revenue = 500000;
report.Print();
```
Output:

```
--- Report Header ---
Title: Q2 Financials
--------------------
Revenue: $500,000.00
--- End of Financials ---
```

Lecture 4: Parent Class's Constructor

Introduction

When you create an object of a derived class, the .NET runtime ensures that the state of the entire object, including the part inherited from the base class, is properly initialized. This means a constructor from the **base class is always called before** the constructor of the derived class runs.

How It Works: The Constructor Chain

1. When you write `new DerivedClass()`, the runtime looks at the `DerivedClass` constructor.
2. Before executing the body of the `DerivedClass` constructor, it first calls a constructor from the `BaseClass`.
3. **By default, it will try to call the public, parameterless constructor of the base class.**
4. After the base class constructor completes, the body of the derived class constructor is executed.

The : base(...) Syntax

What if the base class does not have a parameterless constructor? Or what if you need to call a specific parameterized constructor on the base class to pass up required data?

In this case, you **must** use the : base(...) syntax in the derived class's constructor definition. This explicitly tells the compiler which base class constructor to call and what arguments to pass to it.

Example

Let's make our Employee class require a name and ID upon creation.

C#

```
public class Employee
{
    public string Name { get; private set; }
    public int EmployeeId { get; private set; }

    // The base class ONLY has a parameterized constructor.
    public Employee(string name, int employeeId)
    {
        Console.WriteLine("Base Employee constructor called.");
        this.Name = name;
        this.EmployeeId = employeeId;
    }
}

public class Manager : Employee
{
    public int TeamSize { get; set; }

    // The Manager constructor must now call the Employee constructor.
    // We use the ': base(...)' syntax to pass the values up the chain.
    public Manager(string name, int employeeId, int teamSize) : base(name,
employeeId)
    {
        Console.WriteLine("Derived Manager constructor called.");
        // The base constructor has already set the name and ID.
        // We only need to set the field specific to this class.
        this.TeamSize = teamSize;
    }
}
```

```
// --- In your Main method ---
Manager manager = new Manager("Jane Doe", 101, 12);
```
Output:

```
Base Employee constructor called.
Derived Manager constructor called.
```
Notice the order of execution. The base constructor runs first.

Interview Perspective

- **Question:** "What happens if a base class only has a private constructor?"
- **Ideal Answer:** "If a class only has private constructors, it cannot be inherited from. A derived class would not be able to call the base constructor, which is a requirement, so it would result in a compile-time error."

Lecture 5: Method Hiding

Introduction

Method hiding, also known as **shadowing**, occurs when a derived class defines a method with the **exact same name and signature** as a method in its base class, but without using the `override` keyword. By doing this, the derived class's method **hides** the base class's method. This is not a way to change the base class's behavior; it's a way to provide a completely new, separate method that happens to share the same name.

How It Works: The `new` Keyword and Compile-Time Resolution

While not strictly required, you should always use the `new` keyword on the derived class's method to explicitly tell the compiler that you intend to hide the base method. This suppresses a compiler warning and makes your intent clear.

The most critical concept to understand about method hiding is that the version of the method that gets called is determined at **compile-time**, based on the **type of the reference variable** you are using, not the actual type of the object stored in that variable.

Example

Let's see how the reference type dictates the outcome.

C#

```
public class Animal
{
    public void PrintInfo()
    {
        Console.WriteLine("This is an Animal.");
    }
}

public class Dog : Animal
{
    // 'new' explicitly hides the base class PrintInfo() method.
    public new void PrintInfo()
    {
        Console.WriteLine("This is a Dog.");
    }
}

// --- In your Main method ---
// 1. Create a Dog object and hold it in a Dog reference.
Dog dogReference = new Dog();
// The compiler sees the reference type is 'Dog', so it binds the call to
Dog.PrintInfo().
dogReference.PrintInfo(); // Output: This is a Dog.

Console.WriteLine("--------------------");

// 2. Create a Dog object but hold it in an Animal reference.
Animal animalReference = new Dog();
// The compiler sees the reference type is 'Animal'. Even though the object
// is a Dog, the compiler binds the call to Animal.PrintInfo().
animalReference.PrintInfo(); // Output: This is an Animal.
```

Interview Perspective

- **Question:** "What is method hiding in C#?"
- **Ideal Answer:** "Method hiding is when a derived class defines a method with the same signature as a base class method, using the `new` keyword. This creates a new method that is separate from the base method. The key characteristic is that the method call is resolved at compile-time based on the type of the reference variable, not the run-time type of the object."

Lecture 6: Method Overriding

Introduction

Method overriding is a fundamental technique for achieving **run-time polymorphism**. It allows a derived class to provide a new, specific implementation for a method that is already defined in its base class. Unlike hiding, which creates a new method, overriding provides a replacement implementation for the existing base method within the inheritance hierarchy.

How It Works: The `virtual` and `override` Contract

Method overriding requires a contract between the base and derived classes using two essential keywords:

1. **`virtual` (in the base class):** The method in the base class must be marked with the `virtual` keyword (or `abstract` or `override` itself). This signals, "I am aware that a derived class might want to provide its own version of this method."
2. **`override` (in the derived class):** The method in the derived class must be marked with the `override` keyword and must have the exact same signature as the virtual method in the base class. This signals, "I am intentionally replacing the implementation of the base class's virtual method."

Run-Time Polymorphism

This is the critical difference from method hiding. With overriding, the version of the method that gets called is determined at **run-time**, based on the **actual type of the object stored on the heap**, regardless of the type of the reference variable pointing to it.

Example: A Classic Polymorphism Scenario

C#

```
public class Animal
{
    // Mark this method as 'virtual' so it can be overridden.
    public virtual void MakeSound()
    {
        Console.WriteLine("The animal makes a generic sound.");
    }
}

public class Dog : Animal
{
    // Use 'override' to provide a specific implementation for Dog.
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

public class Cat : Animal
{
    // Use 'override' to provide a specific implementation for Cat.
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
    }
}

// --- In your Main method ---
// We create a list that holds 'Animal' references.
List<Animal> pets = new List<Animal>();

// We can add Dog and Cat objects because they "are" Animals.
pets.Add(new Dog());
pets.Add(new Cat());
pets.Add(new Animal());

// Now, we loop through the list and call the same method on each object.
foreach (Animal pet in pets)
{
```

```
    // At run-time, the CLR checks the actual type of the object 'pet' points
to
    // and calls the appropriate overridden method. This is polymorphism.
    pet.MakeSound();
}
```
Output:

```
Woof!
Meow!
The animal makes a generic sound.
```

Lecture 7: Sealed Classes

Introduction

The sealed keyword is a modifier that, when applied to a class, **prevents any other class from inheriting from it**. It effectively stops the inheritance chain at that point, marking the class as final and not designed for further extension.

How It Works

This is a simple compile-time restriction. If you attempt to derive from a class that has been marked as sealed, the C# compiler will generate an error.

C#

```
// The 'sealed' keyword prevents inheritance.
public sealed class FinalConfiguration
{
    // ... members ...
}

// The following line will cause a compiler error:
// 'MyConfig': cannot derive from sealed type 'FinalConfiguration'
public class MyConfig : FinalConfiguration // ERROR!
{
    // ...
}
```

Why and When to Use It

1. **Design Intent:** To clearly communicate that a class is "complete" and not designed for extension. It prevents misuse of a class that wasn't built to be a parent. The most famous example in the .NET framework is the System.String class, which is sealed.
2. **Security:** To protect classes with security-sensitive logic. By sealing the class, you prevent other developers from deriving from it and overriding its behavior in a way that could bypass security checks.
3. **Performance:** There can be minor performance benefits. When the Just-In-Time (JIT) compiler encounters a method call on a sealed class instance, it knows that the method can never be an overridden version. This allows it to perform optimizations like **devirtualization**, where it can make a more direct and faster method call.

Lecture 8: IMP Points to Remember

Introduction

This lecture summarizes the most critical, practical, and interview-worthy points about inheritance in C#.

1. Inheritance Models an "is-a" Relationship

This is the core conceptual model. Use inheritance when one class truly is a more specific version of another. Don't force an inheritance relationship where it doesn't naturally fit.

2. C# Supports Single Inheritance for Classes

A class can only have one direct base class. This is a fundamental rule to avoid ambiguity. Multiple inheritance of behavior is achieved by implementing multiple interfaces.

3. Base Class Constructors Always Run First

When you instantiate a derived class, a base class constructor is always executed before the derived class constructor. Use the : base(...) syntax to explicitly control which base constructor is called and to pass necessary data up the inheritance chain.

4. Method Hiding vs. Method Overriding: The Core Interview Question

This is one of the most important topics in C# OOP interviews. You must be able to clearly articulate the differences.

- **Keywords:**
    - **Hiding:** Uses the new keyword on the derived method (optional, but a best practice). The base method does not need any special keyword.
    - **Overriding:** Requires the virtual (or abstract) keyword on the base method and the override keyword on the derived method.
- **Method Resolution (The Key Difference):**
    - **Hiding:** The method to be called is determined at **compile-time** based on the **type of the reference variable**.
    - **Overriding:** The method to be called is determined at **run-time** based on the **actual type of the object** stored on the heap. This is run-time polymorphism.
- **Purpose:**
    - **Hiding:** You are introducing a **new, separate** method in the derived class that happens to have the same name. It does not replace the base method; it just hides it.
    - **Overriding:** You are providing a **replacement implementation** for an existing base method, honoring the polymorphic contract set by the virtual keyword.

5. sealed Provides Finality

Use the sealed keyword to explicitly prevent a class from being inherited. This communicates design intent, can improve security, and may offer minor performance optimizations.

Section 12: Namespaces

Lecture 1: Intro to Namespaces

Introduction

A **namespace** is a fundamental organizational mechanism in C# used to group related types (like classes, interfaces, and structs) under a single, hierarchical name. Its primary purposes are to manage complexity in large applications and, most importantly, to **prevent naming conflicts**.

Think of namespaces like a folder system on your computer. You can have two different files named `Report.txt` without any issues, as long as they reside in different folders (e.g., `C:\Work\Report.txt` and `C:\Home\Report.txt`). Similarly, in C#, you can have two different classes named `Logger` as long as they are in different namespaces (e.g., `MyCompany.Logging.Logger` and `ThirdParty.Analytics.Logger`). Without namespaces, it would be impossible to combine code from different teams or libraries without the risk of class names clashing.

How It Works: Fully Qualified Names

Every type in the .NET ecosystem has a **fully qualified name**, which includes its full namespace path followed by the type name. For example, the `string` type we use is actually a shorthand for its fully qualified name: `System.String`.

When you are working within a specific namespace, you can access other types in that same namespace directly. However, to access a type located in a different namespace, you **must** use its fully qualified name.

Example

Let's create two separate namespaces in the same project to see how this works.

C#

```
// --- File 1: Defines a utility class in its own namespace ---
namespace MyProject.Utilities
{
    public class TextHelper
    {
        public void SayHello()
        {
            // We can use 'Console' directly here because its fully qualified
            // name, System.Console, is globally available.
            System.Console.WriteLine("Hello from the TextHelper!");
        }
    }
}

// --- File 2: The main program, in a different namespace ---
namespace MyProject.MainApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // To use the TextHelper class, which is in a different namespace,
            // we MUST use its fully qualified name.
            MyProject.Utilities.TextHelper helper = new
MyProject.Utilities.TextHelper();

            helper.SayHello();
        }
    }
}
```

Lecture 2: Nested Namespaces

Introduction

To create an even more granular and logical organization, namespaces can be **nested** inside other namespaces. This allows you to create a clear hierarchy that mirrors the structure of your application or company.

How It Works

You create a nested namespace using the dot (.) notation. For example, the namespace MyCompany.Billing.Invoicing implies a hierarchy: Invoicing is part of the Billing system, which belongs to MyCompany. This is just like having a folder named Invoicing inside a folder named Billing. This deep organization is essential for very large enterprise applications where thousands of classes must be managed.

Example

Let's expand on our previous example with a nested namespace structure.

C#

```
namespace MyCompany.BusinessLogic.Billing
{
    // This class lives inside a nested namespace.
    public class InvoiceGenerator
    {
        public void CreateInvoice()
        {
            System.Console.WriteLine("Invoice created successfully.");
        }
    }
}

namespace MyCompany.MainApp
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
        // To access the class in the nested namespace, we must provide
the
        // entire path from the top-level namespace down.
        MyCompany.BusinessLogic.Billing.InvoiceGenerator generator =
            new MyCompany.BusinessLogic.Billing.InvoiceGenerator();

        generator.CreateInvoice();
    }
  }
}
```

Lecture 3: Importing Namespaces

Introduction

Typing the fully qualified name for every class you use is tedious and makes code difficult to read. The `using` directive is the C# feature that solves this problem. By placing a `using` directive at the top of your code file, you are telling the compiler, "For this file, I will be frequently using types from the specified namespace, so please recognize them by their short names."

How It Works

When the compiler encounters a type name in your code (like `List` or `Logger`), it first checks if that type is defined in the current namespace. If not, it then searches through all the namespaces that you have imported with `using` directives at the top of the file. As soon as it finds a match, it resolves the type. This removes the need for you to write the fully qualified name everywhere.

Example

Let's refactor our previous example to be much cleaner using the `using` directive.

C#

```
// --- The other class files remain the same ---

// --- In our main program file ---
```

```
// We import the namespaces at the top of the file.
using System;
using MyProject.Utilities;
using MyCompany.BusinessLogic.Billing;

namespace MyProject.MainApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Now the code is much cleaner. The compiler knows where to find
            // these types because of the 'using' directives above.
            Console.WriteLine("Application starting...");

            TextHelper helper = new TextHelper();
            helper.SayHello();

            InvoiceGenerator generator = new InvoiceGenerator();
            generator.CreateInvoice();
        }
    }
}
```

Best Practice: Always place your `using` directives at the very top of your `.cs` file, before the `namespace` declaration.

Lecture 4: Using Alias

Introduction

A **using alias** is a special form of the `using` directive that creates a new, often shorter, name for a specific type. It does not import an entire namespace, but instead creates an alias for a single class, struct, or other type.

Why and When to Use It

1. **To Resolve Naming Conflicts (Primary Use Case):** This is the most important reason. If you import two different namespaces that both contain a class with the same name, the compiler won't know which one you mean. This ambiguity results in a compile-time error. An alias allows you to create a unique name for one or both of the conflicting types.
2. **To Shorten Long Generic Type Names:** If you frequently use a very long and complex generic type, you can create a simple alias for it to make your code much more readable.

Syntax

```
using AliasName = Fully.Qualified.Namespace.TypeName;
```

Example: Resolving a Naming Conflict

The .NET framework has two different Timer classes in two different namespaces. Let's see how to use both in the same file without conflict.

C#

```
// Create an alias for each Timer class to resolve the name clash.
using SystemTimersTimer = System.Timers.Timer;
using SystemThreadingTimer = System.Threading.Timer;

public class TimerManager
{
    public void StartTimers()
    {
        // Now we can use the unique alias names without ambiguity.
        SystemTimersTimer myServiceTimer = new SystemTimersTimer(1000);
        myServiceTimer.Start();
        Console.WriteLine("System.Timers.Timer started.");

        // The 'state' and 'dueTime' parameters are just for the example.
        SystemThreadingTimer myThreadingTimer = new
SystemThreadingTimer(_ => {}, null, 0, 2000);
        Console.WriteLine("System.Threading.Timer started.");
    }
}
```

Lecture 5: using static

Introduction

The using static directive is a feature that allows you to import the **static members** (methods, properties, and fields) of a single class directly into the current scope. This allows you to call those static methods without having to prefix them with the class name.

How It Works

When you write using static System.Math; at the top of your file, the compiler becomes aware of all the public static members of the Math class. When it later sees a method call like Sqrt(25), it will search the imported static classes and find that Sqrt is a static method of Math, resolving the call correctly.

Why and When to Use It

Use this feature sparingly to improve readability when you are making **frequent calls** to the static members of a single class within a file.

- **Good Candidates:** System.Math in a class with heavy calculations, System.Console in a simple console application, or a custom Constants class you've written.
- **Caution:** Overusing using static can sometimes make code less clear, as it might not be immediately obvious where a method like Log() or Calculate() is coming from if its class name is omitted.

Example

C#

```
// Import all static members from the Console and Math classes.
using static System.Console;
using static System.Math;

class Program
{
    static void Main(string[] args)
    {
        // We can now call WriteLine, PI, and Sqrt directly
        // without prefixing them with Console or Math.

        double radius = 10;
```

```
        double area = PI * Pow(radius, 2); // Pow is also from System.Math

        WriteLine($"The area of a circle with radius {radius} is {area:F2}.");

        double root = Sqrt(144);
        WriteLine($"The square root of 144 is {root}.");
    }
}
```

Lecture 6: IMP Points to Remember

- **Primary Purpose: Organization and Conflict Avoidance:** This is the core takeaway. Namespaces are C#'s primary tool for keeping large projects organized and for preventing class name collisions when you use multiple libraries.
- **Every .NET Type Lives in a Namespace:** Remember that nothing exists in a "global" space. Even the simplest `string` is just a C# alias for the `System.String` class. `int` is an alias for `System.Int32`. Understanding this helps you see the .NET framework as one giant, well-organized library.
- **`using` Directive vs. `using` Alias:**
    - A `using` directive (`using System.Collections.Generic;`) imports **all types** from a namespace, allowing you to use them by their short names.
    - A `using` alias (`using UserList = System.Collections.Generic.List<MyApp.User>;`) creates a new, convenient name for **a single, specific type**. Use it to resolve conflicts or simplify long type names.
- **File-Scoped Namespaces (Modern C#):** In C# 10 and later, you can declare a namespace for an entire file without curly braces: `namespace MyCompany.MyProject;`. This reduces nesting and cleans up your files.
- **`global using` (Modern C#):** Also introduced in C# 10, the `global using` directive allows you to define a `using` statement in one file (often a dedicated `GlobalUsings.cs` file) that applies to **your entire project**. This is extremely useful for importing commonly used namespaces like `System`, `System.Linq`, or `System.Collections.Generic` across all your files without having to repeat the directives in every single file.

Section 13: Partial Classes, Static Classes, Enumerations

Lecture 1: Partial Classes

Introduction

A **partial class** is a special feature in C# that allows the definition of a single class to be split across **multiple physical .cs files**. When you compile your application, the C# compiler finds all the separate parts of the partial class and combines them into a single, complete class. This is purely a compile-time feature; at run-time, there is no difference between a regular class and a partial class.

How It Works: The `partial` Keyword

To create a partial class, you use the `partial` keyword just before the `class` keyword in each file where a part of the class is defined. All parts must:

- Use the `partial` keyword.
- Have the exact same class name.
- Be in the same namespace.
- Be in the same project (assembly).

The compiler will merge all members—fields, properties, methods, etc.—from all partial files into the final class definition.

Why and When to Use It

1. **Working with Source Generators (Primary Use Case):** This is the most important and common use of partial classes in modern C#. Many .NET frameworks (like ASP.NET Core, WPF, WinForms) use code generation tools. These tools automatically generate code based on your definitions (e.g., from a UI design or a web endpoint). They place this machine-generated code in one part of a partial class (e.g., MyPage.g.cs). This allows you, the developer, to write your own logic in a separate file for the same class (e.g., MyPage.cs). This separation is crucial because it means you can freely modify your code without worrying that the code generator will overwrite your changes when it runs again.
2. **Managing Very Large Classes:** In some legacy systems or extremely large classes, the partial keyword can be used to split the class into multiple files for better organization. For example, a massive class could have its fields and properties in one file, its public methods in another, and its private helper methods in a third.
    - **Best Practice Note:** While this is possible, a class that is so large it needs to be split often indicates a violation of the **Single Responsibility Principle**. It's usually a sign that the class is doing too much and should be refactored into several smaller, more focused classes.

Example

Let's split a User class definition across two files.

File 1: User.Data.cs

C#

```
namespace MyWebApp
{
    // The first part of the User class, focusing on data.
    public partial class User
    {
        public int Id { get; set; }
        public string Username { get; set; }
        public string Email { get; set; }
    }
}
```

File 2: User.Logic.cs

C#

```
namespace MyWebApp
{
    // The second part of the User class, focusing on behavior.
    public partial class User
    {
        public void SendWelcomeEmail()
        {
            // This method can access the 'Email' property defined in the other
file.
            Console.WriteLine($"Sending welcome email to {this.Email}...");
        }
    }
}
```

At compile time, these two parts are combined into a single User class that has all four members.

Lecture 2: Partial Methods

Introduction

A **partial method** is a special kind of method whose definition can be separated from its implementation. The signature (the declaration) of the method can be defined in one part of a partial class, and the implementation (the method body) can be optionally provided in another part of the same partial class.

How It Works & The "Magic" of Unimplemented Methods

A partial method is declared with the partial keyword, must have a void return type, and is implicitly private.

File 1 (e.g., a generated file): The Declaration

C#

```
public partial class MyComponent
{
    // This defines the SIGNATURE of the partial method. It's a "hook".
```

```
    partial void OnNameChanged(string newName);

    public void SetName(string name)
    {
        // ... logic to set the name ...

        // The generator can safely CALL the partial method.
        OnNameChanged(name);
    }
}
```

The key feature: If no other part of the MyComponent class provides an implementation for OnNameChanged, the C# compiler **completely removes the method declaration AND all calls to it**. The call OnNameChanged(name); is erased from the final compiled code. This means there is **zero performance overhead** for an unimplemented partial method.

Why and When to Use It

This feature is used almost exclusively by **source generators**. The generator defines a partial method as an "optional hook" that the developer can choose to implement. It allows the generated code to provide customizable entry points without forcing the developer to implement them if they are not needed.

Example: Implementing the Partial Method

File 2 (e.g., your handwritten file): The Optional Implementation

C#

```
public partial class MyComponent
{
    // We provide the IMPLEMENTATION for the partial method here.
    // Because this exists, the compiler will keep the call in SetName().
    partial void OnNameChanged(string newName)
    {
        Console.WriteLine($"Notification: Name was changed to '{newName}'.");
    }
}
```

Lecture 3: Static Classes

Introduction

A **static class** is a class that cannot be instantiated and cannot be inherited. It can only contain static members (fields, properties, and methods). It acts as a simple container for a group of related utility functions or constants that do not depend on any object state.

How It Works: A Compile-Time Construct

Marking a class as static is primarily a way of telling the compiler to enforce two strict rules:

1. All members of this class **must** also be declared as static. The compiler will give an error if you try to declare an instance member inside a static class.
2. No one can create an instance of this class using the new keyword, and no other class can inherit from it.

You don't need to create an object; you access the members directly through the class name.

Why and When to Use It

Use a static class when you have a collection of helper or utility methods that are logically related but do not need to be associated with a specific object's state.

Real-World Examples from the .NET Framework

- **System.Math**: Contains methods for mathematical operations like Sqrt(), Max(), Sin(). You don't need a Math object to calculate a square root.
- **System.Console**: Contains methods for interacting with the console window like WriteLine() and ReadKey().
- **System.IO.File**: Contains utility methods for working with files like ReadAllText() and Exists().

Example: A Custom Utility Class

C#

```
// 'static' ensures this class cannot be instantiated.
public static class UnitConverter
{
    private const double KmsInAMile = 1.60934;

    // All members must also be static.
    public static double KilometersToMiles(double km)
    {
        return km / KmsInAMile;
    }

    public static double MilesToKilometers(double miles)
    {
        return miles * KmsInAMile;
    }
}

// --- In your Main method ---
// You call the methods directly on the class.
double miles = UnitConverter.KilometersToMiles(10);
Console.WriteLine($"10km is approximately {miles:F2} miles.");
```

Lecture 4: Enumerations

Introduction

An **enumeration**, or **enum**, is a special value type that allows you to define a set of named integral constants. It provides a way to create a distinct type that represents a fixed set of related values, making your code more readable, maintainable, and type-safe.

Why and When to Use It

The primary purpose of an enum is to **avoid "magic numbers"** and improve code readability. Instead of writing code like if (userStatus == 2), which is meaningless without context, you can write if (status == UserStatus.Suspended). This is self-documenting and far less error-prone. It

restricts a variable to a known set of allowed values, preventing bugs where an invalid integer might be assigned.

How It Works: Underlying Values

By default, an enum's underlying data type is int. The compiler assigns values to the members starting from 0 and incrementing by one.

C#

```
public enum DayOfWeek // Starts at 0
{
    Sunday,    // 0
    Monday,    // 1
    Tuesday,   // 2
    Wednesday, // 3
    Thursday,  // 4
    Friday,    // 5
    Saturday   // 6
}
```

You can customize this behavior:

1. **Change the starting number:**
   C#
   ```
   public enum Month { January = 1, February = 2, /* ... */ }
   ```

2. **Change the underlying type** (to byte, long, etc.):
   C#
   ```
   public enum ErrorCode : ushort { NotFound = 404, ServerError = 500 }
   ```

Example

C#

```
public enum ShippingStatus
{
    Ordered,
    Processing,
    Shipped,
    InTransit,
    Delivered
}
```

```
public class Order
{
    public int OrderId { get; set; }
    public ShippingStatus Status { get; set; }
}

public class ShippingService
{
    public void ProcessOrder(Order order)
    {
        // Using an enum in a switch statement is very clean and readable.
        switch (order.Status)
        {
            case ShippingStatus.Ordered:
                Console.WriteLine("Processing the order...");
                order.Status = ShippingStatus.Processing;
                break;
            case ShippingStatus.Processing:
                Console.WriteLine("Shipping the order...");
                order.Status = ShippingStatus.Shipped;
                break;
            // ... other cases
        }
    }
}
```

Lecture 5: IMP Points to Remember

- **partial is a Compile-Time Feature:** This is the most important takeaway. Partial classes and methods are a tool for organizing source code in different files. The compiler "stitches" them together into one single class before the program runs. At run-time, the CLR has no concept of a "partial class."
- **The Primary Use of partial is Code Generation:** In modern C#, the main reason you'll encounter partial classes is when working with frameworks that automatically generate code (like ASP.NET Core, WPF, Entity Framework). It provides a clean separation between machine-generated code and developer-written code.

- **A static class is a Sealed, Abstract Container:** Thinking of it this way can be helpful. It's "abstract" in that you can't create an instance, and it's "sealed" in that you can't inherit from it. Its sole purpose is to hold a group of static members.
- **static class vs. Singleton Pattern:** This is a good interview topic.
    - A **static class** cannot be instantiated at all. It exists only as a single container for its members. It cannot implement interfaces.
    - The **Singleton Pattern** is a design pattern where a regular class is engineered to ensure only one instance of it is ever created. That single instance is an object that can be passed around, can implement interfaces, and can be inherited from (if not sealed).
- **enum for Type Safety and Readability:** Always use an enum over integer or string constants when you have a variable that should be restricted to a small, fixed set of named values. It prevents bugs and makes the code self-documenting.
- **Casting enums:** You can explicitly cast an integer to an enum type and vice-versa. This can be useful when interoperating with databases or APIs that store the enum as a number. ShippingStatus status = (ShippingStatus)2; // status is now 'Shipped'. int statusValue = (int)ShippingStatus.Shipped; // statusValue is 2.

Section 14: Structs

Lecture 1: Intro and Working with Structs

Introduction

A **struct** (short for structure) is a composite data type that allows you to group a set of related variables under a single name. The single most important characteristic of a struct is that it is a **value type**. This fundamentally distinguishes it from a `class`, which is a **reference type**. This difference dictates how structs are stored in memory, how they are passed between methods, and how they behave during assignment.

Structs are typically used for creating small, lightweight objects that logically represent a single value, such as a coordinate point, a color, or a pair of related numbers.

How It Works: Value Type Semantics and Memory

Because a struct is a value type, a variable of a struct type holds the **actual data itself**, not a reference (or pointer) to the data.

- **Memory Allocation:** Structs are almost always allocated on the **stack**, a region of memory that is very fast and efficient. When a method call finishes, any structs created within it are instantly deallocated as the stack frame is popped. This avoids the overhead of **garbage collection** that comes with class objects on the heap.
- **Assignment Behavior:** When you assign one struct variable to another, you are creating a complete **copy** of the data. Each variable will have its own independent instance.

Example: Demonstrating Copy-on-Assignment

Let's create a simple Point struct to see this behavior in action.

C#

```
public struct Point
{
    // By default, fields in a struct should be public for easy access,
    // as structs are primarily simple data containers.
    public int X;
    public int Y;
}

public class Program
{
    public static void Main(string[] args)
    {
        // point1 holds its own X and Y values directly.
        Point point1 = new Point();
        point1.X = 10;
        point1.Y = 20;

        // When we assign point1 to point2, a complete COPY of the data is
made.
        // point2 is now a new, independent instance with the same values.
        Point point2 = point1;

        Console.WriteLine($"Before change: point1=({point1.X}, {point1.Y}),
point2=({point2.X}, {point2.Y})");

        // Now, let's change the X value of the FIRST point.
        point1.X = 99;
        Console.WriteLine("\n--- Changed point1.X to 99 ---");

        // The change only affects point1. point2 is completely unaffected
        // because it is a separate copy.
        Console.WriteLine($"After change: point1=({point1.X}, {point1.Y}),
point2=({point2.X}, {point2.Y})");
    }
}
```

Output:

Before change: point1=(10, 20), point2=(10, 20)

--- Changed point1.X to 99 ---
After change: point1=(99, 20), point2=(10, 20)

Lecture 2: Structures with Constructors

Introduction

Like classes, structs can declare **constructors** to initialize their fields. Using a constructor is the preferred way to create a struct instance and ensure it starts in a valid and predictable state, rather than setting each field manually after creation.

How It Works & Rules

There are specific rules for struct constructors that differ from class constructors:

1. **Parameterized Constructors:** You can define constructors that take parameters. Within a parameterized constructor, you **must** initialize **all** instance fields of the struct. The C# compiler enforces this rule to ensure the struct is fully initialized.
2. **The Implicit Parameterless Constructor:** Every struct always has a built-in, parameterless constructor that initializes all its fields to their system default values (0, false, null). You **cannot** delete or override this default behavior.
3. **Explicit Parameterless Constructors (C# 10+):** In modern C# (version 10 and later), you are now allowed to explicitly declare your own public, parameterless constructor to provide different default initialization logic. However, the implicit one still exists behind the scenes.

Example

Let's enhance our Point struct with a constructor to make initialization cleaner.

C#

```
public struct Point
{
    public int X;
    public int Y;

    // A public, parameterized constructor.
    public Point(int x, int y)
    {
        // We MUST initialize all fields inside this constructor.
        // If we commented out either line, we would get a compiler error.
        this.X = x;
        this.Y = y;
    }

    public void Display()
    {
        Console.WriteLine($"({X}, {Y})");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // 1. Create a point using the default parameterless constructor.
        // X and Y will both be initialized to 0.
        Point origin = new Point();
        Console.Write("Origin point: ");
        origin.Display(); // Output: (0, 0)

        // 2. Create a point using our custom parameterized constructor.
        Point p1 = new Point(15, 25);
        Console.Write("Custom point: ");
        p1.Display(); // Output: (15, 25)
    }
}
```

Lecture 3: Structures vs. Classes

Introduction

Knowing when to use a struct versus a class is a critical skill for any C# developer. This is one of the most fundamental and frequently asked interview questions because the choice has significant implications for memory management, performance, and program behavior. The core difference is that **structs are value types** and **classes are reference types**.

Detailed Comparison

- **Type & Memory:**
    - **Class:** A **reference type**. The object itself is created on the **heap** (a large pool of memory managed by the garbage collector). The variable holds only a small **reference (or pointer)** to the object's memory location.
    - **Struct:** A **value type**. The object is typically created on the **stack** (a fast, temporary memory region for method calls). The variable holds the **actual data** of the object directly.
- **Assignment Behavior:**
    - **Class:** Assigning a class variable to another **copies the reference**. Both variables then point to the **same single object**.
    - **Struct:** Assigning a struct variable to another **copies the entire data**. This creates a new, independent instance.
- **Passing to Methods:**
    - **Class:** When you pass a class object to a method, a **copy of the reference** is passed. The method can modify the state of the original object.
    - **Struct:** When you pass a struct to a method, a **copy of the entire struct** is passed. The method works on the copy; changes inside the method **do not** affect the original struct.
- **Inheritance:**
    - **Class:** Fully supports single inheritance. A class can inherit from another class.

    - **Struct:** Does **not** support inheritance. A struct cannot inherit from another struct or class (except implicitly from System.ValueType).

- **Default Value:**
    - **Class:** The default value of a class variable is null (it references nothing).
    - **Struct:** The default value is an instance of the struct with all its fields set to their own default values (0, false, etc.). A struct can never be null.

Practical Example: The Method Pass-Through Test

This example clearly demonstrates the fundamental difference in behavior when passing a class versus a struct to a method.

C#

```
// The class (reference type)
public class ClassPoint { public int X; }

// The struct (value type)
public struct StructPoint { public int X; }

public class Program
{
    // This method attempts to change the X value of whatever it receives.
    public static void TryToChangeValue(ClassPoint cp, StructPoint sp)
    {
        cp.X = 100;
        sp.X = 100;
    }

    public static void Main(string[] args)
    {
        ClassPoint myClassPoint = new ClassPoint();
        myClassPoint.X = 10;

        StructPoint myStructPoint = new StructPoint();
        myStructPoint.X = 10;

        Console.WriteLine($"Before method call: ClassPoint.X =
{myClassPoint.X}, StructPoint.X = {myStructPoint.X}");

        // Pass both to the method.
```

```
        TryToChangeValue(myClassPoint, myStructPoint);

        Console.WriteLine($"After method call:  ClassPoint.X =
{myClassPoint.X}, StructPoint.X = {myStructPoint.X}");
    }
}
```

Output:

```
Before method call: ClassPoint.X = 10, StructPoint.X = 10
After method call:  ClassPoint.X = 100, StructPoint.X = 10
```

Conclusion: The change to the ClassPoint persisted because the method received a reference to the original object. The change to the StructPoint was lost because the method only operated on a temporary copy.

Interview Perspective: When should you use a struct?

- **Ideal Answer:** "You should use a class by default. You should only consider using a struct when the type meets several criteria: it logically represents a **single value** (like a number or a coordinate), it is **small** (typically under 16 bytes), it is **immutable** (won't change after creation), and it will **not be frequently 'boxed'** or unboxed."

Lecture 4: Readonly Structures

Introduction

A readonly struct is a special type of structure that is declared as **immutable**. This means that once an instance of the struct is created, its state can never be changed for its entire lifetime. This provides a strong compile-time guarantee of immutability.

How It Works

When you add the readonly modifier to a struct declaration, the C# compiler enforces two rules:

1. All instance fields within the struct **must also be declared as readonly**.
2. Because all fields are readonly, they can only be assigned values either at their declaration or within the struct's constructor.

This creates a powerful guarantee that the struct is truly a "snapshot" of data that cannot be altered.

Why and When to Use It

Use a readonly struct when you want to create small, lightweight, data-centric types that are guaranteed to be immutable. This is extremely useful for:

- Creating predictable and safe code, especially in multi-threaded applications, as immutable types are inherently thread-safe.
- Preventing accidental modification of data.
- Clearly communicating the design intent that an object is just a "value" and should not have its state changed.

Example

A Color or RgbValue is a perfect candidate for a readonly struct.

C#

```
public readonly struct RgbColor
{
    // All fields MUST be readonly.
    public readonly byte R;
    public readonly byte G;
    public readonly byte B;

    public RgbColor(byte r, byte g, byte b)
    {
        // Values can only be set here, in the constructor.
        this.R = r;
        this.G = g;
        this.B = b;
    }

    public void ChangeRed(byte newRed)
    {
        // The following line would cause a compiler error because R is
readonly.
        // this.R = newRed; // ERROR! Cannot assign to 'R' because it is a
readonly field.
    }
}
```

```
// --- In your Main method ---
RgbColor myColor = new RgbColor(255, 165, 0); // Orange
// 'myColor' is now immutable. Its state cannot be changed.
```

Lecture 5: Primitive Types are Structures

Introduction

This lecture reveals a fundamental truth about C#'s type system that clarifies much of the behavior we've seen so far. Most of the basic types that we think of as built-in keywords are, in fact, just convenient C# **aliases** for struct types defined in the .NET System namespace.

How It Works: Keywords as Aliases

Here is a list of some of the most common primitive types and the structs they represent:

- int is an alias for System.Int32
- double is an alias for System.Double
- bool is an alias for System.Boolean
- char is an alias for System.Char
- long is an alias for System.Int64
- decimal is an alias for System.Decimal

The Implication

Because these fundamental types are structs, they are all **value types**. This single fact explains their core behavior:

- They are stored on the stack, making them fast.
- When you assign one int variable to another, the value is copied.
- When you pass an int to a method, a copy of the value is passed.
- They cannot be null (because they are not references).

Since they are fully-featured structs, they even have methods you can call on them.

C#

```
int myNumber = 123;

// We are calling the ToString() method on the System.Int32 struct.
string myString = myNumber.ToString();

// You can even call methods on a literal value!
Console.WriteLine(123.ToString());
```

Interview Perspective

- **Question:** "Is int in C# a primitive type or an object?"
- **Ideal Answer:** "In C#, int is an alias for the System.Int32 struct. Because it's a struct, it is a value type. However, since all structs in .NET implicitly inherit from System.ValueType, which in turn inherits from System.Object, it can be treated like an object when needed (a process called boxing), giving it methods like ToString()."

Lecture 6: IMP Points to Remember

- **Structs are Value Types, Classes are Reference Types:** This is the most important distinction. It dictates how they are stored in memory (stack vs. heap), how they behave on assignment (copy vs. reference), and how they are passed to methods.

- **Structs for Performance:** Because structs are typically allocated on the stack, they do not create "garbage" that the garbage collector needs to clean up. For applications that create and destroy millions of small objects, using structs can lead to significant performance improvements.
- **Copying Behavior is a Double-Edged Sword:** The fact that structs are copied on assignment prevents one part of your code from unexpectedly changing a value used by another part. However, if you expect reference behavior, it can be a source of bugs. Always be conscious of this when passing structs to methods.
- **Choose `class` by Default:** This is the official Microsoft recommendation. The C# language and runtime are highly optimized for classes. You should only choose to use a `struct` when you have a specific, performance-related reason and the type you are creating fits the criteria: it's small, logically represents a single value, and is preferably immutable.
- **Embrace Immutability with `readonly struct`:** `readonly struct` is a powerful feature for creating safe, predictable value types. When a type has no reason to change its state after creation, making it a `readonly struct` is an excellent design choice.
- **All The "Primitives" are Structs:** Remembering that `int`, `double`, `bool`, etc., are all structs under the hood provides a unified mental model for understanding C#'s type system.

Section 15: System.Object class

Lecture 1: Overview of System.Object class

Introduction

In the .NET type system, System.Object (or its C# alias, object) is the most important class. It is the **ultimate base class**, the universal ancestor from which every single type in C# implicitly derives. Whether you create a class, a struct, an enum, or even use a primitive type like int or double, they all ultimately trace their lineage back to System.Object. This design choice is the foundation of C#'s powerful and unified type system.

Internal Class Hierarchy and Design

The hierarchy ensures that every type shares a common foundation. Here's how it's structured:

- System.Object: The root of everything.
    - **Reference Types** (class, delegate, string, etc.): These types inherit directly from System.Object.
    - **Value Types** (struct, enum, int, double, etc.): These types inherit from a special abstract class called System.ValueType, which in turn inherits directly from System.Object.

Why was it designed like that? The Unified Type System

The primary reason for this design is to create a **unified type system**. By having a single, common base class, the .NET framework guarantees that every single variable, regardless of its specific type, shares a baseline set of functionalities. This provides several powerful benefits:

1. **Polymorphism**: It allows you to write methods that can operate on any type. You can create a variable of type object and assign literally any value to it, because everything "is an" object.
2. **Common Services**: It ensures that every object has access to a fundamental set of methods for basic operations like comparing for equality (Equals),

getting a string representation (ToString), and retrieving type information (GetType).

3. **General-Purpose Collections**: Before generics, this design allowed for the creation of collections like ArrayList that could store a mix of any data types, because they could all be treated as their base object type. (This led to boxing/unboxing, which we'll cover soon).

Interview Perspective

- **Question**: "What is the role of System.Object in C#?"
- **Ideal Answer**: "System.Object is the ultimate base class for all types in the .NET framework, creating a unified type system. Every type, including both reference types and value types, implicitly inherits from it. This design guarantees that every object has a common set of methods like ToString(), Equals(), and GetHashCode(), enabling a powerful level of polymorphism where any type can be treated as an object."

Lecture 2: Understanding and Overriding Methods of Object class

Introduction

Since every class you create implicitly inherits from System.Object, it also inherits its public and protected instance methods. These methods provide foundational behavior, but their default implementations are often too generic. To make your classes behave intelligently, you will frequently **override** these methods to provide your own custom logic.

The Four Important Instance Methods

1. ToString()

- **Default Behavior**: The default implementation of ToString() is not very useful. It simply returns a string containing the fully qualified name of the type (e.g., "MyProject.Person").

- **Overriding (virtual)**: This method is marked as virtual, meaning it is designed to be overridden. You should override ToString() in your classes to provide a meaningful, human-readable string representation of the object's current state. This is invaluable for logging and debugging.

Example:
C#
```
public class Car

{
    public string Model { get; set; }
    public int Year { get; set; }

    // Override ToString() to provide a useful description.
    public override string ToString()
    {
        return $"{Year} {Model}";
    }
}

// --- In Main ---
Car myCar = new Car { Model = "Mustang", Year = 2023 };
Console.WriteLine(myCar.ToString()); // Output: 2023 Mustang
```

2. Equals(object obj)

- **Default Behavior**: This is a critical interview topic. The default behavior depends on the type:
    - For **reference types** (class), it performs **reference equality**. It returns true only if two variables point to the exact same object in memory.
    - For **value types** (struct), it performs **value equality**. It uses reflection to compare all fields of the two structs for equality.
- **Overriding (virtual)**: You override Equals() to define a custom meaning of "equality" for your class, usually by comparing the values of key fields or properties.

- **Example**:
  C#

```
public class Point

{
    public int X { get; set; }
    public int Y { get; set; }

    public override bool Equals(object obj)
    {
        // Check for null and if the types match
        if (obj == null || this.GetType() != obj.GetType()) return false;

        Point other = (Point)obj;
        // Define equality as having the same X and Y values.
        return (this.X == other.X) && (this.Y == other.Y);
    }
}
Point p1 = new Point { X = 10, Y = 20 };
Point p2 = new Point { X = 10, Y = 20 };
Point p3 = new Point { X = 5, Y = 5 };

Console.WriteLine(p1.Equals(p2)); // Output: True
Console.WriteLine(p1.Equals(p3)); // Output: False
```

3. GetHashCode()

- **What is a Hash Code?** A hash code is a numeric value of a fixed size that is generated from an object. It's not unique, but a good hashing algorithm will produce a wide distribution of values, minimizing collisions (when two different objects produce the same hash code). The primary purpose of a hash code is to enable high-performance lookups in hash-based collections like Dictionary<TKey, TValue> and HashSet<T>.
- **How do Hash-Based Collections Work?** When you add an item to a Dictionary, the dictionary doesn't loop through all its existing items to see if the key is already there. Instead:
  1. It calls the key's GetHashCode() method to get an integer.
  2. It uses this integer to calculate an index, which points to a specific "bucket" or "slot" in its internal array.

3. It places the key-value pair in that bucket.
   When you look up a key, it repeats the process: it calculates the hash code, goes directly to the correct bucket, and then uses the Equals() method to check only the very few items within that bucket. This is what makes dictionary lookups incredibly fast (averaging O(1) time complexity).

- **The Golden Rule of Overriding: If you override Equals(), you MUST also override GetHashCode()**.
- **Why?** If two objects are considered equal by your Equals() method but they produce different hash codes, the hash-based collection will break. It might place them in different buckets. When you try to look up one of them, the dictionary would go to the wrong bucket, fail to find the object (even though an "equal" one exists elsewhere), and incorrectly report that the item is not in the collection. The contract is: **if a.Equals(b) is true, then a.GetHashCode() must equal b.GetHashCode()**.
- **Example**:
  C#

```
public class Point // Continuing from above

{
    // ... X and Y properties and Equals method ...

    public override int GetHashCode()
    {
        // A modern, simple, and effective way to combine hash codes.
        return HashCode.Combine(X, Y);
    }
}
```

4. GetType()

- **Behavior**: This method is **not** virtual and cannot be overridden. It always returns a Type object containing detailed metadata about the object's exact runtime type.
- **When to Use It**: When you need to inspect an object's type at runtime, often for reflection (a more advanced topic).

Example:
C#
```
Car myCar = new Car { Model = "Mustang", Year = 2023 };

Type carType = myCar.GetType();
Console.WriteLine($"The object is of type: {carType.FullName}");
```

Lecture 3: Boxing

Introduction

Boxing is the process of converting a **value type** instance (like an `int`, `double`, or a custom `struct`) into a **reference type** (`object` or an interface it implements). This allows value types, which normally live on the fast stack, to be treated like objects and stored on the heap, enabling them to be used in collections or methods that expect an `object`.

How It Works: The Internal Mechanics

When a value type is boxed, the .NET runtime performs these steps behind the scenes:

1. **Memory Allocation**: A small amount of memory is allocated on the **heap**. This memory block will act as the "box."
2. **Value Copy**: The value of the value type variable (which is on the stack) is **copied** into the newly allocated box on the heap.
3. **Reference Return**: The resulting variable is now a reference (a memory address) that points to this new box object on the heap.

When It Happens

Boxing happens implicitly whenever you assign a value type to a variable of type `object` or an interface type.

Performance Implications

Boxing is a computationally **expensive operation** and should be avoided in performance-critical code. The cost comes from two places:

1. The memory allocation on the heap is significantly slower than stack allocation.
2. The newly created box object on the heap puts pressure on the **Garbage Collector (GC)**, which will eventually have to clean it up.

Example

C#

```
// 1. 'i' is a value type, living on the stack.
int i = 123;

// 2. 'o' is a reference type variable.
// When 'i' is assigned to 'o', boxing occurs.
object o = i;

// What happened:
// - A new object box was allocated on the heap.
// - The value 123 was copied from the stack into the heap box.
// - The variable 'o' now holds the memory address of that box.

// The most common historical example was with non-generic collections.
System.Collections.ArrayList list = new System.Collections.ArrayList();
list.Add(456); // Boxing occurs here! 456 is put in a box on the heap.
```

Lecture 4: Unboxing

Introduction

Unboxing is the reverse process of boxing. It is the explicit conversion of an object reference (that points to a boxed value type) back into its original value type.

How It Works: The Internal Mechanics

Unboxing is also a multi-step, potentially costly operation:

1. **Type Check**: The runtime first checks if the object being unboxed is actually a boxed instance of the **exact** target value type. If the object is null or is a boxed value of a different type, an **InvalidCastException** is thrown.
2. **Value Copy**: If the type check passes, the value is **copied** from the box on the heap back into a new value type variable, which is allocated on the stack.

Syntax

Unboxing is an **explicit** conversion and requires a cast.

Performance Implications

Unboxing also has a performance cost due to the type checking and the memory copy operation, though it is generally slightly cheaper than boxing as it doesn't involve heap allocation.

Example

C#

```
// Start with a boxed integer.
object o = 123;

// To get the value back, we must explicitly cast it. This is unboxing.
int j = (int)o;

// What happened:
// 1. The runtime checked if 'o' actually points to a boxed 'int'. It does.
// 2. The value 123 was copied from the heap box to the stack variable 'j'.
Console.WriteLine($"Unboxed value: {j}");


// --- Example of a FAILED unboxing ---
object anotherObject = 456; // Boxed int


try
{
    // This will FAIL. You cannot unbox to a different type, even if
    // the conversion would normally be safe (like int to long).
    // You must unbox to the exact original type first.
    long l = (long)anotherObject;
}
catch (InvalidCastException ex)
{
    Console.WriteLine("\nError: " + ex.Message);
}

// The correct way to do the above conversion:
long correctLong = (int)anotherObject; // 1. Unbox to int, 2. Implicitly cast
int to long.
```

Lecture 5: IMP Points to Remember

- **Object is the Universal Ancestor**: Every type in .NET derives from System.Object. This creates a unified type system where any variable can be treated as an object, enabling a powerful form of polymorphism.
- **Override ToString() for Meaningful Debugging**: It is a fundamental best practice to override the ToString() method in all your custom classes. A good ToString() implementation significantly simplifies logging and debugging by providing a clear, human-readable representation of an object's state.
- **The Equals() and GetHashCode() Contract**: This is a critical interview topic. **If you override Equals() to provide value-based equality, you MUST also override GetHashCode()**. The rule is simple: two objects that are considered equal must return the same hash code. Failing to do this will break the functionality of hash-based collections like Dictionary<TKey, TValue> and HashSet<T>.
- **Boxing and Unboxing are Performance Traps**: Be acutely aware that these operations are computationally expensive operations due to heap allocation, garbage collector pressure, and type checking. The introduction of **Generics** (e.g., List<int> instead of ArrayList) was a major language feature designed specifically to eliminate these hidden performance costs by creating type-safe collections that don't require boxing for value types.
- **Understanding GetType() vs. is**: myObject.GetType() == typeof(MyClass) checks if an object is of an exact type. The is operator (myObject is MyBaseClass) is more flexible; it checks for compatibility within an inheritance hierarchy (it returns true if the object is of that type OR any of its derived types). Knowing this distinction shows a deeper understanding of runtime type checking.

Section 17: Handling null

Lecture 1: What is null

Introduction

In C#, **null** is a special literal that represents the absence of a value. It is used with **reference types** (like string, object, or any class instance) to signify that a variable does not point to any object in memory on the heap. It means the reference is empty or non-existent.

By default, any variable of a reference type that has not been assigned an object is initialized to null. Understanding how to handle null is one of the most critical skills for a C# developer, as it is the source of the most common run-time error in .NET programming.

How It Works: The NullReferenceException

The danger of null is that it leads to the dreaded **NullReferenceException**. This exception occurs at **run-time** when you try to access a member (a method, property, or field) on a variable that is currently null. You are essentially trying to press a button on a remote control that doesn't exist. The C# compiler cannot always detect this situation, so the program compiles fine but crashes when it runs.

Example

Let's look at a simple scenario that causes this error.

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        // 'name' is a reference type (string). We explicitly set it to null.
        // This is the same as if it were an uninitialized class field.
        string name = null;
```

```
        // This line looks fine to the compiler. It only knows that 'name' is a string,
        // and strings have a 'Length' property.
        Console.WriteLine("The length of the name is: " + name.Length);

        // When this code runs, the .NET runtime tries to find the object that 'name'
        // points to. Since it points to nothing (null), the program cannot
        // get a 'Length' property from nothing, and it crashes immediately
        // by throwing a NullReferenceException.
    }
}
```

Best Practices & Interview Perspective

Defensive Programming: The key to avoiding this exception is "defensive programming." Before you use an object, especially one coming from a method parameter or a class field, always consider checking if it's null.

C#

```
if (name != null)

{
    Console.WriteLine(name.Length);
}
else
{
    Console.WriteLine("Name is not provided.");
}
```

- **Question:** "What is a NullReferenceException and what causes it?"
- **Ideal Answer:** "A NullReferenceException is a run-time error that occurs when you try to access a member—like a method or property—on a variable that holds a null reference. It means the variable isn't pointing to an actual object instance in memory, so the operation cannot be performed."

Lecture 2: Nullable Types

Introduction

By default, **value types** (int, double, bool, struct, etc.) **cannot be null**. This is because they are not references; a variable of a value type always contains the value itself. An int variable always contains a number, even if it's just 0.

However, there are many real-world scenarios where a value type might logically be "missing" or "undefined." For example, in a database, a Price or DateOfBirth column might allow NULL values. To represent this in C#, the language provides **nullable value types**.

How It Works: The ? Syntax and Nullable<T>

You make a value type nullable by adding a question mark (?) after its name.

Syntax: int?, double?, bool?

This shorthand is an alias for a special generic struct called **System.Nullable<T>**. So, int? is just a cleaner way of writing Nullable<int>. This struct wraps your value type and adds two important properties:

1. **HasValue**: A bool property that is true if the variable holds an actual value and false if it is null.
2. **Value**: A property of type T that gets the underlying value. You should **only** access this property after checking that HasValue is true, otherwise you will get an InvalidOperationException.

Example

C#

```
public class SurveyResponse
{
    // A rating might be optional, so we use a nullable int.
    public int? Rating { get; set; }
}

public class Program
{
    public static void Main(string[] args)
```

```
    {
        SurveyResponse response1 = new SurveyResponse();
        response1.Rating = 5; // Assign a value.

        SurveyResponse response2 = new SurveyResponse();
        response2.Rating = null; // Represents a missing value.

        PrintRating(response1);
        PrintRating(response2);
    }

    public static void PrintRating(SurveyResponse response)
    {
        // ALWAYS check HasValue before accessing .Value
        if (response.Rating.HasValue)
        {
            Console.WriteLine($"Rating provided: {response.Rating.Value}");
        }
        else
        {
            Console.WriteLine("No rating was provided.");
        }
    }
}
```

Output:

```
Rating provided: 5
No rating was provided.
```

Lecture 3: Null Coalescing Operator (??)

Introduction

The **null coalescing operator ??** is a concise binary operator used to provide a **default value** for a nullable type or a reference type if it is `null`. It's a clean and highly readable shortcut for a common type of `if` check.

How It Works

The operator works on two operands: A ?? B

1. It evaluates the left-hand operand (A).
2. If A is **not null**, the expression returns the value of A.
3. If A **is null**, the expression returns the value of the right-hand operand (B).

When to Use It

Use this operator whenever you want to use a variable's value but fall back to a default value if the variable is null. It makes your code much shorter and more expressive.

Example

Let's say we want to get a user's nickname to display, but if they haven't set one, we should use their full name as a fallback.

C#

```
public class User
{
    public string FullName { get; set; }
    public string Nickname { get; set; } // This can be null
}

// --- In your Main method ---
User user1 = new User { FullName = "John Doe", Nickname = "Johnny" };
User user2 = new User { FullName = "Jane Smith", Nickname = null };

// The old way using an if statement:
string displayName1;
if (user1.Nickname != null)
{
    displayName1 = user1.Nickname;
}
else
{
    displayName1 = user1.FullName;
}
Console.WriteLine($"Welcome, {displayName1}"); // Output: Welcome,
Johnny
```

```
// The modern, concise way using the ?? operator:
string displayName2 = user2.Nickname ?? user2.FullName;
Console.WriteLine($"Welcome, {displayName2}"); // Output: Welcome, Jane Smith
```

Lecture 4: Null Propagation Operator (?.)

Introduction

The **null propagation operator ?.** (also known as the null-conditional operator) is a modern C# feature designed to make navigating object chains with potential null values much safer and more concise. It helps you avoid writing deeply nested if statements just to check for null at each step.

How It Works

When you use ?. to access a member (a property or method), C# first checks if the object on the left of the ?. is null.

- If it is **null**, the entire expression immediately **short-circuits** and returns null. It never attempts to access the member on the right, thus avoiding a NullReferenceException.
- If it is **not null**, it proceeds to access the member as usual.

This operator can be chained. If any part of the chain A?.B?.C is null, the entire expression gracefully evaluates to null.

There is also a version for accessing indexers, called the **null-conditional indexer (?[])**.

Example

Consider a common scenario where you need to get a customer's postal code. The customer might be null, or their address might be null.

C#

```
public class Address { public string PostalCode { get; set; } }
public class Customer { public Address ShippingAddress { get; set; } }

public class Program
{
    public static void Main(string[] args)
    {
        Customer customerWithAddress = new Customer
        {
            ShippingAddress = new Address { PostalCode = "501505" }
        };
        Customer customerWithoutAddress = new Customer
{ ShippingAddress = null };
        Customer nullCustomer = null;

        // --- The Old Way: Nested if checks ("pyramid of doom") ---
        if (customerWithAddress != null)
        {
            if (customerWithAddress.ShippingAddress != null)
            {

Console.WriteLine(customerWithAddress.ShippingAddress.PostalCode);
            }
        }

        // --- The Modern, Safe, and Concise Way ---
        // Using the null propagation operator.
        string postal1 = customerWithAddress?.ShippingAddress?.PostalCode;
        string postal2 =
customerWithoutAddress?.ShippingAddress?.PostalCode;
        string postal3 = nullCustomer?.ShippingAddress?.PostalCode;

        // We can use the null coalescing operator to provide a default.
        Console.WriteLine($"Postal Code 1: {postal1 ?? "Not Found"}"); //
Output: 501505
        Console.WriteLine($"Postal Code 2: {postal2 ?? "Not Found"}"); //
Output: Not Found
        Console.WriteLine($"Postal Code 3: {postal3 ?? "Not Found"}"); //
Output: Not Found
```

```
    }
}
```

Lecture 5: IMP Points to Remember

- **null is for Reference Types:** Remember that null means a reference variable points to nothing. Value types cannot be null unless you explicitly make them nullable.
- **NullReferenceException is The Developer's Responsibility:** This was famously called the "billion-dollar mistake" by its inventor. Modern C# features are designed to help you avoid it. Always be defensive and check for null, especially with data from external sources.
- **Use Nullable Types (int?, bool?) for Optional Values:** When a value type logically might not have a value (e.g., data from a database that can be NULL, an optional survey answer), use a nullable type. Always check the **.HasValue** property before accessing the **.Value** property to avoid an InvalidOperationException.
- **Prefer ?? and ?. for Conciseness and Safety:** These operators are not just shortcuts; they lead to significantly cleaner, more readable, and safer code than manual if (x != null) checks. Embrace them in all modern C# code. They are a sign of a developer who knows the language well.
- **Nullable Reference Types (Advanced Interview Point):** In C# 8.0 and later, you can enable a project-level feature called "Nullable Reference Types." When enabled, the compiler's behavior flips:
    - A standard string variable is considered **non-nullable** and the compiler will warn you if you try to assign null to it.
    - To declare a reference type that is allowed to be null, you must use the ? syntax, just like with value types: string?.
    - **Why it matters:** This feature helps you eliminate NullReferenceException errors at **compile-time** instead of waiting for them to crash your program at **run-time**. Mentioning your awareness of this modern feature demonstrates that you are up-to-date with C# best practices for writing robust code.

Section 18: Extension Methods and Pattern Matching

Lecture 1: Extension Methods

Introduction

Extension methods are a powerful C# feature that allow you to "add" new methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type's source code. This is particularly useful for extending types that you don't own, like classes from the .NET Framework (e.g., `string`, `DateTime`) or classes from third-party libraries. They provide the illusion of being instance methods, but are actually a form of syntactic sugar for calling a static method.

How It Works: The `static` and `this` Keywords

To create an extension method, you must follow three strict rules:

1. The method must be defined in a **static class**.
2. The method itself must be **static**.
3. The first parameter of the method must be marked with the **`this`** keyword, followed by the name of the type it extends.

The `this` parameter is what the compiler uses to link the method to the type. When you call the method, you do not provide an argument for this first parameter; it is automatically populated with the instance on which the method was called.

When to Use It & Real-World Examples

The most prominent real-world example of extension methods is **LINQ (Language-Integrated Query)**. Every LINQ method you use on a collection —.Where(), .Select(), .OrderBy(), .ToList()—is an extension method on the IEnumerable<T> interface.

Use extension methods to create reusable helper or utility functions that are logically tied to a specific type.

Example

Let's say we frequently need to count the number of words in a string. Instead of writing a separate utility function everywhere, we can extend the string type itself.

C#

```
// 1. Must be a static class.
public static class StringExtensions
{
    // 2. Must be a static method.
    // 3. 'this string str' indicates this method extends the string type.
    public static int WordCount(this string str)
    {
        if (string.IsNullOrEmpty(str))
        {
            return 0;
        }
        // The 'str' parameter will be the string instance the method is called
on.
        return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}

// --- In your Main method ---
string mySentence = "This is a sample sentence for testing.";

// Now we can call WordCount as if it were a built-in instance method of
the string class!
int count = mySentence.WordCount();

Console.WriteLine($"The sentence is: '{mySentence}'");
Console.WriteLine($"The word count is: {count}"); // Output: 8
```

Interview Perspective

- **Question:** "Can an extension method access private members of the type it extends?"
- **Ideal Answer:** "No. An extension method is just syntactic sugar for calling a static method and passing the instance as the first parameter. It does not have any special access to the type's private or protected members. It can only work with the public interface of the type it extends."

Lecture 2: Pattern Matching

Introduction

Pattern matching is a powerful set of features in modern C# that provides a more sophisticated and readable way to check if a variable conforms to a certain "shape" or "pattern," and to extract information from it if it does. It significantly enhances the capabilities of `if` statements and, most notably, `switch` statements and expressions, turning them into highly advanced control flow tools.

How It Works: Common Patterns

Pattern matching introduces several new patterns you can use in conditional logic.

- **Type Pattern (`is Type variableName`)** This pattern both checks the run-time type of an object and, if the check is successful, assigns the object to a new variable of that specific type.
  C#

  object myValue = "Hello World";

if (myValue is string str) // Checks if myValue is a string, and if so, assigns it to 'str'
{
  Console.WriteLine($"It's a string with length: {str.Length}");
}

Property Pattern ({ PropertyName: pattern }) This pattern allows you to check the values of an object's properties. It's most powerful inside a switch expression.
C#

```
public class Car { public int PassengerCount { get; set; } }

Car myCar = new Car { PassengerCount = 4 };
string message = myCar switch
{
    { PassengerCount: 0 } => "It's empty.",
    { PassengerCount: 1 } => "It's a solo ride.",
    { PassengerCount: 2 } => "A couple's trip.",
    _ => "It's a group."
};
```

Relational Patterns (<, >, <=, >=) These allow switch statements to work with ranges, not just constant values.
C#

```
int score = 85;

string grade = score switch
{
    >= 90 => "A",
    >= 80 => "B",
    >= 70 => "C",
    _ => "F"
};
```

Logical Patterns (and, or, not) These allow you to combine other patterns for more complex conditions.
C#

```
string GetWeatherReport(double temp, bool isRaining) => (temp, isRaining) switch

{
    (> 30.0, false) => "Hot and sunny.",
    (> 20.0, true) => "Warm and rainy.",
    (< 10.0, _ ) and (_, true) => "Cold and rainy.", // _ is a discard pattern
    _ => "Moderate weather."
};
```

Interview Perspective

- **Question:** "How has pattern matching changed the switch statement in modern C#?"
- **Ideal Answer:** "Pattern matching has transformed the switch statement from a simple control structure that only worked with constant integral values into a highly expressive tool. With modern pattern matching, a switch can branch on an object's type, the values of its properties, and numeric ranges using relational and logical patterns, which makes it a much cleaner alternative to complex if-else if chains."

Lecture 3: Implicitly Typed Variables (var)

Introduction

The **var** keyword, introduced in C# 3.0, lets you declare a local variable without explicitly specifying its data type. The compiler infers the type for you based on the value you use to initialize the variable.

How It Works: Compile-Time Type Inference

This is the most critical concept to understand: **var does not create a dynamically typed variable**. C# remains a statically-typed language. The var keyword is purely a **compile-time** feature. The compiler looks at the expression on the right-hand side of the assignment and determines its type. It then replaces the var keyword with that explicit type in the Intermediate Language (IL).

- When you write: var name = "John";
- The compiler generates the same code as if you had written: string name = "John";

Once the type is inferred and compiled, it is locked in. You cannot later assign a value of a different type to that variable.

Rules and Best Practices

- **Rule:** A var variable **must** be initialized at the time of its declaration so the compiler can infer its type. var x; is a compiler error.
- **Rule:** var can only be used for local variables inside a method. It cannot be used for fields or method parameters.
- **Best Practice - When to Use var:** Use it when the type of the variable is **obvious** from the right-hand side of the assignment. This reduces code clutter and improves readability.
    - var user = new User(); (Good: type is clearly User)
    - var names = new List<string>(); (Good: type is clearly List<string>)
    - var name = GetName(); (Good, if GetName() clearly returns a string)
- **Best Practice - When to Avoid var:** Avoid it when the type is not immediately clear, as this can harm readability.
    - var amount = 20; (Is this an int, double, decimal? Be explicit: decimal amount = 20M;)
    - var result = ProcessData(); (What does ProcessData return? Better to be explicit: ProcessResult result = ProcessData();)

Interview Perspective

- **Question:** "Does the var keyword make C# a dynamic language?"
- **Ideal Answer:** "No, absolutely not. var is a feature for compile-time type inference. The variable is still strongly and statically typed; the compiler just figures out the type for you based on the initialization value. Once compiled, there is no difference between a variable declared with var and one declared with an explicit type name."

Lecture 4: Dynamically Typed Variables (dynamic)

Introduction

The **dynamic** keyword, introduced in C# 4.0, allows you to create variables that **bypass static type checking at compile-time**. It fundamentally changes how the compiler treats the variable.

How It Works: Run-Time Resolution

When you declare a variable as dynamic, you are telling the compiler: "Trust me. Don't check anything I do with this variable right now. We will figure it out at **run-time**."

All calls to methods, properties, or operators on a dynamic variable are packaged up and resolved only when the program is actually running. If the member you tried to call exists at run-time, the code works. If it does not exist, the program will crash with a **RuntimeBinderException**.

Why and When to Use It

You should use dynamic with great caution as it sacrifices the core benefit of C#'s static type safety. Its main valid use cases are for interoperability:

- **Working with Dynamic Languages:** Interacting with libraries from languages like Python or IronRuby.
- **Working with the DOM in HTML:** Manipulating HTML elements in certain web contexts.
- **Parsing Dynamic Data Formats:** Working with data like JSON where the structure might not be known ahead of time.
- **COM Interop:** Interacting with older COM components.

var vs. dynamic: The Core Interview Question

This is a fundamental question to test your understanding of C#'s type system.

- **var (Implicitly Typed)**
  - **Typing:** Statically typed.
  - **When Checked:** Type is determined and checked by the **compiler** at **compile-time**.
  - **IntelliSense:** Full IntelliSense and compile-time error checking.
  - **Errors:** Produces compile-time errors if you call a non-existent method.
  - **Example:** var x = 10; x = "hello"; // COMPILE-TIME ERROR
- **dynamic (Dynamically Typed)**
  - **Typing:** Dynamically typed.
  - **When Checked:** Type is resolved by the **runtime** at **run-time**.
  - **IntelliSense:** No IntelliSense support for its members.
  - **Errors:** Bypasses the compiler and throws a **run-time exception** if you call a non-existent method.
  - **Example:** dynamic x = 10; x = "hello"; // NO ERROR. The type of x changes at runtime.

Example

C#

```
// The compiler knows x is an int.
var x = 10;
// Console.WriteLine(x.ToUpper()); // Compile-time error: 'int' does not
contain a definition for 'ToUpper'

// The compiler knows nothing about what y can do. It trusts you.
dynamic y = 10;

// This next line compiles fine, but will CRASH at runtime because the
integer 10
// does not have a ToUpper() method.
try
{
    Console.WriteLine(y.ToUpper());
}
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
{
    Console.WriteLine("RUNTIME ERROR: " + ex.Message);
}

// This is valid, because the type of y can change at runtime.
y = "hello";
Console.WriteLine(y.ToUpper()); // This works now. Output: HELLO
```

Lecture 5: Inner Classes (Nested Classes)

Introduction

An **inner class**, more formally known as a **nested class**, is a class that is declared within the scope of another containing class. It is a way to group classes that are tightly coupled and to control their visibility.

How It Works

A nested class is a member of its containing class. This has two key implications:

1. **Access:** A nested class has access to all members of its containing class, including private fields and methods. This is because it is considered part of the containing class's implementation.
2. **Visibility:** The nested class itself can have an access modifier (public, private, etc.). If you declare a nested class as private, it is completely hidden from the outside world and can only be used by the containing class.

Why and When to Use It

Use a nested class when a class **logically only makes sense in the context of its containing class** and is not intended for general public use. It's a tool for better organization and encapsulation.

The most classic real-world example is the relationship between a LinkedList and its Node. The concept of a Node (which holds a value and a pointer to the next node) is an implementation detail of the linked list. No outside code should ever need to create or work with a Node directly. By making the Node class a private nested class inside the LinkedList, you hide this implementation detail completely.

Example: LinkedList and Node

C#

```
public class MyLinkedList<T>
{
   // A private nested class. It is only visible and usable
   // by the MyLinkedList<T> class.
   private class Node
   {
      public T Value { get; set; }
      public Node Next { get; set; } // Can access other Node objects

      public Node(T value)
      {
         this.Value = value;
      }
   }
```

```
    // The containing class has a field that is an instance of the nested class.
    private Node _head;

    public void Add(T value)
    {
        Node newNode = new Node(value); // We can create Node objects here.
        if (_head == null)
        {
            _head = newNode;
        }
        else
        {
            Node current = _head;
            while (current.Next != null)
            {
                current = current.Next;
            }
            current.Next = newNode;
        }
    }
}

// --- In your Main method ---
MyLinkedList<int> list = new MyLinkedList<int>();
list.Add(1);
list.Add(2);

// The following line would cause a compiler error because the Node class
// is private and nested inside MyLinkedList.
// MyLinkedList<int>.Node myNode = new MyLinkedList<int>.Node(3); // ERROR! Inaccessible.
```

Section 19: GC, Destructors, IDisposable

Lecture 1: Garbage Collection & Generations

Introduction

Garbage Collection (GC) is the .NET runtime's **automatic memory manager**. In languages like C++, developers are responsible for manually allocating and deallocating memory, a process which is a major source of bugs like memory leaks and dangling pointers. In C#, the GC simplifies this by automatically identifying and reclaiming memory occupied by objects that are no longer being used by the application. This process applies to **managed memory**—the memory on the heap that is managed by the .NET Common Language Runtime (CLR).

How It Works: The Generational Hypothesis

The C# garbage collector is highly optimized. It operates on the "generational hypothesis," which observes that **most objects have a very short lifetime** (e.g., local variables in a method call) while a few objects live for the entire duration of the application. Based on this, the GC divides the heap into three "generations" to manage objects efficiently.

- Generation 0 (Gen 0): The Nursery 🌱
  This is where all new, small objects are allocated. GC collections happen most frequently in this generation. When the GC runs on Gen 0, it's a very fast process because it only has to scan a small segment of the heap containing the most recently created objects. Any objects that are still referenced by the application after the scan are considered "survivors." They are promoted to Generation 1.

- Generation 1 (Gen 1): The First Stop
  This generation contains objects that survived a Gen 0 collection. Since they have already survived one collection, the GC assumes they might live a bit longer. Therefore, Gen 1 is collected less frequently than Gen 0. When a Gen 1 collection runs, it also collects everything in Gen 0. Objects that survive a Gen 1 collection are promoted to Generation 2.

- Generation 2 (Gen 2): Long-Term Storage 🏛️
  This generation holds long-lived objects that have survived multiple collections (e.g., application-wide services, cached data, static objects). A Gen 2 collection is a "full" garbage collection, meaning it scans the entire managed heap (Gen 0, Gen 1, and Gen 2). This is the most time-consuming collection, so the GC runs it as infrequently as possible.

Why this design? This generational approach is a major performance optimization. Instead of having to pause the application to check every single object for cleanup every time, the GC can run extremely fast, cheap collections on just the "nursery" (Gen 0), where most objects are expected to be garbage anyway.

Lecture 2: Destructors

Introduction

A **destructor**, known in the CLR as a **finalizer**, is a special class method whose sole purpose is to clean up any **unmanaged resources** held by an object before that object is reclaimed by the garbage collector. Unmanaged resources are things the GC knows nothing about, like raw file handles, database connections, or window handles from the operating system.

Syntax

A destructor has the same name as the class, is prefixed with a tilde (~), and can have no access modifiers or parameters.

```
class MyResourceHolder
{
   // Destructor (Finalizer)
   ~MyResourceHolder()
   {
      // Cleanup code for unmanaged resources would go here.
      Console.WriteLine("Destructor is being called.");
   }
}
```

How It Works: The Finalization Queue

When you create an object that has a destructor, the CLR adds a pointer to it on a special list. Here's the complex process that happens when the object is no longer referenced:

1. During a garbage collection, the GC finds the object is unreachable.
2. Instead of reclaiming its memory, the GC sees it has a finalizer and places a reference to the object on a special **finalization queue**. The object is considered "alive" again for this collection cycle and survives.
3. A dedicated, low-priority finalizer thread runs in the background. It eventually picks up the object from the queue and calls its destructor (~MyResourceHolder()).
4. Only on a **future** garbage collection, after the destructor has run, can the GC finally reclaim the memory occupied by the object.

Best Practices & Interview Perspective

- **Why You Should Almost Never Use Destructors:**
    - **Non-Deterministic:** You have **no control** over when, or even if, the destructor will run. It could run immediately, or minutes later, or not at all before the application shuts down.
    - **Performance Overhead:** Objects with destructors live longer (surviving at least one extra GC cycle) and add significant overhead to the collection process.
    - **Complexity:** Writing them correctly, especially in a thread-safe way, is difficult.
- **Question:** "When would you implement a destructor?"
- **Ideal Answer:** "You should almost never implement a destructor directly. The standard C# pattern is to use IDisposable for deterministic cleanup of unmanaged resources. A destructor should only be used as a last-resort safety net in a class that directly owns an unmanaged resource, to ensure the resource is eventually released even if the user of the class forgets to call Dispose. It's a backup, not the primary cleanup mechanism."

Lecture 3: IDisposable

Introduction

The **IDisposable** interface is the standard, recommended, and professional mechanism in .NET for providing a **deterministic** way for a class to release its unmanaged resources. Deterministic means the cleanup happens at a precise and predictable time—when you decide it should happen.

How It Works: The `Dispose()` Contract

The IDisposable interface is incredibly simple. It is defined in the System namespace and contains only a single method signature:

```
public interface IDisposable
{
   void Dispose();
}
```

A class that implements this interface is making a clear promise: "I am holding onto a scarce or expensive resource (like a file handle or database connection). When you are finished using me, you **must** call my Dispose() method to release that resource immediately and predictably." Inside the Dispose() method, you write the code to perform the actual cleanup.

Real-World Examples

Many classes in the .NET framework that deal with external resources implement IDisposable:

- System.IO.StreamReader and System.IO.StreamWriter (for files)
- System.Data.SqlClient.SqlConnection (for databases)
- System.Net.Http.HttpClient (for network requests)
- System.Drawing.Bitmap (for graphics objects)

Example: A Simple File Logger

```
public class FileLogger : IDisposable
{
    private StreamWriter _streamWriter;
    private bool _isDisposed = false; // To prevent multiple dispose calls

    public FileLogger(string filePath)
    {
        // This opens a file handle - an unmanaged resource.
        _streamWriter = new StreamWriter(filePath, append: true);
    }

    public void Log(string message)
    {
        if (_isDisposed)
        {
            throw new ObjectDisposedException("FileLogger");
        }
        _streamWriter.WriteLine($"{DateTime.Now}: {message}");
    }

    // The implementation of the IDisposable contract.
    public void Dispose()
    {
        if (!_isDisposed)
        {
            // Release the unmanaged resource.
            _streamWriter.Close();
            _streamWriter.Dispose();
            _isDisposed = true;
            Console.WriteLine("Logger disposed. File handle released.");
        }
    }
}
```

Lecture 4: Using Declaration

Introduction

Calling Dispose() manually is possible, but it's risky. What if an exception occurs before the Dispose() line is reached? The resource would never be cleaned up. The **using** statement (and its more modern form, the **using declaration**) is a C# language feature that provides a convenient and completely safe syntax for working with IDisposable objects.

How It Works: Syntactic Sugar for try...finally

The using block is just syntactic sugar. When you write code with using, the C# compiler transforms it into a try...finally block behind the scenes. This guarantees that the .Dispose() method is **always** called, no matter how the code inside the block is exited—whether it completes normally or an exception is thrown.

Syntax: Statement vs. Declaration

- using Statement (Classic):
  The object is created, and its scope is limited to the curly braces {}. Dispose() is called automatically at the closing brace.

```
// logger is only accessible inside these curly braces.
using (FileLogger logger = new FileLogger("log.txt"))
{
    logger.Log("This is the first message.");
    logger.Log("This is the second message.");
} // logger.Dispose() is automatically called here.
```

- using Declaration (Modern C# 8.0+):
  This is a cleaner, more concise syntax that reduces nesting. You declare the variable with using, and its scope lasts until the end of the containing block (usually the end of the method). Dispose() is automatically called when the scope is exited.

```
public void DoLogging()  {
using var logger = new FileLogger("log.txt");
logger.Log("This is a message.");

// logger can be used throughout the rest of the method.
if (true)
{
    logger.Log("Another message.");
}

} // logger.Dispose() is automatically called here, at the end of the DoLogging method.
```

Best Practice: Always use a using statement or declaration when working with any object that implements IDisposable. It is the safest and cleanest way to ensure resources are released correctly.

Lecture 5: Things to Remember

- GC Manages Memory, You Manage Resources
  This is the core takeaway. The Garbage Collector is excellent at managing managed memory (objects on the heap). It knows nothing about unmanaged resources (file handles, database connections, network sockets). You are responsible for releasing unmanaged resources deterministically, and IDisposable is the standard pattern for doing so.
- Avoid Destructors: Favor IDisposable
  Destructors should be seen as a last-resort safety net, not a primary resource management tool. They are non-deterministic, have performance overhead, and complicate garbage collection. Always prefer implementing IDisposable for predictable, immediate cleanup.
- Always Use 'using' for IDisposable Objects
  The using statement/declaration is not optional; it is the best practice for ensuring Dispose() is always called, even in the face of exceptions. It makes your code significantly safer and cleaner.
- **The Full "Dispose Pattern" (Advanced Interview Topic)**
  - **Question:** "How would you correctly implement IDisposable in a class that also has a destructor?"
  - **Ideal Answer:** "You implement the full Dispose Pattern. The class implements IDisposable. You create a `protected virtual void Dispose(bool disposing)` method.
    - The public `Dispose()` method calls `Dispose(true)` and then `GC.SuppressFinalize(this)`. SuppressFinalize tells the GC that you've already cleaned up, so it doesn't need to run the destructor, which is a key performance optimization.
    - The destructor (~ClassName()) calls `Dispose(false)`.
    - The `Dispose(bool disposing)` method does the actual work. Inside an `if (disposing)` block, it cleans up **both managed and unmanaged** resources. The part outside this check (or in an `else`) cleans up **only unmanaged** resources. This pattern ensures that resources are cleaned up correctly whether Dispose is called manually (the normal case) or by the finalizer thread (the backup case)."

Section 20: Delegates and Events

Lecture 1: Intro to Delegates

Introduction

A **delegate** is a special and powerful type in C# that acts as a **type-safe function pointer**. Unlike regular data types that hold data, a delegate variable holds a reference to one or more methods. The key feature is that a delegate is **type-safe**: it can only hold references to methods that have a matching signature (the same return type and the same parameter types).

Delegates are the foundation for many advanced C# features, including event handling, callbacks, and LINQ. They allow you to treat methods as first-class citizens—meaning you can pass a method as an argument to another method, store it in a variable, and call it later. This enables a much more flexible and decoupled software architecture.

How It Works: Defining a Delegate Type

You define a delegate using the `delegate` keyword. This line of code does not create a method itself; it creates a new **type**, a blueprint for a reference that can point to other methods.

Syntax: `access_modifier delegate return_type DelegateName(parameter_list);`

Example: Let's define a delegate type that can hold a reference to any method that takes two integers as input and returns an integer.

```
public delegate int MathOperation(int a, int b);
```

This creates a new type named `MathOperation`. We can now declare variables of this type, just like we would with `int` or `string`.

Why and When to Use It

Use a delegate when you want to make a piece of your code extensible or when you need to decouple components.

- **Callbacks:** You can pass a delegate to a long-running method. When the method is finished, it "calls back" to your delegate to notify you that the work is done.
- **Pluggable Methods:** You can write a method that accepts a delegate as a parameter, allowing the caller to "plug in" their own logic (e.g., a custom sorting algorithm).
- **Event Handling:** Delegates are the underlying mechanism for events, allowing one object to notify other objects that something has happened.

Lecture 2: Single Cast Delegates

Introduction

A **single-cast delegate** is a delegate that holds a reference to exactly one method. When you invoke it, that single method is executed. This is the simplest form of delegate usage.

How It Works: The Four Steps

Using a single-cast delegate involves four distinct steps:

1. **Define the Delegate Type:** Create the blueprint for the method signature.
2. **Define a Matching Method:** Create one or more methods that have the exact same return type and parameter list as the delegate.
3. **Instantiate the Delegate:** Create an object of the delegate type, passing the name of your matching method to its constructor. This "wires up" the delegate to point to your method.
4. **Invoke the Delegate:** Call the delegate variable as if it were a method itself.

Example

Let's use the MathOperation delegate we defined earlier.

```
// 1. Define the delegate type (already done).
public delegate int MathOperation(int a, int b);

public class Calculator
{
    // 2. Define methods that match the delegate's signature.
    public int Add(int x, int y)
    {
        Console.WriteLine("Add method called.");
        return x + y;
    }

    public int Subtract(int x, int y)
    {
        Console.WriteLine("Subtract method called.");
        return x - y;
    }
}

// --- In your Main method ---
Calculator calc = new Calculator();

// 3. Instantiate the delegate to point to the 'Add' method.
MathOperation op = new MathOperation(calc.Add);

// 4. Invoke the delegate. This will execute the calc.Add method.
int result = op(10, 5);
Console.WriteLine($"Result: {result}"); // Output: Result: 15

// You can re-assign the delegate to point to a different method.
op = new MathOperation(calc.Subtract);
result = op(10, 5);
Console.WriteLine($"Result: {result}"); // Output: Result: 5
```

Lecture 3: Multi Cast Delegates

Introduction

A **multicast delegate** is a delegate that holds references to **multiple** methods in an internal list, called an **invocation list**. When you invoke a multicast delegate, all the methods in its list are called in the order they were added. All delegate types in C# are implicitly multicast-capable.

How It Works: The +, +=, -, and -= Operators

You don't use a special type for multicast delegates; you build them by combining single-cast delegates.

- **+ or += (Subscribe):** These operators are used to add a method reference to the delegate's invocation list.
- **- or -= (Unsubscribe):** These operators are used to remove a method reference from the list.

Important Note on Return Values This is a critical interview point. If a multicast delegate has a non-void return type (e.g., int), and you invoke it, multiple methods will run and each will return a value. What is the final result of the invocation? The answer is: it is the **return value of the last method** in the invocation list. All previous return values are discarded. For this reason, multicast delegates are most commonly used with methods that have a void return type.

Example

Let's create a logging system where a single message can be sent to multiple destinations (the console and a file).

```
// A delegate for logging methods. 'void' is ideal for multicast.
public delegate void LogAction(string message);

public class Logger
{
    public void LogToConsole(string msg)
    {
        Console.WriteLine($"CONSOLE LOG: {msg}");
    }
```

```
    public void LogToFile(string msg)
    {
        // In a real app, this would write to a file.
        Console.WriteLine($"FILE LOG: {msg}");
    }
}

// --- In your Main method ---
Logger logger = new Logger();

// Start with an empty delegate.
LogAction loggers = null;

// Use += to subscribe the first method.
loggers += logger.LogToConsole;

// Use += again to subscribe the second method.
loggers += logger.LogToFile;

// Now, invoking 'loggers' will call BOTH methods.
loggers("System is starting up.");

Console.WriteLine("\n--- Removing Console Logger ---");

// Use -= to unsubscribe a method.
loggers -= logger.LogToConsole;

// Now, invoking 'loggers' will only call the remaining method.
loggers("System is shutting down.");
```

Output:

```
CONSOLE LOG: System is starting up.
FILE LOG: System is starting up.

--- Removing Console Logger ---
FILE LOG: System is shutting down.
```

Lecture 4: Events

Introduction

An **event** is a mechanism that enables a class (the **publisher**) to provide notifications to other classes (the **subscribers**) when something interesting happens. Events are a specialized language feature that uses delegates to manage subscriptions in a safe and encapsulated way.

The Problem Events Solve: Encapsulation

Why not just use a public multicast delegate?

`public class BadPublisher { public LogAction SomethingHappened; }`
This is a terrible idea for two reasons:

1. **No Encapsulation:** Any external class can completely wipe out all existing subscribers by writing `myPublisher.SomethingHappened = null;`.
2. **No Control:** Any external class can directly invoke the delegate (`myPublisher.SomethingHappened("Fake event");`), pretending to be the publisher.

The `event` keyword solves these problems. It creates a wrapper around the delegate, exposing only the safe += (subscribe) and -= (unsubscribe) operations to the public. Only the publishing class itself is allowed to invoke the event.

How It Works: The Full Implementation

This shows what the compiler does behind the scenes and is useful when you need custom subscription logic.

1. You declare a **private** delegate field to hold the subscribers.
2. You declare a **public event** using the `event` keyword.
3. You implement the `add` and `remove` accessors for the event, which simply use += and -= on your private delegate field.

Example Let's create a Stock class that raises an event whenever its price changes.

```
// A delegate for the event handler.
public delegate void PriceChangedHandler(decimal oldPrice, decimal
newPrice);

public class Stock
{
    private string _symbol;
    private decimal _price;

    // 1. A private delegate instance to hold the subscribers.
    private PriceChangedHandler _priceChanged;

    // 2. The public event wrapper.
    public event PriceChangedHandler PriceChanged
    {
        // 3. The 'add' accessor is called when a subscriber uses +=
        add
        {
            _priceChanged += value;
        }
        // 3. The 'remove' accessor is called when a subscriber uses -=
        remove
        {
            _priceChanged -= value;
        }
    }

    public decimal Price
    {
        get { return _price; }
        set
        {
            if (_price != value)
            {
                decimal oldPrice = _price;
                _price = value;
                // 4. Raise the event ONLY from within this class.
                // Check for null to ensure there are subscribers.
                _priceChanged?.Invoke(oldPrice, _price);
            }
        }
    }
```

```
    public Stock(string symbol, decimal startingPrice)
    {
        _symbol = symbol;
        _price = startingPrice;
    }
}
```

Lecture 5: Auto-Implemented Events

Introduction

Just as auto-properties simplify property declaration, **auto-implemented events** provide a concise syntax for declaring an event without needing to manually write the add/remove logic or the private delegate field.

How It Works: Compiler Magic

When you use the auto-implemented event syntax, the C# compiler does the work for you.

When you write this: public event PriceChangedHandler PriceChanged;

The compiler automatically generates this behind the scenes:

1. A private, compiler-generated delegate field to hold the subscribers.
2. The full add and remove accessor implementations that safely add and remove subscribers from that hidden field.

When to Use It

This is the standard and default way to declare events in modern C#. You should always use this concise syntax unless you need to add custom, specialized logic to the subscription or unsubscription process (which is extremely rare).

Example: Refactoring the Stock Class

Let's simplify our Stock class using an auto-implemented event.

```
public delegate void PriceChangedHandler(decimal oldPrice, decimal
newPrice);

public class Stock
{
   private string _symbol;
   private decimal _price;

   // Much cleaner! The compiler creates the backing delegate field for us.
   public event PriceChangedHandler PriceChanged;

   public decimal Price
   {
      get { return _price; }
      set
      {
         if (_price != value)
         {
            decimal oldPrice = _price;
            _price = value;
            // We still raise the event the same way.
            PriceChanged?.Invoke(oldPrice, _price);
         }
      }
   }

   public Stock(string symbol, decimal startingPrice)
   {
      _symbol = symbol;
      _price = startingPrice;
   }
}

// A subscriber class
public class StockMonitor
{
   public void OnPriceChanged(decimal oldPrice, decimal newPrice)
   {
      Console.WriteLine($"Stock price changed from {oldPrice:C} to
{newPrice:C}");
   }
}
```

```
// --- In your Main method ---
Stock tesla = new Stock("TSLA", 900);
StockMonitor monitor = new StockMonitor();

// Subscribe the monitor's method to the event using +=
tesla.PriceChanged += monitor.OnPriceChanged;

// This assignment will trigger the event and call the subscriber's method.
tesla.Price = 925;
```

Lecture 6: Anonymous Methods

Introduction

Anonymous methods were introduced in C# 2.0 as a way to write an "inline" block of code directly where a delegate instance is expected. It allows you to define a method's implementation without having to formally declare it with a name in your class. It's a way to attach a small, one-off piece of logic to a delegate or event.

How It Works: The `delegate` Keyword as an Expression

You create an anonymous method using the `delegate` keyword, followed by an optional parameter list and a method body in curly braces.

Syntax: `delegate (parameter_list) { ... method body ... }`

You assign this entire expression to a delegate variable or use it to subscribe to an event.

Why and When to Use It

Anonymous methods were created to reduce the "ceremony" of event handling. Before them, even for a trivial, one-line event handler, you had to create a full, named method in your class. Anonymous methods allowed you to place that logic right where it was needed.

Important Note: While it's important to recognize this syntax in older C# codebases, anonymous methods have been **almost entirely superseded by the more concise and powerful Lambda Expressions** (covered in the next lecture) in modern C# development.

Example Here is how you might see it used in older UI code.

```
// Assume we have a button in a UI application.
Button myButton = new Button();

// Using an anonymous method to handle the Click event.
myButton.Click += delegate (object sender, EventArgs e)
{
    MessageBox.Show("Button was clicked!");
};
```

Lecture 7: Lambda Expressions

Introduction

Lambda expressions, introduced in C# 3.0, are a much more concise, expressive, and powerful syntax for creating anonymous functions. They are the heart of functional programming in C# and are the **preferred modern way** to write inline code wherever a delegate is required. They have largely replaced anonymous methods for all new development.

How It Works: The Lambda Operator =>

The core of a lambda expression is the lambda operator =>, which is read as "goes to". It separates the input parameters on the left from the expression or statement block on the right.

Syntax: (input_parameters) => expression_or_statement_block

Key Features:

- **Type Inference:** The compiler can almost always infer the types of the input parameters, so you usually don't need to write them. (int x, int y) can just be written as (x, y).
- **Expression Lambdas:** For a single-line lambda that returns a value, you can omit the curly braces {} and the return keyword. The compiler handles it for you. This makes them extremely concise.
- **Statement Lambdas:** For lambdas with multiple lines of code, you use curly braces just like a regular method body.

Example: Refactoring with Lambdas

Let's refactor our previous delegate and anonymous method examples.

```
// 1. For a MathOperation delegate
public delegate int MathOperation(int a, int b);

// Old way: new MathOperation(Add);
// Lambda way: The compiler understands that a and b are ints.
MathOperation add = (a, b) => a + b; // Single-line expression lambda
Console.WriteLine(add(10, 5)); // Output: 15

// 2. For the button click event handler
// The compiler infers that sender is an object and e is EventArgs.
myButton.Click += (sender, e) =>
{
    // Multi-line statement lambda
    MessageBox.Show("Button clicked! (from a lambda)");
    // You can have more lines of code here.
};
```

Lecture 8: Inline Lambda Expressions

Introduction

While you can assign a lambda expression to a delegate variable, their true power is realized when they are used **inline**, passed directly as an argument to a method that accepts a delegate. This is the most common pattern you will see in modern C#, especially with LINQ.

How It Works

Many methods in the .NET framework, particularly LINQ extension methods, are designed to take a delegate as a parameter to allow the caller to inject custom logic. Instead of defining a named method or even a separate delegate variable, you can write the lambda expression directly inside the method call's parentheses.

When to Use It

This is the standard pattern for working with LINQ. Methods like .Where(), .Count(), .Select(), .OrderBy(), etc., all take delegates (usually Func or Action) to perform their work.

Example: Using LINQ

Let's find all the even numbers in a list and count how many there are.

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// --- Using an inline lambda expression with Count() ---
// The Count method accepts a delegate that takes an int and returns a bool.
// We provide the logic for that delegate directly as a lambda expression.
int evenCount = numbers.Count(n => n % 2 == 0);
// The expression "n => n % 2 == 0" is read as:
// "Given a number n, return true if n divided by 2 has a remainder of 0."
Console.WriteLine($"There are {evenCount} even numbers."); // Output: 5

// --- Using an inline lambda expression with Where() ---
// The Where method filters a collection based on a condition.
var evenNumbers = numbers.Where(n => n % 2 == 0);
Console.WriteLine("The even numbers are: " + string.Join(", ", evenNumbers));
```

Lecture 9: Func

Introduction

Func is a built-in, generic delegate type provided by the .NET framework. Its purpose is to represent a method that takes zero or more input parameters and **always returns a value**. Using Func saves you from having to define your own custom delegate type for methods that return a value.

How It Works: Signature

The Func delegate has multiple overloaded forms. The key rule is that the **last generic type parameter is always the return type**.

- Func<TResult>: Represents a method that takes **0** parameters and returns a value of type TResult.
- Func<T, TResult>: Represents a method that takes **1** parameter of type T and returns a value of type TResult.
- Func<T1, T2, TResult>: Represents a method that takes **2** parameters of types T1 and T2, and returns a value of type TResult.
- ...and so on, up to 16 input parameters.

Example

Instead of defining public delegate int MathOperation(int a, int b);, we can just use the built-in Func.

```
// This Func represents a method that takes two ints and returns an int.
// It is functionally identical to our custom MathOperation delegate.
Func<int, int, int> addOperation = (x, y) => x + y;
Console.WriteLine($"Result: {addOperation(20, 22)}"); // Output: 42

// This Func represents a method that takes no parameters and returns a string.
Func<string> getGreeting = () => "Hello, World!";
Console.WriteLine(getGreeting());
```

Lecture 10: Action

Introduction

Action is another built-in, generic delegate type. It is the counterpart to Func. Its purpose is to represent a method that takes zero or more input parameters but **does not return a value** (i.e., its return type is void).

How It Works: Signature

Like Func, Action has multiple overloaded forms. Since it never returns a value, all its generic type parameters represent input parameters.

- Action: Represents a method that takes **0** parameters and returns void.
- Action<T>: Represents a method that takes **1** parameter of type T and returns void.
- Action<T1, T2>: Represents a method that takes **2** parameters of types T1 and T2, and returns void.
- ...and so on, up to 16 input parameters.

Example

Instead of defining public delegate void LogAction(string message);, we can use the built-in Action.

```
// This Action represents any method that takes a single string and returns void.
Action<string> logToConsole = (message) => Console.WriteLine($"LOG: {message}");
logToConsole("Application has started.");

// This Action represents a method with no parameters and no return value.
Action showMenu = () =>
{
    Console.WriteLine("1. Start Game");
    Console.WriteLine("2. Load Game");
    Console.WriteLine("3. Exit");
};
showMenu();
```

Lecture 11: Predicate

Introduction

A **Predicate** is a specialized, built-in generic delegate type provided by the .NET framework. Its specific purpose is to represent a method that takes a single input parameter and returns a **bool**. The name "predicate" comes from formal logic, where it means a statement about a subject that can be evaluated as either true or false. In programming, a predicate is used to test if an item meets a certain condition.

How It Works

The Predicate<T> delegate is functionally equivalent to Func<T, bool>. It's simply a more descriptive name for this very common scenario.

Signature: public delegate bool Predicate<in T>(T obj);

It takes one parameter of type T and must return a bool.

Why and When to Use It

Use a Predicate<T> when the sole purpose of your delegate is to test a condition. It makes the code more self-documenting because the name Predicate clearly communicates that a test is being performed. Many built-in .NET methods that filter or search collections specifically accept a Predicate<T>.

Common Examples

The List<T>.FindAll() method is a perfect example. It takes a Predicate<T> and returns a new list containing all the elements that pass the test (i.e., for which the predicate returns true).

```
// Our list of data
List<int> numbers = new List<int> { 10, 45, 52, 88, 95, 23 };

// --- Using a named method ---
// 1. Define a method that matches the Predicate<int> signature.
public bool IsGreaterThan50(int number)
{
    return number > 50;
}

// 2. Create a predicate instance pointing to that method.
Predicate<int> predicate = IsGreaterThan50;
List<int> highNumbers = numbers.FindAll(predicate);

// --- Using a Lambda Expression (Modern and Preferred) ---
// Pass the predicate's logic directly as a lambda.
// This is much more concise.
List<int> highNumbersLambda = numbers.FindAll(n => n > 50);

Console.WriteLine("Numbers greater than 50:");
foreach (int num in highNumbersLambda)
{
    Console.WriteLine(num); // Output: 52, 88, 95
}
```

Lecture 12: EventHandler

Introduction

EventHandler and its generic counterpart, **EventHandler**, are the standard, official delegate types that should be used for all events in the .NET framework. By adhering to this standard, you ensure that your custom events are predictable and can be easily consumed by other developers and integrated with UI frameworks that expect this pattern.

How It Works: The Standard Event Signature

This pattern defines a very specific method signature that all event handlers should follow:

void MethodName(object sender, EventArgs e)

- **object sender**: This parameter always holds a reference to the **object that raised the event**. This allows the subscriber to know where the event came from and, if necessary, to interact with the publisher object (e.g., to get more information from it).
- **EventArgs e**: This parameter is an object that contains **event data**. The base EventArgs class is empty and is used when an event doesn't need to pass any extra information. For events that do need to pass data, you create a custom class that inherits from EventArgs.

The Generic EventHandler<TEventArgs>

This is the modern, type-safe version. TEventArgs must be a type that inherits from EventArgs. This approach is better because it avoids the need for the subscriber to cast the e parameter to get the custom event data.

Example: The Stock Class Revisited

Let's refactor our Stock class to use this standard pattern.

```
// 1. Create a custom class for our event data, inheriting from EventArgs.
public class PriceChangedEventArgs : EventArgs
{
   public decimal OldPrice { get; }
```

```
    public decimal NewPrice { get; }

    public PriceChangedEventArgs(decimal oldPrice, decimal newPrice)
    {
        OldPrice = oldPrice;
        NewPrice = newPrice;
    }
}

// 2. The publisher class.
public class Stock
{
    private decimal _price;

    // 3. The event is now of the standard generic type.
    public event EventHandler<PriceChangedEventArgs> PriceChanged;

    public decimal Price
    {
        get { return _price; }
        set
        {
            if (_price != value)
            {
                decimal oldPrice = _price;
                _price = value;
                // 4. Raise the event by calling the Invoke method.
                OnPriceChanged(new PriceChangedEventArgs(oldPrice,
_price));
            }
        }
    }

    // Helper method to raise the event
    protected virtual void OnPriceChanged(PriceChangedEventArgs e)
    {
        PriceChanged?.Invoke(this, e); // 'this' is the sender
    }

    public Stock(decimal startingPrice) { _price = startingPrice; }
}

// --- In the subscriber ---
// The event handler method now matches the standard signature.
```

```
public void Stock_PriceChanged(object sender, PriceChangedEventArgs
e)
{
   Stock stockThatChanged = (Stock)sender; // We can get the sender
   Console.WriteLine($"Price changed from {e.OldPrice:C} to
{e.NewPrice:C}");
}
```

Lecture 13: Expression Trees

Introduction

This is an advanced but powerful topic. An **Expression Tree** is a special data structure that represents code as **data**. When you assign a lambda expression to a delegate type like Func, the compiler converts it into executable IL code. However, when you assign the same lambda expression to a variable of type Expression<Func<...>>, the compiler does something completely different: it translates the lambda into a tree-like object structure where each node represents a part of the code—an operation, a parameter, a constant, etc.

How It Works

Consider this lambda: x => x > 5.

- **As a Func<int, bool>:** It becomes compiled, executable code that takes an integer and returns true or false.
- **As an Expression<Func<int, bool>>:** It becomes a data structure that looks something like this:
    - **Root Node:** A "GreaterThan" binary expression.
    - **Left Child:** A "ParameterExpression" named "x".
    - **Right Child:** A "ConstantExpression" with the value 5.

Why and When to Use It

The primary use case is for libraries that need to **analyze, inspect, and translate code** into another language or format. The most famous real-world example is **LINQ to SQL** and **Entity Framework Core**. When you write a query like:

var users = dbContext.Users.Where(u => u.Age > 18);
The .Where() method does not take a Func<User, bool>. It takes an Expression<Func<User, bool>>. Entity Framework then inspects the expression tree for u => u.Age > 18, translates it into the SQL string "WHERE Age > 18", and sends that SQL query to the database server. This allows you to write database queries in C# instead of raw SQL.

Interview Perspective

- **Question:** "What is an Expression Tree and why is it used?"
- **Ideal Answer:** "An expression tree is a data structure that represents code as data. Instead of compiling a lambda into executable code, the compiler converts it into an object model that can be inspected and traversed. This is primarily used by libraries like Entity Framework Core to translate C# LINQ queries into another language, such as SQL, which can then be executed on a database."

Lecture 14: Expression-Bodied Members

Introduction

Expression-bodied members are a concise C# syntax feature that allows you to define the body of a method, property, constructor, or other member using a lambda-like => syntax. This is purely syntactic sugar for members that consist of only a single statement or expression.

How It Works

It provides a shorter, more readable alternative to using a full block body { ... } for simple members.

Common Examples

- **Methods:**
  - **Before:** public int Add(int a, int b) { return a + b; }
  - **After:** public int Add(int a, int b) => a + b;
- **Read-Only Properties:**
  - **Before:** public string FullName { get { return $"{FirstName} {LastName}"; } }
  - **After:** public string FullName => $"{FirstName} {LastName}";
- **Constructors (C# 7.0+):**
  - **Before:** public Person(string name) { this.Name = name; }
  - **After:** public Person(string name) => this.Name = name;
- **ToString() Overrides:**
  - **Before:** public override string ToString() { return Name; }
  - **After:** public override string ToString() => Name;

Best Practice: Use expression-bodied members whenever a method or property body is a single, clear line of code. It reduces verbosity and can make classes easier to read at a glance.

Lecture 15: Switch Expression

Introduction

A **switch expression**, introduced in C# 8.0, is a modern, concise, and expression-based alternative to the traditional switch statement. Since it is an **expression**, it must always produce a value.

How It Works: Syntax

The new syntax is much more compact:

- It starts with the variable being switched on, followed by the switch keyword.
- It uses curly braces { ... } to contain the arms.
- Each arm is a pattern => result, (a pattern, a lambda arrow, the result for that pattern, and a comma).
- The _ (discard pattern) is used as the default case and is required if the compiler cannot prove that all possible inputs are handled.

Why and When to Use It

Use a switch expression when you need to get a single value based on a series of pattern matches. It is far cleaner than a traditional switch statement where every case block contains a return statement.

Example: Refactoring a Grade Calculator

```
// --- Traditional switch statement ---
public string GetGrade(int score)
{
    switch (score)
    {
        case >= 90:
            return "A";
        case >= 80:
            return "B";
        case >= 70:
            return "C";
        default:
            return "F";
    }
}

// --- Modern switch expression ---
public string GetGradeModern(int score) => score switch
{
    >= 90 => "A",
    >= 80 => "B",
```

```
    >= 70 => "C",
    _ => "F"  // The discard pattern handles the default case.
};
```

Lecture 16: Things to Remember

- **Delegates are Type-Safe Function Pointers:** This is the foundation. They allow you to treat methods as variables, enabling powerful callback and event-driven architectures.
- **Events are Encapsulated Delegates:** The event keyword is a crucial wrapper around a delegate. It restricts access to only += (subscribe) and -= (unsubscribe) from the outside, while only allowing the publishing class to invoke the event.
- **Lambdas are the Modern Standard:** For any inline, anonymous function, lambda expressions (=>) are the preferred, concise, and powerful syntax. They have largely replaced the older anonymous method syntax.
- **Use Func, Action, and Predicate:** Don't define your own custom delegates if you don't have to. These three built-in generic delegates cover over 99% of common use cases and are universally understood by other C# developers.
- **The Standard Event Pattern (EventHandler<T>):** When creating custom events, always use the EventHandler or generic EventHandler<TEventArgs> delegate types. This ensures your components are consistent with the rest of the .NET framework.
- **Expressions vs. Statements:** Modern C# features like Expression-Bodied Members and Switch Expressions are part of a trend in the language to favor **expressions** (which produce a value) over **statements** (which perform an action). This often leads to more concise, readable, and functional-style code.

Section 22: Collections

Lecture 1: Introduction to Collections

Introduction

While arrays are fundamental, they have one major limitation: they are **fixed-size**. Once you create an array, you cannot change its length. In most real-world applications, you don't know ahead of time how many items you'll need to store, or that number might need to change as the program runs.

Collections are classes designed to solve this problem. They are dynamic data structures created to efficiently store, manage, and manipulate groups of objects. Their key advantage over arrays is that they can **grow and shrink** in size as needed. The .NET framework provides a rich set of collection classes, found primarily in the System.Collections and System.Collections.Generic namespaces.

Why and When to Use Them

You should use a collection whenever you are working with a group of objects and:

- You don't know the number of items beforehand.
- You need to frequently add or remove items.
- You need more powerful searching, sorting, and manipulation capabilities than what a basic array offers.

Best Practice: Generic vs. Non-Generic

The System.Collections namespace contains older, non-generic collections like ArrayList. These are legacy types that store everything as object, leading to a lack of type safety and performance issues due to boxing. You should **always prefer the modern, generic collections** found in the System.Collections.Generic namespace, such as List<T> and Dictionary<TKey, TValue>. They provide full type safety at compile time and significantly better performance.

Lecture 2: List<T>

Introduction

The **List<T>** class is the most common, general-purpose, and important collection class in .NET. It is essentially a **dynamically-sized array**. It gives you the benefits of an array—fast, indexed access to elements—with the flexibility of being able to easily add and remove items. If you need a simple list of items and you're not sure which collection to use, List<T> is almost always the right choice to start with.

How It Works: The Internal Array and Capacity

This is a critical concept for understanding its performance. A List<T> is a clever wrapper around an underlying array.

1. When you create a new List<T>, it allocates a small internal array (e.g., with a capacity of 4).
2. As you add items, they are placed into this internal array.
3. When you try to add an item and the internal array is full, the List<T> performs a **resizing operation**: it creates a **new, larger array** (typically double the size of the old one), **copies all the elements** from the old array to the new one, and then discards the old array.

Essential Properties

- **Count**: Gets the number of elements **currently contained** in the list. This is what you will use most often.
- **Capacity**: Gets or sets the total number of elements the internal data structure can hold **without resizing**. Count is always less than or equal to Capacity. Understanding this distinction is important for performance tuning with very large lists.

Example

```
List<string> shoppingList = new List<string>();
Console.WriteLine($"Initial Count: {shoppingList.Count}, Initial Capacity:
{shoppingList.Capacity}");

shoppingList.Add("Apples");
shoppingList.Add("Bananas");

Console.WriteLine($"After adding 2 items: Count: {shoppingList.Count},
Capacity: {shoppingList.Capacity}");
// The capacity will have been set to a default, likely 4.
```

Lecture 3: Add and AddRange

Introduction

These are the primary methods for adding new elements to the end of a List<T>.

Add(T item)

- **What it does:** Adds a single item to the **end** of the list.
- **How it Works:** It places the new item at the index specified by the current Count, and then increments the Count. If the Count becomes equal to the Capacity before adding, it will trigger the resizing operation mentioned in the previous lecture.

AddRange(IEnumerable<T> collection)

- **What it does:** Efficiently adds all the elements from another collection (like an array or another list) to the **end** of the list.
- **How it Works:** It's more efficient than calling Add in a loop because it can calculate the total new size required, perform a single resizing operation if needed, and then copy the elements over in a highly optimized block operation.

Example

```
// Start with a list of numbers.
List<int> numbers = new List<int> { 10, 20 };

// Use Add to add a single element.
numbers.Add(30);
// List is now: { 10, 20, 30 }

// Create an array to add multiple elements.
int[] moreNumbers = { 40, 50 };

// Use AddRange to add all elements from the array.
numbers.AddRange(moreNumbers);
// List is now: { 10, 20, 30, 40, 50 }

Console.WriteLine("Final list: " + string.Join(", ", numbers));
```

Lecture 4: Insert and InsertRange

Introduction

While Add and AddRange always work at the end of the list, Insert and InsertRange give you the power to add elements at a **specific index**.

Insert(int index, T item)

- **What it does:** Inserts a single item into the list at the specified zero-based index.
- **How it Works:** This can be a **slow operation** for large lists. To insert an item at index i, the list must shift all existing elements from index i to the end one position to the right to make room. The cost of this operation grows as the list gets larger.

InsertRange(int index, IEnumerable<T> collection)

- **What it does:** Inserts the elements of another collection into the list at the specified index.
- **Performance:** Like Insert, this can also be slow due to the shifting of existing elements.

Example

```
List<string> guests = new List<string> { "Alice", "Charlie", "David" };
Console.WriteLine("Original guest list: " + string.Join(", ", guests));

// We forgot Bob! Let's insert him at the correct position (index 1).
guests.Insert(1, "Bob");

// The list is now: { "Alice", "Bob", "Charlie", "David" }
// "Charlie" and "David" were shifted one position to the right.
Console.WriteLine("Corrected guest list: " + string.Join(", ", guests));
```

Lecture 5: Remove, RemoveAt, RemoveRange, RemoveAll, Clear

Introduction

These methods provide a comprehensive toolkit for deleting elements from a List<T>.

Methods for Deletion

- **Remove(T item)**
  - **What it does:** Searches for the specified item and removes the **first occurrence** it finds.
  - **Return Value:** It returns a bool—true if the item was found and removed, false otherwise.

- **RemoveAt(int index)**
    - **What it does:** Removes the element at the specified zero-based index.
    - **Performance:** Like Insert, this can be a slow operation for large lists because all subsequent elements must be shifted one position to the left to fill the gap.
- **RemoveRange(int index, int count)**
    - **What it does:** Removes a block of count elements starting at index.
- **RemoveAll(Predicate<T> match)**
    - **What it does:** This is a very powerful method that removes **all elements** that match the condition defined by the provided predicate.
    - **Usage:** You typically provide the condition as a lambda expression.
- **Clear()**
    - **What it does:** Removes **all** elements from the list.
    - **How it Works:** It sets the Count property to 0 but does not change the Capacity. The internal array is not deallocated, which is an optimization if you intend to immediately start adding new items to the list again.

Example

```
List<int> scores = new List<int> { 10, 25, 88, 42, 95, 15, 88 };
Console.WriteLine("Original: " + string.Join(", ", scores));

// Remove the first occurrence of the number 88.
scores.Remove(88);
Console.WriteLine("After Remove(88): " + string.Join(", ", scores));

// Remove the element at index 2 (which is now 42).
scores.RemoveAt(2);
Console.WriteLine("After RemoveAt(2): " + string.Join(", ", scores));

// Remove all scores that are less than 20.
scores.RemoveAll(score => score < 20);
Console.WriteLine("After RemoveAll(< 20): " + string.Join(", ", scores));

// Clear the rest.
scores.Clear();
Console.WriteLine($"After Clear: Count is {scores.Count}, Capacity is
{scores.Capacity}");
```

Lecture 6: IndexOf, BinarySearch, Contains

Introduction

These methods are used to find elements within a list.

Methods for Searching

- **IndexOf(T item)**
    - **What it does:** Searches for an item and returns the zero-based index of its **first occurrence**.
    - **How it Works:** It performs a **linear search** (O(n)), starting from the beginning.
    - **Return Value:** If the item is found, it returns the index. If not found, it returns **-1**.

- **Contains(T item)**
    - **What it does:** A simple helper that returns a bool indicating whether an item exists in the list.
    - **How it Works:** It's a convenient wrapper that internally calls IndexOf and checks if the result is not -1.
- **BinarySearch(T item)**
    - **What it does:** A highly efficient search method.
    - **Prerequisite:** Just like with arrays, the list **MUST BE SORTED** before you call this method. If it's not sorted, the results are unpredictable and incorrect.
    - **How it Works:** It uses the "divide and conquer" binary search algorithm (O(log n)), which is much faster than a linear search for large lists.
    - **Return Value:** If found, it returns the index. If not found, it returns a negative number that can be used to determine where the item would be inserted.

Example

```
List<string> planets = new List<string> { "Mercury", "Venus", "Earth",
"Mars" };

// Contains
bool hasEarth = planets.Contains("Earth");
Console.WriteLine($"Does the list contain Earth? {hasEarth}"); // True

// IndexOf
int marsIndex = planets.IndexOf("Mars");
Console.WriteLine($"The index of Mars is: {marsIndex}"); // 3

int plutoIndex = planets.IndexOf("Pluto");
Console.WriteLine($"The index of Pluto is: {plutoIndex}"); // -1

// BinarySearch requires a sorted list.
planets.Sort(); // Now the list is { "Earth", "Mars", "Mercury", "Venus" }
Console.WriteLine("\nSorted list: " + string.Join(", ", planets));
int venusIndex = planets.BinarySearch("Venus");
Console.WriteLine($"After sorting, the index of Venus is: {venusIndex}"); //
3
```

Lecture 7: Sort and Reverse

Introduction

These are two essential instance methods on the List<T> class that allow you to rearrange the order of elements directly within the list. Both of these methods modify the list **in place**, meaning they do not create a new list but rather reorder the elements of the original one.

Sort()

- **What it does:** Sorts the elements in the entire list.
- **How it Works:** It uses a highly efficient introspective sort algorithm (a hybrid of Quicksort, Heapsort, and Insertion sort). The sorting behavior depends on the type T:
    - For primitive types like int and string, it uses their natural default order (ascending numerical or alphabetical).
    - For your own custom class objects, the class must implement the **IComparable<T>** interface to define its "natural" sort order. If it doesn't, calling Sort() will throw an exception. Alternatively, you can pass a custom sorting logic (an IComparer<T> or a Comparison<T> delegate) as a parameter to the Sort method.
- **When to Use It:** Whenever you need to order the elements of a list, for example, to display a list of names alphabetically or a list of scores from lowest to highest.

Reverse()

- **What it does:** Reverses the sequence of the elements in the entire list. The first element becomes the last, the second becomes the second-to-last, and so on.
- **How it Works:** It swaps the elements in place, making it a very efficient operation.

Example

```
List<int> scores = new List<int> { 88, 54, 95, 72, 68 };
Console.WriteLine("Original list:  " + string.Join(", ", scores));

// Sort the list in place (ascending).
scores.Sort();
Console.WriteLine("After Sort():   " + string.join(", ", scores));

// Reverse the now-sorted list to get descending order.
scores.Reverse();
Console.WriteLine("After Reverse(): " + string.Join(", ", scores));
```

Output:

```
Original list:  88, 54, 95, 72, 68
After Sort():   54, 68, 72, 88, 95
After Reverse(): 95, 88, 72, 68, 54
```

Lecture 8: ToArray and ForEach

Introduction

These are two useful utility methods on the List<T> class for converting the list and performing actions on its elements.

ToArray()

- **What it does:** Creates and returns a **new array** of type T[] containing all the elements from the list in the same order.
- **Why and When to Use It:**
    - When you need to pass your collection to a legacy method or an API that specifically requires an array.
    - When you want a fixed "snapshot" of the list at a certain point in time. The returned array is a separate copy, so future changes to the original list will not affect the array.

ForEach(Action<T> action)

- **What it does:** Executes a given action on **each element** of the list. The action is provided as a delegate, almost always as a concise lambda expression.
- **How it differs from a foreach loop:** This is a common interview point. A foreach loop is a C# language construct that works on any collection that implements IEnumerable<T>. The ForEach() method is a specific method that **only exists on the List<T> class**. While they can achieve similar results, the foreach loop is generally considered more readable and is universally applicable, making it the preferred choice in most style guides.

Example

```
List<string> technologies = new List<string> { "C#", "SQL", "Azure" };

// --- Using ToArray ---
string[] techArray = technologies.ToArray();
Console.WriteLine($"The type of techArray is:
{techArray.GetType().Name}");

// --- Using ForEach ---
Console.WriteLine("\nPrinting items using ForEach():");
technologies.ForEach(tech => Console.WriteLine($"- {tech}"));
```

Lecture 9: The Find Family of Methods

Introduction

This group of powerful methods allows you to search a list for elements that match a specific condition. They all take a Predicate<T> as a parameter, which is a delegate that takes an element T and returns a bool. You will almost always provide this condition as a lambda expression.

The Methods

- **Exists(Predicate<T> match)**: Returns true if **at least one** element in the list matches the predicate. It stops searching as soon as it finds a match.
- **Find(Predicate<T> match)**: Returns the **first element** that matches the predicate. If no match is found, it returns the default value for the type T (e.g., null for objects, 0 for int).
- **FindLast(Predicate<T> match)**: Same as Find, but it starts searching from the **end** of the list and works backwards.
- **FindAll(Predicate<T> match)**: Returns a **new List<T>** containing **all elements** from the original list that match the predicate.
- **FindIndex(Predicate<T> match)**: Returns the zero-based **index** of the first element that matches the predicate. Returns -1 if no match is found.
- **FindLastIndex(Predicate<T> match)**: Same as FindIndex, but it searches backwards from the end.

Example

Let's use a list of Product objects to demonstrate these methods.

```
public class Product { public string Name; public decimal Price; }

List<Product> products = new List<Product>
{
   new Product { Name = "Laptop", Price = 1200 },
   new Product { Name = "Mouse", Price = 25 },
   new Product { Name = "Keyboard", Price = 75 },
   new Product { Name = "Monitor", Price = 300 }
};

// --- Exists ---
bool isAnythingExpensive = products.Exists(p => p.Price > 1000);
Console.WriteLine($"Are there any expensive products?
{isAnythingExpensive}"); // True

// --- Find ---
Product cheapItem = products.Find(p => p.Price < 50);
Console.WriteLine($"First cheap item found: {cheapItem.Name}"); //
Mouse
```

```
// --- FindAll ---
List<Product> affordableItems = products.FindAll(p => p.Price < 200);
Console.WriteLine("\nAffordable Items:");
foreach (var item in affordableItems)
{
    Console.WriteLine($"- {item.Name}"); // Mouse, Keyboard
}

// --- FindIndex ---
int keyboardIndex = products.FindIndex(p => p.Name == "Keyboard");
Console.WriteLine($"\nThe index of Keyboard is: {keyboardIndex}"); // 2
```

Lecture 10: ConvertAll

Introduction

The ConvertAll method is a specific feature of the List<T> class that creates a **new list** by converting every element from the original list to a new type. It applies a specified conversion logic to each element individually.

How It Works

It takes a Converter<TInput, TOutput> delegate as its parameter. This delegate is essentially a Func<TInput, TOutput>—it takes an item of the input type and returns a corresponding item of the output type. ConvertAll iterates through the source list, calls your converter delegate for each item, and adds the returned value to the new list it creates.

When to Use It & The LINQ Select Method

ConvertAll is useful when you need to project or transform every item in a list into a new form. For example, getting a list of all product names from a list of Product objects.

Important Note: In modern C#, this operation is more commonly performed using the **LINQ extension method .Select()**. The .Select() method does the exact same thing but has the advantage of working on **any** collection that implements IEnumerable<T>, not just List<T>. Understanding ConvertAll is good, but for new code, .Select() is generally preferred for its versatility.

Example

Let's convert a list of number strings into a list of actual integers.

```
List<string> numberStrings = new List<string> { "10", "25", "50", "100" };

// Use ConvertAll with a lambda expression that calls int.Parse.
List<int> integers = numberStrings.ConvertAll(s => int.Parse(s));

// The 'integers' list is now a new List<int> { 10, 25, 50, 100 }.

int sum = 0;
foreach(int num in integers)
{
   sum += num;
}
Console.WriteLine($"The sum is: {sum}"); // Output: 185
```

Lecture 11: Dictionary

Introduction

A **Dictionary** is a collection of **key-value pairs**. Think of it like a real-world dictionary: you use a unique word (the **key**) to look up its definition (the **value**). In C#, every key in a Dictionary<TKey, TValue> must be unique, and it is used to store and retrieve its corresponding value. Dictionaries are incredibly powerful and are one ofthe most widely used collection types for high-speed data lookups.

How It Works: The Hash Table

Internally, a Dictionary is implemented using a **hash table**. This is a critical concept for understanding its performance. When you add a key-value pair:

1. The dictionary calls the key's **GetHashCode()** method to generate an integer hash code.
2. It uses this hash code to calculate an index, which points to a specific "bucket" in its internal array.

3. It stores the key-value pair in that bucket.

When you want to retrieve a value using its key, the dictionary calculates the key's hash code again, goes directly to the correct bucket, and then uses the key's Equals() method to find the exact match among the few items in that bucket. This process is extremely fast, with additions, removals, and lookups all averaging **O(1)**, or constant time, performance.

Key Methods and Properties

- **Adding Items:**
  - myDict.Add(key, value): Adds a new pair. Throws an ArgumentException if the key already exists.
  - myDict[key] = value: The indexer syntax. If the key exists, it **overwrites** the existing value. If the key does not exist, it **creates** a new entry. This is often more convenient than Add.
- **Accessing Items:**
  - value = myDict[key]: Retrieves the value for the given key. Throws a KeyNotFoundException if the key does not exist.
- **Safe Access and Checking:**
  - ContainsKey(key): Returns true if the key exists, false otherwise. This is the safe way to check before using the indexer.
  - bool success = TryGetValue(key, out TValue value): The preferred modern way. It tries to get the value. If successful, it returns true and puts the value in the out parameter. If not, it returns false and does not throw an exception.
- **Removing Items:**
  - Remove(key): Removes the key-value pair with the specified key.
- **Iterating:**
  - You can iterate over myDict.Keys, myDict.Values, or the dictionary itself, which gives you KeyValuePair<TKey, TValue> objects.

Example

```
// Create a dictionary to store student IDs and names.
Dictionary<int, string> studentRoster = new Dictionary<int, string>();

// Add items using the Add method and the indexer.
studentRoster.Add(101, "Alice");
studentRoster[102] = "Bob";
studentRoster[103] = "Charlie";

// Access a value.
string studentName = studentRoster[102];
Console.WriteLine($"Student 102 is: {studentName}");

// Safely check for a key before access.
if (studentRoster.ContainsKey(104))
{
    Console.WriteLine(studentRoster[104]);
}
else
{
    Console.WriteLine("Student 104 not found.");
}

// Iterate over the key-value pairs.
Console.WriteLine("\n--- Roster ---");
foreach (KeyValuePair<int, string> student in studentRoster)
{
    Console.WriteLine($"ID: {student.Key}, Name: {student.Value}");
}
```

Lecture 12: SortedList

Introduction

A **SortedList<TKey, TValue>** is a collection of key-value pairs that maintains its elements **sorted by the key**. Whenever you add a new item, it is automatically inserted into the correct position to keep the entire collection sorted.

How It Works: Internal Arrays

Unlike a Dictionary's hash table, a SortedList internally uses **two separate arrays**: one to store the keys and one to store the values. It keeps these arrays synchronized and sorted at all times.

Performance Trade-offs

This internal structure leads to a very specific performance profile, which is a key interview topic.

- **Additions and Removals are SLOW (O(n))**: To add a new item, the SortedList may have to shift a large number of elements in its internal arrays to make room in the correct sorted position.
- **Lookups are FAST (O(log n))**: Because the keys are always sorted, it can use a highly efficient **binary search** to find an item by its key.

Why and When to Use It

You should choose a SortedList over a Dictionary only in scenarios where:

1. You need your collection to **always remain sorted** by key.
2. Your usage pattern involves **many more lookups** than insertions or deletions.

If you just need fast lookups and don't care about sort order, Dictionary<TKey, TValue> is almost always the better choice due to its faster add/remove performance.

Example

```
SortedList<int, string> appointmentSchedule = new SortedList<int, string>();

// Items are added and automatically sorted by the key (the time).
appointmentSchedule.Add(1100, "Meet with Bob");
appointmentSchedule.Add(900, "Team Standup");
appointmentSchedule.Add(1400, "Deploy to Production");

Console.WriteLine("--- Today's Schedule (Sorted) ---");
foreach (var appointment in appointmentSchedule)
{
```

```
    Console.WriteLine($"{appointment.Key}: {appointment.Value}");
}
```
Output:

```
--- Today's Schedule (Sorted) ---
900: Team Standup
1100: Meet with Bob
1400: Deploy to Production
```

Lecture 13: Hashtable

Introduction

A **Hashtable** is the **non-generic, legacy version** of Dictionary<TKey, TValue>. It comes from the System.Collections namespace and was heavily used before generics were introduced in C# 2.0.

How It Works

It functions similarly to a Dictionary, using a hash table for fast lookups. The crucial difference is that it stores both its keys and values as the object type.

Disadvantages and Why Not to Use It

You should **avoid using Hashtable in all modern C# code**. The generic Dictionary<TKey, TValue> is superior in every way.

1. **Not Type-Safe:** You can add keys and values of any type to the same Hashtable, which can lead to logical errors. myHashtable.Add(1, "Alice"); // int key, string value myHashtable.Add("Bob", 2); // string key, int value

2. **Performance Issues (Boxing):** If you use value types (like int) as keys or values, they must be **boxed** into an object to be stored. This causes performance degradation.

3. **Requires Casting:** When you retrieve a value, it comes back as object. You must perform an **explicit cast** back to its original type, which is verbose and risks a run-time InvalidCastException.

Interview Perspective

- **Question:** "What's the difference between a Dictionary and a Hashtable?"
- **Ideal Answer:** "A Hashtable is the older, non-generic collection, while Dictionary is the modern, generic equivalent. The main difference is that Dictionary is type-safe—it enforces that all keys and values are of the specified generic types at compile time. Hashtable stores everything as object, which requires casting on retrieval and causes performance issues due to boxing with value types. For these reasons, Dictionary<TKey, TValue> should always be preferred in modern code."

Lecture 14: HashSet

Introduction

A **HashSet<T>** is a collection that contains **unique elements** in no particular order. It is designed for high-performance set operations and to enforce uniqueness among its items. Think of it as a bag where you can only have one of each specific item.

How It Works: A Hash Table for Uniqueness

Like a Dictionary, a HashSet<T> is based on a **hash table** internally. This allows it to perform additions, removals, and—most importantly—checks for containment at an extremely fast, average O(1) speed.

Its key feature is that it **automatically enforces uniqueness**. If you try to add an item that, according to its Equals() method, already exists in the set, the Add method simply returns false and the collection remains unchanged. No exception is thrown.

Why and When to Use It

1. **Maintaining Uniqueness:** When you need a collection that guarantees all its items are unique. For example, keeping track of all the unique visitors to a website.

2. **High-Speed Lookups:** When your primary need is to quickly check if an item exists in a collection, without needing to store a "value" like in a Dictionary. myHashSet.Contains(item) is much faster than myList.Contains(item) for large collections.
3. **Set Operations:** It provides efficient methods for mathematical set operations like UnionWith, IntersectWith, and ExceptWith.

Example

```
// Create a hash set to store unique tags for a blog post.
HashSet<string> tags = new HashSet<string>();

tags.Add("c#");
tags.Add("programming");
tags.Add("guide");

// Try to add a duplicate item.
bool wasAdded = tags.Add("c#"); // The Add method returns false.

Console.WriteLine($"Was 'c#' added again? {wasAdded}");
Console.WriteLine($"Total unique tags: {tags.Count}"); // The count is still 3.

Console.WriteLine("\nFinal Tags:");
foreach (string tag in tags)
{
    Console.WriteLine($"- {tag}");
}

// Check for existence - this is very fast.
if (tags.Contains("guide"))
{
    Console.WriteLine("\nThe post is filed under 'guide'.");
}
```

Lecture 15: ArrayList

Introduction

An **ArrayList** is the **non-generic, legacy version** of List<T>. It comes from the older System.Collections namespace and was the primary tool for creating dynamically-sized collections before generics were introduced in C# 2.0.

How It Works

It functions similarly to a List<T>, using a dynamic internal array that resizes itself as needed. The crucial difference is that it stores all of its elements as the base type **object**.

Disadvantages and Why Not to Use It

You should **avoid using ArrayList in all modern C# code**. The generic List<T> is superior in every way for these critical reasons:

Not Type-Safe: You can add any type of data to the same ArrayList, which can lead to logical errors and messy code.

```
ArrayList mixedList = new ArrayList();

mixedList.Add(10);          // An integer
mixedList.Add("Hello");     // A string
mixedList.Add(new Product()); // A custom object
```

Performance Issues (Boxing): If you add value types (like int, double, bool, or structs), they must be **boxed** into an object to be stored on the heap. This causes significant performance degradation.

Requires Casting: When you retrieve an item, it comes back as object. You must perform an **explicit cast** back to its original type. This is verbose and risks a run-time InvalidCastException if you cast to the wrong type.

```
int myNumber = (int)mixedList[0]; // Requires an explicit cast.
```

Interview Perspective

- **Question:** "What's the difference between an ArrayList and a List<T>?"
- **Ideal Answer:** "An ArrayList is the older, non-generic collection, while List<T> is the modern, generic equivalent. The main difference is that List<T> is type-safe—it enforces that all elements are of type T at compile time. ArrayList stores everything as object, which requires casting on retrieval and causes significant performance issues due to boxing with value types. For these reasons, List<T> should always be preferred in modern code."

Lecture 16: Stack

Introduction

A Stack<T> is a generic collection that operates on a **LIFO (Last-In, First-Out)** principle. You can visualize it as a stack of plates or a pile of books. You can only add a new item to the **top**, and you can only remove an item from the **top**. The last item you pushed onto the stack is the first item you can pop off.

Key Methods

- **Push(T item)**: Adds an item to the **top** of the stack.
- **Pop()**: **Removes and returns** the item at the top of the stack. If the stack is empty, this method will throw an InvalidOperationException.
- **Peek()**: **Returns** the item at the top of the stack **without removing it**. This is useful for inspecting the next item without consuming it. It also throws an exception if the stack is empty.
- **Count**: A property that gets the number of items currently in the stack.

When to Use It Use a stack for any problem that requires LIFO logic.

- **Undo Functionality:** In a text editor, each action (typing, deleting) is pushed onto an "undo" stack. When the user hits "Undo," you pop the last action off the stack and reverse it.

- **Expression Parsing:** Compilers use stacks to evaluate mathematical expressions.
- **Managing Function Calls:** The .NET call stack itself operates on this principle.

Example: Browser History "Back" Button

```
Stack<string> browserHistory = new Stack<string>();

// User navigates through pages.
browserHistory.Push("google.com");
browserHistory.Push("[google.com/search?q=csharp](https://google.com/search?q=csharp)");
browserHistory.Push("docs.microsoft.com");

Console.WriteLine($"Currently on page: {browserHistory.Peek()}");

// User clicks the "Back" button.
string previousPage = browserHistory.Pop();
Console.WriteLine($"\nClicked Back. Left page: {previousPage}");
Console.WriteLine($"Now on page: {browserHistory.Peek()}");
```

Lecture 17: Queue

Introduction

A **Queue<T>** is a generic collection that operates on a **FIFO (First-In, First-Out)** principle. You can visualize it as a checkout line at a store or a queue for a rollercoaster. The first item that goes into the queue is the first item to come out.

Key Methods

- **Enqueue(T item)**: Adds an item to the **end** (the "back") of the queue.
- **Dequeue()**: **Removes and returns** the item at the **beginning** (the "front") of the queue. If the queue is empty, this method will throw an InvalidOperationException.

- **Peek()**: **Returns** the item at the beginning of the queue **without removing it**. It also throws an exception if the queue is empty.
- **Count**: A property that gets the number of items currently in the queue.

When to Use It

Use a queue for any problem where items must be processed in the order they were received.

- **Task Scheduling:** Processing print jobs, network requests, or messages in the order they arrive.
- **Customer Service:** Handling support tickets in the order they were submitted.
- **Algorithms:** Implementing a Breadth-First Search (BFS) on a tree or graph.

Example: A Print Job Queue

```
Queue<string> printQueue = new Queue<string>();

// Users send documents to the printer.
printQueue.Enqueue("DocumentA.docx");
printQueue.Enqueue("Presentation.pptx");
printQueue.Enqueue("Spreadsheet.xlsx");

Console.WriteLine($"Jobs in queue: {printQueue.Count}");

// The printer processes jobs in FIFO order.
while (printQueue.Count > 0)
{
   string documentToPrint = printQueue.Dequeue();
   Console.WriteLine($"Printing: {documentToPrint}");
}

Console.WriteLine($"Jobs in queue: {printQueue.Count}");
```

Lecture 18: Collection of Objects

Introduction

All the generic collections we have discussed (List<T>, Dictionary<TKey, TValue>, etc.) can, of course, hold objects of your own custom classes. This is one of their most powerful features, allowing you to create dynamic, type-safe collections of complex data.

How It Works: Storing References, Not Objects

This is a crucial clarification that builds on our understanding of reference types. When you create a collection of a class type, such as List<Student>, the list's internal array does **not** store the full Student objects themselves. Instead, it stores **references** (memory addresses) to the Student objects, which live separately on the heap.

The Implication: If you retrieve an object from a list and modify it, you are modifying the **original object**. This is because both your local variable and the list's internal slot are pointing to the exact same object in memory.

Example

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Major { get; set; }
}

// --- In your Main method ---
List<Student> students = new List<Student>();

Student s1 = new Student { Id = 101, Name = "Alice", Major = "Physics" };
students.Add(s1);

// Get a reference to the student object from the list.
// 'studentToUpdate' now points to the same object as 's1'.
Student studentToUpdate = students[0];

// Modify the object using the new reference.
studentToUpdate.Major = "Astrophysics";
```

```
// The change is reflected in the original object because there is only one object.
Console.WriteLine($"Student s1's new major is: {s1.Major}"); // Output: Astrophysics
// The change is also visible when accessing it from the list again.
Console.WriteLine($"The major for student {students[0].Id} is now: {students[0].Major}");
```

Lecture 19: Object Relations

Introduction

A very common and important task in object-oriented design is modeling **relationships** between different types of objects. Collections are the primary tool for modeling **"one-to-many"** or **"has-a"** relationships. For example, a Team "has a" list of Players. An Author "has a" collection of Books.

How to Implement

You model this relationship by creating a class that contains a collection as one of its properties or fields. A Team class would have a List<Player> property.

Best Practice: It is a strong best practice to **initialize the collection property** either at the point of declaration or in the constructor. This prevents the property from being null and saves you from NullReferenceException errors. It is also good practice to expose the collection as a read-only property ({ get; }) and provide public methods (AddPlayer, RemovePlayer) to manage it. This gives the containing class full control over its internal list, which is better for encapsulation.

Example

```
public class Player
{
    public string Name { get; set; }
}

public class Team
{
```

```
    public string TeamName { get; set; }

    // The collection property. Initializing it here ensures it's never null.
    // The private setter means external code can't replace the whole list.
    public List<Player> Players { get; private set; } = new List<Player>();

    // Provide a public method to manage the internal list.
    public void AddPlayer(Player player)
    {
        // We could add validation here, e.g., max number of players.
        this.Players.Add(player);
        Console.WriteLine($"{player.Name} has been added to
{this.TeamName}.");
    }
}

// --- In your Main method ---
Team myTeam = new Team { TeamName = "Warriors" };
myTeam.AddPlayer(new Player { Name = "Alice" });
myTeam.AddPlayer(new Player { Name = "Bob" });

Console.WriteLine($"\nThere are {myTeam.Players.Count} players on the
{myTeam.TeamName}.");
```

Lecture 20: Built-in Collection Classes and Interfaces Hierarchy

Introduction

The .NET collections are built on a rich hierarchy of **interfaces** that define common capabilities. Understanding this hierarchy allows you to write more flexible and polymorphic code by programming against an interface rather than a concrete class.

The Key Interfaces

- **IEnumerable<T> (The base of everything)**
  - **Represents:** Anything that can be **iterated over**. It is the most fundamental collection interface.
  - **Key Member:** GetEnumerator(), which returns an IEnumerator<T>.
  - **Usage:** The C# foreach loop works on any object that implements IEnumerable<T>.
- **ICollection<T>**
  - **Inherits from:** IEnumerable<T>.
  - **Represents:** A generic collection that can be modified.
  - **Adds Members:** Count, Add(), Remove(), Contains(), Clear().
- **IList<T>**
  - **Inherits from:** ICollection<T>.
  - **Represents:** A collection whose elements can be accessed by a zero-based **index**.
  - **Adds Members:** An indexer this[int index], IndexOf(), Insert(), RemoveAt().
  - **Primary Implementation:** List<T>.
- **IDictionary<TKey, TValue>**
  - **Inherits from:** ICollection<KeyValuePair<TKey, TValue>>.
  - **Represents:** A collection of key-value pairs.
  - **Adds Members:** Keys, Values, and an indexer this[TKey key].
  - **Primary Implementation:** Dictionary<TKey, TValue>.

Why It Matters

By writing a method that accepts an IEnumerable<string>, you can pass it a List<string>, a string[] array, or a Queue<string>, making your method incredibly versatile.

Lecture 21: IEnumerable and IEnumerator

Introduction

This is one of the most important and foundational topics for understanding how collections work in .NET. The **IEnumerable<T>** interface is the most basic collection interface. It represents a sequence of items that can be **iterated over** (or "enumerated"). Its single purpose is to provide a standard way to get an "iterator" for the sequence. Any class that implements IEnumerable<T> is making a promise: "You can loop through me with a foreach loop."

The object that actually performs the iteration is called an **enumerator**, and it is defined by the **IEnumerator<T>** interface.

How It Works: The Iterator Pattern

The relationship between these two interfaces forms the **Iterator Design Pattern**:

1. The collection class (the **Enumerable**) implements IEnumerable<T>. Its only job is to have one method called GetEnumerator().
2. The GetEnumerator() method returns a new **Enumerator** object that implements IEnumerator<T>.
3. The Enumerator is a separate, stateful object that acts like a "cursor" or a "pointer." It knows where it is in the sequence and how to get to the next item.

The IEnumerator<T> Members An enumerator has three key members:

- **bool MoveNext()**: This is the engine of the iteration. It advances the cursor to the next element in the sequence. It returns true if it successfully moved to the next element, and false if it has passed the end of the collection.
- **T Current**: This is a read-only property that returns the element at the current position of the cursor.
- **void Dispose()**: Releases any resources held by the iterator.

The foreach Transformation (Critical Interview Topic)

When you write a simple foreach loop, the C# compiler transforms it into more complex code that uses the enumerator directly.

When you write this:

```
foreach (int number in myList)
{
   Console.WriteLine(number);
}
```

The compiler generates code similar to this:

```
IEnumerator<int> enumerator = myList.GetEnumerator();
try
{
   while (enumerator.MoveNext())
   {
      int number = enumerator.Current;
      Console.WriteLine(number);
   }
}
finally
{
   // The finally block ensures Dispose is called even if an error occurs.
   if (enumerator != null)
   {
      enumerator.Dispose();
   }
}
```

Why this design? Decoupling the collection (IEnumerable) from the iteration process (IEnumerator) allows multiple iterations to happen over the same collection simultaneously and independently. Two different foreach loops on the same list will each get their own separate enumerator object, so they won't interfere with each other.

Lecture 22: Iterator and Yield Return

Introduction

While you can manually create a class that implements IEnumerable<T> and IEnumerator<T>, it is complex and tedious. To simplify this, C# provides a powerful language feature called an **iterator**. An iterator is a special kind of method that uses the **yield return** keyword to implement an IEnumerable<T> sequence easily.

How It Works: Compiler Magic and State Machines

When the compiler sees a method with a yield return statement, it does something amazing. It automatically generates a hidden **state machine** class behind the scenes. This generated class implements both IEnumerable<T> and IEnumerator<T> for you.

- When you first call the iterator method, it doesn't run any code. It just returns the new enumerator object.
- The first time MoveNext() is called on the enumerator (e.g., by a foreach loop), your code runs until it hits the first yield return statement.
- It then "yields" that value back to the caller and **pauses** its execution, preserving its current state (like the values of its local variables).
- The next time MoveNext() is called, your method **resumes** execution right where it left off, and continues until it hits the next yield return.
- This continues until the method finishes, at which point MoveNext() will return false.

Deferred Execution A key benefit of this is **deferred execution**. The code in your iterator method does not run until you actually start iterating over it. This is extremely efficient, especially for large or infinite sequences, as you only generate the items as they are needed.

Example

Let's create a method that generates all even numbers up to a maximum, without having to create a big list in memory first.

```
// This method is an iterator. It returns an IEnumerable<int>.
public IEnumerable<int> GetEvenNumbers(int maxNumber)
{
    Console.WriteLine("--- Iterator method started ---");
    for (int i = 2; i <= maxNumber; i += 2)
    {
        Console.WriteLine($"Yielding {i}");
        yield return i; // Pause and return this value.
    }
    Console.WriteLine("--- Iterator method finished ---");
}

// --- In your Main method ---
IEnumerable<int> evenNumbers = GetEvenNumbers(10); // The method has NOT run yet!

Console.WriteLine("About to start foreach loop...");
// The code inside GetEvenNumbers will only execute as the loop asks for items.
foreach (int number in evenNumbers)
{
    Console.WriteLine($"  Received {number} in the loop.");
}
```

Output:

```
About to start foreach loop...
--- Iterator method started ---
Yielding 2
  Received 2 in the loop.
Yielding 4
  Received 4 in the loop.
Yielding 6
  Received 6 in the loop.
Yielding 8
  Received 8 in the loop.
Yielding 10
  Received 10 in the loop.
--- Iterator method finished ---
```

Lecture 23: Custom Collections

Introduction

By implementing the IEnumerable<T> interface, you can make your own custom classes compatible with the foreach loop and LINQ, allowing them to be treated like standard collections.

How It Works

The simplest way to implement IEnumerable<T> on a custom class that already uses an internal collection (like a List<T>) is to simply "pass through" the request. Your class's GetEnumerator() method can just return the result of calling GetEnumerator() on the internal list. You will also need to implement the non-generic IEnumerable.GetEnumerator() as well, which can just call the generic version.

Example: A Custom Playlist Class

```
public class Song
{
    public string Title { get; set; }
}

// This class implements IEnumerable<Song> to be foreach-compatible.
public class Playlist : IEnumerable<Song>
{
    private List<Song> _songs = new List<Song>();

    public void AddSong(Song song)
    {
        _songs.Add(song);
    }

    // Implementation for the generic interface
    public IEnumerator<Song> GetEnumerator()
    {
        return _songs.GetEnumerator();
    }

    // Explicit implementation for the non-generic legacy interface
```

```
    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
}

// --- In your Main method ---
Playlist myPlaylist = new Playlist();
myPlaylist.AddSong(new Song { Title = "Bohemian Rhapsody" });
myPlaylist.AddSong(new Song { Title = "Stairway to Heaven" });

// We can now use foreach directly on our custom Playlist object!
foreach (Song song in myPlaylist)
{
    Console.WriteLine($"Now playing: {song.Title}");
}
```

Lecture 24: Custom Collection with IList

Introduction

If you want your custom collection to have full list-like capabilities, including indexed access (myCollection[i]), you would implement the **IList<T>** interface.

How It Works

This is a much more advanced task because IList<T> inherits from ICollection<T> and IEnumerable<T>. This means you must provide an implementation for **all** of their members, including Count, Add, Remove, Clear, IndexOf, the indexer this[int], and more.

For most real-world scenarios, it is far simpler and more effective to **use a List<T> as a private backing field** and expose the functionality you need through your own public methods, rather than re-implementing the entire IList<T> interface from scratch.

Lecture 25: Custom Collection with IDictionary

Introduction

Similarly, if you want to create a custom key-value collection, you would implement the **IDictionary<TKey, TValue>** interface.

How It Works

Like IList<T>, this is a complex interface that requires you to implement all the functionality of a dictionary, including an indexer for keys, Add, Remove, ContainsKey, and properties like Keys and Values.

Again, this is rarely necessary. The built-in Dictionary<TKey, TValue> is highly optimized and sufficient for almost all use cases. It is generally better to use a Dictionary internally and expose its functionality in a controlled way.

Lecture 26: IEquatable

Introduction

The **IEquatable<T>** interface provides a type-safe method for determining **value equality**. While all objects inherit the Equals(object obj) method from System.Object, implementing the generic IEquatable<T> interface is a best practice that offers better performance and type safety.

How It Works

This interface defines a single method: bool Equals(T other).

The benefits of implementing this are:

1. **Type Safety:** The other parameter is strongly typed to your class or struct (T). This means you do not need to perform a type check or cast other from object, making your code cleaner and safer.

2. **Performance for Value Types:** When a value type (a struct) is passed to the object.Equals(object obj) method, it must be **boxed**, which hurts performance. The IEquatable<T>.Equals(T other) method takes the struct directly as a value type, completely avoiding the boxing penalty. Many collection methods, like List<T>.Contains(), are optimized to use this interface if it's available.

Best Practice

The standard best practice is to have your type implement **both** IEquatable<T> and override object.Equals. Your overridden object.Equals should simply call your type-safe IEquatable<T>.Equals implementation. And remember the golden rule: **If you implement equality, you must also override GetHashCode()**.

Example

```
public struct Point : IEquatable<Point>
{
    public int X { get; set; }
    public int Y { get; set; }

    // Implementation of the IEquatable<T> interface
    public bool Equals(Point other)
    {
        Console.WriteLine("Type-safe Equals(Point) called.");
        return this.X == other.X && this.Y == other.Y;
    }

    // Override of the standard object.Equals
    public override bool Equals(object obj)
    {
        if (obj is Point)
        {
            return this.Equals((Point)obj);
        }
        return false;
    }

    // Must override GetHashCode when overriding Equals
    public override int GetHashCode()
    {
        return HashCode.Combine(X, Y);
```

```
    }
}
```

Lecture 27: IComparable

Introduction

The **IComparable<T>** interface allows you to define a type's **natural sort order**. By implementing this interface, you are telling the .NET framework how instances of your type should be compared to one another to determine which one comes "first" in a sorted list.

How It Works

This interface defines a single method: `int CompareTo(T other)`.

The integer value that this method returns has a specific meaning:

- **Less than 0:** The current instance comes **before** the `other` instance in the sort order.
- **Zero:** The instances are considered **equal** in sort order.
- **Greater than 0:** The current instance comes **after** the `other` instance in the sort order.

When to Use It

You should implement `IComparable<T>` on your custom objects when you want built-in methods like `List<T>.Sort()` or `Array.Sort()` to work "out of the box" without needing to provide any extra sorting logic. It defines the default, most logical way to order your objects.

Example

Let's make our Product class sortable by its Price by default.

```
public class Product : IComparable<Product>
{
    public string Name { get; set; }
    public decimal Price { get; set; }

    // Implement the CompareTo method to define the natural sort order.
    public int CompareTo(Product other)
    {
        // If the other product is null, this instance is greater.
        if (other == null) return 1;

        // Compare based on the Price property.
        return this.Price.CompareTo(other.Price);
    }

    public override string ToString() => $"{Name} ({Price:C})";
}

// --- In your Main method ---
List<Product> products = new List<Product>
{
    new Product { Name = "Laptop", Price = 1200.00M },
    new Product { Name = "Mouse", Price = 25.50M },
    new Product { Name = "Keyboard", Price = 75.00M }
};

// Because Product implements IComparable, we can just call Sort().
products.Sort();

Console.WriteLine("Products sorted by price:");
products.ForEach(p => Console.WriteLine(p));
```

Lecture 28: IComparer

Introduction

The **IComparer<T>** interface is used to create an **external helper class** that provides a **custom or secondary sort order**. This decouples the sorting logic from the object itself, allowing you to sort the same collection of objects in many different ways.

How It Works

This interface defines a single method: int Compare(T x, T y).

The return value has the same meaning as in IComparable: return < 0 if x comes before y, 0 if they are equal, and > 0 if x comes after y.

When to Use It

1. When the class you want to sort **does not implement IComparable<T>**, and you cannot modify its source code (e.g., it's from a third-party library).
2. When you want to sort the same collection of objects in **multiple different ways**. For example, you might want to sort a list of Product objects sometimes by price, sometimes by name, and sometimes by stock level.

Example

Let's create a custom comparer to sort our Product objects by name, ignoring their natural sort order (which is by price).

```
// 1. Create a separate class that implements IComparer<T>.
public class ProductNameComparer : IComparer<Product>
{
   public int Compare(Product x, Product y)
   {
      // Use the string's built-in CompareTo method for alphabetical
sorting.
      return x.Name.CompareTo(y.Name);
   }
}
```

```
// --- In your Main method, using the same products list ---

// 2. Create an instance of our custom comparer.
ProductNameComparer nameComparer = new
ProductNameComparer();

// 3. Pass the comparer instance to the Sort() method.
products.Sort(nameComparer);

Console.WriteLine("\nProducts sorted by name:");
products.ForEach(p => Console.WriteLine(p));
```

Lecture 29: Covariance

Introduction

Covariance is an advanced generics concept that provides more flexibility in assignments. It allows you to use a **more derived** type than specified by the generic parameter. Covariance applies to "output" scenarios, like method return types.

The core idea is based on the "is-a" relationship: a `string` is an `object`, so a sequence of `strings` should be usable where a sequence of `objects` is expected.

How It Works: The `out` Generic Modifier

For an interface or delegate to be covariant, its generic type parameter `T` must be marked with the `out` keyword (`IEnumerable<out T>`). This signals to the compiler that the type `T` is only ever used in **output positions** (e.g., as a return value of a method) and never as an input parameter.

The `IEnumerable<T>` interface is the classic example of a covariant interface.

Example

```
// A list of strings is created.
IEnumerable<string> strings = new List<string> { "hello", "world" };

// This assignment is allowed because IEnumerable<T> is covariant.
// You can assign a sequence of a more derived type (string)
// to a variable of a sequence of a less derived type (object).
IEnumerable<object> objects = strings;

// You can now iterate over the collection of objects.
foreach (object obj in objects)
{
    Console.WriteLine(obj);
}

// IMPORTANT: This would NOT work with List<T>, because List<T> is not covariant.
// List<object> objectList = new List<string>(); // COMPILE ERROR!
// This is because List<T> has an Add(T item) method, which uses T as an input.
```

Lecture 30: Contravariance

Introduction

Contravariance is the logical opposite of covariance. It allows you to use a **less derived** (more general) type than specified by the generic parameter. Contravariance applies to **"input"** scenarios, like method parameters.

How It Works: The `in` Generic Modifier

For an interface or delegate to be contravariant, its generic type parameter T must be marked with the `in` keyword (`Action<in T>`). This signals that the type T is only ever used in **input positions** (as a parameter to a method).

The Analogy

"A method that can handle any Animal is also a method that can handle a Giraffe." If you need a method that takes a Giraffe, it is perfectly safe to provide a method that can take any Animal, because a Giraffe "is an" Animal.

The Action<T> delegate is the classic example of a contravariant delegate.

Example

```
// 1. Create an Action that can work on any object (the more general type).
Action<object> printObject = obj =>
Console.WriteLine(obj.GetHashCode());

// 2. We need an Action that works on strings (the more derived type).
// It is safe to assign the more general action to the more specific delegate variable.
Action<string> printString = printObject;

// 3. We can now call printString with a string.
// It will execute the logic from printObject.
printString("Hello, World!");
```

Lecture 31: IMP Points to Remember

- **Always Prefer Generic Collections:** List<T> and Dictionary<TKey, TValue> should always be used over their legacy, non-generic counterparts (ArrayList, Hashtable) for type safety and performance.
- **Choose the Right Collection for the Job:**
    - List<T>: For a general-purpose, dynamically-sized list of items.
    - Dictionary<TKey, TValue>: For extremely fast lookups based on a unique key.
    - HashSet<T>: To guarantee all items are unique or to perform set operations.

    - Stack<T>: For Last-In, First-Out (LIFO) logic.
    - Queue<T>: For First-In, First-Out (FIFO) logic.
- **IEnumerable<T> is the Foundation:** The foreach loop and all of LINQ operate on this fundamental interface. Understanding the iterator pattern (GetEnumerator, MoveNext, Current) is key to understanding how iteration works in .NET.
- **Use yield return for Custom Iterators:** yield return is the modern, simple way to implement IEnumerable<T>. It allows you to create sequences on-the-fly without allocating a full list in memory, which is known as **deferred execution**.
- **Comparison Interfaces Make Your Objects Smarter:**
    - Implement **IEquatable<T>** for type-safe value equality.
    - Implement **IComparable<T>** to define a type's natural sort order for List<T>.Sort().
    - Use **IComparer<T>** to provide multiple, custom ways to sort objects.
- **Variance Enables Greater Flexibility:** Remember the core ideas:
    - **Covariance (out)**: A list of strings can be treated as a list of objects.
    - **Contravariance (in)**: An action that works on an object can be used as an action that works on a string.

Section 23: Anonymous Types

Lecture 1: Anonymous Types

Introduction

An **anonymous type** is a feature in C# that allows you to create a simple object to hold a set of read-only properties without having to explicitly define a full class first. It's a convenient, on-the-fly way to create a temporary data structure. The compiler generates a class for you behind the scenes, but this class is "anonymous" because it has no name that you can use in your code.

How It Works: Compiler Generation

The var keyword is essential when working with anonymous types. Because you don't know the name of the class the compiler is about to create, you need var to hold a reference to the object.

When you write this:

```
var user = new { Name = "Alice", Age = 30 };
```

The C# compiler does this behind the scenes:

1. It invents a unique, "unspeakable" internal name for a new class, something like <>f__AnonymousType0.

It generates the actual class definition with read-only properties matching what you provided:

```
internal class <>f__AnonymousType0

{
   public string Name { get; }
   public int Age { get; }

   public <>f__AnonymousType0(string name, int age)
   {
      this.Name = name;
```

```
        this.Age = age;
    }

    // It also auto-generates Equals, GetHashCode, and ToString!
}
```

2. It then replaces your original line with:
   `<>f__AnonymousType0 user = new <>f__AnonymousType0("Alice", 30);`

Because you can't type that mangled name, `var` is the only practical way to declare the variable.

Key Characteristics

- **Read-Only:** All properties of an anonymous type are read-only. Once created, the object is immutable.
- **Type Inference:** The compiler infers the property names and types from the initialization syntax.
- **Reference Type:** The generated type is a class (a reference type) that inherits directly from `System.Object`.
- **Compiler Optimization:** If you create two anonymous types in the same assembly that have the same properties with the same types in the same order, the compiler is smart enough to reuse the same generated class definition.

Why and When to Use It

The primary use case is for holding temporary, intermediate data, especially in **LINQ queries**. When you use a `.Select()` clause to project data from a larger object into a smaller, temporary shape, an anonymous type is perfect.

Example

```
var product = new
{
   Name = "Laptop",
   Price = 1200.50M,
   IsAvailable = true
};

Console.WriteLine($"Product Name: {product.Name}");
Console.WriteLine($"Price: {product.Price:C}");

// The following line would cause a compiler error because the properties
are read-only.
// product.Name = "Desktop"; // ERROR!
```

Lecture 2: Nested Anonymous Types

Introduction

An anonymous type is not limited to holding only primitive types. One of its properties can be another anonymous type, allowing you to create simple, nested, hierarchical data structures on the fly.

How It Works

This works naturally because the compiler infers a property's type from the value it is assigned. If you assign a new anonymous type object as a property value, the compiler will generate the necessary nested class structure for you.

Example

Let's create an anonymous type for a user that includes a nested Address object.

```
var userProfile = new
{
   UserId = 101,
   Username = "jane.doe",
   ContactInfo = new // This is a nested anonymous type
```

```
    {
        Email = "jane.doe@example.com",
        PhoneNumber = "555-1234"
    },
    IsActive = true
};

// You can access the nested properties using standard dot notation.
Console.WriteLine($"Username: {userProfile.Username}");
Console.WriteLine($"Email: {userProfile.ContactInfo.Email}");

// The nested object itself can be accessed.
var contact = userProfile.ContactInfo;
Console.WriteLine($"Phone: {contact.PhoneNumber}");
```

Lecture 3: Anonymous Arrays

Introduction

An **anonymous array** is an array whose element type is implicitly inferred by the compiler from the elements you place inside the array initializer. It's a convenient shorthand that saves you from having to specify the type explicitly.

How It Works

You create an anonymous array using the `new[] { ... }` syntax, without putting a type name before the square brackets. The compiler will look at all the elements in the initializer and determine the best common type for them.

Rules

1. All elements in an anonymous array initializer must be of the same type, or be implicitly convertible to a single common type.
    - `new[] { 1, 5, 10 }` creates an `int[]`.
    - `new[] { 1, 2.5, 3 }` creates a `double[]` because `int` can be safely converted to `double`.
    - `new[] { 1, "hello" }` will cause a compiler error because there is no common type between `int` and `string`.

2. You can create an array of anonymous types. The compiler will check that all anonymous objects in the initializer have the same property names and types.

Example

```
// Compiler infers this is an int[] array.
var numbers = new[] { 2, 4, 6, 8 };

// Create an array of anonymous types.
var products = new[]
{
    new { Name = "Apple", Category = "Fruit" },
    new { Name = "Broccoli", Category = "Vegetable" },
    new { Name = "Orange", Category = "Fruit" }
};

// We can now iterate over this strongly-typed anonymous array.
foreach (var product in products)
{
    Console.WriteLine($"{product.Name} is in the {product.Category}
category.");
}
```

Lecture 4: Points to Remember

- **Temporary, Read-Only Data Containers:** This is the core purpose of anonymous types. They are designed for creating simple, immutable data structures for immediate, local use. They are not a replacement for defining proper class or struct types for your application's business model.
- **LINQ's Best Friend:** The most important real-world use case for anonymous types is in the Select clause of LINQ queries. They are perfect for projecting the results of a query into a new, flat shape without having to define a new class for every possible query result.

  ```
  // Example with LINQ
  var productNamesAndPrices = productList.Select(p => new { p.Name, p.Price });
  ```

- **Local Scope:** Because you cannot write down the name of an anonymous type, their use is effectively limited to the local scope of the method where they are created. You cannot, for example, have a method that returns an anonymous type in a strongly-typed way (you would have to return object or dynamic, which loses the benefit of type safety and IntelliSense).
- **var is Essential:** Reinforce that the var keyword is almost always used to hold a reference to an anonymous type object because you don't know the compiler-generated class name to write it down yourself. It is the key that unlocks the feature.

Section 24: Tuples

Lecture 1: Tuple class

Introduction

The System.Tuple class was the original way of creating tuples in the .NET Framework, before modern language improvements. A **tuple** is a data structure that holds a specific number and sequence of elements of potentially different types. It's a simple way to group a small, fixed set of related data points together without needing to create a custom class.

How It Works

Tuple is a **reference type** (a class). This means when you create a Tuple, memory is allocated on the **heap**, which can be less performant than stack allocation for small data structures.

The elements within a Tuple are accessed through read-only properties with generic, non-descriptive names: Item1, Item2, Item3, and so on.

Why It's a "Legacy" Type

While functional, the System.Tuple class has two significant drawbacks that have led to it being almost entirely replaced by the modern ValueTuple.

1. **Poor Performance:** Being a class, it requires heap allocation and can create pressure on the garbage collector.
2. **Poor Readability:** The property names Item1, Item2, etc., provide no semantic meaning. When you read product.Item1, you have no idea what that data represents without looking up the original declaration.

Example

```
// Creating an instance of the Tuple class.
Tuple<string, int, decimal> product = new Tuple<string, int,
decimal>("Laptop", 15, 1299.99M);

// Accessing elements is verbose and not very readable.
string productName = product.Item1;
int quantity = product.Item2;
decimal price = product.Item3;

Console.WriteLine($"Product: {productName}, Quantity: {quantity}, Price:
{price:C}");
```

Lecture 2: Value Tuples

Introduction

Value Tuples, introduced in C# 7.0, are the modern, lightweight, and preferred way to work with tuples. They were designed specifically to overcome the performance and readability limitations of the System.Tuple class.

How It Works

The most crucial difference is that a value tuple is a **value type** (a struct). This makes it far more efficient for temporary data grouping, as it is typically allocated on the **stack**, avoiding heap allocation and garbage collection overhead.

Syntax and Features

- **Literal Syntax:** You create a value tuple using a simple and clean literal syntax with parentheses.
- **Named Elements:** This is their biggest advantage. You can give meaningful names to the elements, which dramatically improves code readability. These names are available through IntelliSense.
- **Mutability:** Unlike anonymous types, the elements of a value tuple are mutable public fields.

Why and When to Use It

The primary and most powerful use case for value tuples is as a **return type from a method**. It provides a clean and efficient way to return multiple values without the ceremony of creating a custom class or using multiple out parameters.

Example

```
public (string, int) GetUser() // The method's return type is a value tuple.
{
    string name = "Alice";
    int age = 30;
    return (name, age);
}

// --- In your Main method ---

// 1. Creating a value tuple with semantic names.
(string name, int age) person = ("Bob", 42);

// 2. Accessing elements using the meaningful names.
Console.WriteLine($"Name: {person.name}, Age: {person.age}");

// You can still access them by their default names if you want.
Console.WriteLine($"Item1: {person.Item1}, Item2: {person.Item2}");

// 3. Receiving a tuple from a method.
var user = GetUser();
Console.WriteLine($"User from method: {user.Item1} is {user.Item2} years old.");
```

Lecture 3: Deconstructing

Introduction

Deconstruction is the process of "unpacking" the elements of a tuple (or another compatible object) into separate, distinct variables. It provides a convenient syntax to get the individual components of a grouped data structure into their own clearly named local variables.

How It Works: Syntax Variations

1. **Deconstructing into Newly Declared Variables:** This is the most common form. You declare the types and names of the variables you want to create on the left side of the assignment.

   ```
   (string name, int age) = GetUser();
   Console.WriteLine($"Deconstructed name: {name}");
   ```

2. **Deconstructing with var:** You can use the `var` keyword for a more concise syntax. The compiler will infer the correct types for the new variables.

   ```
   var (name, age) = GetUser();
   ```

3. **Deconstructing into Existing Variables:** You can also deconstruct into variables that have already been declared.

   ```
   string existingName;
   int existingAge;
   (existingName, existingAge) = GetUser();
   ```

Deconstructing Other Types

This feature is not limited to tuples. Any type can be made deconstructible by implementing a special Deconstruct method. For example, KeyValuePair<TKey, TValue> from a dictionary supports deconstruction.

Example

```
var userScores = new Dictionary<string, int> { { "Alice", 95 } };

// A foreach loop over a dictionary yields KeyValuePair objects.
// We can deconstruct the KeyValuePair directly in the loop!
foreach ((string name, int score) in userScores)
{
    Console.WriteLine($"{name} scored {score}.");
}
```

Lecture 4: Discards

Introduction

A **discard**, represented by the underscore character _, is a special placeholder variable that you use when you are required to provide a variable but you intentionally want to **ignore its value**.

How It Works

The underscore is not a real variable with a memory location. It's a signal to the compiler that "a value will be produced here, but I don't care about it and will never use it." The compiler can often use this information to generate more efficient code by avoiding the allocation of memory for the discarded value.

Why and When to Use It

The most common use case is with **deconstruction** when you only care about some of the values being returned. It allows you to selectively unpack a tuple. It's also frequently used with out parameters when you call a method for its return value but don't need the out argument.

Example

```
// This method returns three values.
public (string, string, int) GetUserDetails()
{
    return ("John", "Doe", 42);
}

// --- In your Main method ---

// Scenario 1: We only care about the first name and age.
// We use a discard '_' to ignore the last name.
(string firstName, _, int age) = GetUserDetails();
Console.WriteLine($"{firstName} is {age} years old.");

// Scenario 2: We only care if a string is a valid number, not the number itself.
// We can discard the 'out' parameter.
if (int.TryParse("123", out _))
{
    Console.WriteLine("The string is a valid number.");
}
```

Lecture 5: IMP Points to Remember

- **ValueTuple is the Modern Standard:** Always prefer the modern value tuple syntax, e.g. (int, string), over the legacy System.Tuple class. ValueTuple provides vastly better performance due to stack allocation and superior readability due to named elements.
- **Primary Use Case: Returning Multiple Values:** The killer feature of value tuples is providing a clean, simple, and efficient way to return multiple values from a single method call. It is almost always a better choice than using multiple out parameters.
- **Tuples are for Temporary Data Grouping:** Tuples are excellent for intermediate, local data that you don't want to create a full class or struct for. However, if a data structure is complex, long-lived, or needs to be passed through multiple layers of your application, that is a strong sign that you should define a proper, named class or struct instead.
- **Deconstruction and Discards Improve Readability:** Use deconstruction var (name, age) = ... to immediately unpack tuple results into clearly named local variables. Use the discard _ to explicitly signal that you are intentionally ignoring a part of a result, which makes your code's intent clearer to other developers.
- **Mutability:** Remember that the elements of a value tuple are mutable public fields by default (myTuple.Name = "New Name";), which is a key difference from anonymous types where properties are read-only.

Section 25: LINQ

Lecture 1: Intro and LINQ Basics

Introduction

LINQ, which stands for **Language-Integrated Query**, is a powerful and unified set of features in C# that provides a consistent, SQL-like syntax for querying data. Before LINQ, if you wanted to get data from a database, an XML file, or an in-memory collection, you had to learn a completely different API for each one. LINQ provides a single, elegant way to work with all of them.

The two key benefits of LINQ are:

1. **Declarative Syntax:** You describe **what** you want, not **how** to get it. You write a query that says "give me all products where the price is over 100," and LINQ figures out the underlying loops and conditions for you.
2. **Strong Typing:** LINQ queries are part of the C# language itself. This means your queries are checked by the compiler for type errors, and you get full IntelliSense support, which prevents many common bugs.

The Two Syntaxes

You can write LINQ queries in two ways. Both are compiled into the exact same code, so the choice is a matter of readability and style.

1. **Query Syntax (SQL-like):** This syntax looks very similar to SQL. It's often preferred for more complex queries involving joins or multiple conditions because it can be easier to read.

   ```
   var results = from student in students
   where student.Score > 90
   select student.Name;
   ```

2. **Method Syntax (Fluent API):** This syntax uses a chain of extension methods called on the collection. This is the **more common and powerful** syntax, as every LINQ query is ultimately translated into this form by the compiler. All LINQ operators are available in this syntax.

```
var results = students.Where(student => student.Score > 90)
.Select(student => student.Name);
```

How It Works: Extension Methods & Deferred Execution

- **Extension Methods:** LINQ is built almost entirely on **extension methods** for the IEnumerable<T> interface. This is why you can call methods like .Where() and .Select() on any array or list. To access them, you must add the using System.Linq; directive to your file.
- **Deferred Execution:** This is a critical concept. When you define a LINQ query, the query **does not run immediately**. It is only a "plan" for how to get the data. The query is only executed when you start iterating over its results, for example, by using a foreach loop or by calling a method like .ToList() or .Count(). This allows LINQ to be extremely efficient by combining multiple operations into a single pass over the data source.

Lecture 2: LINQ to Arrays and Collections

Introduction

The most common and straightforward use of LINQ is for querying in-memory collections like List<T> and arrays. This allows you to filter, sort, and transform your data with incredibly concise and readable code, replacing complex and error-prone for and if loops.

How It Works

To use LINQ on a collection, you first need the using System.Linq; directive at the top of your file. This brings all the LINQ extension methods into scope. You then simply call the desired methods on your collection variable.

The result of most LINQ query operators (like Where and OrderBy) is another IEnumerable<T>. This allows you to chain operators together to build up a more complex query. Remember that due to **deferred execution**, this chain of operations is only a plan. The work isn't done until you materialize the results.

Example

Let's find all numbers greater than 50 in a list and display them.

```
// The data source
List<int> scores = new List<int> { 25, 88, 42, 95, 15, 71, 99 };

// --- Method Syntax (Preferred) ---
// 1. Define the query. No work is done here yet.
IEnumerable<int> highScoresQuery = scores.Where(score => score > 50);

// --- Query Syntax (Alternative) ---
IEnumerable<int> highScoresQuerySql = from score in scores
                                      where score > 50
                                      select score;

Console.WriteLine("Query has been defined. Now executing with foreach
loop...");

// 2. Execute the query by iterating over it.
// The 'Where' filter is applied as the loop pulls each item.
foreach (int score in highScoresQuery)
{
    Console.WriteLine(score);
}
```

Output:

```
Query has been defined. Now executing with foreach loop...
88
95
71
99
```

Lecture 3: OrderBy

Introduction

OrderBy and its counterpart OrderByDescending are the primary LINQ methods for sorting sequences. They allow you to order a collection based on a specific property or value of its elements.

How It Works

- **OrderBy(keySelector)**: Sorts the elements in **ascending** order.
- **OrderByDescending(keySelector)**: Sorts the elements in **descending** order.

The keySelector is a lambda expression that tells the method what to sort by. For example, p => p.Price tells it to sort by the Price property.

For secondary sorting, you can chain ThenBy() or ThenByDescending(). This allows you to sort by one criterion, and then for all items that are equal in that first sort, sort them by a second criterion.

Example

```
public class Product { public string Category; public string Name; }

List<Product> products = new List<Product>
{
   new Product { Category = "Fruit", Name = "Apple" },
   new Product { Category = "Vegetable", Name = "Carrot" },
   new Product { Category = "Fruit", Name = "Banana" }
};

// Sort by a single criterion: Name
var sortedByName = products.OrderBy(p => p.Name);
// Result: Apple, Banana, Carrot

// Sort by Category (primary), then by Name (secondary)
var sortedByCategoryThenName = products.OrderBy(p => p.Category)
                                       .ThenBy(p => p.Name);
// Result: Apple (Fruit), Banana (Fruit), Carrot (Vegetable)
// Note: Apple comes before Banana within the Fruit category.
```

```
Console.WriteLine("--- Sorted by Category, then Name ---");
foreach (var product in sortedByCategoryThenName)
{
    Console.WriteLine($"{product.Category.PadRight(10)} |
{product.Name}");
}
```

Lecture 4: First and FirstOrDefault

Introduction

These are **element operators** used to retrieve the **very first element** from a sequence. Choosing between them depends on how you want to handle the case where the sequence is empty or no element matches your criteria.

How It Works: The Critical Difference

- **First()**
    - **Behavior:** Returns the first element of a sequence.
    - **Exception Handling:** If the sequence is **empty**, it throws an **InvalidOperationException**.
    - **With a Predicate:** First(predicate) returns the first element that matches the condition. If **no element matches**, it also throws an InvalidOperationException.
    - **When to Use:** Use First() when your code **expects and requires** that an element exists. In this case, an empty sequence or no match is an exceptional error condition that should stop the program.

- **FirstOrDefault()**
    - **Behavior:** Returns the first element of a sequence.
    - **Exception Handling:** If the sequence is **empty**, it **gracefully returns the default value** for the type (null for reference types, 0 for int, false for bool, etc.). It does **not** throw an exception.
    - **With a Predicate:** FirstOrDefault(predicate) returns the first element that matches. If **no element matches**, it also returns the default value.
    - **When to Use:** Use FirstOrDefault() when it is a **normal, valid scenario** for an element to be missing. This is the safer and more common choice.

Example

```
List<int> numbers = new List<int> { 10, 20, 30 };
List<int> emptyList = new List<int>();

// --- Successful Cases ---
int firstNum = numbers.First(); // 10
int firstOrDefaultNum = numbers.FirstOrDefault(); // 10

// --- Unsuccessful Cases ---
int? firstOrDefaultEmpty = emptyList.FirstOrDefault(); // Returns
default(int), which is 0. No exception.

try
{
   int firstEmpty = emptyList.First(); // Throws InvalidOperationException!
}
catch (InvalidOperationException)
{
   Console.WriteLine("First() threw an exception on an empty list, as
expected.");
}
```

Lecture 5: Last and LastOrDefault

Introduction

These methods are the direct counterparts to First and FirstOrDefault. They are used to retrieve the **very last element** from a sequence.

How It Works: Same Rules, Opposite End

The exception handling rules are identical to First and FirstOrDefault:

- **Last()**: Throws an InvalidOperationException if the sequence is empty or if no element matches the predicate.
- **LastOrDefault()**: Gracefully returns the default value for the type if the sequence is empty or no element matches.

Performance Note

For simple collections like List<T> that know their count and can be indexed from the end, these operations are fast. However, for a generic IEnumerable<T> that can only be iterated forward, calling Last() might require iterating through the entire sequence to find the end, which can be a slow (O(n)) operation.

Example

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };

string lastName = names.Last(); // "Charlie"
string lastOrDefaultName = names.LastOrDefault(); // "Charlie"

// Find the last name that starts with "A"
string lastA = names.LastOrDefault(name => name.StartsWith("A")); // "Alice"

// Find the last name that starts with "Z"
string lastZ = names.LastOrDefault(name => name.StartsWith("Z")); // null (default for string)
Console.WriteLine($"Last name starting with Z: {lastZ ?? "Not Found"}");
```

Lecture 6: ElementAt and ElementAtOrDefault

Introduction

These are LINQ element operators used to retrieve an element at a **specific zero-based index** within a sequence. While this seems similar to using the array/list indexer (e.g., myList[5]), these methods have the advantage of working on **any** sequence that implements IEnumerable<T>, even those that don't have a built-in indexer.

How It Works & When to Use It

The key difference is performance and applicability. The direct indexer (myList[index]) is an O(1) operation for lists and arrays because it can calculate the memory location directly. The ElementAt(index) method, when used on a simple forward-only IEnumerable<T>, may have to iterate through all the elements from the beginning up to the desired index, making it a potentially slow O(n) operation.

You should use ElementAt when you are working with a generic IEnumerable<T> and need to get an element by its position.

The Exception Handling Pattern

These methods follow the same safety pattern as First and Last.

- **ElementAt(index)**: Returns the element at the specified index. If the index is negative or greater than or equal to the size of the collection, it throws an **ArgumentOutOfRangeException**.
- **ElementAtOrDefault(index)**: Returns the element at the specified index. If the index is out of bounds, it **gracefully returns the default value** for the type (null, 0, etc.) and does not throw an exception.

Example

```
string[] colors = { "Red", "Green", "Blue" };

// --- Successful Case ---
string secondColor = colors.ElementAt(1);
Console.WriteLine($"The element at index 1 is: {secondColor}"); // Output:
Green

// --- Unsuccessful Case ---
// The valid indices are 0, 1, and 2. Index 5 is out of bounds.
string nonExistentColor = colors.ElementAtOrDefault(5);

// 'nonExistentColor' will be null (the default for string), no exception is
thrown.
Console.WriteLine($"The element at index 5 is: {nonExistentColor ?? "Not
Found"}"); // Output: Not Found

try
{
   // This call will throw an exception.
   string errorColor = colors.ElementAt(5);
}
catch (ArgumentOutOfRangeException)
{
   Console.WriteLine("ElementAt(5) threw an exception, as expected.");
}
```

Lecture 7: Single and SingleOrDefault

Introduction

These are the **strictest** element operators in LINQ. They are used to retrieve the **one and only** element from a sequence that satisfies a condition. Their purpose is to enforce an assumption of uniqueness. If that assumption is violated (by finding zero elements, or more than one), they will throw an exception.

How It Works: The Strict Rules

This is a very common interview topic used to distinguish between First and Single.

- **Single()**
  - Returns the element **only if exactly one** element exists in the sequence (or satisfies the predicate).
  - Throws InvalidOperationException if the sequence is **empty**.
  - Throws InvalidOperationException if **more than one** element is found.
- **SingleOrDefault()**
  - Returns the element **only if exactly one** element exists.
  - Returns the **default value** (null, 0, etc.) if the sequence is **empty**.
  - Throws InvalidOperationException if **more than one** element is found.

Why and When to Use It

Use Single or SingleOrDefault when your business logic dictates that there **must be exactly one (or zero-or-one) result**. A classic example is looking up a record by its unique primary key in a database. You expect to find exactly one record. Finding zero is a problem, and finding more than one indicates data corruption. First would be wrong in this case, as it would happily return the first record and hide the fact that there were duplicates.

Example

```
List<Product> products = new List<Product>
{
   new Product { Id = 10, Category = "Electronics" },
   new Product { Id = 20, Category = "Books" },
   new Product { Id = 30, Category = "Electronics" }
};

// --- Successful Case ---
// We expect exactly one product with ID 20.
Product book = products.Single(p => p.Id == 20);
Console.WriteLine($"Found single product: {book.Category}");
```

```
// --- "More than one" Exception ---
try
{
   // This will throw an exception because more than one element
matches.
   Product elec = products.Single(p => p.Category == "Electronics");
}
catch (InvalidOperationException)
{
   Console.WriteLine("Single() failed for 'Electronics' because more than
one was found.");
}

// --- "Not Found" Case with SingleOrDefault ---
// We expect zero or one product with ID 99.
Product notFound = products.SingleOrDefault(p => p.Id == 99);
Console.WriteLine($"Found product with ID 99? {notFound == null}"); //
True, returns null gracefully.
```

Lecture 8: Select

Introduction

Select is the primary **projection** operator in LINQ. Its job is to take a sequence of elements and **transform each element into a new form**. It projects the data from one shape into another. This is one of the most powerful and frequently used LINQ methods.

How It Works

The Select method takes a lambda expression (Func<TSource, TResult>) that is applied to every element in the source collection. The value returned by your lambda expression for each element becomes an element in the new output sequence. The output sequence (IEnumerable<TResult>) can be of a completely different type than the input sequence (IEnumerable<TSource>).

Why and When to Use It

1. **To select a single property** from a collection of complex objects. For example, getting a list of just the names from a list of Product objects.
2. **To transform each element into a new object.** This is often used with anonymous types to create a new sequence containing only a subset of the original data.

Example

```
public class Product { public string Name; public decimal Price; }

List<Product> products = new List<Product>
{
    new Product { Name = "Laptop", Price = 1200 },
    new Product { Name = "Mouse", Price = 25 }
};

// --- Use Case 1: Select a single property (string) ---
IEnumerable<string> productNames = products.Select(p => p.Name);
// The result is an IEnumerable<string>
Console.WriteLine("Product Names: " + string.Join(", ", productNames));

// --- Use Case 2: Project into a new anonymous type ---
var productsWithTax = products.Select(p => new
{
    ProductName = p.Name,
    PriceWithTax = p.Price * 1.08M // Add 8% tax
});
// The result is an IEnumerable of the new anonymous type.

Console.WriteLine("\nProducts with Tax:");
foreach (var p in productsWithTax)
{
    Console.WriteLine($"- {p.ProductName} costs {p.PriceWithTax:C}");
}
```

Lecture 9: Min, Max, Count, Sum, Average

Introduction

These are the primary **aggregate operators** in LINQ. They take an entire sequence of values and compute a single, summary result from them.

How They Work: Immediate Execution

This is a key difference from most other LINQ operators. Aggregate operators **do not use deferred execution**. They must iterate through the entire sequence immediately to compute their value. They return a single value (like an int or double), not an IEnumerable<T>.

The Methods

- **Count()**: Returns the number of elements in a sequence as an int. It can also take a predicate to count only the elements that match a condition (e.g., numbers.Count(n => n > 50)).
- **Min() and Max()**: Returns the minimum or maximum value in a sequence. For sequences of objects, they can take a selector to find the min/max of a specific property (e.g., products.Max(p => p.Price)).
- **Sum()**: Calculates the sum of a sequence of numeric values.
- **Average()**: Calculates the average of a sequence of numeric values.

Important Note: Calling Min, Max, or Average on an empty sequence will throw an InvalidOperationException. Count and Sum will gracefully return 0.

Example

```
int[] scores = { 88, 95, 72, 100, 65 };

// Count
int totalScores = scores.Count(); // 5
int scoresOver90 = scores.Count(s => s >= 90); // 2

// Min / Max
int lowestScore = scores.Min(); // 65
```

```
int highestScore = scores.Max(); // 100

// Sum / Average
int totalPoints = scores.Sum(); // 420
double averageScore = scores.Average(); // 84.0

Console.WriteLine($"Total Scores: {totalScores}");
Console.WriteLine($"Number of A-grade scores: {scoresOver90}");
Console.WriteLine($"Highest score: {highestScore}");
Console.WriteLine($"Average score: {averageScore}");
```

Lecture 10: Things to Remember

- **LINQ is Declarative:** LINQ allows you to write queries that describe **what** you want, not the step-by-step process of **how** to get it. This makes code more readable and less error-prone.
- **Deferred Execution is Key:** Remember that most LINQ operators that return a sequence (like Where, Select, OrderBy) use deferred execution. The query is only executed when you iterate over it (e.g., with foreach) or materialize it (e.g., with .ToList() or .ToArray()). Aggregate operators (Count, Sum, etc.) execute immediately.
- **First() vs. Single(): A Critical Distinction:** This is a classic interview question used to test your precision.
    - **First()**: Finds the first element that matches and stops. It's okay if there are more matching elements later in the sequence. Use it when you just need "any" match.
    - **Single()**: Finds an element that matches and then continues to the end of the sequence to **ensure there are no others**. It is used to enforce a uniqueness constraint. Finding more than one is an error.
- **...OrDefault() for Safety:** Use FirstOrDefault(), SingleOrDefault(), LastOrDefault(), etc., whenever it is a normal, valid business scenario for no element to be found. This avoids exceptions and leads to cleaner conditional logic.
- **Select is for Projection:** Select is your primary tool for transforming the shape of your data—whether it's picking a single property or creating a new anonymous type with a subset of data. It is arguably the most powerful and frequently used LINQ operator.

Section 26: String, DateTime, Math

Lecture 1: Intro to Strings

Introduction

The **string** data type is one of the most fundamental and frequently used types in any programming language. In C#, a `string` represents a sequence of characters. While it often feels like a simple, primitive type, it is actually a complex and highly optimized **reference type** with some very special behaviors. Understanding these nuances is critical for writing efficient and correct code.

The Two Most Important Characteristics of a String

1. **It is a Reference Type:** When you create a string variable, the variable itself holds a **reference** to the actual sequence of characters, which is stored on the heap. However...

2. **It Exhibits Value-Type Behavior (It is Immutable):** This is the most crucial concept to grasp about strings. **Strings in C# are immutable.** This means that once a string object has been created in memory, its content can **never be changed**. Any operation that appears to modify a string—like converting it to uppercase or replacing a character—does not change the original string. Instead, it creates a **brand new string object** in memory with the modified content and returns a reference to that new object.

Example: Demonstrating Immutability

```
string originalMessage = "Hello";
Console.WriteLine($"Original message: '{originalMessage}'");

// The ToUpper() method does NOT change the original string.
// It creates a NEW string in memory and returns a reference to it.
string upperMessage = originalMessage.ToUpper();

Console.WriteLine($"'originalMessage' is still: '{originalMessage}'"); // Output: Hello
Console.WriteLine($"'upperMessage' is a new string: '{upperMessage}'"); // Output: HELLO
```

This immutable nature ensures that strings are predictable and safe to pass between different parts of an application without fear that one part will unexpectedly change the string for another.

Lecture 2: How String Objects are Created

Introduction

There are several ways to create a string object, but the most common method involves a special optimization by the .NET runtime called the **intern pool**.

How It Works: String Literals and the Intern Pool

When you declare a string using a **string literal** (text enclosed in double quotes), the CLR performs a special check.

```
string s1 = "Hello World";
string s2 = "Hello World";
```

1. When the line for s1 is encountered, the CLR checks a special internal table called the **intern pool** to see if the string literal "Hello World" already exists.

2. Since it doesn't, the CLR creates the "Hello World" string object on the heap and places a reference to it in the intern pool. The variable s1 gets a reference to this object.

3. When the line for s2 is encountered, the CLR checks the intern pool again. This time, it finds "Hello World".

4. Instead of creating a new object, the CLR gives s2 a reference to the **exact same object** that s1 points to.

This process is a memory optimization that ensures identical string literals only occupy one location in memory for the entire application.

Creating Strings with new

You can also create a string using the new keyword, just like any other class. When you do this, you are explicitly telling the runtime to **bypass the intern pool** and create a new object on the heap, even if an identical string already exists. This is rarely necessary.

```
string s3 = new string("Hello World"); // This creates a new object.
```

Interview Perspective

- **Question:** "What is the difference between string s1 = "text"; and string s2 = new string("text");?"
- **Ideal Answer:** "Declaring a string with a literal ("text") utilizes the .NET intern pool. If an identical string already exists in the pool, the new variable will get a reference to the existing object. Using new string("text") explicitly bypasses the intern pool and always creates a new string object on the heap, even if an identical one already exists."

Lecture 3: The System.String class

Introduction

The string keyword in C# is not a true primitive type; it is a convenient **alias** for the **System.String** class. This is a fundamental concept. Because string is an alias for a class, every string variable you create is a true object, which gives it access to a rich set of powerful instance methods for manipulation and inspection.

How It Works

When you write string name = "John";, the compiler treats it exactly as if you had written System.String name = "John";. This means you can call methods directly on the variable or even on a string literal.

```
string greeting = "Hello, World!";

// Calling an instance method on the string variable.
int length = greeting.Length; // .Length is actually a property

// You can even call methods on a literal.
bool startsWithHello = "Hello, World!".StartsWith("Hello");
```

This design provides the best of both worlds: the simple, intuitive feel of a primitive type with the power and functionality of a full-featured class. In the following lectures, we will explore the most important methods of this class.

Lecture 4: ToUpper() and ToLower()

Introduction

These are two simple but essential methods for case conversion.

Methods

- **string ToUpper()**: Returns a **new string** in which all the characters of the original string are converted to uppercase.
- **string ToLower()**: Returns a **new string** in which all the characters of the original string are converted to lowercase.

How It Works & Immutability

It is crucial to remember that strings are immutable. These methods **do not** change the original string. They create a new string in memory with the converted case and return a reference to it.

When to Use It

Case conversion is vital for performing case-insensitive comparisons. Before comparing two strings, a common practice is to convert both to a consistent case (usually upper or lower) to ensure that "Apple" and "apple" are treated as equal.

Example

```
string userInput = "CSharp";
string dbValue = "csharp";

Console.WriteLine($"Original userInput: {userInput}");

// To perform a case-insensitive comparison
if (userInput.ToUpper() == dbValue.ToUpper())
{
   Console.WriteLine("The strings are equal when ignoring case.");
}

// Remember to capture the result of the conversion
string lowerCaseVersion = userInput.ToLower();
Console.WriteLine($"Lowercase version: {lowerCaseVersion}");
Console.WriteLine($"Original userInput is still unchanged: {userInput}");
```

Lecture 5: Substring()

Introduction

The Substring() method allows you to extract a portion of a string, creating a new string from a part of the original.

Method Overloads

1. **string Substring(int startIndex)**: Returns a new string that is a substring of the original. The substring starts at the specified startIndex and continues to the **end** of the string.

2. **string Substring(int startIndex, int length)**: Returns a new string that is a substring of the original. The substring starts at startIndex and has a specified length.

How It Works

Like all string manipulation methods, Substring is immutable. It allocates new memory for the new, shorter string and returns a reference to it. The original string remains untouched.

Example

```
string data = "ID:12345-USERNAME:JohnDoe";

// Find the starting position of the username.
int userStartIndex = data.IndexOf("USERNAME:") + "USERNAME:".Length;

// Extract the username from that position to the end of the string.
string username = data.Substring(userStartIndex);
Console.WriteLine($"Extracted Username: {username}"); // Output:
JohnDoe

// Extract just the ID number (which has a fixed length of 5).
int idStartIndex = data.IndexOf(":") + 1;
string id = data.Substring(idStartIndex, 5);
Console.WriteLine($"Extracted ID: {id}"); // Output: 12345
```

Lecture 6: Replace()

Introduction

The Replace() method is used to find all occurrences of a specified character or string within the current string and replace them with another character or string.

Method Overloads

- **string Replace(char oldChar, char newChar)**: Replaces all occurrences of a character.
- **string Replace(string oldValue, string newValue)**: Replaces all occurrences of a string.

How It Works

This method is case-sensitive. It returns a **new string** with the replacements made. If the oldValue is not found in the string, it returns a reference to the original, unchanged string.

Example

```
string reportText = "This report is incomplete and contains incomplete data.";

// Replace all occurrences of the word "incomplete".
string correctedText = reportText.Replace("incomplete", "preliminary");

Console.WriteLine($"Original: {reportText}");
Console.WriteLine($"Corrected: {correctedText}");

string filePath = "C:\\Users\\MyUser\\Documents";
// A common use is to normalize directory separators.
string webPath = filePath.Replace('\\', '/');
Console.WriteLine($"Web-friendly path: {webPath}");
```

Lecture 7: Format()

Introduction

String.Format() is a powerful static method that allows you to create a new string by inserting formatted values into placeholders within a template string. This technique is known as **composite formatting**.

How It Works

The method takes a format string containing numbered placeholders (e.g., {0}, {1}) and a list of arguments. It replaces each placeholder with the string representation of the corresponding argument from the list.

You can also include **format specifiers** to control how the value is displayed (e.g., formatting a number as currency or a date in a specific style).

String Interpolation ($) - The Modern Alternative

While String.Format is powerful, **string interpolation**, introduced in C# 6.0, is the modern, more readable, and preferred way to format strings. You prefix the string with a $ and place the variable names directly inside curly braces {}. The compiler translates this into String.Format calls behind the scenes.

Example

```
string name = "Alice";
int score = 95;
decimal price = 24.99m;

// --- Using String.Format (The Old Way) ---
string formattedString = String.Format("User {0} scored {1} points. The item costs {2:C}.", name, score, price);
// {2:C} formats the third argument as Currency.

// --- Using String Interpolation (The Modern, Preferred Way) ---
string interpolatedString = $"User {name} scored {score} points. The item costs {price:C}.";

Console.WriteLine(interpolatedString);
// Output: User Alice scored 95 points. The item costs $24.99.
```

Lecture 8: IsNullOrEmpty

Introduction

String.IsNullOrEmpty is a static utility method that provides a convenient and safe way to check if a string is either **null** or an **empty string ("")**. It combines two common checks into a single, highly readable method call.

How It Works

This method is a simple shortcut for the common condition: if (myString == null || myString == "").

By using IsNullOrEmpty, you first guard against a NullReferenceException (by checking for null) and then check if the string has any characters.

When to Use It

This should be your standard way to check for empty or non-existent string data, especially when validating method parameters or user input. It prevents NullReferenceException errors and makes your code's intent clearer than writing out the full || condition.

Example

```
public void ProcessUsername(string username)
{
   // Use IsNullOrEmpty as a "guard clause" at the beginning of the
method.
   if (String.IsNullOrEmpty(username))
   {
      Console.WriteLine("Username cannot be empty. Process halted.");
      return;
   }

   Console.WriteLine($"Processing user: {username}");
   // ... continue with processing ...
}
```

```
// --- In your Main method ---
ProcessUsername("JohnDoe"); // Output: Processing user: JohnDoe
ProcessUsername("");        // Output: Username cannot be empty. Process halted.
ProcessUsername(null);      // Output: Username cannot be empty. Process halted.
```

Lecture 9: Split

Introduction

The Split() method is an instance method that breaks a string into an array of substrings based on a specified set of delimiters. It is an essential tool for parsing structured string data.

How It Works

The most common overload takes an array of characters (char[]) to use as delimiters. A very useful second parameter is StringSplitOptions.RemoveEmptyEntries, which tells the method to discard any empty strings that result from the split. This is helpful when your data might have multiple delimiters next to each other (e.g., "apple,,banana").

When to Use It

Use Split whenever you need to parse a single string that contains multiple pieces of data separated by a consistent character. The most common use case is parsing a line from a comma-separated value (CSV) file.

Example

```
// A string representing a row of comma-separated data.
// Note the extra space after the comma.
string csvLine = "John,Doe,35,New York";

// Define the character to split on.
char[] delimiters = { ',' };

// Split the string into an array of substrings.
string[] dataParts = csvLine.Split(delimiters);

Console.WriteLine("Data parts found:");
foreach (string part in dataParts)
{
    // We will need to trim the whitespace later.
    Console.WriteLine($"- '{part}'");
}

// Example with multiple delimiters and removing empty entries
string tags = "c#; programming,.net; guide";
char[] tagDelimiters = { ';', ',', '.' };
string[] cleanTags = tags.Split(tagDelimiters,
StringSplitOptions.RemoveEmptyEntries);
Console.WriteLine("\nClean tags: " + String.Join(", ", cleanTags));
```

Lecture 10: Trim

Introduction

The Trim() method is an instance method used to create a new string by removing all leading and trailing **whitespace** characters from the original string.

How It Works

"Whitespace" includes common characters like spaces, tabs (\t), and newlines (\n). The Trim() method does not affect whitespace within the middle of the string. It has two variations:

- **TrimStart()**: Removes whitespace only from the beginning of the string.
- **TrimEnd()**: Removes whitespace only from the end of thestring.

Like all string methods, it is immutable and returns a **new string**.

When to Use It

It is essential for cleaning up user input before processing or saving it. Users frequently add accidental leading or trailing spaces when typing. Trim() ensures that these extra spaces don't cause issues in comparisons, database lookups, or file paths.

Example

```
string rawUserInput = "   John Doe   ";
Console.WriteLine($"Raw: '{rawUserInput}'");

// Use Trim to clean up the input.
string cleanedInput = rawUserInput.Trim();
Console.WriteLine($"Cleaned: '{cleanedInput}'");

// Using TrimStart and TrimEnd
string messyPath = "   C:\\MyFiles\\   ";
string cleanStart = messyPath.TrimStart(); // "C:\\MyFiles\\   "
string cleanEnd = messyPath.TrimEnd();     // "   C:\\MyFiles\\"
```

Lecture 11: ToCharArray

Introduction

The ToCharArray() method is a simple instance method that converts a string into a character array (char[]).

How It Works

It creates a new character array where each element of the array is a character from the original string, in the same order.

When to Use It

You would use this when you need to perform complex, character-by-character manipulation that is easier or more efficient to do with an array. Since strings are immutable, you can't change individual characters. But you can convert the string to a mutable char array, perform your modifications on the array, and then convert it back to a string.

Example

Let's manually reverse a string using this technique.

```
string original = "Hello";

// 1. Convert the string to a char array.
char[] chars = original.ToCharArray();

// 2. Perform manipulations on the array (e.g., Array.Reverse).
Array.Reverse(chars);

// 3. Convert the modified char array back into a new string.
string reversed = new string(chars);

Console.WriteLine($"Original: {original}");
Console.WriteLine($"Reversed: {reversed}"); // Output: olleH
```

Lecture 12: Equals

Introduction

Comparing strings for equality is a very common task, and the `string` type provides multiple ways to do it, each with different performance and correctness implications.

The == Operator

This is a critical interview point. For most reference types, the == operator performs **reference equality** (checking if two variables point to the same object). However, the `string` type **overloads** this operator to perform a **value equality** check. This means `s1 == s2` will compare the actual character contents of the strings. This is done because it is the intuitive and expected behavior for strings. The comparison it performs is case-sensitive and culture-aware.

The String.Equals() Method

For more control, you should use the static `String.Equals()` method. Its most robust overload is: `public static bool Equals(string a, string b, StringComparison comparisonType)`

The StringComparison Enum

This enum is crucial for specifying **how** the comparison should be done.

- **StringComparison.Ordinal**: This performs a fast, byte-by-byte comparison. It does not consider linguistic rules. It is the **best choice for performance** and should be used when comparing things that are not for human display, like file paths, API keys, or protocol identifiers.
- **StringComparison.OrdinalIgnoreCase**: Same as `Ordinal`, but case-insensitive. This is the **recommended choice for most case-insensitive comparisons**.
- **StringComparison.CurrentCulture**: This performs a slower comparison that respects the linguistic rules of the user's current operating system culture (e.g., how different cultures sort characters).

- **StringComparison.CurrentCultureIgnoreCase**: Same as CurrentCulture, but case-insensitive. Use this when comparing strings for display to an end-user where cultural context matters.

Example

```
string s1 = "Strasse";
string s2 = "Straße"; // A German character that is equivalent to "ss" in some contexts.

// Using the default == operator (culture-aware)
// In some cultures, this might be true. In an Ordinal compare, it's false.

// --- The robust way ---
// Ordinal compare: Are the bytes identical?
bool areEqualOrdinal = String.Equals(s1, s2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: {areEqualOrdinal}"); // Output: False

// Culture-aware compare: Do these represent the same word in this culture?
bool areEqualCulture = String.Equals(s1, s2, StringComparison.CurrentCulture);
Console.WriteLine($"Current Culture comparison: {areEqualCulture}"); // Can be True on systems with German culture settings.
```

Lecture 13: Join

Introduction

String.Join is a powerful static method that does the opposite of Split. It concatenates the elements of a collection (like an array or List<string>) into a single new string, inserting a specified separator between each element.

How It Works

It takes a separator string and a collection of items. It iterates through the collection, efficiently building a new string by appending each item's string representation and the separator.

When to Use It

Use String.Join anytime you need to create a formatted, delimited string from a list of items. It is far more efficient and cleaner than using a loop and manually appending strings with a + operator. Common uses include creating a comma-separated list for display or generating a line for a CSV file.

Example

```
string[] names = { "Alice", "Bob", "Charlie", "David" };

// Join the names with a comma and a space.
string commaSeparatedNames = String.Join(", ", names);
Console.WriteLine(commaSeparatedNames); // Output: Alice, Bob, Charlie, David

// Join file path segments.
string[] pathSegments = { "C:", "Users", "JohnDoe", "Documents" };
string fullPath = String.Join("\\", pathSegments);
Console.WriteLine(fullPath); // Output: C:\Users\JohnDoe\Documents
```

Lecture 14: CompareTo

Introduction

The CompareTo method is an instance method that compares the current string object to another string to determine their relative position in a **sort order**. This method is the implementation of the IComparable interface for strings, and it's what methods like Array.Sort and List<T>.Sort use under the hood to sort strings alphabetically.

How It Works: The Return Value

CompareTo returns an integer with a specific meaning:

- **Less than 0:** The current string instance comes **before** the other string in the sort order (e.g., "Apple" comes before "Banana").
- **Zero:** The strings are **equal** in sort order.
- **Greater than 0:** The current string instance comes **after** the other string in the sort order (e.g., "Cat" comes after "Banana").

The comparison is culture-aware and case-sensitive by default.

Example

```
string s1 = "Apple";
string s2 = "Banana";
string s3 = "Apple";

int result1 = s1.CompareTo(s2);
// "Apple" comes before "Banana", so the result will be a negative number.
Console.WriteLine($"Comparing '{s1}' to '{s2}': {result1}");

int result2 = s2.CompareTo(s1);
// "Banana" comes after "Apple", so the result will be a positive number.
Console.WriteLine($"Comparing '{s2}' to '{s1}': {result2}");

int result3 = s1.CompareTo(s3);
// The strings are equal.
Console.WriteLine($"Comparing '{s1}' to '{s3}': {result3}"); // Output: 0
```

Lecture 15: StartsWith and EndsWith

Introduction

These are simple but very useful boolean instance methods for checking if a string begins or ends with a specific substring.

How They Work

- **bool StartsWith(string value)**: Returns true if the string instance begins with the specified substring.
- **bool EndsWith(string value)**: Returns true if the string instance ends with the specified substring.

By default, these comparisons are case-sensitive. However, they have important overloads that accept a StringComparison enum value, allowing you to easily perform case-insensitive checks.

Example

```
string filename = "MyReport.PDF";

// Case-sensitive check (default)
bool endsWithPdfLower = filename.EndsWith(".pdf");
Console.WriteLine($"Ends with '.pdf'? {endsWithPdfLower}"); // Output: False

// Case-insensitive check (best practice)
bool endsWithPdfIgnoreCase = filename.EndsWith(".pdf", StringComparison.OrdinalIgnoreCase);
Console.WriteLine($"Ends with '.pdf' (ignore case)? {endsWithPdfIgnoreCase}"); // Output: True

string url = "[https://www.google.com](https://www.google.com)";
if (url.StartsWith("https://"))
{
    Console.WriteLine("URL is secure.");
}
```

Lecture 16: Contains

Introduction

The Contains method is a boolean instance method that checks if a specified substring exists **anywhere** within the string.

How It Works

It returns true if the substring is found, and false otherwise. Like StartsWith and EndsWith, it is case-sensitive by default. Modern versions of .NET include an overload that accepts a StringComparison enum for case-insensitive checks.

Interview Perspective: In older .NET versions, Contains did not have a case-insensitive overload. The common workaround, and a good thing to know for an interview, was to use myString.IndexOf("substring", StringComparison.OrdinalIgnoreCase) >= 0.

Example

```
string sentence = "The quick brown fox jumps over the lazy dog.";

bool hasFox = sentence.Contains("fox");
Console.WriteLine($"Contains 'fox'? {hasFox}"); // Output: True

bool hasCat = sentence.Contains("cat");
Console.WriteLine($"Contains 'cat'? {hasCat}"); // Output: False

// Case-insensitive check
bool hasQuickIgnoreCase = sentence.Contains("QUICK",
StringComparison.OrdinalIgnoreCase);
Console.WriteLine($"Contains 'QUICK' (ignore case)?
{hasQuickIgnoreCase}"); // Output: True
```

Lecture 17: IndexOf and LastIndexOf

Introduction

These instance methods are used to find the position of a character or substring within a string.

- **IndexOf(char/string value)**: Searches the string from beginning to end and returns the zero-based index of the **first occurrence** of the specified character or substring.
- **LastIndexOf(char/string value)**: Searches the string from end to beginning and returns the zero-based index of the **last occurrence** of the specified character or substring.

How It Works

If the specified character or substring is not found, both methods return **-1**. They also have overloads that allow you to specify a starting index for the search, giving you more control over where the search begins.

Example

```
string path = "C:\\Users\\JohnDoe\\Documents\\MyFile.txt";

// Find the first occurrence of 'o'
int firstO = path.IndexOf('o');
Console.WriteLine($"First 'o' is at index: {firstO}"); // Output: 10

// Find the last occurrence of 'o'
int lastO = path.LastIndexOf('o');
Console.WriteLine($"Last 'o' is at index: {lastO}"); // Output: 22

// Find the index of the file extension dot.
int lastBackslash = path.LastIndexOf('\\');
string filename = path.Substring(lastBackslash + 1);
Console.WriteLine($"Filename is: {filename}"); // Output: MyFile.txt
```

Lecture 18: IsNullOrWhiteSpace

Introduction

String.IsNullOrWhiteSpace is a static utility method that provides a robust check for "empty" strings. It is an enhancement over String.IsNullOrEmpty.

How It Works

This method returns true if a string is:

- null
- An empty string ("")
- Consists **only of whitespace characters** (spaces, tabs, newlines, etc.).

This is what makes it more powerful than IsNullOrEmpty, which would return false for a string like " ".

When to Use It

This should be your **preferred method** for validating user input or any string that should contain actual content. It correctly handles cases where a user might enter just spaces into a required text field.

Example

```
string s1 = null;
string s2 = "";
string s3 = "   \t   "; // Just whitespace
string s4 = "Hello";

Console.WriteLine($"s1 is null or whitespace?
{String.IsNullOrWhiteSpace(s1)}"); // True
Console.WriteLine($"s2 is null or whitespace?
{String.IsNullOrWhiteSpace(s2)}"); // True
Console.WriteLine($"s3 is null or whitespace?
{String.IsNullOrWhiteSpace(s3)}"); // True
Console.WriteLine($"s4 is null or whitespace?
{String.IsNullOrWhiteSpace(s4)}"); // False
```

Lecture 19: Strings with For Loop

Introduction

While C# and its LINQ library provide many high-level, convenient methods for string manipulation, interviewers often ask you to solve string problems using basic loops. This isn't because you would do it this way in daily work, but because it tests your fundamental problem-solving skills, your logical thinking, and your ability to handle edge cases manually. A string can be treated like a read-only array of characters (char). You can access any character by its index using myString[i] and get its total length with myString.Length. This makes the for loop a perfect tool for low-level string manipulation. This lecture covers a comprehensive set of classic interview problems solved using this fundamental approach.

Category 1: Basic Manipulation & Counting

These examples cover fundamental iteration and conditional logic.

Example 1: Count a Specific Character

- **Task:** Write a function to count how many times a specific character appears in a string.
- **Logic:** Initialize a counter to zero. Loop through each character of the string. In each iteration, if the character at the current index matches the target character, increment the counter.

```
public int CountCharacter(string input, char charToFind)
{
    if (string.IsNullOrEmpty(input))
    {
        return 0;
    }

    int count = 0;
    for (int i = 0; i < input.Length; i++)
    {
        if (input[i] == charToFind)
        {
            count++;
        }
    }
    return count;
}
```

Example 2: Count Vowels and Consonants

Task: Write a function that counts the number of vowels and consonants in a string.

Logic: Iterate through the string. For each character, first check if it's a letter. If it is, then check if it's one of the vowels (a, e, i, o, u). If so, increment the vowel counter; otherwise, increment the consonant counter. It's best practice to convert the character to lowercase first to simplify the vowel check.

```
  public void CountVowelsAndConsonants(string input, out int
vowelCount, out int consonantCount)

 {
    vowelCount = 0;
    consonantCount = 0;
    if (string.IsNullOrEmpty(input)) return;

    string vowels = "aeiou";
    for (int i = 0; i < input.Length; i++)
    {
       char c = char.ToLower(input[i]);
       if (char.IsLetter(c))
       {
          if (vowels.Contains(c))
          {
             vowelCount++;
          }
          else
          {
             consonantCount++;
          }
       }
    }
 }
```

Example 3: Extract All Digits from a String

Task: Write a function that takes a string and returns a new string containing only the digits found in the original.

Logic: Use a StringBuilder for efficient string building. Iterate through the input string. For each character, use the built-in char.IsDigit() method to check if it's a number. If it is, append it to the StringBuilder.

```
  public string ExtractDigits(string input)

 {
    if (string.IsNullOrEmpty(input)) return "";

    StringBuilder digitsOnly = new StringBuilder();
    for (int i = 0; i < input.Length; i++)
```

```
    {
        if (char.IsDigit(input[i]))
        {
            digitsOnly.Append(input[i]);
        }
    }
    return digitsOnly.ToString();
}
```

Category 2: Reversing & Palindromes

These examples test classic algorithmic patterns.

Example 4: Reverse a String Manually

Task: Write a function that returns a new string with its characters in reverse order.

Logic: The most efficient way is to iterate through the original string from back to front and build a new string using a StringBuilder.

```
public string ReverseString(string input)

{
    if (string.IsNullOrEmpty(input)) return input;

    StringBuilder reversedString = new StringBuilder(input.Length);
    for (int i = input.Length - 1; i >= 0; i--)
    {
        reversedString.Append(input[i]);
    }
    return reversedString.ToString();
}
```

Example 5: Check for a Palindrome

Task: Write a function that returns true if a string is a palindrome (reads the same forwards and backwards).

Logic: Use a "two-pointer" approach. One pointer starts at the beginning (left) and one starts at the end (right). Compare the characters at each pointer. If they don't match, it's not a palindrome. If they do, move both pointers one step closer to the middle and repeat until they cross.

```
public bool IsPalindrome(string input)

{
    if (string.IsNullOrEmpty(input)) return true;

    int left = 0;
    int right = input.Length - 1;
    while (left < right)
    {
        if (char.ToLower(input[left]) != char.ToLower(input[right]))
        {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Example 6: Check if Two Strings are Anagrams

Task: Determine if two strings are anagrams of each other (i.e., they contain the same characters in different orders, like "listen" and "silent").

Logic: A common approach is to use frequency count arrays. If both strings have the same counts of each character, they are anagrams.

```
public bool AreAnagrams(string s1, string s2)

{
    if (s1.Length != s2.Length) return false;

    int[] charCounts = new int[256]; // For ASCII characters

    // Count characters for the first string
    for (int i = 0; i < s1.Length; i++)
    {
        charCounts[s1[i]]++;
    }

    // Decrement counts for the second string
    for (int i = 0; i < s2.Length; i++)
    {
        charCounts[s2[i]]--;
    }

    // If they are anagrams, all counts should be zero.
    for (int i = 0; i < charCounts.Length; i++)
    {
        if (charCounts[i] != 0)
        {
            return false;
        }
    }
    return true;
}
```

Category 3: Character Frequency & Duplicates

These examples test your ability to use helper data structures to track information.

Example 7: Find the Most Frequent Character

Task: Find the character that appears most often in a string.

Logic: Use a frequency map (an integer array) to store the count for each possible character. Iterate through the string once to populate the map, then iterate through the map to find the character with the highest count.

```
public char FindMostFrequentCharacter(string input)

{
    if (string.IsNullOrEmpty(input)) throw new
ArgumentException("Input is empty.");

    int[] charCounts = new int[256];
    for (int i = 0; i < input.Length; i++)
    {
        charCounts[input[i]]++;
    }

    int maxCount = -1;
    char maxChar = ' ';
    for (int i = 0; i < charCounts.Length; i++)
    {
        if (charCounts[i] > maxCount)
        {
            maxCount = charCounts[i];
            maxChar = (char)i;
        }
    }
    return maxChar;
}
```

Example 8: Remove Duplicate Characters

Task: Return a new string with all duplicate characters removed (e.g., "programming" becomes "progamin").

Logic: Iterate through the input string. Use a HashSet<char> to keep track of characters you have already "seen" and added to your result. A HashSet provides an extremely fast Contains check.

```
public string RemoveDuplicates(string input)

{
    if (string.IsNullOrEmpty(input)) return input;

    StringBuilder result = new StringBuilder();
    HashSet<char> seenCharacters = new HashSet<char>();

    for (int i = 0; i < input.Length; i++)
    {
        if (!seenCharacters.Contains(input[i]))
        {
            seenCharacters.Add(input[i]);
            result.Append(input[i]);
        }
    }
    return result.ToString();
}
```

Example 9: Find the First Non-Repeated Character

Task: Find the first character in a string that does not repeat itself anywhere else in the string.

Logic: This often requires two passes. The first pass populates a frequency map (like a Dictionary<char, int>). The second pass iterates through the original string again and returns the first character it finds that has a count of 1 in the frequency map.

```
public char? FindFirstNonRepeatedChar(string input)

{
    if (string.IsNullOrEmpty(input)) return null;

    var charCounts = new Dictionary<char, int>();
    for (int i = 0; i < input.Length; i++)
    {
        char c = input[i];
        if (charCounts.ContainsKey(c))
        {
```

```
            charCounts[c]++;
        }
        else
        {
            charCounts[c] = 1;
        }
    }

    for (int i = 0; i < input.Length; i++)
    {
        if (charCounts[input[i]] == 1)
        {
            return input[i];
        }
    }
    return null; // No non-repeated character found
}
```

Category 4: Word & Sentence Manipulation

These examples test your ability to break a problem down into smaller steps.

Example 10: Reverse the Order of Words in a Sentence

Task: Given "the quick brown fox", return "fox brown quick the".

Logic: This is not a simple character reversal. First, split the string into an array of words. Then, iterate through that array backwards, appending each word to a StringBuilder.

```
public string ReverseWords(string sentence)

{
    if (string.IsNullOrEmpty(sentence)) return sentence;

    string[] words = sentence.Split(' ');
    StringBuilder reversedSentence = new StringBuilder();
```

```
    for (int i = words.Length - 1; i >= 0; i--)
    {
        reversedSentence.Append(words[i]);
        if (i > 0) // Add a space unless it's the last word
        {
            reversedSentence.Append(" ");
        }
    }
    return reversedSentence.ToString();
}
```

Example 11: Capitalize the First Letter of Each Word

Task: Convert a string to "title case" where the first letter of each word is capitalized.

Logic: Split the sentence into an array of words. Loop through the array. For each word, create a new word string by taking the first character, converting it to uppercase, and appending the rest of the word's substring. Join the modified words back together.

```
public string ToTitleCase(string sentence)

{
    if (string.IsNullOrWhiteSpace(sentence)) return sentence;

    string[] words = sentence.Split(' ');
    for (int i = 0; i < words.Length; i++)
    {
        if(words[i].Length > 0)
        {
            words[i] = char.ToUpper(words[i][0]) + words[i].Substring(1);
        }
    }
    return string.Join(" ", words);
}
```

Example 12: Find the Longest Word in a Sentence

Task: Find and return the longest word from a given sentence.

Logic: Split the sentence into an array of words. Initialize a variable longestWord to an empty string. Loop through the words array. In each iteration, if the current word's length is greater than the length of longestWord, update longestWord.

```
public string FindLongestWord(string sentence)

{
    if (string.IsNullOrWhiteSpace(sentence)) return "";

    string[] words = sentence.Split(' ');
    string longestWord = "";

    for(int i = 0; i < words.Length; i++)
    {
        if (words[i].Length > longestWord.Length)
        {
            longestWord = words[i];
        }
    }
    return longestWord;
}
```

Category 5: Advanced String Logic

These examples mimic more difficult interview challenges.

Example 13: String to Integer (atoi)

Task: Manually implement the int.Parse method without using built-in parsing functions.

Logic: This is complex. You need to handle optional leading whitespace, an optional sign (+ or -), and then build the number digit by digit while checking for overflow.

```
// Note: This is a simplified version. A full implementation is much more complex.
public int Atoi(string s)
{
    if (string.IsNullOrEmpty(s)) return 0;

    long result = 0;
    int sign = 1;
    int i = 0;

    while (i < s.Length && s[i] == ' ') i++; // Skip whitespace

    if (i < s.Length && (s[i] == '+' || s[i] == '-'))
    {
        sign = (s[i] == '-') ? -1 : 1;
        i++;
    }

    while (i < s.Length && char.IsDigit(s[i]))
    {
        result = result * 10 + (s[i] - '0'); // 'char' - '0' gives the int value
        if (result * sign > int.MaxValue) return int.MaxValue;
        if (result * sign < int.MinValue) return int.MinValue;
        i++;
    }

    return (int)(result * sign);
}
```

Example 14: Find a Substring (Implement IndexOf)

Task: Manually implement string.IndexOf to find the starting index of a substring (needle) within a larger string (haystack).

Logic: Use nested loops. The outer loop iterates through the haystack. The inner loop starts from the current outer loop position and checks if the characters match the needle sequentially.

```
public int IndexOf(string haystack, string needle)

{
    if (string.IsNullOrEmpty(needle)) return 0;
    if (haystack.Length < needle.Length) return -1;

    for (int i = 0; i <= haystack.Length - needle.Length; i++)
    {
        int j;
        for (j = 0; j < needle.Length; j++)
        {
            if (haystack[i + j] != needle[j])
            {
                break; // Mismatch, break inner loop
            }
        }
        // If the inner loop completed without a break, we found a match.
        if (j == needle.Length)
        {
            return i;
        }
    }
    return -1; // Not found
}
```

Example 15: Validate an IP Address String

Task: Check if a string is a valid IPv4 address (e.g., "192.168.1.1").

Logic: An IP address must have four parts separated by three dots. Each part must be a number between 0 and 255 and cannot have leading zeros (unless it's just "0").

```
public bool IsValidIpAddress(string ip)

{
    string[] parts = ip.Split('.');
    if (parts.Length != 4) return false;

    for (int i = 0; i < parts.Length; i++)
    {
        // Use TryParse for safe number conversion
        if (!int.TryParse(parts[i], out int num)) return false;

        // Check range 0-255
        if (num < 0 || num > 255) return false;

        // Check for leading zeros (e.g., "01" is invalid)
        if (parts[i].Length > 1 && parts[i][0] == '0') return false;
    }
    return true;
}
```

Lecture 20: StringBuilder

Introduction

The **StringBuilder** class, found in the System.Text namespace, is a crucial tool for building and manipulating strings efficiently. Unlike the standard string class which is immutable, StringBuilder represents a **mutable** sequence of characters.

Why and When to Use It: Performance

This is the most important reason to use a StringBuilder. Because strings are immutable, performing many concatenations in a loop (e.g., result = result + nextString;) is extremely inefficient. Each + operation creates a brand new string object in memory, leading to a large number of temporary objects that the Garbage Collector must clean up.

A StringBuilder avoids this by using a single, mutable internal buffer (an array of characters). When you append text, it adds to this buffer. If the buffer gets full, it allocates a new, larger buffer and copies the content, but this resizing happens far less frequently than with simple string concatenation.

Best Practice: Any time you are building a string within a loop or through a significant number of concatenations, you should use a StringBuilder.

Key Methods

- **Append(value)**: Adds text or the string representation of an object to the end of the buffer.
- **AppendLine(value)**: Adds text followed by a new line character.
- **Insert(index, value)**: Inserts text at a specific position.
- **Remove(startIndex, length)**: Removes a specified number of characters.
- **Replace(oldValue, newValue)**: Replaces all occurrences of a string with another.
- **ToString()**: This is the final step. It converts the mutable StringBuilder content into a regular, immutable string.

Example: Efficient String Concatenation

```
// Inefficient way using string concatenation
// string s = "";
// for (int i = 0; i < 10000; i++)
// {
//     s += i; // Creates 10,000 temporary string objects! Very slow.
// }

// Efficient way using StringBuilder
Console.WriteLine("Building a long string efficiently...");
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.Append(i); // Modifies the internal buffer. Very fast.
    sb.Append(", ");
}
string finalResult = sb.ToString(); // Create the final string only once.
Console.WriteLine("Done. The final string is very long.");
// Console.WriteLine(finalResult);
```

Lecture 21: DateTime

Introduction

The **DateTime** struct is the primary type in .NET for representing a specific moment in time, including both the date and the time of day. As it is a struct, it is a value type and cannot be null (though DateTime? can be).

Creating DateTime Instances

- **Current Time:**
    - **DateTime.Now**: Gets a DateTime object set to the current date and time on the **local machine**, respecting its time zone settings. Use this for display purposes for the user.
    - **DateTime.UtcNow**: Gets the current **Coordinated Universal Time**. This is the recommended choice for **storing timestamps** in a database or logs, as it is unambiguous and not affected by daylight saving changes or server locations.
- **Specific Time:**
    - You can create a specific date and time using its constructor: new DateTime(year, month, day, hour, minute, second)

Key Properties

Once you have a DateTime object, you can easily access its individual components:

- .Year, .Month, .Day
- .Hour, .Minute, .Second, .Millisecond
- .DayOfWeek: Returns a DayOfWeek enum value (e.g., DayOfWeek.Saturday).
- .DayOfYear: Returns the day number of the year (1 to 366).

- .TimeOfDay: Returns a TimeSpan representing just the time portion of the date.

Formatting for Display

The .ToString() method is very powerful and accepts format specifiers to control how the date is displayed.

```
DateTime now = DateTime.Now;

// Standard formats
Console.WriteLine($"Short date (d): {now.ToString("d")}");
Console.WriteLine($"Long date (D): {now.ToString("D")}");
Console.WriteLine($"Full (G): {now.ToString("G")}");

// Custom format for logging or APIs
Console.WriteLine($"ISO 8601 Style (s): {now.ToString("s")}");
Console.WriteLine($"Custom yyyy-MM-dd: {now.ToString("yyyy-MM-dd HH:mm:ss")}");
```

Lecture 22: Date Subtraction and Addition

Introduction

Performing calculations with DateTime objects is a common requirement, such as finding the duration between two events or calculating a future date.

Date Subtraction

When you subtract one DateTime from another (date2 - date1), the result is not another DateTime. The result is a **TimeSpan** object, which represents a duration or interval of time.

The TimeSpan object has useful properties to get the duration in different units:

- TotalDays
- TotalHours
- TotalMinutes
- TotalSeconds

Example:

```
DateTime projectStartDate = new DateTime(2025, 1, 15);
DateTime projectEndDate = new DateTime(2025, 7, 12);

TimeSpan projectDuration = projectEndDate - projectStartDate;

Console.WriteLine($"The project has been running for
{projectDuration.Days} days.");
Console.WriteLine($"Which is {projectDuration.TotalHours} hours in
total.");
```

Date Addition

You cannot add two DateTime objects together. Instead, you add a TimeSpan to a DateTime, or you use the built-in Add... methods. Since DateTime is a value type, these methods are immutable; they return a **new** DateTime object with the result and do not modify the original.

Example:

```
DateTime today = new DateTime(2025, 7, 12);

// Using Add methods is very readable
DateTime nextWeek = today.AddDays(7);
DateTime lastMonth = today.AddMonths(-1); // Use negative numbers to
subtract
DateTime twoHoursAndThirtyMinutesLater =
today.AddHours(2).AddMinutes(30);

Console.WriteLine($"Today is: {today:d}");
Console.WriteLine($"One week from now will be: {nextWeek:d}");
Console.WriteLine($"One month ago was: {lastMonth:d}");
```

Lecture 23: Math

Introduction

System.Math is a built-in **static class** that provides constants and static methods for common trigonometric, logarithmic, and other mathematical functions. You do not create an instance of the Math class; you call its methods directly on the class name.

Common Constants and Methods

- **Constants:**
  - Math.PI: The ratio of the circumference of a circle to its diameter (3.14159...).
  - Math.E: The natural logarithmic base (2.71828...).
- **Basic Operations:**
  - Math.Abs(x): Returns the absolute value of a number.
    Math.Abs(-5) is 5.
  - Math.Sign(x): Returns -1, 0, or 1 to indicate the sign of a number.
  - Math.Max(x, y): Returns the larger of two numbers.
  - Math.Min(x, y): Returns the smaller of two numbers.
- **Powers and Roots:**
  - Math.Pow(x, y): Returns a specified number raised to the specified power (x^y).
  - Math.Sqrt(x): Returns the square root of a number.
- **Rounding:**
  - Math.Round(x): Rounds a value to the nearest integral value.
  - Math.Ceiling(x): Always rounds **up** to the next whole number.
    Math.Ceiling(9.01) is 10.
  - Math.Floor(x): Always rounds **down** to the previous whole number.
    Math.Floor(9.99) is 9.

Interview Perspective: Banker's Rounding

A common interview question relates to Math.Round. By default, it uses a strategy called "Round to Nearest Even," or Banker's Rounding. This means that if a number is exactly halfway between two integers (like 2.5), it will round to the **nearest even integer**.

- Math.Round(2.5) returns **2**.
- Math.Round(3.5) returns **4**. This strategy is used to reduce statistical bias in large financial or scientific calculations.

Lecture 24: Regular Expressions

Introduction

Regular Expressions (or **Regex**) are a powerful, standardized mini-language for **pattern matching in text**. They are used for complex validation, searching, finding, and replacing text that would be difficult or impossible with standard string methods.

The Regex Class

In C#, you work with regular expressions using the System.Text.RegularExpressions.Regex class. The most common methods are static helpers on this class.

Common Regex Characters (a mini-dictionary)

- **Characters:**
  - .: Matches any single character except newline.
  - \d: Matches any digit (0-9).
  - \w: Matches any word character (letters, numbers, underscore).
  - \s: Matches any whitespace character (space, tab, newline).

- **Quantifiers:**
  - *: 0 or more times.
  - +: 1 or more times.
  - ?: 0 or 1 time.
  - {n}: Exactly n times. {n,m}: Between n and m times.
- **Anchors:**
  - ^: Start of the string.
  - $: End of the string.
- **Groups and Ranges:**
  - [...]: Matches any single character within the brackets (e.g., [aeiou]).
  - (...): Creates a capturing group.
  - |: Acts as an OR operator.

Real-World Examples

- **Example 1: Validating a US Phone Number**
  - **Task:** Check if a string matches the format XXX-XXX-XXXX.
    string pattern = @"^\d{3}-\d{3}-\d{4}$"; string phone1 = "555-123-4567"; // Valid string phone2 = "555-123-456"; // Invalid Console.WriteLine("Is'phone1'avalidphonenumber? Regex.IsMatch(phone1,pattern)");Console.WriteLine("Is '{phone2}' a valid phone number? {Regex.IsMatch(phone2, pattern)}");
- **Example 2: Extracting a Key and Value**
  - **Task:** Extract the key and value from a string like "ApiKey=xyz123;".
    string input = "ApiKey=xyz123;"; string pattern = @"^(?
    Match match = Regex.Match(input, pattern);
    if (match.Success) { string key = match.Groups["key"].Value; string value = match.Groups["value"].Value; Console.WriteLine($"\nFound Key: {key}, Value: {value}"); }

Section 27: IO, Serialization, Encoding

Lecture 1: Intro to Number Systems

Introduction

In our daily lives, we use the **decimal (base-10)** number system, which utilizes ten unique digits (0 through 9). However, computers operate on a fundamentally different system. Understanding the number systems that computers use—primarily **binary (base-2)**—and the human-readable representations like **hexadecimal (base-16)** is crucial for working with low-level data, memory, networking, and file formats.

How It Works: The Concept of a "Base"

The **base** (or **radix**) of a number system is the number of unique digits used to represent numbers. The position of a digit determines its value, which is based on powers of that base.

- **Decimal (Base-10):** Uses 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The place values are powers of 10: 1 (10^0), 10 (10^1), 100 (10^2), etc.
- **Binary (Base-2):** Uses 2 digits (0, 1). The place values are powers of 2.
- **Hexadecimal (Base-16):** Uses 16 symbols (0-9 and A-F). The place values are powers of 16.

Lecture 2: Binary Number System

Introduction

The **binary (base-2)** number system is the native language of all digital computers. It uses only two digits, **0** and **1**, to represent all data. Each binary digit is called a **bit**. Eight bits together form a **byte**, which is the standard unit of data storage in computing.

How It Works: Place Values as Powers of 2

In a binary number, each position, reading from right to left, represents an increasing power of 2.

Power of 2	128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Binary Digit	0	1	0	1	1	0	1	0

Mathematical Conversions

- **Binary to Decimal:** To convert a binary number to decimal, you multiply each binary digit by its corresponding power-of-2 place value and add the results. **Example: Convert binary 1101 to decimal.**

  1. (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)
  2. (1 * 8) + (1 * 4) + (0 * 2) + (1 * 1)
  3. 8 + 4 + 0 + 1 = 13. So, 1101 in binary is 13 in decimal.

- **Decimal to Binary:** To convert a decimal number to binary, you use the method of **repeated division by 2**. You repeatedly divide the decimal number by 2, recording the remainder each time. The binary result is the sequence of remainders read from **bottom to top**. **Example: Convert decimal 22 to binary.**

  1. 22 / 2 = 11, Remainder **0**
  2. 11 / 2 = 5, Remainder **1**
  3. 5 / 2 = 2, Remainder **1**
  4. 2 / 2 = 1, Remainder **0**
  5. 1 / 2 = 0, Remainder **1** Reading the remainders from bottom to top gives **10110**.

C# Syntax

You can declare a binary literal in C# by prefixing the number with 0b.

```
int myValue = 0b10110; // This is the integer 22
Console.WriteLine(myValue);
```

Lecture 3: Octal Number System

Introduction

The **octal (base-8)** number system uses eight unique digits: 0, 1, 2, 3, 4, 5, 6, and 7. It was more common in older computing systems because it provides a convenient way to represent binary numbers (since 8 = 2^3, one octal digit represents exactly three binary digits).

How It Works: Place Values as Powers of 8

Each position in an octal number represents a power of 8 (1, 8, 64, 512, etc.).

Mathematical Conversions

- **Octal to Decimal: Example: Convert octal 37 to decimal.**
    1. (3 * 8^1) + (7 * 8^0)
    2. (3 * 8) + (7 * 1)
    3. 24 + 7 = 31. So, 37 in octal is 31 in decimal.
- **Decimal to Octal:** Use repeated division by 8. **Example: Convert decimal 140 to octal.**
    1. 140 / 8 = 17, Remainder **4**
    2. 17 / 8 = 2, Remainder **1**
    3. 2 / 8 = 0, Remainder **2** Reading from bottom to top gives **214**.

C# Syntax

C# does not have a native literal prefix for octal numbers. To work with them, you must convert from a string, specifying the base.

```
string octalString = "214";
int myValue = Convert.ToInt32(octalString, 8); // The '8' specifies base-8.
Console.WriteLine(myValue); // Output: 140
```

Lecture 4: Hexadecimal Number System

Introduction

The **hexadecimal (base-16)** number system is extremely common in modern programming, especially for anything related to memory, graphics, and low-level data. It uses 16 unique symbols: the numbers **0-9** and the letters **A-F**.

- A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

Why It's Used

Hexadecimal provides a much more compact and human-readable way to represent binary data. Since 16 = 2^4, **one hexadecimal digit represents exactly four binary digits** (also known as a "nibble"). This makes it very easy to represent a full byte (FF in hex is 11111111 in binary). It's used everywhere for:

- **Memory Addresses:** 0x00A1F8C0
- **Color Codes:** #FF0000 for Red (FF for Red, 00 for Green, 00 for Blue)
- **Raw Byte Data:** Displaying the contents of a file or network packet.

Mathematical Conversions

- **Hex to Decimal: Example: Convert hex 1A to decimal.**
    1. (1 * 16^1) + (A * 16^0)
    2. (1 * 16) + (10 * 1)
    3. 16 + 10 = 26. So, 1A in hex is 26 in decimal.
- **Decimal to Hex:** Use repeated division by 16. **Example: Convert decimal 250 to hex.**
    1. 250 / 16 = 15, Remainder **10** (which is A)
    2. 15 / 16 = 0, Remainder **15** (which is F) Reading from bottom to top gives **FA**.

C# Syntax

You declare a hexadecimal literal in C# by prefixing the number with 0x.

```
int redComponent = 0xFF; // FF in hex is 255 in decimal
Console.WriteLine(redComponent);
```

Lecture 5: Intro to Character Encoding

Introduction

Character encoding is a system that defines a mapping between characters (the letters, numbers, and symbols you see on screen) and the numerical values that a computer can actually store and process. This is necessary because computers fundamentally only understand numbers, specifically binary data (0s and 1s). An encoding standard provides the crucial link between the human-readable character 'A' and its binary representation, for example, 01000001.

The Problem: Why So Many Encodings?

Early encoding systems were simple and designed for a specific language (usually English). They didn't have enough room to represent all the characters from all the world's languages (like €, é, И, 日). This led to the development of different standards for different regions and, ultimately, to the creation of **Unicode**, a universal standard that aims to solve this problem once and for all.

Lecture 6: ASCII Character Encoding

Introduction

ASCII (American Standard Code for Information Interchange) is one of the earliest and most influential character encoding standards. It formed the basis for most subsequent encodings.

How It Works

Standard ASCII is a **7-bit** encoding scheme. With 7 bits, it can represent 2^7, or **128**, unique characters. This is enough to include:

- Uppercase English letters (A-Z)
- Lowercase English letters (a-z)
- Numbers (0-9)
- Common punctuation (!, @, ?, etc.)
- Control characters (like newline, tab, and backspace).

Since computers typically work with 8-bit bytes, the extra bit was often used for proprietary "extended ASCII" variations, which included graphics or language-specific characters. However, these variations were not standardized, leading to major compatibility problems when files were shared between different systems.

Limitations

The primary limitation of ASCII is its small size. It cannot represent characters from other languages, accented characters (like é), or a vast range of symbols (like © or €), which made it unsuitable for a global computing environment.

Lecture 7: Unicode Character Encoding

Introduction

Unicode is the modern, universal character encoding standard that solves the limitations of ASCII. Its goal is to provide a unique numerical value, called a **code point**, for every character from every language in the world, including a vast array of symbols, emojis, and historical scripts.

How It Works: Unicode vs. UTF

It's important to distinguish between the Unicode standard and its encoding schemes.

- **Unicode** is the standard itself—the giant table that maps characters like 'A' to the code point U+0041 or '€' to U+20AC.
- An **encoding scheme** (like UTF-8) defines how these numeric code points are actually stored as a sequence of bytes in memory or a file.

The two most important encoding schemes are:

- **UTF-8 (Unicode Transformation Format - 8-bit)**
  - This is the **dominant encoding on the web** and for most files today.
  - It is a **variable-width** encoding. It cleverly uses only **1 byte** for all the common ASCII characters, making it 100% backward compatible with ASCII. For other characters, it uses 2, 3, or up to 4 bytes.
  - Its efficiency with English text and its broad compatibility make it the default choice for most applications.
- **UTF-16**
  - This is the native encoding used by the .NET `char` and `string` types **internally**.
  - It is also variable-width, using **2 bytes** for most common characters (covering most modern languages) and 4 bytes (a "surrogate pair") for less common characters, like some emojis.

Interview Perspective

- **Question:** "What character encoding does a C# string use internally?"
- **Ideal Answer:** "A C# string represents text as a sequence of UTF-16 code units. This means each `char` in the string is a 16-bit value."
- **Question:** "What's the difference between ASCII and UTF-8?"
- **Ideal Answer:** "ASCII is a very old, limited 7-bit character set that can only represent 128 English characters and symbols. UTF-8 is a modern, variable-width encoding scheme for the entire Unicode standard. It's backward compatible with ASCII, meaning any ASCII file is also a valid UTF-8 file, but it can also represent every character in the world by using multiple bytes for non-ASCII characters."

Lecture 8: Intro to System.IO namespace

Introduction

The **System.IO** namespace is the primary namespace in .NET for performing **IO (Input/Output)** operations. It contains a rich set of classes for working with the file system, including reading from and writing to files, manipulating directories, and working with data streams.

Key Concepts

There are two main approaches to working with files and directories provided by this namespace:

1. **Static Helper Classes (File and Directory)**
    - These classes provide **static methods** for quick, one-off operations. For example, File.ReadAllText() reads an entire file in a single line of code.
    - They are very simple to use for common tasks.
    - They can be less efficient for performing many repeated operations on the same file because security checks are performed each time a method is called.
2. **Instance Classes (FileInfo and DirectoryInfo)**
    - These classes represent a specific file or directory **as an object**. You first create an instance of the class (e.g., FileInfo myFile = new FileInfo("path/to/file.txt");), and then you call instance methods on that object (e.g., myFile.Delete(), myFile.CopyTo(...)).
    - This approach is more efficient for performing multiple operations on the same file or directory because the initial security checks are done only once when the object is created.

The Concept of a Stream A **Stream** is a fundamental abstraction in System.IO. It represents a sequence of bytes that can be read from or written to. This is a powerful concept because it provides a unified way to handle data transfer. The source or destination of the bytes could be a file (FileStream), a network connection (NetworkStream), or even just a block of memory (MemoryStream), but your code can treat them all as a generic Stream.

Lecture 9: File class

Introduction

The **File** class is a **static helper class** that provides a set of simple, static methods for performing common file operations like reading, writing, copying, moving, and deleting files. You do not create an instance of the File class; you call its methods directly.

When to Use It

Use the File class for simple, single-shot operations on a file. If you just need to quickly read all the text from a small file or check if a file exists, the File class is the easiest and most readable way to do it.

Key Static Methods

- **bool Exists(string path)**: Checks if a file exists at the specified path.
- **string ReadAllText(string path)**: Opens a file, reads its entire contents into a single string, and then closes the file.
- **string[] ReadAllLines(string path)**: Opens a file, reads all lines into a string[] array, and then closes the file.
- **void WriteAllText(string path, string contents)**: Creates a new file (or overwrites an existing one) and writes the specified string to it.
- **void AppendAllText(string path, string contents)**: Appends the specified string to the end of a file. If the file doesn't exist, it creates it.
- **void WriteAllLines(string path, string[] contents)**: Writes each string from the array as a new line in a file.

- **void Copy(string sourcePath, string destPath)**: Copies a file.
- **void Move(string sourcePath, string destPath)**: Moves (or renames) a file.
- **void Delete(string path)**: Deletes a file.

Lecture 10: Read/Write Operations using File class

Introduction

This lecture provides practical examples of the most common read and write methods from the static File class. These methods are designed for simplicity and convenience, especially when dealing with text files.

How It Works: Loading into Memory

It's crucial to understand that the "ReadAll" methods (ReadAllText, ReadAllLines) load the **entire contents** of the file into memory at once. This makes them very easy to use but can be inefficient and consume a large amount of RAM if you are working with very large files (e.g., gigabytes in size). For large files, stream-based operations are the correct approach.

Example 1: Writing and Appending to a Text File

This example demonstrates how to create a file, write initial content, and then append more content without deleting what was already there.

```
string filePath = "MyLog.txt";

// --- Writing to a file ---
// This will create 'MyLog.txt' or overwrite it if it already exists.
string initialContent = "Log started at " + DateTime.Now + "\n";
File.WriteAllText(filePath, initialContent);
Console.WriteLine("Initial log created.");

// --- Appending to a file ---
// AppendAllText adds to the end of the file. If the file doesn't exist, it creates it.
string nextEntry = "User 'Alice' logged in.\n";
```

```
File.AppendAllText(filePath, nextEntry);
Console.WriteLine("New entry appended.");
```

Example 2: Reading a Text File

This example shows the two primary ways to read a text file's contents.

string filePath = "MyLog.txt"; // Assuming this file was created by the above example.

```
// --- Reading the entire file into a single string ---
Console.WriteLine("\n--- Reading file with ReadAllText ---");
if (File.Exists(filePath))
{
    string fileContents = File.ReadAllText(filePath);
    Console.WriteLine(fileContents);
}

// --- Reading the file into an array of strings, one per line ---
Console.WriteLine("\n--- Reading file with ReadAllLines ---");
if (File.Exists(filePath))
{
    string[] fileLines = File.ReadAllLines(filePath);
    Console.WriteLine($"File has {fileLines.Length} lines.");
    // We can now process each line individually.
    foreach (string line in fileLines)
    {
        if (line.Contains("Alice"))
        {
            Console.WriteLine($"Found user entry: '{line.Trim()}'");
        }
    }
}
```

Lecture 11: FileInfo class

Introduction

The **FileInfo** class is an **instance-based** way to interact with a file. Instead of calling static methods on the File class, you first create a FileInfo object that represents a specific file on the disk. You then call instance methods and access properties on that object.

Why and When to Use It

Use a FileInfo object when you need to perform **multiple operations** on the same file or retrieve several pieces of information about it. It is more efficient than using the static File class repeatedly because the operating system security checks are performed only **once** when the FileInfo object is created, not on every single method call.

Key Instance Properties and Methods

- **Properties:**
    - Exists: A boolean indicating if the file exists.
    - Name: The name of the file including the extension (e.g., "MyFile.txt").
    - FullName: The full path to the file.
    - DirectoryName: The full path to the file's containing directory.
    - Length: The size of the file in bytes.
    - Extension: The file's extension (e.g., ".txt").
    - CreationTime, LastAccessTime, LastWriteTime: DateTime objects for file timestamps.
- **Methods:**
    - CopyTo(string destFileName)
    - MoveTo(string destFileName)
    - Delete()
    - OpenRead(), OpenWrite(), Open(): Methods to get a FileStream for the file.

Example

```
string filePath = "MyLog.txt";

// Create a FileInfo object. No disk access happens here yet.
FileInfo fileInfo = new FileInfo(filePath);

// Now, let's check its properties. The first property access will perform the
security check.
if (fileInfo.Exists)
{
    Console.WriteLine($"File Name: {fileInfo.Name}");
    Console.WriteLine($"Full Path: {fileInfo.FullName}");
    Console.WriteLine($"Size: {fileInfo.Length} bytes");
    Console.WriteLine($"Last Modified: {fileInfo.LastWriteTime}");

    // Perform an action
    // fileInfo.CopyTo("MyLog_Backup.txt");
}
else
{
    Console.WriteLine("File does not exist.");
}
```

Lecture 12: Directory class

Introduction

The **Directory** class is the static helper class for working with directories (folders). It provides a simple set of static methods for creating, moving, deleting, and enumerating directories and their contents.

When to Use It

Use the Directory class for simple, single-shot directory operations, just like using the File class for files.

Key Static Methods

- **bool Exists(string path)**: Checks if a directory exists.
- **DirectoryInfo CreateDirectory(string path)**: Creates all directories in the specified path if they don't already exist.
- **void Delete(string path, bool recursive)**: Deletes a directory. The recursive parameter is crucial: if true, it deletes the directory and all its contents (subdirectories and files); if false (the default), it will only delete the directory if it is empty.
- **void Move(string sourceDirName, string destDirName)**: Moves a directory and its contents to a new location.
- **string[] GetFiles(string path)**: Returns an array of strings containing the full paths of the files in the specified directory.
- **string[] GetDirectories(string path)**: Returns an array of strings containing the full paths of the subdirectories.
- **IEnumerable<string> EnumerateFiles(string path)**: A more efficient, LINQ-friendly version of GetFiles. It returns an IEnumerable<string> that allows you to start processing the file paths immediately, without waiting for the entire list to be gathered first. This is preferred for directories with a very large number of files.

Example

```
string newDirPath = @"C:\Temp\MyNewDirectory";
Console.WriteLine($"Checking for directory: {newDirPath}");

if (!Directory.Exists(newDirPath))
{
    Console.WriteLine("Directory does not exist. Creating it...");
    Directory.CreateDirectory(newDirPath);
}

// Now it exists. Let's get files from the current folder to demonstrate.
string[] filesInCurrentFolder = Directory.GetFiles("."); // "." means current directory
Console.WriteLine($"\nFound {filesInCurrentFolder.Length} files here.");

// Clean up
Console.WriteLine($"\nDeleting directory: {newDirPath}");
Directory.Delete(newDirPath);
```

Lecture 13: DirectoryInfo class

Introduction

The **DirectoryInfo** class is the instance-based counterpart to the static Directory class. It represents a specific directory as an object, which you can then query for properties and call methods on.

When to Use It

Use a DirectoryInfo object when you need to perform **multiple operations** or retrieve multiple pieces of information about the same directory. Like FileInfo, it is more efficient in these scenarios because the initial security checks are performed only once when the object is created.

Key Instance Properties and Methods

- **Properties:** Exists, Name, FullName, Parent (returns another DirectoryInfo object), Root.
- **Methods:** Create(), Delete(bool recursive), MoveTo(string destDirName), GetFiles(), GetDirectories(), EnumerateFiles().

Example

```
// Note: The '@' symbol creates a verbatim string literal, so we don't need to escape backslashes.
string projectPath = @"C:\Users\MyUser\Documents\MyProject";

DirectoryInfo dirInfo = new DirectoryInfo(projectPath);

if (dirInfo.Exists)
{
    Console.WriteLine($"Full Name: {dirInfo.FullName}");
    Console.WriteLine($"Parent Directory: {dirInfo.Parent.Name}");

    Console.WriteLine("\n--- C# Files in this directory ---");
```

```
    // GetFiles() on a DirectoryInfo object returns an array of FileInfo objects.
    FileInfo[] csFiles = dirInfo.GetFiles("*.cs"); // Use a search pattern

    foreach (FileInfo file in csFiles)
    {
        Console.WriteLine($"- {file.Name} ({file.Length} bytes)");
    }
}
```

Lecture 14: DriveInfo class

Introduction

The **DriveInfo** class is a simple utility class used to retrieve information about the logical drives on a computer (e.g., C:, D:, etc.).

How It Works

You can get a list of all drives on the system using the static method DriveInfo.GetDrives(). Alternatively, you can get information about a specific drive by creating an instance and passing the drive letter to its constructor: DriveInfo cDrive = new DriveInfo("C");.

Key Properties

- **Name**: The drive name, e.g., "C:\".
- **DriveType**: An enum that tells you the type of drive (e.g., Fixed, CDRom, Network, Removable).
- **IsReady**: A boolean that is true if the drive is ready (e.g., a disk is in the CD-ROM drive). You should always check this before accessing other properties.
- **TotalSize**: The total size of the drive in bytes.
- **TotalFreeSpace**: The total amount of free space available on the drive in bytes.
- **AvailableFreeSpace**: The amount of free space available to the **current user** (respecting quotas).
- **VolumeLabel**: The user-defined volume label of the drive.

Example

```
// Get all logical drives on the system.
DriveInfo[] allDrives = DriveInfo.GetDrives();

Console.WriteLine("--- Available Drives ---");
foreach (DriveInfo d in allDrives)
{
    Console.WriteLine($"Drive {d.Name}");
    Console.WriteLine($"  Drive type: {d.DriveType}");

    // Only get more info if the drive is ready.
    if (d.IsReady == true)
    {
        Console.WriteLine($"  Volume label: {d.VolumeLabel}");
        long totalSizeGB = d.TotalSize / (1024 * 1024 * 1024);
        long freeSpaceGB = d.TotalFreeSpace / (1024 * 1024 * 1024);
        Console.WriteLine($"  Total size: {totalSizeGB} GB");
        Console.WriteLine($"  Available space: {freeSpaceGB} GB");
    }
}
```

Lecture 15: FileStream class

Introduction

The **FileStream** class is the primary class in the System.IO namespace for **low-level, byte-by-byte reading and writing** to a file. Unlike the static File class which performs single, complete operations, a FileStream gives you direct access to the underlying file handle and represents the file as a **stream of bytes**. This makes it the ideal and most efficient tool for working with very large files or for handling binary files where you need precise control.

How It Works: Creating and Configuring a Stream

You create a FileStream object by providing a file path and, most importantly, specifying how you intend to interact with the file using the FileMode and FileAccess enums.

- **FileMode Enum**: Tells the operating system how to open the file. Common values are:
    - **FileMode.Create**: Creates a new file. If the file already exists, it will be **overwritten**.
    - **FileMode.CreateNew**: Creates a new file. If the file already exists, it throws an exception.
    - **FileMode.Open**: Opens an existing file. Throws an exception if the file does not exist.
    - **FileMode.OpenOrCreate**: Opens a file if it exists; otherwise, creates a new one.
    - **FileMode.Append**: Opens an existing file and places the cursor at the end, ready for writing.
- **FileAccess Enum**: Tells the operating system what you plan to do with the file.
    - **FileAccess.Read**: You can only read from the stream.
    - **FileAccess.Write**: You can only write to the stream.
    - **FileAccess.ReadWrite**: You can do both.

Important: IDisposable

A FileStream object holds onto a scarce, unmanaged operating system resource (a file handle). Therefore, it implements the **IDisposable** interface. It is **critical** that you always create and use a FileStream within a **using** statement or declaration to ensure the file handle is properly closed and released, even if errors occur.

Lecture 16: Read/Write operations using FileStream class

Introduction

While you can read and write raw bytes directly with a FileStream, it's often cumbersome. A more common pattern is to wrap the FileStream with a higher-level helper class that is designed to work with streams, such as a **StreamWriter** or **StreamReader** for text.

How It Works: Stream Helpers

These helper classes take a Stream object in their constructor. They don't open the file themselves; they simply provide a more convenient API for interacting with the underlying stream provided to them. The using block should still manage the lifetime of the FileStream, and the helper will be disposed along with it.

Example: Writing and Reading a Text File Line-by-Line

This example demonstrates the robust way to handle text files of any size.

```
string filePath = "my_app_data.log";

// --- Writing with StreamWriter ---
// The using declaration ensures fs and writer are closed/disposed properly.
using var fsWrite = new FileStream(filePath, FileMode.Create, FileAccess.Write);
using var writer = new StreamWriter(fsWrite);

Console.WriteLine("Writing data to file...");
writer.WriteLine("Log Entry 1: Application Start");
writer.WriteLine("Log Entry 2: User 'Alice' logged in.");
writer.WriteLine("Log Entry 3: Data processed.");
// The writer may buffer data; it gets flushed to the file when disposed.


// --- Reading with StreamReader ---
Console.WriteLine("\nReading data from file...");
using var fsRead = new FileStream(filePath, FileMode.Open, FileAccess.Read);
using var reader = new StreamReader(fsRead);
```

```
string line;
// The ReadLine() method returns null when it reaches the end of the file.
// This is the standard pattern for reading a text file to the end.
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
```

Lecture 17: BinaryWriter and BinaryReader

Introduction

BinaryWriter and **BinaryReader** are helper classes used to read and write **primitive data types** (like int, double, bool, string, etc.) to a stream as **binary data**. Unlike StreamWriter, which writes "123" as three text characters, BinaryWriter writes the four raw bytes that represent the integer 123.

When to Use Them

Use these classes when you want to create custom, compact binary file formats. This approach is much more efficient in terms of file size and parsing speed compared to text-based formats like XML or JSON. However, the major tradeoff is that the resulting file is not human-readable.

How It Works

You wrap a FileStream with a BinaryWriter or BinaryReader. It is absolutely critical that you **read the data back in the exact same order and with the exact same data types** as you used to write it. The binary format contains no metadata about the types; it's just a sequence of bytes that you must interpret correctly.

Example

```
string filePath = "player.dat";
string playerName = "John";
int level = 10;
double health = 95.5;
```

```
// --- Writing binary data ---
using (var fs = new FileStream(filePath, FileMode.Create))
using (var writer = new BinaryWriter(fs))
{
    Console.WriteLine("Writing binary data...");
    writer.Write(playerName);
    writer.Write(level);
    writer.Write(health);
}

// --- Reading binary data ---
using (var fs = new FileStream(filePath, FileMode.Open))
using (var reader = new BinaryReader(fs))
{
    Console.WriteLine("\nReading binary data...");
    // Must read back in the exact same order and type!
    string readName = reader.ReadString();
    int readLevel = reader.ReadInt32();
    double readHealth = reader.ReadDouble();

    Console.WriteLine($"Name: {readName}, Level: {readLevel}, Health:
{readHealth}");
}
```

Lecture 18: Intro to Serialization

Introduction

Serialization is the process of converting an object's state (the values of its fields and properties) into a format that can be easily stored or transmitted.
Deserialization is the reverse process: taking that stored data and using it to reconstruct the original object in memory.

Why It's Used

Serialization is a cornerstone of modern applications and is used everywhere:

- **Persistence:** Saving an object's state to a file on disk so it can be reloaded when the application starts again (e.g., saving game progress, application settings).

- **Communication:** Sending a complex data object across a network to another service via a web API.
- **Caching:** Storing a frequently used, expensive-to-create object in a cache (like Redis) for faster retrieval later.

The most common serialization formats today are **JSON** and **XML**, because they are text-based and human-readable. A less common format is **binary serialization**.

Lecture 19: Binary Serialization

Introduction

Binary serialization is the process of converting an object's state into a compact, non-human-readable stream of bytes. This format is typically smaller and faster to parse than text-based formats like JSON or XML. In the legacy .NET Framework, this was done using a class called the BinaryFormatter.

How It Worked: The [Serializable] Attribute

To make a class serializable with this older method, you had to mark the class with the [Serializable] attribute. This signaled to the runtime that it was allowed to access the class's fields (even private ones) to read their values during serialization and set them during deserialization.

Important Note: Modern Status and Security Warning

This is a critical point for any modern C# developer. The BinaryFormatter is now considered **obsolete and insecure**. Its use is **highly discouraged** by Microsoft and is disabled by default in new ASP.NET Core and .NET 5+ applications.

Why is it insecure? A malicious actor could craft a binary payload that, when deserialized by BinaryFormatter, could execute arbitrary code on the server, leading to a serious security vulnerability.

Interview Perspective

- **Question:** "When would you use binary serialization?"
- **Ideal Answer:** "In modern development, you generally shouldn't. The standard `BinaryFormatter` has known security vulnerabilities and is obsolete. For most use cases, the modern standard is JSON serialization using a library like `System.Text.Json` or `Newtonsoft.Json`. If a compact binary format is absolutely required for performance, a safer, contract-based binary serializer like protobuf-net would be the appropriate choice over the old `BinaryFormatter`."

Lecture 20: Json Serialization

Introduction

JSON (JavaScript Object Notation) is the modern, de facto standard for data interchange on the web and in software applications. It is a lightweight, **human-readable** text format that represents data as a collection of key-value pairs (like a dictionary) and ordered lists (like an array).

In modern .NET, the built-in, high-performance library for handling JSON is **`System.Text.Json`**.

How It Works

The static `JsonSerializer` class provides two primary methods:

- **`JsonSerializer.Serialize(object)`**: This method takes an object instance and returns its JSON string representation.
- **`JsonSerializer.Deserialize<T>(string)`**: This generic method takes a JSON string and reconstructs it into a new object of the specified type `T`.

The serializer automatically maps the properties of your C# object to the keys in the JSON text.

When to Use It

This is the default choice for almost all modern serialization tasks:

- Creating and consuming web APIs (REST APIs).
- Writing and reading configuration files.
- Storing simple object data in text files or databases.

Example

```
public class User
{
    public int Id { get; set; }
    public string Username { get; set; }
    public bool IsActive { get; set; }
}

// --- In your Main method ---
User user = new User { Id = 101, Username = "alice", IsActive = true };

// 1. Serialize the object to a JSON string
string jsonString = JsonSerializer.Serialize(user, new JsonSerializerOptions
{ WriteIndented = true });
Console.WriteLine("--- Serialized JSON ---");
Console.WriteLine(jsonString);

// 2. Deserialize the JSON string back into a new object
User newUser = JsonSerializer.Deserialize<User>(jsonString);
Console.WriteLine("\n--- Deserialized Object ---");
Console.WriteLine($"ID: {newUser.Id}, Username: {newUser.Username}");
```

Output:

```
--- Serialized JSON ---
{
  "Id": 101,
  "Username": "alice",
  "IsActive": true
}

--- Deserialized Object ---
ID: 101, Username: alice
```

Lecture 21: Xml Serialization

Introduction

XML (eXtensible Markup Language) is another text-based format for representing data. It uses tags enclosed in angle brackets to create a hierarchical, tree-like structure. While JSON has become more popular for web APIs due to its simplicity, XML is still heavily used in many enterprise environments, configuration files (like `.csproj`), and for standards like SOAP web services.

How It Works: The `XmlSerializer`

In C#, you work with XML serialization primarily using the `System.Xml.Serialization.XmlSerializer` class. Unlike the JSON serializer, it has some specific constraints:

- The class being serialized **must have a public, parameterless constructor**.
- It only serializes **public fields and public properties**.

When to Use It

- When interacting with legacy enterprise systems or web services that require XML.
- For complex configuration files where XML's structure, support for comments, and schemas are beneficial.
- When working with standards that are built on XML, like SAML for authentication.

Example

Let's serialize the same User object to XML to see the difference.

```
public class User // Must have a public parameterless constructor for
XmlSerializer
{
    public int Id { get; set; }
    public string Username { get; set; }
    public bool IsActive { get; set; }
}

// --- In your Main method ---
User user = new User { Id = 101, Username = "alice", IsActive = true };
XmlSerializer serializer = new XmlSerializer(typeof(User));

// To write to the console, we can use a StringWriter.
using (var writer = new StringWriter())
{
    serializer.Serialize(writer, user);
    string xmlString = writer.ToString();

    Console.WriteLine("--- Serialized XML ---");
    Console.WriteLine(xmlString);
}
```

Output:

```
--- Serialized XML ---
<?xml version="1.0" encoding="utf-16"?>
<User xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Id>101</Id>
  <Username>alice</Username>
  <IsActive>true</IsActive>
</User>
```

As you can see, XML is significantly more verbose than JSON for the same data.

Section 28: Exception Handling

Lecture 1: Intro to Exception Handling

Introduction

An **exception** is an error condition or an unexpected event that occurs during the execution of a program, disrupting its normal, top-to-bottom flow. These are not syntax errors that the compiler can catch before you run the code; they are **run-time errors**. Examples include trying to divide by zero, accessing a file that doesn't exist, or connecting to a database that is offline.

Exception handling is the process of responding to these run-time errors in a structured and controlled way. Without it, any exception would immediately terminate your application, resulting in a crash, a poor user experience, and potential data corruption. A robust exception handling strategy allows you to gracefully manage errors, log them for debugging, inform the user of the problem, and allow the application to continue running if possible.

How It Works: The Exception Object

When an error occurs, the .NET runtime creates an **exception object** that contains detailed information about the error, such as its type (e.g., `FileNotFoundException`), a descriptive message, and a **stack trace** indicating where in the code the error happened. The runtime then "throws" this object, halting the current execution path and looking for a compatible error handler to "catch" it.

Lecture 2: Try-Catch-Finally

Introduction

The try-catch-finally block is the fundamental C# construct for handling exceptions. It allows you to define a block of "protected" code and specify how to handle errors that may occur within it.

How It Works: The Three Blocks

1. **The try Block:**
    - This is where you place the code that might throw an exception. Execution starts here. If no exception occurs, the entire try block completes, and execution skips over the catch block(s).
2. **The catch Block:**
    - This is the **error handler**. If an exception is thrown inside the try block, the normal flow stops immediately, and the runtime looks for a matching catch block.
    - A catch block specifies the type of exception it can handle (e.g., catch (FormatException ex)). You can have multiple catch blocks to handle different types of exceptions.
    - The ex variable holds the exception object, which you can use to log the error message and other details.
3. **The finally Block:**
    - This block is **always** executed, regardless of what happened in the try and catch blocks. It will run if no exception occurred, if an exception was caught, or even if an exception was thrown but not caught.
    - **Its purpose is cleanup.** This is where you put code that absolutely must run to release resources, such as closing a file or a database connection.

Example

```
public void ProcessUserInput()
{
   Console.WriteLine("Please enter a number:");
   string input = Console.ReadLine();

   try
   {
      Console.WriteLine("--> Entering the 'try' block.");
      int number = int.Parse(input);
      Console.WriteLine($"You entered the valid number: {number}");
   }
   catch (FormatException ex)
   {
      Console.WriteLine("--> Entering the 'catch' block.");
      Console.WriteLine("Error: That was not a valid number.");
      Console.WriteLine($"Exception details: {ex.Message}");
   }
   finally
   {
      Console.WriteLine("--> Entering the 'finally' block. This always runs.");
   }

   Console.WriteLine("Method has finished.");
}
```

Lecture 3: FormatException

Introduction

A **FormatException** is the specific exception thrown when the format of an argument is incorrect for the method it's being passed to.

Common Cause

The most common cause by far is using a `Parse` method on a string that is not in a valid format. For example, `int.Parse("abc")` will throw a `FormatException` because "abc" cannot be converted into an integer. Similarly, `DateTime.Parse("yesterday")` will also fail.

How to Handle It: Prevention with TryParse

While you *can* wrap a Parse call in a try-catch block, this is generally considered poor practice for validation. A thrown exception is a relatively "expensive" operation in terms of performance.

The **best practice** is to prevent the exception from happening in the first place by using the TryParse pattern. Methods like int.TryParse and DateTime.TryParse return a bool indicating success or failure and do not throw an exception on invalid input, which is much more efficient for handling expected validation failures.

Example

```
string userInput = "not-a-number";

// --- Bad Practice: Using try-catch for validation ---
try
{
   int number = int.Parse(userInput);
}
catch (FormatException)
{
   Console.WriteLine("FormatException was caught!");
}

// --- Good Practice: Using TryParse ---
if (int.TryParse(userInput, out int number))
{
   // This won't run
}
else
{
   Console.WriteLine("TryParse gracefully handled the invalid input
without an exception.");
}
```

Lecture 4: IndexOutOfRangeException

Introduction

An **IndexOutOfRangeException** is the exception thrown when you attempt to access an element of an array or a list using an index that is outside its valid bounds.

Common Cause

This is almost always a **logic error** in your code, often called an "off-by-one" error.

- The valid indices for an array of Length N are 0 through N-1.
- A common mistake is using <= in a for loop condition: for (int i = 0; i <= myArray.Length; i++). When i becomes equal to myArray.Length, this will throw the exception. The condition should be i < myArray.Length.

How to Handle It: Fix the Bug

You should **never** catch an IndexOutOfRangeException. Catching it would be like putting a bandage on a deep wound without treating the cause. The presence of this exception indicates a flaw in your program's logic that needs to be found and fixed.

Example

```
int[] numbers = { 10, 20, 30 }; // Length is 3. Valid indices are 0, 1, 2.

try
{
   // This attempts to access the element at index 3, which does not exist.
   int value = numbers[3];
}
catch (IndexOutOfRangeException ex)
{
   Console.WriteLine($"Caught an error: {ex.Message}");
}
```

Lecture 5: NullReferenceException

Introduction

This is one of the most common exceptions in C# and programming in general. A **NullReferenceException** occurs at run-time when you try to access a member (like a method, property, or field) on a variable that is currently null.

Common Cause

It's caused by trying to dereference a "null pointer"—that is, following a reference that doesn't point to any object in memory.

- Forgetting to initialize a reference type variable before using it (new MyClass()).
- A method that was expected to return an object instead returned null, and the calling code did not check for it.

How to Handle It: Prevent It

Like IndexOutOfRangeException, this usually indicates a logic error. You should **not** catch this exception. The best practice is to **prevent it** by writing defensive code:

- Always check for null before using an object that might be null.
- Use the null-conditional operator (?.) and null coalescing operator (??) to handle potential nulls safely and concisely.

Example

```
public class Person { public string Name { get; set; } }

// This variable holds a null reference. It doesn't point to a Person object.
Person person = null;

try
{
   // This will crash because you can't get the 'Name' from nothing.
   string name = person.Name;
```

```
}
catch (NullReferenceException ex)
{
    Console.WriteLine($"Caught an error: {ex.Message}");
    Console.WriteLine("This happened because the 'person' variable was
null.");
}
```

Lecture 6: OverflowException

Introduction

An **OverflowException** is the error that occurs when an arithmetic operation produces a result that is outside the allowable range of the destination data type. For example, the `int` data type has a maximum value of 2,147,483,647. If you try to add 1 to a variable holding this value, you will cause an overflow.

How It Works: checked and unchecked Contexts

This is a critical interview topic. By default, for performance reasons, integer arithmetic in C# is **unchecked**. This means if an overflow occurs, the value will simply "wrap around" (becoming a large negative number in the case of `int.MaxValue + 1`) **silently, without throwing an exception**.

To enable overflow checking, you must explicitly create a `checked` context.

- **The `checked` keyword:** You can wrap a block of code in `checked { ... }`. If an overflow occurs inside this block, an `OverflowException` will be thrown.
- **Compiler Flag:** You can also enable overflow checking for an entire project in the project's build settings.

When to Use It

You should use a `checked` block in any critical financial or mathematical calculation where a silent overflow would lead to incorrect data and serious bugs.

Example

```
int a = int.MaxValue; // 2,147,483,647

// --- Unchecked context (default) ---
int resultUnchecked = a + 1;
// The value wraps around to the smallest possible integer.
Console.WriteLine($"Unchecked result: {resultUnchecked}"); // Output:
-2147483648

// --- Checked context ---
try
{
   // This block will now throw an exception.
   checked
   {
      int resultChecked = a + 1;
   }
}
catch (OverflowException ex)
{
   Console.WriteLine($"Caught an expected OverflowException!");
}
```

Lecture 7: InvalidCastException

Introduction

An **InvalidCastException** is the exception thrown for an explicit cast or conversion that is not valid. This happens when you try to force a variable of one type into another, incompatible type at run-time.

Common Causes

1. **Unboxing to the Wrong Type:** This is a classic cause. If you box an `int` into an `object`, you can only unbox it back to an `int` (or a nullable `int`). Trying to unbox it directly to a `long` or `double` will fail, even though the numeric conversion would be possible.

2. **Casting Between Incompatible Reference Types:** If you have an object of type Dog, you can safely cast it to its base type Animal. However, you cannot cast that Dog object to an unrelated type like Car.

How to Handle It: Prevention with is and as

Like other logic errors, the best practice is to prevent this exception. Instead of a risky explicit cast, you should use the is or as operators.

- **The is operator:** Returns true if an object can be safely cast to a specific type, allowing you to proceed with the cast inside an if block. if (myObject is Dog d) { ... }
- **The as operator:** Attempts to cast an object. If successful, it returns the casted object. If the cast is invalid, it **returns null instead of throwing an exception**. Dog myDog = myObject as Dog;

Example

```
object boxedInt = 123; // This is a boxed int.

try
{
   // This will fail. You cannot unbox an int directly to a long.
   long myLong = (long)boxedInt;
}
catch (InvalidCastException)
{
   Console.WriteLine("Failed to unbox an int directly to a long.");
}

// The correct way: unbox first, then cast.
long myCorrectLong = (int)boxedInt;
Console.WriteLine($"Correct conversion: {myCorrectLong}");
```

Lecture 8: InvalidOperationException

Introduction

This is a more general exception that is thrown when a method call is invalid or inappropriate given the object's **current state**. The operation itself is valid, but the object is not in a condition to perform it successfully.

Common Causes from the .NET Framework

- Trying to call .Value on a nullable type when its .HasValue property is false.
- Trying to call Pop() or Peek() on an empty Stack.
- Trying to call Dequeue() or Peek() on an empty Queue.
- Trying to read from a database connection that has been closed.

How to Handle It

This almost always indicates a logic error. The solution is to **check the object's state before calling the method**. You shouldn't catch this exception; you should prevent it.

Example

```
Stack<int> myStack = new Stack<int>();

// myStack is currently empty.

try
{
   // This is an invalid operation on an empty stack.
   myStack.Pop();
}
catch (InvalidOperationException)
{
   Console.WriteLine("Caught InvalidOperationException from popping an empty stack.");
}
```

```
// The correct, defensive way to write this code:
if (myStack.Count > 0)
{
    myStack.Pop();
}
else
{
    Console.WriteLine("Cannot pop, the stack is empty.");
}
```

Lecture 9: ArgumentNullException

Introduction

ArgumentNullException is a specific type of argument exception that should be thrown by a method when one of its parameters is null, but the method requires a non-null value to function correctly.

How It Works

It is a subclass of ArgumentException. Its purpose is to provide a very clear error message indicating *which* parameter was null, which is extremely helpful for debugging.

When to Use It (Best Practice)

As a developer writing a method that will be used by others, you should perform **guard clause** checks at the very beginning of your method. If a critical parameter is null, you should immediately throw new ArgumentNullException() to fail fast and provide a clear error.

Using nameof: It is a best practice to use the nameof operator to specify the parameter name. This avoids using "magic strings" and ensures that if you rename the parameter later, the error message will update automatically.

Example

```
public void ProcessUser(User user)
{
   // This is a guard clause.
   if (user == null)
   {
      // Fail fast with a specific error.
      throw new ArgumentNullException(nameof(user));
   }

   Console.WriteLine($"Processing user: {user.Name}");
}
```

Lecture 10: InnerException

Introduction

The InnerException property, which exists on all exception objects, is a crucial mechanism for preserving the original error when you catch a low-level exception and then throw a new, more specific, higher-level exception.

Why It's Important

It creates a "chain" or "stack" of exceptions. This is vital for debugging because it allows you to see the **full story** of what went wrong. Without preserving the inner exception, the original root cause of the problem is lost forever, making it much harder to diagnose the issue.

How It Works

When you catch a low-level exception (e.g., a SqlException) and decide to throw a new, more meaningful exception (e.g., a UserUpdateException), you should **pass the original exception** into the constructor of the new exception.

Syntax: throw new MyCustomException("Could not update user.", ex);

Example

```
try
{
   // This line will throw a FormatException.
   int.Parse("abc");
}
catch (FormatException ex) // 'ex' holds the original error.
{
   // We catch the specific, low-level error...
   // ...and throw a new, more meaningful application-level exception.
   // We pass the original exception 'ex' as the InnerException.
   throw new ApplicationException("Error processing input data.", ex);
}
```

When you inspect the ApplicationException, its InnerException property will contain the original FormatException, complete with its own message and stack trace.

Lecture 11: ArgumentOutOfRangeException

Introduction

ArgumentOutOfRangeException is the exception that should be thrown when the **value** of an argument is outside the allowable range of values for a method's parameter.

Common Causes

- Passing a negative number to a method that only accepts non-negative numbers (e.g., Take(-5) from a list).
- Passing an index to a Substring method that is larger than the length of the string.

ArgumentNullException vs. ArgumentOutOfRangeException

This is a key distinction.

- Use ArgumentNullException when the argument reference itself **is null**.
- Use ArgumentOutOfRangeException when the argument is not null, but its **value** is invalid.

Example

```
public string GetMonthName(int month)
{
   if (month < 1 || month > 12)
   {
      // The value of 'month' is outside the valid range of 1-12.
      throw new ArgumentOutOfRangeException(nameof(month), "Month
must be between 1 and 12.");
   }
   // ... logic to return month name ...
   return "A month";
}
```

Lecture 12: ArgumentException

Introduction

ArgumentException is the base class for all argument-related exceptions. You should throw this exception when an argument to a method is invalid, but none of the more specific exceptions (ArgumentNullException, ArgumentOutOfRangeException) are appropriate.

When to Use It

It's a general-purpose "invalid argument" error. A common use case is when a string parameter is required but an empty string is passed. It's not null (ArgumentNullException), and it's not out of a numeric range (ArgumentOutOfRangeException), so the general ArgumentException is the best fit.

Example

```
public void SetUsername(string username)
{
   // A username can't be null.
   if (username == null)
   {
      throw new ArgumentNullException(nameof(username));
   }

   // A username also can't be empty or just whitespace.
   if (string.IsNullOrWhiteSpace(username))
   {
      // ArgumentException is perfect for this kind of validation failure.
      throw new ArgumentException("Username cannot be empty or
whitespace.", nameof(username));
   }

   // ... set the username ...
}
```

Lecture 13: Custom Exceptions

Introduction

While the .NET framework provides a rich hierarchy of exception types, it is a crucial best practice to create your own **custom exception classes** for errors that are specific to your application's domain logic. Throwing a `ProductOutOfStockException` is far more meaningful and clear than throwing a generic `InvalidOperationException`.

Why and When to Use It

1. **Clarity and Specificity:** Custom exceptions allow you to create highly specific error handlers. A `catch` block that catches a `DuplicateUsernameException` knows exactly what went wrong and can respond appropriately, perhaps by showing a targeted error message to the user.

2. **Carrying Extra Data:** A custom exception class can have its own properties to carry extra, context-rich information about the error. For example, a ProductOutOfStockException could carry the ProductId of the item that was out of stock.

How to Implement (Best Practices)

Follow this standard pattern when creating a custom exception:

1. Create a new class that inherits from System.Exception (or a more specific base exception, like ArgumentException).
2. By convention, the class name should end with the word "Exception".
3. Provide the three standard constructors that allow for setting a message and an inner exception, which ensures your custom exception works well with the rest of the .NET ecosystem.

Example

```
// 1. Create the custom exception class.
public class InsufficientBalanceException : Exception
{
    public decimal AmountShort { get; } // A custom property

    // 2. Provide the three standard constructors.
    public InsufficientBalanceException() { }

    public InsufficientBalanceException(string message) : base(message) { }

    public InsufficientBalanceException(string message, Exception
innerException) : base(message, innerException) { }

    // 3. Add a custom constructor to carry extra data.
    public InsufficientBalanceException(string message, decimal
amountShort) : base(message)
    {
        this.AmountShort = amountShort;
    }
}

public class BankAccount
{
```

```
    public decimal Balance { get; private set; } = 100;

    public void Withdraw(decimal amount)
    {
        if (amount > Balance)
        {
            // Throw our new, specific exception with context.
            decimal shortBy = amount - Balance;
            throw new InsufficientBalanceException("Withdrawal amount
exceeds balance.", shortBy);
        }
        Balance -= amount;
    }
}
```

Lecture 14: Stack Trace

Introduction

The **stack trace** is a snapshot of the **call stack** at the moment an exception was thrown. It is arguably the **single most important piece of information for debugging** an error. It provides a "breadcrumb trail" or a "chain of evidence" that shows the exact sequence of method calls that led to the exception, allowing a developer to pinpoint the source of the problem.

How Is It Maintained Internally?

To understand the stack trace, you must understand the **call stack**. The call stack is a region of memory that keeps track of the active methods in your program. It operates on a Last-In, First-Out (LIFO) principle.

1. When your program starts, a "stack frame" for your `Main` method is pushed onto the stack.
2. If `Main` calls `MethodA`, a new stack frame for `MethodA` is pushed on top of `Main`'s frame.
3. If `MethodA` calls `MethodB`, a frame for `MethodB` is pushed on top of `MethodA`'s frame.
4. When `MethodB` finishes, its frame is popped off, and control returns to `MethodA`.

When an exception is thrown in MethodB, the runtime halts execution and captures the current state of this call stack. It then formats this information into a human-readable string, including the namespace, class name, method name, and often the file name and line number for each frame. This resulting string is the stack trace.

Example

```
public class Program
{
   public static void Main(string[] args)
   {
      try
      {
         MethodA();
      }
      catch (Exception ex)
      {
         Console.WriteLine("--- EXCEPTION CAUGHT ---");
         Console.WriteLine($"Error Message: {ex.Message}");
         Console.WriteLine("\n--- STACK TRACE ---");
         Console.WriteLine(ex.StackTrace);
      }
   }

   public static void MethodA()
   {
      Console.WriteLine("Inside MethodA, about to call MethodB.");
      MethodB();
   }

   public static void MethodB()
   {
      Console.WriteLine("Inside MethodB, about to call MethodC.");
      MethodC();
   }

   public static void MethodC()
   {
      Console.WriteLine("Inside MethodC, about to throw an exception.");
      throw new InvalidOperationException("Something went wrong in
Method C!");
   }
}
```

The output stack trace would look like this (simplified):

```
--- STACK TRACE ---
   at MyProject.Program.MethodC() in C:\MyApp\Program.cs:line 35
   at MyProject.Program.MethodB() in C:\MyApp\Program.cs:line 29
   at MyProject.Program.MethodA() in C:\MyApp\Program.cs:line 23
   at MyProject.Program.Main(String[] args) in C:\MyApp\Program.cs:line 11
```

This clearly shows the "breadcrumb trail": Main called A, which called B, which called C, where the error occurred.

Lecture 15: Logging Exceptions

Introduction

Simply catching an exception is not enough. If you catch an error and do nothing with it (an "empty catch block"), you have effectively hidden a problem. This is a dangerous practice. **Logging** the details of an exception is critical for monitoring application health, diagnosing issues in a production environment, and understanding why and where errors are occurring.

What to Log

When you log an exception, you should always capture as much information as possible. At a minimum:

- The **Timestamp** of when the error occurred (DateTime.UtcNow).
- The **Type** of the exception (e.g., System.IO.FileNotFoundException).
- The exception's **Message** property.
- The full **StackTrace**.
- **Crucially, the InnerException**. If an inner exception exists, you must log its details as well, and recursively check if *it* has an inner exception, all the way down the chain. This ensures you capture the root cause.

Example

This example shows a simple helper class to demonstrate the principle of logging all important details. In a real application, you would use a robust, third-party logging library.

```
public static class Logger
{
    public static void LogException(Exception ex)
    {
        Console.WriteLine("--- ERROR LOGGED ---");
        Console.WriteLine($"Timestamp: {DateTime.UtcNow}");

        // Use a loop to log the entire exception chain.
        Exception currentEx = ex;
        int level = 0;
        while (currentEx != null)
        {
            Console.WriteLine($"\nLevel {level}:");
            Console.WriteLine($"  Type: {currentEx.GetType().FullName}");
            Console.WriteLine($"  Message: {currentEx.Message}");
            Console.WriteLine($"  StackTrace:\n{currentEx.StackTrace}");

            currentEx = currentEx.InnerException;
            level++;
        }
        Console.WriteLine("--- END OF LOG ---");
    }
}

// --- In a try-catch block ---
// catch (Exception ex)
// {
//     Logger.LogException(ex);
// }
```

Lecture 16: System.Exception

Introduction

System.Exception is the base class for **all** exceptions in the .NET framework. Every specific exception, like FormatException, ArgumentNullException, and even your own custom exceptions, ultimately inherits from this class.

How It Works

Because of inheritance, a catch (Exception ex) block is a general-purpose, **"catch-all"** handler. It will catch any possible exception that could be thrown in your try block.

When to Use It (Best Practices)

Catching the generic Exception type should be done with caution and is generally only appropriate in specific places.

- **Good Use:** At the highest level of your application (e.g., in Program.Main or a global error handler in a web app). Here, it acts as a **final safety net** to log any totally unexpected error and prevent the entire application from crashing.
- **Bad Use:** In lower-level business logic. In these areas, you should try to catch **more specific** exception types whenever possible. For example, if you are working with files, you should specifically catch (FileNotFoundException ex) or catch (IOException ex). Catching the generic Exception can hide bugs by accidentally catching and "handling" errors you weren't expecting and didn't actually know how to deal with properly.

Best Practice: Always order your catch blocks from **most specific to most general**.

Example

```
try
{
   // ... code that could throw different errors ...
}
catch (ArgumentNullException ex) // Most specific
{
   // Handle the case of a null argument.
}
catch (ArgumentException ex) // More general argument error
{
   // Handle other invalid argument cases.
}
catch (Exception ex) // The final catch-all safety net
{
   // Log any other unexpected error.
}
```

Lecture 17: Catch When / Exception Filters

Introduction

Exception filters, introduced in C# 6.0, allow you to add a conditional check to a catch block using the **when** keyword. This provides a more powerful and precise way to decide whether to handle an exception.

How It Works

A `catch` block with a `when` clause will only execute if both the exception type matches **and** the boolean condition in the `when` clause evaluates to `true`.

Syntax: `catch (ExceptionType ex) when (condition)`

Why It's Better Than `if` Inside `catch`

This is the key insight and a great interview point.

- If you use an `if` statement inside a `catch` block, the exception is considered **handled** the moment execution enters the `catch` block. The

stack unwinding stops. If your `if` condition is false, the exception is effectively "swallowed" unless you manually re-throw it.

- With `when`, the condition is evaluated **before** the stack unwinding stops. If the `when` condition is `false`, the runtime continues searching up the call stack for another suitable `catch` block. This is better for logging and debugging because you don't "handle" an exception you aren't prepared for, allowing a more appropriate handler further up the stack to deal with it.

Example

Let's say we only want to handle a specific type of web error, like a timeout, but let other web errors propagate.

```
try
{
   // ... code that makes a web request ...
   // For this example, we'll just throw it manually.
   throw new WebException("The request timed out.",
WebExceptionStatus.Timeout);
}
// This catch block will ONLY execute if the exception is a WebException
AND its status is Timeout.
catch (WebException ex) when (ex.Status ==
WebExceptionStatus.Timeout)
{
   Console.WriteLine("Caught a timeout error specifically. Retrying...");
}
// Any other WebException (like NameResolutionFailure) would NOT be
caught here.
catch (WebException ex)
{
   Console.WriteLine($"Caught a different web error: {ex.Status}");
}
```

Lecture 18: Things to Remember

- **Use Exceptions for Exceptional Circumstances:** Exceptions should be for genuine run-time errors (file not found, network down), not for normal program flow control. Using exceptions to break out of a loop, for instance, is a major anti-pattern.
- **Prevent Exceptions When Possible:** Defensive coding is better than reactive coding. Use `int.TryParse` instead of catching a `FormatException`. Check for `null` instead of catching a `NullReferenceException`.
- **Catch Specific Exceptions First:** Always order your `catch` blocks from the most specific type to the most general (`Exception`) type.
- **Don't "Swallow" Exceptions:** An empty `catch` block (`catch { }`) is dangerous. It hides problems. At a minimum, always log the exception you catch so you have a record that something went wrong.
- **Preserve the Stack Trace:** This is a critical interview topic. When you catch an exception (`ex`) and need to re-throw it, you must use **`throw;`** by itself.
    - `throw;`: Preserves the original stack trace, so you know where the error truly originated.
    - `throw ex;`: This **resets** the stack trace to the current line, effectively destroying the original point of failure. You should almost never do this.
- **Clean Up with `finally`:** Use the `finally` block (or the `using` statement for `IDisposable` objects) to guarantee that critical cleanup code (like releasing file handles or database connections) is always executed, whether an exception occurred or not.

Section 30: C# 9 and C# 10 - New Features

Lecture 1: Top-Level Statements

Introduction

Top-level statements, introduced in C# 9, are a major feature designed to simplify the creation of simple programs and make C# more approachable for beginners. This feature allows you to write your main program's executable code directly in a `Program.cs` file, eliminating the need for the traditional boilerplate ceremony of a `Program` class and a `static void Main` method.

How It Works: Compiler Generation

This feature is powerful syntactic sugar. When you write code using top-level statements, you are not changing how .NET programs fundamentally work. Behind the scenes, the C# compiler takes your statements and generates the familiar `Program` class and `Main` method for you. You only write the code that would have gone inside the `Main` method.

Rules

- You can only use top-level statements in **one file** per project.
- That file becomes the entry point for your application.
- You can still define methods and classes in the same file, but they must appear after the top-level statements.

Example: Before and After

Before (Traditional C#):

```
using System;

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

After (Modern C# with Top-Level Statements):

```
using System;

Console.WriteLine("Hello, World!");

// You can access command line args via the magic 'args' variable.
Console.WriteLine($"There are {args.Length} arguments.");
```

Interview Perspective

- **Question:** "What are top-level statements?"
- **Ideal Answer:** "Top-level statements are a C# 9 feature that allows you to write the entry point of a console application without the traditional `Program` class and `Main` method boilerplate. It's syntactic sugar where the compiler generates that class structure for you, making simple programs and scripts much cleaner and easier to write."

Lecture 2: File-Scoped Namespaces

Introduction

File-scoped namespaces, introduced in C# 10, are a stylistic improvement that allows you to declare a namespace for an entire file without needing to wrap your code in curly braces {}.

How It Works

You declare the namespace at the very top of the file, ending the line with a semicolon. Everything in the file below this declaration is considered part of that namespace.

Syntax: namespace MyProject.MyFeature;

Why and When to Use It

The sole purpose is to **reduce horizontal nesting and boilerplate code**. This makes your code files cleaner, less indented, and slightly easier to read. It is the recommended style for all new C# 10+ projects.

Example: Before and After

Before (Block-Scoped Namespace):

```
namespace MyProject
{
    public class MyClass
    {
        public void MyMethod()
        {
            // Code is indented one level deep
        }
    }
}
```

After (File-Scoped Namespace):

```
namespace MyProject; // No opening brace

public class MyClass
{
    public void MyMethod()
    {
        // Code is not indented, leading to cleaner files.
    }
}
// No closing brace
```

Lecture 3: Global Using

Introduction

The global using directive, introduced in C# 10, is a powerful feature that allows you to declare a using directive in **one place** and have it apply to your **entire project**.

How It Works

You prefix a standard using directive with the global keyword. When the compiler sees this, it treats that using statement as if it were present at the top of every single .cs file in the project.

Best Practice: It is common practice to create a single, dedicated file (e.g., GlobalUsings.cs) to hold all of your global using directives for better organization.

Why and When to Use It

Its purpose is to **significantly reduce boilerplate code**. In any given project, you often repeat the same using directives (using System;, using System.Linq;, using System.Collections.Generic;) in dozens or hundreds of files. global using eliminates this repetition completely.

Example

Step 1: Create a GlobalUsings.cs file:

```
// This using statement now applies to every file in the project.
global using System.Linq;
global using System.Collections.Generic;
```

Step 2: Write your class files without repeating the directives.

File: ProductService.cs

```
// No need for 'using System.Linq;' or 'using System.Collections.Generic;'
here.
public class ProductService
{
    public List<string> GetProductNames(List<Product> products)
    {
        // We can directly use List<T> and LINQ's .Select() method.
        return products.Select(p => p.Name).ToList();
    }
}
```

File: UserService.cs

```
// No need for 'using System.Linq;' or 'using System.Collections.Generic;'
here either.
public class UserService
{
    public int CountActiveUsers(List<User> users)
    {
        // We can directly use List<T> and LINQ's .Count() method.
        return users.Count(u => u.IsActive);
    }
}
```

Lecture 4: Module Initializers

Introduction

A **module initializer** is an advanced C# 9 feature that allows you to write code that will run **once** when an entire assembly (a module, e.g., a .dll or .exe) is loaded by the .NET runtime.

How It Works

You create a method and mark it with the [ModuleInitializer] attribute. This method must adhere to three rules:

1. It must be static.
2. It must be parameterless.
3. It must return void.

The runtime guarantees that this method will be called before any other code in that assembly is accessed—before any field initializers run and before any static constructors are called.

Why and When to Use It

This is a low-level feature for library authors or for scenarios requiring early initialization that isn't tied to a specific class.

- **Pre-caching data:** A library could start pre-caching required data as soon as it's loaded.
- **Complex Setup:** Running one-time setup logic, like registering components with a dependency injection framework or setting up interop with native libraries.

Example

```
using System.Runtime.CompilerServices;

class MyLibrary
{
    // This method will run automatically when the assembly is loaded.
    [ModuleInitializer]
    internal static void Initialize()
    {
        Console.WriteLine("MyLibrary module is initializing...");
        // Perform one-time setup here.
    }
}
```

Lecture 5: Nullable Reference Types

Introduction

This is a major language feature (introduced in C# 8, but central to modern C#) that redefines how reference types work in order to help developers prevent NullReferenceException errors at compile time.

How It Works: Flipping the Default Assumption

When you enable this feature in your project file (<Nullable>enable</Nullable>), the compiler's default assumption about reference types flips.

- **Non-Nullable by Default (string)** A standard reference type like string is now considered **non-nullable**. The compiler expects it to always hold a valid object reference. It will issue a warning if you:
    - Try to assign null to it.
    - Don't initialize it to a non-null value before use.
- **Explicitly Nullable (string?)** To declare a reference type that is **allowed** to be null, you must explicitly use the nullable annotation ?. When you do this, the compiler then enforces a new rule: it will warn you if you try to use this variable **without first checking if it is null**.

Why It's Important

This feature moves null-related errors from unpredictable **run-time crashes** to predictable **compile-time warnings**. It forces you to be explicit and conscious about where null is allowed in your program, making your code dramatically safer and more robust.

Example

```
// In a project with <Nullable>enable</Nullable>

public void PrintName(string firstName, string? middleName, string
lastName)
{
    // The compiler trusts that 'firstName' and 'lastName' are not null.
    Console.WriteLine($"First: {firstName.ToUpper()}");

    // The compiler gives a WARNING on the next line because
'middleName' could be null.
    // Console.WriteLine($"Middle initial: {middleName.First()}");

    // This is the correct, safe way to handle the nullable type.
    if (middleName != null)
    {
        Console.WriteLine($"Middle initial: {middleName.First()}");
    }

    Console.WriteLine($"Last: {lastName.ToUpper()}");
}
```

Lecture 6: Null-Forgiving Operator

Introduction

The **null-forgiving operator** (also known as the null-suppression operator) is the exclamation mark (!). It is a special operator used only when you have enabled **Nullable Reference Types**. Its purpose is to tell the C# compiler to **suppress** a possible "may be null" warning.

How It Works: A Promise to the Compiler

This operator has **no effect at run-time**. It does not check for null or prevent a NullReferenceException. It is purely a compile-time instruction. When you use myNullableVariable!, you are making a promise to the compiler:

"I, the developer, know more than your static analysis in this specific situation. I have performed a check or have other knowledge that guarantees this value is not null right here. Please stop warning me about it."

If your guarantee is wrong and the variable is null at run-time, your program will still crash with a NullReferenceException. You are taking on the responsibility for the null check yourself.

Why and When to Use It (With Caution)

You should use this operator sparingly. Overusing it defeats the entire purpose of the nullable reference types feature. The primary valid use case is when you have checked for null in a way that the compiler's flow analysis cannot understand.

Example: Using a Helper Method The compiler can't see inside the IsNotNull method, so it doesn't know that user is safe to use after the call.

```
public void ProcessUser(User? user)
{
   // This helper method throws an exception if the user is null.
   EnsureNotNull(user);

   // Even after the check, the compiler still warns that 'user' may be null
here.
   // We use the null-forgiving operator '!' to tell the compiler it's safe.
   Console.WriteLine($"Processing user: {user!.Name}");
}

public void EnsureNotNull(object? obj)
{
   if (obj == null)
   {
      throw new ArgumentNullException();
   }
}
```

Lecture 7: Target-Typed 'new' Expressions

Introduction

This C# 9 feature provides a more concise syntax for object instantiation. It allows you to omit the type name in a new expression when the compiler can already infer the type from the context (i.e., from the variable declaration or method parameter).

How It Works

If the compiler already knows what type of object needs to be created, you can simply use new() instead of new MyClassName().

Why and When to Use It

It reduces code repetition and boilerplate, especially when working with long type names or generic types. It's most useful in two common scenarios:

1. **Initializing Fields and Properties:** When declaring a field or property, the type is already specified on the left side.
2. **Passing Arguments to Methods:** When passing a new object as an argument, the method's parameter type provides the context.

Example

```
// A class to demonstrate with
public class User { public User(string name) { Name = name; } public string Name; }

public class UserManager
{
    // Use Case 1: Initializing a field.
    // Before: private List<User> _users = new List<User>();
    private List<User> _users = new(); // C# 9: Type is inferred from '_users'.

    public void AddUser()
    {
        // Use Case 2: Declaring a local variable.
        // Before: User newUser = new User("Alice");
        User newUser = new("Alice"); // C# 9: Type is inferred from 'newUser'.

        _users.Add(newUser);
    }
}
```

Lecture 8: Intro & Why to use Pattern Matching

Introduction

Pattern matching is a powerful suite of features in modern C# that provides a more declarative and readable way to check if a variable conforms to a certain "shape" or "pattern," and to extract information from it if it does.

Why Use It: Beyond Simple Equality

Before pattern matching, `if` and `switch` statements were limited. You could check for equality (`if (x == 5)`), or check a type (`if (x is MyClass)`), but combining these checks was often verbose and clunky.

Pattern matching evolves this by allowing you to express complex conditional logic based on both the **type** of an object and its **state** (the values of its properties) in a single, concise expression. It fundamentally makes your code more declarative—you describe the "pattern" of the data you're interested in, rather than writing a long series of imperative checks. This leads to code that is often more readable, less error-prone, and easier to maintain.

The following lectures will explore the specific patterns that make this possible.

Lecture 9: Pattern Matching - Type Pattern

Introduction

The **type pattern** is the most fundamental pattern. It improves upon the old `is` operator by not only checking the run-time type of an object but also, if the check succeeds, assigning that object to a new, strongly-typed variable that is immediately available for use.

How It Works

This pattern avoids the old, clunky two-step process of checking a type and then casting it.

The Old Way:

```
if (myObject is string)
{
   string s = (string)myObject;
   Console.WriteLine($"The string has {s.Length} characters.");
}
```

The Modern Type Pattern: is Type variableName

```
if (myObject is string s) // Check and assign in one step.
{
   // The variable 's' is now in scope and is strongly-typed as a string.
   Console.WriteLine($"The string has {s.Length} characters.");
}
```

Example

```
public void ProcessValue(object value)
{
   // Use the type pattern to handle different data types gracefully.
   if (value is string s)
   {
      Console.WriteLine($"Received a string with length: {s.Length}");
   }
   else if (value is int i)
   {
      Console.WriteLine($"Received an integer. Its square is: {i * i}");
   }
}
```

Lecture 10: Pattern Matching - Switch-Case Pattern

Introduction

Starting in C# 7.0, the traditional switch statement was enhanced to support pattern matching, allowing it to switch on the **type** of an object, not just constant values like integers or strings.

How It Works

You can use the type pattern directly in a case label. This provides a much cleaner and more readable alternative to a long chain of if-else if statements when you need to perform different actions based on an object's runtime type.

Syntax: case Type variableName:

Example

Let's refactor the previous example to use a switch statement, which is often more organized when there are many types to check.

```
public abstract class Shape { }
public class Circle : Shape { public double Radius; }
public class Rectangle : Shape { public double Width, Height; }

public void PrintShapeInfo(Shape shape)
{
   switch (shape)
   {
      case Circle c: // If 'shape' is a Circle, assign it to 'c'.
         Console.WriteLine($"This is a circle with radius {c.Radius}.");
         break;

      case Rectangle r: // If 'shape' is a Rectangle, assign it to 'r'.
         Console.WriteLine($"This is a rectangle with width {r.Width}.");
         break;

      case null: // You can even specifically check for null.
         Console.WriteLine("The shape is null.");
         break;

      default:
         Console.WriteLine("This is an unknown shape.");
         break;
   }
}
```

Lecture 11: Pattern Matching - when Pattern

Introduction

The when keyword can be used to add an additional boolean condition, or **filter**, to a case in a switch statement. This allows you to create much more specific and powerful case labels.

How It Works

A case with a when clause will only be executed if both the primary pattern (e.g., the type pattern) matches **and** the condition in the when clause evaluates to true.

Syntax: case Type variableName when (condition):

When to Use It

Use a when clause to distinguish between objects of the same type but with different states. For example, you can have one case for a Product when its IsOnSale property is true, and another case for the same Product type when it's false.

Example

```
public class Order { public decimal Total; }

public void ProcessOrder(object order)
{
   switch (order)
   {
      // This case ONLY matches if the object is an Order AND its Total is
over 5000.
      case Order o when o.Total > 5000:
         Console.WriteLine("Applying high-value order protocol.");
         break;

      // This case matches any other Order that didn't match the case
above.
      case Order o:
         Console.WriteLine("Processing standard order.");
```

```
            break;

        default:
            Console.WriteLine("Not a valid order object.");
            break;
    }
}
```

Lecture 12: Pattern Matching - Switch Expression

Introduction

A **switch expression**, introduced in C# 8.0 and enhanced in later versions, is a modern, concise, and expression-based alternative to the traditional switch statement. The key difference is that a switch statement performs actions, while a switch expression **produces a value**.

How It Works: Syntax

The syntax is much more compact and functional in style:

variableToInspect switch { pattern1 => result1, pattern2 => result2, _ => defaultResult };

- **variable switch { ... }**: The variable being checked comes first.
- **pattern => result,**: Each line is an "arm." It consists of a pattern, the => operator, the result to produce if the pattern matches, and a comma.
- **_**: The discard pattern acts as the default case and is often required by the compiler to ensure the expression is exhaustive (handles all possible inputs).

Why and When to Use It

Use a switch expression when you need to get a single value based on a series of pattern matches. It is far cleaner and less error-prone than a traditional switch statement where every case block contains a return statement.

Example

Let's refactor our Shape example to a method that returns a descriptive string for each shape type.

```
public string GetShapeDescription(Shape shape) => shape switch
{
    // The pattern is on the left, the result is on the right.
    Circle c => $"A red circle with radius {c.Radius}",
    Rectangle r => $"A blue rectangle with width {r.Width}",
    _ => "An unknown shape." // The discard pattern for the default case.
};
```

Lecture 13: Pattern Matching - Relational and Logical Patterns

Introduction

C# 9 significantly enhanced pattern matching by introducing **relational patterns** (<, >, <=, >=) and **logical patterns** (and, or, not). These allow you to use comparisons and combine patterns directly within a case or switch expression arm.

How It Works

These keywords allow you to create powerful, declarative checks that are highly readable.

- **Relational Patterns:** Compare a value against a constant.
- **Logical Patterns:** Combine other patterns. and requires both patterns to match. or requires either pattern to match. not negates a pattern.

Example: A Temperature Categorizer

```
public string CategorizeTemperature(double temp) => temp switch
{
    < -10.0 => "Extreme cold",
    >= -10.0 and < 0 => "Freezing",
    >= 0 and < 15.0 => "Cold",
    >= 15.0 and < 25.0 => "Moderate",
    >= 25.0 and <= 35.0 => "Hot",
    > 35.0 => "Extreme heat"
};

Console.WriteLine(CategorizeTemperature(20)); // Output: Moderate
Console.WriteLine(CategorizeTemperature(32)); // Output: Hot
```

Lecture 14: Pattern Matching - Property Pattern

Introduction

The **property pattern** allows you to match an object based on the **values of its public properties or fields**. This lets your conditional logic depend not just on the type of an object, but on its internal state.

How It Works

You use curly braces { } to specify the properties you want to check. Inside the braces, you list the property names followed by a colon and another pattern to match against.

Syntax: { PropertyName1: pattern1, PropertyName2: pattern2 }

Example: Calculating Vehicle Toll

Let's create a method that calculates a vehicle toll based on its type and properties.

```
public abstract class Vehicle { }
public class Car : Vehicle { public int PassengerCount { get; set; } }
public class Bus : Vehicle { public int Capacity { get; set; } }
public class DeliveryTruck : Vehicle { public int WeightKg { get; set; } }

public decimal CalculateToll(Vehicle vehicle) => vehicle switch
{
    // Property pattern checks the PassengerCount property.
    Car { PassengerCount: 0 } => 2.00m + 0.50m, // Empty car extra charge
    Car { PassengerCount: 1 } => 2.00m,
    Car { PassengerCount: 2 } => 1.50m,

    // Property pattern with a relational pattern.
    Bus { Capacity: > 40 } => 5.00m,
    Bus => 3.50m, // Matches any other bus

    // Property pattern with a 'when' clause for more complex logic.
    DeliveryTruck t when t.WeightKg > 5000 => 7.50m,
    DeliveryTruck => 4.50m,

    _ => 2.50m // Default for unknown vehicles
};
```

Lecture 15: Pattern Matching - Tuple Pattern

Introduction

The **tuple pattern** allows you to perform pattern matching on the elements of a tuple. You can match against literal values or use other patterns for each element of the tuple.

How It Works

You use parentheses (pattern1, pattern2, ...) to match the corresponding elements of an input tuple. This is extremely powerful in switch expressions where the input is a tuple representing a combination of states.

Example: Rock-Paper-Scissors Game Logic

```
public string GetGameResult(string player1Move, string player2Move) =>
(player1Move, player2Move) switch
{
    // Player 1 wins
    ("rock", "scissors") => "Player 1 wins",
    ("scissors", "paper") => "Player 1 wins",
    ("paper", "rock") => "Player 1 wins",

    // Player 2 wins
    ("scissors", "rock") => "Player 2 wins",
    ("paper", "scissors") => "Player 2 wins",
    ("rock", "paper") => "Player 2 wins",

    // Draw (using a 'when' clause for a more complex condition)
    (var p1, var p2) when p1 == p2 => "It's a draw",

    // Default case
    _ => "Invalid moves"
};

Console.WriteLine(GetGameResult("rock", "scissors")); // Output: Player 1
wins
Console.WriteLine(GetGameResult("paper", "paper"));  // Output: It's a
draw
```

Lecture 16: Pattern Matching - Positional Pattern

Introduction

The **positional pattern** provides another way to deconstruct an object and match against its values. It relies on an accessible Deconstruct method being present on the object's type.

How It Works

Instead of using property names in curly braces, you use parentheses (pattern1, pattern2, ...) to match the values that would be returned by the Deconstruct method in order. record types (which we'll see later) are perfect for this as they get a Deconstruct method automatically.

Example

Let's define a Point with a Deconstruct method and use positional patterns.

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    // This method enables positional pattern matching.
    public void Deconstruct(out int x, out int y)
    {
        x = this.X;
        y = this.Y;
    }
}

public string CheckPoint(Point p) => p switch
{
    // Positional pattern to check if the point is at the origin.
    (0, 0) => "At the origin",

    // Positional pattern using relational patterns.
    (> 0, > 0) => "In the first quadrant",

    // Positional pattern capturing a value.
    (var x, 0) => $"On the X-axis at {x}",

    // Default case
    _ => "Somewhere else"
};
```

Lecture 17: Pattern Matching - Extended Property Pattern

Introduction

The extended property pattern, introduced in C# 10, is a simplification that makes it easier to write patterns that check nested properties.

How It Works

Instead of nesting property patterns, you can now use a "dot" notation directly within a single set of curly braces.

The Old Way (before C# 10): { Customer: { Address: { Country: "USA" } } }

The New, Flatter Way (C# 10+): { Customer.Address.Country: "USA" }

Example

```
public class Address { public string Country { get; set; } }
public class Customer { public Address Address { get; set; } }
public class Order { public Customer Customer { get; set; } }

public decimal GetShippingRate(Order order) => order switch
{
    // Using the extended property pattern is much cleaner.
    { Customer.Address.Country: "USA" } => 5.00m,
    { Customer.Address.Country: "Canada" } => 10.00m,
    _ => 25.00m
};
```

Lecture 18: Need of Immutable Classes

Introduction

Immutability is a programming concept where an object's state **cannot be changed** after it has been created. In contrast, a **mutable** object is one whose state can be modified after creation. While mutability is the default for most classes in C#, creating immutable types is a powerful technique for writing safer, more predictable, and more maintainable code.

The Problem with Mutability

Mutable objects can introduce significant complexity and bugs, especially in large applications.

1. **Unpredictable State Changes:** When a mutable object is passed around to different parts of an application, any part of that application can change the object's properties. This can lead to unexpected side effects where a change in one system causes a bug in a completely different, seemingly unrelated system. Debugging these issues can be a nightmare because you have to track down every piece of code that could have possibly touched the object.

2. **Multi-threading Issues:** When multiple threads try to read from and write to the same mutable object concurrently, it can lead to **race conditions** and data corruption. Protecting mutable state in a multi-threaded environment requires complex and error-prone locking mechanisms (lock statements).

The Benefit of Immutability

Immutable objects solve these problems.

- **Predictability:** If you have a reference to an immutable object, you can be 100% certain that its data is the same as when it was created. It can be passed around freely without fear of side effects.

- **Inherent Thread-Safety:** Since their state can never change, immutable objects are inherently thread-safe. Multiple threads can read from the same immutable object concurrently without any need for locks.

This leads to code that is simpler to reason about and eliminates a whole class of common bugs.

Lecture 19: Immutable Classes

Introduction

Before the new features in C# 9, creating an immutable class required a significant amount of boilerplate code and strict adherence to a specific pattern. Understanding this traditional pattern helps appreciate the modern features like init properties and records.

How to Implement (The Traditional Way)

To make a class immutable, you must ensure that its state cannot be changed after the object has been constructed. This involves these key steps:

1. **Make all private fields readonly**: This is the most important step. A readonly field can only be assigned a value at the time of its declaration or within the constructor.
2. **Provide only get accessors for all public properties**: By omitting the set accessor, you prevent external code from changing the property's value.
3. **Initialize all fields via the constructor**: The constructor becomes the one and only place where the object's state is set.

Example

```
public class Point
{
    // 1. All fields are private and readonly.
    private readonly int _x;
    private readonly int _y;

    // 2. Properties only have a 'get' accessor.
    public int X { get { return _x; } }
    public int Y { get { return _y; } }

    // 3. State is set only once, in the constructor.
    public Point(int x, int y)
    {
        _x = x;
        _y = y;
    }
```

```
}

// --- In your Main method ---
Point p1 = new Point(10, 20);
// p1.X = 30; // ERROR! Property or indexer 'Point.X' cannot be assigned to
-- it is read only.
```

Lecture 20: Init-Only Properties

Introduction

Init-only properties, introduced in C# 9, simplify the creation of immutable objects. An `init` accessor is a new type of property accessor that can only be called **during object initialization**.

How It Works

An `init` accessor acts like a `set` accessor, but with a crucial restriction. You can only assign a value to an `init`-only property:

- In the object initializer syntax (when you use `new { ... }`).
- In the constructor of the class.

Once the object's construction is complete, the `init`-only property becomes effectively `readonly`.

Why and When to Use It

`init` solves a major problem with traditional immutable classes. To use the convenient object initializer syntax (`new Point { X = 10, Y = 20 }`), properties needed to have a public `set` accessor, which meant they weren't truly immutable. `init` gives you the best of both worlds: **the flexibility of object initializers with the safety of immutability**.

Example

```
public class WeatherObservation
{
    // These properties can be set during initialization, but not after.
    public DateTime ObservationTime { get; init; }
    public double TemperatureCelsius { get; init; }
    public double WindSpeedKph { get; init; }
}

// --- In your Main method ---

// We can use the convenient object initializer syntax.
var observation = new WeatherObservation
{
    ObservationTime = DateTime.UtcNow,
    TemperatureCelsius = 25.0,
    WindSpeedKph = 15.5
};

// But after initialization, any attempt to change the properties
// will result in a compiler error.
// observation.TemperatureCelsius = 26.0; // ERROR! Init-only property
can only be assigned in an object initializer...
```

Lecture 21: Readonly Structs

Introduction

The readonly modifier can be applied to an entire struct declaration to create a fully **immutable value type**. This is a powerful feature for creating small, lightweight data containers that are guaranteed to be safe from modification.

How It Works

When you add the readonly modifier to a struct declaration (public readonly struct Point), the C# compiler enforces two rules:

1. All instance fields within the struct **must also be declared as readonly**.
2. All instance properties must be **read-only** (i.e., they can only have a get accessor).

This creates a compile-time guarantee that once an instance of the struct is created, its state can never change.

Why and When to Use It

Use a readonly struct when you want to represent a "value" that is small and should not change. It's perfect for things like coordinates, colors, or complex numbers. Because they are both immutable and value types (stack-allocated), they can lead to very safe and high-performance code, especially in multi-threaded or mathematical applications.

Example

```
// 'readonly' guarantees the entire struct is immutable.
public readonly struct RgbColor
{
   // All fields must also be readonly.
   public readonly byte R;
   public readonly byte G;
   public readonly byte B;

   public RgbColor(byte r, byte g, byte b)
   {
      this.R = r;
      this.G = g;
      this.B = b;
   }
}
```

Lecture 22: Parameterless Struct Constructors

Introduction

Before C# 10, developers could not declare their own parameterless constructor for a struct. Every struct had an implicit, unchangeable parameterless constructor that simply initialized all its fields to their default system values (0, false, etc.). C# 10 removed this restriction.

How It Works (C# 10 and later)

You can now declare a public, parameterless constructor in a struct to provide custom default initialization logic. However, the system's implicit default constructor (which zeroes out memory) still exists and will be used when you create an array of structs, for example.

When to Use It

Use this feature when you want an instance of a struct created with new MyStruct() to have a different, more sensible default state than all zeros.

Example

Let's create a Point struct where the default new() constructor places it at (1, 1) instead of (0, 0).

```
public struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    // C# 10 allows this explicit parameterless constructor.
    public Point()
    {
        X = 1;
        Y = 1;
    }
}


// --- In your Main method ---
```

```
// This now calls our custom parameterless constructor.
Point p1 = new Point();
Console.WriteLine($"p1 is at ({p1.X}, {p1.Y})"); // Output: (1, 1)

// However, creating an array still uses the implicit default constructor.
Point[] points = new Point[1];
Point p2 = points[0];
Console.WriteLine($"p2 is at ({p2.X}, {p2.Y})"); // Output: (0, 0)
```

Lecture 23: Intro and Working with Records

Introduction

Record types, introduced in C# 9, are a new kind of reference type designed specifically for creating **immutable data models**. A record is essentially syntactic sugar that tells the compiler to automatically generate a lot of the boilerplate code that you would normally have to write by hand for a data-centric, immutable class.

How It Works: Positional Records and Generated Code

The most concise way to declare a record is using **positional syntax**.

When you write this single line:

```
public record Person(string FirstName, string LastName);
```

The C# compiler generates the following for you:

1. A constructor that takes FirstName and LastName.
2. Public **init-only** properties for FirstName and LastName.
3. A Deconstruct method, allowing you to easily unpack the object into variables.
4. An overridden ToString() method that provides a clean, readable output of the object's state.
5. Overridden Equals() and GetHashCode() methods that perform **value-based equality**, not the default reference equality of classes. This is a major feature.

Example

```
// This one line of code generates all the features mentioned above.
public record Person(string FirstName, string LastName);

// --- In your Main method ---

Person person1 = new Person("John", "Doe");

// 1. The generated ToString() is automatically used by WriteLine.
Console.WriteLine(person1); // Output: Person { FirstName = John, LastName = Doe }

// 2. The properties are init-only, making the object immutable.
// person1.FirstName = "Jane"; // ERROR! Init-only property.
```

Lecture 24: Nested Records

Introduction

Just as you can nest a class inside another class, you can also define a record type within the scope of another class or record. This follows the same logic as nested classes.

Why and When to Use It

Use a nested record when a small, immutable data container is **tightly coupled** to and only ever used by its containing class. This is an organizational tool that keeps the helper data structure close to the only class that uses it, preventing it from "polluting" the public namespace.

Example

Imagine an Order class that needs to store shipping information. The ShippingAddress is a simple data structure only relevant to the Order.

```
public class Order
{
    // This is a nested record. It's only accessible through the Order class.
    public record Address(string Street, string City, string PostalCode);

    public int OrderId { get; init; }
    public Address ShippingDestination { get; init; }

    public void PrintShippingLabel()
    {
        // We can use the nested record's properties here.
        Console.WriteLine($"--- SHIPPING LABEL for Order #{OrderId} ---");
        Console.WriteLine(ShippingDestination.Street);
        Console.WriteLine($"{ShippingDestination.City},
{ShippingDestination.PostalCode}");
    }
}

// --- In your Main method ---
// To create an instance, you must qualify it with the containing class
name.
var shippingAddress = new Order.Address("123 Main St", "Anytown",
"12345");
var order = new Order { OrderId = 101, ShippingDestination =
shippingAddress };

order.PrintShippingLabel();
```

Lecture 25: Understanding Immutability of Records

Introduction

Records are designed from the ground up to support **immutability**. This means that once a record object is created, its state should not change. This is primarily achieved through init-only properties and non-destructive mutation.

However, it is crucial to understand that this immutability is **shallow** by default when reference types are involved.

How they are Immutable

1. **Init-Only Properties:** When you use the concise positional syntax (public record Person(string Name);), the compiler generates init-only properties. This prevents you from changing the value after the object has been initialized.
2. **with Expressions:** As we'll see next, instead of changing a record, you create a new record with modified values using a with expression. This is called non-destructive mutation.

Shallow Immutability: The Important Caveat

This is a critical interview point. If a record contains a property that is a **mutable reference type** (like a List<T>), the record itself is only **shallowly immutable**.

- The **reference** to the list cannot be changed. You cannot assign a completely new list to the property.
- However, the **contents of the list itself can still be modified**. You can still call .Add() or .Remove() on the list, which mutates the state of the original record object.

Example

```
// A record with a mutable collection property.
public record BlogPost(string Title, List<string> Tags);

// --- In your Main method ---
var post = new BlogPost("My First Post", new List<string> { "c#",
"learning" });

Console.WriteLine(post);
// Output: BlogPost { Title = My First Post, Tags =
System.Collections.Generic.List`1[System.String] }

// This would be an error because 'Tags' is an init-only property.
// post.Tags = new List<string>(); // ERROR!
```

```
// BUT, we can still modify the *contents* of the list that the record points to.
post.Tags.Add("new");

// The original 'post' object has now been mutated.
Console.WriteLine(post);
// Output: BlogPost { Title = My First Post, Tags = System.Collections.Generic.List`1[System.String] }
// (Note: The ToString() doesn't show list contents, but the list now has 3 items)
```

Lecture 26: Records - Equality

Introduction

One of the most powerful and important features of records is that they exhibit **value-based equality** by default. This is a major departure from classes, which use reference equality by default.

How It Works

The C# compiler automatically overrides the Equals() method and the == and != operators for record types.

- **Reference Equality (for classes):** classA == classB is only true if both variables point to the exact same object in memory.
- **Value-Based Equality (for records):** recordA == recordB is true if the two records are of the **same type** and the values of all their corresponding **public properties** are equal.

The compiler also generates a matching GetHashCode() override based on the values of all public properties, satisfying the rule that equal objects must have equal hash codes.

Example

```
public record Person(string FirstName, string LastName);

// --- In your Main method ---
var p1 = new Person("John", "Doe");
var p2 = new Person("John", "Doe"); // A separate object with the same data.
var p3 = new Person("Jane", "Doe");

// For regular classes, p1 == p2 would be FALSE.
// For records, it's TRUE because their values are the same.
Console.WriteLine($"p1 == p2: {p1 == p2}"); // Output: True

Console.WriteLine($"p1.Equals(p2): {p1.Equals(p2)}"); // Output: True

Console.WriteLine($"p1 == p3: {p1 == p3}"); // Output: False
```

Lecture 27: Records - 'with' Expression

Introduction

The **with expression** is a special feature for records that provides a simple syntax for performing **non-destructive mutation**. Since records are meant to be immutable, you don't change them directly. Instead, you create a copy of the record with one or more properties changed to new values.

How It Works

The with expression creates a shallow copy of the original record object and then applies the property changes specified in the object initializer block. The original record object is left completely untouched.

Syntax: var newRecord = oldRecord with { PropertyName = newValue };

Example

```
public record Book(string Title, string Author, int PublicationYear);

// --- In your Main method ---
var originalBook = new Book("The Hobbit", "J.R.R. Tolkien", 1937);

// Let's create a new record for a "revised edition".
// It creates a copy of originalBook and then changes the
PublicationYear.
var revisedEdition = originalBook with { PublicationYear = 2007 };

Console.WriteLine(originalBook);
// Output: Book { Title = The Hobbit, Author = J.R.R. Tolkien,
PublicationYear = 1937 }

Console.WriteLine(revisedEdition);
// Output: Book { Title = The Hobbit, Author = J.R.R. Tolkien,
PublicationYear = 2007 }

// The original object is unchanged.
Console.WriteLine($"Original year is still: {originalBook.PublicationYear}");
```

Lecture 28: Records - Deconstruct

Introduction

When you declare a record using the concise positional syntax, the compiler automatically generates a Deconstruct method for you. This allows you to easily unpack the properties of a record into separate variables, just like with tuples.

How It Works

The generated Deconstruct method has an out parameter for each positional parameter defined in the record's primary constructor. This enables the deconstruction syntax.

Example

```
public record Person(string FirstName, string LastName, int Age);

// --- In your Main method ---
var person = new Person("John", "Doe", 42);

// Deconstruct the record's properties into new variables.
var (fn, ln, age) = person;

Console.WriteLine($"The first name is {fn}");
Console.WriteLine($"The last name is {ln}");
Console.WriteLine($"The age is {age}");
```

Lecture 29: Records - ToString

Introduction

A simple but significant benefit of using records is that the compiler automatically overrides the `ToString()` method to provide a clean, readable, built-in text representation of the object's state.

How It Works

Unlike a standard class, which by default would only print its type name (e.g., `MyProject.Person`), the record's generated `ToString()` method prints the type name followed by the names and values of all its public properties, formatted in a style that looks similar to an object initializer. This makes debugging and logging with records much more convenient.

Example

```
public record Person(string FirstName, string LastName);

// --- In your Main method ---
var person = new Person("John", "Doe");

// Console.WriteLine automatically calls the .ToString() method on the object.
Console.WriteLine(person);
```
Output: Person { FirstName = John, LastName = Doe }

Lecture 30: Records - Inheritance

Introduction

Record types support inheritance, allowing you to create hierarchies of immutable data models. However, the rules are specific:

- A record can inherit from another record.
- A record cannot inherit from a class.
- A class cannot inherit from a record.

How It Works

Inheritance for records follows the standard syntax and rules. The derived record inherits all the properties of the base record. The compiler-generated features like value-based equality and ToString() are automatically updated to include the properties from both the base and the derived records.

Example

```
// Base record
public record Person(string FirstName, string LastName);

// Derived record that inherits from Person.
// It passes the common properties up to the base constructor.
public record Employee(string FirstName, string LastName, int
EmployeeId) : Person(FirstName, LastName);

// --- In your Main method ---
var employee = new Employee("Jane", "Smith", 101);

// The generated ToString() includes properties from both base and
derived records.
Console.WriteLine(employee);
// Output: Employee { FirstName = Jane, LastName = Smith, EmployeeId
= 101 }
```

Lecture 31: Records - sealed ToString

Introduction

This C# 10 feature gives you control over the ToString() behavior in a record inheritance hierarchy. By marking an overridden ToString() method as sealed, you can "lock it down" and prevent any further derived records from generating their own ToString() implementation.

How It Works

If you create a sealed override of ToString in a base record, any record that inherits from it will use this "sealed" implementation instead of generating a new one that includes its own properties.

When to Use It

Use this when you want to define a final, canonical string representation for an entire record hierarchy and you do not want the output to change even when new derived records are created.

Example

```
// Base record with a sealed ToString
public record Person(string Name)
{
    // This implementation is now final for all derived records.
    public sealed override string ToString()
    {
        return $"Person: {Name}";
    }
}

// Derived record
public record Employee(string Name, int EmployeeId) : Person(Name);

// --- In your Main method ---
var employee = new Employee("John Doe", 101);

// This now calls the sealed implementation from the Person base record.
Console.WriteLine(employee); // Output: Person: John Doe
```

Lecture 32: Record Structs

Introduction

Record structs, introduced in C# 10, combine the compiler-generated benefits of records with the performance and semantics of a **value type (struct)**.

How It Works

You declare them using `record struct`. Like a class-based record, a record struct automatically gets:

- Value-based equality (`Equals`, `GetHashCode`, `==`, `!=`).
- A clean `ToString()` implementation.
- A `Deconstruct` method.
- `init`-only properties (if declared as `readonly record struct`).

The key difference is that it is a **value type**. It lives on the stack (typically), is copied on assignment, and cannot be null.

Why and When to Use It

Use a record struct for small, immutable data models where value type semantics and high performance are desired. It's an excellent modern replacement for a simple readonly struct because you get all the equality and display methods generated for free.

Example

```
// A readonly record struct is a fully immutable value type.
public readonly record struct Point(int X, int Y);

// --- In your Main method ---
var p1 = new Point(10, 20);
var p2 = new Point(10, 20); // A separate instance with the same values.

// Like a record class, it has value-based equality.
Console.WriteLine($"p1 == p2: {p1 == p2}"); // Output: True

// It has a clean ToString() implementation.
Console.WriteLine(p1); // Output: Point { X = 10, Y = 20 }
```

Lecture 33: Command Line Arguments

Introduction

Command-line arguments are a way to pass data into a console application when it is launched from the command line. This is a fundamental feature that allows your applications to be configured or controlled dynamically, making them useful for automation and scripting.

How It Works: The args Array

The .NET runtime automatically collects any arguments provided after the executable name and makes them available to your program as an array of strings (string[]).

- **In a traditional Main method:** They are passed into the static void Main(string[] args) parameter.
- **In modern Top-Level Statements:** A "magic" variable named args is automatically available in the global scope.

Each space-separated value on the command line becomes a separate element in the args array. If an argument contains spaces, it must be enclosed in double quotes.

Real-World Examples

Example 1: A Simple Greeter

- **Command Line:** MyGreeter.exe "John Smith"

Code:

```
// In a top-level statement file (Program.cs)

if (args.Length > 0)
{
    string name = args[0];
    Console.WriteLine($"Hello, {name}!");
}
else
{
    Console.WriteLine("Hello, World!");
}
```

Example 2: A File Processing Utility

- **Command Line:** FileConverter.exe --input C:\data\report.csv --output C:\reports\report.json

Logic: This requires more sophisticated parsing. You would typically loop through the args array looking for "flags" (like --input) and then treat the next

element as the value for that flag.

```
// A simplified parsing example

string inputFile = null;
string outputFile = null;

for (int i = 0; i < args.Length; i++)
{
    if (args[i] == "--input" && i + 1 < args.Length)
    {
        inputFile = args[i + 1];
    }
    else if (args[i] == "--output" && i + 1 < args.Length)
    {
        outputFile = args[i + 1];
    }
}

Console.WriteLine($"Input file is: {inputFile}");
Console.WriteLine($"Output file is: {outputFile}");
```

Lecture 34: Partial Methods Return Type

Introduction

This C# 9 feature is an enhancement to the partial methods we discussed earlier. In previous versions of C#, partial methods were required to have a `void` return type. Now, they can return a value.

How It Works

When a partial method is declared with a return type (e.g., `partial bool IsValid();`), it creates a new rule. The compiler will no longer just remove the method call if no implementation is found. Instead, if an implementation is not provided in another part of the partial class, the compiler will generate an **error**.

Why and When to Use It

This change makes partial methods much more useful in **source generation** scenarios. A code generator can now define a "hook" method that is expected to return a value (like a validation result or a computed property). The generated code can then call this method and use its return value in its logic, with the compiler guaranteeing that the developer has provided the necessary implementation.

Lecture 35: Static Anonymous Functions

Introduction

This C# 9 feature is a performance optimization that allows you to add the `static` modifier to a lambda expression or an anonymous method.

How It Works

By declaring a lambda as `static`, you are making a promise to the compiler that this function **will not capture** (access) any variables from its containing scope. It cannot access local variables, parameters from the outer method, or the `this` reference of the class. It can only work with the parameters passed directly to it.

Syntax: `static (x, y) => x + y`

Why and When to Use It

This is primarily a performance optimization. A non-static lambda that "captures" outer variables requires the compiler to create a hidden "closure" object to hold those variables, which results in a small memory allocation. A `static` anonymous function avoids this allocation.

Use it when your helper lambda is "pure"—meaning its output depends only on its input parameters—to prevent accidental capturing of state and to give the compiler a hint for minor optimizations.

Example

```
public void ProcessData(List<int> numbers)
{
   int multiplier = 2;

   // This lambda CANNOT be static because it captures 'multiplier' from
the outer scope.
   var result1 = numbers.Select(n => n * multiplier);

   // This lambda CAN be static because it only uses its own parameter 'n'.
   var result2 = numbers.Select(static n => n * 2);
}
```

Lecture 36: Constant Interpolated Strings

Introduction

This is a quality-of-life feature from C# 10. It allows you to declare const strings that are built using string interpolation ($"").

How It Works

The key restriction is that all the values used inside the interpolated placeholders ({...}) must also be **compile-time constants**. You cannot use variables or method calls.

Example

```
public static class ApiRoutes
{
   private const string ApiBase = "/api/v1";

   // This is valid because ApiBase is also a const.
   public const string GetAllUsers = $"{ApiBase}/users";
   public const string GetUserById = $"{GetAllUsers}/{{id}}";
}
```

Lecture 37: Interface Default Methods and Static Members

Introduction

Starting with C# 8.0, interfaces became significantly more powerful, blurring the lines slightly with abstract classes by allowing them to contain code.

Default Interface Methods

- **What It Is:** You can now provide a default implementation (a method body) for a method directly inside an interface.
- **Why It's Important:** This is a crucial feature for **API evolution**. Imagine you have a public `ILogger` interface used by hundreds of developers. If you need to add a new method to it, like `LogWarning`, all existing classes that implement `ILogger` would break because they don't have an implementation for the new method. With default interface methods, you can add `LogWarning` to the interface with a default implementation. Existing classes will continue to work, and new classes can choose to either use the default or provide their own override.

Static Members in Interfaces

- **What It Is:** Interfaces can now contain `static` members, including methods, properties, and events.
- **Why It's Important:** This is useful for placing factory methods or utility functions directly on the interface where they are most relevant. For example, you could have an `ICalculator.Add(a, b)` static method. This is a key feature for enabling generic math in modern C#.

Private and Protected Members in Interfaces

- **What It Is:** Interfaces can now have `private` and `protected` members (both static and instance).
- **Why It's Important:** These are typically used as helper methods to share common logic between multiple default method implementations within the interface itself, avoiding code duplication.

Example

```
public interface ILogger
{
   // A required method for all implementers.
   void LogError(string message);

   // A default interface method. Existing classes don't have to implement
this.
   void LogInfo(string message)
   {
      // Call a private helper.
      Log("INFO", message);
   }

   // A private method to share logic within the interface.
   private static void Log(string level, string message)
   {
      Console.WriteLine($"[{level}] - {message}");
   }
}
```

Lecture 38: Index-From-End Operator

Introduction

The index-from-end operator ^, introduced in C# 8, provides a concise syntax for accessing elements from the end of an indexable collection.

How It Works

The ^ operator tells the indexer to count from the end rather than the beginning.

- ^1 is the **last** element.
- ^2 is the second-to-last element.
- ^0 is **not** the last element. It is equivalent to the collection's Length, so attempting to access myArray[^0] will result in an IndexOutOfRangeException.

Example

```
string[] names = { "Alice", "Bob", "Charlie", "David", "Eve" };

// Get the last element.
string lastName = names[^1]; // Eve

// Get the second-to-last element.
string secondToLastName = names[^2]; // David

Console.WriteLine($"The last name is: {lastName}");
Console.WriteLine($"The second-to-last name is: {secondToLastName}");
```

Lecture 39: Range Struct

Introduction

The .. operator, also introduced in C# 8, is used to create a Range object. It provides a simple and powerful syntax for specifying a "slice" or sub-section of a collection.

How It Works

The range operator is used inside an indexer []. The number on the left of .. is the inclusive start of the range, and the number on the right is the **exclusive** end of the range. It can be used with both standard integer indices and the index-from-end ^ operator.

Syntax Variations:

- myArray[1..4]: Gets elements from index 1 up to (but not including) index 4.
- myArray[..3]: From the start up to index 3.
- myArray[3..]: From index 3 to the end.
- myArray[2..^1]: From index 2 up to the second-to-last element.

Example

```
string[] names = { "Alice", "Bob", "Charlie", "David", "Eve" };
// Indices:         0       1       2         3       4
// From-End:       ^5      ^4      ^3        ^2      ^1

// Get the middle three names (Bob, Charlie, David)
string[] middleNames = names[1..4]; // Get from index 1 up to 4.
Console.WriteLine("Middle names: " + string.Join(", ", middleNames));

// Get all names except the first and last.
string[] innerNames = names[1..^1];
Console.WriteLine("Inner names: " + string.Join(", ", innerNames));
```

Lecture 40: Things to Remember

- **C# is Becoming More Concise:** Many modern features like top-level statements, file-scoped namespaces, global usings, and expression-bodied members are all about reducing boilerplate and making C# code cleaner and quicker to write.
- **Immutability is a Key Theme:** Records, init-only properties, and readonly record structs are powerful tools that push developers towards safer, immutable designs. Understanding and advocating for immutability is a sign of a modern developer.
- **Pattern Matching Supercharges Conditional Logic:** Modern pattern matching has transformed if and switch into highly expressive tools. You should prefer switch expressions with patterns over long if-else if chains for clarity and power.
- **var is Static, dynamic is Dynamic:** This is a crucial distinction. var is a compile-time convenience for strongly-typed variables. dynamic bypasses the compiler and defers type checking to run-time, which should be used with great caution.
- **Interfaces are More Powerful:** With default implementations and static members, interfaces can now be used for API evolution and to group related utility methods, making them more flexible than ever before.

Section 31: Threading

Lecture 1: Intro to Concurrent Execution

Introduction

To master threading, we must first understand the concepts it enables: **concurrency** and **parallelism**. While they sound similar, they represent different ideas about how tasks are executed.

- **Concurrency:** This is the concept of managing multiple tasks that are making progress over **overlapping time periods**. The tasks don't have to be running at the exact same instant. Think of a chef in a kitchen preparing a meal. They might start boiling water on the stove, then turn to chop vegetables while the water heats up, then switch back to add pasta to the boiling water. Multiple tasks (boiling, chopping) are "in progress" at once, and the chef intelligently switches between them to keep everything moving forward. This is concurrency.
- **Parallelism:** This is the **simultaneous execution** of multiple tasks. This requires hardware with multiple processing units, like a multi-core CPU. In our kitchen analogy, this would be having two separate chefs, one dedicated to chopping vegetables and another dedicated to managing the stove, both working at the exact same time. This is parallelism.

Threading is the primary mechanism in C# for achieving **concurrency**. On a computer with a multi-core processor, this concurrency can then lead to true **parallelism**.

Why It's Important

Modern applications need to be responsive. Imagine a desktop application where you click a "Process Report" button. If that report takes 30 seconds to generate and the work is done on the main application thread, the entire user interface (UI) will **freeze** for those 30 seconds. The user won't be able to click other buttons, move the window, or interact with the application in any way.

Concurrent execution solves this. You can offload the long-running report generation to a separate, background thread. This frees up the main UI thread to continue responding to user input, keeping the application fluid and responsive. The same principle applies to a web server handling multiple incoming requests simultaneously.

Lecture 2: Intro to Threading

Introduction

A **thread** is the smallest sequence of programmed instructions that can be managed independently by an operating system's scheduler. You can think of it as an **independent path of execution** within a single process.

Every application you run is a **process**. A process is a container that encapsulates all the resources for an application, such as its memory space and file handles. By default, every process has at least one thread, the **Main Thread**. By creating additional threads, you enable your application to perform multiple operations concurrently.

How It Works Internally

Understanding the relationship between threads, memory, and the operating system is crucial.

- **The Call Stack:** Each thread gets its **own independent call stack**. This is the most critical concept. The call stack is a region of memory that keeps track of which methods are currently being executed. Because each thread has its own stack, Thread A can be inside `Method1()` while Thread B is simultaneously inside `Method2()`, and their local variables and parameters will not interfere with each other. They are completely isolated at the stack level.
- **Shared Heap Memory:** While stacks are separate, all threads within the same process **share the same heap memory**. This means that if two threads have a reference to the same object on the heap, they can both read from and write to that object's fields. This shared access is what makes threading powerful, but it is also the primary source of complexity and bugs (which we will cover in "Thread Synchronization").
- **The OS Scheduler and Time-Slicing:** The operating system's scheduler is the ultimate manager. On a single-core CPU, it creates the illusion of parallelism through a process called **time-slicing**. It gives each thread a tiny slice of CPU time (e.g., 20 milliseconds) to run. When the time is up, it performs a **context switch**: it saves the exact state of the current thread, loads the state of the next thread, and lets it run for its time slice. This switching happens so fast that it appears as though the threads are running simultaneously.

- **True Parallelism:** On a modern multi-core CPU, the OS scheduler can assign different threads to run on different physical cores at the exact same time. If you have two threads and two cores, you can achieve true parallelism.

Lecture 3: Main Thread

Introduction

The **Main Thread** is the primary thread that is automatically created when any C# application starts. It is the thread that begins execution in your Program.Main method and is the parent of all other threads you might create within your application.

The Special Role of the UI Thread

In applications with a graphical user interface (GUI), such as WPF, WinForms, or MAUI, the Main Thread has a special, critical role: it is also the **UI Thread**.

The UI Thread is responsible for managing a message loop that handles:

1. **User Input:** All mouse clicks, key presses, and touch events.
2. **System Events:** Window resizing, focus changes, etc.
3. **Rendering:** All painting and updating of the controls on the screen.

Why It Must Not Be Blocked

If you perform a long-running, CPU-intensive operation directly on the UI Thread, you are blocking this message loop. The thread gets stuck inside your long-running method and cannot get back to its primary job of processing messages. From the user's perspective, the consequences are immediate and severe:

- The application window **freezes**.
- Buttons and other controls do not respond to clicks.
- The window does not repaint itself correctly if moved or resized.

- After a few seconds, the operating system will often gray out the application and display a "(Not Responding)" message in the title bar.

This is the primary motivation for using background threads in UI applications—to offload any work that takes more than a fraction of a second from the UI thread to keep the application responsive and provide a good user experience.

Lecture 4: Overview Thread class

Introduction

The **System.Threading.Thread** class is the fundamental class in .NET for creating and controlling a thread directly. While modern C# offers higher-level, easier-to-use abstractions for concurrency (like the Task Parallel Library, which we'll see later), understanding the Thread class is essential for grasping the core principles of how threading works at a lower level. It gives you direct control over the execution path.

Key Members

Here are the most important properties and methods of the Thread class that we will be covering in the subsequent lectures.

- **Constructor:** new Thread(ThreadStart) or new Thread(ParameterizedThreadStart): Creates a new thread object, associating it with a delegate that points to the method the thread will execute.
- **Start()**: A non-blocking method that transitions the thread to a runnable state. The OS scheduler will then begin executing it.
- **Sleep(int milliseconds)**: A static method that pauses the *currently executing* thread for a specified duration.
- **Join()**: A blocking method that pauses the *calling* thread until the thread it is called on has completed its execution.
- **Priority**: A property that allows you to set a ThreadPriority enum value (Lowest, Normal, Highest, etc.) to give a hint to the OS scheduler about the thread's importance.

- **ThreadState**: A property that returns an enum indicating the current state of the thread (e.g., Running, Stopped, WaitSleepJoin).
- **Interrupt()**: A method used to interrupt a thread that is in a blocked or waiting state (like Sleep or Join), causing it to throw a ThreadInterruptedException.
- **IsBackground**: A boolean property that determines if a thread is a background or foreground thread. The application will not terminate until all **foreground** threads have completed.

Lecture 5: Single-Threaded App

Introduction

A single-threaded application executes all its tasks **sequentially**, one after the other, on the Main Thread. No task can begin until the previous task has completely finished. This is the simplest programming model to understand and debug, as the flow of execution is linear and predictable. However, it is not suitable for applications that need to remain responsive while performing long-running work.

Execution Analysis

In a single-threaded model, if TaskA takes 5 seconds and TaskB takes 5 seconds, the total execution time will be 10 seconds. The Main method will call HeavyTaskA() and will **block**—it will wait at that line of code without proceeding—until HeavyTaskA returns. Only then will it proceed to call HeavyTaskB().

Example

This example demonstrates the sequential nature of a single-threaded program. Notice in the output how Task B started. does not appear until after Task A finished. is printed.

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Main thread started. Starting sequential tasks...");

      HeavyTaskA();
      HeavyTaskB();

      Console.WriteLine("Main thread finished.");
   }

   static void HeavyTaskA()
   {
      Console.WriteLine("  Task A started.");
      for (int i = 1; i <= 5; i++)
      {
         Console.WriteLine("    Task A is working...");
         Thread.Sleep(500); // Simulate 0.5 seconds of work
      }
      Console.WriteLine("  Task A finished.");
   }

   static void HeavyTaskB()
   {
      Console.WriteLine("  Task B started.");
      for (int i = 1; i <= 5; i++)
      {
         Console.WriteLine("    Task B is working...");
         Thread.Sleep(500); // Simulate 0.5 seconds of work
      }
      Console.WriteLine("  Task B finished.");
   }
}
```

Simplified Output:

```
Main thread started. Starting sequential tasks...
  Task A started.
    Task A is working...
    ... (repeats 5 times) ...
```

```
  Task A finished.
  Task B started.
    Task B is working...
    ... (repeats 5 times) ...
  Task B finished.
Main thread finished.
```

Lecture 6: Multiple Threads

Introduction

By creating new `Thread` objects, we can break free from sequential execution and perform multiple tasks concurrently. Each task will run on its own thread, with its execution path managed independently by the operating system scheduler. This allows long-running tasks to execute in the background without blocking the Main Thread.

How It Works: The 3-Step Process

1. **Define the Work:** Create the method that the new thread will execute. This method must have a `void` return type and, for a simple thread, no parameters.
2. **Create the `Thread` object:** Instantiate a `new Thread()`. The constructor takes a delegate that points to your work method. The `ThreadStart` delegate is used for methods with no parameters. The compiler is smart enough that you can usually just pass the method name directly.
3. **Start the Thread:** Call the `thread.Start()` method. This is a **non-blocking** call. It returns immediately to the calling thread after telling the OS scheduler to place the new thread in a "runnable" state. The OS will then begin executing it when CPU resources become available.

Example

Let's refactor the previous example to run HeavyTaskA and HeavyTaskB on separate, concurrent threads.

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Main thread started.");

      // Create new Thread objects, pointing them to the methods to
execute.
      Thread threadA = new Thread(HeavyTaskA);
      Thread threadB = new Thread(HeavyTaskB);

      // Set thread names for better debugging output.
      threadA.Name = "TaskA_Thread";
      threadB.Name = "TaskB_Thread";

      // Start both threads. These calls return immediately.
      threadA.Start();
      threadB.Start();

      // The Main thread does not wait and finishes its work right away.
      Console.WriteLine("Main thread has finished its tasks.");
   }
   // HeavyTaskA and HeavyTaskB methods are the same as before...
}
```

Analyzing the Output: If you run this code, you will see the output from "Task A" and "Task B" interleaved. The OS scheduler switches between threadA and threadB, giving each a slice of CPU time. You will also see that the "Main thread has finished" message appears almost instantly, because the Start() calls did not block it. The application as a whole, however, will not exit until both background threads have also finished.

Lecture 7: Thread.Sleep

Introduction

Thread.Sleep(int milliseconds) is a static method that pauses the **currently executing thread** for a specified duration. It is a way for a thread to voluntarily yield its time slice back to the operating system and enter a waiting state.

How It Works: Internal State

When a thread calls Thread.Sleep(), the OS scheduler transitions it from a "Running" state to a **WaitSleepJoin** state. The OS will not schedule this thread to run on a CPU core again until the specified time interval has elapsed.

It's important to understand that Sleep is **not a precise timer**. It guarantees a *minimum* pause, but the OS may take longer to reschedule the thread, especially on a busy system. You are telling the thread to sleep for *at least* the specified time.

Special Use Cases

- **Thread.Sleep(0)**: This is a special signal to the OS. It tells the scheduler to immediately end the current thread's time slice and yield the CPU to any other threads of the **same priority** that are ready to run. If there are no other ready threads, the current thread may continue immediately.
- **Thread.Sleep(1)**: This is often used inside "busy-wait" loops (loops that continuously check a condition) to prevent the loop from consuming 100% of a CPU core. It yields to the OS scheduler, greatly reducing CPU usage while still checking the condition frequently.

Example

Using Thread.Sleep inside our task methods helps to simulate I/O operations (like waiting for a web request or a file to download) and makes the concurrent execution more obvious. The examples in the previous lectures already use Thread.Sleep(500) for this purpose.

Lecture 8: Thread.Join

Introduction

The instance method thread.Join() is a fundamental synchronization mechanism. It **blocks the calling thread** until the thread on which Join was called has completed its execution.

How It Works: Waiting for Completion

Think of it like a project manager (the Main Thread) telling a team member (a worker thread) to go complete a task. Instead of moving on to other work, the project manager decides to wait at the team member's desk until they return and say, "I'm finished." The Join() method is that act of waiting. The Main Thread will enter the WaitSleepJoin state until the worker thread terminates.

Why and When to Use It

Its primary use case is to ensure that one or more background computations are fully complete before the main thread proceeds. This is essential when:

1. The main thread needs to use the **results** calculated by the background thread(s).
2. The main thread needs to ensure all background work (like saving files) is finished before the application can safely exit.

Example

Let's modify our multi-threaded example so the Main Thread waits for both tasks to complete before the program exits.

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Main thread: Starting background tasks...");

      Thread threadA = new Thread(HeavyTaskA);
      Thread threadB = new Thread(HeavyTaskB);

      threadA.Start();
```

```
        threadB.Start();

        // The 'Join' method blocks the Main thread here.
        // It will not proceed past this line until 'threadA' has finished.
        threadA.Join();
        Console.WriteLine("Main thread: Task A has completed.");

        // Similarly, wait for threadB to finish.
        threadB.Join();
        Console.WriteLine("Main thread: Task B has completed.");

        Console.WriteLine("Main thread: All background tasks are finished.
Program exiting.");
    }
    // ... HeavyTaskA and HeavyTaskB methods ...
}
```

Lecture 9: ThreadPriority

Introduction

The ThreadPriority is an enum property on a Thread object that allows you to give a **hint** to the operating system's scheduler about the relative importance of a thread. It does not provide a guarantee of execution order, but rather suggests how the scheduler should allocate CPU time slices among runnable threads.

How It Works

The ThreadPriority enum has five levels:

- Lowest
- BelowNormal
- Normal (This is the default for all threads)
- AboveNormal
- Highest

The OS scheduler uses this priority as a factor when deciding which thread to run next. A thread with a Highest priority is more likely to get more frequent and/or longer time slices than a thread with a Lowest priority.

Why and When to Use It (With Extreme Caution)

Best Practice: You should almost **never** change a thread's priority from Normal. Manually managing thread priorities is very difficult and can lead to serious problems:

- **Starvation:** If you set a thread's priority too high, it might consume so much CPU time that it "starves" other important threads, including operating system threads, making the entire system unresponsive.
- **Priority Inversion:** A complex scenario where a low-priority thread holds a lock that a high-priority thread needs, but the low-priority thread never gets scheduled to run and release the lock, effectively blocking the high-priority thread forever.

Instead of tweaking priorities, it is always better to design a correct and efficient concurrent architecture using proper synchronization mechanisms.

Interview Perspective

- **Question:** "When should you change a thread's priority?"
- **Ideal Answer:** "In general, you should avoid changing thread priorities. The OS scheduler is very sophisticated and is usually better at managing thread execution than manual adjustments. Changing priorities can lead to issues like thread starvation and priority inversion, which can destabilize an application. It's better to focus on writing efficient, non-blocking code and using proper synchronization than to rely on priority tweaks."

Lecture 10: Thread.Interrupt

Introduction

The thread.Interrupt() method is a mechanism for one thread to interrupt another thread that is in a **blocked or waiting state**. A thread is in this state (the WaitSleepJoin state) if it has called Thread.Sleep(), thread.Join(), or is waiting on a synchronization primitive like Monitor.Wait().

How It Works

When threadB calls threadA.Interrupt(), one of two things happens:

1. If threadA is currently blocked in a waiting state, it will wake up immediately and a **ThreadInterruptedException** will be thrown on threadA.
2. If threadA is currently running (not blocked), the interrupt request is held. The next time threadA enters a waiting state, the exception will be thrown immediately.

Why and When to Use It

This is an older mechanism for implementing **cooperative cancellation**. It's a way to "wake up" a sleeping or waiting thread and signal that it should stop what it's doing and shut down gracefully. The waiting thread is expected to have a try...catch block to handle the ThreadInterruptedException and perform cleanup.

Modern Practice: For all new code, the CancellationToken and CancellationTokenSource framework is the standard, safer, and more robust way to handle cancellation and should be used instead of Thread.Interrupt().

Example

```
public static void Worker()
{
   try
   {
      Console.WriteLine("Worker thread: Going to sleep for 10 seconds.");
      Thread.Sleep(10000); // Enter a long sleep.
```

```
        Console.WriteLine("Worker thread: Woke up normally."); // This line will be skipped.
    }
    catch (ThreadInterruptedException)
    {
        Console.WriteLine("Worker thread: My sleep was interrupted! Cleaning up and exiting.");
    }
}

// --- In your Main method ---
Thread workerThread = new Thread(Worker);
workerThread.Start();

Console.WriteLine("Main thread: Waiting for 2 seconds before interrupting worker.");
Thread.Sleep(2000);

// Main thread interrupts the sleeping worker thread.
workerThread.Interrupt();
```

Lecture 11: Thread States

Introduction

The ThreadState property of a Thread object allows you to query the current execution state of that thread. The state is represented by the ThreadState enum. This is primarily useful for monitoring and debugging purposes; you should not use it for synchronization, as a thread's state can change rapidly between the time you check it and the time you act on that information.

How It Works: A Bitmask Enum

ThreadState is a bitmask enum, which means a thread can technically be in multiple states at once (e.g., WaitSleepJoin and Background).

Key States Explained

- **Unstarted**: The Thread object has been created, but the Start() method has not yet been called.
- **Running**: The thread has been started and is either currently executing code on a CPU core or is ready and waiting for the OS scheduler to give it a time slice.
- **WaitSleepJoin**: The thread is **blocked**. It is not consuming CPU time and is waiting for something to happen (e.g., Thread.Sleep timer to elapse, another thread to Join, or a lock to be released).
- **Stopped**: The thread has completed execution and has terminated.
- **Background**: This is a special flag, not a state in itself. It determines the thread's effect on the application's lifetime.

Foreground vs. Background Threads

This is a critical distinction for application stability.

- **Foreground Threads (Default):** A .NET application will **not terminate** until all of its foreground threads have completed. If your Main method finishes but a foreground worker thread is still running, the process will hang and continue to run.
- **Background Threads:** You can mark a thread as a background thread by setting its IsBackground property to true (myThread.IsBackground = true;). The application **will exit** as soon as all foreground threads have finished, even if background threads are still running. The runtime will abruptly terminate any running background threads.

Best Practice: Use background threads for work that can be safely abandoned if the application closes (e.g., polling for status updates, logging). Use foreground threads for critical work that must be completed before the application can exit (e.g., saving user data to a file).

Lecture 12: Thread Parameters

Introduction

The basic ThreadStart delegate is for methods that take no parameters. But what if you need to pass initial data or configuration to the new thread when it starts? The traditional way to do this was with the ParameterizedThreadStart delegate.

How It Works: ParameterizedThreadStart

This is a delegate that points to a method that takes a **single parameter of type object**. You can then pass your data to the thread using the thread.Start(data) overload.

The Major Drawback: Lack of Type Safety

This approach is **not type-safe** and is considered poor practice in modern C#. Because the parameter is an object, you must perform an **explicit (and potentially unsafe) cast** inside the thread method to get the data back to its original type. This can lead to run-time errors if the wrong type of data is passed.

Example (Legacy Approach)

```
public static void Worker(object data)
{
   // We must cast the object back to its expected type.
   // This is risky if the wrong data is passed to Start().
   int loopCount = (int)data;

   for (int i = 0; i < loopCount; i++)
   {
      Console.WriteLine($"Worker is on iteration {i}");
   }
}

// --- In your Main method ---
Thread workerThread = new Thread(new
ParameterizedThreadStart(Worker));
// Pass the integer 5 to the thread. It will be boxed into an object.
workerThread.Start(5);
```

Modern Approach: The need for ParameterizedThreadStart has been almost completely eliminated by lambda expressions, which can "capture" variables from the containing scope in a type-safe way.

Lecture 13: Custom Thread Object

Introduction

Before lambda expressions became common, the standard, **type-safe** alternative to ParameterizedThreadStart was to create a custom class to hold both the data and the logic for the worker thread.

How It Works

Instead of trying to pass data directly into the thread's Start method, you encapsulate everything the thread needs into its own object.

1. Create a normal class (e.g., Worker).
2. Add public properties to this class to hold the data the thread needs (e.g., LoopCount, Message).
3. Create an instance method in this class that will perform the work, using the data from its own properties.
4. In your Main method, you create an instance of this Worker class, set its properties, and then create a new Thread that points to the worker.Run() instance method.

Why It's Better

This method is completely **type-safe**. There is no casting from object involved. All the data the thread needs is neatly contained and strongly typed within its own dedicated object.

Example

```
public class Worker
{
   // Data for the thread is stored as strongly-typed properties.
   public int Iterations { get; set; }
   public string Message { get; set; }

   // The work method has access to the instance's properties.
   public void Run()
   {
      for (int i = 0; i < Iterations; i++)
      {
         Console.WriteLine($"{Message} - Iteration {i}");
      }
   }
}

// --- In your Main method ---
// 1. Create and configure the worker object.
Worker myWorker = new Worker { Iterations = 5, Message = "Processing
data" };

// 2. Create a thread that points to the instance's Run method.
Thread workerThread = new Thread(myWorker.Run);
workerThread.Start();
```

Lecture 14: Callback

Introduction

A **callback** is a common programming pattern where you pass a method as an argument to another method. In the context of threading, it's a mechanism that allows a worker thread to notify the creating thread that it has completed its task and, optionally, to pass back a result.

How It Works: Using Delegates

1. **Define a delegate** for the callback method's signature. This delegate defines what the "callback" function must look like. For example, delegate void ResultCallback(string result);.

2. The worker class's constructor will accept this delegate as a parameter and store it in a private field.

3. When the worker thread finishes its long-running task, its very last step is to **invoke** the stored callback delegate, passing the result back.

4. The main thread provides a method that matches the delegate signature and passes it when creating the worker object. This method will be executed when the work is done.

Note: The callback method will execute on the **worker thread's context**, not the main thread's.

Example

```
// 1. Define the delegate for the callback.
public delegate void DownloadCompleteCallback(string fileContent);

public class Downloader
{
    private string _url;
    private DownloadCompleteCallback _callback;

    // 2. The constructor accepts the callback delegate.
    public Downloader(string url, DownloadCompleteCallback callback)
    {
        _url = url;
        _callback = callback;
    }

    // This method will be run on a new thread.
    public void Download()
    {
        Console.WriteLine($"Downloading from {_url}...");
        Thread.Sleep(3000); // Simulate a 3-second download.
        string content = "This is the content of the file.";

        // 3. Invoke the callback to send the result back.
        _callback(content);
    }
}

public class Program
{
    // 4. This is the method the main thread provides as the callback.
```

```
    public static void OnDownloadFinished(string content)
    {
        Console.WriteLine($"Callback executed! Download finished. Content:
'{content}'");
    }

    public static void Main(string[] args)
    {
        Downloader downloader = new Downloader("[http://example.com/
data.txt](http://example.com/data.txt)", OnDownloadFinished);
        Thread downloadThread = new Thread(downloader.Download);
        downloadThread.Start();
        Console.WriteLine("Main thread continues its own work...");
    }
}
```

Lecture 15: Shared Resources

Introduction

A **shared resource** is any piece of data or resource—a class field, a static variable, a list, a file on disk—that can be accessed by **multiple threads concurrently**. While this sharing of data is what makes multi-threading powerful, it is also the primary source of its complexity and danger.

The Problem: Race Conditions

When two or more threads attempt to read, modify, and write back to a shared resource at the same time, a **race condition** can occur. The final state of the resource depends on the unpredictable timing and scheduling of the threads. The thread that "wins the race" and writes its value last will overwrite the work of the other threads, leading to data corruption.

This happens because common operations like incrementing a number (`counter++`) are **not atomic**. At a low level, `counter++` is actually three separate steps:

1. **Read** the current value of `counter` from memory into a CPU register.
2. **Modify** the value in the register (add 1).
3. **Write** the new value from the register back to memory.

A Classic Race Condition Scenario: Imagine a shared counter with a value of 10.

1. **Thread A** reads the value 10.
2. The OS scheduler performs a context switch.
3. **Thread B** reads the value 10.
4. **Thread B** adds 1 (result is 11) and writes 11 back to memory. The counter is now 11.
5. The OS switches back to **Thread A**. Thread A still has the old value 10 that it read earlier.
6. **Thread A** adds 1 (result is 11) and writes 11 back to memory.

The counter should be 12, but because Thread A's work was based on stale data, its result overwrote Thread B's result. One of the increments was lost.

Example: Demonstrating a Race Condition

This code starts two threads that each increment a shared counter 1,000,000 times. The expected final result is 2,000,000, but the actual result will almost always be less due to race conditions.

```
class Program
{
   static int sharedCounter = 0;

   static void Main(string[] args)
   {
      List<Thread> threads = new List<Thread>();
      for (int i = 0; i < 2; i++) // Create 2 threads
      {
         Thread t = new Thread(IncrementCounter);
         threads.Add(t);
         t.Start();
      }

      foreach (Thread t in threads)
      {
         t.Join(); // Wait for both threads to finish
      }

      Console.WriteLine($"Expected result: 2,000,000");
      Console.WriteLine($"Actual result:   {sharedCounter}");
```

```
    }

    static void IncrementCounter()
    {
        for (int i = 0; i < 1000000; i++)
        {
            // This is the CRITICAL SECTION where the race condition occurs.
            sharedCounter++;
        }
    }
}
```

Lecture 16: Thread Synchronization

Introduction

Thread synchronization is the mechanism by which we coordinate the execution of multiple threads to ensure that they access shared resources in a safe and predictable manner. The primary goal of synchronization is to prevent race conditions and ensure data integrity.

The Goal: Mutual Exclusion and Critical Sections

The way we prevent race conditions is by enforcing **mutual exclusion**. This means ensuring that only one thread can access a specific piece of code at any given time. This protected block of code is known as a **critical section**.

The Restroom Analogy

Think of a shared resource as a single-person public restroom, and threads as people who need to use it.

- **The Critical Section:** The restroom itself.
- **The Problem:** If two people try to enter at the same time, it's chaos.
- **The Solution:** A lock on the door.
- **Mutual Exclusion:** A person (thread) who wants to use the restroom must first **acquire the lock**. While they have the lock, the door is locked, and no one else can enter. Other people who arrive must wait in a queue. When

the person is done, they **release the lock**, allowing the next person in the queue to acquire it and enter.

C# provides several "locking" mechanisms, called **synchronization primitives**, to achieve this. The most fundamental of these is the Monitor class, which is exposed through the easy-to-use lock keyword.

Lecture 17: Monitor

Introduction

The **System.Threading.Monitor** class is a low-level, powerful tool in .NET for providing synchronized access to a block of code by using an exclusive lock. While you will more commonly use the lock keyword, it's important to understand that the lock keyword is just a convenient syntax for what the Monitor class is doing behind the scenes.

How It Works: Enter, Exit, and the Lock Object

You need an object to act as the "key" for the lock. This should be a private readonly object field in your class.

The two primary static methods are:

- **Monitor.Enter(object lockObject)**: A thread calls this to acquire an exclusive lock on the lockObject. If another thread already holds the lock, this thread will **block** (wait) until the lock is released.
- **Monitor.Exit(object lockObject)**: The thread that holds the lock calls this to release it, allowing the next waiting thread to acquire it.

The try...finally Pattern (Crucial)

It is absolutely critical that Monitor.Exit is placed in a **finally** block. This guarantees that the lock is always released, even if an exception occurs inside the critical section. Forgetting to release a lock will cause all other threads waiting for it to be blocked forever, a condition known as a **deadlock**.

Example: Fixing the Race Condition

```
class Program
{
   static int sharedCounter = 0;
   // The private object used as the lock "key".
   private static readonly object counterLock = new object();

   // ... Main method is the same ...

   static void IncrementCounter()
   {
      for (int i = 0; i < 1000000; i++)
      {
         // This try...finally block is essential for safety.
         try
         {
            // Acquire the lock. Only one thread can pass this line at a time.
            Monitor.Enter(counterLock);

            // This is now the critical section. It is protected.
            sharedCounter++;
         }
         finally
         {
            // Release the lock, allowing the next thread to enter.
            Monitor.Exit(counterLock);
         }
      }
   }
}
```

Running this code will now correctly produce the result of 2,000,000.

Lecture 18: lock

Introduction

The **lock** keyword is a much cleaner, safer, and the **preferred** C# language feature for achieving mutual exclusion. It simplifies the process of acquiring and releasing a lock on a shared resource.

How It Works: Syntactic Sugar

This is the key insight for interviews. The lock statement is just **syntactic sugar** for the Monitor.Enter and Monitor.Exit pattern wrapped in a try...finally block.

When you write this:

```
lock (counterLock)
{
   // Critical section code...
   sharedCounter++;
}
```

The C# compiler automatically generates this for you:

```
Monitor.Enter(counterLock);
try
{
   // Critical section code...
   sharedCounter++;
}
finally
{
   Monitor.Exit(counterLock);
}
```

Why It's Better

- **Readability:** It is far more concise and clearly expresses the intent of protecting a block of code.
- **Safety:** It makes it **impossible** to accidentally forget to release the lock, as the try...finally block is guaranteed by the compiler. This completely prevents deadlocks caused by unreleased locks.

Best Practice: Always use the lock statement instead of manually using Monitor.Enter/Exit.

Example: Refactoring with lock

```
class Program
{
   static int sharedCounter = 0;
   private static readonly object counterLock = new object();

   // ... Main method is the same ...

   static void IncrementCounter()
   {
      for (int i = 0; i < 1000000; i++)
      {
         // Use the much cleaner and safer lock statement.
         lock (counterLock)
         {
            // This is the critical section. Only one thread
            // can be inside this block at any given time.
            sharedCounter++;
         }
      }
   }
}
```

Lecture 19: ManualResetEvent

Introduction

A **ManualResetEvent** is a synchronization primitive that acts like a gate. It can be in one of two states: **signaled** (the gate is open) or **non-signaled** (the gate is closed). Threads can be made to wait at the closed gate, and another thread can open the gate to release them.

How It Works

It uses three key methods:

- **WaitOne()**: When a thread calls this method, it checks the state of the gate.
    - If the gate is **open (signaled)**, the thread passes through without blocking.
    - If the gate is **closed (non-signaled)**, the thread blocks and enters the WaitSleepJoin state until the gate is opened.
- **Set()**: This method **opens the gate** by changing its state to signaled. This has a crucial effect: it releases **all** threads that are currently blocked on WaitOne().
- **Reset()**: This method **manually closes the gate**, changing its state back to non-signaled. Any subsequent calls to WaitOne() will block again.

The "Manual" part of its name is key: once you open the gate with Set(), it **stays open** until you explicitly close it with Reset().

Why and When to Use It

Use a ManualResetEvent when you need one thread to signal to **one or more** other threads that a one-time event has occurred. It's perfect for scenarios like:

- "Initialization is complete, all worker threads can now proceed."
- "A required resource is now available."
- Signaling the end of a process to multiple listeners.

Example: Waiting for Initialization

Imagine a main thread needs to perform some setup, and several worker threads must wait for that setup to be complete before they start their work.

```
class Program
{
   // Start in the non-signaled (closed) state.
   static ManualResetEvent gate = new ManualResetEvent(false);

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

```
        // Start worker threads. They will immediately block on
gate.WaitOne().
        new Thread(Worker).Start("Worker 1");
        new Thread(Worker).Start("Worker 2");
        new Thread(Worker).Start("Worker 3");

        Console.WriteLine("Main thread is performing initialization...");
        Thread.Sleep(3000); // Simulate 3 seconds of setup work.

        Console.WriteLine("\nInitialization complete! Opening the gate...");
        gate.Set(); // Open the gate. All three workers will be released.
    }

    static void Worker(object id)
    {
        Console.WriteLine($"{id} is waiting at the gate.");
        gate.WaitOne(); // This thread will block here until gate.Set() is called.
        Console.WriteLine($"{id} passed the gate and is now working.");
    }
}
```

Lecture 20: AutoResetEvent

Introduction

An **AutoResetEvent** is another gate-like synchronization primitive, very similar to ManualResetEvent. It also has WaitOne() and Set() methods and can be in a signaled or non-signaled state. However, it has one critical difference in behavior.

How It Works: The Automatic Reset

This is the key distinction. When a thread is waiting on an AutoResetEvent and another thread calls Set() to open the gate, it allows **only one** waiting thread to pass through. Immediately after letting that single thread proceed, the gate **automatically resets** back to the closed (non-signaled) state.

Think of it like a subway turnstile: it unlocks to let one person through, and then immediately locks again. A ManualResetEvent is like a main gate that, once opened, lets everyone rush through until it's manually closed.

Why and When to Use It

Use an AutoResetEvent when you want to release waiting threads **one at a time** in a controlled, sequential manner. It's often used in producer-consumer scenarios where one thread produces a piece of data and then signals exactly one consumer thread to wake up and process it.

Example: A Simple Ping-Pong

Let's create two threads that take turns "pinging" and "ponging" each other.

```
class Program
{
   // Two gates. 'pingGate' starts open, 'pongGate' starts closed.
   static AutoResetEvent pingGate = new AutoResetEvent(true);
   static AutoResetEvent pongGate = new AutoResetEvent(false);

   static void Main(string[] args)
   {
      new Thread(Ping).Start();
      new Thread(Pong).Start();
   }

   static void Ping()
   {
      for (int i = 0; i < 5; i++)
      {
         pingGate.WaitOne(); // Wait for my turn (starts open).
         Console.WriteLine("Ping!");
         pongGate.Set(); // Open the gate for Pong's turn.
      }
   }

   static void Pong()
   {
      for (int i = 0; i < 5; i++)
      {
         pongGate.WaitOne(); // Wait for my turn (starts closed).
         Console.WriteLine("    Pong!");
         pingGate.Set(); // Open the gate for Ping's turn.
      }
   }
}
```

Lecture 21: Wait and Pulse

Introduction

Monitor.Wait(), Monitor.Pulse(), and Monitor.PulseAll() are low-level methods on the Monitor class that allow threads to coordinate more advanced interactions while they are **inside a locked region**. They are used to build producer-consumer patterns where a thread needs to wait for a specific condition to become true before it can proceed.

How It Works: The Lock Token Exchange

This pattern is complex and must be understood precisely.

1. A "consumer" thread must first acquire a lock using lock(lockObject).
2. Inside the lock, it checks a condition (e.g., while (queue.Count == 0)).
3. If the condition isn't met, it calls **Monitor.Wait(lockObject)**. This method does two things atomically: it **temporarily releases the lock** and puts the current thread into a special waiting queue for that lock. Releasing the lock is crucial, as it allows other threads (like the producer) to enter the critical section.
4. Another "producer" thread acquires the same lock. It does some work (e.g., adds an item to the queue).
5. The producer then calls **Monitor.Pulse(lockObject)** to wake up **one** waiting thread, or **Monitor.PulseAll(lockObject)** to wake up **all** waiting threads.
6. The woken-up consumer thread does not run immediately. It is moved from the waiting queue to the "ready" queue. It must now **re-acquire the lock** before it can proceed from the point where it called Wait.

When to Use It

For building complex, custom producer-consumer queues or other synchronization patterns where threads need to wait for state changes inside a

critical section. In modern C#, higher-level abstractions like BlockingCollection<T> are often a better and safer choice for these scenarios.

Example: A Simple Producer-Consumer Queue

```
class Program
{
    static Queue<string> _tasks = new Queue<string>();
    static readonly object _lock = new object();

    static void Main(string[] args)
    {
        new Thread(Consumer).Start();
        Thread.Sleep(1000); // Give the consumer time to start and wait.

        // Producer adds tasks
        for (int i = 0; i < 5; i++)
        {
            lock (_lock)
            {
                string task = $"Task {i + 1}";
                _tasks.Enqueue(task);
                Console.WriteLine($"Produced: {task}");
                // Pulse the waiting consumer thread to let it know there's work.
                Monitor.Pulse(_lock);
            }
            Thread.Sleep(500);
        }
    }

    static void Consumer()
    {
        while (true)
        {
            lock (_lock)
            {
                // If the queue is empty, release the lock and wait.
                while (_tasks.Count == 0)
                {
                    Console.WriteLine("Consumer is waiting...");
                    Monitor.Wait(_lock); // Releases the lock until pulsed.
                }

                string task = _tasks.Dequeue();
```

```
            Console.WriteLine($"    Consumed: {task}");
        }
        Thread.Sleep(1000);
    }
  }
}
```

Lecture 22: Monitor with ManualResetEvent

Introduction

The Monitor (used via lock) and ManualResetEvent are both synchronization primitives, but they solve different fundamental problems. It's rare to use them together in a single, tight pattern, but it's crucial to understand their distinct roles.

Comparing Their Purpose

- **Monitor / lock**
  - **Purpose:** To enforce **mutual exclusion**.
  - **Question it Answers:** "Is anyone else currently inside this critical section of code?"
  - **Analogy:** A key to a single restroom. Only one thread can hold the key and be inside the room at a time. It's for protecting a block of code or shared data from simultaneous access.
- **ManualResetEvent**
  - **Purpose:** To provide **signaling**.
  - **Question it Answers:** "Has a specific, one-time event that I am waiting for already occurred?"
  - **Analogy:** A starting pistol for a race. Multiple threads can wait at the starting line (WaitOne()). When another thread fires the pistol (Set()), all waiting threads start the race. It's for notifying threads about a change in the application's state.

When to Choose Which

- Choose **lock** when you need to protect a shared variable or a block of code to prevent race conditions during read/write operations. This is the most common synchronization need.
- Choose **ManualResetEvent** when you need one thread to broadcast a notification to multiple other threads, allowing them all to proceed at once after some condition has been met (like application initialization being complete).

Because they solve different problems, you would use them in different parts of your application's logic, not typically to lock the same resource.

Lecture 23: Intro to Concurrent Collections

Introduction

The standard collection classes we've learned about (List<T>, Dictionary<TKey, TValue>) are **not thread-safe**. If multiple threads try to add or remove items from a standard List<T> at the same time, the list's internal state (its internal array and count variable) can become corrupted, leading to lost data or exceptions.

Before concurrent collections were introduced, the only way to solve this was to manually wrap **every single access** to the shared collection with a lock statement. This can be error-prone and can cause performance bottlenecks if many threads are competing for the same single lock.

The **concurrent collections**, found in the System.Collections.Concurrent namespace, are a set of specialized collection classes designed from the ground up to be **thread-safe**. They use highly efficient, low-level synchronization mechanisms (like fine-grained locks and lock-free techniques) internally, allowing multiple threads to add and remove items safely and often more performantly than a manually locked collection.

When to Use Them

You should use a concurrent collection whenever you have a scenario where multiple threads need to produce or consume items from a single, shared collection.

Lecture 24: CSV with Threads

Introduction

A perfect real-world scenario to demonstrate the power of concurrent collections is processing a large data file with multiple threads. Imagine you have a CSV file with millions of rows, and you want to process each row and add a result to a shared list.

The Old Way: lock with List<T>

To do this with a standard List<T>, you would need to wrap every call to results.Add() inside a lock block. This ensures that only one thread can modify the list at a time, preventing corruption.

```
List<string> results = new List<string>();
object listLock = new object();

// Inside each worker thread's method...
string processedData = "some result";
lock (listLock)
{
    results.Add(processedData);
}
```

This works, but the lock can become a bottleneck if additions are very frequent.

The Modern Way: Using a Concurrent Collection

We can replace the List<T> and the manual lock with a thread-safe collection. A **ConcurrentBag<T>** is an excellent choice for this. It is an **unordered** collection designed for scenarios where multiple threads are adding items concurrently.

Example

This example simulates multiple threads processing data and adding it to a shared collection. Notice how the ConcurrentBag removes the need for any explicit locking in our code.

```
// ConcurrentBag is a thread-safe, unordered collection.
ConcurrentBag<string> processedLines = new ConcurrentBag<string>();
string[] linesToProcess = { "line1", "line2", "line3", "line4", "line5", "line6" };

// Create tasks to simulate concurrent processing
Task task1 = Task.Run(() =>
{
    processedLines.Add($"Worker 1 processed: {linesToProcess[0]}");
    processedLines.Add($"Worker 1 processed: {linesToProcess[1]}");
});

Task task2 = Task.Run(() =>
{
    processedLines.Add($"Worker 2 processed: {linesToProcess[2]}");
    processedLines.Add($"Worker 2 processed: {linesToProcess[3]}");
});

Task.WaitAll(task1, task2); // Wait for tasks to finish

Console.WriteLine($"Total items processed: {processedLines.Count}");
// Note: The order of items in a ConcurrentBag is not guaranteed.
```

Other useful concurrent collections include:

- **ConcurrentQueue<T>**: A thread-safe FIFO queue.
- **ConcurrentStack<T>**: A thread-safe LIFO stack.
- **ConcurrentDictionary<TKey, TValue>**: A thread-safe dictionary.

Lecture 25: Semaphore

Introduction

A **Semaphore** is a synchronization primitive that limits the number of threads that can access a particular resource or a pool of resources concurrently. It maintains a "count" of available slots.

How It Works: The Bouncer Analogy

Think of a semaphore like a bouncer at a nightclub that has a maximum capacity of 10 people.

- The semaphore is initialized with a count of 10.
- When a thread wants to "enter the club" (access the resource), it calls **WaitOne()** (or Wait() on SemaphoreSlim). This decrements the count.
- If the count is greater than zero, the thread is allowed in.
- If the count is already zero, the thread must wait outside until someone leaves.
- When a thread is finished with the resource, it calls **Release()**, which increments the count and allows one waiting thread to enter.

When to Use It

Use a semaphore to control access to a **limited pool of resources**.

- Limiting access to a fixed number of database connections.
- Throttling the number of concurrent web service calls.
- Controlling access to a limited number of software licenses.

The modern, lightweight version for use within a single application is **SemaphoreSlim**.

Example: Throttling Database Connections

```
// This semaphore will only allow 3 threads access at a time.
static SemaphoreSlim dbConnectionPool = new SemaphoreSlim(3);

static void Main(string[] args)
{
   // Start 10 threads that all want to access the database.
   for (int i = 1; i <= 10; i++)
   {
      new Thread(AccessDatabase).Start(i);
   }
}

static void AccessDatabase(object id)
{
   Console.WriteLine($"Thread {id} is waiting to access the database...");
```

```
    dbConnectionPool.Wait(); // Wait for an open slot.

    Console.WriteLine($"--> Thread {id} ENTERED the database.");
    Thread.Sleep(2000); // Simulate doing database work.
    Console.WriteLine($"<-- Thread {id} is EXITING the database.");

    dbConnectionPool.Release(); // Release the slot for the next thread.
}
```

Lecture 26: Mutex

Introduction

A **Mutex** (short for **Mutual Exclusion**) is a synchronization primitive that is very similar to a lock, as it is used to ensure that only one thread can access a resource at a time. However, it has one critical difference.

The Key Difference: System-Wide vs. Process-Local

This is a very important interview topic.

- A **lock (Monitor)** is **local to a single process**. It can only be used to synchronize threads within your own running application.
- A **Mutex** is a **system-wide** or **cross-process** synchronization primitive. This means a Mutex can be given a name, and it can be used to synchronize threads across **multiple, different running applications**.

When to Use It

1. **Cross-Process Synchronization:** When you need to protect a shared resource that is accessible by multiple separate processes, such as a specific file on disk that only one application should be writing to at a time.
2. **Enforcing a Single Instance Application:** A very common use case is to ensure that only one instance of your application can run at a time. On startup, the application tries to acquire a named Mutex. If it succeeds, it knows it's the first instance. If it fails, it means another instance already holds the Mutex, and the new instance can then exit.

Example: Basic Usage Pattern

```
// A named Mutex can be shared across processes.
private static Mutex singleInstanceMutex = new Mutex(true,
"MyUniqueAppName123");

// In Main:
// Try to acquire the mutex. The '0' means don't wait.
bool isNewInstance = singleInstanceMutex.WaitOne(0, false);

if (!isNewInstance)
{
   Console.WriteLine("Another instance of this application is already
running. Exiting.");
   return;
}

// ... run application logic ...

// Release the mutex on exit.
singleInstanceMutex.ReleaseMutex();
```

Lecture 27: Thread Pool

Introduction

The **thread pool** is a pool of pre-created, reusable worker threads managed by the .NET runtime. Its purpose is to efficiently execute many short-lived, background tasks without the performance overhead of creating and destroying a new thread for every single task.

The Problem It Solves

Creating a `new Thread()` is an expensive operation for the operating system. It requires allocating memory for the thread's stack and setting up kernel resources. If your application needs to perform hundreds of small background tasks, creating and destroying a new thread for each one is extremely inefficient and will hurt performance.

How It Works

The .NET runtime maintains a set of warm, ready-to-go worker threads.

1. When you want to run a task, you queue it to the thread pool (e.g., using ThreadPool.QueueUserWorkItem).
2. The pool picks up your task and assigns it to the next available worker thread.
3. When the task is done, the thread is **not destroyed**. It is returned to the pool, ready to execute the next task.

This reuse of threads avoids the expensive creation/destruction overhead.

Modern Usage: Task.Run()

Important Best Practice: In all modern C#, you should **never** use ThreadPool.QueueUserWorkItem directly. You should use the **Task Parallel Library (TPL)**, specifically **Task.Run()**.

```
Task.Run(() =>
{
   // This code will be executed on a thread pool thread.
   Console.WriteLine("Hello from the thread pool!");
});
```

Task.Run() uses the thread pool behind the scenes but provides a much richer, more powerful, and easier-to-use API. It returns a Task object that allows you to wait for completion, get return values, and handle exceptions in a structured way.

Lecture 28: CountDownEvent

Introduction

A CountdownEvent is a synchronization primitive used to wait for a specific number of signals to occur. It allows a main thread to start multiple parallel operations and then wait for **all of them** to complete before it proceeds.

How It Works

1. You initialize the CountdownEvent with a count, representing the number of tasks you are waiting for. For example, new CountdownEvent(3).
2. The main thread calls **myEvent.Wait()**. This will block the main thread until the event's internal count reaches zero.
3. Multiple worker threads perform their tasks. Each time a worker finishes its task, it calls **myEvent.Signal()**, which decrements the internal counter by one.
4. When the final worker calls Signal(), the count becomes zero. This automatically releases the main thread that was blocked on Wait().

When to Use It

It is perfect for "fork-join" scenarios, where a main thread "forks" its work out to multiple workers and then needs to "join" them all back together before continuing.

Example

```
// Initialize the event to wait for 3 signals.
static CountdownEvent countdown = new CountdownEvent(3);

static void Main(string[] args)
{
   Console.WriteLine("Main: Starting 3 workers.");
   new Thread(() => Worker("Worker 1", 2000)).Start();
   new Thread(() => Worker("Worker 2", 3000)).Start();
   new Thread(() => Worker("Worker 3", 4000)).Start();

   Console.WriteLine("Main: Waiting for all workers to complete...");
   countdown.Wait(); // Blocks here until the count reaches zero.

   Console.WriteLine("Main: All workers have completed. Proceeding with
final task.");
}

static void Worker(string id, int sleepTime)
{
   Console.WriteLine($"{id} is starting work.");
```

```
    Thread.Sleep(sleepTime);
    Console.WriteLine($"{id} has finished work.");
    countdown.Signal(); // Decrement the counter.
}
```

Lecture 29: Points to Remember

- **Shared Resources Lead to Race Conditions:** This is the core problem of multithreading. Unsynchronized access to shared data will lead to corruption.
- **lock is Your Primary Tool for Mutual Exclusion:** For protecting a critical section of code to prevent race conditions, the `lock` statement is your standard, safe, and go-to solution.
- **Signaling with Events:** Use `ManualResetEvent` to signal a one-time event to many threads simultaneously. Use `AutoResetEvent` to release waiting threads one by one. Use `CountdownEvent` to wait for a fixed number of tasks to complete.
- **Use Concurrent Collections for Shared Data:** For shared lists, dictionaries, etc., always prefer the thread-safe collections in `System.Collections.Concurrent` (like `ConcurrentBag` or `ConcurrentDictionary`) over manually locking a standard collection.
- **Mutex is for Cross-Process Synchronization:** Remember that `Mutex` is the special tool needed when you need to synchronize access to a system-wide resource across different applications, not just different threads in the same application.
- **Task.Run is the Modern Way to Use the Thread Pool:** In modern C#, always use `Task.Run()` to execute work on a background thread. It is a higher-level, more powerful, and safer abstraction than using the `ThreadPool` directly.