Chapter 1: Introduction to C#.NET 1.1 Introduce C# its features and applications.

Provide an overview of C# programming language.

Explain the features and advantages of using C#.

Discuss the applications and industries where C# is commonly used.

1.2 Introduce the structure of C#.

Describe the basic structure of a C# program.

Explain the different parts of a C# program, such as namespaces, classes, methods, and statements.

1.3 Analyze the variables of C#.

Define variables and their role in programming.

Discuss the different data types available in C# for declaring variables.

Explain variable naming conventions and rules.

1.4 Describe the Identifiers of C#.

Define identifiers and their importance in C#.

Discuss the rules and conventions for naming identifiers in C#.

Provide examples of valid and invalid identifiers.

1.5 Introduce the keywords of C#.

Explain keywords and their significance in C#.

List and describe the commonly used keywords in C#.

Discuss the restrictions and limitations on using keywords as identifiers.

1.6 Explain data types in C#.

Discuss the different data types available in C#.

Explain value types, reference types, and built-in data types.

Provide examples of using different data types in C#.

1.7 State the C# type conversion.

Explain type conversion and its role in C# programming.

Discuss implicit and explicit type conversion.

Provide examples of type conversion in C#.

1.8 Introduce to C# operators.

Explain operators and their usage in C#.

Discuss arithmetic, relational, logical, assignment, and other operators in C#.

Provide examples of using operators in C#.

Chapter 2: Control Statements 2.1 Introduce the control statements.

Explain control statements and their purpose in programming.

Discuss the different types of control statements in C#.

2.2 Demonstrate if, if else, and if else ladder and compute it.

Explain the if statement and its usage for conditional execution.

Demonstrate the if-else statement and its application for branching logic.

Illustrate the use of an if-else ladder for multiple conditions.

2.3 Demonstrate the switch statement and its functions.

Explain the switch statement and its purpose.

Demonstrate the syntax and usage of the switch statement.

Discuss the scenarios where the switch statement is useful.

2.4 Illustrate the for loop and deduce its usage.

Explain the for loop and its role in repetitive execution.

Demonstrate the syntax and usage of the for loop.

Provide examples of using the for loop in different scenarios.

2.5 Illustrate the do-while loop and deduce its usage.

Explain the do-while loop and its purpose.

Demonstrate the syntax and usage of the do-while loop.

Discuss situations where the do-while loop is applicable.

2.6 Illustrate the while loop and deduce its usage.

Explain the while loop and its significance.

Demonstrate the syntax and usage of the while loop.

Provide examples of using the while loop for iterative tasks.

2.7 Classify loop control statements and compare their features.

Discuss loop control statements such as break, continue, and goto.

Explain the purpose and usage of each loop control statement.

Compare the features and differences between these statements.

Chapter 3: Arrays 3.1 Introduce the arrays and their usage.

Explain arrays and their importance in C#.

Discuss the benefits and applications of using arrays.

Describe the concept of indexing in arrays.

3.2 Demonstrate the declaration and initialization of an array.

Explain how to declare and initialize arrays in C#.

Discuss single-dimensional and multi-dimensional arrays.

Provide examples of declaring and initializing arrays.

3.3 Illustrate data access from an array.

Explain how to access elements in an array using indexes.

Discuss the concept of bounds checking and handling array out-of-bounds errors.

Provide examples of accessing and manipulating array elements.

3.4 Introduce multidimensional arrays.

Explain multidimensional arrays and their usage.

Discuss rectangular and jagged arrays.

Provide examples of working with multidimensional arrays.

3.5 Compare and deduce the applications of jagged arrays, param arrays, and the array class.

Discuss the concept of jagged arrays and their applications.

Explain the usage of param arrays for variable-length parameter lists.

Introduce the array class and its functionalities.

Chapter 4: Strings 4.1 Introduce strings, their usages, and functions.

Explain strings and their significance in C# programming.

Discuss the applications and importance of strings.

Introduce various string operations and functions.

4.2 Demonstrate the creation of a string object.

Explain how to create string objects in C#.

Discuss string literals, string concatenation, and string interpolation.

Provide examples of creating and manipulating strings.

4.3 Demonstrate the methods of the string class and deduce their usages.

Explain the methods available in the string class.

Discuss commonly used string methods for manipulation and transformation.

Provide examples of using string class methods.

4.4 Introduce string functions and examine their usage.

Discuss built-in string functions and their purpose.

Explain the usage of string functions for various operations.

Provide examples of using string functions in C#.

Chapter 5: Structures 5.1 Introduction to structures, their features, and necessities.

Explain structures and their role in C# programming.

Discuss the features and advantages of using structures.

Explain when to use structures instead of classes.

5.2 Demonstration of defining structures and their usage.

Explain how to define structures in C#.

Discuss the syntax and rules for creating structures.

Provide examples of using structures in C# programs.

5.3 Compare and evaluate class vs. structure and demonstrate it.

Discuss the differences between classes and structures.

Compare their features, behavior, and memory allocation.

Provide examples to showcase the use cases of classes and structures.

Chapter 6: Pointers 6.1 Introduce pointers, their features, and applications.

Explain pointers and their significance in C# programming.

Discuss the role of pointers in memory management and low-level operations.

Provide examples of scenarios where pointers are useful.

6.2 Differentiate between the advantages and disadvantages of pointers.

Discuss the advantages and disadvantages of using pointers.

Explain the potential risks and challenges associated with pointers.

Compare the benefits and drawbacks of using pointers in C#.

6.3 Demonstrate the access of data value using a pointer.

Explain how to declare and use pointers in C#.

Demonstrate how to access data values using pointers.

Discuss memory addresses and pointer arithmetic.

6.4 Illustrate the passing of pointers as parameters to methods.

Explain how to pass pointers as parameters to methods.

Discuss the advantages and use cases of passing pointers.

Provide examples of using pointers as method parameters.

6.5 Demonstrate the access of array elements using a pointer.

Explain how to access array elements using pointers.

Discuss the benefits and challenges of using pointers with arrays.

Provide examples of using pointers to manipulate array elements.

Chapter 7: Working with Databases 7.1 Introduce databases, their features, and necessity in a programming environment.

Explain the concept of databases and their importance.

Discuss the role of databases in storing and managing data.

Highlight the necessity of database integration in programming.

7.2 Demonstrate the database environment setup and configure the requirements.

Explain how to set up a database environment for C# programming.

Discuss the required software, configurations, and connections.

Provide step-by-step instructions for configuring the database environment.

7.3 Illustrate the connection of a C# program with a database.

Explain how to establish a connection between a C# program and a database.

Discuss different database connection methods and libraries in C#.

Provide examples of connecting a C# program to a database.

7.4 Demonstrate the read and write operations from the database.

Explain how to perform read and write operations on a database using C#.

Discuss querying data, inserting, updating, and deleting records.

Provide examples of performing CRUD (Create, Read, Update, Delete) operations with a database using C#.

Chapter 1: Introduction to C#.NET

1.1 Introduce C# its features and applications. C# (pronounced "C sharp") is a modern, object-oriented programming language developed by Microsoft. It is part of the .NET framework and is widely used for developing a variety of applications, including desktop, web, and mobile applications. C# combines the power of C++ with the simplicity of Visual Basic, making it a popular choice among developers. Some of its key features include type safety, garbage collection, scalability, and interoperability. C# is used in industries such as finance, healthcare, gaming, and enterprise software development.

1.2 Introduce the structure of C#. A C# program has a hierarchical structure consisting of namespaces, classes, methods, and statements. Namespaces are used to organize code into logical units. Classes are the fundamental building blocks of C# programs and contain data members and methods. Methods are blocks of code that perform specific tasks. Statements are individual instructions within methods that control the flow of program execution.

1.3 Analyze the variables of C#. In C#, variables are used to store and manipulate data. Variables have a data type that determines the type of data they can hold, such as integers, floating-point numbers, characters, or boolean values. Variables in C# can be declared using the syntax: data\_type variable\_name; and can be assigned values using the assignment operator (=).

1.4 Describe the Identifiers of C#. Identifiers are names given to various elements in a C# program, such as variables, classes, methods, and namespaces. Identifiers must follow certain rules and conventions. They should start with a letter or an underscore, can contain letters, digits, and underscores, and cannot be a reserved keyword. Good naming conventions help make code more readable and maintainable.

1.5 Introduce the keywords of C#. Keywords are reserved words in C# that have predefined meanings and cannot be used as identifiers. They are used to define the syntax and structure of the language. Examples of C# keywords include if, else, for, while, class, void, and return. Understanding and correctly using keywords is essential for writing valid and meaningful C# code.

1.6 Explain data types in C#. Data types define the kind of data that can be stored in a variable or used in a program. C# supports a wide range of data types, including integers, floating-point numbers, characters, strings, booleans, and more. Data types determine the size and memory representation of the data, as well as the operations that can be performed on it. Choosing the appropriate data type is important for efficient memory usage and accurate data manipulation.

1.7 State the C# type conversion. Type conversion, also known as type casting, is the process of converting a value from one data type to another. C# supports two types of type conversions: implicit and explicit. Implicit type conversion is performed automatically by the compiler when there is no risk of data loss. Explicit type conversion requires the programmer to explicitly specify the conversion using casting operators. Understanding type conversion is crucial when working with different data types and performing operations on them.

1.8 Introduce to C# operators. Operators in C# are symbols that perform operations on operands (variables, literals, or expressions). C# supports various types of operators, such as arithmetic operators (+, -, \*, /), relational operators (>, <, ==, !=), logical operators (&&, ||, !), and assignment operators (=, +=, -=, \*=). Operators allow for mathematical calculations, comparison of values, logical operations, and more. Understanding the different operators and their precedence is essential for writing effective C# code.

Chapter 2: Control Statements

2.1 Introduce the control statements. Control statements in C# are used to control the flow of program execution based on specific conditions. They enable the program to make decisions and perform different actions based on different inputs and conditions. Control statements include if, if-else, switch, for, while, do-while, and more.

2.2 Demonstrate if, if else, and if else ladder and compute it. The if statement is used to execute a block of code if a given condition is true. It is the simplest form of a control statement. The if-else statement extends the if statement by allowing an alternative block of code to be executed if the condition is false. The if-else ladder is a series of if-else statements where multiple conditions are evaluated in sequence, and the corresponding code block is executed based on the first true condition.

2.3 Demonstrate the switch statement and its functions. The switch statement provides an efficient way to select one of many possible code blocks to execute based on the value of an expression. It evaluates the expression and matches it with specific cases defined within the switch statement. The code block associated with the matching case is executed. The switch statement is particularly useful when dealing with multiple options or menu-driven programs.

2.4 Illustrate the for loop and deduce its usage. The for loop is used to repeat a block of code a specific number of times. It consists of three parts: initialization, condition, and increment/decrement. The loop executes as long as the condition is true, with the initialization and increment/decrement statements controlling the loop's progress. The for loop is commonly used when the number of iterations is known or determined in advance.

2.5 Illustrate the do-while loop and deduce its usage. The do-while loop is a variation of the while loop. It executes a block of code at least once, regardless of the condition, and then continues to execute as long as the condition is true. The condition is evaluated at the end of each iteration, allowing the loop to execute until the condition becomes false. The do-while loop is useful when you want to ensure that a block of code executes at least once.

2.6 Illustrate the while loop and deduce its usage. The while loop repeatedly executes a block of code as long as a given condition is true. It evaluates the condition before each iteration. If the condition is true, the loop continues executing. If the condition is false, the loop terminates, and control moves to the next statement after the loop. The while loop is used when the number of iterations is not known in advance.

2.7 Classify loop control statements and compare their features. Loop control statements are used to control the flow of loops. Some common loop control statements in C# are break, continue, and goto. The break statement is used to exit a loop prematurely. The continue statement skips the remaining code within the loop and proceeds to the next iteration. The goto statement transfers control to a labeled statement within the same method. These control statements provide flexibility in controlling loop execution and handling specific scenarios.

Chapter 3: Arrays

3.1 Introduce the arrays and their usage. Arrays are data structures used to store multiple values of the same data type. They provide a convenient way to work with collections of data. Arrays in C# are fixed in size and have a specific type for their elements. They allow for efficient storage and retrieval of data.

3.2 Demonstrate the declaration and initialization of an array. In C#, arrays are declared by specifying the data type of the elements followed by the array name and square brackets ([]). The size of the array is determined at the time of declaration. Arrays can be initialized during declaration or separately using the assignment operator (=) and comma-separated values enclosed in curly braces ({ }).

3.3 Illustrate data access from an array. Accessing elements in an array is done using indexes. The index represents the position of an element within the array. Indexing starts from 0 for the first element. Elements can be accessed or modified using the array name followed by the index enclosed in square brackets ([]).

3.4 Introduce multidimensional arrays. Multidimensional arrays in C# are arrays with more than one dimension. They can have two or more dimensions, forming a matrix or a table-like structure. Multidimensional arrays allow for organizing data in a tabular format and accessing elements using multiple indices.

3.5 Compare and deduce the applications of jagged arrays, param arrays, and the array class. Jagged arrays in C# are arrays of arrays, where each element can be an array of different lengths. They allow for creating irregularly shaped arrays. Param arrays are used to create methods that can accept a variable number of arguments. The array class provides various methods and properties for working with arrays, such as sorting, searching, resizing, and more. Understanding the applications of these different types of arrays is important for efficient data storage and manipulation.

Chapter 4: Strings

4.1 Introduce strings, their usages, and functions. Strings are used to represent sequences of characters in C#. They are commonly used for text manipulation, input/output operations, and string processing. Strings can store alphanumeric characters, symbols, and special characters. They play a vital role in various applications, such as handling user input, storing data, and displaying messages.

4.2 Demonstrate the creation of a string object. In C#, strings can be created using string literals or by instantiating the string class. String literals are enclosed in double quotation marks (" "). String objects can be created using the new keyword and assigning a value to them.

4.3 Demonstrate the methods of the string class and deduce their usages. The string class in C# provides a wide range of methods for manipulating and processing strings. These methods enable operations such as concatenation, substring extraction, case conversion, searching, replacing, and more. Understanding the functionality and usage of these string class methods is essential for effective string manipulation in C#.

4.4 Introduce string functions and examine their usage. In addition to the methods provided by the string class, C# also includes built-in functions specifically designed for string operations. These functions are part of the System namespace and can be used without instantiating a string object. Examples of string functions include Length, Compare, Concat, Format, and more. Understanding and utilizing these functions can simplify string processing tasks and improve code efficiency.

Chapter 5: Structures

5.1 Introduction to structures, their features, and necessities. Structures in C# are user-defined data types that can contain members such as fields, properties, and methods. They are used to encapsulate related data and provide a convenient way to organize and manipulate complex data structures. Structures are value types and are allocated on the stack, making them more efficient in terms of memory usage and performance.

5.2 Demonstration of defining structures and their usage. To define a structure in C#, the struct keyword is used, followed by the structure name and a code block that contains the members. Members can include fields, properties, methods, and constructors. Structures can be instantiated like objects and their members accessed using dot notation.

5.3 Compare and evaluate class vs. structure and demonstrate it. Classes and structures are both used to define custom data types in C#. While classes are reference types, structures are value types. Classes support inheritance and are allocated on the heap, while structures do not support inheritance and are allocated on the stack. Choosing between classes and structures depends on factors such as the desired behavior, memory requirements, and performance considerations. Understanding the differences and making the appropriate choice is crucial for designing efficient and maintainable code.

Chapter 6: Pointers

6.1 Introduce pointers, their features, and applications. Pointers in C# provide direct memory access and manipulation. They are variables that hold memory addresses and allow for low-level operations such as reading and writing data directly to memory. Pointers are primarily used for advanced scenarios such as working with unmanaged code, interoperation with other programming languages, and performance optimizations.

6.2 Differentiate between advantages and disadvantages of pointers. Pointers offer several advantages, including direct memory access, fine-grained control over memory resources, and the ability to interact with unmanaged code. However, they also come with disadvantages such as potential security risks, memory leaks, and difficulty in understanding and debugging code that uses pointers. Pointers should be used with caution and only in scenarios where their benefits outweigh their drawbacks.

6.3 Demonstrate the access of data value using a pointer. To access data using a pointer in C#, the address-of operator (&) is used to get the memory address of a variable. The dereference operator (\*) is used to access the value at a memory address. By manipulating pointers, you can read, write, and modify data directly in memory.

6.4 Illustrate the passing of pointers as parameters to methods. C# allows passing pointers as method parameters, enabling direct manipulation of data outside the method's scope. This can be useful when modifying large data structures or when working with external resources. However, caution must be exercised to ensure proper memory management and avoid unintended side effects.

6.5 Demonstrate the access of array elements using a pointer. Pointers can be used to access array elements directly in memory. By incrementing the pointer address and dereferencing it, individual elements of an array can be accessed or modified. This can provide a more efficient way of working with arrays, especially in scenarios where direct memory manipulation is required.

Chapter 7: Working with Databases

7.1 Introduce databases, their features, and necessity in a programming environment. Databases are structured repositories used for storing, managing, and retrieving large amounts of data. They provide a structured and efficient way to organize and access data. Databases are crucial in programming environments as they enable the storage and retrieval of information for applications, allowing for data-driven functionality and persistence.

7.2 Demonstrate the database environment setup and configure the requirements. Setting up a database environment involves installing and configuring database management system (DBMS) software, such as Microsoft SQL Server, MySQL, or Oracle. This includes configuring server settings, establishing connections, creating databases, and defining database schemas. The database environment setup is essential for interacting with databases from C# programs.

7.3 Illustrate the connection of a C# program with a database. To establish a connection between a C# program and a database, a database connection string is used. The connection string contains information such as the database server, credentials, and other connection parameters. C# provides libraries and APIs, such as ADO.NET, to connect to various database systems, execute queries, and retrieve or modify data.

7.4 Demonstrate the read and write operations from the database. Once connected to a database, C# programs can perform read and write operations using SQL queries or stored procedures. Read operations retrieve data from the database, while write operations insert, update, or delete data. The retrieved data is typically stored in C# objects or data structures for further processing or presentation. Proper handling of database connections, transactions, and error handling is crucial for robust and secure database operations.

Chapter 1:

1.1 Introduction to C# - its features and applications:

C# (pronounced "C sharp") is a powerful, object-oriented programming language developed by Microsoft. It combines the features of C and C++ with the simplicity and ease of use of Visual Basic. Some key features of C# include:

* Object-Oriented: C# supports encapsulation, inheritance, and polymorphism, allowing developers to create reusable and modular code.
* Type Safety: C# enforces strong type checking at compile-time, reducing the likelihood of runtime errors.
* Garbage Collection: C# automatically manages memory by reclaiming objects that are no longer in use, reducing the burden on developers.
* Integration with .NET Framework: C# is tightly integrated with the .NET Framework, providing access to a vast library of pre-built classes and components.
* Cross-Platform Development: With the introduction of .NET Core, C# can be used to build applications that run on various platforms, including Windows, macOS, and Linux.

**Applications of C# include:**

* Desktop Applications: C# can be used with frameworks like Windows Presentation Foundation (WPF) and Windows Forms to create intuitive and feature-rich desktop applications.
* Web Applications: C# can be used with ASP.NET and ASP.NET Core to develop robust and scalable web applications.
* Mobile Applications: C# can be used with Xamarin or .NET MAUI to build cross-platform mobile apps for iOS and Android.
* Game Development: C# is widely used in game development with frameworks like Unity, providing a high-level language for scripting game logic and behavior.
* IoT and Embedded Systems: C# can be used with frameworks like .NET IoT to build applications for Internet of Things (IoT) devices and embedded systems.

**1.2 Structure of C#:**

The structure of a C# program consists of various elements organized in a specific manner. A basic structure of a C# program includes:

1. **Namespace Declaration:** It defines the namespace in which the program resides, helping to organize and group related code.

namespace MyNamespace

{

// Program code goes here

}

1. **Class Declaration:** It defines a class, which is a blueprint for creating objects. Classes encapsulate data and behavior into a single unit.

namespace MyNamespace

{

class MyClass

{

// Class members go here

}

}

1. **Main Method:** It serves as the entry point for the program and contains the code that will be executed first.

namespace MyNamespace

{

class MyClass

{

static void Main(string[] args)

{

// Code to be executed goes here

}

}

}

**1.3 Variables in C#:**

Variables in C# are used to store and manipulate data. They have a data type, a name, and a value. C# supports various built-in data types, including integers, floating-point numbers, characters, booleans, and strings. Here are some examples of variable declarations and assignments:

int age; // Declaration of an integer variable

age = 25; // Assignment of a value to the age variable

double height = 1.75; // Declaration and assignment of a floating-point variable

char grade = 'A'; // Declaration and assignment of a character variable

bool isStudent = true; // Declaration and assignment of a boolean variable

string name = "John"; // Declaration and assignment of a string variable

**1.4 Identifiers in C#:**

Identifiers in C# are used to give names to various program elements such as variables, classes, methods, and namespaces. They must follow specific rules and conventions. Here are some guidelines for naming identifiers in C#:

* They must start with a letter or an underscore.
* They can contain letters, digits, and underscores.
* They are case-sensitive, meaning "myVariable" and "myvariable" are considered different.
* They cannot be a reserved keyword.
* It is recommended to use descriptive and meaningful names.

Examples of valid identifiers:

int age;

string firstName;

double \_salary;

**1.5 Keywords in C#:**

Keywords in C# are reserved words that have a special meaning in the language and cannot be used as identifiers. They represent language constructs and control the behavior of the program. Examples of C# keywords include if, else, switch, for, while, class, int, string, and many others.

int age = 25;

if (age > 18)

{

Console.WriteLine("You are an adult.");

}

else{

Console.WriteLine("You are a kid.");

}

In the example above, if and else are keywords used for conditional statements, while int is a keyword used for declaring an integer variable.

**1.6 Data Types in C#:**

Data types in C# specify the type of data that can be stored in variables. C# supports two categories of data types: value types and reference types.

* **Value types** represent simple data types, such as integers, floating-point numbers, booleans, and characters. They are stored directly in memory and occupy a fixed size.

Examples of value types:

int age = 25;

double height = 1.75;

bool isStudent = true;

char grade = 'A';

* **Reference types** represent complex data types, such as classes, interfaces, arrays, and delegates. They store a reference (memory address) to the actual data, which is allocated on the heap. Examples of reference types:

string name = "John";

object obj = new object();

int[] numbers = new int[] { 1, 2, 3 };

**1.7 C# Type Conversion:**

Type conversion, also known as type casting, is the process of converting a value from one data type to another. C# supports two types of type conversions: implicit and explicit.

* **Implicit conversion** is done automatically by the compiler when there is no risk of data loss or precision. For example:

int number = 10;

double result = number; // Implicit conversion from int to double

* **Explicit conversion** requires the use of casting operators and may result in data loss or exceptions. For example:

double value = 3.14;

int approximation = (int)value; // Explicit conversion from double to int

**1.8 C# Operators:**

Operators in C# are symbols or keywords that perform specific operations on operands (variables or values). C# supports a wide range of operators for arithmetic, assignment, comparison, logical operations, and more.

1. **Arithmetic operators** are used for performing mathematical calculations:

int a = 5;

int b = 2;

int sum = a + b; // Addition

int difference = a - b; // Subtraction

int product = a \* b; // Multiplication

int quotient = a / b; // Division

int remainder = a % b; // Modulus (remainder)

1. **Assignment operators** are used to assign values to variables:

int x = 10;

x += 5; // Equivalent to x = x + 5

x -= 3; // Equivalent to x = x - 3

1. **Comparison operators** are used to compare values:

int a = 5;

int b = 3;

bool isEqual = (a == b); // Equality comparison

bool isGreaterThan = (a > b); // Greater than comparison

bool isLessThan = (a < b); // Less than comparison

1. **Logical operators** are used for logical operations:

bool condition1 = true;

bool condition2 = false;

bool result = (condition1 && condition2); // Logical AND

bool result2 = (condition1 || condition2); // Logical OR

bool result3 = !condition1; // Logical NOT

These are just a few examples of C# operators. Understanding and using operators effectively is crucial for performing various calculations, comparisons, and logical operations in C#.

Chapter 2

**2.1 Introduce the control statements:**

Control statements in C# are fundamental constructs that enable programmers to control the flow of execution in their programs. These statements allow you to make decisions, repeat actions, and perform different operations based on certain conditions. They are essential for creating dynamic and flexible programs that can respond to various scenarios.

**2.2 if, if-else, and if-else ladder:**

* **The if statement** is one of the most basic control statements in C#. It allows you to execute a block of code if a specified condition is true. Here's the syntax:

if (condition)

{

// Code to be executed if the condition is true

}

**For example:**

int age = 18;

if (age >= 18)

{

Console.WriteLine("You are an adult.");

}

In this example, the code inside the if block will be executed only if the condition age >= 18 evaluates to true.

* **The if-else statement** extends the functionality of the if statement by providing an alternative block of code to be executed when the condition is false. Here's the syntax:

if (condition)

{

// Code to be executed if the condition is true

}else

{

// Code to be executed if the condition is false

}

**For example:**

int age = 16;

if (age >= 18)

{

Console.WriteLine("You are an adult.");

}else

{

Console.WriteLine("You are a minor.");

}

In this case, if the condition age >= 18 is false, the code inside the else block will be executed instead.

* **The if-else ladder** is used when you have multiple conditions to be checked. It allows you to specify different blocks of code for different conditions. Here's the syntax:

if (condition1)

{

// Code to be executed if condition1 is true

}else if (condition2)

{

// Code to be executed if condition1 is false and condition2 is true

}else

{

// Code to be executed if all conditions are false

}

**For example:**

int score = 85;if (score >= 90)

{

Console.WriteLine("Excellent!");

}else if (score >= 80)

{

Console.WriteLine("Good!");

}else if (score >= 70)

{

Console.WriteLine("Average.");

}else

{

Console.WriteLine("Needs improvement.");

}

In this example, different messages will be displayed based on the value of the score variable.

**2.3 switch statement and its functions:**

The switch statement provides an alternative way to perform different actions based on the value of a variable or an expression. It simplifies code that would otherwise require multiple if-else statements. The switch statement consists of multiple case labels and an optional default label.

**Here's the syntax:**

switch (expression)

{

case value1:

// Code to be executed if expression matches value1

break;

case value2:

// Code to be executed if expression matches value2

break;

// Add more cases as needed

default:

// Code to be executed if expression doesn't match any case

break;

}

**For example:**

int day = 3;

switch (day)

{

case 1:

Console.WriteLine("Monday");

break;

case 2:

Console.WriteLine("Tuesday");

break;

case 3:

Console.WriteLine("Wednesday");

break;

// Add more cases for other days

default:

Console.WriteLine("Invalid day");

break;

}

In this example, the output will be "Wednesday" since the value of the day variable is 3.

**2.4 for loop and its usage:**

The for loop is used to execute a block of code repeatedly for a specified number of times. It provides a convenient way to iterate over a range of values or a collection. The for loop consists of three parts: initialization, condition, and iteration.

**Here's the syntax:**

for (initialization; condition; iteration)

{

// Code to be executed in each iteration

}

**For example:**

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Value of i: " + i);

}

In this example, the for loop will execute five times, printing the value of the variable i from 1 to 5.

The initialization part is executed only once at the beginning of the loop. The condition is checked before each iteration, and if it evaluates to true, the code inside the loop is executed. After each iteration, the iteration part is executed to update the loop variable. If the condition becomes false, the loop terminates.

**2.5 do-while loop and its usage:**

The do-while loop is similar to the while loop but with a slight difference in the execution order. It executes a block of code at least once before checking the condition. If the condition is true, the loop continues to execute.

**Here's the syntax:**

do

{

// Code to be executed

} while (condition);

**For example:**

int i = 1;

do{

Console.WriteLine("Value of i: " + i);

i++;

} while (i <= 5);

In this example, the code inside the do block will be executed first, and then the condition i <= 5 will be checked. If the condition is true, the loop will continue executing. If the condition is false, the loop will terminate.

The key difference between a do-while loop and a while loop is that the do-while loop guarantees at least one execution of the loop body, regardless of the condition.

**2.6 while loop and its usage:**

The while loop is used to execute a block of code repeatedly as long as a specified condition is true. It is suitable when the number of iterations is not known beforehand.

**Here's the syntax:**

while (condition)

{

// Code to be executed

}

**For example:**

int i = 1;

while (i <= 5)

{

Console.WriteLine("Value of i: " + i);

i++;

}

In this example, the code inside the while block will be executed only if the condition i <= 5 is true. If the condition is false initially, the code inside the while block will not execute.

The while loop checks the condition before each iteration. If the condition is true, the code inside the loop is executed. If the condition becomes false, the loop terminates.

**2.7 Loop Control Statements and their features:**

In addition to the basic control statements, C# provides loop control statements that allow you to modify the behavior of loops:

* **break statement:** It is used to terminate the innermost loop in which it appears and transfer control to the statement immediately following the loop. The break statement is commonly used to exit a loop prematurely based on certain conditions.

**For example:**

for (int i = 1; i <= 10; i++)

{

if (i == 5)

break; // Exit the loop when i equals 5

Console.WriteLine("Value of i: " + i);

}

In this example, the loop will terminate when the value of i becomes 5.

* **continue statement:** It is used to skip the remaining code inside the loop for the current iteration and proceed to the next iteration. The continue statement is commonly used to skip specific iterations based on certain conditions.

**For example:**

for (int i = 1; i <= 5; i++)

{

if (i == 3)

continue; // Skip the current iteration when i equals 3

Console.WriteLine("Value of i: " + i);

}

In this example, the value of i will not be printed when it is equal to 3.

* **goto statement:** It is used to transfer control to a labeled statement within the same method. The goto statement allows you to jump to a specific section of code based on certain conditions.

For example:

int age = 16;

if (age < 18)

goto underage; // Jump to the 'underage' label

else

Console.WriteLine("You are an adult.");

underage:

Console.WriteLine("You are underage.");

In this example, the program will jump to the label "underage" if the age is less than 18.

It's important to note that the use of the goto statement should be limited and used with caution, as it can make code harder to read and understand.

These loop control statements provide additional flexibility and control over the flow of execution within loops. Understanding and using them effectively can help you create more efficient and flexible programs.