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.