VISUAL PROGRAMMING

By: Sagar Chhabriya (CS-2k22 batch)

SEM-5

SUBMITTED TO

Sir Mohammad Faiz Lakhani

Contents

Visual Programming

Introduction to Visual Programming

Visual programming is a type of programming language or environment that allows users to design and interact with a program visually. Instead of writing code line-by-line, users create programs using graphical elements and actions, often through a drag-and-drop interface. This is particularly useful for non-programmers or those who find traditional text-based coding difficult. Visual programming can be a more intuitive way to design software, especially for applications such as educational software, game development, and system automation.

Event-Driven Programming

Event-driven programming is a paradigm in which the flow of the program is determined by events—such as user actions (e.g., mouse clicks, key presses), sensor outputs, or messages from other programs. It is commonly used in graphical user interfaces (GUIs), interactive applications, and real-time systems. In this paradigm, the program waits for an event to occur and responds to it by executing an event handler or callback function. The events can be generated from various sources, and each event triggers a specific set of instructions.

Different Visual Programming Languages

1. **Scratch**: A popular visual programming language designed for children and beginners to learn programming concepts. It uses blocks to represent code snippets that can be dragged and connected to create programs.
2. **LabVIEW**: A graphical programming language used in system design, particularly in automation and control systems. It uses a block diagram approach to represent algorithms and logic.
3. **Blockly**: A visual language developed by Google that uses blocks to represent code. It can generate code in various text-based programming languages, such as JavaScript, Python, and PHP.
4. **VPL (Visual Programming Languages)**: These languages focus on creating programs visually rather than through text. They are often used in educational contexts or for building applications like workflows or simulations.

IDEs (Integrated Development Environments) Used in Visual Programming

- **Microsoft Visual Studio**: While primarily used for traditional programming languages like C# and VB.NET, Visual Studio also supports visual programming through its drag-and-drop interface for designing forms and GUI elements.
- **Xojo**: A cross-platform IDE that enables the development of desktop and web applications through visual programming.
- **Thimble**: A web-based IDE designed for creating websites through visual programming, making it accessible to beginners.

- **Node-RED**: A flow-based development tool for wiring together hardware devices, APIs, and online services, often used for the Internet of Things (IoT) applications.

.NET Framework

.NET Framework Architecture

The .NET Framework is a software development platform developed by Microsoft for building and running applications primarily on Windows. It provides a large class library and supports several programming languages, including C#, VB.NET, and F#. The framework is built around several core components, making it easier for developers to create applications for web, desktop, mobile, and other environments. The key elements of the .NET Framework architecture include:

1. **Common Language Runtime (CLR)**: The runtime environment that manages the execution of .NET applications, including memory management, garbage collection, and exception handling.
2. **Base Class Library (BCL)**: A comprehensive library of classes, interfaces, and value types that provide commonly used functionality such as file input/output, string manipulation, networking, and database connectivity.
3. **Assemblies**: Compiled code that is used by the CLR. Assemblies can be executables (EXE) or dynamic link libraries (DLL), and they contain metadata that describes the types and methods they expose.
4. **Application Models**: Includes various frameworks such as Windows Forms, ASP.NET, and WPF for building user interfaces and handling application logic.

CLR (Common Language Runtime)

The CLR is the heart of the .NET Framework and provides various essential services for applications, including:

- **Memory Management**: The CLR manages memory allocation and garbage collection. This eliminates the need for developers to manually manage memory, reducing the risk of memory leaks and other related issues.
- **Security**: The CLR provides a level of security by enforcing code access security (CAS), which ensures that code only has access to resources and operations that it is allowed to.
- **Exception Handling**: It standardizes exception handling across all .NET languages, making error handling predictable.
- **Just-in-Time (JIT) Compilation**: The CLR compiles Intermediate Language (IL) code into machine code at runtime, enabling applications to be optimized for the host system's architecture.

Feature	Regular Languages	Visual Programming Languages (VPL)
Definition	A programming language that only uses text.	A programming language that uses graphics or blocks in place of text.
Efficiency	It is not fast and efficient as every block has some code with it, which takes time, and it also includes graphics.	These are quite fast and efficient.
Memory Usage	Requires more memory due to the inclusion of graphics.	Requires less memory as compared to regular languages.
Examples	JavaScript, C, C++, Java, Python, etc.	Mblock, Blockly, Scratch, etc.

Why Choose Visual Programming?

- Visual elements are easier to understand, making development less effortful.
- There's no need for extensive learning to get started.
- The chances of making mistakes are lower because there is no complex syntax.
- It helps you grasp basic programming concepts, making it easier to transition to more advanced languages later.

C# Language Specification

Language Features

C# is a modern, object-oriented, and type-safe programming language developed by Microsoft. It is part of the .NET ecosystem and is primarily used for building Windows applications, web applications, and services. Some of the key features of C# include:

- **Object-Oriented Programming**: C# supports classes, inheritance, polymorphism, and encapsulation, which are the core concepts of object-oriented programming.
- **Garbage Collection**: C# handles memory management automatically, eliminating the need for manual memory allocation and deallocation.
- **Type Safety**: C# ensures type safety, which means that variables are only used in ways that are consistent with their type.
- **Cross-Platform Development**: With the .NET Core framework, C# applications can run on multiple platforms such as Windows, macOS, and Linux.

- **LINQ (Language Integrated Query)**: C# includes LINQ for querying collections in a declarative manner, similar to SQL.
- **Asynchronous Programming**: C# has native support for asynchronous programming with the async and await keywords, making it easier to write non-blocking code.

Language Basic Constructs/Core C#

Namespaces: C# organizes code into namespaces, which help in grouping related classes and other types.

```
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      // Program code
    }
  }
}
```

Classes and Methods: The fundamental building blocks in C# are classes, and methods are used to define behaviour.

```
public class Car
{
  public string Make { get; set; }
  public string Model { get; set; }

  public void Drive()
  {
    Console.WriteLine("Driving the car...");
  }
}
```

Control Flow: C# uses standard control structures such as if, else, switch, for, foreach, while, and do-while for decision-making and loops.

Comments

C# supports single-line and multi-line comments:

1. **Single-line comment**:

   // This is a single-line comment

2. **Multi-line comment**:

   /*

   This is a multi-line comment.

   It can span multiple lines.

   */

XML Documentation Comment (used to document code for API generation):

```
/// <summary>
/// This method adds two integers.
/// </summary>
/// <param name="a">First integer</param>
/// <param name="b">Second integer</param>
/// <returns>Sum of a and b</returns>
public int Add(int a, int b)
{
  return a + b;
}
```

Variables

Variables are used to store data, and each variable must be declared with a specific type. C# is a statically-typed language, meaning the type of a variable is determined at compile time.

- **Declaring variables**:
- int age = 30;
- string name = "John";
- bool isActive = true;

Scalar and Composite Variables

- **Scalar Variables**: These are basic variables that hold a single value, such as integers, floating-point numbers, and booleans.

- Examples of scalar variables:
- int x = 10; // Integer
- double y = 3.14; // Floating-point number
- bool isTrue = true; // Boolean
- char letter = 'A'; // Character

Composite Variables:

- These are complex types that can store multiple values, such as arrays, lists, tuples, and objects.
  - Examples of composite variables:
  - int[] numbers = { 1, 2, 3 }; // Array
  - List<string> names = new List<string> { "John", "Doe" }; // List

---

Nullable Type Variables

In C#, value types (like int, double, bool) cannot be assigned null by default. However, C# allows you to create nullable value types using the ? syntax, which means the variable can hold both a value of its type and a null value.

- **Declaring nullable types**:
- int? nullableInt = null; // Nullable integer
- bool? isCompleted = null; // Nullable boolean

The Nullable<T> struct is used under the hood to support nullable types.

Data Types in C#

C# has a rich set of data types, including value types, reference types, and pointer types.

1. **Value Types**: These store data directly, and each variable holds its own copy of the data. They are typically stored on the stack.
   - **Examples**:
     - int, char, bool, double, float, decimal, struct, enum
2. **Reference Types**: These store a reference (or address) to the actual data. They are typically stored on the heap, and multiple variables can reference the same data.
   - **Examples**:
     - class, interface, delegate, array, string

Value Types and Reference Types

Value Types:

- **Stored in Stack**: Value types store their data directly, and when a variable is assigned to another, a copy of the data is made.
- **Examples**:
  - int (integer)
  - double (floating-point number)
  - char (character)
  - bool (boolean)
  - struct (user-defined structure)
  - enum (enumeration)

```
int a = 5;
int b = a;  // b is a copy of a
b = 10;     // Changing b doesn't affect a
```

Reference Types:

- **Stored in Heap**: Reference types store references (addresses) to the actual data, meaning multiple variables can point to the same data.
- **Examples**:
  - class (reference type object)
  - string (immutable reference type)
  - array (a reference to a collection of elements)
  - delegate (a reference to a method)
  - interface (contract for classes)

```
class Car
{
   public string Model { get; set; }
}

Car car1 = new Car();
car1.Model = "Tesla";

Car car2 = car1;  // car2 now references the same object as car1
car2.Model = "BMW"; // car1.Model will also be "BMW" since both reference the same object
```

Operators in C#

Operators in C# are used to perform operations on variables and values. They are divided into different categories, based on their functionality. Below is an overview of the primary operators, unary operators, shift operators, arithmetic operators, short-circuit logical operators, relational operators, logical (bitwise) operators, and assignment operators in C#.

Primary Operators

Primary operators are the most commonly used operators in C#. They include arithmetic, logical, relational, bitwise, and assignment operators.

Unary Operators

Unary operators operate on a single operand. They include:

Increment (++): Increases the value of a variable by 1.

```
int x = 5;

x++;  // x is now 6
```

Decrement (--): Decreases the value of a variable by 1.

```
int x = 5;

x--;  // x is now 4
```

Negation (-): Reverses the sign of a numeric value.

```
int x = 5;

int y = -x;  // y is now -5
```

Logical NOT (!): Inverts the boolean value of an expression.

```
bool isActive = true;

bool isInactive = !isActive;  // isInactive is false
```

Unary Plus (+): Indicates a positive value (though rarely used in practice, it can be used explicitly).

```
int x = 5;

int y = +x;  // y is 5
```

Shift Operators

Shift operators shift the bits of a variable left or right, used for bit-level manipulation:

Left Shift (<<): Shifts the bits of a variable to the left.

```
int x = 4;  // 0000 0100 in binary
int y = x << 1;  // y becomes 8, 0000 1000 in binary
```

Right Shift (>>): Shifts the bits of a variable to the right.

```
int x = 8;  // 0000 1000 in binary
int y = x >> 1;  // y becomes 4, 0000 0100 in binary
```

Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations:

Addition (+): Adds two operands.

```
int x = 5, y = 3;
int sum = x + y;  // sum is 8
```

Subtraction (-): Subtracts the second operand from the first.

```
int x = 5, y = 3;
int difference = x - y;  // difference is 2
```

Multiplication (*): Multiplies two operands.

```
int x = 5, y = 3;
int product = x * y;  // product is 15
```

Division (/): Divides the numerator by the denominator.

```
int x = 10, y = 2;
int quotient = x / y;  // quotient is 5
```

Modulus (%): Returns the remainder of the division.

```
int x = 10, y = 3;
int remainder = x % y;  // remainder is 1
```

Short Circuit Logical Operators

Short-circuit logical operators are used to perform logical operations in Boolean expressions. They evaluate expressions from left to right and stop if the result is determined early.

Logical AND (&&): Returns true if both operands are true.

```
bool x = true, y = false;
bool result = x && y;  // result is false
```

Logical OR (||): Returns true if at least one operand is true.

```
bool x = true, y = false;
bool result = x || y;  // result is true
```

Logical NOT (!): Inverts the boolean value of a condition.

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

Relational Operators

Relational operators are used to compare two operands and return a boolean result.

Equal to (==): Checks if two operands are equal.

```
int x = 5, y = 5;
bool result = (x == y);  // result is true
```

Not equal to (!=): Checks if two operands are not equal.

```
int x = 5, y = 3;
bool result = (x != y);  // result is true
```

Greater than (>): Checks if the left operand is greater than the right operand.

```
int x = 5, y = 3;
bool result = (x > y);  // result is true
```

Less than (<): Checks if the left operand is less than the right operand.

```
int x = 3, y = 5;
bool result = (x < y);  // result is true
```

Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.

```
int x = 5, y = 5;
bool result = (x >= y);  // result is true
```

Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

```
int x = 3, y = 5;
bool result = (x <= y);  // result is true
```

Logical (Bitwise) Operators

Bitwise operators perform operations on the individual bits of operands.

AND (&): Performs a bitwise AND operation.

```
int x = 5;  // 0101 in binary
int y = 3;  // 0011 in binary
int result = x & y;  // result is 1, 0001 in binary
```

OR (|): Performs a bitwise OR operation.

```
int x = 5;  // 0101 in binary
int y = 3;  // 0011 in binary
int result = x | y;  // result is 7, 0111 in binary
```

XOR (^): Performs a bitwise XOR operation.

```
int x = 5;  // 0101 in binary
int y = 3;  // 0011 in binary
int result = x ^ y;  // result is 6, 0110 in binary
```

Complement (~): Inverts all the bits of the operand.

```
int x = 5;  // 0101 in binary
int result = ~x;  // result is -6, 1010 in binary (two's complement)
```

Left Shift (<<): Shifts bits to the left.

```
int x = 5;  // 0101 in binary
int result = x << 1;  // result is 10, 1010 in binary
```

Right Shift (>>): Shifts bits to the right.

```
int x = 8;  // 1000 in binary
int result = x >> 1;  // result is 4, 0100 in binary
```

Assignment Operators

Assignment operators are used to assign values to variables.

Simple Assignment (=): Assigns the value of the right operand to the left operand.

```
int x = 5;
```

Add and Assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.

```
int x = 5;

x += 3;  // x is now 8
```

Subtract and Assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

```
int x = 5;

x -= 3;  // x is now 2
```

Multiply and Assign (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

```
int x = 5;

x *= 3;  // x is now 15
```

Divide and Assign (` /=`): Divides the left operand by the right operand and assigns the result to the left operand.

```
int x = 6;

x /= 3;  // x is now 2
```

Modulus and Assign (%=): Calculates the modulus of the left operand by the right operand and assigns the result to the left operand.

```
int x = 7;

x %= 3;  // x is now 1
```

Control Statements in C#

Control statements in C# allow you to dictate the flow of execution in your program. They are used to perform different actions based on conditions or to repeat blocks of code. C# provides several control statements, including conditional statements (like if and switch) and iteration statements (like for, while, do-while, and foreach).

If and Switch Statements

1. **If Statement** The if statement is used to evaluate a condition. If the condition is true, the code inside the block is executed. If the condition is false, the code inside the block is skipped.

   Syntax:

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

   Example:

   ```
   int age = 18;
   if (age >= 18)
   {
     Console.WriteLine("You are an adult.");
   }
   ```

2. **If-Else Statement** An if-else statement is used when there are two possible outcomes: one when the condition is true and another when the condition is false.

   Syntax:

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

   Example:

   ```
   int age = 16;
   if (age >= 18)
   {
     Console.WriteLine("You are an adult.");
   }
   else
   {
     Console.WriteLine("You are a minor.");
   }
   ```

3. **Else If Statement** The else if statement allows you to evaluate multiple conditions. If the first condition is false, the program checks the next condition in line.

   Syntax:

```
if (condition1)
{
  // Code for condition1
}
else if (condition2)
{
  // Code for condition2
}
else
{
  // Code if none of the conditions are true
}
```

   Example:

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

4. **Switch Statement** The switch statement is used when you have multiple conditions based on the value of a single variable. It is generally more efficient and easier to read than using multiple if-else statements when comparing the same variable to different values.

   Syntax:

```
switch (variable)
{
  case value1:
    // Code for value1
    break;
  case value2:
    // Code for value2
    break;
  default:
```

```
        // Code if no match
        break;
    }
```

Example:

```
int day = 3;
switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  default:
    Console.WriteLine("Invalid day");
    break;
}
```

Iteration Statements

Iteration statements are used to repeat a block of code multiple times, depending on a condition. The primary iteration statements in C# are the for, while, do-while, and foreach loops.

1. **For Loop** A for loop is used when the number of iterations is known beforehand. It has three parts: initialization, condition, and increment/decrement.

   Syntax:

   ```
   for (initialization; condition; increment/decrement)
   {
     // Code to execute during each iteration
   }
   ```

   Example:

   ```
   for (int i = 0; i < 5; i++)
   {
     Console.WriteLine(i);  // Outputs 0 to 4
   }
   ```

2. **While Loop** A while loop is used when the number of iterations is not known in advance, and the loop continues as long as the specified condition is true.

   Syntax:

   ```
   while (condition)
   {
   ```

```
    // Code to execute during each iteration
}
```

Example:

```
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);  // Outputs 0 to 4
    i++;
}
```

3. **Do-While Loop** A do-while loop is similar to the while loop, but it ensures that the loop body is executed at least once, even if the condition is false initially. The condition is checked after the loop body is executed.

Syntax:

```
do
{
    // Code to execute during each iteration
} while (condition);
```

Example:

```
int i = 0;
do
{
    Console.WriteLine(i);  // Outputs 0 to 4
    i++;
} while (i < 5);
```

4. **For Each Loop** The foreach loop is used to iterate over collections (such as arrays or lists). It automatically iterates through each element of the collection without the need for an index variable.

Syntax:

```
foreach (var item in collection)
{
    // Code to execute for each item in the collection
}
```

Example:

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
foreach (string day in days)
{
    Console.WriteLine(day);  // Outputs the names of all the days
}
```

Branch or Jump Statements in C#

Branch or jump statements are used to alter the flow of control in a program. They allow you to exit from loops, skip iterations, jump to specific parts of code, or return from methods. The key branch or jump statements in C# are break, continue, goto, and return.

Break Statement

The break statement is used to exit a loop or a switch statement prematurely. When a break statement is encountered inside a loop or switch, the control is transferred to the statement immediately following the loop or switch.

Syntax:

- break;

Example:

```
for (int i = 0; i < 10; i++)
{
   if (i == 5)
   {
      break;  // Exit the loop when i is 5
   }
   Console.WriteLine(i);
}
```

Continue Statement

The continue statement is used to skip the current iteration of a loop and move to the next iteration. It can be used in for, while, and do-while loops.

Syntax:

- continue;

Example:

```
for (int i = 0; i < 10; i++)
{
   if (i % 2 == 0)
   {
      continue;  // Skip even numbers
   }
   Console.WriteLine(i);  // Outputs 1, 3, 5, 7, 9
}
```

Goto Statement

The goto statement transfers control to a specified label within the code. It is generally discouraged in most programming because it can make the code harder to read and maintain. However, it may be useful in some situations like error handling.

Syntax:

- goto label;
- label: // Declaration of the label

Example:

```
int i = 0;
startLoop:
if (i < 5)
{
   Console.WriteLine(i);
   i++;
   goto startLoop;  // Go back to the start of the loop
}
```

Return Statement

The return statement is used to exit from a method and optionally return a value. If the method is of a void type, it simply exits the method without returning a value.

Syntax:

- return; // For void methods
- return value; // For methods with a return type

Example:

```
int Add(int x, int y)
{
   return x + y;  // Return the sum of x and y
}

void PrintMessage()
{
   Console.WriteLine("Hello, World!");
   return;  // Exit the method early
}
```

Constants in C#

Constants are values that are defined at compile-time and cannot be changed during the execution of the program. They are declared using the const keyword.

Const Keyword

The const keyword is used to declare a constant field or local variable. Once a value is assigned to a const, it cannot be modified during the program's execution.

- **Syntax**:

  const dataType constantName = constantValue;

- **Example**:

  const double Pi = 3.14159; // Pi is a constant and cannot be changed

---

Read-Only Members

In addition to constants, C# also supports readonly members. A readonly field can be assigned only once, either at the time of declaration or within a constructor. Unlike constants, readonly fields can be set at runtime, but they cannot be modified after the constructor finishes execution.

Syntax:

- readonly dataType fieldName;

Example:

```
class Circle
{
  public readonly double radius;

  public Circle(double radius)
  {
    this.radius = radius;  // Assign the value in the constructor
  }
}
```

Methods in C#

Methods are blocks of code that perform a specific task. In C#, methods can accept parameters, return values, and be called to execute their functionality. Methods are defined using the void keyword (if they don't return a value) or a specific return type.

Method Parameters vs. Arguments

- **Method Parameters** are variables defined in the method declaration that represent the values or references that will be passed into the method.
- **Arguments** are the actual values or references passed to the method when calling it.

Example:

```
void Greet(string name)  // 'name' is the parameter
{
   Console.WriteLine("Hello, " + name);
}

Greet("John");  // "John" is the argument
```

Passing by Reference and by Value in C#

C# supports both **pass by value** and **pass by reference** when passing parameters to methods.

1. **Pass by Value**: A copy of the variable is passed to the method. Changes made to the parameter inside the method do not affect the original variable.

   Syntax:

   ```
   void ModifyValue(int num)
   {
      num = 10;  // Only modifies the local copy of num
   }
   ```

   Example:

   ```
   int number = 5;
   ModifyValue(number);
   Console.WriteLine(number);  // Outputs: 5
   ```

2. **Pass by Reference**: The reference (memory address) of the variable is passed to the method. Changes made to the parameter inside the method affect the original variable.

   Syntax:

   ```
   void ModifyReference(ref int num)
   ```

```
{
    num = 10;  // Modifies the original value of num
}
```

Example:

```
int number = 5;
ModifyReference(ref number);
Console.WriteLine(number);  // Outputs: 10
```

Note: You must use the ref keyword for passing by reference, both in the method definition and when calling the method.

3. **Out Parameters**: An alternative to ref, out parameters are used to return multiple values from a method. The variable passed as an out parameter doesn't need to be initialized before calling the method.

   Syntax:

```
void CalculateArea(out int area)
{
    area = 100;  // area must be assigned before use
}
```

   Example:

```
int result;
CalculateArea(out result);
Console.WriteLine(result);  // Outputs: 100
```

Named Arguments

Named arguments allow you to specify the arguments by their parameter names rather than relying on the order in which they appear in the method declaration. This can make the method calls more readable and allow you to pass arguments in any order.

Syntax:

```
void DisplayInfo(string name, int age)
{
    Console.WriteLine("Name: " + name + ", Age: " + age);
}

DisplayInfo(age: 25, name: "Alice");  // Arguments can be passed in any order
```

Optional Arguments

Optional arguments allow you to specify default values for method parameters. If the caller does not provide an argument for the optional parameter, the default value is used.

Syntax:

```
void DisplayInfo(string name, int age = 30)  // 'age' has a default value
{
   Console.WriteLine("Name: " + name + ", Age: " + age);
}
```

Example:

```
DisplayInfo("John");  // Outputs: Name: John, Age: 30
DisplayInfo("Alice", 25);  // Outputs: Name: Alice, Age: 25
```

Params Keyword

The params keyword allows you to pass a variable number of arguments to a method. It is used when you want to pass an array of arguments to a method without explicitly creating an array.

Syntax:

```
void PrintNumbers(params int[] numbers)
{
   foreach (var num in numbers)
   {
      Console.WriteLine(num);
   }
}
```

Example:

```
PrintNumbers(1, 2, 3, 4, 5);  // Outputs: 1 2 3 4 5
```

Note: You can only have one params parameter, and it must be the last parameter in the method signature.

Extension Methods

Extension methods allow you to add new functionality to existing types without modifying their original code. You define extension methods as static methods in a static class, and the first parameter of the method specifies the type to extend.

Syntax:

```
public static class StringExtensions
{
   public static bool IsPalindrome(this string str)
   {
      char[] arr = str.ToCharArray();
      Array.Reverse(arr);
      return new string(arr) == str;
   }
}
```

Example:

```
string word = "racecar";
bool isPalindrome = word.IsPalindrome();  // Returns true
```

Note: The this keyword in the first parameter specifies the type that the method extends.

Properties in C#

Properties in C# are members of a class or struct that provide a mechanism to read, write, or compute the values of private fields. They can be thought of as a combination of a method and a field.

Auto-Implemented Properties

Auto-implemented properties simplify property declarations by automatically generating a private, anonymous field to store the property value. When you declare an auto-implemented property, you don't need to explicitly define a private backing field.

Syntax:

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

Example:

```
class Person
{
   public string Name { get; set; } // Auto-implemented property
   public int Age { get; set; }
}

var person = new Person();
person.Name = "John";
person.Age = 30;
```

Auto-implemented properties are useful when you don’t need additional logic in the getter and setter.

Properties Overriding

In C#, you can override properties in derived classes just like methods. You need to use the override keyword and ensure the property in the base class is marked as virtual, abstract, or override.

Syntax:

```
public virtual int MyProperty { get; set; }
```

Example:

```
class Animal
{
   public virtual string Sound { get; set; } // Base class property
}
```

```
class Dog : Animal
{
    public override string Sound { get; set; }  // Overriding the property in derived class
}

var dog = new Dog();
dog.Sound = "Bark";
Console.WriteLine(dog.Sound);  // Outputs: Bark
```

Object-Oriented Programming (OOP) Aspects in C#

C# supports all four core principles of Object-Oriented Programming: **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. Let's explore key OOP concepts such as classes, namespaces, and various types of classes in C#.

Namespaces

Namespaces are used in C# to organize classes, structs, interfaces, and other types into groups. This helps prevent naming conflicts and makes the code easier to maintain.

Syntax:

```
namespace MyNamespace
{
    class MyClass
    {
        // Class definition
    }
}
```

Example:

```
using MyNamespace;

namespace AnotherNamespace
{
```

```
    class Program
    {
        static void Main()
        {
            MyClass myClass = new MyClass();
            Console.WriteLine("Namespace Example");
        }
    }
}
```

Class

A **class** in C# defines the blueprint for objects. It can contain fields, properties, methods, and events. Objects are instances of classes.

Syntax:

```
public class MyClass
{
    public int MyProperty { get; set; }

    public void MyMethod()
    {
        Console.WriteLine("Method in MyClass");
    }
}
```

Example:

```
MyClass myObject = new MyClass();
myObject.MyMethod();  // Outputs: Method in MyClass
```

Partial Class

A **partial class** allows the definition of a class to be split into multiple files. This is particularly useful when working with large classes or when generating code in tools like Visual Studio.

Syntax:

```
    public partial class MyClass
    {
```

```
    public void MethodOne()
    {
        Console.WriteLine("Method One");
    }
}

public partial class MyClass
{
    public void MethodTwo()
    {
        Console.WriteLine("Method Two");
    }
}
```

Example:

```
MyClass obj = new MyClass();
obj.MethodOne();  // Outputs: Method One
obj.MethodTwo();  // Outputs: Method Two
```

Static Class

A **static class** is a class that cannot be instantiated, and all its members must be static. It is typically used to group utility methods that don't need an instance of the class.

Syntax:

```
public static class MathHelper
{
    public static int Add(int x, int y)
    {
        return x + y;
    }
}
```

Example:

```
int sum = MathHelper.Add(5, 10);  // Outputs: 15
```

Note: You cannot create an instance of a static class.

Sealed Class

A **sealed class** is a class that cannot be inherited. This is useful when you want to prevent other classes from deriving from your class.

Syntax:

```
public sealed class MySealedClass
{
  public void MyMethod()
  {
    Console.WriteLine("Method in Sealed Class");
  }
}
```

Example:

```
MySealedClass obj = new MySealedClass();

obj.MyMethod();  // Outputs: Method in Sealed Class
```

Note: If a class is marked as sealed, it cannot be inherited.

Abstract Class

An **abstract class** is a class that cannot be instantiated directly. It is used to define common functionality for derived classes, and it can contain abstract methods that must be implemented by the derived classes.

Syntax:

```
public abstract class Animal
{
  public abstract void MakeSound();  // Abstract method
}

public class Dog : Animal
{
  public override void MakeSound()
  {
    Console.WriteLine("Bark");
  }
}
```

Example:

```
Dog dog = new Dog();

dog.MakeSound();  // Outputs: Bark
```

Note: An abstract class can contain both abstract methods (without implementation) and regular methods (with implementation).

Interface in C#

An **interface** in C# is a contract that defines a set of methods and properties that a class must implement. It does not contain any implementation, only the signatures of the methods or properties. A class that implements an interface must provide the implementation for all the methods and properties declared in the interface.

Syntax:

```
public interface IAnimal
{
  void MakeSound();
  void Eat();
}
```

Example:

```
public class Dog : IAnimal
{
  public void MakeSound()
  {
    Console.WriteLine("Bark");
  }

  public void Eat()
  {
    Console.WriteLine("Eating food");
  }
}

var dog = new Dog();
dog.MakeSound();  // Outputs: Bark
dog.Eat();  // Outputs: Eating food
```

Note: A class can implement multiple interfaces.

Inheritance in C#

Inheritance is one of the core principles of Object-Oriented Programming (OOP). It allows a class to inherit members (fields, properties, methods) from another class. The class that is inherited from is

called the **base class** or **parent class**, and the class that inherits is called the **derived class** or **child class**.

Syntax:

```
public class Animal
{
  public void Eat() { Console.WriteLine("Eating..."); }
}

public class Dog : Animal  // Dog inherits from Animal
{
  public void Bark() { Console.WriteLine("Barking..."); }
}
```

Example:

```
Dog dog = new Dog();
dog.Eat();  // Inherited from Animal class
dog.Bark();  // From Dog class
```

Note: The derived class inherits all public and protected members of the base class.

Method Overloading in C#

Method overloading occurs when multiple methods in the same class have the same name but different parameter types, number of parameters, or both.

Syntax:

```
public void PrintMessage(string message) { Console.WriteLine(message); }
public void PrintMessage(int number) { Console.WriteLine(number); }
```

Example:

```
class Printer
{
  public void PrintMessage(string message) { Console.WriteLine(message); }
  public void PrintMessage(int number) { Console.WriteLine(number); }
}

Printer printer = new Printer();
printer.PrintMessage("Hello, world!");  // Outputs: Hello, world!
printer.PrintMessage(123);  // Outputs: 123
```

Note: The method signature must differ in the type and/or number of parameters.

Method Overriding in C#

Method overriding allows a derived class to provide a specific implementation for a method that is already defined in the base class. This is possible when the method in the base class is marked as virtual and the derived class uses the override keyword.

Syntax:

```
public virtual void MakeSound() { Console.WriteLine("Animal sound"); }
```

Example:

```
class Animal
{
  public virtual void MakeSound() { Console.WriteLine("Animal sound"); }
}

class Dog : Animal
{
  public override void MakeSound() { Console.WriteLine("Bark"); }
}

Dog dog = new Dog();
dog.MakeSound();  // Outputs: Bark
```

Note: The method in the derived class must have the same signature as the method in the base class.

Method Hiding in C#

Method hiding occurs when a method in the derived class has the same name as a method in the base class, but does not use the override keyword. Instead, the new keyword is used to hide the base class method. Method hiding does not follow polymorphism, and the method in the base class is hidden in the derived class.

Syntax:

```
public new void MakeSound() { Console.WriteLine("Dog sound"); }
```

Example:

```
class Animal
{
  public void MakeSound() { Console.WriteLine("Animal sound"); }
}
```

```
class Dog : Animal
{
   public new void MakeSound() { Console.WriteLine("Dog sound"); }
}
Animal animal = new Dog();
animal.MakeSound();  // Outputs: Animal sound (not Dog sound)
```

Note: When you hide a method, it does not participate in polymorphism, and the base method is still called unless the reference is of the derived type.

Constructors, Base Keyword, and This Keyword

Constructors are special methods used to initialize objects. The base keyword refers to the base class constructor, while the this keyword refers to the current instance of the class.

Constructors

A constructor is used to initialize objects when they are created. It has the same name as the class and no return type.

Syntax:

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

   public Person(string name)
   {
      Name = name;  // Constructor initializing property
   }
}

var person = new Person("Alice");
```

Base Keyword

The base keyword is used to call a constructor or method of the base class. It is also used to refer to base class members that are hidden or overridden.

Syntax:

```
public class Animal
{
   public Animal(string name) { Console.WriteLine(name); }
}

public class Dog : Animal
```

```
{
    public Dog(string name) : base(name)  // Call base class constructor
    {
        Console.WriteLine("Dog constructor");
    }
}

var dog = new Dog("Buddy");  // Outputs: Buddy
                             // Outputs: Dog constructor
```

This Keyword

The this keyword refers to the current instance of the class. It is commonly used to access instance members and differentiate between method parameters and class fields when they have the same name.

Syntax:

```
public class Person
{
    private string name;

    public Person(string name)
    {
        this.name = name;  // 'this' distinguishes the field from the parameter
    }
}
```

Access Modifiers in C#

Access modifiers determine the visibility and accessibility of class members (fields, methods, properties) from other classes. C# supports several access modifiers: **Public**, **Private**, **Protected**, and **Internal**.

Public Access Modifier

The public modifier makes the member accessible from any other class.

Syntax:

```
public int MyProperty;
```

Example:

```
public class MyClass
{
```

```
    public int MyProperty;
}
```

Private Access Modifier

The private modifier restricts the member to be accessible only within the class where it is defined.

Syntax:

```
private int MyProperty;
```

Example:

```
public class MyClass
{
  private int MyProperty;
}
```

Protected Access Modifier

The protected modifier allows the member to be accessible within the class where it is defined and in derived classes.

Syntax:

```
protected int MyProperty;
```

Example:

```
public class Animal
{
  protected int Age;
}

public class Dog : Animal
{
  public void DisplayAge() { Console.WriteLine(Age); }
}
```

Internal Access Modifier

The internal modifier makes the member accessible only within the same assembly (project) but not outside it.

Syntax:

```
    internal int MyProperty;
```

Example:

```
public class MyClass
{
  internal int MyProperty;
}
```

Enumerations in C#

An **enumeration (enum)** is a distinct type in C# that defines a set of named constants. It is used to assign meaningful names to numeric values, improving code readability and maintainability.

Syntax:

```
enum Days
{
  Sunday,    // Default value is 0
  Monday,    // Default value is 1
  Tuesday,   // Default value is 2
  Wednesday, // Default value is 3
  Thursday,  // Default value is 4
  Friday,    // Default value is 5
  Saturday   // Default value is 6
}
```

Example:

```
using System;

enum Days
{
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday
}

class Program
{
  static void Main(string[] args)
  {
    Days today = Days.Monday;
```

```
        Console.WriteLine(today);  // Outputs: Monday
        Console.WriteLine((int)today);  // Outputs: 1 (Default value assigned to Monday)
    }
}
```

Custom Values in Enums: You can assign specific values to enum members, instead of letting them take the default values starting from 0.

```
enum Days
{
    Sunday = 1,
    Monday = 2,
    Tuesday = 3,
    Wednesday = 4,
    Thursday = 5,
    Friday = 6,
    Saturday = 7
}
```

Example:

```
Days today = Days.Friday;
Console.WriteLine((int)today);  // Outputs: 6
```

Flags Enum: You can use an enum to represent a set of flags (bitwise operations).

```
[Flags]
enum Permissions
{
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4
}

class Program
{
    static void Main(string[] args)
    {
        Permissions userPermissions = Permissions.Read | Permissions.Write;
        Console.WriteLine(userPermissions);  // Outputs: Read, Write
    }
}
```

Structures in C#

A **structure (struct)** is a value type that is used to define a data structure. Unlike classes, structures are typically used to encapsulate small groups of related data. Structures can have fields, properties, methods, and constructors, but they cannot inherit from other types or be inherited.

Syntax:

```
struct Point
{
   public int X;
   public int Y;

   public Point(int x, int y)  // Constructor
   {
      X = x;
      Y = y;
   }

   public void Display()
   {
      Console.WriteLine($"X: {X}, Y: {Y}");
   }
}
```

Example:

```
using System;

struct Point
{
   public int X;
   public int Y;

   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }

   public void Display()
   {
      Console.WriteLine($"X: {X}, Y: {Y}");
   }
}

class Program
```

```
{
    static void Main(string[] args)
    {
        Point p = new Point(10, 20);
        p.Display();  // Outputs: X: 10, Y: 20
    }
}
```

Key Points:

- Structures are value types, meaning that they are stored on the stack, and copying a structure creates a new copy of the data.
- A structure cannot have a default constructor (a parameterless constructor), but it can have other constructors.
- Structures do not support inheritance or polymorphism, but they can implement interfaces.

Partial Structures in C#

A **partial structure** allows a structure to be split across multiple files. This is useful for large structures or when auto-generated code must be split from user code. Each part of the structure must use the partial keyword.

Syntax:

```
// Part 1 of the structure (in file 1)
public partial struct Person
{
    public string Name;
}

// Part 2 of the structure (in file 2)
public partial struct Person
{
    public int Age;
}
```

Example:

File 1 (PersonPart1.cs):

```
public partial struct Person
{
   public string Name;
}
```

File 2 (PersonPart2.cs):

```
public partial struct Person
{
   public int Age;
}
```

Main Program (Program.cs):

```
using System;

class Program
{
   static void Main(string[] args)
   {
      Person person = new Person();
      person.Name = "Alice";
      person.Age = 30;
      Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
   }
}
```

Output:

- Name: Alice, Age: 30

- **Note**: Partial structures work similarly to partial classes. Each part of the structure can be in a separate file, but they must all be part of the same namespace and assembly. The partial keyword indicates that the structure is split across multiple parts.

Strings and Characters in C#

In C#, **strings** are a sequence of characters and are widely used for storing and manipulating text. Strings are immutable, meaning that once a string is created, it cannot be modified. C# provides various methods and properties to work with strings and characters efficiently.

Strings as Immutable Objects

In C#, **strings** are immutable, meaning their values cannot be changed once created. If you modify a string, a new string object is created with the modified value, while the original string remains unchanged.

Example:

```
string str = "Hello";

str = str + " World";  // A new string is created, the original "Hello" remains unchanged

Console.WriteLine(str);  // Outputs: Hello World
```

Verbatim Strings

A **verbatim string** is prefixed with the @ symbol and allows you to include special characters like newlines and backslashes without needing to escape them. It is particularly useful for file paths or multi-line strings.

Syntax:

- `string path = @"C:\Users\Documents\File.txt";  // No need to escape backslashes`
- `string multiline = @"This is a`

  `multi-line`

  `string.";  // Can span multiple lines`

Example:

```
string path = @"C:\Users\JohnDoe\Documents\Report.txt";

Console.WriteLine(path);  // Outputs: C:\Users\JohnDoe\Documents\Report.txt
```

Format Strings

Format strings allow you to format the output of a string by inserting values into placeholders. This is done using the string.Format() method or interpolated strings.

Syntax:

```
string formatted = string.Format("Hello, {0}!", "Alice");

Console.WriteLine(formatted);  // Outputs: Hello, Alice!
```

Example:

```
int age = 25;
string formatted = string.Format("I am {0} years old.", age);
Console.WriteLine(formatted);  // Outputs: I am 25 years old.
```

Alternatively, you can use **string interpolation** (C# 6 and later):

```
string formatted = $"I am {age} years old.";
Console.WriteLine(formatted);  // Outputs: I am 25 years old.
```

Substrings

You can extract a **substring** from a string using the Substring() method. The method takes two arguments: the starting index and the length of the substring.

Syntax:

```
string sub = str.Substring(startIndex, length);
```

Example:

```
string str = "Hello, World!";
string sub = str.Substring(7, 5);  // Extracts "World"
Console.WriteLine(sub);  // Outputs: World
```

Accessing Individual Characters of Strings

You can access individual characters of a string using an index, as strings are indexed from 0.

Syntax:

```
char character = str[index];
```

Example:

```
string str = "Hello";
char ch = str[1];  // Accesses 'e' (second character)
Console.WriteLine(ch);  // Outputs: e
```

Replacing Substrings

You can replace a substring in a string using the Replace() method. It replaces all occurrences of a specified substring with another.

Syntax:

```
string result = str.Replace(oldValue, newValue);
```

Example:

```
string str = "Hello, World!";
string newStr = str.Replace("World", "C#");
Console.WriteLine(newStr);  // Outputs: Hello, C#!
```

Removing Substrings

The Remove() method removes a specified number of characters starting from a given index.

Syntax:

```
string result = str.Remove(startIndex, length);
```

Example:

```
string str = "Hello, World!";
string newStr = str.Remove(5, 7);  // Removes ", World"
Console.WriteLine(newStr);  // Outputs: Hello
```

Removing Trailing and Leading White Spaces

The Trim(), TrimStart(), and TrimEnd() methods remove white spaces from a string. Trim() removes spaces from both ends, while TrimStart() and TrimEnd() remove spaces only from the start or end, respectively.

Syntax:

- `string result = str.Trim();`
- `string resultStart = str.TrimStart();`
- `string resultEnd = str.TrimEnd();`

Example:

- `string str = "  Hello, World!  ";`
- `string trimmed = str.Trim();  // Removes leading and trailing spaces`
- `Console.WriteLine(trimmed);  // Outputs: Hello, World!`

Finding Index of Substrings and Characters

The IndexOf() method finds the index of the first occurrence of a substring or character.

Syntax:

```
int index = str.IndexOf(value);
```

Example:

```
string str = "Hello, World!";
int index = str.IndexOf("World");  // Finds the index of "World"
Console.WriteLine(index);  // Outputs: 7
```

Upper Case and Lower Case Strings

You can convert a string to upper case or lower case using the ToUpper() and ToLower() methods.

Syntax:

```
string upper = str.ToUpper();
string lower = str.ToLower();
```

Example:

```
string str = "Hello, World!";
string upperStr = str.ToUpper();  // Converts to "HELLO, WORLD!"
string lowerStr = str.ToLower();  // Converts to "hello, world!"
Console.WriteLine(upperStr);  // Outputs: HELLO, WORLD!
Console.WriteLine(lowerStr);  // Outputs: hello, world!
```

Copying Strings

You can create a copy of a string using the Clone() method or by simply assigning the string to another variable, as strings are immutable.

Syntax:

```
string copy = str.Clone();
```

Example:

```
string str = "Hello, World!";
string copy = (string)str.Clone();
Console.WriteLine(copy);  // Outputs: Hello, World!
```

Comparing Strings

The Compare() and CompareTo() methods compare two strings lexicographically. Compare() returns an integer indicating the comparison result, while CompareTo() compares the current string to another.

Syntax:

```
int result = string.Compare(str1, str2);

int result = str1.CompareTo(str2);
```

Example:

```
string str1 = "apple";

string str2 = "banana";

int result = string.Compare(str1, str2);

Console.WriteLine(result);  // Outputs a negative value because "apple" is less than "banana"
```

Concatenating Strings

You can concatenate strings using the + operator or the Concat() method.

Syntax:

```
string result = str1 + str2;

string result = string.Concat(str1, str2);
```

Example:

```
string str1 = "Hello";

string str2 = "World!";

string concatenated = str1 + " " + str2;

Console.WriteLine(concatenated);  // Outputs: Hello World!
```

Inserting Strings

The Insert() method inserts a substring at a specified position in the original string.

Syntax:

```
string result = str.Insert(index, value);
```

Example:

- string str = "Hello, World!";
- string newStr = str.Insert(7, "C# "); // Inserts "C# " at index 7
- Console.WriteLine(newStr); // Outputs: Hello, C# World!

Splitting Strings

The Split() method splits a string into an array of substrings based on a delimiter.

Syntax:

```
string[] parts = str.Split(delimiter);
```

Example:

```
string str = "apple,banana,orange";
string[] fruits = str.Split(',');
foreach (string fruit in fruits)
{
   Console.WriteLine(fruit);
}
// Outputs:
// apple
// banana
// orange
```

Joining Strings

The Join() method combines an array of strings into a single string with a specified separator.

Syntax:

```
string result = string.Join(separator, array);
```

Example:

```
string[] words = { "apple", "banana", "orange" };
string result = string.Join(", ", words);
Console.WriteLine(result);  // Outputs: apple, banana, orange
```

Collections in C#

In C#, collections are data structures that hold groups of objects or values. One of the most commonly used collection types is **Arrays**, which allow you to store multiple elements in a single variable.

Arrays in C#

An **array** is a collection of elements of the same type that are stored in a contiguous block of memory. Arrays are zero-indexed, meaning the first element is at index 0. Arrays are considered objects in C#, and they have methods and properties like any other object.

Arrays as Objects

In C#, arrays are objects. This means they are instances of the Array class, and they have methods and properties such as Length, which returns the number of elements in the array.

Example:

```
int[] numbers = new int[5];  // Array of 5 integers

Console.WriteLine(numbers.Length);  // Outputs: 5
```

Single Dimensional Arrays

A **single-dimensional array** is a simple list of elements. It is the most basic form of an array, where each element can be accessed using a single index.

Syntax:

```
int[] array = new int[5];  // Declaring an array of integers with 5 elements
```

Example:

```
int[] numbers = { 1, 2, 3, 4, 5 };  // Initializing a single-dimensional array

Console.WriteLine(numbers[0]);  // Outputs: 1 (First element)

Console.WriteLine(numbers[4]);  // Outputs: 5 (Last element)
```

Multi-Dimensional Arrays

A **multi-dimensional array** is an array that contains other arrays. It is essentially an array of arrays, and you can use multiple indices to access elements. The most common multi-dimensional array is the **two-dimensional array** (like a matrix).

Syntax:

```
int[,] matrix = new int[3, 3];  // Declaring a 2D array (3x3 matrix)
```

Example:

```
int[,] matrix = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };  // Initializing a 2D array

Console.WriteLine(matrix[0, 0]);  // Outputs: 1 (Element at row 0, column 0)

Console.WriteLine(matrix[2, 2]);  // Outputs: 9 (Element at row 2, column 2)
```

Jagged Arrays (Arrays of Arrays): A **jagged array** is an array whose elements are arrays, and the arrays can have different lengths. Jagged arrays are different from multi-dimensional arrays because each "row" is a separate array object, and they can be of different sizes.

Syntax:

```
int[][] jaggedArray = new int[3][];  // Declaring a jagged array with 3 rows

jaggedArray[0] = new int[] { 1, 2, 3 };

jaggedArray[1] = new int[] { 4, 5 };

jaggedArray[2] = new int[] { 6, 7, 8, 9 };
```

Example:

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

jaggedArray[0] = new int[] { 1, 2, 3 };

jaggedArray[1] = new int[] { 4, 5 };

jaggedArray[2] = new int[] { 6, 7, 8, 9 };


Console.WriteLine(jaggedArray[0][0]);  // Outputs: 1 (First element in the first row)

Console.WriteLine(jaggedArray[2][3]);  // Outputs: 9 (Last element in the third row)
```

Mixed Jagged and Multi-Dimensional Arrays

It's possible to mix jagged and multi-dimensional arrays by using a jagged array as one or more elements of a multi-dimensional array. This approach gives more flexibility but also introduces complexity.

Example:

```
int[,] multiDimArray = new int[2, 3];  // 2D array
int[][] jaggedArray = new int[2][];  // Jagged array inside the 2D array

jaggedArray[0] = new int[] { 1, 2 };
jaggedArray[1] = new int[] { 3, 4, 5 };

// Storing jagged array inside multi-dimensional array
multiDimArray[0, 0] = jaggedArray[0][0];  // Accessing element of jagged array
```

```
multiDimArray[0, 1] = jaggedArray[1][2];  // Accessing element of jagged array
```

Passing Arrays as Arguments

Arrays can be passed to methods as arguments in C#. When you pass an array to a method, you are passing the reference to the array, not a copy of the array.

Syntax:

```
void MethodName(int[] arr) { ... }
```

Example:

```
static void PrintArray(int[] arr)
{
  foreach (int num in arr)
  {
    Console.Write(num + " ");
  }
}

static void Main()
{
  int[] numbers = { 1, 2, 3, 4, 5 };
  PrintArray(numbers);  // Outputs: 1 2 3 4 5
}
```

Params Keyword and Arrays in Argument Passing

The params keyword allows you to pass an arbitrary number of arguments to a method, and those arguments are automatically treated as an array.

Syntax:

```
void MethodName(params int[] arr) { ... }
```

Example:

```
static void PrintNumbers(params int[] numbers)
{
  foreach (int num in numbers)
  {
    Console.Write(num + " ");
  }
}

static void Main()
{
  PrintNumbers(1, 2, 3, 4, 5);  // Outputs: 1 2 3 4 5
}
```

This is a convenient way to pass a variable number of parameters to a method without having to explicitly define an array.

ArrayList

The ArrayList class in C# is a part of the **System.Collections** namespace and is a non-generic collection that can hold elements of any type. Unlike arrays, an ArrayList can dynamically resize as elements are added or removed. However, it is recommended to use **generic collections** like List<T> in modern C# programming for type safety.

Syntax:

```
ArrayList list = new ArrayList();
```

Example:

```
using System;
using System.Collections;

class Program
{
  static void Main()
  {
    ArrayList list = new ArrayList();
    list.Add(1);  // Adding an integer
    list.Add("Hello");  // Adding a string
    list.Add(3.14);  // Adding a double

    foreach (var item in list)
    {
      Console.WriteLine(item);
    }
  }
}
```

Output:

- 1
- Hello
- 3.14
- You can access elements in an ArrayList using the index just like an array:
- Console.WriteLine(list[1]); // Outputs: Hello
- **Note**: Since ArrayList is non-generic, it can store elements of any data type, but you lose compile-time type safety, so it is better to use List<T> (generic version) where possible.

Threads in C#

In C#, **threads** are the basic units of execution within a program. A **thread** is a lightweight process that is used to perform tasks concurrently. Multi-threading allows a program to perform multiple operations at the same time, making it more efficient, especially for CPU-bound or I/O-bound operations.

What is a Thread?

A **thread** is a sequence of instructions that can be executed independently of other code. When you create a thread in C#, it runs concurrently with other threads in the program. The operating system manages threads by allocating time slices to each thread for execution. Each thread in an application shares the same memory space, but has its own execution stack.

In C#, the **System.Threading** namespace provides classes and methods to manage threads.

Creating and Starting a Thread

To create and start a thread in C#, you can use the Thread class. A Thread is created by passing a delegate or a method that will execute when the thread starts.

Syntax:

```
Thread thread = new Thread(MethodName);
thread.Start();
```

Example:

```
using System;
using System.Threading;

class Program
{
  static void Main()
  {
    Thread thread = new Thread(PrintNumbers);
    thread.Start();
  }

  static void PrintNumbers()
  {
    for (int i = 1; i <= 5; i++)
    {
      Console.WriteLine(i);
      Thread.Sleep(1000);  // Pauses for 1 second
    }
  }
```

```
}
```

Explanation: The PrintNumbers method will run in a separate thread, printing numbers from 1 to 5 with a 1-second delay between each print.

Thread States

A thread in C# goes through various states during its lifecycle:

1. **Unstarted**: The thread has been created but not started yet.
2. **Running**: The thread is executing its code.
3. **Waiting**: The thread is waiting for another thread to complete.
4. **Blocked**: The thread is blocked, typically waiting for a resource.
5. **Terminated**: The thread has completed its execution or has been aborted.

You can check the current state of a thread using the ThreadState enumeration.

Example:

```
Thread thread = new Thread(MethodName);
thread.Start();
Console.WriteLine(thread.ThreadState);  // Outputs: Running, if the thread is running
```

Thread Methods and Properties

- **Start()**: Starts the thread, causing it to begin executing the method it was assigned.
- **Sleep(int milliseconds)**: Pauses the thread for a specified amount of time.
- **Join()**: Makes the calling thread wait until the thread being called completes its execution.
- **Abort()**: Aborts the thread (this method is deprecated, and its use is discouraged).
- **IsAlive**: Returns true if the thread is still running.

Example:

```
Thread thread = new Thread(PrintNumbers);
thread.Start();
thread.Join();  // Wait for the thread to finish before proceeding
Console.WriteLine("Thread completed.");
```

Thread Safety

When multiple threads share data or resources, **thread safety** becomes a concern. If one thread is modifying a shared resource while another thread is trying to read or modify it, it can lead to **race conditions** and unpredictable behavior.

To avoid these issues, C# provides several synchronization mechanisms:

- **Locks (Monitor)**: The lock keyword or Monitor class ensures that only one thread can access a resource at a time.

Example:

```
private static object lockObj = new object();


static void SafeMethod()
{
   lock (lockObj)  // Ensures only one thread can execute this block at a time
   {
      // Critical section
   }
}
```

- **Mutex**: A Mutex is a synchronization primitive that can be used to manage access to a resource across multiple threads or even multiple processes.

Example:

```
Mutex mutex = new Mutex();
mutex.WaitOne();  // Acquire the mutex
// Critical section
mutex.ReleaseMutex();  // Release the mutex
```

- **Semaphore**: A Semaphore is used to limit the number of threads that can access a resource at a given time.

Thread Pooling

Instead of creating and managing threads manually, C# provides a **ThreadPool** that can be used to execute tasks on threads that are managed by the runtime. This helps to reduce the overhead of creating and destroying threads and is a more efficient approach for handling a large number of short-lived tasks.

You can use the ThreadPool class to queue work items that will be executed by threads from the pool.

Example:

```
using System;
using System.Threading;


class Program
{
   static void Main()
   {
      ThreadPool.QueueUserWorkItem(WorkItem);
   }


   static void WorkItem(object state)
   {
      Console.WriteLine("Thread from thread pool");
   }
}
```

The ThreadPool.QueueUserWorkItem() method queues a method to be executed on a thread pool thread.

Background Threads

A **background thread** is a thread that runs in the background and does not prevent the application from exiting. If all foreground threads in the application terminate, the application will exit, regardless of whether any background threads are still running.

Example:

```
Thread thread = new Thread(PrintNumbers);
thread.IsBackground = true;  // Mark the thread as a background thread
thread.Start();
```

Task Parallel Library (TPL)

In modern C# programming, the **Task Parallel Library (TPL)** is often preferred over using raw threads because it provides higher-level abstractions like Task, Parallel.For, and Parallel.ForEach, which make

it easier to perform parallel processing and manage tasks. TPL automatically handles thread pooling and synchronization.

Example using Task:

```
using System;
using System.Threading.Tasks;

class Program
{
  static void Main()
  {
    Task.Run(() => PrintNumbers());
  }

  static void PrintNumbers()
  {
    for (int i = 1; i <= 5; i++)
    {
      Console.WriteLine(i);
      Task.Delay(1000).Wait();  // Pauses for 1 second
    }
  }
}
```

Windows Forms Applications in C#

Windows Forms (WinForms) is a graphical user interface (GUI) framework provided by the .NET Framework to build desktop applications. WinForms allows you to create rich user interfaces that interact with the user through windows, buttons, textboxes, and other UI elements.

Windows Forms Overview

Windows Forms is a part of the .NET Framework that provides tools for creating rich client-side applications. It provides a variety of **controls**, **events**, and **layouts** to build functional and interactive user interfaces. The controls in a Windows Forms application are the building blocks of the UI.

Creating Windows Forms Applications

To create a Windows Forms application in Visual Studio:

1. Open Visual Studio.
2. Go to **File > New > Project**.
3. Select **Windows Forms App (.NET Framework)**.
4. Choose a project name and location.
5. Click **Create**.

Visual Studio will generate a form (Form1.cs) with a default window where you can add controls, write event handlers, and design your application.

Creating Event Handlers

An **event handler** is a method that responds to user interactions (such as clicks or key presses) with the controls in your form.

- **Example**: Creating an event handler for a button click.
    1. Drag a **Button** control from the toolbox to the form.
    2. Double-click the button to create a Click event handler.

```
private void button1_Click(object sender, EventArgs e)
{
   MessageBox.Show("Button clicked!");
}
```

In this example, when the button is clicked, a message box appears with the message "Button clicked!".

Different Controls in Windows Forms

There are a variety of controls in Windows Forms that enable user interaction. Below are some of the commonly used controls:

Button Control

A **Button** is a control that the user can click to trigger an action, such as submitting a form or opening a new window.

Example:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello, World!");
}
```

Properties: Text, Size, Location, BackColor, ForeColor.

Textbox Control

A **Textbox** is a control that allows the user to input text.

Example:

```
private void button1_Click(object sender, EventArgs e)
{
    string inputText = textBox1.Text;
    MessageBox.Show("You entered: " + inputText);
}
```

Properties: Text, MaxLength, PasswordChar, ReadOnly, Multiline.

Label Control

A **Label** is used to display static text in your form, typically used to guide the user or show information.

Example:

```
label1.Text = "Please enter your name:";
```

Properties: Text, Font, ForeColor, BackColor.

PictureBox Control

A **PictureBox** is used to display images in your form.

Example:

```
pictureBox1.Image = Image.FromFile("image.jpg");
```

Properties: Image, SizeMode, BorderStyle.

Checkbox Control

A **Checkbox** allows the user to select or deselect an option.

Example:

```
private void button1_Click(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    MessageBox.Show("Checkbox is checked!");
  }
  else
  {
    MessageBox.Show("Checkbox is unchecked!");
  }
}
```

Properties: Checked, Text, CheckState.

RadioButton Control

A **RadioButton** allows the user to select one option from a group of mutually exclusive options.

Example:

```
private void button1_Click(object sender, EventArgs e)
{
  if (radioButton1.Checked)
  {
```

```
        MessageBox.Show("Option 1 selected");
    }
    else if (radioButton2.Checked)
    {
        MessageBox.Show("Option 2 selected");
    }
}
```

Properties: Checked, Text, GroupName.

ComboBox Control

A **ComboBox** is a control that allows the user to choose from a drop-down list of items.

Example:

```
comboBox1.Items.Add("Item 1");
comboBox1.Items.Add("Item 2");
comboBox1.SelectedIndex = 0;  // Selects the first item by default
```

Properties: Items, SelectedItem, SelectedIndex, DropDownStyle.

Timer Control

The **Timer** control is used to execute code at specified intervals.

Example:

```
private void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToString();
}

private void Form1_Load(object sender, EventArgs e)
{
    timer1.Start();
}
```

Properties: Interval, Enabled.

ProgressBar Control

The **ProgressBar** control is used to show the progress of a task.

Example:

```
private void button1_Click(object sender, EventArgs e)
{
   progressBar1.Minimum = 0;
   progressBar1.Maximum = 100;
   for (int i = 0; i <= 100; i++)
   {
      progressBar1.Value = i;
      System.Threading.Thread.Sleep(50);
   }
}
```

Properties: Minimum, Maximum, Value, Step.

RichTextBox Control

The **RichTextBox** control allows the user to display or edit text with various formatting options (such as bold, italics, etc.).

Example:

```
richTextBox1.AppendText("This is a rich text box.");
richTextBox1.SelectionFont = new Font("Arial", 12, FontStyle.Bold);
```

Properties: Text, SelectionFont, SelectionColor.

MenuStrip Control

The **MenuStrip** control provides a way to add a menu bar to your Windows Forms application.

Example:

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
{
```

```
    MessageBox.Show("New file created!");
}
```

Properties: Items, Text.

ContextMenuStrip Control

The **ContextMenuStrip** control provides a context menu that appears when the user right-clicks on a control or form.

Example:

```
private void contextMenuStrip1_Opening(object sender,
System.ComponentModel.CancelEventArgs e)
{
    if (listBox1.SelectedItems.Count == 0)
    {
        e.Cancel = true;  // Cancel opening the menu if no item is selected
    }
}
```

Properties: Items, ShowImageMargin.

DataGridView Control

The **DataGridView** control is used to display and manipulate tabular data in your application.

Example:

```
private void Form1_Load(object sender, EventArgs e)
{
    dataGridView1.Rows.Add("John", "Doe", 30);
    dataGridView1.Rows.Add("Jane", "Smith", 25);
}
```

Properties: Columns, Rows, DataSource.

MDI Applications in Windows Forms

MDI (Multiple Document Interface) applications allow you to open and manage multiple child windows (forms) within a single parent window. This is commonly used in applications that need to handle multiple documents or views simultaneously, such as in word processors or image editing software.

Creating an MDI Application

In a Windows Forms application, you can set up an MDI parent form and then open child forms within it. Here's how you can create an MDI application:

1. **Set the MDI Parent Form**:
    - In the parent form's properties, set the IsMdiContainer property to true.
2. **Creating a Child Form**:
    - To create a child form, set the MdiParent property of the child form to the parent form.

Example:

```
private void openChildFormToolStripMenuItem_Click(object sender, EventArgs e)
{
  FormChild childForm = new FormChild();
  childForm.MdiParent = this;  // Set MDI parent
  childForm.Show();
}
```

In this example, the openChildFormToolStripMenuItem_Click event handler creates a new instance of FormChild, sets its MdiParent property to the parent form, and then shows it as a child form.

Dialog Boxes in Windows Forms

Dialog boxes are special types of forms that allow users to interact with the application through predefined options such as opening a file, saving data, or picking a color.

Modal and Modeless Dialog Boxes

- **Modal Dialog Boxes**: A modal dialog box requires the user to respond before they can continue interacting with the rest of the application. The user cannot interact with the parent form until the modal dialog is closed.

    Example:

    - MessageBox.Show("This is a modal dialog box.");
- **Modeless Dialog Boxes**: A modeless dialog box allows the user to interact with both the dialog and the parent form at the same time. The user can switch between the dialog and the main window.

Example:

- FormChild childForm = new FormChild();
- childForm.Show(); // This is modeless

Common Dialog Boxes

Windows Forms provides several common dialog boxes that you can use in your application to interact with the user. These include dialogs for choosing colors, fonts, and files.

ColorDialog

The **ColorDialog** allows the user to select a color from a standard color palette or by defining a custom color.

Example:

```
ColorDialog colorDialog = new ColorDialog();
if (colorDialog.ShowDialog() == DialogResult.OK)
{
   this.BackColor = colorDialog.Color;  // Set form's background color
}
```

Properties: Color (the selected color).

FontDialog

The **FontDialog** allows the user to select a font, style, and size for text formatting.

Example:

```
FontDialog fontDialog = new FontDialog();
if (fontDialog.ShowDialog() == DialogResult.OK)
{
   label1.Font = fontDialog.Font;  // Apply selected font to label
}
```

Properties: Font (the selected font).

OpenFileDialog

The **OpenFileDialog** allows the user to select a file from the file system to open in the application.

Example:

```
OpenFileDialog openFileDialog = new OpenFileDialog();
if (openFileDialog.ShowDialog() == DialogResult.OK)
{
   string fileName = openFileDialog.FileName;  // Get the selected file path
   // Open and process the file
}
```

Properties: FileName, Filter (used to restrict file types).

SaveFileDialog

The **SaveFileDialog** allows the user to specify a location and name for a file they wish to save.

Example:

```
SaveFileDialog saveFileDialog = new SaveFileDialog();
if (saveFileDialog.ShowDialog() == DialogResult.OK)
{
   string fileName = saveFileDialog.FileName;  // Get the selected file path
   // Save data to the file
}
```

Properties: FileName, Filter (used to restrict file types).

DialogResult Enumeration

The **DialogResult** enumeration represents the result of a dialog box, typically used to determine what action the user has taken (e.g., OK, Cancel, Yes, No).

Example:

```
DialogResult result = MessageBox.Show("Do you want to save changes?", "Confirm",
MessageBoxButtons.YesNo);
if (result == DialogResult.Yes)
{
   // Save changes
}
else
{
```

```
    // Discard changes
}
```

Common values:

- **OK**: The user clicked "OK".
- **Cancel**: The user clicked "Cancel".
- **Yes**: The user clicked "Yes".
- **No**: The user clicked "No".

Working with Files in C#

Introduction to the File System

The .NET Framework provides several classes for working with files and directories, allowing you to perform operations such as reading, writing, and managing files. These classes are part of the **System.IO** namespace.

- Common operations include:
  - Creating, reading, writing, and deleting files.
  - Navigating directories and subdirectories.
  - Getting information about files and directories.

DriveInfo Class

The **DriveInfo** class provides information about the drives on the computer, such as the drive type, available space, and total space.

Example:

```
DriveInfo drive = new DriveInfo("C");
MessageBox.Show("Drive Type: " + drive.DriveType);
MessageBox.Show("Available Space: " + drive.AvailableFreeSpace);
```

Properties: AvailableFreeSpace, TotalSize, DriveType, Name.

DirectoryInfo Class

The **DirectoryInfo** class is used to work with directories. It provides methods for creating, deleting, and querying directories.

Example:

```
DirectoryInfo dirInfo = new DirectoryInfo(@"C:\MyDirectory");
if (!dirInfo.Exists)
{
   dirInfo.Create();  // Create the directory if it doesn't exist
}
```

Methods: Create(), Delete(), GetFiles(), GetDirectories().

FileInfo Class

The **FileInfo** class provides methods for working with individual files, such as getting file properties, copying, or moving files.

Example:

```
FileInfo fileInfo = new FileInfo(@"C:\MyFile.txt");
if (fileInfo.Exists)
{
   MessageBox.Show("File Size: " + fileInfo.Length);
}
```

Properties: Length, Extension, CreationTime.

Methods: CopyTo(), MoveTo(), Delete().

Introduction to LINQ

LINQ (Language Integrated Query) is a powerful feature in C# that allows you to query various data sources (such as arrays, lists, XML, SQL databases, and more) using a unified syntax. LINQ provides a consistent model for working with data, making it easier to manipulate data from different sources without needing to use separate query languages.

LINQ integrates query capabilities directly into C# through language syntax. It allows you to perform operations like filtering, sorting, grouping, and joining data with just a few lines of code, making it highly readable and concise.

LINQ is available through the System.Linq namespace and provides both **LINQ to Objects** (for in-memory data structures) and **LINQ to SQL** (for querying databases), among other variations.

Basic LINQ Query and Query Operations

A basic LINQ query can be written in two ways: **query syntax** and **method syntax**. Both syntaxes are functionally equivalent, and the choice between them is largely a matter of preference.

1. Query Syntax (Declarative Syntax)

In this syntax, LINQ queries resemble SQL-like statements. Here's an example:

Example:

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


var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;


foreach (var number in evenNumbers)
{
   Console.WriteLine(number);
}
```

This LINQ query selects even numbers from the array numbers using the where clause for filtering and select for projecting the result.

2. Method Syntax (Fluent Syntax)

In method syntax, LINQ queries are written using methods like Where(), Select(), OrderBy(), etc. Here's the equivalent query written using method syntax:

Example:

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

var evenNumbers = numbers.Where(num => num % 2 == 0);

foreach (var number in evenNumbers)
{
   Console.WriteLine(number);
}
```

This approach uses Where() for filtering and Select() for projecting, mimicking the SQL WHERE and SELECT clauses.

LINQ to Collections

LINQ to Collections enables querying in-memory collections like arrays, lists, and dictionaries.

Obtaining Data Source

The data source for a LINQ query can be any collection that implements IEnumerable<T>. This could include arrays, lists, dictionaries, etc.

Example:

```
var names = new List<string> { "John", "Jane", "Jake", "Jessica" };
var query = from name in names
        where name.StartsWith("J")
        select name;

foreach (var name in query)
{
   Console.WriteLine(name);
}
```

This query filters a list of names to only include those that start with "J".

LINQ Query Operations

LINQ provides several operations for manipulating and querying data sources. These operations can be categorized as follows:

Filtering

Filtering in LINQ is accomplished using the Where() clause, which restricts the elements of a collection based on a condition.

Example:

```
var numbers = new int[] { 1, 2, 3, 4, 5, 6 };

var evenNumbers = numbers.Where(num => num % 2 == 0);

foreach (var num in evenNumbers)
{
   Console.WriteLine(num);
}
```

Here, the Where() method filters the numbers array to include only even numbers.

Ordering

LINQ allows you to sort the data in ascending or descending order using OrderBy() and OrderByDescending().

Example:

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

var sortedNumbers = numbers.OrderBy(num => num);

foreach (var num in sortedNumbers)
{
   Console.WriteLine(num);
}
```

This query sorts the numbers in ascending order.

Grouping

LINQ provides a way to group data using the GroupBy() method, which organizes the data into groups based on a key.

Example:

```
var people = new List<(string Name, int Age)>
{
   ("John", 25),
   ("Jane", 30),
   ("Jake", 25),
   ("Jessica", 30)
};
```

```
var groupedByAge = people.GroupBy(person => person.Age);

foreach (var group in groupedByAge)
{
  Console.WriteLine($"Age: {group.Key}");
  foreach (var person in group)
  {
    Console.WriteLine(person.Name);
  }
}
```

This query groups people by their age, and for each group, it prints the names of people within that group.

Joining

LINQ provides the Join() method for combining data from two collections based on a common key.

Example:

```
var students = new List<(int StudentId, string Name)>
{
  (1, "John"),
  (2, "Jane")
};

var grades = new List<(int StudentId, string Grade)>
{
  (1, "A"),
  (2, "B")
};

var studentGrades = from student in students
                    join grade in grades
                    on student.StudentId equals grade.StudentId
                    select new { student.Name, grade.Grade };

foreach (var sg in studentGrades)
{
  Console.WriteLine($"{sg.Name}: {sg.Grade}");
}
```

This query joins two lists (students and grades) on the StudentId and projects the student's name along with their grade.

Selecting

The Select() method allows you to project data into a new shape or form, enabling transformations such as selecting specific fields or creating new anonymous types.

Example:

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

var squaredNumbers = numbers.Select(num => num * num);

foreach (var num in squaredNumbers)
{
   Console.WriteLine(num);
}
```

This query squares each number in the numbers array and prints the results.

ADO.NET

ADO.NET (ActiveX Data Objects .NET) is a data access technology in .NET used to connect to relational databases, retrieve, manipulate, and update data. It provides a set of classes for accessing and working with data in databases, XML files, and other data sources. ADO.NET provides two approaches for interacting with data: connected and disconnected.

Introduction to ADO.NET

ADO.NET provides the tools to:

- Connect to a data source (like a SQL database).
- Execute commands (SQL queries, stored procedures).
- Retrieve data in a variety of formats (DataReader, DataSet).
- Manipulate and update data in the database.

ADO.NET works with different data providers (SQL Server, Oracle, MySQL, etc.) and is a core part of the .NET Framework.

Connected Data Access

In connected data access, a connection to the database is kept open while data is being retrieved or manipulated. The connection is open for the duration of the interaction with the data source, and once the data has been fetched, it is closed. This is typically used when you need real-time data from the database.

Key Components:

1. **SqlConnection**: Manages the connection to the data source.
2. **SqlCommand**: Executes SQL queries or stored procedures.
3. **SqlDataReader**: Retrieves a forward-only stream of data from the database.

Example:

```
SqlConnection conn = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", conn);

conn.Open();
SqlDataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
   Console.WriteLine(reader["EmployeeName"]);
}

reader.Close();
```

```
conn.Close();
```

Disconnected Data Access

Disconnected data access in ADO.NET allows data to be fetched from the database and stored in a local object (like DataSet or DataTable). The connection is closed after the data is retrieved, and the data can be worked with offline. This approach is useful when working with large data sets or when you need to manipulate the data before making changes to the database.

Key Components:

1. **SqlConnection**: Establishes the connection to the database.
2. **SqlDataAdapter**: Fills a DataSet or DataTable with data.
3. **DataSet/DataTable**: Stores data in memory as tables and can hold multiple related tables.
4. **SqlCommand**: Executes SQL queries or commands to manipulate data.

Example:

```
SqlConnection conn = new SqlConnection(connectionString);
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Employees", conn);

DataSet ds = new DataSet();
adapter.Fill(ds, "Employees");

foreach (DataRow row in ds.Tables["Employees"].Rows)
{
   Console.WriteLine(row["EmployeeName"]);
}
```

In this example, data is fetched into a DataSet and can be worked with offline.

CRUD Operations (Create, Read, Update, Delete)

ADO.NET enables basic CRUD operations, which are essential for interacting with data in a database.

1. **Create (Insert)**:

```
SqlCommand cmd = new SqlCommand("INSERT INTO Employees (Name) VALUES ('John Doe')", conn);

conn.Open();

cmd.ExecuteNonQuery();

conn.Close();
```

2. **Read (Select)**:

   ```
   SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", conn);
   conn.Open();
   SqlDataReader reader = cmd.ExecuteReader();
   while (reader.Read())
   {
      Console.WriteLine(reader["EmployeeName"]);
   }
   reader.Close();
   conn.Close();
   ```

3. **Update**:

   ```
   SqlCommand cmd = new SqlCommand("UPDATE Employees SET Name = 'Jane Doe' WHERE EmployeeID = 1", conn);
   conn.Open();
   cmd.ExecuteNonQuery();
   conn.Close();
   ```

4. **Delete**:

   ```
   SqlCommand cmd = new SqlCommand("DELETE FROM Employees WHERE EmployeeID = 1", conn);
   conn.Open();
   cmd.ExecuteNonQuery();
   conn.Close();
   ```

Entity Framework

Entity Framework (EF) is an Object-Relational Mapping (ORM) framework for .NET that allows developers to interact with databases using objects rather than raw SQL queries. EF abstracts the database interactions and allows working with entities, collections, and relationships using strongly typed classes.

EF simplifies database access by mapping database tables to classes and allowing LINQ queries to interact with the data.

Introduction to Entity Framework

Entity Framework provides several workflows for managing database operations:

- **Code First**: Developers write C# classes that represent the data model, and EF automatically generates the database schema.
- **Database First**: The database schema already exists, and EF generates the classes to represent the database tables.
- **Model First**: Developers create a conceptual model (e.g., using Entity Designer) and EF generates both the database schema and classes.

Code First Workflow (New Database)

In Code First, the developer defines C# classes that represent the data model, and EF automatically generates the database schema based on these classes.

Define the model (C# classes):

```
public class Employee
{
  public int EmployeeID { get; set; }
  public string Name { get; set; }
}
```

Create a DbContext class:

```
public class ApplicationDbContext : DbContext
{
  public DbSet<Employee> Employees { get; set; }
}
```

Run the migration to generate the database:

- Add-Migration InitialCreate
- Update-Database

This creates a new database and tables based on the model classes.

Code First Workflow (Existing Database)

When using Code First with an existing database, you can reverse-engineer the database schema into C# classes using EF's "Scaffolding" feature. This allows EF to generate the appropriate models based on the existing database structure.

Example:

- Scaffold-DbContext "YourConnectionString" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

This command generates the DbContext and model classes based on the existing database.

Model First

In the **Model First** approach, developers design a conceptual model (often using a visual designer), and EF generates both the database schema and classes from the model.

1. **Design the model**: You can use Visual Studio's Entity Data Model Designer to design the database model.
2. **Generate classes and database**: EF generates C# classes and the database schema from the visual model.

Database First

In the **Database First** approach, the database schema exists, and EF generates C# classes to represent the database tables. You can use Visual Studio's tools to generate a model from an existing database.

1. **Create a Model from Database**: Right-click the project > Add > New Item > ADO.NET Entity Data Model > Generate from database.
2. **Select the database**: EF generates the classes based on the tables, views, and stored procedures in the database.

Subject Work Code

The code for this subject is available on GitHub. You can access and review the code repository using the following link:

GitHub Repository: Visual Programming

Please refer to this link for all related code and further development details.

The End