Expressions and Control Flow in C#

Building Blocks for Effective Programming

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Overview of Expressions in Programming

Expressions are a fundamental concept in programming that allow developers to perform operations on data and variables. An expression is any valid unit of code that resolves to a value. It can consist of variables, operators, constants, and function calls. In C#, expressions are used in a variety of contexts, such as calculations, condition evaluations, and assignments. Understanding expressions is crucial because they form the basis of more complex programming constructs.

Evaluation of Expressions

Expressions are evaluated according to the rules of precedence and associativity, which determine the order in which operations are performed. This evaluation ultimately produces a result that can be used in further operations or decision-making processes within a program.

Importance and Use Cases of Different Types of Expressions

Expressions are essential for several reasons:

- Expressions enable arithmetic computations, which are fundamental to any application involving numerical data, such as financial software, games, and scientific simulations.
- Logical and relational expressions are used to make decisions within a program, such as determining the flow of control based on conditions.
- Expressions allow for the manipulation of data through operations like assignment, incrementing, and function calls.
- Well-structured expressions can make code more readable and easier to maintain. Clear expressions help in understanding the logic of the program and in debugging.

Categories of Expressions

Expressions in C# can be broadly categorized into three types: Arithmetic, Logical, and Relational. Each type serves different purposes and involves different operators and constructs.

Arithmetic Expressions

Arithmetic expressions perform mathematical operations on numeric data types. These include addition, subtraction, multiplication, division, and modulus operations. Arithmetic expressions are used in scenarios where numerical calculations are required.

Logical Expressions

Logical expressions evaluate to a boolean value (true or false) and are used to combine or negate boolean expressions. They play a crucial role in decision-making processes within a program. The primary logical operators in C# are AND (`&&`), OR (`||`), and NOT (`!`).

Relational Expressions

Relational expressions compare two values and return a boolean result based on the relationship between them. These are essential for making comparisons and decisions. Common relational operators include equal to (`==`), not equal to (`!=`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`).

Arithmetic Expressions

Arithmetic expressions in C# are expressions that perform mathematical operations on numeric values. These expressions involve the use of arithmetic operators to compute values based on operands (variables, constants, or both). The result of an arithmetic expression is always a numeric value.

Examples of Arithmetic Expressions

```
int a = 10;
int b = 5;
int sum = a + b; // sum is 15
int difference = a - b; // difference is 5
int product = a * b; // product is 50
int quotient = a / b; // quotient is 2
int remainder = a % b; // remainder is 0
```

In this example, `a` and `b` are operands, and the operators (`+`, `-`, `*`, `/`, `%`) perform the arithmetic operations.

Operators

C# provides several arithmetic operators to perform basic mathematical operations. Here are the primary arithmetic operators:

Addition (+)

Adds two operands.

```
int sum = a + b; // Adds a and b
```

Subtraction (-)

Subtracts the second operand from the first.

```
int difference = a - b; // Subtracts b from a
```

Multiplication (*)

Multiplies two operands.

```
int product = a * b; // Multiplies a and b
```

Division (/)

Divides the first operand by the second. Note that dividing by zero will cause a runtime error.

```
int quotient = a / b; // Divides a by b
```

Modulus (%)

Returns the remainder of the division of the first operand by the second.

```
int remainder = a % b; // Finds the remainder of a divided by b
```

Order of Operations (Precedence and Associativity)

When multiple arithmetic operations are present in a single expression, the order in which the operations are performed is determined by operator precedence and associativity rules.

Operator Precedence

Determines which operator is evaluated first. Operators with higher precedence are evaluated before operators with lower precedence.

- **High Precedence:** Multiplication (*), Division (/), Modulus (%)
- **Low Precedence:** Addition (+), Subtraction (-)

Associativity

Determines the order in which operators of the same precedence are evaluated. Most arithmetic operators have left-to-right associativity, meaning they are evaluated from left to right.

```
int result = 10 + 5 * 2 - 3;
// The multiplication (*) is performed first: 5 * 2 = 10
// Then addition (+) and subtraction (-) are performed left to right:
   10 + 10 - 3 = 17
```

To explicitly define the order of operations, parentheses can be used. Operations within parentheses are evaluated first, regardless of operator precedence.

```
int result = (10 + 5) * 2 - 3;
// The expression inside parentheses is evaluated first: (10 + 5) = 15
// Then multiplication (*): 15 * 2 = 30
// Finally, subtraction (-): 30 - 3 = 27
```

Examples and Practice Problems

Let's work through some examples and practice problems to solidify our understanding of arithmetic expressions and their order of operations.

Example 1

```
int a = 8;
int b = 3;
int result = a * b + (a - b);
// Step-by-step evaluation:
// 1. Parentheses: (a - b) = (8 - 3) = 5
// 2. Multiplication: a * b = 8 * 3 = 24
// 3. Addition: 24 + 5 = 29
```

Example 2

```
int x = 4;
int y = 2;
int z = 1;
int result = x + y * z - x / y;
// Step-by-step evaluation:
// 1. Multiplication: y * z = 2 * 1 = 2
// 2. Division: x / y = 4 / 2 = 2
// 3. Addition and Subtraction from left to right: 4 + 2 - 2 = 4
```

Practice Problems

Calculate the following expression:

```
int result = 12 / 4 + 5 * 3 - 2;
// Expected result: 17
```

Evaluate the expression with parentheses:

```
int result = (8 + 2) * (5 - 3) + 6 / 2;
// Expected result: 22
```

Determine the result of the following expression:

```
int a = 7;
int b = 3;
int c = 5;
int result = a * b % c + b / c;
// Expected result: 1
```

By understanding the definition, operators, order of operations, and through practicing examples, you will gain a solid foundation in working with arithmetic expressions in C#.

Logical Expressions

Logical expressions are expressions that evaluate to a boolean value: true or false. These expressions are primarily used in decision-making processes within a program, allowing the program to execute different code paths based on certain conditions. Logical expressions often involve comparisons and logical operators to combine or negate boolean values.

Definition and Examples of Logical Expressions

```
bool isAdult = age >= 18; // Evaluates to true if age is 18 or older
bool isSenior = age >= 65; // Evaluates to true if age is 65 or older
bool canVote = isAdult && !isSenior; // Evaluates to true if the person
    is an adult but not a senior
```

In this example, `isAdult` and `isSenior` are logical expressions that use relational operators, and `canVote` combines these expressions using logical operators.

Operators: AND (`&&`), OR (`||`), NOT (`!`)

C# provides three primary logical operators to work with boolean values: AND, OR, and NOT.

AND (`&&`)

The result is true only if both operands are true.

```
bool result = true && false; // result is false
```

OR (`||`)

The result is true if at least one of the operands is true.

```
bool result = true || false; // result is true
```

NOT (`!`)

The result is the negation of the operand.

```
bool result = !true; // result is false
```

Truth Tables and Logical Equivalences

A truth table is a mathematical table used to determine the result of a logical expression based on all possible combinations of its inputs. Here are the truth tables for the AND, OR, and NOT operators:

AND (`&&`) Truth Table

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

OR (||) Truth Table

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

NOT (!) Truth Table

A	!A
true	false
false	true

Combining Logical Expressions with Examples

Logical expressions can be combined to form more complex conditions using the AND, OR, and NOT operators. These combinations allow for sophisticated decision-making in programs.

Example 1

```
int age = 25;
bool hasLicense = true;
bool canDrive = (age >= 18) && hasLicense; // Evaluates to true if age
   is 18 or older and has a license
```

Example 2

```
bool isWeekend = (day == "Saturday") || (day == "Sunday"); // Evaluates
    to true if the day is either Saturday or Sunday
```

Example 3

```
bool isRaining = false;
bool hasUmbrella = true;
bool shouldGoOutside = !isRaining || hasUmbrella; // Evaluates to true
   if it is not raining or has an umbrella
```

Examples and Practice Problems

Let's look at some more examples and practice problems to solidify our understanding of logical expressions.

Example 1

```
int temperature = 30;
bool isHot = temperature > 25;
bool isHumid = humidity > 70;
bool uncomfortableWeather = isHot && isHumid; // Evaluates to true if
    both isHot and isHumid are true
```

Example 2

```
bool isStudent = true;
bool hasID = true;
bool canEnterLibrary = isStudent || hasID; // Evaluates to true if
    either isStudent or hasID is true
```

Practice Problems

Determine if a person is eligible to vote. A person is eligible if they are at least 18 years old and are a citizen.

```
int age = 20;
bool isCitizen = true;
bool canVote = (age >= 18) && isCitizen;
// Expected result: canVote is true
```

Evaluate if a store is open. The store is open if it is not a holiday and the time is between 9 AM and 9 PM.

```
bool isHoliday = false;
int hour = 10;
bool isOpen = !isHoliday && (hour >= 9 && hour <= 21);
// Expected result: isOpen is true
```

Check if a person qualifies for a senior discount. A person qualifies if they are 65 years old or older, or if they are a member of the senior club.

```
int age = 60;
bool isSeniorClubMember = true;
bool qualifiesForDiscount = (age >= 65) || isSeniorClubMember;
// Expected result: qualifiesForDiscount is true
```

Relational Expressions

Relational expressions are used to compare two values and determine the relationship between them. The result of a relational expression is a boolean value: true if the relationship holds, and false otherwise. These expressions are used in decision-making processes within a program, allowing the program to execute different code paths based on the comparison results.

Examples of Relational Expressions

```
int a = 10;
int b = 5;
bool isEqual = (a == b); // isEqual is false because 10 is not equal to
    5
bool isNotEqual = (a != b); // isNotEqual is true because 10 is not
   equal to 5
bool isGreater = (a > b); // isGreater is true because 10 is greater
   than 5
bool isLess = (a < b); // isLess is false because 10 is not less than 5
bool isGreaterOrEqual = (a >= b); // isGreaterOrEqual is true because
   10 is greater than or equal to 5
bool isLessOrEqual = (a <= b); // isLessOrEqual is false because 10 is
   not less than or equal to 5
```

In this example, `a` and `b` are operands, and the relational operators compare these values to produce boolean results.

Operators

C# provides several relational operators to compare values. Here are the primary relational operators:

Equal to (==)

Checks if two values are equal.

```
bool isEqual = (a == b); // true if a is equal to b
```

Not equal to (!=)

Checks if two values are not equal.

```
bool isNotEqual = (a != b); // true if a is not equal to b
```

Greater than (>)

Checks if the first value is greater than the second value.

```
bool isGreater = (a > b); // true if a is greater than b
```

Less than (<)

Checks if the first value is less than the second value.

```
bool isLess = (a < b); // true if a is less than b
```

Greater than or equal to (>=)

Checks if the first value is greater than or equal to the second value.

```
bool isGreaterOrEqual = (a >= b); // true if a is greater than or equal
    to b
```

Less than or equal to (<=)

Checks if the first value is less than or equal to the second value.

```
bool isLessOrEqual = (a <= b); // true if a is less than or equal to b
```

Using Relational Expressions in Conditional Statements

Relational expressions are often used in conditional statements to control the flow of a program. These expressions determine whether a block of code should be executed based on the comparison of values.

Example: Using Relational Expressions in an if Statement

```
int score = 85;

if (score >= 90) {
    Console.WriteLine("Grade: A");
} else if (score >= 80) {
    Console.WriteLine("Grade: B");
} else if (score >= 70) {
    Console.WriteLine("Grade: C");
} else if (score >= 60) {
    Console.WriteLine("Grade: D");
} else {
    Console.WriteLine("Grade: F");
}
```

In this example, relational expressions like `score >= 90` and `score >= 80` determine which block of code is executed based on the value of `score`.

Examples and Practice Problems

Let's look at some more examples and practice problems to solidify our understanding of relational expressions.

Example 1: Checking Age for Voting Eligibility

```
int age = 20;
bool canVote = (age >= 18);

if (canVote) {
    Console.WriteLine("You are eligible to vote.");
} else {
    Console.WriteLine("You are not eligible to vote.");
}
```

Example 2: Comparing Two Numbers

```
int num1 = 15;
int num2 = 20;
```

```
if (num1 > num2) {
    Console.WriteLine("num1 is greater than num2");
} else if (num1 < num2) {
    Console.WriteLine("num1 is less than num2");
} else {
    Console.WriteLine("num1 is equal to num2");
}
```

Practice Problems

Write a program to check if a number is positive, negative, or zero.

```
int number = -5;

if (number > 0) {
    Console.WriteLine("The number is positive.");
} else if (number < 0) {
    Console.WriteLine("The number is negative.");
} else {
    Console.WriteLine("The number is zero.");
}
// Expected output: "The number is negative."
```

Determine if a person is eligible for a senior discount. A person is eligible if they are 65 years old or older.

```
int age = 70;
bool eligibleForDiscount = (age >= 65);

if (eligibleForDiscount) {
    Console.WriteLine("You are eligible for a senior discount.");
} else {
    Console.WriteLine("You are not eligible for a senior discount.");
}
// Expected output: "You are eligible for a senior discount."
```

Write a program to find the largest of three numbers.

```
int num1 = 12;
int num2 = 25;
int num3 = 7;

if (num1 >= num2 && num1 >= num3) {
    Console.WriteLine("num1 is the largest.");
} else if (num2 >= num1 && num2 >= num3) {
    Console.WriteLine("num2 is the largest.");
} else {
    Console.WriteLine("num3 is the largest.");
}
// Expected output: "num2 is the largest."
```

Overview of Control Flow in Programming

Control flow refers to the order in which individual statements, instructions, or function calls are executed or evaluated within a program. In programming, control flow is the

mechanism that dictates the direction the program takes based on certain conditions or the values of variables. Proper control flow is crucial for implementing complex logic and ensuring that programs behave as intended.

Importance of Conditional Statements in Decision Making

Conditional statements are fundamental in programming because they enable decision-making. They allow programs to execute different blocks of code based on specific conditions, making the software dynamic and responsive to different inputs and scenarios. Without conditional statements, programs would follow a linear and unvarying path, which would significantly limit their functionality.

Different Types of Conditional Statements: `if`, `switch`, `ternary operator`

If Statements

Syntax and Structure: The `if` statement evaluates a boolean expression and executes a block of code if the expression is true.

Use Cases: Simple decision-making scenarios where there is one condition to check.

Switch Statements

Syntax and Structure: The `switch` statement evaluates a variable against a list of values (cases) and executes the corresponding block of code.

Use Cases: Scenarios where there are multiple possible values for a variable, and different actions are required for each value.

Ternary Operator

Syntax and Structure: The ternary operator (`?:`) is a shorthand for `if-else` statements that assigns a value based on a boolean expression.

Use Cases: Simple conditional assignments that can be expressed in a single line of code.

When and Why to Use Each Type

If Statements

Use `if` statements for straightforward condition checks and when there are multiple conditions that may require nested or sequential evaluations. They are versatile and suitable for most decision-making scenarios.

Switch Statements

Use `switch` statements when you need to compare a single variable against a set of discrete values. They are more readable and efficient than multiple `if-else-if` statements for these scenarios.

Ternary Operator

Use the ternary operator for concise, simple conditional assignments. It is ideal for situations where you want to quickly assign a value based on a condition without the need for multiple lines of code.

Syntax and Structure of If Statements

If Statement

The if statement is used to execute a block of code only if a specified condition is true. The basic syntax is:

```
if (condition) {
    // code to execute if condition is true
}
```

If-Else Statement

The if-else statement provides an alternative block of code that executes if the condition is false. The syntax is:

```
if (condition) {
    // code to execute if condition is true
} else {
    // code to execute if condition is false
}
```

If-Else-If Ladder

The if-else-if ladder allows for multiple conditions to be checked in sequence. Each condition is evaluated in order, and the corresponding block of code executes when a true condition is found. If none of the conditions are true, the else block executes. The syntax is:

```
if (condition1) {
    // code to execute if condition1 is true
} else if (condition2) {
    // code to execute if condition2 is true
} else if (condition3) {
    // code to execute if condition3 is true
} else {
    // code to execute if none of the above conditions are true
}
```

Explanation of If-Else and If-Else-If Ladder

The if-else structure is used for binary decisions where there are two possible outcomes. If the condition is true, the first block of code runs; otherwise, the second block runs.

The if-else-if ladder extends this logic to multiple conditions. It checks each condition in order, executing the corresponding block of code for the first true condition it finds. If no conditions are true, the else block executes as a fallback.

Use Cases and Best Practices

Use Cases

- Use if statements to handle simple, single-condition scenarios.
- Use if-else statements when there are two possible outcomes, such as validating user input and providing feedback.
- Use if-else-if ladders for more complex decision-making processes that involve multiple conditions, such as determining grades based on score ranges.

Best Practices

- Ensure that each condition is simple and easy to understand. Complex conditions can be broken down into multiple if statements.
- Deeply nested if-else statements can be hard to read and maintain. Consider using logical operators or refactoring the code to simplify it.
- Conditions should be meaningful and clearly reflect the decision being made. Use descriptive variable names to enhance readability.
- Always provide a default action (using else) to handle unexpected cases and improve code robustness.

Practical Applications

Example 1: Checking for Voting Eligibility

```
int age = 20;
if (age >= 18) {
    Console.WriteLine("You are eligible to vote.");
} else {
    Console.WriteLine("You are not eligible to vote.");
}
```

Example 2: Grading System

```
int score = 85;
if (score >= 90) {
    Console.WriteLine("Grade: A");
} else if (score >= 80) {
```

```
    Console.WriteLine("Grade: B");
} else if (score >= 70) {
    Console.WriteLine("Grade: C");
} else if (score >= 60) {
    Console.WriteLine("Grade: D");
} else {
    Console.WriteLine("Grade: F");
}
```

Example 3: Checking Temperature Range

```
int temperature = 60;
if (temperature > 80) {
    Console.WriteLine("It's very hot outside.");
} else if (temperature > 70) {
    Console.WriteLine("It's hot outside.");
} else if (temperature > 60) {
    Console.WriteLine("It's warm outside.");
} else if (temperature > 50) {
    Console.WriteLine("It's cool outside.");
} else {
    Console.WriteLine("It's cold outside.");
}
```

Exercise 1: Determine if a Number is Positive, Negative, or Zero

Write a program to check if a number is positive, negative, or zero.

```
int number = -5;

if (number > 0) {
    Console.WriteLine("The number is positive.");
} else if (number < 0) {
    Console.WriteLine("The number is negative.");
} else {
    Console.WriteLine("The number is zero.");
}
```

Exercise 2: Check if a Person is Eligible for a Senior Discount

A person is eligible for a senior discount if they are 65 years old or older.

```
int age = 70;
bool eligibleForDiscount = (age >= 65);

if (eligibleForDiscount) {
    Console.WriteLine("You are eligible for a senior discount.");
} else {
    Console.WriteLine("You are not eligible for a senior discount.");
}
```

Exercise 3: Find the Largest of Three Numbers

Write a program to find the largest of three numbers.

```
int num1 = 12;
int num2 = 25;
int num3 = 7;

if (num1 >= num2 && num1 >= num3) {
    Console.WriteLine("num1 is the largest.");
} else if (num2 >= num1 && num2 >= num3) {
    Console.WriteLine("num2 is the largest.");
} else {
    Console.WriteLine("num3 is the largest.");
}
```

These examples and exercises provide practical applications of if, if-else, and if-else-if statements, illustrating how they are used to control program flow based on different conditions.

Syntax and Structure of Switch Statements

A switch statement in C# allows you to select one of many code blocks to execute. It is particularly useful when you need to compare the value of a variable against a set of predefined constants. The structure of a switch statement is as follows:

```
switch (variable)
{
    case value1:
        // Code to execute if variable equals value1
        break;
    case value2:
        // Code to execute if variable equals value2
        break;
    // Additional cases as needed
    default:
        // Code to execute if variable does not match any case
        break;
}
```

Explanation

- `switch (variable)`: The variable is evaluated once, and its value is compared with each case.
- `case value1`: If the variable equals `value1`, the corresponding block of code executes.
- `break`: The break statement terminates the switch statement, preventing fall-through to subsequent cases.
- `default`: The default block executes if the variable does not match any of the case values.

Comparison Between Switch and If-Else-If

Switch Statement

- Easier to read when dealing with multiple discrete values.
- Generally more efficient than multiple `if-else-if` statements, especially when the number of cases is large.
- Can only compare the variable against constants or expressions that resolve to constants.

If-Else-If Statement

- Can handle a wider range of conditions, including complex logical expressions and ranges.
- Can become cumbersome and difficult to read with many conditions.
- May be less efficient compared to a switch statement when evaluating many discrete values.

Use Cases and Best Practices

Use Cases

- When a variable can take one of several predefined constant values.
- Handling user input that selects from a set of options.
- Implementing state transitions based on the current state value.

Best Practices

- Use switch statements for better readability when comparing a variable against multiple discrete values.
- Always use break statements to prevent fall-through, unless intentional.
- Always include a default case to handle unexpected values and improve robustness.

Handling Multiple Cases and Default Case

Multiple cases can execute the same block of code by listing them together without a break statement between them. The default case acts as a catch-all for any values that do not match the specified cases.

```
char grade = 'B';

switch (grade)
{
    case 'A':
    case 'B':
    case 'C':
        Console.WriteLine("Passing grade.");
```

```
        break;
    case 'D':
    case 'F':
        Console.WriteLine("Failing grade.");
        break;
    default:
        Console.WriteLine("Invalid grade.");
        break;
}
```

In this example, grades 'A', 'B', and 'C' are handled by the same block of code, indicating a passing grade.

Examples and Practical Applications

Example 1: Menu Selection

```
int option = 2;

switch (option)
{
    case 1:
        Console.WriteLine("Option 1 selected.");
        break;
    case 2:
        Console.WriteLine("Option 2 selected.");
        break;
    case 3:
        Console.WriteLine("Option 3 selected.");
        break;
    default:
        Console.WriteLine("Invalid option.");
        break;
}
```

Example 2: Day of the Week

```
int day = 3;

switch (day)
{
    case 1:
        Console.WriteLine("Monday");
        break;
    case 2:
        Console.WriteLine("Tuesday");
        break;
    case 3:
        Console.WriteLine("Wednesday");
        break;
    case 4:
        Console.WriteLine("Thursday");
        break;
    case 5:
        Console.WriteLine("Friday");
```

```
        break;
    case 6:
        Console.WriteLine("Saturday");
        break;
    case 7:
        Console.WriteLine("Sunday");
        break;
    default:
        Console.WriteLine("Invalid day.");
        break;
}
```

Exercises with Solutions

Exercise 1: Determine the Season

Write a program that takes a month (1-12) and prints the corresponding season.

```
int month = 4;
string season;

switch (month)
{
    case 12:
    case 1:
    case 2:
        season = "Winter";
        break;
    case 3:
    case 4:
    case 5:
        season = "Spring";
        break;
    case 6:
    case 7:
    case 8:
        season = "Summer";
        break;
    case 9:
    case 10:
    case 11:
        season = "Autumn";
        break;
    default:
        season = "Invalid month";
        break;
}

Console.WriteLine("The season is: " + season);
// Expected output: "The season is: Spring"
```

Exercise 2: Simple Calculator

Write a program that takes two numbers and an operator (`+`, `-`, `*`, `/`) and performs the corresponding operation.

```
double num1 = 10;
double num2 = 5;
char operation = '+';
double result;

switch (operation)
{
    case '+':
        result = num1 + num2;
        break;
    case '-':
        result = num1 - num2;
        break;
    case '*':
        result = num1 * num2;
        break;
    case '/':
        if (num2 != 0)
        {
            result = num1 / num2;
        }
        else
        {
            Console.WriteLine("Cannot divide by zero.");
            return;
        }
        break;
    default:
        Console.WriteLine("Invalid operator.");
        return;
}

Console.WriteLine("The result is: " + result);
// Expected output: "The result is: 15"
```

By understanding the syntax and structure of switch statements, comparing them with if-else-if statements, following best practices, and handling multiple cases and default cases, you can effectively use switch statements in your C# programs.

Syntax and Structure of the Ternary Operator (?:)

The ternary operator, also known as the conditional operator, is a shorthand way of writing an if-else statement that assigns a value based on a boolean condition. The ternary operator in C# uses the ? and : symbols and has the following syntax:

```
condition ? value_if_true : value_if_false;
```

Explanation

- `condition`: A boolean expression that evaluates to true or false.
- `value_if_true`: The value assigned if the condition is true.
- `value_if_false`: The value assigned if the condition is false.

This operator allows you to write concise conditional assignments in a single line of code.

Comparison with if-else Statements

Ternary Operator

- Allows for more concise code, making it easier to read and write for simple conditional assignments.
- All logic is contained within a single line.
- Ideal for straightforward conditions where both possible values are immediately clear.

If-Else Statement

- Can handle more complex logic and multiple statements within each condition block.
- Requires multiple lines of code.
- Better suited for more complex conditions where the logic needs to be clearly separated.

Example

```
// Using if-else statement
int age = 18;
string eligibility;
if (age >= 18)
{
    eligibility = "You are eligible to vote.";
}
else
{
    eligibility = "You are not eligible to vote.";
}

// Using ternary operator
eligibility = (age >= 18) ? "You are eligible to vote." : "You are not
   eligible to vote.";
```

Use Cases and Best Practices

Use Cases

- When you need to assign a value based on a single condition.
- Situations where using multiple lines for an if-else statement would be overkill.

Best Practices

- Ensure that the ternary operator is used for simple conditions to maintain code readability. Avoid using it for complex conditions that would be clearer with an if-else statement.
- Avoid nesting ternary operators, as this can make the code difficult to read and understand.
- Use parentheses to clearly delineate the condition and the true/false values, especially when using the ternary operator within larger expressions.

Examples and Practical Applications

Example 1: Determining Pass or Fail Status

```
int score = 75;
string result = (score >= 60) ? "Pass" : "Fail";
Console.WriteLine(result); // Output: "Pass"
```

Example 2: Finding the Larger of Two Numbers

```
int num1 = 10;
int num2 = 20;
int max = (num1 > num2) ? num1 : num2;
Console.WriteLine(max); // Output: 20
```

Example 3: Assigning a Discount Based on Membership Status

```
bool isMember = true;
double discount = isMember ? 0.1 : 0.05; // 10% discount for members,
    5% for non-members
Console.WriteLine("Discount: " + (discount * 100) + "%"); // Output: "
    Discount: 10%"
```

Exercises with Solutions

Exercise 1: Determine if a Number is Even or Odd

Write a program that takes an integer and uses the ternary operator to determine if the number is even or odd.

```
int number = 4;
string parity = (number % 2 == 0) ? "Even" : "Odd";
Console.WriteLine(parity); // Expected output: "Even"
```

Exercise 2: Determine Voting Eligibility

Write a program that takes an age and uses the ternary operator to determine if the person is eligible to vote (18 years or older).

```
int age = 17;
string canVote = (age >= 18) ? "Eligible to vote" : "Not eligible to
    vote";
Console.WriteLine(canVote); // Expected output: "Not eligible to vote"
```

Exercise 3: Assigning Grades Based on Score

Write a program that takes a score and uses the ternary operator to assign a grade. Scores 90 and above get 'A', below 90 but above or equal to 80 get 'B', and below 80 get 'C'.

```
int score = 85;
string grade = (score >= 90) ? "A" : (score >= 80) ? "B" : "C";
Console.WriteLine("Grade: " + grade); // Expected output: "Grade: B"
```

Concept of Nesting Conditional Statements

Nesting conditional statements involves placing one or more conditional statements inside another. This allows for more complex decision-making processes where multiple conditions need to be checked sequentially. Nesting can be done with both if statements and switch statements, and it can also involve combining different types of conditional statements.

Nesting if Statements

Nesting if statements involves placing an if or else if statement inside another if or else if block. This structure allows for checking multiple related conditions in a hierarchical manner.

Syntax and Structure

```
if (condition1)
{
    if (condition2)
    {
        // Code to execute if both condition1 and condition2 are true
    }
    else
    {
        // Code to execute if condition1 is true and condition2 is
   false
    }
}
else
{
    // Code to execute if condition1 is false
}
```

Nesting switch Statements

Nesting switch statements involves placing a switch statement inside one of the case blocks of another switch statement. This can be useful when the decision-making process depends on multiple variables or multiple levels of conditions.

Syntax and Structure

```
switch (variable1)
{
    case value1:
        switch (variable2)
        {
            case valueA:
                // Code to execute if variable1 equals value1 and
   variable2 equals valueA
                break;
            case valueB:
                // Code to execute if variable1 equals value1 and
   variable2 equals valueB
                break;
            // Additional cases for variable2
        }
        break;
    case value2:
        // Code to execute if variable1 equals value2
        break;
    // Additional cases for variable1
}
```

Combining Different Conditional Statements

Combining different conditional statements means using a mix of if, else if, else, and switch statements together to achieve the desired control flow. This approach allows for flexibility in handling complex decision-making scenarios.

Example

```
if (condition1)
{
    if (condition2)
    {
        // Code to execute if both condition1 and condition2 are true
    }
    else
    {
        switch (variable)
        {
            case value1:
                // Code to execute if condition1 is true, condition2 is
    false, and variable equals value1
                break;
            case value2:
```

```
                // Code to execute if condition1 is true, condition2 is
    false, and variable equals value2
                break;
            // Additional cases for variable
        }
    }
}
else
{
    // Code to execute if condition1 is false
}
```

Examples and Practical Applications

Example 1: Nested if Statements

```
int score = 85;
bool extraCredit = true;

if (score >= 90)
{
    Console.WriteLine("Grade: A");
}
else if (score >= 80)
{
    if (extraCredit)
    {
        Console.WriteLine("Grade: A (with extra credit)");
    }
    else
    {
        Console.WriteLine("Grade: B");
    }
}
else if (score >= 70)
{
    Console.WriteLine("Grade: C");
}
else
{
    Console.WriteLine("Grade: F");
}
```

Example 2: Nested switch Statements

```
int day = 5;
int month = 12;

switch (month)
{
    case 12:
        switch (day)
        {
            case 25:
                Console.WriteLine("Merry Christmas!");
```

```
                break;
            case 31:
                Console.WriteLine("Happy New Year's Eve!");
                break;
            default:
                Console.WriteLine("It's December.");
                break;
        }
        break;
    case 7:
        switch (day)
        {
            case 4:
                Console.WriteLine("Happy Independence Day!");
                break;
            default:
                Console.WriteLine("It's July.");
                break;
        }
        break;
    default:
        Console.WriteLine("It's a regular day.");
        break;
}
```

Exercises with Solutions

Exercise 1: Determine the Category of a Person Based on Age and Employment Status

Write a program that categorizes a person as a "Student", "Employed", "Unemployed", or "Retired" based on their age and employment status.

```
int age = 25;
bool isEmployed = true;

if (age < 18)
{
    Console.WriteLine("Student");
}
else
{
    if (isEmployed)
    {
        if (age < 65)
        {
            Console.WriteLine("Employed");
        }
        else
        {
            Console.WriteLine("Retired");
        }
    }
    else
    {
        if (age < 65)
        {
```

```
            Console.WriteLine("Unemployed");
        }
        else
        {
            Console.WriteLine("Retired");
        }
    }
}
// Expected output: "Employed"
```

Exercise 2: Determine the Day Type Based on Day of the Week and Time

Write a program that determines if it is a "Weekday Morning", "Weekday Evening", "Weekend Morning", or "Weekend Evening" based on the day of the week and the time of the day.

```
string dayOfWeek = "Saturday";
int hour = 10;

if (dayOfWeek == "Saturday" || dayOfWeek == "Sunday")
{
    if (hour < 12)
    {
        Console.WriteLine("Weekend Morning");
    }
    else
    {
        Console.WriteLine("Weekend Evening");
    }
}
else
{
    if (hour < 12)
    {
        Console.WriteLine("Weekday Morning");
    }
    else
    {
        Console.WriteLine("Weekday Evening");
    }
}
// Expected output: "Weekend Morning"
```