Lecture Notes: Understanding Compilation in Programming

1. What is Compilation?

Compilation is the process of transforming human-readable source code into a format that a computer can understand and execute. This is crucial because computers cannot directly interpret high-level programming languages.

Key Points:

- **Source Code:** The human-readable text document, e.g., a *.cs (C#) file.
- **Compilation Process:** Transforms source code into machine-readable code.
- **Compiler:** A program that performs the translation from source code to executable code.

Example:

- **Source Code File:** `Program.cs`
- **Compiled Executable:** `Program.exe`

2. Role of the Compiler

A **compiler** is essential in converting source code into an executable application. It takes the written code and processes it into a format that the computer can execute.

Steps:

1. **Write Code:** Create a *.cs file with the program code.
2. **Build Process:** The compiler translates the source code.
3. **Executable Output:** The result is an executable file, e.g., `Program.exe`.

3. The Compilation Process in Action

Let's walk through the compilation process using Visual Studio:

Steps:

1. **Clean Solution:** Remove previously built files by selecting `Build > Clean Solution`.
2. **Explore Project Folder:** Open the project folder to locate source files and directories (`Bin` and `Obj`).
3. **Build Solution:** Rebuild the project by selecting `Build > Build Solution`.
4. **Output Files:** Check the `Bin > Debug > .net6.0` folder for the compiled files.

Files Generated:

- **Executable File:** `Program.exe`
- **DLL Files:** Dynamic Link Libraries used by the executable.

Image: Visual Studio Compilation Process


Visual Studio Compilation Process

4. Understanding *.dll Files

DLL (Dynamic Link Library) files contain compiled code that can be used by executable files. These files are crucial for the functionality of the application.

Key Points:

- **Purpose:** Contains reusable code that the executable relies on.
- **Dependency:** The executable (`*.exe`) depends on the `*.dll` file. Without the DLL, the executable cannot run.

Example:

- **Compiled DLL File:** `MyLibrary.dll`
- **Executable Dependency:** `Program.exe` uses `MyLibrary.dll`.

5. Summary

To summarize:

- **Compilation** translates source code into machine-readable code.
- **Compiler** is the tool that performs the compilation.
- **Executable Files (exe):** The final application that can run independently.
- **DLL Files:** Contain code that the executable uses, and are essential for the executable to function.

Practical Examples

Example 1: Cleaning and Building a Solution in Visual Studio

```
1. Open Visual Studio.
2. Select Build > Clean Solution.
3. Right-click the project and select "Open Folder in File Explorer".
4. Navigate to the Bin > Debug > .net6.0 folder (empty initially).
5. Select Build > Build Solution.
6. Observe the generated files in the Bin > Debug > .net6.0 folder.
```

Example 2: Running the Executable

```
1. Navigate to the Bin > Debug > .net6.0 folder.
2. Double-click the `Program.exe` file to run the application.
```

Lecture Notes: Understanding Variables in C#

Introduction

In this lecture, we will cover the basics of variables in C#, focusing on two primary data types: `int` and `string`. We will also discuss variable declaration and initialization, their differences, and explore a useful Visual Studio

shortcut for duplicating lines of code.

Detailed Content

1. What are Variables?

Variables are used to store values in a program. Think of them as labeled boxes that can hold data.

Key Points:

- **Name:** Every variable has a name, e.g., `userInput`.
- **Type:** Each variable has a type, e.g., `string` or `int`.
- **Value:** The actual data stored in the variable, e.g., `"A"` for a string type.

2. Two Important C# Types: `int` and `string`

`int`: Represents whole numbers.

- Example: `int number = 4;`

`string`: Represents textual data.

- Example: `string userInput = "ABC";`

3. Variable Declaration and Initialization

Declaration is specifying that a variable of a certain type and name will exist.

- Example: `int number;`

Initialization is assigning an initial value to the variable.

- Example: `number = 4;`

Combined Declaration and Initialization:

- Example: `int number = 4;`

Key Points:

- A variable's type cannot change once declared.
- A variable must be declared and initialized before use.

4. Visual Studio Shortcut for Duplicating Lines

Shortcut: `Ctrl + D`

- This duplicates the line of code where the cursor is located.
- Useful for quickly copying lines of code.

5. Practical Examples

Example 1: Declaring and Initializing Variables

```
string userInput = "A"; // Declaration and initialization
Console.WriteLine(userInput); // Prints: A

userInput = "ABC"; // Modify the value
Console.WriteLine(userInput); // Prints: ABC

int number = 4; // Declare and initialize int variable
Console.WriteLine(number); // Prints: 4

number = 7; // Modify the value
Console.WriteLine(number); // Prints: 7
```

Example 2: Separate Declaration and Initialization

```
int number; // Declare variable
number = 4; // Initialize variable
Console.WriteLine(number); // Prints: 4
```

6. Common Mistakes and Errors

Using Uninitialized Variables:

- Causes a compile-time error.
- Example: `int number; Console.WriteLine(number);` // Error: Use of unassigned local variable 'number'.

Incorrect Type Assignment:

- Causes a compile-time error.
- Example: `int number = "ABC";` // Error: Cannot implicitly convert type 'string' to 'int'.

Declaring Multiple Variables in a Single Line:

- Example: `int a = 1, b = 6;`
- Example: `string name, lastName;`

Lecture Notes: Naming Variables in C#

Detailed Content

1. How to Name Variables

Basic Rules:

- **Reserved Keywords:** Cannot be used as variable names, e.g., `int`, `class`.

- **Allowed Characters:** Letters, digits, and the underscore character (_).
- **First Character:** Cannot be a digit.
- **Case Sensitivity:** C# is case sensitive; `count` and `Count` are different variables.

Examples of Invalid Names:

- Using a reserved keyword: `int int = 5;`
- Starting with a digit: `int 1number = 5;`
- Using invalid characters: `int my-variable = 5;`

Valid Naming Conventions:

- **Underscore Usage:** Allowed but not recommended as the first character.
- **Lower Camel Case:** Preferred convention, e.g., `firstName`, `lastName`.

Examples of Valid Names:

- `int count = 5;`
- `string firstName = "John";`
- `int user_age = 30;`

2. Using Reserved Keywords as Variable Names

To use reserved keywords as variable names, prepend them with the `@` symbol:

- Example: `int @class = 5;`

3. Clean Code Practices

Importance of Meaningful Names:

- Names should be meaningful and precise.
- Avoid abbreviations: `birthday` instead of `bd`, `firstElement` instead of `fe`.
- Be descriptive: Use `firstName`, `middleName`, `lastName` instead of `name1`, `name2`, `name3`.

Example of Poor Naming:

```
string a = "2023";
string b = "06";
string c = "28";
Console.WriteLine(a + "/" + b + "/" + c);
```

Example of Good Naming:

```
string year = "2023";
string month = "06";
string day = "28";
Console.WriteLine(year + "/" + month + "/" + day);
```

Key Points:

- Focus on clean code from the beginning.
- Avoid bad habits by practicing good naming conventions.
- Renaming variables as needed improves code clarity and maintainability.

4. Visual Studio Shortcut for Renaming Variables

Shortcut: `Ctrl + R, R`

- This renames the selected variable throughout the entire codebase.

Example:

- Select the variable with a bad name.
- Press `Ctrl + R, R`.
- Rename the variable and press `Enter`.

Practical Tips for Naming Variables

1. **Avoid Abbreviations:**
    - `birthday` is clearer than `bd`.
    - `firstElement` is better than `fe`.
2. **Be Precise:**
    - Use `firstName`, `middleName`, `lastName` instead of `name1`, `name2`, `name3`.
3. **Take Your Time:**
    - Naming variables is hard even for experienced developers.
    - Martin Fowler: "There are only two hard things in computer science: cache invalidation and naming things."
4. **Use Renaming Feature:**
    - Don't hesitate to rename variables to improve code clarity.
    - Visual Studio's `Ctrl + R, R` shortcut makes renaming easy.

Lecture Notes: Basic C# Operators and Operator Precedence

1. Basic Arithmetic Operators

Operators:

- **Addition (`+`)**: Adds two numbers.
- **Subtraction (`-`)**: Subtracts one number from another.
- **Multiplication (`*`)**: Multiplies two numbers.
- **Division (`/`)**: Divides one number by another.

Example:

```
int a = 10;
int b = 5;

int sum = a + b;         // Addition: 15
int difference = a - b; // Subtraction: 5
int product = a * b;     // Multiplication: 50
int quotient = a / b;    // Division: 2
```

2. Operator Overloading with Different Types

String Concatenation:

- The addition operator (+) can also be used to concatenate strings.

Example:

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName; // "John Doe"
```

Combining Different Types:

- The addition operator (+) can concatenate a string with an integer, resulting in a string.

Example:

```
string message = "The number is ";
int number = 42;
string result = message + number; // "The number is 42"
```

Invalid Operations:

- Other operators, like subtraction (-), cannot be used between strings and integers.

Example:

```
string message = "The number is ";
int number = 42;
string result = message - number; // Compilation error
```

3. Operator Precedence

Precedence Rules:

- Operators have different precedence levels, determining the order of evaluation in expressions.
- Multiplication (*) and division (/) have higher precedence than addition (+) and subtraction (-).

Example:

```
int result = 10 + 5 * 2; // 10 + (5 * 2) = 20
int result = (10 + 5) * 2; // (10 + 5) * 2 = 30
```

String Concatenation with Mixed Operators:

- When combining different operators, the precedence rules apply. For operators with the same precedence, the evaluation is from left to right.

Example:

```
string message = "The result is ";
int a = 10;
int b = 5;
string result = message + a - b; // Compilation error

string result = message + (a - b); // "The result is 5"
```

Using Parentheses:

- Parentheses can be used to control the order of evaluation explicitly.

Example:

```
string message = "The sum is ";
int a = 10;
int b = 5;
string result = message + (a + b); // "The sum is 15"
```

4. Unary Operators

Increment (++) and Decrement (--):

- Increment (++) increases a number by one.
- Decrement (--) decreases a number by one.

Example:

```
int a = 10;
a++; // a is now 11
```

```
int b = 5;
b--; // b is now 4
```

Lecture Notes: The Purpose of the `var` Keyword in C#

1. Explicitly Typed Variables

Definition:

- Explicitly typed variables are declared with a specific type.
- The type must be stated explicitly at the time of declaration.

Example:

```
string name = "Alice";
int age = 30;
```

2. The `var` Keyword

Definition:

- The `var` keyword allows for the declaration of implicitly typed variables.
- The compiler infers the type based on the value assigned at the time of initialization.

Example:

```
var name = "Alice"; // Inferred as string
var age = 30;       // Inferred as int
```

Explanation:

- Even though the types are not explicitly stated, they are still strongly typed.
- The variable `name` is inferred as a `string`, and `age` is inferred as an `int`.
- After initialization, their types cannot be changed.

Example:

```
name = 42; // Compilation error: cannot convert from int to string
```

3. Benefits of Using `var`

Readability and Conciseness:

- Using `var` can make the code cleaner and more concise, especially when the type is obvious from the context.

Example:

```
var greeting = "Hello, world!";
var number = 100;
```

Reducing Redundancy:

- Declaring types explicitly can be redundant when the type is evident from the assigned value.

Example:

```
// Explicit
string message = "Hello, world!";

// Implicit
var message = "Hello, world!";
```

4. Limitations of `var`

Initialization Requirement:

- Variables declared with `var` must be initialized at the time of declaration.
- The compiler needs an initial value to infer the type.

Example:

```
var uninitialized; // Compilation error: implicitly-typed variables must be
initialized
```

Clarity:

- In some cases, explicitly declaring the type can be clearer, especially when dealing with complex types or the result of calculations.

Example:

```
var result = CalculateSomething(); // Not clear what type 'result' is
int result = CalculateSomething(); // Clear that 'result' is an int
```

5. Best Practices

Use `var` for Readability:

- Use `var` when the type is evident from the context or when it makes the code more readable.

Use Explicit Types for Clarity:

- Use explicit types when the type is not immediately clear or when dealing with complex expressions.

Example:

```
// Using var for readability
var firstName = "John";
var age = 25;

// Using explicit types for clarity
Dictionary<int, List<string>> data = GetData();
```

Reading user input from the console

1. **Reading User Input from Console**:
    - Use `Console.ReadLine()` to capture text input from the user.
    - Store the input in a string variable for further use.
2. **Code Snippets**:
    - Code snippets are predefined templates or shortcuts for inserting common code patterns quickly.
    - Example: Instead of `Console.WriteLine`, use `cw` and press Tab twice to insert the snippet.
3. **Role of Warnings**:
    - Warnings are advisory messages from the compiler, suggesting potential issues or improvements in your code.
    - They don't prevent compilation but are good practice to address for cleaner, more efficient code.
    - Example: Nullable reference type warnings may suggest improving type safety in your code.
4. **Understanding Errors vs Warnings**:
    - Errors halt compilation and must be fixed for the program to run.
    - Warnings provide suggestions for improvement but don't stop the program from compiling.

Lecture Notes: The Bool Type and Operators in C#

1. The Bool Type

Definition:

- `bool` is the simplest type in C#.
- It can only take two values: `true` or `false`.

Examples:

```
bool isUserLoggedIn = true; // Explicitly typed
var hasCompletedTask = false; // Implicitly typed
```

Usage:

- Bools are used to represent truth values in conditions and logical expressions.

Example:

```
bool isInputABC = userInput == "ABC";
```

2. Equality and Inequality Operators

Equality Operator (`==`):

- Checks if two values are equal.
- Returns `true` if they are equal, otherwise `false`.

Example:

```
bool isEqual = userInput == "ABC"; // true if userInput is "ABC"
```

Inequality Operator (`!=`):

- Checks if two values are not equal.
- Returns `true` if they are not equal, otherwise `false`.

Example:

```
bool isNotEqual = userInput != "ABC"; // true if userInput is not "ABC"
```

Logical Negation Operator (`!`):

- Negates a boolean value.
- Turns `true` into `false` and `false` into `true`.

Example:

```
bool isNotABC = !(userInput == "ABC"); // true if userInput is not "ABC"
```

3. Comparison Operators

Greater Than (`>`):

- Checks if one value is greater than another.
- Returns `true` if it is, otherwise `false`.

Example:

```
int number = 10;
bool isGreaterThanFive = number > 5; // true
```

Less Than (`<`):

- Checks if one value is less than another.
- Returns `true` if it is, otherwise `false`.

Example:

```
bool isLessThanTen = number < 10; // false
```

Greater Than or Equal To (`>=`):

- Checks if one value is greater than or equal to another.
- Returns `true` if it is, otherwise `false`.

Example:

```
bool isGreaterOrEqualToTen = number >= 10; // true
```

Less Than or Equal To (`<=`):

- Checks if one value is less than or equal to another.
- Returns `true` if it is, otherwise `false`.

Example:

```
bool isLessOrEqualToSix = number <= 6; // false
```

4. The Modulo Operator (`%`)

Definition:

- The modulo operator returns the remainder of the division of two numbers.
- Useful for checking divisibility and determining even or odd numbers.

Example:

```
int remainder = 10 % 3; // remainder is 1
bool isEven = number % 2 == 0; // true if number is even
```

Lecture Notes: The AND and OR Logical Operators in C#

Introduction

In this lecture, we will delve into two fundamental logical operators in C#: the AND (&&) and OR (||) operators. We will also discuss the concept of short-circuiting optimization and the importance of the order in which logical expressions are evaluated. Finally, we will cover best practices for naming boolean variables.

Detailed Content

1. The AND Operator (&&)

Definition:

- The AND operator is used to check if multiple conditions are all true.
- It returns `true` only if all conditions are met.

Example:

```
int number = 7;
bool result = (number > 4) && (number < 9); // true, because 7 is greater than 4
and less than 9
```

2. The OR Operator (||)

Definition:

- The OR operator is used to check if at least one of multiple conditions is true.
- It returns `true` if any one of the conditions is met.

Example:

```
int number = 8;
bool result = (number == 2) || (number > 6); // true, because 8 is greater than 6
```

3. Combining AND and OR Operators

Example:

- You can combine both AND and OR operators in a single expression.
- Use parentheses for clarity, as it helps in understanding the order of evaluation.

```
int number = 15;
bool result = (number == 123) || ((number % 2 == 0) && (number < 20)); // false
```

4. Short-Circuiting Optimization

Definition:

- Short-circuiting is an optimization technique where the evaluation of expressions stops as soon as the result is determined.

OR Operator:

- If the left side of an OR expression is `true`, the right side is not evaluated.
- This saves computation time, especially with complex calculations.

Example:

```
bool isTrue = (number > 5) || PerformComplexCalculation(); //
PerformComplexCalculation() is not called if (number > 5) is true
```

AND Operator:

- If the left side of an AND expression is `false`, the right side is not evaluated.

Example:

```
bool isFalse = (number < 3) && PerformComplexCalculation(); //
PerformComplexCalculation() is not called if (number < 3) is false
```

Best Practice:

- Place lightweight operations on the left side and heavier ones on the right to potentially skip the heavier ones.

5. Naming Boolean Variables

Best Practice:

- Name boolean variables in the form of a question, making it clear that they can be `true` or `false`.

Examples:

```
bool isNumberEven = (number % 2 == 0);
bool hasUserLoggedIn = false;
```

Lecture Notes: Understanding Scopes in C#

1. What is Scope?

Definition:

- Scope defines the part of the program where a particular variable is accessible.
- The variables we've been learning about so far are called local variables.

Types of Variables:

- There are other kinds of variables in C#, but we'll learn about them later in the course.

2. Scope of Local Variables

Example:

```
int userChoice = 1;
if (userChoice == 1)
{
    int number = 10;
    Console.WriteLine(number); // Works fine
}
else
{
    // Console.WriteLine(number); // Error: 'number' does not exist in the current
context
}

Console.WriteLine(userChoice); // Works fine
// Console.WriteLine(number); // Error: 'number' does not exist in the current
context
```

Explanation:

- The variable `userChoice` is declared outside the `if` statement and can be used anywhere after its declaration.
- The variable `number` is declared inside the `if` statement's code block and can only be used within that block.
- The `number` variable is not accessible in the `else` block or outside the `if/else` statement.

3. Understanding Code Blocks

Definition:

- A code block is a piece of code limited by curly braces `{}`.

Scope within Code Blocks:

- Variables declared inside a code block are only accessible within that block.

4. Nested `if` Statements

Example:

```
if (userChoice == 1)
{
    int number = 10;
    if (number > 5)
    {
        Console.WriteLine(number); // Works fine
        Console.WriteLine(userChoice); // Works fine
    }
}
```

Explanation:

- The inner `if` statement is nested within the outer `if` statement.
- The inner `if` statement has access to variables declared in the outer `if` statement (`number`) as well as variables declared outside the `if` statement (`userChoice`).

5. Variable Naming Conflicts

Example:

```
if (userChoice == 1)
{
    int number = 10;
}
else
{
    int number = 20; // Works fine, as this is a different scope
    Console.WriteLine(number); // Outputs: 20
}
```

Explanation:

- Variables with the same name can exist in different scopes independently.
- The `number` variable in the `if` block and the `number` variable in the `else` block are separate and can have different values.

Lecture Notes: Understanding Methods in C#

1. Introduction to Methods

Definition:

- A method is a piece of code that can be executed multiple times by referring to its name.

- Methods can be parameterized and behave differently based on the parameters passed.

Terminology:

- **Method:** Commonly used term in C#. In other programming languages, it might be referred to as a "function."

2. Defining a Void Method

Example:

```
void PrintSelectedOption(string option)
{
    Console.WriteLine("Selected option: " + option);
}
```

Explanation:

- **Method Name:** `PrintSelectedOption` (should start with a capital letter).
- **Return Type:** `void` (indicates the method does not return any value).
- **Parameter List:** `(string option)` (a single parameter of type `string`).

Important Points:

- The body of the method is contained in curly braces `{}`.
- There is no semicolon `;` after the method definition.

3. Calling a Method

Example:

```
PrintSelectedOption("See all TODOs");
```

Explanation:

- The method `PrintSelectedOption` is called with the argument `"See all TODOs"`.
- When called, the method executes the code within its body using the provided argument.

4. Parameters vs. Arguments

Definitions:

- **Parameters:** The names listed in the method definition.
- **Arguments:** The actual values passed to the method.

Example:

- In `void PrintSelectedOption(string option)`, `option` is the parameter.

- In `PrintSelectedOption("See all TODOs")`, `"See all TODOs"` is the argument.

5. Eliminating Code Repetition

Before using methods:

```
if (userInput == "S")
{
    Console.WriteLine("Selected option: See all TODOs");
}
else if (userInput == "A")
{
    Console.WriteLine("Selected option: Add a new TODO");
}
else if (userInput == "R")
{
    Console.WriteLine("Selected option: Remove a TODO");
}
else if (userInput == "E")
{
    Console.WriteLine("Selected option: Edit a TODO");
}
```

After using methods:

```
void PrintSelectedOption(string option)
{
    Console.WriteLine("Selected option: " + option);
}

if (userInput == "S")
{
    PrintSelectedOption("See all TODOs");
}
else if (userInput == "A")
{
    PrintSelectedOption("Add a new TODO");
}
else if (userInput == "R")
{
    PrintSelectedOption("Remove a TODO");
}
else if (userInput == "E")
{
    PrintSelectedOption("Edit a TODO");
}
```

Advantages:

- Code is more concise and easier to maintain.
- Reduces the risk of errors when making changes.

6. Practical Example

Define the Method:

```
void PrintSelectedOption(string option)
{
    Console.WriteLine("Selected option: " + option);
}
```

Use the Method:

```
if (userInput == "S")
{
    PrintSelectedOption("See all TODOs");
}
else if (userInput == "A")
{
    PrintSelectedOption("Add a new TODO");
}
else if (userInput == "R")
{
    PrintSelectedOption("Remove a TODO");
}
else if (userInput == "E")
{
    PrintSelectedOption("Edit a TODO");
}
```

Explanation:

- The method `PrintSelectedOption` is defined once and used multiple times.
- This ensures that the message format is consistent and can be easily changed if needed.

Lecture Notes: Non-Void Methods in C#

Introduction

In this lecture, we will cover the following topics:

1. Defining non-void methods.
2. The purpose of the `return` keyword.
3. Debugging methods.
4. Removing the `else` keyword after an `if` statement under certain conditions.

5. Using the Quick Actions and Refactorings menu.
6. Understanding code refactoring.

Detailed Content

1. Defining Non-Void Methods

Example:

```
int Add(int a, int b)
{
    return a + b;
}
```

Explanation:

- **Method Name:** `Add`.
- **Return Type:** `int` (indicates the method returns an integer value).
- **Parameter List:** `(int a, int b)` (two parameters of type `int`).
- **Body:** Contains the code to be executed when the method is called.

Important Rule:

- A non-void method must always return a value. All possible code paths must lead to a return statement.

Example of Incorrect Method:

```
int Add(int a, int b)
{
    if (a > 10)
    {
        return a + b;
    }
    // Error: Not all code paths return a value
}
```

2. The `return` Keyword

- The `return` keyword is used to return a value from a method.
- The execution of the method ends when the `return` statement is encountered.

Example:

```
int Add(int a, int b)
{
```

```
    return a + b; // Method execution ends here
}
```

3. Debugging Methods

Setting Breakpoints:

- You can set a breakpoint to pause the execution and inspect the values of variables.
- Use `F10` to move to the next line of code.
- Use `F11` to step into a method and debug its execution.

Example:

```
int result = Add(7, 8); // Set a breakpoint here
// Press F11 to step into the Add method
```

4. Removing `else` Keyword

Example:

```
bool IsLong(string input)
{
    if (input.Length > 10)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Refactored Code:

```
bool IsLong(string input)
{
    if (input.Length > 10)
    {
        return true;
    }
    return false; // The else keyword is not needed
}
```

Further Simplification:

```
bool IsLong(string input)
{
    return input.Length > 10;
}
```

Explanation:

- The `return` keyword ends the method execution, so the `else` keyword is not necessary if the `if` block contains a return statement.
- Directly returning the result of the comparison simplifies the code.

5. Quick Actions and Refactorings Menu

Example:

- If you need to define a method that you haven't yet declared, you can use Visual Studio's Quick Actions and Refactorings menu to generate a method template.

Steps:

1. Right-click on the method name (e.g., `IsLong`).
2. Select "Quick Actions and Refactorings" or press `Alt+Enter`.
3. Choose "Generate method 'IsLong'".

Generated Method:

```
bool IsLong(string input)
{
    // TODO: Implement method
}
```

6. Code Refactoring

Definition:

- Refactoring is the process of restructuring existing code without changing its external behavior.

Example:

- Simplifying the `IsLong` method from using `if` statements to directly returning the result of a comparison.

Lecture Notes: Types of Programming Languages and Method Parameters

Statically Typed vs Dynamically Typed Languages

Statically Typed Languages (e.g., C#):

- During compilation, the compiler checks for type mismatches.
- Variables are bound to a specific type and cannot change unless explicitly cast.
- Example:

```
int num = 10;
string name = "John";
// Type errors are caught during compilation
```

Dynamically Typed Languages (e.g., Python):

- Types are resolved at runtime.
- Variables can change types dynamically.
- Example:

```
num = 10
name = "John"
# No type errors at compile-time; types are resolved at runtime
```

Advantages of Static Typing:

- Early detection of type-related errors during compilation.
- Code tends to be more robust and less prone to runtime errors related to type mismatches.

Disadvantages of Static Typing:

- Requires strict adherence to type rules, which can make the language less flexible compared to dynamically typed languages.

Correct Usage of Method Parameters and Return Types

Method Return Type:

- Must match the declared return type in the method signature.
- Example:

```
bool IsGreaterThan(int a, int b)
{
    return a > b; // Correct usage
}
```

Method Parameters:

- Must match the declared parameter types in the method signature.
- Example:

```
void PrintName(string name)
{
    Console.WriteLine(name); // Correct usage
}
```

Type Compatibility:

- You cannot assign a value of one type to a variable of another incompatible type.
- Example:

```
bool result = IsGreaterThan(5, 3); // Correct assignment
// string result = IsGreaterThan(5, 3); // Error: Cannot implicitly convert
type 'bool' to 'string'
```

Runtime Errors vs Compilation Errors

Runtime Errors:

- Occur during the execution of the program.
- Examples include division by zero, null reference exceptions, etc.

Compilation Errors:

- Occur during compilation.
- Examples include syntax errors, type mismatch errors, etc.

Preferable Errors:

- Compilation errors are preferable because they catch potential issues before the program runs, leading to more robust and reliable code.

Video Lecture: Parsing and Converting Strings to Integers

Introduction to Parsing

In this video, we will cover the concept of parsing and specifically how to convert a string representation of a number into an actual integer using C#.

Understanding Console Input

So far, when we've been reading input from the console using `Console.ReadLine()`, we've always received a string as the user's input. For example, if the user types '123', it's treated as a string "123".

Issue with Type Mismatch

Let's say we want the user to input an integer, but when we try to assign the result of `Console.ReadLine()` directly to an `int` variable, it causes a compilation error. This is because `Console.ReadLine()` returns a `string`, not an `int`.

Introducing Parsing

Parsing is the process of converting a string representation of data into its corresponding data type. In C#, to convert a string that represents an integer into an actual `int`, we use the `int.Parse()` method.

Using int.Parse()

Let's demonstrate how `int.Parse()` works:

```
string userInput = Console.ReadLine(); // User inputs "123"
int number = int.Parse(userInput); // Convert string "123" to int
```

Here:

- `userInput` is initially a string containing "123".
- `int.Parse(userInput)` converts this string into the integer `123`, stored in the `number` variable.

Ensuring Proper Input

It's important to note that `int.Parse()` expects the string to be convertible to an integer. If the user enters something that cannot be parsed as an integer (like "ABC"), it will throw a `FormatException`.

Conclusion

`int.Parse()` is a straightforward and effective method for converting strings to integers in C#. It's essential for scenarios where user input needs to be interpreted as numeric data.

Video Lecture: Handling Exceptions in C#

What are Exceptions?

Exceptions are a mechanism in C# (and many other programming languages) to handle runtime errors. When an unexpected or undesired situation occurs during program execution, C# throws an exception to indicate the problem.

Understanding the Example

Let's review the scenario where we encountered an exception:

- We attempted to convert a user-inputted string (which was not a valid integer) into an integer using `int.Parse()`.
- The user inputted a string that couldn't be interpreted as an integer, resulting in a `System.FormatException`.

Exception Handling

Exceptions need to be handled to prevent the application from crashing. If an exception isn't caught and handled, the program terminates abruptly, as we observed.

Common Exceptions

There are numerous types of exceptions in C#, each representing a different kind of runtime error:

- **FormatException**: Occurs when a string cannot be parsed into a specified format (like converting a non-numeric string to an integer).
- **DivideByZeroException**: Occurs when attempting to divide a number by zero.
- **ArgumentNullException**: Occurs when a method is passed a null argument that it doesn't expect.

Handling Exceptions

In future lessons, we'll cover how to handle exceptions properly, ensuring that our applications can recover from errors gracefully. This involves using try-catch blocks to catch exceptions and handle them in a way that allows the program to continue running without crashing.

Learning from Exceptions

When encountering an exception, it's crucial to read the exception message to understand the issue. Often, the message provides clues on how to fix the problem. If unsure, searching the exception message online can provide insights and solutions from other developers who have encountered similar issues. Certainly! Here's how you can structure your README file for the lecture notes on string interpolation in C#:

Lecture Notes: Understanding C# Programming - Lecture X

String Concatenation vs. String Interpolation

String concatenation involves combining strings using the `+` operator, which can lead to cluttered code and potential errors when managing multiple variables. String interpolation, marked by the `$` symbol, allows for embedding variables directly within the string, enhancing readability and maintainability.

Syntax and Usage

To use string interpolation:

- Prefix the string with `$`.
- Enclose variables or expressions in `{ }` within the string.

Example:

```
string name = "John";
int age = 30;
Console.WriteLine($"Hello, my name is {name} and I am {age} years old.");
```

Output: `Hello, my name is John and I am 30 years old.`

Benefits of String Interpolation

- **Readability:** Simplifies code by eliminating concatenation operators.
- **Maintainability:** Facilitates changes to variable names or formatting.
- **Performance:** Optimized by the compiler for efficiency compared to concatenation in some scenarios.

Lecture Notes: Understanding Switch Statements in C#

Overview of Switch Statements

Switch statements offer an alternative to using multiple if/else statements when dealing with scenarios where different actions are taken based on the value of a variable or expression.

Anatomy of a Switch Statement

```
char userChoice = 'A'; // Example variable

switch (userChoice)
{
    case 'S':
        Console.WriteLine("Selected option: Show TODOs");
        break;
    case 'A':
    case 'a': // Handling both uppercase and lowercase 'A'
        Console.WriteLine("Selected option: Add a new TODO");
        break;
    default:
        Console.WriteLine("Invalid choice");
        break;
}
```

- **Expression:** The value or variable inside the parentheses after `switch`.
- **Case Labels:** Each `case` represents a specific value that the expression might match.
- **Default Case:** Executes if none of the `case` values match the expression.
- **Break Statement:** Ends the execution of the switch block; without it, the code would continue to execute the next case.

Handling Case Insensitivity

To handle case insensitivity (e.g., treating 'A' and 'a' the same):

- Include both versions in a single case label, as shown above.

Example Exercise: Converting Points to Grades

Let's create a method `ConvertPointsToGrade` that converts a score into a grade based on the following table:

- 0-3 points: Grade F
- 4-6 points: Grade C
- 7-9 points: Grade B
- 10 points: Grade A
- Other: Invalid input (return '!')

```
public string ConvertPointsToGrade(int points)
{
    switch (points)
    {
        case 0:
        case 1:
        case 2:
        case 3:
            return "F";
        case 4:
        case 5:
        case 6:
            return "C";
        case 7:
        case 8:
        case 9:
            return "B";
        case 10:
            return "A";
        default:
            return "!";
    }
}
```

- **Return Statements:** Directly returns the grade based on the points without needing a variable, utilizing the return statement to exit the switch block.

Lecture Notes: Understanding Characters (Chars) in C#

Welcome to this lecture on chars in C#! In this session, we'll explore what chars are and why they are a suitable choice for representing single characters in your C# programs.

Introduction to Chars

In programming, a **char** (short for character) is a data type that represents a single Unicode character. Unlike strings, which represent sequences of characters, a char can only hold a single character.

Using Chars in C#

```
public char ConvertPointsToGrade(int points)
{
    switch (points)
    {
```

```
            case 0:
            case 1:
            case 2:
            case 3:
                return 'F';
            case 4:
            case 5:
            case 6:
                return 'C';
            case 7:
            case 8:
            case 9:
                return 'B';
            case 10:
                return 'A';
            default:
                return '!';
        }
    }
```

- **Char Syntax:** Chars are enclosed in single quotes (`'`). This differentiates them from strings, which use double quotes (`"`).

Characteristics of Chars

- **Single Character:** Chars are designed to hold a single Unicode character, including letters, digits, symbols, and control characters.
- **Versatility:** They can represent any single character, not just alphabetical letters.

Key Differences from Strings

- **String vs Char:** While strings are collections of characters, chars are singular and more efficient when you only need to store or manipulate a single character.

Conclusion

Chars are essential in C# for representing individual characters efficiently. By using chars instead of strings when appropriate, you optimize memory usage and code clarity, especially when dealing with single-character data.

Lecture Notes: Understanding Loops in C#

Introduction to Loops

Loops are a fundamental concept in programming that allow us to execute a block of code repeatedly until a specified condition is met or for a fixed number of times.

Types of Loops in C#

1. **While Loop:**

- Executes a block of code repeatedly as long as a specified condition is true.

```
while (condition)
{
    // Code to be executed
}
```

2. **Do-While Loop:**
   - Similar to a while loop, but it guarantees at least one execution of the block of code, even if the condition is false initially.

```
do
{
    // Code to be executed
} while (condition);
```

3. **For Loop:**
   - Executes a block of code a specified number of times.

```
for (initialization; condition; increment/decrement)
{
    // Code to be executed
}
```

4. **Foreach Loop:**
   - Iterates over elements in a collection (like arrays or lists).

```
foreach (var item in collection)
{
    // Code to be executed for each item
}
```

Utilizing Loops in Practical Applications

- **Example Scenario:** Implementing a loop in our To-Do list application to repeatedly prompt the user until a valid option is selected.

Conclusion

Loops are essential for automating repetitive tasks and iterating over data structures in C#. Understanding when and how to use each type of loop is crucial for writing efficient and readable code.

Lecture Notes: Understanding the While Loop in C#

In this lecture, we dive into the concept of the `while` loop in C#. The `while` loop executes a block of code repeatedly as long as a specified condition is true. Let's explore its syntax, behavior, and practical application.

Introduction to the While Loop

The `while` loop is used when we want to repeat a block of code multiple times based on a condition.

```
int number = 0; // Initialize an integer variable

while (number < 10)
{
    number++; // Increment the number by 1
    Console.WriteLine($"Number is {number}"); // Output current value of number
}
```

How the While Loop Works

1. **Initialization:**
    - Initialize a variable (`number` in this case) before the loop starts.
2. **Condition:**
    - Define a condition (`number < 10`) inside the parentheses after `while`. The loop continues to execute as long as this condition is true.
3. **Loop Body:**
    - Code inside the curly braces `{ }` is executed repeatedly as long as the condition remains true.
    - In each iteration, `number` is incremented by 1 (`number++`) and its current value is printed to the console.
4. **Exiting the Loop:**
    - Once `number` reaches 10, the condition (`number < 10`) evaluates to false.
    - The loop exits, and execution continues with the code after the loop.

Lecture Notes: Understanding the Do-While Loop in C#

In this lecture, we explore the `do-while` loop in C#, which is similar to the `while` loop but guarantees that the loop body is executed at least once before checking the loop condition.

Introduction to the Do-While Loop

The `do-while` loop structure ensures that the loop body executes at least once, regardless of the initial condition evaluation. This is particularly useful when you want to perform an action before checking if further iterations are necessary.

Syntax and Behavior

```
string word; // Declare a string variable to store user input

do
{
    Console.WriteLine("Please enter a word longer than ten letters:");
    word = Console.ReadLine(); // Read user input
}
while (word.Length <= 10); // Check condition after executing the loop body
```

- **Initialization:**
  - Initialize variables outside the loop if needed (`word` in this case).
- **Loop Body:**
  - The block of code inside `{ }` executes first, ensuring at least one execution.
- **Condition Check:**
  - After executing the loop body, the condition (`word.Length <= 10`) is evaluated.
  - If true, the loop repeats; if false, the loop exits.

Practical Example: Input Validation

- **Scenario:** Asking the user to enter a word longer than ten letters using a `do-while` loop until valid input is provided.

Comparison with While Loop

- **Difference:**
  - In a `while` loop, the condition is checked before entering the loop body.
  - Using a default initialization in a `while` loop may require careful handling to avoid unintended behavior.

Lecture Notes: Understanding the For Loop in C#

Introduction to the For Loop

The `for` loop is typically used to execute a block of code a given number of times. It's useful for scenarios where the number of iterations is known beforehand.

Example: Printing a Message Five Times

Let's start with a simple example where we print a message to the console five times.

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("This is loop iteration number " + i);
}
```

Breakdown of the For Loop

When defining a `for` loop, three sections must be specified:

1. **Initializer Section**:
    - This section is executed only once before the loop starts.
    - It's used to declare and initialize the loop variable.
    - Example: `int i = 0;`
    - The loop variable cannot be accessed outside the loop.
2. **Condition Section**:
    - This section is evaluated before each iteration.
    - If the condition is `true`, the loop body executes.
    - If the condition is `false`, the loop exits.
    - Example: `i < 5;`
3. **Iterator Section**:
    - This section defines what happens after each iteration of the loop.
    - Typically, it increments or decrements the loop variable.
    - Example: `i++` (increments `i` by 1)

Detailed Example

Let's modify the previous example to print the value of `i` with each loop iteration:

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

Execution Flow

1. **Initialization**:
    - `int i = 0;` initializes `i` to `0`.
2. **Condition Check**:
    - `i < 5;` checks if `i` is less than `5`.
    - If `true`, the loop body executes.

3. **Loop Body**:
    - `Console.WriteLine("Loop iteration " + i);` prints the value of `i`.
4. **Iteration**:
    - `i++` increments `i` by 1.
    - The condition is checked again, and the process repeats.

Modifying the Iterator

We can change the iterator to increment `i` by 2 instead of 1:

```
for (int i = 0; i < 5; i += 2)
{
    Console.WriteLine("Loop iteration " + i);
}
```

- **Output**: `0, 2, 4` (only even numbers are printed)

Looping Backward

We can also use the `for` loop to iterate backward:

```
for (int i = 10; i >= 5; i--)
{
    Console.WriteLine("Loop iteration " + i);
}
```

- **Output**: `10, 9, 8, 7, 6, 5` (loop iterates from 10 to 5)

Avoiding Infinite Loops

Be cautious of infinite loops. An infinite loop occurs if the condition always evaluates to `true`:

```
for (int i = 10; i >= 5; i++)
{
    Console.WriteLine("Loop iteration " + i);
}
```

- **Issue**: `i` is incremented instead of decremented, so `i` will never be less than `5`, resulting in an infinite loop.

Key Takeaways

- **Initializer Section**: Declares and initializes the loop variable.
- **Condition Section**: Determines if the loop should continue.

- **Iterator Section**: Defines what happens after each iteration.
- **Flexibility**: Can increment, decrement, or modify the loop variable as needed.
- **Common Use Cases**: Iterating over arrays, collections, or executing repetitive tasks.

Lecture Notes: Using the `break` Keyword in Loops

In this lecture, we will learn about the `break` keyword, an important control statement that allows us to manage the execution flow within loops more effectively.

Introduction to the `break` Keyword

The `break` keyword is used to immediately terminate the execution of a loop, regardless of the loop's condition. It can be very useful in situations where we need to exit a loop based on a specific condition.

Simple Example: Exiting a Loop

Let's start with a basic example where we use a `for` loop to iterate 100 times. We will use the `break` keyword to stop the loop when a condition is met:

```
for (int i = 0; i < 100; i++)
{
    if (i > 25)
    {
        break;
    }
    Console.WriteLine("Loop iteration " + i);
}
```

Explanation

- The loop is set to iterate 100 times.
- An `if` condition checks if `i` is greater than 25.
- When `i` becomes 26, the `break` statement is executed, terminating the loop immediately.
- **Output**: Numbers from 0 to 25 are printed to the console.

Practical Example: User Input Validation

We want to ask the user to provide a number larger than ten. If the user enters a number that is ten or smaller, we will continue asking for input. The loop will only exit when the user provides a number larger than ten.

Using a `do-while` Loop

Since we need to ask the user for a number at least once, the `do-while` loop is appropriate:

```
int number;
do
{
    Console.Write("Enter a number larger than ten: ");
    number = int.Parse(Console.ReadLine());
} while (number <= 10);
```

- The `do` block executes at least once, asking the user for input.
- The condition `number <= 10` is checked after each iteration.
- The loop continues until the user enters a number larger than ten.

Adding a "Stop" Option

To make the loop more flexible, let's add an option for the user to type "stop" to exit the loop:

```
string input;
do
{
    Console.Write("Enter a number larger than ten or type 'stop' to exit: ");
    input = Console.ReadLine();

    if (input == "stop")
    {
        break;
    }

    number = int.Parse(input);
} while (number <= 10);
```

- The loop now accepts both numbers and the word "stop".
- If the user types "stop", the `break` statement terminates the loop immediately.
- **Output**: The loop stops either when the user types "stop" or enters a number larger than ten.

Practical Use of `break`

The `break` keyword can simplify code and enhance readability, especially when dealing with complex conditions or user input scenarios. It provides a clean and straightforward way to exit loops when necessary.

Lecture Notes: Using the `continue` Keyword in Loops

In this lecture, we will learn about the `continue` keyword, which helps control loop execution by skipping the current iteration and moving to the next one. It is useful in scenarios where certain conditions are met, and we want to bypass specific parts of the loop.

Introduction to the `continue` Keyword

The `continue` keyword works similarly to the `break` keyword, but instead of terminating the loop entirely, it skips the current iteration and proceeds with the next one.

Simple Example: Skipping Specific Iterations

Let's say we want to print numbers from 0 to 20, except for those divisible by three. We can achieve this using the `continue` keyword:

```
for (int i = 0; i <= 20; i++)
{
    if (i % 3 == 0)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Explanation

- The loop iterates from 0 to 20.
- The `if` condition checks if `i` is divisible by 3 using the modulo operator (`%`).
- If the condition is met, the `continue` keyword skips the current iteration.
- **Output**: Numbers from 0 to 20 are printed, excluding those divisible by 3.

Practical Example: User Input Validation with `continue`

Previously, we wrote a code segment that asks the user for a number larger than ten or the word "stop" to exit the loop. However, it is prone to errors if the user input is invalid. Let's improve this code by using the `continue` keyword to handle invalid inputs gracefully.

Original Code

```
string input;
do
{
    Console.Write("Enter a number larger than ten or type 'stop' to exit: ");
    input = Console.ReadLine();

    if (input == "stop")
    {
        break;
    }

    int number = int.Parse(input);
} while (number <= 10);
```

Improved Code with `continue`

We will enhance the code to check if the input is a valid number before attempting to parse it. If not, we will use the `continue` keyword to skip to the next iteration:

```
string input;
int number;
do
{
    Console.Write("Enter a number larger than ten or type 'stop' to exit: ");
    input = Console.ReadLine();

    if (input == "stop")
    {
        break;
    }

    if (!input.All(char.IsDigit))
    {
        Console.WriteLine("Invalid input. Please enter a valid number.");
        continue;
    }

    number = int.Parse(input);
} while (number <= 10);
```

- The `input.All(char.IsDigit)` method checks if all characters in the input are digits.
- If the input is not a valid number, a message is displayed, and the `continue` keyword skips the parsing step.
- **Output**: The loop continues until a valid number greater than ten is entered or the word "stop" is typed.

Handling Unassigned Variables

In the improved code, we encountered a compilation error due to the unassigned `number` variable. This happens because the `continue` keyword can bypass the variable assignment. To resolve this, we ensure that `number` is assigned a value before any condition that uses it:

```
string input;
int number = 0; // Initialize the variable to avoid compilation errors
do
{
    Console.Write("Enter a number larger than ten or type 'stop' to exit: ");
    input = Console.ReadLine();

    if (input == "stop")
    {
        break;
    }

    if (!input.All(char.IsDigit))
```

```
    {
        Console.WriteLine("Invalid input. Please enter a valid number.");
        continue;
    }

    number = int.Parse(input);
} while (number <= 10);
```

Lecture Notes: Program Performance and Loops

What is Program Performance?

Program performance is a measure of:

1. **Execution Speed**: How fast a program runs.
2. **Resource Consumption**: How much memory and other resources a program uses.

As programmers, we are responsible for ensuring that our programs run efficiently. Achieving great performance is a complex task and often requires careful consideration of various factors.

Focus on Loops

Loops can significantly impact program performance, especially nested loops, which can lead to a large number of executions.

Example: Nested Loops

Consider the following code:

```
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 12; j++)
    {
        // Code to be executed
    }
}
```

Here, the innermost code is executed 5 * 12 = 60 times. If the code inside the loop is fast, the performance impact may not be noticeable. However, if the code takes time to execute (e.g., 100 milliseconds), the total execution time would be significant (e.g., 6 seconds).

Strategies to Improve Loop Performance

Avoiding Nested Loops

- **General Rule**: Avoid nested loops if possible, especially those nested more than two times.
- **Alternatives**: Use data structures or algorithms that reduce the need for nested loops.

Breaking Loop Execution Early

- **Use Case**: If a loop is searching for an element in a collection, break the loop once the element is found.
- **Benefit**: This avoids unnecessary iterations, improving performance.

```
for (int i = 0; i < collection.Length; i++)
{
    if (collection[i] == target)
    {
        // Element found, break the loop
        break;
    }
}
```

Moving Performance-Heavy Code Outside Loops

- **Example**: Avoid making slow database calls inside the innermost loop.
- **Improvement**: Move such calls to the outermost loop or outside the loop entirely.

```
// Slow database call outside the loop
var data = GetDatabaseData();

for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 12; j++)
    {
        // Use the data retrieved outside the loop
        Console.WriteLine(data);
    }
}
```

Key Takeaways

- **Program Performance**: Measure of execution speed and resource consumption.
- **Loops Impact**: Loops, especially nested loops, can significantly affect performance.
- **Optimization Strategies**:
    - Avoid nested loops when possible.
    - Break loop execution early if the task is completed.
    - Move performance-heavy code outside loops or to outer loops.

Lecture Notes: Arrays in C#

In this lecture, we will learn about arrays, the index from end operator, how to initialize arrays, and the main disadvantage of arrays.

What are Arrays?

An array is the most basic collection type in C#, used to store multiple elements of the same type.

Key Characteristics

- **Fixed Size**: Once created, the size of an array cannot be changed.
- **Type-Specific**: All elements in an array must be of the same type.

Declaring and Initializing Arrays

1. **Declaration**: Specify the type of elements and use square brackets.

```
int[] numbers;
```

2. **Initialization**: Use the `new` keyword to create an instance of the array with a specified size.

```
numbers = new int[3];
```

    - The `new` keyword is used to create instances of classes. Arrays are objects of a class in C#.
    - When an array is created, it is filled with default values. For integers, the default value is `0`.

Accessing Array Elements

- Use indices to access elements.

```
Console.WriteLine(numbers[0]); // Access the first element
```

- Arrays are zero-indexed, meaning the first element is at index `0`, and the last element is at `array length - 1`.

Modifying Array Elements

- Assign values to elements using indices.

```
numbers[0] = 10;
```

- Arrays store elements of a specific type, so you cannot assign a value of a different type.

```
// This will cause a compilation error
numbers[0] = "Hello";
```

Indexing from the End

- Use the caret (^) symbol to access elements from the end.

```
int[] array = { 1, 2, 3, 4, 5 };
Console.WriteLine(array[^1]); // Access the last element
```

Array Initializer

- Simplify array creation and initialization using the array initializer.

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

- The size and type are inferred from the provided elements.

Example: Sum of Array Elements

Calculate the sum of all elements in an array:

```
int[] numbers = [] { 1, 2, 3, 4, 5 };
int sum = 0;

for (int i = 0; i < numbers.Length; i++)
{
    sum += numbers[i];
}

Console.WriteLine(sum); // Output: 15
```

Common Mistakes

- **Index Out of Bounds**: Trying to access an element at an invalid index will cause a runtime error.

```
// This will cause an error because the array length is 5
Console.WriteLine(numbers[5]);
```

- **Using the Wrong Operator in Loops**: Using <= instead of < in loop conditions can cause runtime errors.

```
for (int i = 0; i <= numbers.Length; i++) // Incorrect
{
    Console.WriteLine(numbers[i]); // This will cause an error
}
```

Disadvantage of Arrays

The main disadvantage of arrays is their **fixed size**.

- You must know the number of elements in advance.
- Once created, you cannot add or remove elements dynamically.

Example: Online Store

In an online store scenario, you might need to store items added to a shopping cart. The number of items can vary greatly, making arrays a poor choice. Instead, use collections with flexible sizes like lists.

Lecture Notes: Multidimensional Arrays in C#

In this lecture, we will learn about multidimensional arrays in C#. We will explore how to declare, initialize, and use these arrays, focusing primarily on two-dimensional arrays.

What are Multidimensional Arrays?

In the previous lecture, we learned about single-dimensional arrays, which are like collections of boxes holding values of the same type, each indexed by a single index. Multidimensional arrays extend this concept to multiple dimensions.

Two-Dimensional Arrays

A two-dimensional array can be thought of as a matrix or a grid, where each element is stored under a pair of indices (row and column). For example, a 4x3 two-dimensional array can store 12 elements.

Three-Dimensional Arrays

A three-dimensional array can be visualized as a cube, similar to a Rubik's cube. For example, a 4x5x3 array can store 60 elements.

Declaring and Initializing Multidimensional Arrays

Declaration

To declare a two-dimensional array of characters:

```
char[,] grid;
```

Initialization

1. **Initialize with Size**:

```
grid = new char[4, 3];
```

This creates a 4x3 array, initially filled with default values (for `char`, the default is `'\0'`).

2. **Initialize with Values**:

```
char[,] grid = {
    { 'a', 'b', 'c' },
    { 'd', 'e', 'f' },
    { 'g', 'h', 'i' },
    { 'j', 'k', 'l' }
};
```

This initializes the array with specific values.

Accessing Elements

To access or modify elements, provide two indices (row and column):

```
char element = grid[0, 1]; // Access element at row 0, column 1
grid[2, 2] = 'z'; // Set element at row 2, column 2 to 'z'
```

Printing Multidimensional Arrays

Nested Loops

To print all elements of a two-dimensional array, use nested loops:

```
for (int i = 0; i < grid.GetLength(0); i++) // Outer loop for rows
{
    for (int j = 0; j < grid.GetLength(1); j++) // Inner loop for columns
    {
        Console.Write(grid[i, j] + " ");
    }
    Console.WriteLine(); // New line after each row
}
```

- `GetLength(0)` returns the number of rows.
- `GetLength(1)` returns the number of columns.

Example

```
char[,] grid = {
    { 'a', 'b', 'c' },
    { 'd', 'e', 'f' },
    { 'g', 'h', 'i' },
```

```
    { 'j', 'k', 'l' }
};

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

Explanation of the Loop

1. **Outer Loop**: Iterates over the rows.
2. **Inner Loop**: Iterates over the columns within each row.
3. **GetLength**: Used to get the length of each dimension (number of rows and columns).

Common Mistakes

- **Single Index**: Using a single index with multidimensional arrays will cause a compilation error.

```
char element = grid[0]; // Incorrect, should be grid[0, 0]
```

- **Index Out of Bounds**: Accessing an element with indices outside the array bounds will cause a runtime error.

```
char element = grid[4, 3]; // Incorrect, valid indices are 0-3 for rows and
0-2 for columns
```

Conclusion

Multidimensional arrays are a powerful feature in C# that allow you to store and manipulate complex data structures like matrices or grids. Understanding how to declare, initialize, and use these arrays is essential for handling more advanced programming scenarios.

In the next lecture, we will learn about the last type of loop in C#: the `foreach` loop.

Lecture Notes: Using the `foreach` Loop in C#

In this lecture, we will learn how to use the `foreach` loop. This loop simplifies the process of iterating over collections such as arrays.

Introduction to the `foreach` Loop

We have already learned about arrays, the basic collection type in C#. Often, we need to execute some code for each element in a collection. The `foreach` loop allows us to do this more simply and cleanly than using a traditional `for` loop.

Traditional `for` Loop

To print all elements of an array using a `for` loop:

```
string[] words = { "apple", "banana", "cherry" };

for (int i = 0; i < words.Length; i++)
{
    Console.WriteLine(words[i]);
}
```

This works fine, but it requires using the loop variable `i` to access each element by its index.

Introducing the `foreach` Loop

The `foreach` loop is designed to iterate over collections without needing to manage index variables. Here's how it looks:

```
string[] words = { "apple", "banana", "cherry" };

foreach (string word in words)
{
    Console.WriteLine(word);
}
```

Explanation

- **Syntax**: The `foreach` loop starts with the `foreach` keyword, followed by a variable declaration (`string word`) and the collection to iterate over (`in words`).
- **Iteration**: In each iteration, the variable `word` represents the current element of the array. In the first iteration, it is the first element; in the second iteration, it is the second element, and so on.
- **No Index**: Unlike the `for` loop, the `foreach` loop does not give us access to the index of the elements. This is fine in cases where we don't need the index.

Advantages of `foreach`

- **Readability**: The `foreach` loop is more readable and concise, making the code easier to understand.
- **Safety**: It reduces the risk of errors related to index management, such as off-by-one errors.

Example with `foreach`

Here's the previous example rewritten using `foreach`:

```
string[] words = { "apple", "banana", "cherry" };

foreach (string word in words)
{
    Console.WriteLine(word);
}
```

Use Cases for `foreach`

The `foreach` loop is particularly useful when:

- You need to iterate over all elements in a collection.
- You do not need to modify the collection elements.
- You do not need the index of the elements.

Lecture Notes: Understanding Lists in C#

In this lecture, we will learn about lists, understand how they differ from arrays, and explore some crucial methods we can use with lists.

Introduction to Lists

Arrays vs. Lists

- **Arrays**:
    - Have a fixed size.
    - Once created, the size cannot be changed.
    - Suitable when the size of the collection is known upfront (e.g., coordinates of a triangle, chessboard).
- **Lists**:
    - Have a dynamic size.
    - Elements can be added or removed.
    - Suitable for collections where the size can change (e.g., items in a shopping cart, to-do lists).

Declaring and Initializing a List

To declare and initialize a list of strings:

```
List<string> words = new List<string>();
```

- `List` is a class, so we use the `new` keyword to create it.
- Angle brackets `<string>` specify the type of elements in the list.

Alternatively, we can initialize a list with items:

```
List<string> words = new List<string> { "apple", "banana", "cherry" };
```

Properties and Methods

- **Count Property**:
  - Similar to the `Length` property in arrays, but called `Count` for lists.
  - Example:

```
Console.WriteLine(words.Count); // Output: 3
```

- **Add Method**:
  - Adds an element to the list.
  - Example:

```
words.Add("date");
Console.WriteLine(words.Count); // Output: 4
```

- **Remove Method**:
  - Removes the first occurrence of the specified element.
  - Example:

```
words.Remove("banana");
Console.WriteLine(words.Count); // Output: 3
```

- **RemoveAt Method**:
  - Removes the element at the specified index.
  - Example:

```
words.RemoveAt(0); // Removes the first element
Console.WriteLine(words.Count); // Output: 2
```

- **AddRange Method**:
  - Adds a range of elements to the list.
  - Example:

```

```

```
words.AddRange(new string[] { "elderberry", "fig", "grape" });
Console.WriteLine(words.Count); // Output: 5
```

- **IndexOf Method**:
  - Returns the index of the first occurrence of a specified element.
  - Returns `-1` if the element is not found.
  - Example:

```
int index = words.IndexOf("cherry");
Console.WriteLine(index); // Output: 1
```

- **Contains Method**:
  - Checks if an element exists in the list.
  - Returns a boolean value.
  - Example:

```
bool hasApple = words.Contains("apple");
Console.WriteLine(hasApple); // Output: true
```

- **Clear Method**:
  - Removes all elements from the list.
  - Example:

```
words.Clear();
Console.WriteLine(words.Count); // Output: 0
```

Using `foreach` Loop with Lists

The `foreach` loop is an efficient way to iterate over lists:

```
foreach (string word in words)
{
    Console.WriteLine(word);
}
```

Lecture Notes: Understanding the `out` Keyword in C#

In this lecture, we will learn about the `out` keyword, its purpose, and how to use it. Understanding this concept will help us handle user input more effectively in our to-do list app.

The Need for the `out` Keyword

Scenario

We want to read a number provided by the user and validate if the input is actually a number. Previously, we either:

- Ignored invalid input, letting the app crash if it wasn't parsable to an `int`.
- Checked if all characters in the input were digits.

There is a better way using the `TryParse` method, which we'll cover in the next lecture. First, we need to understand the `out` keyword.

Returning Multiple Values

Sometimes, we need to return more than one result from a method. For example:

- Given an array of numbers, return a list of positive numbers and the count of non-positive numbers.

Example without `out`

```
public List<int> GetPositiveNumbers(int[] numbers)
{
    List<int> result = new List<int>();
    foreach (int number in numbers)
    {
        if (number > 0)
        {
            result.Add(number);
        }
    }
    return result;
}
```

This method returns a list of positive numbers but doesn't provide the count of non-positive numbers.

Using the `out` Keyword

Method Definition with `out`

To also return the count of non-positive numbers:

```
public List<int> GetPositiveNumbers(int[] numbers, out int countOfNonPositive)
{
    List<int> result = new List<int>();
    countOfNonPositive = 0;
```

```
    foreach (int number in numbers)
    {
        if (number > 0)
        {
            result.Add(number);
        }
        else
        {
            countOfNonPositive++;
        }
    }
    return result;
}
```

Calling the Method with `out`

```
int[] numbers = { -1, 2, -3, 4, 0 };
int nonPositiveCount;
List<int> positiveNumbers = GetPositiveNumbers(numbers, out nonPositiveCount);

Console.WriteLine("Positive Numbers: " + string.Join(", ", positiveNumbers));
Console.WriteLine("Count of Non-Positive Numbers: " + nonPositiveCount);
```

Key Points

1. **Parameter Declaration**:
    - In the method signature, the parameter is declared with the `out` modifier.
    - Inside the method, this parameter must be assigned a value before the method returns.
2. **Method Call**:
    - When calling the method, the `out` keyword must be used with the variable passed as the `out` parameter.
    - This variable does not need to be initialized before being passed to the method.
3. **Variable Initialization**:
    - Inside the method, the `out` parameter is treated as an uninitialized variable and must be assigned a value.

Why `out` is Necessary

Without the `out` keyword, passing a variable to a method creates a copy. Any modifications inside the method do not affect the original variable. The `out` keyword allows the method to directly modify the passed variable.

Example without `out`

```
public List<int> GetPositiveNumbers(int[] numbers, int countOfNonPositive)
{
    List<int> result = new List<int>();
    countOfNonPositive = 0;

    foreach (int number in numbers)
    {
        if (number > 0)
        {
            result.Add(number);
        }
        else
        {
            countOfNonPositive++;
        }
    }
    return result;
}

// Calling the method
int[] numbers = { -1, 2, -3, 4, 0 };
int nonPositiveCount = 0;
List<int> positiveNumbers = GetPositiveNumbers(numbers, nonPositiveCount);

Console.WriteLine("Count of Non-Positive Numbers: " + nonPositiveCount); //
Output: 0
```

The output is incorrect because the `countOfNonPositive` variable inside the method is a copy of the original variable.

Lecture Notes: Parsing Strings to Integers with TryParse

In this lecture, we will learn how to parse a string to an integer without risking a runtime error. We'll also cover a useful keyboard shortcut for formatting code.

Parsing Strings to Integers

The Problem with `int.Parse`

Previously, we used the `int.Parse` method to convert a string to an integer. While this method works, it causes a runtime error if the input is not a valid number.

Example of `int.Parse`

```
string input = "abc";
int number = int.Parse(input); // This will throw a runtime error
```

Solution: `TryParse`

The `TryParse` method addresses this issue by:

1. Returning a `bool` indicating whether the parsing was successful.
2. Using an `out` parameter to return the parsed integer if the parsing is successful.

Syntax of `TryParse`

```
string input = "123";
bool isParsed = int.TryParse(input, out int number);

if (isParsed)
{
    Console.WriteLine("Parsed number: " + number);
}
else
{
    Console.WriteLine("Invalid input.");
}
```

Key Points

1. **Return Type**: `TryParse` returns a `bool` indicating success or failure.
2. **Out Parameter**: If parsing is successful, the parsed integer is assigned to the `out` parameter.
3. **Default Value**: If parsing fails, the `out` parameter is set to the default value for integers (0).

Example in Practice

```
string input = "456";
bool isParsed = int.TryParse(input, out int number);

if (isParsed)
{
    Console.WriteLine("Parsed number: " + number);
}
else
{
    Console.WriteLine("Invalid input.");
}
```

Handling User Input

To keep asking the user for a valid number until they provide one, we can use a `do-while` loop.

Example with `do-while` Loop

```
bool isParsed;
int number;

do
{
    Console.WriteLine("Please enter a number:");
    string input = Console.ReadLine();
    isParsed = int.TryParse(input, out number);

    if (!isParsed)
    {
        Console.WriteLine("Invalid input. Please try again.");
    }
} while (!isParsed);

Console.WriteLine("You entered a valid number: " + number);
```

Formatting Code with Keyboard Shortcut

If the formatting of your code gets messy, you can use the `Ctrl+K+D` shortcut in Visual Studio to automatically format it.

Example of Code Formatting

```
// Messy code
Console.WriteLine("Please enter a number:");string input = Console.ReadLine();
bool isParsed = int.TryParse(input, out int number);if (isParsed)
{Console.WriteLine("Parsed number: " + number);}else {Console.WriteLine("Invalid
input.");}

// Formatted code using Ctrl+K+D
Console.WriteLine("Please enter a number:");
string input = Console.ReadLine();
bool isParsed = int.TryParse(input, out int number);
if (isParsed)
{
    Console.WriteLine("Parsed number: " + number);
}
else
{
    Console.WriteLine("Invalid input.");
}
```

Steps to Format Code

1. Select the code you want to format.
2. Press `Ctrl+K+D`.

Summary

- The `TryParse` method is a safer alternative to `int.Parse` for converting strings to integers.
- It returns a `bool` indicating success and uses an `out` parameter to return the parsed integer.
- Use a `do-while` loop to repeatedly ask the user for input until a valid number is provided.
- The `Ctrl+K+D` shortcut in Visual Studio can quickly format your code.

With these tools and techniques, we are now ready to finish the implementation of our to-do list app.

Lecture Notes: Understanding the Need for Object-Oriented Programming

In this lecture, we will analyze the issues in the code of the Todo list app. This analysis will help us understand why object-oriented programming (OOP) is essential. We will also explore procedural programming, its problems, and the concept of anti-patterns, specifically spaghetti code. Finally, we will learn what high-quality code must always be ready for.

Procedural Programming

Structure of the Todo List App

- The program is simple and executes code from top to bottom.
- Methods are created to improve readability and reusability.
- The main logic is centralized in one file, which can become unwieldy as the application grows.

Problems with Procedural Programming

1. **Scalability**: As the application grows, the file size increases, making the code harder to read and maintain.
2. **Limited Access Control**: There is no way to limit access to certain functions, which can be necessary for maintaining critical data.
3. **Complex Main Logic**: Even with functions moved to separate files, the main logic remains long and complex.
4. **No Separation of Concerns**: High-level business decisions and low-level technical details are not separated. This lack of separation can make future changes difficult.
5. **Hard to Adapt to Changes**: Procedural code is not flexible enough to adapt to new requirements or changes in existing requirements, such as switching from console input to a graphical user interface or adding new storage options (local vs. cloud).

Example of Procedural Code Issues

- If a user wants to switch from storing TODOs locally to storing them in the cloud, procedural code would require numerous `if` statements, making the code harder to maintain and extend.

Anti-patterns: Spaghetti Code

- **Definition**: Spaghetti code is a disorganized and tangled codebase that is difficult to follow and maintain.
- **Problems**:
    - Hard to modify a single part without affecting others.

    - Future changes are painful and error-prone.

High-Quality Code Requirements

- Must be easy to modify and extend.
- Should accommodate changes in requirements without major refactoring.
- Needs to separate high-level logic from low-level details for better maintainability and flexibility.

Object-Oriented Programming (OOP)

Introduction to OOP

- **Origin**: Developed in the 1960s and gained popularity in the 1990s.
- **Purpose**: To address the limitations of procedural programming by organizing code into objects that encapsulate data and behavior.
- **Main Concepts**:
    - **Encapsulation**: Bundling data with the methods that operate on that data.
    - **Abstraction**: Hiding complex implementation details and exposing only the necessary parts.
    - **Inheritance**: Creating new classes based on existing ones to promote code reuse.
    - **Polymorphism**: Allowing methods to do different things based on the object they are called on.

Benefits of OOP

1. **Improved Modularity**: Code is organized into discrete objects, making it easier to manage and understand.
2. **Enhanced Reusability**: Objects and classes can be reused across different parts of the application.
3. **Better Maintainability**: Changes in requirements can be managed more efficiently due to the modular nature of OOP.
4. **Scalability**: Applications can grow without becoming unmanageable.

Example of OOP Structure

- **Class Definition**: A blueprint for creating objects (instances).
- **Objects**: Instances of classes that encapsulate data and behavior.
- **Methods**: Functions defined within classes that operate on the object's data.

Summary

- Procedural programming can lead to complex, unmanageable codebases, especially as applications grow.
- Spaghetti code is an example of an anti-pattern that results from procedural programming.
- High-quality code should be modular, maintainable, and adaptable to changes.
- Object-oriented programming addresses these issues by organizing code into objects, promoting better design and maintainability.

By understanding these concepts, we are better prepared to create robust, scalable, and maintainable applications using object-oriented programming.

Lecture Notes: Introduction to Object-Oriented Programming

What is Object-Oriented Programming (OOP)?

Definition:

- Object-oriented programming is a coding paradigm focused on the concept of objects, which contain data and methods.
- These objects can model real-world features like a person or an address, as well as more abstract concepts like database connections.

Key Concepts

Classes, Objects, and Instances

- **Classes**: Blueprints that define what data and methods the objects will contain.
- **Objects**: Instances created from classes. They hold the data defined by the class and can perform the methods defined in the class.
- **Instances**: Specific realizations of a class. Each instance can hold different data but will have the same methods.

Composition of Objects

- **Simple Composition**: An object contains other objects. For example, a `Person` object can contain an `Address` object.
- **Complex Composition**: An object contains a list of other objects. For instance, a `Person` object can have a list of `Address` objects representing different types of addresses (home, work, correspondence).

Methods in Objects

- **Example**: A `Rectangle` object can contain two integers representing the lengths of its sides. It can also have methods to calculate the rectangle's area and circumference.

Abstract Concepts in OOP

- **Example**:
  - `DatabaseConnector` object provides communication with a database.
  - `TextFileReader` object reads files from the computer's memory and returns their content as strings.

Creating Classes and Instances

- **Class Definition**: A class is defined once and can be used to create many instances.
- **Instances**: These can vary in their data but will share the same methods. For example, two `Person` instances will have different names and addresses but will have the same methods like `GetFullName()`.

Examples of Classes

- **Built-in Classes**: Lists, arrays, strings in C# standard library.
- **Custom Classes**: Defined when built-in types are not enough, allowing for more specific and complex data structures.

Structs and Records

- **Other Types**: Besides classes, C# allows defining custom types using structs and records.
- **Technical Differences**: There are differences in how they operate, but for simplicity, we refer to all custom types as classes in a general discussion.

Benefits of Object-Oriented Programming

1. **Modularity**: Code is organized into discrete objects, making it easier to manage and understand.
2. **Maintainability**: Modular code is easier to maintain and modify.
3. **Reusability**: Objects and classes can be reused across different parts of the application.
4. **Flexibility**: OOP provides ways to adjust the behavior of classes to changing needs.
5. **Understandability**: Responsibilities of single classes are simpler to grasp compared to tangled procedural code.
6. **Control**: Better control over who can use specific pieces of code, reducing the likelihood of errors.

Fundamental Concepts of OOP

- **Encapsulation**: Bundling data with the methods that operate on that data.
- **Polymorphism**: Allowing methods to do different things based on the object they are called on.
- **Abstraction**: Hiding complex implementation details and exposing only the necessary parts.
- **Inheritance**: Creating new classes based on existing ones to promote code reuse.

These fundamental concepts will be covered in more detail once we have some hands-on experience with working with classes and objects.

Summary

- **Object-Oriented Programming** is centered around objects, which are instances of classes.
- **Classes** act as blueprints for creating objects, defining their data and methods.
- **Benefits** of OOP include modularity, maintainability, reusability, flexibility, understandability, and control.
- **Fundamental Concepts** of OOP (encapsulation, polymorphism, abstraction, inheritance) are crucial and will be explored further in practical examples.

Lecture Notes: Practical Use of Object-Oriented Programming with DateTime

Introduction to Object-Oriented Programming with DateTime

Objective:

- Understand the practical application of object-oriented programming (OOP) using the DateTime type in C#.

- Learn about constructors and their role in creating instances of classes and structs.

Key Concepts

Objects Containing Data and Methods

- **Objects**: Contain data and methods that can represent both concrete and abstract concepts.
- **Example**: DateTime type represents date and time, providing access to data (like year, month, day) and methods (like adding days or finding the day of the week).

The DateTime Type

- **Use Cases**: Representing dates of birth, subscription dates, user activity timestamps, etc.
- **Capabilities**: Access year, month, day, hour, minute, and additional information like the day of the week. Perform operations like adding days, months, or years.

Creating and Using a DateTime Object

Creating a DateTime Object

- **Syntax**: Use the `new` keyword followed by the type name and parentheses, which call the constructor.
- **Constructor**: A special method used to create new instances of a class or struct. Can take parameters to initialize the instance.

Example:

```
DateTime internationalPizzaDay2022 = new DateTime(2023, 2, 9);
```

- This creates a DateTime object representing February 9, 2023.
- The constructor takes three integer parameters: year, month, and day.

Retrieving Data from a DateTime Object

- **Accessing Data**: Use properties to retrieve the year, month, and day.

Example:

```
Console.WriteLine(internationalPizzaDay2022.Year);  // Outputs: 2023
Console.WriteLine(internationalPizzaDay2022.Month); // Outputs: 2
Console.WriteLine(internationalPizzaDay2022.Day);   // Outputs: 9
```

Using Methods in a DateTime Object

- **Methods**: DateTime objects contain methods to manipulate dates, such as adding years, months, or days.

Example:

```
DateTime nextYear = internationalPizzaDay2022.AddYears(1);
Console.WriteLine(nextYear); // Outputs: 2024-02-09 00:00:00
```

Calculating the Day of the Week

- **Day of the Week**: Use the `DayOfWeek` property to find out what day of the week a date falls on.

Example:

```
Console.WriteLine(internationalPizzaDay2022.DayOfWeek); // Outputs: Thursday
```

Benefits of Object-Oriented Programming

- **Encapsulation**: The DateTime struct encapsulates the complex logic of date manipulation, making it easy to use without knowing the underlying implementation.
- **Reusability**: Once defined, the DateTime struct can be reused in multiple places in the application, reducing redundancy.
- **Maintainability**: Changes to the DateTime logic are made in one place, improving maintainability.
- **Abstraction**: Users of the DateTime type do not need to understand the complex calculations; they only need to know how to use its properties and methods.

Summary

- **Object-Oriented Programming**: Central to OOP is the concept of objects that contain both data and methods.
- **DateTime Type**: A practical example of OOP, providing a way to represent and manipulate dates.
- **Constructors**: Special methods used to create instances of classes or structs.
- **Benefits**: Encapsulation, reusability, maintainability, and abstraction make OOP a powerful paradigm for writing modular and manageable code.

In the next lecture, we will dive deeper into constructors and how they are used to initialize objects in object-oriented programming.

Lecture Notes: Understanding Abstraction in Object-Oriented Programming

Introduction to Abstraction

Objective:

- Learn what abstraction is in object-oriented programming (OOP).
- Understand the benefits of hiding implementation details from the users of a class.

Key Concepts

What is Abstraction?

- **Definition**: Abstraction is a principle in OOP where classes expose only essential data and methods while hiding the underlying details.
- **Purpose**: Simplifies interaction with objects by providing only the necessary functionality and concealing complex implementation details.

Example: The DateTime Type

Using DateTime

- **Essentials Exposed**: The DateTime type allows users to access and manipulate dates and times without needing to know the internal workings.
- **Hidden Details**: How DateTime calculates the day of the week, handles daylight saving time, leap years, and other complexities are hidden from the user.

Real-world Analogy: Driving a Car

- **User Interaction**: Users interact with the car through the steering wheel, pedals, and dashboard.
- **Hidden Mechanisms**: The internal workings of the engine and other components are hidden from the user.

Internal Representation of DateTime

- **dateData**: DateTime internally represents dates as a number called dateData, which is the number of ticks since January 1st, year one. Each tick is 100 nanoseconds.
- **Complexity Hidden**: The conversion of this number into human-readable components (year, month, day, etc.) is handled internally.

Benefits of Hiding Details

1. **Simplicity for Users**: Users can interact with the type without needing to understand the complex logic.
2. **Flexibility for Developers**: Developers can change the internal implementation (e.g., optimizing performance) without affecting users.

Designing Classes with Abstraction

Public Interface

- **Definition**: The public interface includes all data and methods accessible from outside the class.
- **Accessibility**: Internal data and methods are not accessible, ensuring users interact only with the intended interface.

Benefits of a Stable Public Interface

- **Consistency**: The public interface remains stable even if internal implementations change, ensuring backward compatibility.

- **Example**: Changing the engine or battery of a car doesn't change how the car is driven, so user manuals remain the same.

Practical Application

- **Design Principle**: When designing classes, expose only the necessary interface and hide implementation details.
- **Future-Proofing**: This approach makes the code easier to maintain and extend.

Summary

- **Abstraction**: A fundamental OOP principle that simplifies the use of classes by hiding complex implementation details.
- **DateTime Example**: Illustrates how complex logic can be hidden while providing an easy-to-use interface.
- **Public Interface**: Ensures users interact with classes in a consistent way, even if internal details change.

In the next lectures, we will continue to explore other fundamental OOP concepts like encapsulation, inheritance, and polymorphism, building on the foundation of abstraction.

Lecture Notes: Creating Your First C# Class

Introduction to Classes

Objective:

- Define your first C# class.
- Understand what fields are and their default values.
- Learn about the default constructor.

Key Concepts

Defining a Class

- **Class Declaration**: Use the `class` keyword followed by the class name.
- **Syntax**: No semicolons are needed after the class declaration.
- **Naming Convention**: Class names should start with a capital letter.

```
public class Rectangle
{
    // Fields declaration
    public int width;
    public int height;
}
```

Fields of a Class

- **Definition**: Fields are variables that belong to an object of a class.

- **Instance Specific**: Each instance of a class can have different values for its fields.

```
public class Rectangle
{
    public int width;
    public int height;
}
```

Creating an Instance of a Class

- **Instance Creation**: Use the `new` keyword followed by the class name and parentheses.
- **Example**: Creating a `Rectangle` object.

```
Rectangle rect1 = new Rectangle();
```

Default Constructor

- **Definition**: A constructor is a method used to instantiate objects of a class.
- **Default Constructor**: If no constructor is defined, a parameterless constructor is automatically created.
- **Usage**: The default constructor initializes fields to their default values.

```
Rectangle rect1 = new Rectangle();
```

Field Default Values

- **Automatic Initialization**: If fields are not initialized, they are set to their type's default values.
- **Default for int**: The default value for an integer field is `0`.

```
public class Rectangle
{
    public int width;
    public int height;
}
```

Example Code

- **Creating and Using a Rectangle Object**: Demonstrating the default values of fields.

```
public class Rectangle
{
    public int width;
    public int height;
}
```

```
public class Program
{
    public static void Main()
    {
        Rectangle rect1 = new Rectangle();
        Console.WriteLine($"Width: {rect1.width}, Height: {rect1.height}");
    }
}
```

Common Mistakes

- **Semicolons**: No semicolons are needed after class declarations.
- **Initialization**: Ensure fields are initialized either by the default constructor or explicitly to avoid unexpected values.

Example Code Output

- **Expected Output**: Since the fields are not initialized, their default values are printed.

```
Width: 0, Height: 0
```

Compilation Error

- **Access Protection**: An error might occur if the fields are not accessible due to protection level. We'll cover this in the next lecture.

```
// Error: cannot access Rectangle.height due to its protection level
Console.WriteLine(rect1.height);
```

Summary

- **Classes**: Blueprints for creating objects with specific fields.
- **Fields**: Variables belonging to class instances, automatically initialized to default values.
- **Constructors**: Methods to create instances of a class, with a default parameterless constructor provided if none is defined.
- **Next Lecture**: We'll discuss access protection and how to resolve the compilation error related to field access.

By understanding these concepts, you can start creating and working with your own classes in C#, paving the way for more complex object-oriented programming techniques.

Lecture Notes: Data Hiding and Access Modifiers in C#

Introduction

Objective:

- Understand what data hiding is and its benefits.
- Learn about class members and access modifiers.
- Explore the effects of public and private access modifiers.
- Recognize the default access modifier for fields.

Key Concepts

Data Hiding

- **Definition**: Data hiding involves restricting access to certain class members (fields and methods) from outside the class.
- **Purpose**: Ensures that objects maintain valid states by controlling how and when data is modified.
- **Example**: Preventing direct modification of a rectangle's dimensions to ensure they are always positive.

Class Members

- **Definition**: Class members are the components that make up a class, mainly fields (variables) and methods (functions).

Access Modifiers

- **Purpose**: Control the visibility and accessibility of class members.
- **Types**:
  - **Public**: Accessible from any code.
  - **Private**: Accessible only within the same class.
  - **Protected**: Accessible within the same class and by derived class instances (not covered in this lecture).

Default Access Modifier

- **Fields**: If no access modifier is specified, fields are private by default in C#.

Using Access Modifiers

- **Private Fields**: Ensuring fields are private can protect data integrity by preventing external modification.

```
public class Rectangle
{
    private int width;
    private int height;

    // Dummy method to prove internal access
    public void PrintDimensions()
    {
        Console.WriteLine($"Width: {width}, Height: {height}");
    }
}
```

- **Public Fields**: If fields need to be accessed externally, they can be made public.

```
public class Rectangle
{
    public int width;
    public int height;
}
```

Example Code

- **Private Access**: Demonstrating data hiding and internal access.

```
public class Rectangle
{
    private int width;
    private int height;

    public void PrintDimensions()
    {
        Console.WriteLine($"Width: {width}, Height: {height}");
    }
}

public class Program
{
    public static void Main()
    {
        Rectangle rect1 = new Rectangle();
        // Console.WriteLine(rect1.width); // This will cause a compilation error
due to private access
        rect1.PrintDimensions(); // This works because PrintDimensions is public
and can access private fields
    }
}
```

- **Public Access**: Allowing external access to fields.

```
public class Rectangle
{
    public int width;
    public int height;
}

public class Program
{
    public static void Main()
    {
```

```
        Rectangle rect1 = new Rectangle();
        rect1.width = 5;
        rect1.height = 10;
        Console.WriteLine($"Width: {rect1.width}, Height: {rect1.height}");
    }
}
```

Best Practices

- **Data Hiding**: Only make members public if absolutely necessary to reduce the risk of invalid states.
- **Encapsulation**: Use private fields and provide public methods to control access if needed.

```
public class Rectangle
{
    private int width;
    private int height;

    // Public method to set dimensions
    public void SetDimensions(int width, int height)
    {
        if (width > 0 && height > 0)
        {
            this.width = width;
            this.height = height;
        }
        else
        {
            throw new ArgumentException("Width and height must be positive
numbers.");
        }
    }

    // Public method to print dimensions
    public void PrintDimensions()
    {
        Console.WriteLine($"Width: {width}, Height: {height}");
    }
}
```

Summary

- **Data Hiding**: Protects data integrity by restricting external access to class members.
- **Class Members**: Fields and methods that belong to a class.
- **Access Modifiers**: Keywords that control the visibility of class members (`public`, `private`, `protected`).
- **Default Modifier**: Fields are private by default if no access modifier is specified.
- **Best Practices**: Use private fields and public methods to ensure valid object states.

Understanding and utilizing data hiding and access modifiers will help you create robust, maintainable, and secure C# applications.

Lecture Notes: Custom Constructors in C#

Introduction

Objective:

- Define custom constructors in a class.
- Understand the recommended naming conventions for fields.

Key Concepts

Fields and Default Values

- **Fields**: Variables that belong to an object of a class.
- **Default Values**: Fields get initialized with the default value for their type if not explicitly set.
  - For integers, the default value is zero.
- **Regular Variables**: Variables outside classes are uninitialized if not set.

Public Fields and Access Issues

- **Public Fields**: Allow direct access and modification, leading to potential invalid states.
  - Example: Setting a width to a negative value.
- **Abstraction**: Exposing implementation details is not ideal.

Naming Conventions

- **Public Fields**: Should start with a capital letter.
  - Example: `public int Width;`
- **Private Fields**: Should start with an underscore followed by a lowercase letter.
  - Example: `private int _width;`

Custom Constructors

- **Definition**: A constructor initializes an object when it is created.
- **Naming**: Must match the class name.
- **No Return Type**: Unlike other methods, constructors do not have a return type.
- **Invocation**: Can only be called when creating an object.

Creating a Constructor

- **Purpose**: Assign values to fields.
- **Syntax**:

```
public class Rectangle
{
    public int Width;
    public int Height;

    // Constructor
```

```
    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }
}
```

- **Default Constructor**: If no constructor is defined, a parameterless constructor is autogenerated.
  - Defining a custom constructor removes the autogenerated default constructor.

Example Code

- **Class Definition with Constructor**:

```
public class Rectangle
{
    public int Width;
    public int Height;

    // Constructor
    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }
}
```

- **Creating Instances**:

```
public class Program
{
    public static void Main()
    {
        // Creating first rectangle
        Rectangle rect1 = new Rectangle(5, 10);
        Console.WriteLine($"Width: {rect1.Width}, Height: {rect1.Height}");

        // Creating second rectangle
        Rectangle rect2 = new Rectangle(7, 14);
        Console.WriteLine($"Width: {rect2.Width}, Height: {rect2.Height}");
    }
}
```

Field Initialization

- **Inline Initialization**: Fields can be initialized at the point of declaration.

```
public class Rectangle
{
    public int Width = 3;
    public int Height = 4;

    // Constructor
    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }
}
```

- **Constructor Overwrites**: Constructor values overwrite inline initialization values.

```
public class Program
{
    public static void Main()
    {
        Rectangle rect1 = new Rectangle(5, 10);
        Console.WriteLine($"Width: {rect1.Width}, Height: {rect1.Height}");
    }
}
```

Summary

- **Fields**: Initialized with default values if not explicitly set.
- **Naming Conventions**: Public fields start with a capital letter, private fields with an underscore and lowercase letter.
- **Custom Constructors**: Used to initialize objects, named after the class, without a return type, and only called when creating an object.
- **Field Initialization**: Can be done inline or via constructor, with constructor values taking precedence.

Understanding how to define and use custom constructors is essential for creating well-structured and robust C# applications.

Lecture Notes: Top-Level Statements and Code Structure in C#

Introduction

Objective:

- Understand the requirement for C# code to be within classes.
- Learn about top-level statements introduced in .NET 6.

Key Concepts

Code Containment in C#

- **Code within Classes**: In C#, all code must belong to some class. No variables or methods can exist outside a class.
    - This is different from some other languages like C++ where functions can exist outside any type.

Top-Level Statements

- **Introduction in .NET 6**: Top-level statements allow code to be written without explicitly defining a class and a `Main` method.
- **Purpose**: Simplifies the code for beginners by removing boilerplate code.

Traditional Code Structure

- **Pre-.NET 6**: A console application would start with a `Program` class and a `Main` method.

```
public class Program
{
    public static void Main(string[] args)
    {
        // Your code here
    }
}
```

Example Code with Top-Level Statements

- **Simplified Code**: You can write your executable code directly at the top of the file.

```
Console.WriteLine("Hello, World!");
```

- **Behind the Scenes**: The code is automatically placed into a `Program` class and a `Main` method by the compiler.

Custom Types and Separation

- **Defining Custom Types**: If you need to define custom types (classes, structs, etc.), they should be placed below the top-level statements or in separate files.
- **Best Practices**: In real-life projects, it's recommended to place each class in its own file.

Using Top-Level Statements

- **Choice**: Using top-level statements is optional. You can still explicitly define the `Program` class and `Main` method if preferred.
- **Project Templates**: When creating a new project, you can opt to include or exclude top-level statements.

Example: Explicit Program Class

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

Compilation Error Outside Classes

- **Code Outside Classes**: Writing code outside a class or type definition will result in a compilation error.

```
// This will cause a compilation error
void MyMethod()
{
    Console.WriteLine("This is outside any class");
}
```

Benefits of Code within Classes

- **Modular and Maintainable Code**: Forces adherence to the object-oriented paradigm, leading to modular, maintainable code.
- **Reusability**: Provides building blocks that can be easily reused.

Summary

- **Containment**: All C# code must be within classes or structs.
- **Top-Level Statements**: Simplifies initial coding by removing boilerplate code in .NET 6 and later.
- **Flexibility**: Developers can choose to use traditional class and `Main` method definitions.
- **Best Practices**: Maintain modular code by defining custom types in separate files or below top-level statements.

Understanding the structure and organization of C# code, especially with the introduction of top-level statements, helps in writing clear and maintainable programs.

To enhance our Rectangle class with methods for calculating its area and circumference, let's proceed step by step:

Step 1: Define the Rectangle Class

First, define the `Rectangle` class with private fields for `width` and `height` and a constructor to initialize these fields.

```
public class Rectangle
{
    private int _width;
    private int _height;

    public Rectangle(int width, int height)
    {
        _width = width;
        _height = height;
    }
}
```

Step 2: Add Methods to Calculate Area and Circumference

Next, add two methods to calculate the area and circumference of the rectangle. These methods will be public so that they can be accessed from outside the class.

```
public class Rectangle
{
    private int _width;
    private int _height;

    public Rectangle(int width, int height)
    {
        _width = width;
        _height = height;
    }

    public int CalculateArea()
    {
        return _width * _height;
    }

    public int CalculateCircumference()
    {
        return 2 * (_width + _height);
    }
}
```

Step 3: Naming Conventions for Methods

Notice that the methods are named `CalculateArea` and `CalculateCircumference`. It's a good practice in C# to start method names with a verb, indicating an action performed by the method. This helps distinguish methods from properties or fields, which typically start with nouns.

Step 4: Access Modifiers

By default, methods without an explicit access modifier are private. Private methods can only be accessed within the class itself. To allow these methods to be accessed from outside the class, make them public

explicitly.

```
public class Rectangle
{
    private int _width;
    private int _height;

    public Rectangle(int width, int height)
    {
        _width = width;
        _height = height;
    }

    public int CalculateArea()
    {
        return _width * _height;
    }

    public int CalculateCircumference()
    {
        return 2 * (_width + _height);
    }
}
```

Step 5: Using the Rectangle Class

Now, you can create instances of the `Rectangle` class and use its methods to calculate the area and circumference:

```
class Program
{
    static void Main()
    {
        Rectangle rectangle1 = new Rectangle(5, 10);
        Rectangle rectangle2 = new Rectangle(3, 7);

        Console.WriteLine($"Rectangle 1 Area: {rectangle1.CalculateArea()}");
        Console.WriteLine($"Rectangle 1 Circumference:
{rectangle1.CalculateCircumference()}");

        Console.WriteLine($"Rectangle 2 Area: {rectangle2.CalculateArea()}");
        Console.WriteLine($"Rectangle 2 Circumference:
{rectangle2.CalculateCircumference()}");
    }
}
```

Summary

In this example, we covered:

- Defining a class (`Rectangle`) with private fields and a constructor.
- Adding public methods (`CalculateArea` and `CalculateCircumference`) to perform calculations on the class's data.
- Explaining the default access modifier (private) for methods and how to make them accessible (public).
- Demonstrating the naming convention for methods in C# (starting with a verb).

These principles ensure that your code is structured effectively, adhering to best practices in C# programming.

In object-oriented programming, encapsulation and data hiding are closely related but distinct concepts:

Encapsulation

Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, typically a class. The idea is to keep the implementation details of an object hidden from the outside world, while exposing a public interface through which other parts of the program can interact with the object. This helps in organizing code logically and promotes modularity and reusability.

Benefits of Encapsulation:

1. **Modularity:** Each class encapsulates its own functionality, promoting easier maintenance and updates.
2. **Information Hiding:** By controlling access to data (making fields private), encapsulation helps prevent unauthorized access and ensures data integrity.
3. **Ease of Use:** Users of a class only need to know its public interface (methods), not its internal details, making code simpler and less error-prone.

Data Hiding

Data hiding, on the other hand, is a specific aspect of encapsulation. It involves making the internal state (fields or properties) of a class private, so they cannot be directly accessed or modified from outside the class. Instead, access is controlled through public methods (getters and setters) that enforce validation rules or perform necessary actions when data is accessed or modified.

Why Data Hiding Matters:

1. **Maintaining Consistency:** By restricting direct access to data, we can ensure that the data remains in a valid state.
2. **Encouraging Proper Usage:** Users of a class are encouraged to interact with the object through defined methods, reducing the risk of unintended side effects or errors.
3. **Flexibility in Implementation:** Internal details of a class can be changed without affecting other parts of the program, as long as the public interface remains consistent.

Relationship between Encapsulation and Data Hiding

Encapsulation often enables data hiding by design. When fields are private, they are hidden from external classes or modules, and access is mediated through public methods. This promotes better control over how data is manipulated and ensures that changes to the internal representation of data do not break the functionality of other parts of the program.

Conclusion

In summary, while encapsulation bundles data and methods together within a class, data hiding focuses specifically on restricting access to the internal state of an object. Both concepts are essential for creating robust, maintainable, and secure object-oriented programs, but they serve distinct purposes in achieving these goals.

Method overloading in C# allows us to define multiple methods in the same class with the same name but different parameter lists. This enables us to provide variations of a method that perform similar tasks but operate on different types or numbers of parameters. Here's a breakdown of method overloading and its rules:

Method Overloading

Method overloading is the practice of defining multiple methods in the same class with the same name but different parameters. This allows for more intuitive and flexible usage of methods within a class. Overloaded methods can differ in:

- **Number of Parameters:** Methods can have a different number of parameters.
- **Parameter Types:** Methods can have parameters of different types.
- **Parameter Order:** Methods can have parameters of the same types but in a different order.

Rules for Method Overloading

1. **Parameter Signature Must Differ:** Two methods in the same class cannot have the exact same parameter types in the same order. This means that the compiler must be able to distinguish between overloaded methods based on their parameter types or number.
2. **Return Type Overloading Not Allowed:** Overloading based solely on return type is not allowed in C#. Methods must be distinguished by their parameter types or numbers, not by their return types.

Example of Correct Method Overloading

Let's consider the `MedicalAppointment` class example:

```
public class MedicalAppointment
{
    private string patientName;
    private DateTime appointmentDate;

    // Constructor
    public MedicalAppointment(string patientName, DateTime appointmentDate)
    {
        this.patientName = patientName;
        this.appointmentDate = appointmentDate;
    }

    // Reschedule method with DateTime parameter
    public void Reschedule(DateTime newDate)
    {
        this.appointmentDate = newDate;
    }
```

```
    // Overloaded Reschedule method with month and day parameters
    public void Reschedule(int month, int day)
    {
        this.appointmentDate = new DateTime(this.appointmentDate.Year, month,
day);
    }

    // Method to move appointment by months and days
    public void MoveAppointment(int months, int days)
    {
        this.appointmentDate =
this.appointmentDate.AddMonths(months).AddDays(days);
    }
}
```

Explanation

- The `MedicalAppointment` class demonstrates method overloading with two `Reschedule` methods:
  - One takes a `DateTime` parameter to set a new date directly.
  - The other takes an `int` for month and an `int` for day to adjust the appointment date.
- The `MoveAppointment` method demonstrates a different operation that adjusts the appointment date by adding months and days.

Conclusion

Method overloading provides flexibility and improves code readability by allowing methods to share the same name while catering to different parameter requirements. However, it's important to ensure that overloaded methods are distinguishable by their parameter lists to avoid ambiguity. Choosing meaningful names for methods is crucial when overloading to maintain clarity and intent in your code.

Constructors in C# can be overloaded just like methods, allowing us to provide different ways to initialize objects of a class. When overloading constructors, we need to ensure each constructor has a unique parameter list, which distinguishes it from others. Let's explore constructor overloading and how to call one constructor from another using the `this` keyword:

Constructor Overloading

Constructor overloading allows us to define multiple constructors within the same class, each accepting different parameters. This enables flexibility in object initialization based on the provided arguments.

Example of Constructor Overloading

Let's continue with the `MedicalAppointment` class example and add overloaded constructors:

```
public class MedicalAppointment
{
    private string patientName;
    private DateTime appointmentDate;

    // Constructor with both name and appointment date
```

```
    public MedicalAppointment(string patientName, DateTime appointmentDate)
    {
        this.patientName = patientName;
        this.appointmentDate = appointmentDate;
    }

    // Constructor with name only, default appointment date is a week from now
    public MedicalAppointment(string patientName)
        : this(patientName, DateTime.Now.AddDays(7))
    {
        // This constructor calls the other constructor using "this" keyword
        // Sets appointmentDate to a week from now if not provided
    }

    // Constructor with name and days from now
    public MedicalAppointment(string patientName, int daysFromNow)
        : this(patientName, DateTime.Now.AddDays(daysFromNow))
    {
        // This constructor calls the other constructor using "this" keyword
        // Sets appointmentDate to current date plus daysFromNow
    }

    // Other methods of the MedicalAppointment class can follow...
}
```

Explanation

1. **Multiple Constructors:** The `MedicalAppointment` class now has three constructors:
    - One that initializes both `patientName` and `appointmentDate`.
    - One that initializes only `patientName` and defaults `appointmentDate` to a week from the current date.
    - One that initializes `patientName` and calculates `appointmentDate` based on a specified number of days from the current date.
2. **Using `this` Keyword:** In the second and third constructors, the `this` keyword is used to call another constructor from within the same class:
    - `this(patientName, DateTime.Now.AddDays(7))` calls the first constructor with `patientName` and a default appointment date set to a week from now.
    - `this(patientName, DateTime.Now.AddDays(daysFromNow))` calls the first constructor with `patientName` and a calculated appointment date based on `daysFromNow`.

Benefits of Constructor Overloading

- **Code Reusability:** By overloading constructors and using the `this` keyword to delegate initialization logic, we avoid duplicating code and adhere to the DRY (Don't Repeat Yourself) principle.
- **Flexibility:** Users of the class can initialize objects in different ways based on their specific needs, whether providing all parameters or using default values provided by overloaded constructors.

Conclusion

Constructor overloading in C# allows for cleaner, more readable code by providing multiple ways to initialize objects within a class. By using the `this` keyword to call other constructors, we can reuse initialization logic, avoiding redundancy and improving maintainability. This approach aligns with object-oriented principles and best practices in software development.

Expression-bodied methods in C# are a concise and modern way to define methods that consist of a single expression or statement. Here's a summary of how they work and when to use them:

Expression vs Statement

- **Expression:** Evaluates to a value. Examples include arithmetic operations (`1 + 2`), method calls that return a value (`GetString()`), or even a simple variable (`x`).
- **Statement:** Performs an action and does not return a value. Examples include method calls like `Console.WriteLine("Hello")`, conditional statements (`if`), or loops (`for`).

Using Expression-Bodied Methods

In C#, when a method has only one statement or expression, you can use the expression-bodied syntax to make your code more concise:

Syntax:

```
// Regular method
public int Add(int a, int b)
{
    return a + b;
}

// Expression-bodied method
public int Add(int a, int b) => a + b;
```

Benefits:

1. **Conciseness:** Reduces boilerplate code, especially for simple methods.
2. **Readability:** Makes the intent of the method clearer by focusing on what the method returns rather than the mechanics of returning it.
3. **Modern C# Usage:** Embraces newer features of the language that promote cleaner, more maintainable code.

Limitations:

- **Single Expression/Statement:** You can only use expression-bodied syntax when the method contains exactly one expression or statement. If there are multiple lines of code or statements involved, you must use the traditional block syntax (`{ ... }`).

Example of Expression-Bodied Method:

```
public bool IsEven(int number) => number % 2 == 0;
```

When to Use:

- **Short Methods:** For methods that are concise and straightforward, using expression-bodied syntax improves readability without sacrificing clarity.
- **Getters and Setters:** Especially useful for property accessors (`get` and `set` methods) and methods that perform simple calculations or return results directly.

Conclusion:

Expression-bodied methods in C# are a powerful feature that enhances code readability and maintainability, particularly for small methods with clear and direct purposes. By understanding the distinction between expressions and statements, you can leverage this feature effectively in your C# projects to write cleaner and more expressive code.

The "this" keyword in C# serves two primary purposes within a class context: referencing the current instance of the class and disambiguating between instance variables and parameters or local variables. Here's a summary of how it works based on your examples:

1. Referring to the Current Instance

When you use the "this" keyword, you're explicitly referring to the current instance of the class. This is particularly useful in scenarios where you need to pass the current object instance as an argument or use its methods and properties within the class itself.

Example:

```
public class MedicalAppointment
{
    private DateTime appointmentDate;
    private string patientName;

    public MedicalAppointment(DateTime date, string name)
    {
        appointmentDate = date;
        patientName = name;
    }

    public DateTime GetDate()
    {
        return this.appointmentDate;
    }

    public void Reschedule(DateTime newDate)
    {
        this.appointmentDate = newDate;
        PrintDate(); // Using 'this' to pass the current instance to PrintDate
```

```
method
    }

    public void PrintDate()
    {
        Console.WriteLine("Appointment Date: " + this.GetDate());
    }
}
```

In the `Reschedule` method, `this.appointmentDate` refers to the instance field `appointmentDate`, and `this.PrintDate()` ensures that the `PrintDate` method is called on the current instance of `MedicalAppointment`.

2. Disambiguating between Instance Variables and Parameters

Another use case for the "this" keyword is to disambiguate between instance variables and local variables or parameters that have the same name. This situation typically arises when method parameters have the same name as instance fields.

Example:

```
public class MedicalAppointment
{
    private string patientName;

    public MedicalAppointment(string patientName)
    {
        this.patientName = patientName; // Using 'this' to refer to the instance
field
    }

    public void SetPatientName(string patientName)
    {
        this.patientName = patientName; // Disambiguating between parameter and
field
    }
}
```

Here, `this.patientName` inside the constructor and `SetPatientName` method ensures that we are referring to the instance field `patientName`, not the method parameter with the same name.

Best Practices

- **Naming Conventions:** Following standard naming conventions (like prefixing private fields with an underscore _) can reduce the need for using `this` to disambiguate names.
- **Clarity:** While using `this` can make your intentions clear, overusing it where unnecessary might clutter the code. Use it judiciously where it enhances readability and clarifies intent.

Understanding and correctly using the "this" keyword in C# helps maintain code clarity and ensures correct referencing of instance members within class methods and constructors.

Using optional parameters in C# is a powerful feature that allows constructors and methods to be more flexible and easier to use without creating multiple overloads. Here's a summary of how optional parameters work based on your examples:

Using Optional Parameters

1. **Defining Default Values:** Optional parameters allow you to define default values that will be used if no value is provided when calling the constructor or method.

```
public class MedicalAppointment
{
    private string patientName;
    private DateTime appointmentDate;

    // Constructor with optional parameter daysFromNow
    public MedicalAppointment(string patientName, int daysFromNow = 7)
    {
        this.patientName = patientName;
        this.appointmentDate = DateTime.Today.AddDays(daysFromNow);
    }
}
```

   In this example, `daysFromNow` is an optional parameter with a default value of 7. If `daysFromNow` is not provided when creating a `MedicalAppointment`, it defaults to 7 days from today.

2. **Usage of Optional Parameters:** Optional parameters can be omitted when calling the constructor or method, and the default value will be used instead.

```
// Creating a MedicalAppointment with and without specifying daysFromNow
var appointment1 = new MedicalAppointment("John Doe");        // Uses
default daysFromNow (7 days)
var appointment2 = new MedicalAppointment("Jane Smith", 10);  // Specifies
daysFromNow as 10
```

   - `appointment1` will have `patientName` set to "John Doe" and `appointmentDate` set to today + 7 days.
   - `appointment2` will have `patientName` set to "Jane Smith" and `appointmentDate` set to today + 10 days.

3. **Limitations:**

   - Default values must be compile-time constants. Complex expressions that can only be evaluated at runtime cannot be used.
   - Optional parameters must appear after all required parameters. They cannot precede required parameters in the parameter list.

4. **Ambiguity and Best Practices:**
    - When overloading methods or constructors with optional parameters, if a method without optional parameters matches the provided arguments, it takes precedence over the one with optional parameters.
    - It's recommended to use optional parameters sparingly to avoid confusion. If a method has many optional parameters, it might be a sign that the method is trying to do too much and could benefit from refactoring.

Example Scenario:

Consider a scenario where you have constructors for `MedicalAppointment`:

```
public MedicalAppointment(string patientName, int daysFromNow = 7)
{
    this.patientName = patientName;
    this.appointmentDate = DateTime.Today.AddDays(daysFromNow);
}

public MedicalAppointment(string patientName)
{
    this.patientName = patientName;
    this.appointmentDate = DateTime.Today.AddDays(7); // Default 7 days from today
}
```

In this case:

- Calling `new MedicalAppointment("John Doe")` would use the constructor with a single parameter (`patientName`) and default `daysFromNow` to 7 days.
- Calling `new MedicalAppointment("Jane Smith", 10)` explicitly sets `daysFromNow` to 10 days from today.

Using optional parameters can streamline your code and make it more readable when used appropriately. However, it's crucial to ensure that the default values and their usage align with the expected behavior of your class or method.

Validating constructor parameters is crucial to ensure that objects are created with valid data. Let's go through the process of adding validation to constructor parameters and learn about the `nameof` expression and why having public fields can be risky.

Validating Constructor Parameters

To ensure that our `Rectangle` class only accepts valid widths and heights, we can add validation logic to the constructor. Here's an updated version of the `Rectangle` class with validation:

```
public class Rectangle
{
    public int Width { get; private set; }
    public int Height { get; private set; }
```

```
    public Rectangle(int width, int height)
    {
        Width = ValidateParameter(width, nameof(Width));
        Height = ValidateParameter(height, nameof(Height));
    }

    private int ValidateParameter(int value, string paramName)
    {
        if (value <= 0)
        {
            Console.WriteLine($"{paramName} must be positive. Setting default
value to 1.");
            return 1;
        }
        return value;
    }
}
```

`nameof` Expression

The `nameof` expression is a feature in C# that returns the name of a variable, type, or member as a string. It's particularly useful for ensuring that names in your code are consistent and reducing the risk of errors during refactoring.

In the above code, `nameof(Width)` and `nameof(Height)` return the strings `"Width"` and `"Height"` respectively. This way, if we rename the `Width` or `Height` properties in the future, the `nameof` expressions will automatically update, avoiding the need to manually change string literals.

Risks of Public Fields

Having public fields can be risky because it allows external code to modify the fields directly, bypassing any validation logic. This can lead to invalid states in your objects. In our `Rectangle` class, making `Width` and `Height` public would allow anyone to assign negative values to them, even though we validated them in the constructor.

To mitigate this risk, we should make the fields private and provide public properties with private setters or public read-only properties:

```
public class Rectangle
{
    private int width;
    private int height;

    public int Width
    {
        get => width;
        private set => width = ValidateParameter(value, nameof(Width));
    }
```

```
    public int Height
    {
        get => height;
        private set => height = ValidateParameter(value, nameof(Height));
    }

    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }

    private int ValidateParameter(int value, string paramName)
    {
        if (value <= 0)
        {
            Console.WriteLine($"{paramName} must be positive. Setting default
value to 1.");
            return 1;
        }
        return value;
    }
}
```

Summary

- **Validation:** Ensures that objects are created with valid data by adding validation logic in the constructor or properties.
- `nameof` **Expression:** Helps maintain code consistency and reduces errors during refactoring by returning the name of a variable, type, or member as a string.
- **Public Fields Risks:** Public fields can be modified directly, bypassing validation logic. Using private fields with public properties ensures validation is applied consistently.

By following these practices, we can create robust and maintainable classes that enforce their own invariants and protect against invalid states.

Preventing Field Modification with `readonly`

To ensure fields are never set to invalid values, you can use the `readonly` modifier. A `readonly` field can only be assigned at the time of declaration or within a constructor. Once the object is constructed, it is not possible to change the value of a `readonly` field.

Here is an updated version of the `Rectangle` class using `readonly` fields:

```
public class Rectangle
{
    public readonly int Width;
    public readonly int Height;

    public Rectangle(int width, int height)
```

```
    {
        Width = ValidateParameter(width, nameof(Width));
        Height = ValidateParameter(height, nameof(Height));
    }

    private int ValidateParameter(int value, string paramName)
    {
        if (value <= 0)
        {
            Console.WriteLine($"{paramName} must be positive. Setting default
value to 1.");
            return 1;
        }
        return value;
    }
}
```

In this code, the `Width` and `Height` fields are `readonly`, meaning they can only be assigned during object construction and cannot be modified afterward.

Immutable Objects

Making all fields of an object `readonly` makes the whole object immutable. Immutability means that once an object is created, it will never be modified. Immutable objects have various advantages, such as easier reasoning about program behavior and better concurrency properties, because their state cannot change after they are created.

`const` vs. `readonly`

`const`

- A `const` field must be assigned at declaration and cannot be modified afterward.
- The value assigned to a `const` field must be a compile-time constant.
- `const` fields are implicitly static, meaning they belong to the type itself rather than any instance.
- `const` fields are typically used for values that are known at compile time and do not change, such as mathematical constants or fixed configuration values.

Here is an example of using `const`:

```
public class Constants
{
    public const int RectangleSides = 4;
}
```

`readonly`

- A `readonly` field can be assigned either at declaration or within a constructor.
- The value of a `readonly` field can be set at runtime, but once set, it cannot be changed.

- `readonly` fields are used when the value should not change after the object is constructed, but the exact value may not be known until runtime.

Here is an example of using `readonly`:

```
public class Rectangle
{
    public readonly int Width;
    public readonly int Height;

    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }
}
```

Example: `Rectangle` Class with `const` and `readonly`

Here's an example that combines `const` and `readonly` fields:

```
public class Rectangle
{
    public const int Sides = 4; // Known at compile-time, never changes
    public readonly int Width;  // Set at runtime, cannot be changed afterward
    public readonly int Height; // Set at runtime, cannot be changed afterward

    public Rectangle(int width, int height)
    {
        Width = ValidateParameter(width, nameof(Width));
        Height = ValidateParameter(height, nameof(Height));
    }

    private int ValidateParameter(int value, string paramName)
    {
        if (value <= 0)
        {
            Console.WriteLine($"{paramName} must be positive. Setting default
value to 1.");
            return 1;
        }
        return value;
    }
}
```

In this example, `Sides` is a `const` field, and `Width` and `Height` are `readonly` fields. This design ensures that the number of sides of a rectangle is fixed and known at compile-time, while the width and height are validated and set during object construction and cannot be changed afterward.

Understanding the Limitations of Fields and Introducing Methods

In the previous lecture, we made fields `readonly` to avoid the risk of setting them to invalid values. However, what if we want to allow these fields to be assigned again, at least within the `Rectangle` class? Essentially, we want these fields to be public for reading but private for writing.

This approach ensures that developers of this class have full control over how these fields are set and can validate the values they are set to. However, with fields alone, this is not possible.

Let's see how we can address this using methods:

1. **Making Fields Private and Adding Getter Methods**

   First, we make the fields private and create methods to read their values.

```
public class Rectangle
{
    private int height;

    public Rectangle(int height)
    {
        this.height = height;
    }

    public int GetHeight()
    {
        return height;
    }
}
```

   Now, the `height` field is private, and we have a `GetHeight` method to read its value from outside the class. However, setting this field is not possible from the outside as there is no `SetHeight` method.

2. **Adding Setter Methods with Validation**

   We may sometimes want to add extra logic when setting the field's value, such as validating if the value being set is valid. We can do this by adding a setter method:

```
public class Rectangle
{
    private int height;

    public Rectangle(int height)
    {
        this.height = ValidateHeight(height);
    }

    public int GetHeight()
    {
        return height;
```

```
    }

    public void SetHeight(int height)
    {
        this.height = ValidateHeight(height);
    }

    private int ValidateHeight(int height)
    {
        if (height <= 0)
        {
            Console.WriteLine("Height must be positive. Setting default
value to 1.");
            return 1;
        }
        return height;
    }
}
```

In this example, we have added a `SetHeight` method that validates the value before setting it. We removed the `readonly` keyword to allow the field to be modified within the class.

Moving Towards Properties

In many cases, instead of having public fields, it is better to have them private and expose special methods (getter and setter) to manipulate them. These getter and setter methods are so commonly used that the creators of C# introduced properties as a part of the language to simplify this pattern.

Properties in C#

Properties in C# are a way to provide controlled access to the fields of a class. They allow you to define getter and setter methods more concisely. Properties can include logic for validation or other processing whenever a field is read or written.

In the next lecture, we will dive into properties and how they can help us manage the access and modification of fields in a more elegant and concise manner.

Introduction to Properties

In this lecture, we will learn about properties in C#. This is a crucial topic, so let's focus and understand it well. Properties will be used extensively throughout the course, so you will have many opportunities to get familiar with them.

We will cover:

- What properties are and how they are used.
- The concept of backing fields and accessors.
- Differences between fields and properties.
- When to use fields and when to use properties.

Recap: Methods for Getting and Setting Field Values

In the previous lecture, we saw that instead of having a simple public field, it's often better to use methods for getting and setting its value. For example, instead of directly accessing the `height` field, we used `GetHeight` and `SetHeight` methods.

Introducing Properties

Instead of creating getter and setter methods manually, we can use properties. Let's look at how properties are defined and used.

1. **Defining Properties (Old Syntax)**

   Let's start with the old way of defining properties, which shows better what is happening under the hood.

```
public class Rectangle
{
    private int _width;

    public int Width
    {
        get { return _width; }
        set { _width = value; }
    }
}
```

   - `_width` is a private field (backing field).
   - `Width` is a public property with `get` and `set` accessors.
   - The getter returns the value of `_width`.
   - The setter assigns a new value to `_width`.

2. **Adding Validation in the Setter**

   We can add validation logic to the setter.

```
public int Width
{
    get { return _width; }
    set
    {
        if (value <= 0)
        {
            Console.WriteLine("Width must be positive. Setting default value
to 1.");
            _width = 1;
        }
        else
        {
            _width = value;
        }
```

```
    }
}
```

This setter ensures that the width is always positive.

3. **Using Properties**

   From outside the class, properties are used like fields.

```
Rectangle rect = new Rectangle();
rect.Width = 15; // This will call the setter
int width = rect.Width; // This will call the getter
```

4. **Modern Syntax for Properties**

   The modern, shorter syntax for properties can be used if the getter and setter only get and set the backing field.

```
public int Width { get; set; }
```

   The compiler automatically generates the backing field. This is equivalent to the longer syntax shown earlier.

Differences Between Fields and Properties

1. **Conceptual Differences**
   - **Fields** are like variables.
   - **Properties** are like methods used to get or set values of private fields.
2. **Access Modifiers**
   - For properties, we can have different access modifiers for the getter and setter.
   - For fields, this is not possible.

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

   This property can be read publicly but can only be set privately.

3. **Polymorphism**
   - Properties can be overridden in derived classes (like methods).
   - Fields cannot be overridden.

When to Use Fields vs. Properties

- **Fields** should always be private.
- **Properties** should be used to expose components of a class to the outside world.

Summary

- Properties provide a way to control the access and modification of fields.
- They can include logic for validation or other processing.
- Using properties can help avoid many issues associated with directly accessing fields.

In the next lectures, we will dive deeper into properties and explore more advanced topics in object-oriented programming. This will give you a solid understanding of when and how to use properties effectively.

Introduction to Object Initializers and the `init` Accessor

In this video, we will learn about object initializers and the `init` accessor. Object initializers provide a convenient way to set properties when creating an object. The `init` accessor, introduced in C# 9, allows for immutability while still enabling initialization through object initializers.

Understanding Object Initializers

1. **Basic Usage**

   Let's start with a simple `Person` class with two properties: `Name` and `YearOfBirth`.

   ```
   public class Person
   {
       public string Name { get; set; }
       public int YearOfBirth { get; set; }
   }
   ```

   Typically, we would create an instance of this class using a constructor:

   ```
   Person person = new Person();
   person.Name = "John";
   person.YearOfBirth = 1981;
   ```

   Alternatively, we can use an object initializer:

   ```
   Person person = new Person
   {
       Name = "John",
       YearOfBirth = 1981
   };
   ```

   This approach gives the same result as using the constructor followed by property assignments.

2. **Partial Initialization**

   We don't need to set all the properties when using object initializers. If we skip some properties, they will be assigned their default values.

```
Person person = new Person
{
    Name = "John"
};
// YearOfBirth will be set to 0 (default value for int)
```

3. **Using Constructors with Object Initializers**

   We can use object initializers with constructors that take parameters:

```
public Person(string name)
{
    Name = name;
}

Person person = new Person("John")
{
    YearOfBirth = 1981
};
```

   In this case, the constructor sets the `Name` property, and the object initializer sets the `YearOfBirth` property.

   If a property is set in both the constructor and the object initializer, the value from the object initializer takes precedence.

Limitations and the `init` Accessor

1. **Setters are Required**

   Object initializers only work if the properties have public setters. If a setter is not present or is private, object initialization will not work.

```
public class Person
{
    public string Name { get; }
    public int YearOfBirth { get; set; }
}

Person person = new Person
{
    // Name = "John" // This will cause a compilation error
```

```
    YearOfBirth = 1981
};
```

2. **Introducing the `init` Accessor**

   With C# 9, a new accessor called `init` has been introduced. This allows properties to be set during object initialization but not modified afterward.

```
public class Person
{
    public string Name { get; init; }
    public int YearOfBirth { get; init; }
}

Person person = new Person
{
    Name = "John",
    YearOfBirth = 1981
};

// After object creation, the properties cannot be changed
// person.Name = "Doe"; // This will cause a compilation error
```

Choosing Between Object Initializers and Constructors

1. **Constructors**

   Constructors require all parameters to be provided, ensuring that all necessary data is set during object creation.

```
public Person(string name, int yearOfBirth)
{
    Name = name;
    YearOfBirth = yearOfBirth;
}

Person person = new Person("John", 1981);
```

2. **Object Initializers**

   Object initializers allow for more flexibility, as not all properties need to be set. This can make code more readable, especially when there are many properties.

```
Person person = new Person
{
    Name = "John"
```

```
    // YearOfBirth is not set, will default to 0
};
```

Summary

- **Object Initializers**: Provide a convenient way to set properties during object creation. They require public setters.
- `init` **Accessor**: Introduced in C# 9, allows properties to be set during object initialization but not modified afterward, ensuring immutability.
- **Constructors vs. Object Initializers**: Constructors ensure all necessary data is set, while object initializers offer flexibility and readability.

In practice, constructors are used more often for ensuring all required parameters are provided, but object initializers are also useful when flexibility and readability are important.

Introduction to Computed Properties

In this lecture, we will learn to create computed properties and understand when to use them versus parameterless methods. Computed properties do not wrap any field but instead return a result based on some logic.

Creating Computed Properties

1. **Example of a Computed Property**

   Let's add a `Description` property to a `Rectangle` class that returns a string describing the rectangle object.

   ```
   public class Rectangle
   {
       public double Width { get; set; }
       public double Height { get; set; }

       public string Description => $"Rectangle with width {Width} and height
   {Height}.";
   }
   ```

   As you can see, the syntax for creating computed properties is almost the same as for expression-bodied methods. The property returns a string that describes the rectangle, and the result is computed each time the property is accessed.

2. **Performance Considerations**

   If a computed property is accessed multiple times, its value is recalculated each time. For simple logic, this is not an issue, but for more complex calculations, it can cause performance problems.

   ```
   public double ComplexCalculation => /* some complex logic here */;
   ```

Accessing such a property repeatedly can be costly.

Choosing Between Properties and Methods

1. **Properties Represent Data**

   Properties should be used to represent data. They should be quick to access and not involve performance-heavy computations.

   ```
   public double Area => Width * Height;
   public double Circumference => 2 * (Width + Height);
   ```

2. **Methods Represent Actions**

   Methods should be used to represent actions or operations, especially when they involve significant processing.

   ```
   public double CalculateComplexValue()
   {
       // some complex logic here
   }
   ```

   Using methods for performance-heavy calculations sets clear expectations that the operation might take time or have side effects.

3. **Guidelines for Choosing**

   - **Properties**: Use for simple data retrieval without performance concerns.
   - **Methods**: Use for actions or complex calculations.

   Users expect properties to work like fields, i.e., quickly and without causing runtime errors or performance issues. Therefore, avoid putting complex logic into computed properties.

Summary

- **Computed Properties**: Return a result based on some logic and do not wrap fields. They should be quick to access and represent data.
- **Parameterless Methods**: Represent actions and are suitable for performance-heavy computations or operations with side effects.
- **Choosing Between Properties and Methods**: Properties for data and methods for actions. Consider performance and user expectations when deciding.

By following these guidelines, we can ensure our code is efficient, maintainable, and meets user expectations.

Understanding Static Classes and Methods

In this lecture, we will learn about static classes, static methods, and why all `const` fields are implicitly static. We'll discuss their uses, limitations, and how they can optimize your code.

Static Classes and Methods

1. **Stateful vs. Stateless Classes**
   - **Stateful Classes**: Classes like `Rectangle` have instances with different data (e.g., different width and height).
   - **Stateless Classes**: Classes like `Calculator` contain only methods and no instance-specific data. All instances of such classes would behave the same, making instance creation unnecessary.

2. **Static Methods**
   - **Definition**: Static methods belong to the class itself, not to any specific instance. They cannot access instance data (fields or properties).

```
public class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

   - **Calling Static Methods**: Static methods are called on the class itself, not on an instance.

```
int result = Calculator.Add(3, 5);
```

3. **Static Classes**
   - **Definition**: A static class cannot be instantiated and can only contain static members (methods, properties, fields).

```
public static class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

   - **Benefits**: They group utility methods together without needing instances, ensuring no unnecessary object creations, and saving memory.

Static Methods in Non-Static Classes

1. **Adding Static Methods**
    - Non-static classes can contain static methods. These methods are called on the class itself.

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }

    public static double CalculateDiagonal(double width, double height)
    {
        return Math.Sqrt(width * width + height * height);
    }
}
```

    - **Calling Static Methods**: Use the class name, not an instance.

```
double diagonal = Rectangle.CalculateDiagonal(3, 4);
```

2. **Access Restrictions**
    - Static methods cannot access instance fields or methods because they do not belong to any specific instance.

```
public static void SomeStaticMethod()
{
    // Error: Cannot access instance members from static method
    // double area = Width * Height;
}
```

The `const` Keyword

1. **Constant Fields**
    - Fields declared with the `const` keyword are constant and their values cannot change. They are implicitly static because the value is the same across all instances.

```
public class MathConstants
{
    public const double Pi = 3.14159;
}
```

    - **Accessing Constant Fields**: Like static fields, constant fields are accessed using the class name.

```
double circleCircumference = 2 * MathConstants.Pi * radius;
```

- **Memory Efficiency**: Since `const` fields are the same for all instances, storing them in the class itself saves memory.

Best Practices

1. **Static Methods for Utility Functions**
   - Use static methods for utility functions that do not depend on instance data.
2. **Mark Private Methods Static if Possible**
   - If a private method does not use instance data, make it static. This indicates that the method does not modify the object's state and can slightly improve performance.
3. **Use `const` for Immutable Values**
   - Use the `const` keyword for values that do not change. This ensures they are shared across all instances, saving memory and improving performance.

Summary

- **Static Methods**: Belong to the class, not instances, and cannot access instance data.
- **Static Classes**: Cannot be instantiated and only contain static members.
- **`const` Fields**: Immutable and implicitly static, ensuring consistent values across all instances.
- **Best Practices**: Use static methods for stateless operations, mark non-instance-dependent private methods as static, and use `const` for immutable values.

By understanding and applying these concepts, you can write more efficient and readable code in C#.

Understanding Static Fields, Properties, and Constructors

In this lecture, we will learn about static fields, properties, and constructors. We'll discuss their purposes, how to use them, and the potential risks associated with them.

Static Fields and Properties

1. **Definition**
   - **Static Fields and Properties**: These belong to the class as a whole rather than any specific instance. They are shared across all instances of the class.

```
public class Rectangle
{
    public static int InstanceCount { get; private set; }

    public Rectangle()
    {
        InstanceCount++;
```

```
    }
}
```

2. **Use Case**

   - **Tracking Instance Count**: To keep track of the number of `Rectangle` instances created, use a static field or property. Increment this count in the constructor.

```
public class Rectangle
{
    public static int InstanceCount { get; private set; }

    public Rectangle()
    {
        InstanceCount++;
    }
}
```

   - **Accessing Static Properties**: Access static properties using the class name, not an instance.

```
Console.WriteLine(Rectangle.InstanceCount);
```

3. **Initialization**

   - **Default Initialization**: Static fields and properties are initialized to their default values if not explicitly initialized.

```
public static DateTime FirstUsage { get; private set; } = DateTime.Now;
```

Static Constructors

1. **Definition**

   - **Static Constructors**: These initialize static fields and properties. They do not have access modifiers and use the `static` keyword.

```
public class Rectangle
{
    public static DateTime FirstUsage { get; private set; }

    static Rectangle()
    {
        FirstUsage = DateTime.Now;
    }
}
```

2. **Execution**
    - **Timing**: The static constructor is called before the first instance of the class is created or any static members are accessed.

Risks and Best Practices

1. **Caution with Static Members**
    - **Shared State**: Static fields and properties can lead to unexpected behavior if not used carefully, as they are shared across all instances.
    - **Thread Safety**: If multiple threads access and modify static fields simultaneously, it can cause concurrency issues.
2. **Professional Experience**
    - **Avoidance in Production**: Many experienced developers avoid using static fields and properties in production code due to the risks associated with shared state and concurrency issues.
3. **Understanding Legacy Code**
    - **Reading Code**: Even if you avoid using static members, understanding them is crucial as you may encounter them in legacy code or other developers' code.

Summary

- **Static Fields and Properties**: Shared across all instances of a class, useful for tracking class-wide data.
- **Static Constructors**: Initialize static members, called before the first instance is created or any static members are accessed.
- **Risks**: Shared state can lead to concurrency issues; use static members cautiously.

By understanding and carefully applying these concepts, you can make informed decisions about when and how to use static fields, properties, and constructors in C#.