Microsoft Visual C# .NET

Introducing C#

C# (pronounced "C-sharp") is a programming language introduced by Microsoft in 2000. C# is intended to be a simple, modern, general-purpose, object-oriented programming language.

It is part of the .NET platform and is widely used for building many types of applications.

The name C# is inspired by the musical scale. On a musical scale, sharp (#) written against a note means that the note should be made half step higher in pitch.

Why C#?

C# is popular because it is:

- Easy to learn : Clear syntax similar to C, C++, and Java.
- Object-Oriented : Supports concepts like classes, objects, inheritance, polymorphism, etc.
- Powerful : Used to build desktop apps, web apps, mobile apps, games, cloud services, and more.
- Secure and Reliable : Strong type-checking, garbage collection, and good memory management.
- Cross-Platform : Using .NET Core / .NET 6+, C# apps can run on Windows, Linux, and macOS.

Key Features of C#

- Object-Oriented Programming : Supports classes, objects, inheritance, interfaces, polymorphism.
- Strongly Typed : Variables must have a type.
- Automatic Memory Management : Garbage Collector removes unused objects.
- Rich Library : .NET provides thousands of built-in functions.
- Asynchronous Programming: Using async and await.

Using C# for Writing Programs

Similar to the various programming languages, C# also has some predefined keywords that can be used for writing programs.
For example, class is a keyword in C# that is used to define classes. Keywords are reserved words that have a special meaning.

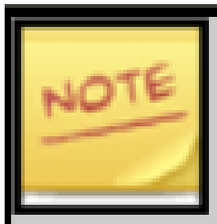
Further, the syntax for C# defines rules for grammatical arrangement of these keywords. For example, the syntax of declaring a class in C# is:

class <class name> { ... }

In the preceding syntax, the braces, known as delimiters, are used to indicate the start and end of a class body.

Classes in C#

C# classes are the primary building blocks of the language. C# also provides certain predefined set of classes and methods.
These classes and *methods* provide various basic functionalities that you may want to implement in your application.
For example, if you want to perform some input/output operations in your application, you can use predefined classes available in the System.IO namespace.



A namespace is a collection of classes. C# provides some predefined namespaces that contain classes, which provide commonly used functionality. To use the classes defined in these namespaces, you need to include the relevant namespace in your program.

You can also create your own namespaces that contain classes that you need to use across several programs.

Basic Structure of a C# Program

```
using System;
class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello, C# World");
     }
}
```

The preceding class declaration includes the method, Main()that will display the message, “Hello, World!” on your screen.

The preceding code includes the following components:

- The classkeyword
- The class name
- The Main()method
- The System.Console.WriteLine() method
- Escape sequences

Let us discuss these components in detail.

The class Keyword

The class keyword is used to declare a class. In the preceding code, the class keyword declares the class, HelloWorld.

The Class Name

The class keyword is followed by the name of the class. In the preceding code, HelloWorld is the name of the class defined by using the class keyword.

Class Naming Conventions in C#

Class names should follow certain naming conventions or guidelines. A class name:

- Should be meaningful (strongly recommended).
- Should ideally be a noun.
- Can use either the Pascal case or Camel case. In Pascal case, the first letter is capitalized and the rest of the letters are in lower case, such as Myclass. In Camel case, the first letter is in lower case and the first letter of each subsequent word is capitalized, such as myClassor intEmployeeDetails.

Rules for Naming Classes in C#

In addition to the conventions, there are certain rules that must be followed while naming classes in C#. The name of classes:

- Must begin with a letter. This letter may be followed by a sequence of letters, digits (0-9), or '_'. The first character in a class name cannot be a digit.

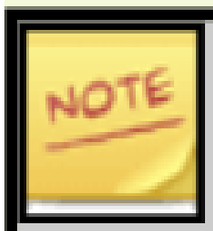
- Must not contain an embedded space or a symbol like ? -+ ! @ # % ^ & * ( ) [ ] { } . , ; : " ' / and \. However, an underscore ('_') can be used wherever a space is required.

- Must not use a keyword for a class name. For example, you cannot declare a class called public.

The Main() Method

- The first line of code that a C# compiler looks for in the source file is the Main()method. This method is the entry point for an application. This means that the execution of code starts from the Main()method.

- The Main() method is ideally used to create objects and invoke the methods of various classes that constitute the program.



`string[] args` in the preceding code is an optional argument passed to the `Main()` method. Arguments that are passed to methods will be explained later in the course.

You must include a `Main()` method in a class to make your program executable.

The System.Console.WriteLine() Method

Console is a class that belongs to the System namespace. The Console class includes a predefined WriteLine()method.

This method displays the enclosed text on the user's screen. The Console class has various other methods that are used for various input/ output operations.

The dot character (.) is used to access the WriteLine()method, which is present in the Console class of the System namespace.

The System.Console.WriteLine()statement can also be written as Console.WriteLine() if the statement, using System is included as the first line of the code.

The following code snippet is an example of the Console.WriteLine()method:

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

Escape Sequences

To display special characters, such as the new line character or the backspace character, you need to include appropriate escape sequences in your code. An escape sequence is a combination of characters consisting of a backslash (\ ) followed by a letter or a combination of digits. The following table lists some of the escape sequences provided in C#.

Escape Sequence	Sequence Description
\'	Single quotation
\ "	markDouble quotation mark
\\	Backslash
\0	NULL
\a	Alert
\b	Backspace
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Activity: Creating Classe

Problem Statement

Write a program to display the following details on the screen:

Event: Tennis Match
Venue: Star Sports Complex
Time: 4:00 p.m. to 6:00 p.m

```
public class Event
{
        public static void Main(string[] args)
      {
System.Console.WriteLine("Event: \tTennis Match\n");

System.Console.WriteLine("Venue: \tStar Sports Complex\n");

System.Console.WriteLine("Time: \t4:00 p.m. to 6:00 p.m. \n");
     }
}
```

Using Variables

A variable in C# is a named memory location used to store data and its value can change during program execution. Each variable must be declared with a specific data type that defines the kind of values it can hold.

Types of Variables

1) Local variables
2) Instance variables or Non - Static Variables
3) Static Variables or Class Variables
4) Constant Variables
5) Read only Variables

Local Variable

A variable defined within a block or method or constructor is called local variable.

- These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.

- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variables only within that block.

Example

```
using System;
class StudentDetails {
        public void StudentAge(){

        // local variable age
        int age = 0;

        age = age + 10;
        Console.WriteLine("Student age is : " + age);
    }

    public static void Main(String[] args)
    {
        // Creating object
        StudentDetails obj = new StudentDetails();

        // calling the function
        obj.StudentAge();
    }
}
```

Instance Variables or Non - Static Variables

Instance variables in C# are non-static variables declared inside a class but outside methods or constructors.

They are created when an object is instantiated and destroyed when the object is destroyed and can have access specifiers.

```
using System;

class Marks {
    // These variables are instance variables. These variables
are in a class and are not inside any function
    int engMarks;
    int mathsMarks;
    int phyMarks;

    public static void Main(String[] args)
    {
        // first object
        Marks obj1 = new Marks();
        obj1.engMarks = 90;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 93;

        // displaying marks for first object
        Console.WriteLine("Marks for first object:");
        Console.WriteLine(obj1.engMarks);
        Console.WriteLine(obj1.mathsMarks);
        Console.WriteLine(obj1.phyMarks);
    }
}
```

Static Variables or Class Variables

Static variables in C# are class-level variables declared with the static keyword inside a class but outside methods or constructors.

Only one copy exists per class, shared by all objects and they are created at program start and destroyed when execution ends.

Note: To access static variables, there is no need to create any object of that class, simply access the variable as:

class_name.variable_name;

Example

```
using System;
class Emp {

    // static variable salary
    static double salary;
    static String name = "Akshat";

    public static void Main(String[] args)
    {
        // accessing static variable without object
        Emp.salary = 100000;
         Console.WriteLine(Emp.name + "'s average salary:" +
Emp.salary);
    }
}
```

Difference between Instance variable & Static variable

- Each object has its own copy of instance variables, whereas only one copy of a static variable exists per class, shared by all objects.

- Changes in an instance variable affect only that object, while changes in a static variable are reflected across all objects.

- Instance variables are accessed through object references, whereas static variables can be accessed directly using the class name.

- Instance variables are initialized every time an object is created (n times for n objects), while a static variable is initialized only once in the class lifecycle.

Constants Variables

If a variable is declared by using the keyword "const" then it as a constant variable and these constant variables can't be modified once after their declaration, so it's must initialize at the time of declaration only.

```
class Test
{
 public const int MAX = 100;
         public void Display()
        {
            // MAX = 200; ✕ Error
             Console.WriteLine(MAX);
        }
}
```

Important Points about Constant Variables:

- The behavior of constant variables will be similar to the behavior of static variables i.e. initialized one and only one time in the life cycle of a class and doesn't require the instance of the class for accessing or initializing.

- The difference between a static and constant variable is, static variables can be modified whereas constant variables can't be modified once it declared.

Read-Only Variables

A read only variable in C# is a variable whose value can only be assigned at the time of declaration or within a constructor of the same class and once assigned, it cannot be modified for the lifetime of the object.

Example

```
class Test
{
    public readonly int id;

    public Test(int x)
    {
        id = x;   // Allowed
    }

    public void Display()
    {
        // id = 20; ✕ Error
        Console.WriteLine(id);
    }
}
```

```
using System;
class Program {

    // instance variable
    int a = 80;

    // static variable
    static int b = 40;

    // Constant variables
    const float max = 50;

    // readonly variables
    readonly int k =100;

    public static void Main()
    {

        Program obj = new Program();

        Console.WriteLine("The value of a is = " + obj.a);
        Console.WriteLine("The value of b is = " + Program.b);
        Console.WriteLine("The value of max is = " + Program.max);
        Console.WriteLine("The value of k is = " + obj.k);
    }
}
```

const vs readonly

Feature	const	readonly
Assignment time	Compile time	Runtime
Where to assign	Only at declaration	Declaration or constructor
Static by default	Yes	No
Object-specific value	✗ No	☑ Yes
Supports reference types	✗ No	☑ Yes
Performance	Faster	Slightly slower

Naming Variables in C#

In C#, the following rules are used for naming variables:

- A variable name must begin with a letter or an underscore ('_'), which may be followed by a sequence of letters, digits (0-9), or underscores. The first character in a variable name cannot be a digit.

- A variable name should not contain any embedded spaces or symbols, such as ? ! @ # + -% ^ & * ( ) [ ] { } . , ; : " ' /and \. However, an underscore can be used wherever a space is required, like High_Score.

- A variable name must be unique. For example, to store four different numbers, four unique variable names need to be used.

- A variable name can have any number ofcharacters. Keywords cannot be used as variable names. For example, you cannot declare a variable named classas it is a keyword in C#.

Some examples of valid variable names are:

- Game_level
- High_score123
- This_variable_name_is_very_long

Some examples of invalid variable names are:

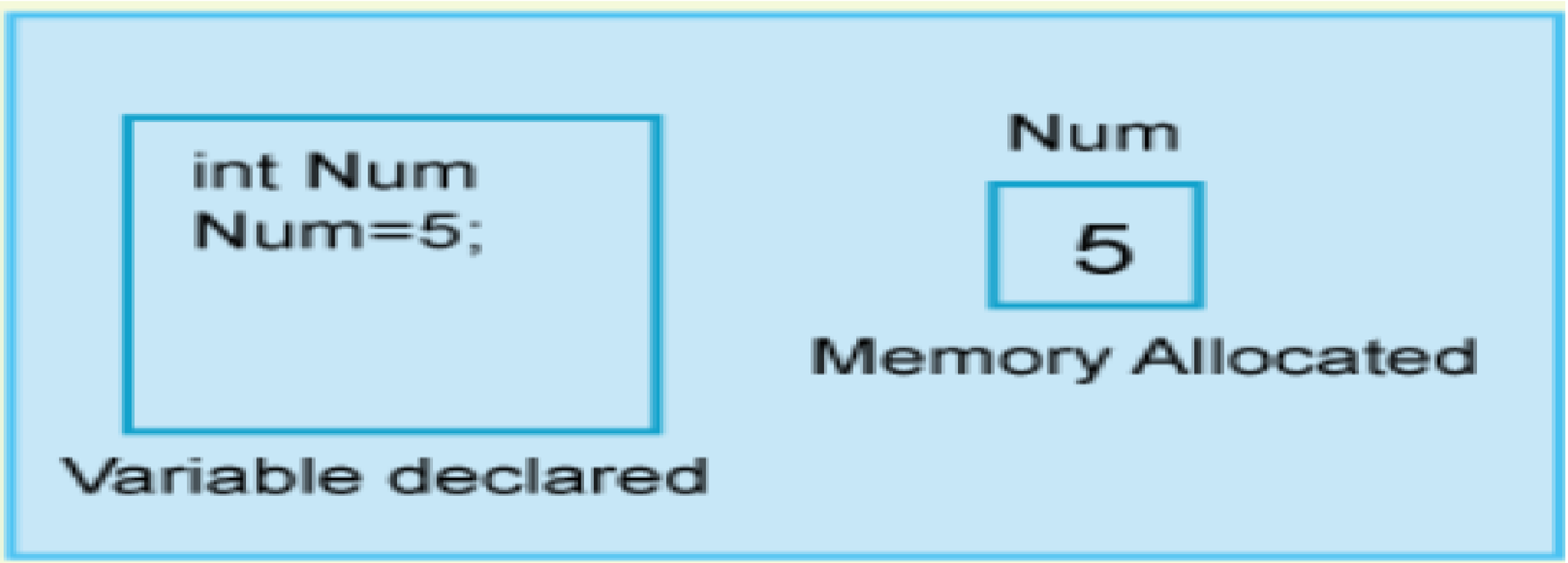
- #score
- $trank

C# is a case-sensitive language. This means that the `TennisPlayerName` *variable is not the same as the* `tennisplayername` *variable. In other words, Uppercase letters are considered distinct from lowercase letters.*

Types of Data Types

C# supports the following data types:

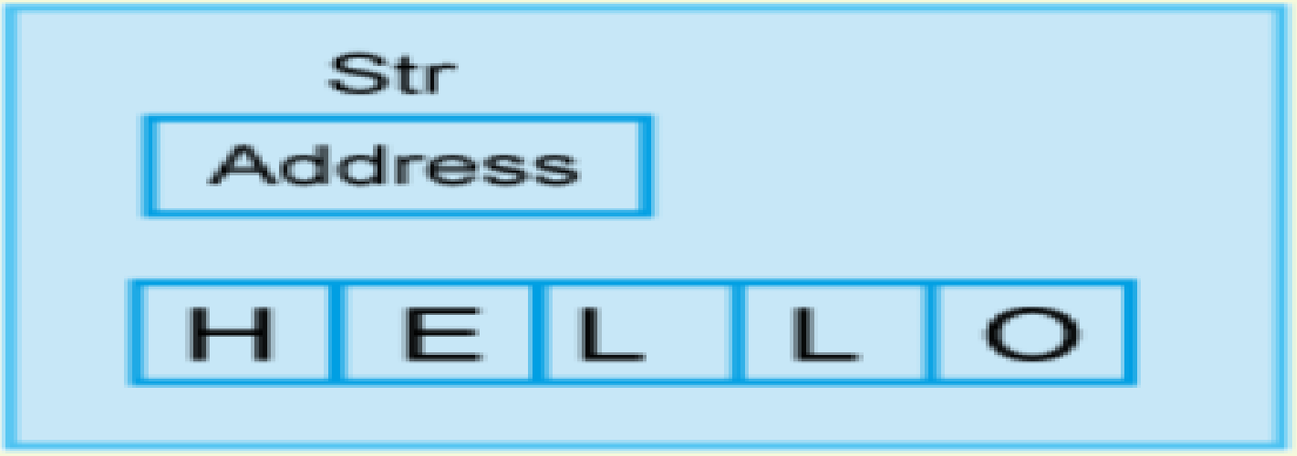
Value types: The value types directly contain data. Some examples of value types are char, int, and float, which can be used for storing alphabets, integers, and floating point values, respectively. When you declare an int variable, the system allocates memory to store the value. The following figure shows the memory allocation of an int variable.



Reference types:

The reference type variables, instead of containing data, contain a reference (address) to the data stored in the memory. More than one reference type variable can be created to refer to the same memory location. This means that if the value in the referenced memory location is modified, all the referring variables automatically reflect the changed value.

The example of a reference type is the string data type. The following figure shows the memory allocation of a string value HELLO in a variable named Str.



Memory Allocation in Reference Type

The Dynamic Data Type

A variable declared with the dynamic data type can be used to store a value of any type. A variable of this data type is declared by using the dynamic keyword. Consider the following statement: dynamic id; In the preceding statement, the variable id is declared as a dynamic variable. This means that it can be used to store any type of value, as shown in the following statements:

- id = 1; //id is considered as int
- id = "A001"; //id is considered as string

In the preceding statements, the variable, id inherits the data type of the assigned value. In the first assignment statement, id is assigned an int value. Therefore, it is treated as an integer. In the second assignment statement, it is assigned a string value. Therefore, it is considered a string.

Dynamic variables can use the methods associated with the data type they are resolved to. For example, a dynamic variable containing a string value can access all the string related attributes and methods, as shown in the following code snippet:

```
dynamic d = "Sample";
int len = d.Length;
```

For dynamic variables, type checking is done at run time and not at compile time. This is known as duck typing.
For example, consider the following code snippet:

```
dynamic var = 1;
int len = var.Length;
```

In the preceding code snippet, no error will be thrown at compile time because dynamic variables are allowed to access members associated with any type. However, at run time the code will generate an error, as var is resolved as an int type and length property is not associated with int data type.

Accepting and Storing Values in Variables

At times programs need to accept a value from a user and store the value in a variable to perform some processing. To understand how to accept a value from a user and store it in a variable, consider the following code snippet:

```
String name;

name = Console.ReadLine();
```

In the preceding code snippet, the Console.ReadLine()method is used to accept data from the user and store it in a variable named name.

The Console.ReadLine()method is a method of the Console class, which is a part of the System namespace.

The Console.ReadLine()method, by default, accepts the data in the string format. If you want to store the data in any other format, you need to convert the data type of the data provided by the user. For this purpose, you can use the Convert()method. This method informs the compiler to convert one data type to the other. This is known as explicit conversion.

For example, consider the following code snippet:

```
int Number;
Number = Convert.ToInt32 (Console.ReadLine());
```

Here, the Convert.ToInt32()method converts the data provided by the user to the int data type.

In addition to explicit conversion, there are instances when the compiler performs implicit conversions, which are done automatically (without any explicit coding). For example, implicit conversion converts the int data type to float or float data type to int, automatically.

Just a Minute

The Console.ReadLine() method, by default, accepts the data in the _____format.

- string
- int
- char
- float

Submit

Activity: Writing and Executing a C# Program

Problem Statement

David is the member of a team that is developing the Automatic Ranking software for a tennis tournament. He has been assigned the task of creating a program that should accept the following details of a tennis player and display the same on the screen:

- Name
- Rank

You need to help David to create the program.

```
using System;
class TennisPlayer
{
string TennisPlayerName;
int Rank;
```

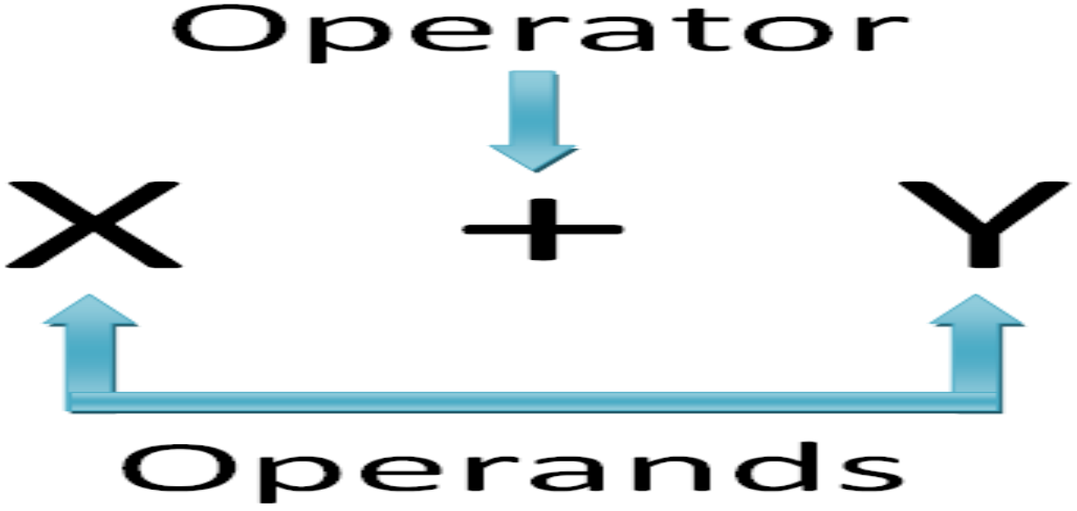
```
        public void PrintPlayerDetails()
        {
                Console.WriteLine("The details of the Tennis Players are: ");
                Console.WriteLine("Name: ");
                Console.WriteLine (TennisPlayerName);
                Console.WriteLine("Rank: ");
                Console.WriteLine(Rank);
        }

        public void GetPlayerDetails ()
        {
                Console.WriteLine("Enter the details of the Tennis Players ");
                Console.WriteLine("\n Enter TennisPlayer Name: ");
                TennisPlayerName=Console.ReadLine();
                Console.WriteLine("Rank: ");
                Rank=Convert.ToInt32 (Console.ReadLine());
        }
}
class Tennis
{
        public static void Main(string[] args)
        {
                TennisPlayer P1=new TennisPlayer();
                P1.GetPlayerDetails();
                P1.PrintPlayerDetails();
        }
}
```

Operators and Programming Constructs

An operator is a set of one or more characters that is used for computations or comparisons. Operators can change one or more data values, called operands, into a new data value.
Consider the following expression:

X + Y

The preceding expression uses two operands, X and Y, and an operator, +, to add the values of both the variables. The following figure shows the operator and operands used in the preceding expression.



Types of Operators in C#

You can use the following types of operators in C# programs:

- Arithmetic operators
- Arithmetic assignment operators
- Unary operators
- Comparison operators
- Logical operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables. The following table describes the commonly used arithmetic operators.

Operator	Description	Example
+	Used to add two numbers	X=Y+Z; If Y is equal to 20 and Z is equal to 2, X will have the value 22.
−	Used to subtract two numbers	X=Y−Z; If Y is equal to 20 and Z is equal to 2, X will have the value 18.
*	Used to multiply two numbers	X=Y*Z; If Y is equal to 20 and Z is equal to 2, X will have the value 40.
/	Used to divide one number by another	X=Y/Z; If Y is equal to 21 and Z is equal to 2, X will have the value 10.
%	Used to divide two numbers and return the remainder. The operator is called as modulus operator	X=Y%Z; If Y is equal to 21 and Z is equal to 2, X will contain the value 1.

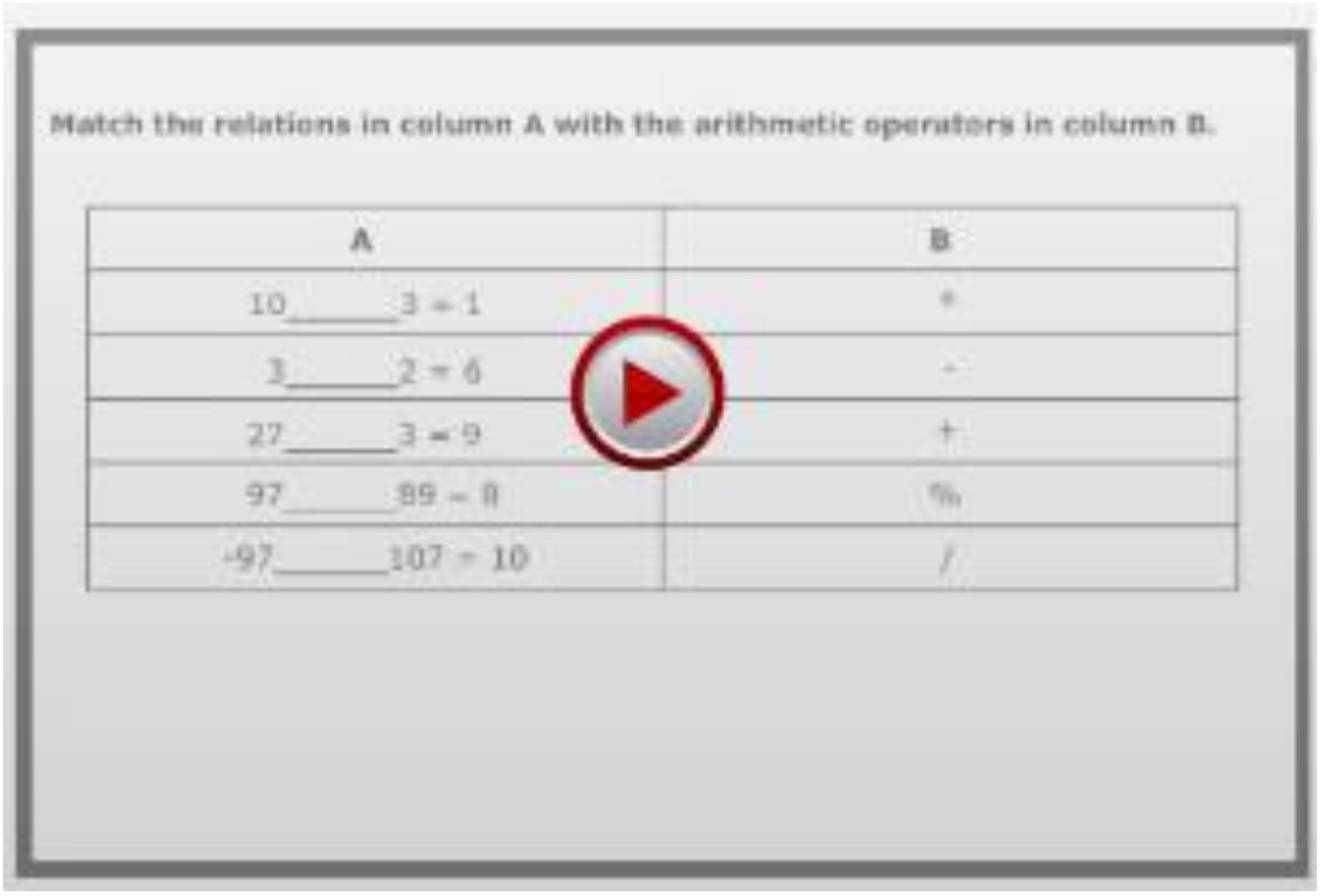
Arithmetic Assignment Operators

Arithmetic assignment operators are used to perform arithmetic operations on two given operands and to assign the resultant value to any one of them.
The following table lists the usage and describes the commonly used arithmetic assignment operators.

Operator	*Usage*	*Description*
+=	X+=Y;	*Same as:* X = X + Y;
-=	X-=Y;	*Same as:* X = X - Y;
*=	X*=Y;	*Same as:* X = X * Y;
/=	X/=Y;	*Same as:* X = X / Y;
%=	X%=Y;	*Same as:* X = X % Y;

Let's Practice

Match the relations in column A with the arithmetic operators in column B.



Unary Operators

Unary operators are used to increment or decrement the value of an operand by 1. The following table explains the usage of the increment and decrement operators.

Operator	Usage	Description	Example
++	++Operand; (Preincrement operator) Or, Operand++; (Postincrement operator)	Used to increment the value of an operand by 1	Y = ++X; If the initial value of X is 5, after the execution of the preceding statement, values of both X and Y will be 6.This is because it will first increment the value of X by 1 and then assign it to Y. Y = X++; If the initial value of X is 5, after the execution of the preceding statement, value of X will be 6 and the value of Y will be 5. This is because it will first assign the value of X to Y and then increment the value of X by 1.
--	--Operand; (Predecrement operator) Or, Operand--; (Postdecrement operator)	Used to decrement the value of an operand by 1	Y = --X; If the initial value of X is 5, after the execution of the preceding statement, values of X and Y will be 4. This is because it will first decrement the value of X by 1 and then assign the value to Y. Y = X--; If the initial value of X is 5, after the execution of the preceding statement, value of X will be 4 and the value of Y will be 5. This is because it will first assign the value of X to Y and then decrement the value of X by 1.

Comparison Operators

Comparison operators are used to compare two values and perform an action on the basis of the result of that comparison. Whenever you use a comparison operator, the expression results in a boolean value, true or false. The following table explains the usage of some commonly used comparison operators.

Operator	Usage	Description	Example (In the following examples, the value of X is assumed to be 20 and the value of Y is assumed to be 25)
<	expression1 < expression2	Used to check whether expression1 is less than expression2	bool Result; Result = X < Y; Result will have the value true.
>	expression1 > expression2	Used to check whether expression1 is greater than expression2	bool Result; Result = X > Y; Result will have the value false.
<=	expression1 <= expression2	Used to check whether expression1 is less than or equal to expression2	bool Result; Result = X <= Y; Result will have the value true.
>=	expression1 >= expression2	Used to check whether expression1 is greater than or equal to expression2	bool Result; Result = X >= Y; Result will have the value false.
==	expression1 == expression2	Used to check whether expression1 is equal to expression2	bool Result; Result = X == Y; Result will have the value false.
!=	expression1 != expression2	Used to check whether expression1 is not equal to expression2	bool Result; Result = X != Y; Result will have the value true.

Logical Operators

Logical operators are used to evaluate expressions and return a boolean value. The following table explains the usage of logical operators.

Operator	Usage	Description	Example

&&	`expression1 && expression2`	Returns true if both `expression1` and `expression2` are true	`bool Result;`
			`string str1, str2;`
			`str1 = "Korea";`
			`str2 = "France";`
			`Result = (str1== "Korea") && (str2== "France");`
			`Console.WriteLine (Result .ToString());`
			
			The preceding code snippet will print `true` on the screen because `str1` has the value `Korea` and `str2` has the value `France`.

!	`! expression`	Returns true if the `expression` is false	`bool Result`
			`int x;`
			`x = 20;`
			`Result = (!( x == 10))`
			`Console.WriteLine(Result .ToString());`
			
			The preceding code snippet will print `true` on the screen because the expression `(x==10)` is `false`.

||	expression1 || expression2	Returns true if either expression1 or expression2 or both of them are true	bool Result; string str1, str2; str1 = "Korea"; str2 = "France"; Result = (str1== "Germany") || (str2== "France"); Console.WriteLine (Result .ToString()); The preceding code snippet will print true on the screen because str2 has the value France.
^	expression1 ^ expression2	Returns true if either expression1 or expression2 is true. It returns false if both expression1 and expression2 are true or if both expression1 and expression2 are false	bool Result; string str1, str2; str1 = "Korea"; str2 = "France"; Result = (str1== "Korea") ^ (str2== "France"); Console.WriteLine (Result .ToString()); This will print false on the screen because both the expressions are true.

Expression in C#

In C#, an expression is a combination of variables, constants, operators, and method calls that is evaluated to produce a value.

General form

operand operator operand

Example:

int result = 10 + 5;

Here, 10 + 5 is an expression.

Types of Expressions in C#

1. Arithmetic Expressions

Used for mathematical calculations.

```
int a = 10, b = 3;
int sum = a + b;
int diff = a - b;
int mul = a * b;
int div = a / b;
int mod = a % b;
```

2. Relational (Comparison) Expressions

Used to compare values. Result is true or false.

```
int x = 10, y = 20;
bool r1 = x > y;
bool r2 = x <= y;
bool r3 = x == y;
bool r4 = x != y;
```

3. Logical Expressions

Used to combine conditions.

```
bool a = true, b = false;
bool result1 = a && b;   // AND
bool result2 = a || b;   // OR
bool result3 = !a;       // NOT
```

4. Assignment Expressions

Assign values to variables.

```
int x = 10;
x += 5;   // x = x + 5
x -= 2;   // x = x - 2
```

4. Assignment Expressions

Assign values to variables.

```
int x = 10;
x += 5;   // x = x + 5
x -= 2;   // x = x - 2
```

5. Conditional (Ternary) Expression

Short form of if-else.

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
```

6. Bitwise Expressions

Operate on bits.

```
int a = 5, b = 3;

int and = a & b;

int or  = a | b;

int xor = a ^ b;

int shiftLeft  = a << 1;

int shiftRight = a >> 1;
```

C# Decision Making

Decision Making in programming is similar to decision making in real life. In programming too, a certain block of code needs to be executed when some condition is fulfilled.

A programming language uses control statements to control the flow of execution of program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

Conditional Statements of C#

There are few conditional statements of C# is mentioned below:

- if
- if-else
- if-else-if
- Nested if
- Switch
- Nested Switch

If Statement

The if statement checks the given condition. If the condition evaluates to be true then the block of code/statements will execute otherwise not.

Syntax

```
if ( condition ) {
      //code to be executed
}
```

Note: If the curly brackets { } are not used with if statements then the statement just next to it is only considered associated with the if statement.

Example :

```
using System;

public class Student
{
        public static void Main(string[] args)
        {
                string name = "Donald";

        // Using if statement
                if (name == "Donald")
                {
                        Console.WriteLine("You are Donald");
                }
        }
}
```

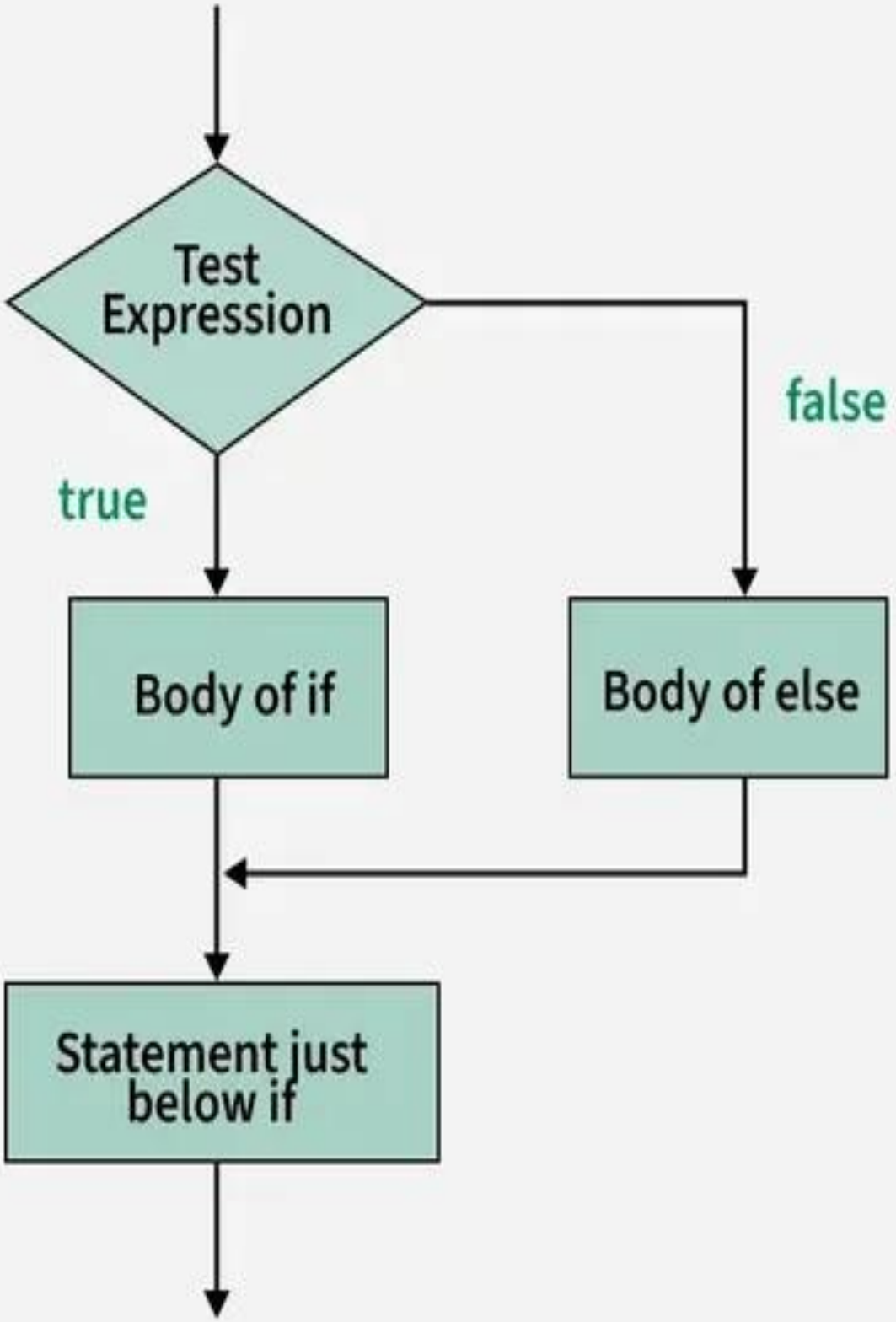
The if...else Construct

The if statement in the if...else conditional construct is followed by a logical expression where data is compared and a decision is made on the basis of the result of the comparison. The following statements depict the syntax for the if...else construct

```
if (expression)
{
        statements;
}
else
{
        statements;
}
```

want to have dinner or you want to go to sleep. The condition on which you will make a decision will be your hunger. If you are hungry, you will have dinner; else you will go to sleep. The following code snippet shows the decision making of the preceding example by using the if...else construct:

Flowchart


Test Expression
false
true
Body of if
Body of else
Statement just below if

```
using System;

namespace video
{
   class Game
  {
      static void Main(string[] args)
       {
          int Age;
         Console.WriteLine("Enter Your Age: ");
         Age = Convert.ToInt32 (Console.ReadLine());
        if (Age<12)
        {
        Console.WriteLine("Sorry! This game is for
children above 11 years.");
        }
        else
        {
           Console.WriteLine("Play the Game.");
        }
      }
  }
 }
```

Namespaces in C#

Namespaces help you to create logical groups of related classes and interfaces, which can be used by any language targeting the .NET Framework. Namespaces allow you to organize your classes so that they can be easily accessed in other applications.

Namespaces can be used to avoid any naming conflicts between classes, which have the same names. For example, you can use two classes with the same name in an application provided they belong to different namespaces. You can access the classes belonging to a namespace by simply importing the namespace into the application.

The .NET Framework uses a dot (.) as a delimiter between classes and namespaces. For example, System.Console represents the Console class of the System namespace. Namespaces are also stored in assemblies.

If-else-if ladder Statement

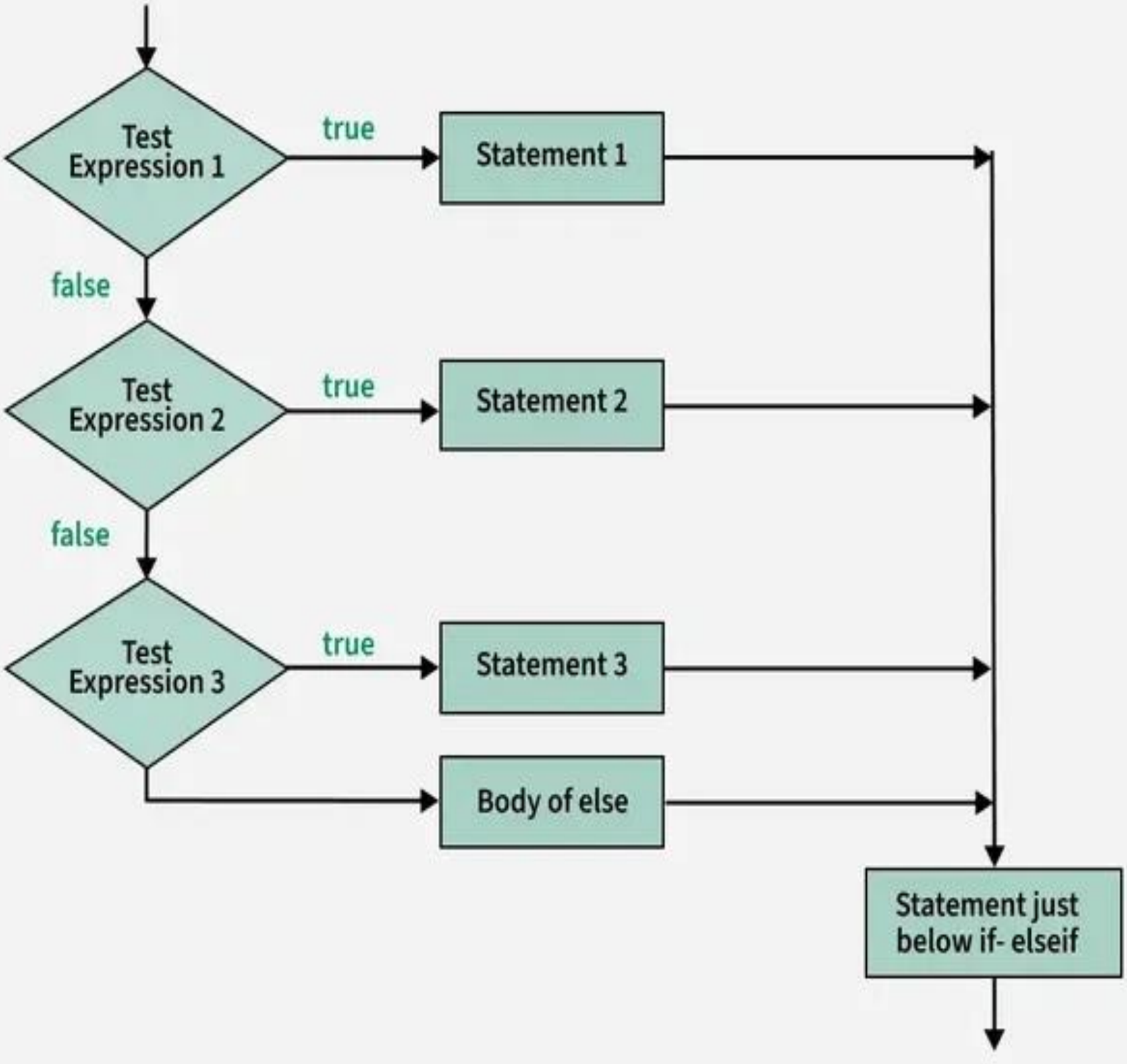
The if-else-if ladder is used when you need to test multiple conditions one after another.

- The program evaluates each condition in order.
- As soon as one condition is true, its block executes and the ladder ends.
- If none of the conditions are true, the else block (if provided) executes.

Syntax

```
f(condition1)
{
// code to be executed if condition1 is true
}
else if(condition2)
{
// code to be executed if condition2 is true
}
else if(condition3)
{
// code to be executed if condition3 is true
}
...
else
{
// code to be executed if all the conditions are false
}
```

Flowchart


Test Expression 1
true
Statement 1
false
Test Expression 2
true
Statement 2
false
Test Expression 3
true
Statement 3
Body of else
Statement just below if- elseif

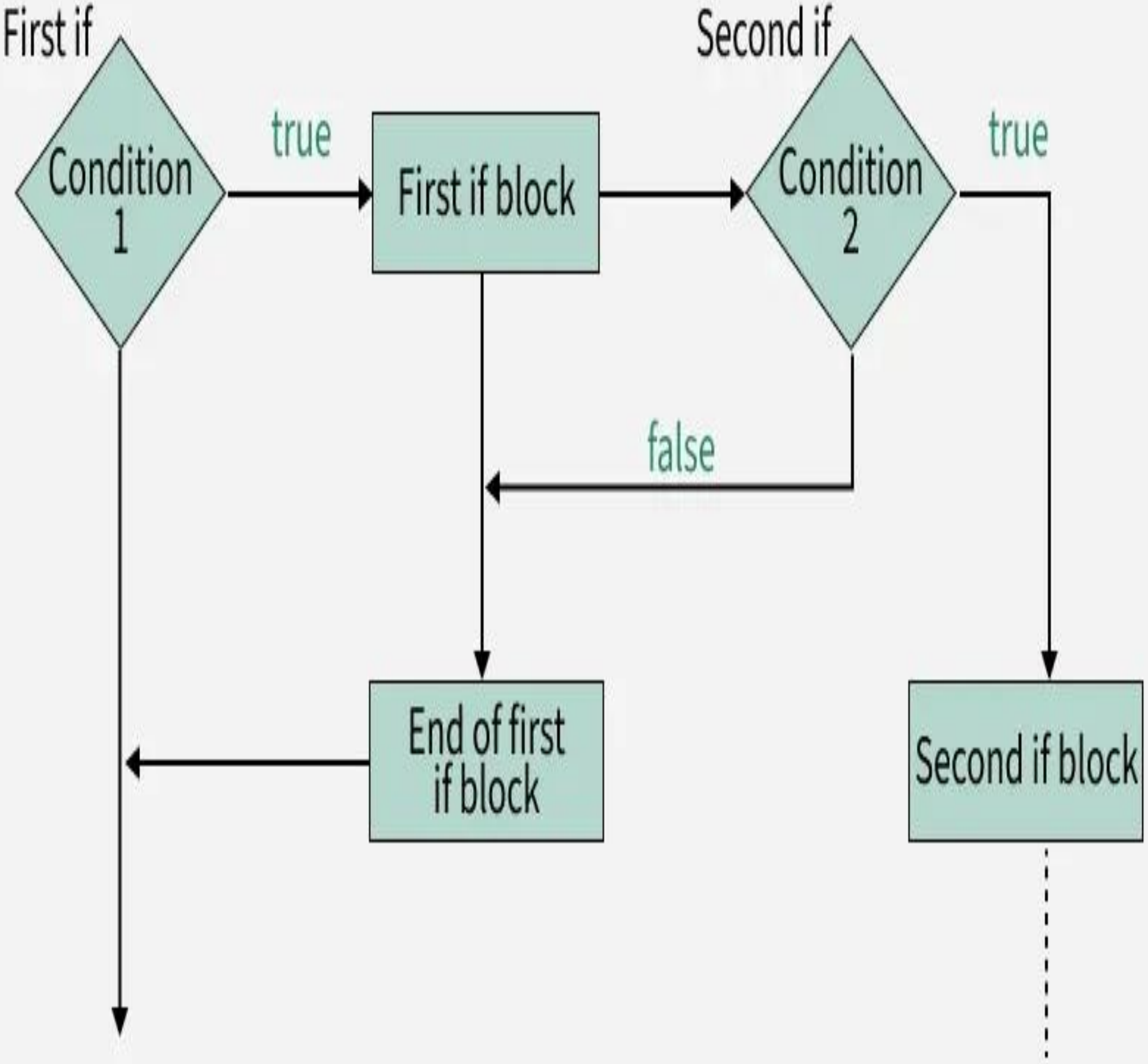
```
using System;

public class GradeAssigment
{
    public static void Main(string[] args)
    {
        int score = 85;

        if (score >= 90)
        {
            Console.WriteLine("Grade: A");
        }
        else if (score >= 80)
        {
          // This block executes because the first condition was false,
            // but this condition (85 >= 80) is true.
            Console.WriteLine("Grade: B");
        }
        else if (score >= 70)
        {
            Console.WriteLine("Grade: C");
        }
        else if (score >= 60)
        {
            Console.WriteLine("Grade: D");
        }
        else
        {
            // The 'else' block runs if none of the preceding conditions are true.
            Console.WriteLine("Grade: F");
        }
    }
}
```

Nested - If Statement

If statement inside an if statement is known as nested if. if statement in this case is the target of another if or else statement. When more than one condition needs to be true and one of the condition is the sub-condition of parent condition, nested if can be used.

Syntax

```
if (condition1)
{
      // code to be executed
      // if condition2 is true
      if (condition2)
      {
            // code to be executed
            // if condition2 is true
      }
}
```


First if
Condition 1
true
First if block
Second if
Condition 2
true
false
End of first if block
Second if block

```
using System;

public class NestedIfExample
{
    public static void Main()
    {
        int userAge = 20;
        bool hasLicense = true;

        Console.WriteLine($"Checking eligibility for a user aged {userAge} with a license status of {hasLicense}.");

        // Outer if statement: Check if the user is old enough to drive
        if (userAge >= 16)
        {
            Console.WriteLine("User is old enough to drive.");

            // Nested if statement: Check if the user also has a license
            if (hasLicense)
            {
                Console.WriteLine("User is eligible to rent a car.");
            }
            else
            {
                Console.WriteLine("User is old enough to drive, but needs a license to rent a car.");
            }
        }
        else
        {
            Console.WriteLine("User is not old enough to drive or rent a car.");
        }
    }
}
```

Switch Statement

Switch statement is an alternative to long if-else-if ladders. The expression is checked for different cases and the one match is execute.

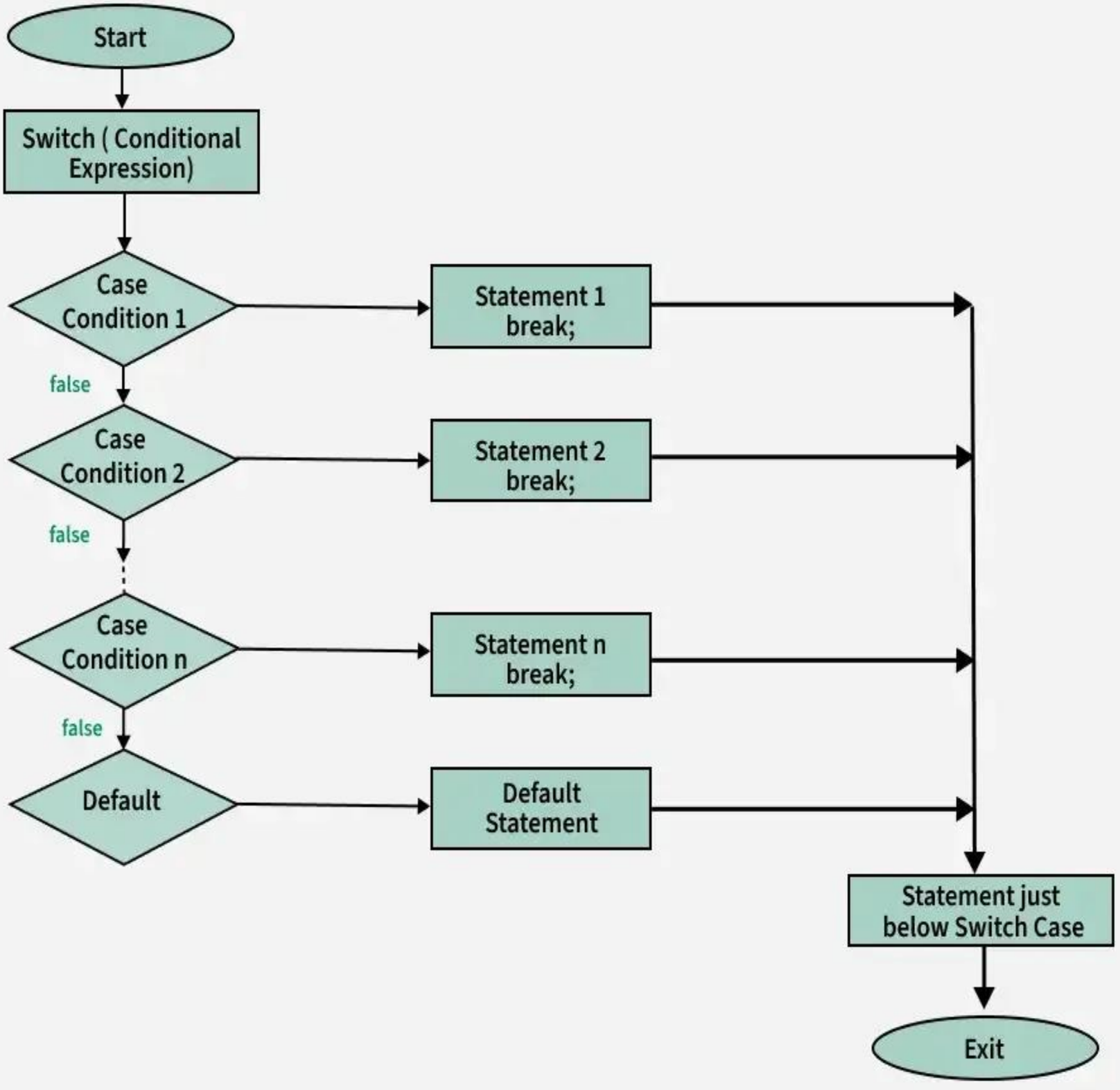
Break Statement is used to move out of the switch. If the break is not used, the control will flow to all cases below it until break is found or switch comes to an end.

There is default case (optional) at the end of switch, if none of the case matches then default case is executed.

Syntax

```
switch (expression)
{
        case value1: // statement sequence
                              break;
        case value2: // statement sequence
                              break;
        .
        .
        .
        case valueN: // statement sequence
                               break;
        default: // default statement sequence
}
```

Flowhart


Start
Switch (Conditional Expression)
Case Condition 1
Statement 1 break;
false
Case Condition 2
Statement 2 break;
false
Case Condition n
Statement n break;
false
Default
Default Statement
Statement just below Switch Case
Exit

```
using System;
public class VowelCheckOptimized
{
    public static void Main()
    {
        char inputChar = 'B';
        // Convert the input character to lowercase for simpler case
matching
        char lowerCaseChar = char.ToLower(inputChar);

        Console.WriteLine($"Checking if the character '{inputChar}' is a
vowel (using ToLower())...");

        switch (lowerCaseChar)
        {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                            Console.WriteLine($"'{inputChar}' is a vowel.");
                            break;
            default:
                Console.WriteLine($"'{inputChar}' is a consonant or not an
alphabet.");
                            break;
        }
    }
}
```

Using Loop Constructs

Loop constructs are used to execute one or more lines of code again and again. In C#, the following loop constructs can be used:

- The while loop
- The do...while loop
- The for loop

The while Loop

The while loop construct executes a block of statements till the condition given in the while loop holds true. The while statement always checks the condition before executing the statements within the loop. When the execution reaches the last statement in the while loop, the control is passed back to the beginning of the loop. If the condition still holds true, the statements within the loop are executed again. The execution of the statements within the loop continues until the condition evaluates to false.

The following statements depict the syntax for the while loop construct:

```
while (expression)
{
    statements;
}
```

The following code provides an example of the while loop construct:

```
using System;
class Variable
{
        static void Main(string[]args)
        {
                int var;
                var = 100;
                while (var < 200)
                {
        Console.WriteLine ("Value of variable is: {0}", var);
                var = var + 10;
                }
        }
}
```

You can use the break statement to exit the while loop structure. The following code snippet shows the usage of the break statement:

```
int var;
var = 100;
while (true)
{
        Console.WriteLine ("Value of var: {0} " + var);
        var = var + 10;
        if (var == 200)
                break;
}
```

In the preceding code snippet, the while (true) statement will always evaluate to true and the while block will run an indefinite number of times. In such a case, the break statement is used to exit the while loop block on the basis of the condition given within the while block. In the preceding example, the control moves out of the while loop when the value of var becomes equal to 200.

Just a Minute

State whether the following statement is True or False.
In the while loop, the while statement always checks the condition after executing the statements within the loop.

- True
- False

Submit

The do...while Loop

The do...while loop construct is similar to the while loop construct. Both will continue looping until the loop condition becomes false. However, the statements within the do...while loop are executed at least once, as the statements in the block are executed before the condition is checked.

The following statements show the syntax for the do...while loop construct:

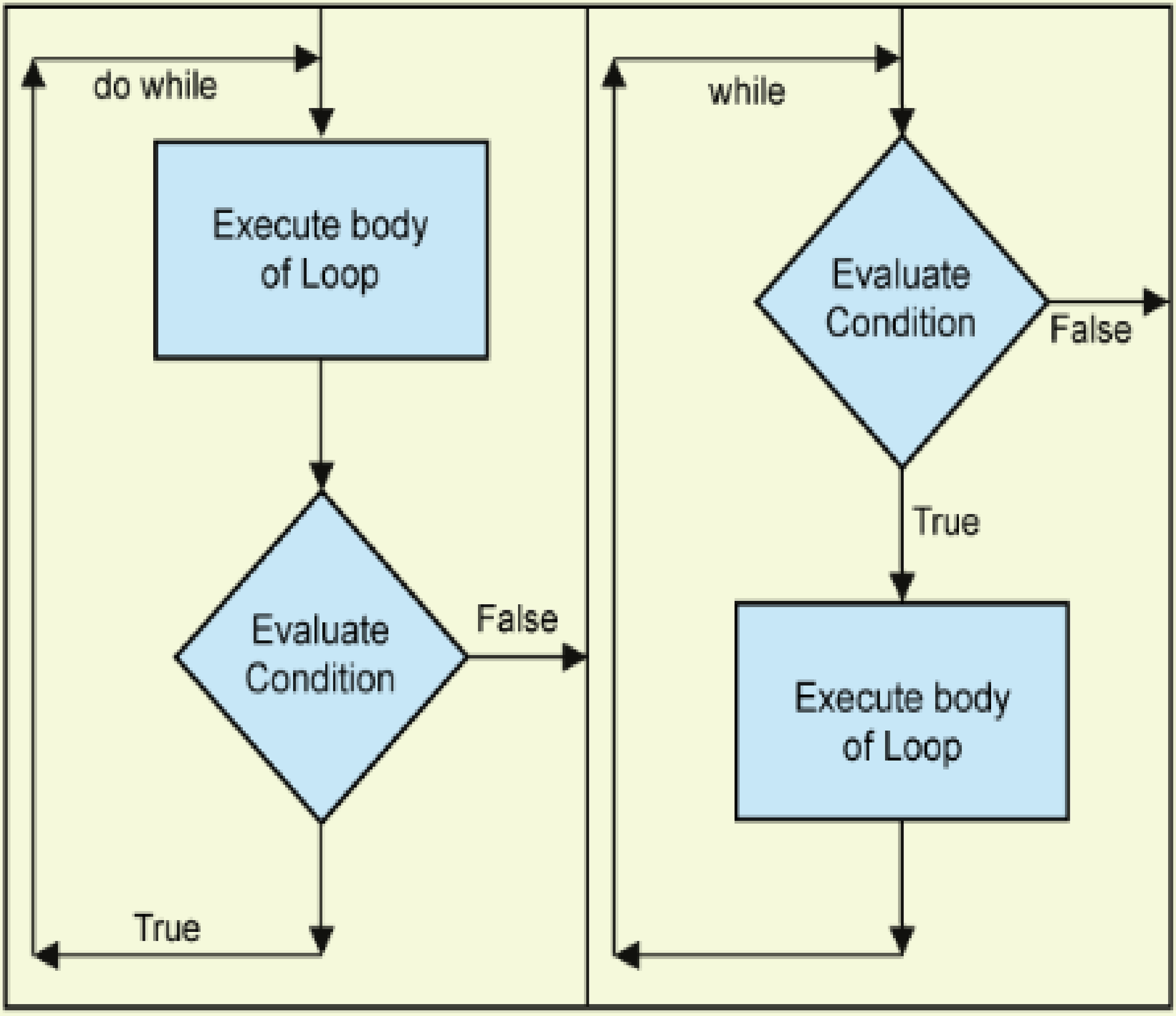
```
do
{
        statements;

}while(expression);
```

The following statements show the syntax for the do...while loop construct:

```
do
{
 statements;
}while(expression);
```

Consider the following example of the do...while loop construct:

```
int var;
var = 100;
do
{
 Console.WriteLine ("Value of var: " + var); /* will print the value 100 */
var = var + 10; // value of var becomes 110

} while (var < 100);

/* 110 < 100 is not true. Hence, program comes outof the loop */
```



Difference in Execution of the do...while and while loops

The for Loop

Usually, the for loop structure is used to execute a block of statements for a specific number of times. However, it can also be used to execute a block of statements indefinitely. The following statements show the syntax of the for loop construct:

The for loop structure is used to execute a block of statements for a specific number of times. The following statements show the syntax of the for construct:

```
for (initialization; termination;
increment/decrement)
{
    statements;
}
```

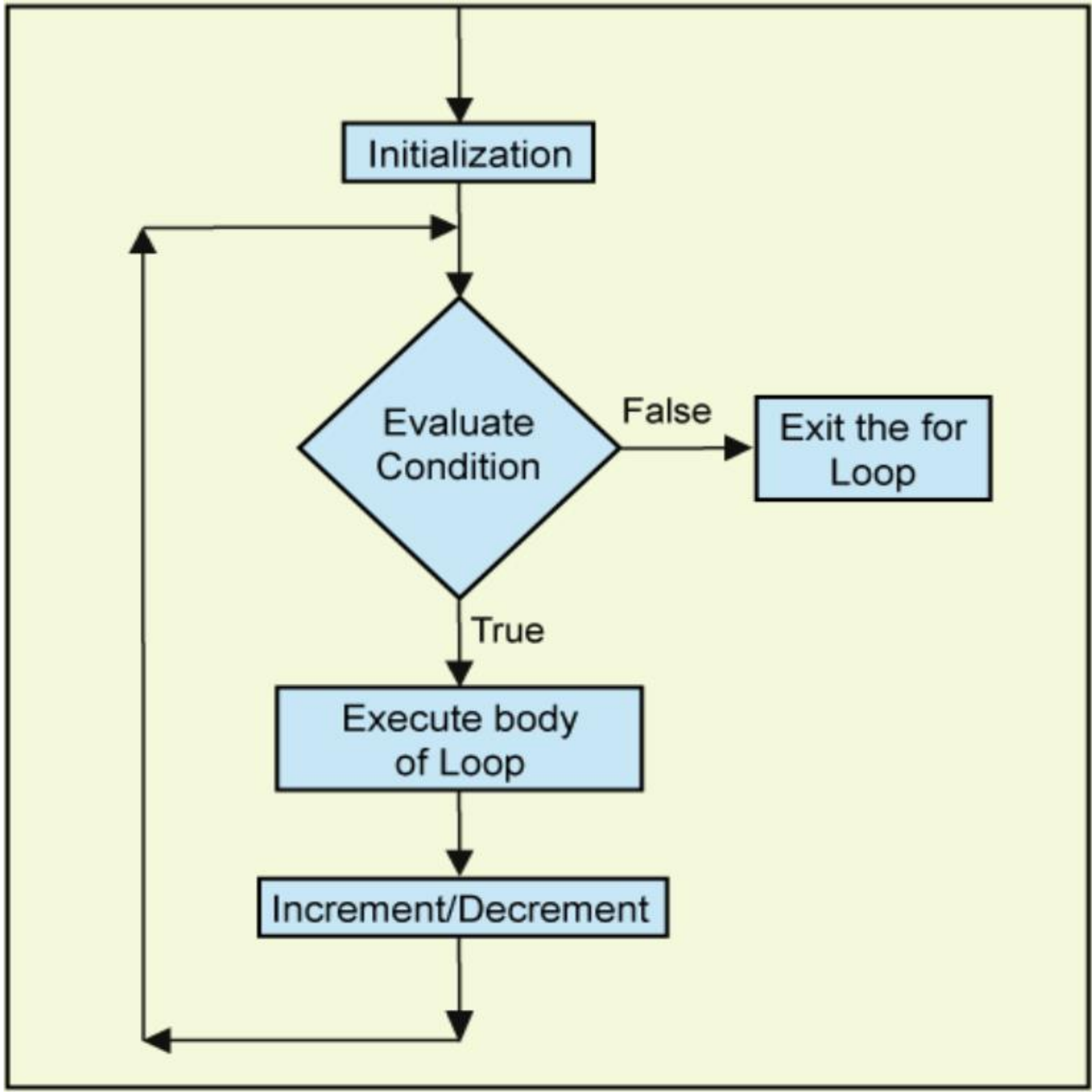
In the for loop, first of all the initialization expression is executed. It is executed only once at the beginning of the loop. Then the termination expression is evaluated for each iteration to determine when to terminate the loop.
When the expression evaluates to false, the loop terminates. Then, through the loop, the statements within the body of the for loop are executed.

Finally, the increment or decrement expression gets invoked for each iteration. All these components are optional.

You can create an infinite loop by keeping all the three expressions blank, as shown in the following code snippet:

```
for ( ; ; )
{
      statements
}
```

The sequence of execution of a for loop construct is shown in the following figure.


Initialization
Evaluate Condition
False
Exit the for Loop
True
Execute body of Loop
Increment/Decrement

Example

```
using System;
class Variable
{
        static void Main(string[] args)
        {
                int var;
                for (var=10; var <20; var++)
                {
                Console.WriteLine ("Value of variable is:{0} ",
var);
                }
        }
}
```

The break and continue Statements

There may be situations where there is a need to exit a loop before the loop condition is checked after an iteration.

The break statement is used to exit from the loop. It prevents the execution of the remaining loop.

The continue statement is used to skip all the subsequent instructions and take the control back to the loop.

The following code accepts five numbers and prints the sum of all the positive numbers:

```
using System;
class BreakContinue
{
	static void Main(string[] args)
	{
	int incr, SumNum, number;
	for(SumNum = number = incr = 0; incr < 5; incr += 1)

	{
	Console.WriteLine("Enter a positive number");
	number = Convert.ToInt32 (Console.ReadLine());
	if (number <= 0) // Non-positive numbers
		continue; // Continue to inctr+=1 in the for loop
	SumNum = SumNum + number;
	}
	Console.WriteLine("The sum of positive numbers
entered is {0}", SumNum);
	}
}
```

Activity: Fibonacci Series Using Loop Constructs

Write a program that generates the Fibonacci series up to 200.

```
using System;
class Fibonacci
{
    static void Main(string[] args)
    {
        int number1;
            int number2;
        number1=number2=1;
        Console.WriteLine("{0}", number1);
        while (number2 < 200)
        {
        Console.WriteLine(number2);
        number2 += number1;
        number1 = number2-number1;
        }
    }
}
```

Exercises

1. Write a program to identify whether a character entered by a user is a vowel or a consonant.
2. Write a program to identify whether the number entered by a user is even or odd.
3. Write a program to accept a number from the user and display all the prime numbers from one up to the number entered by user.
4. Write a program to accept two numbers and check if the first is divisible by the second. In addition, an error message should be displayed if the second number is zero.
5. Write a program to accept two numbers and display the quotient as a result. In addition, an error message should be displayed if the second number is zero or greater than the first number.

6. Write a program to enter a number from 1 to 7 and display the corresponding day of the week.
7. Write a program to display the highest of any 10 numbers entered.
8. Write a program to print the product of the first 10 even numbers.
9. Enter a year and determine whether the year is a leap year or not. A leap year is a non century year that is divisible by 4. A century year is a year divisible by 100, such as 1900. A century year, which is divisible by 400, such as 2000, is also a leap year.

Hint: If a year is divisible by 4 and is not divisible by 100 or divisible by 400, it is a leap year.

Working with Class Members

Working with Attributes and Methods

In a class, you need to store the data related to an object in the form of *attributes*.

You also need to divide the functionalities of a class into logical units called methods. Methods are useful to perform repetitive tasks, such as getting specific records and text.

You can reuse code written in a method because it can be executed any number of times by calling the same method again and again.

Defining the Scope of Attributes and Methods

The scope of a class member refers to the portion of the application from where the member can be read and/or written to. It is also known as the accessibility of a member.

The scope of attributes and methods can be defined by using *access specifiers*. You can use various types of access specifiers to specify the extent of the visibility of an attribute or method.

Types of Access Specifiers

C# supports the following access specifiers:

- public
- private
- protected
- internal
- protected internal

The public Access Specifier

The public access specifier allows a class to share its member variables and member functions with other classes (within or outside the assembly in which the class is defined).

The following code shows the use of the public access specifier:

```
using System;
class Bike
{
public string BikeColor;

/*Since the variable is public, it can be accessed outside
the class definition.*/
}
class Result
{
   static void Main(string[] args)
   {
        Bike Honda = new Bike();
        Honda.BikeColor = "blue";
   }
}
```

In the preceding code, the BikeColor variable is a public data member. Therefore, it can be accessed outside the class.

The private Access Specifier

The private access specifier allows a class to hide its member variables and member functions from other class objects and functions.

Therefore, the private members of a class are not visible outside the class.

The following code shows the usage of the private access specifier:

```
using System;
class Car
{
        
        string Model;
        private void Honk()
        {
        Console.WriteLine("PARRP PARRP!");
        }

        public void SetModel()
        {
        Console.WriteLine("Enter the model name: ");
        Model = Console.ReadLine();
        }

        public void DisplayModel()
        {
        Console.WriteLine("The model is: {0}", Model);
        }
}
```

```
class Display
{
        static void Main(string[] args)
        {
                Car Ford = new Car();

                Ford.SetModel(); //Accepts the model name

                Ford.DisplayModel();
               //Displays the model name

                Ford.Honk();
                 /*error! private members cannot be accessed
                    outside the class definition */

                 Console.WriteLine(Ford.Model);
                / *error! private members cannot be accessed
                   outside the class definition */
        }
}
```

In the preceding code, the DisplayModel() methods, defined in the Car class, can be called from the Main() method because these are public member functions.

However, the Honk() method cannot be accessed through the Ford object because it is a private member function.

When you do not specify any data member as public, protected, or private, then the default access specifier for a data member is private.

In the following example, the data member, Model is private, even though it has not been explicitly specified as private:

```
class Car
{
        char Model;...
}
```

The protected Access Specifier

The protected access specifier allows a class to hide its member variables and member functions from other class objects and functions, except the subclass.

The subclass can be from the same assembly as the protected class or from another assembly.

```
using System;
class Person
{
    // protected variable
     protected string name;

     // protected method
     protected void ShowName()
    {
        Console.WriteLine("Name: " + name);
    }
}
```

```
// Derived class
class Student : Person
{
    public void SetName(string n)
   {
          name = n;        // Accessing protected member
   }
   public void Display()
    {
          ShowName();      // Accessing protected method
    }
}
class Program
{
       static void Main()
       {
              Student s = new Student();
              s.SetName("Akshat Upadhyay");
              s.Display();
       }
}
```

The internal Access Specifier

The internal access specifier allows a class to expose its member variables and member functions to other class functions and objects within the assembly in which the member is defined.

The default access specifier for a class is internal.

The following code depicts the usage of the internal access specifier:

```
using System;
class Bike
{
        internal string BikeColor;
/*Since the variable is internal, it can be accessed
outside the class definition.*/
}
class Result
{
        static void Main(string[] args)
        {
                Bike Honda = new Bike();
                Honda.BikeColor = "blue";
        }
}
```

In the preceding code, Bike and Result are two classes defined within the same assembly. The BikeColor variable is an internal member of the Bike class. Therefore, it can be accessed from outside the class. However, it cannot be accessed from the classes outside the assembly in which the class, Bike is contained.

The protected internal Access Specifier

The protected internal access specifier allows a class to expose its member variables and member functions to the containing class, child classes, or classes within the same assembly. In addition, it allows access to the derived classes outside the assembly.

The protected internal members are different from protected members as they are accessible within their class and sub classes, as well as, within the other classes in the same assembly. However, the protected members are only accessible within their class and sub classes.

They are also different from internal members as internal members are accessible only within the same assembly. They are not accessible even from the sub classes in other assembly.

What is Assemblies

In C#, an assembly is a compiled unit of code that contains one or more types (classes, interfaces, structs, etc.) along with metadata and MSIL (Microsoft Intermediate Language).

It is the basic building block of a .NET application.

i.e An assembly is a file produced by the C# compiler that can be executed or reused by other applications.

Types of Assemblies

1. Executable Assembly Has an entry point (Main() method)
   Extension: .exe
   Example: Console or Windows application

2. Class Library Assembly
   No entry point
   Used by other applications
   Extension: .dll

Contents of an Assembly

An assembly contains:

MSIL code
Metadata (information about types, methods, references)
Manifest (assembly name, version, culture, security info)
Resources (images, strings, etc.)

EXE vs DLL

Feature	EXE (Executable) File	DLL (Dynamic Link Library) File
Purpose	Contains a program that can be run directly	Contains code and resources used by other programs
Entry Point	Has an entry point (Main method)	No entry point; cannot be run directly
Execution	Executable by itself; double-click to run	Cannot be executed directly; called by other programs
Usage	Starts applications	Provides reusable code and resources for applications
File Extension	.exe	.dll

MSIL in .NET

MSIL (Microsoft Intermediate Language) is an intermediate, CPU-independent language generated by .NET compilers (such as C#, VB.NET, F#).

When you write a .NET program, it is not compiled directly to machine code. Instead, it is compiled into MSIL.

How MSIL works

- Source Code (C#, VB.NET, etc.)
- Compiler converts it into MSIL + Metadata
- CLR (Common Language Runtime) loads MSIL
- JIT (Just-In-Time) Compiler converts MSIL into native machine code
- Program executes on the target machine

Features of MSIL

- Platform independent (same MSIL runs on any system with CLR)

- Language independent (generated from any .NET language)

- Contains instructions for:

  - Method calls
  - Exception handling
  - Object creation
  - Security checks

- Stored inside .exe or .dll files