C# Basic

C# Basic syntax and structure of the C# language. Understand concepts like variables, data types, operators, control structures (if statements, loops), functions, and classes.

Saroj Kumar Jha

CodingWithSaroj

Please subscribe

C#

C# (pronounced "C sharp") is a popular, general-purpose, object-oriented programming language developed by Microsoft. It was first introduced in the early 2000s as part of the Microsoft .NET framework and has since gained widespread adoption among developers.

C# is often used for developing desktop applications, web applications, mobile apps, and games. It combines elements of C and C++ with features from other modern programming languages such as Java. C# is known for its simplicity, readability, and strong typing, making it relatively easy to learn and use.

Some key features and concepts in C# include:

Object-oriented programming: C# supports classes, objects, inheritance, and polymorphism, allowing developers to build modular and reusable code.

Garbage collection: C# includes automatic memory management through a garbage collector, which helps developers avoid memory leaks and manually deallocating memory.

Strong typing: C# enforces strong type checking at compile-time, helping to catch errors early in the development process.

C#

Language Integrated Query (LINQ): C# includes LINQ, a powerful querying syntax that enables developers to query and manipulate data from various sources such as databases, XML, and collections.

Asynchronous programming: C# has built-in support for asynchronous programming using the async and await keywords, allowing developers to write responsive and scalable applications.

Exception handling: C# provides robust exception handling mechanisms to catch and handle runtime errors, improving the reliability of applications.

To develop C# applications, you typically use an Integrated Development Environment (IDE) like Visual Studio or JetBrains Rider, which provide tools for writing, debugging, and compiling C# code.

C# is a versatile language that can be used for a wide range of applications, from small utilities to large enterprise systems. It has a vibrant developer community and is supported by a rich ecosystem of libraries and frameworks.

Agenda

Let’s Start

Variable

Operators

If loop

Switch case

While

For

ForNext

Break; Continue;

Goto

Agenda

- Class
  - Fields
  - Methods
  - Constructor
  - Properties
  - etc.
- Inheritance
- Abstract Class
- Interface
- Static Class
- Structure
- Enum
- Array
- Collections
- In Next Video
- Extension method
- Delegates and Event

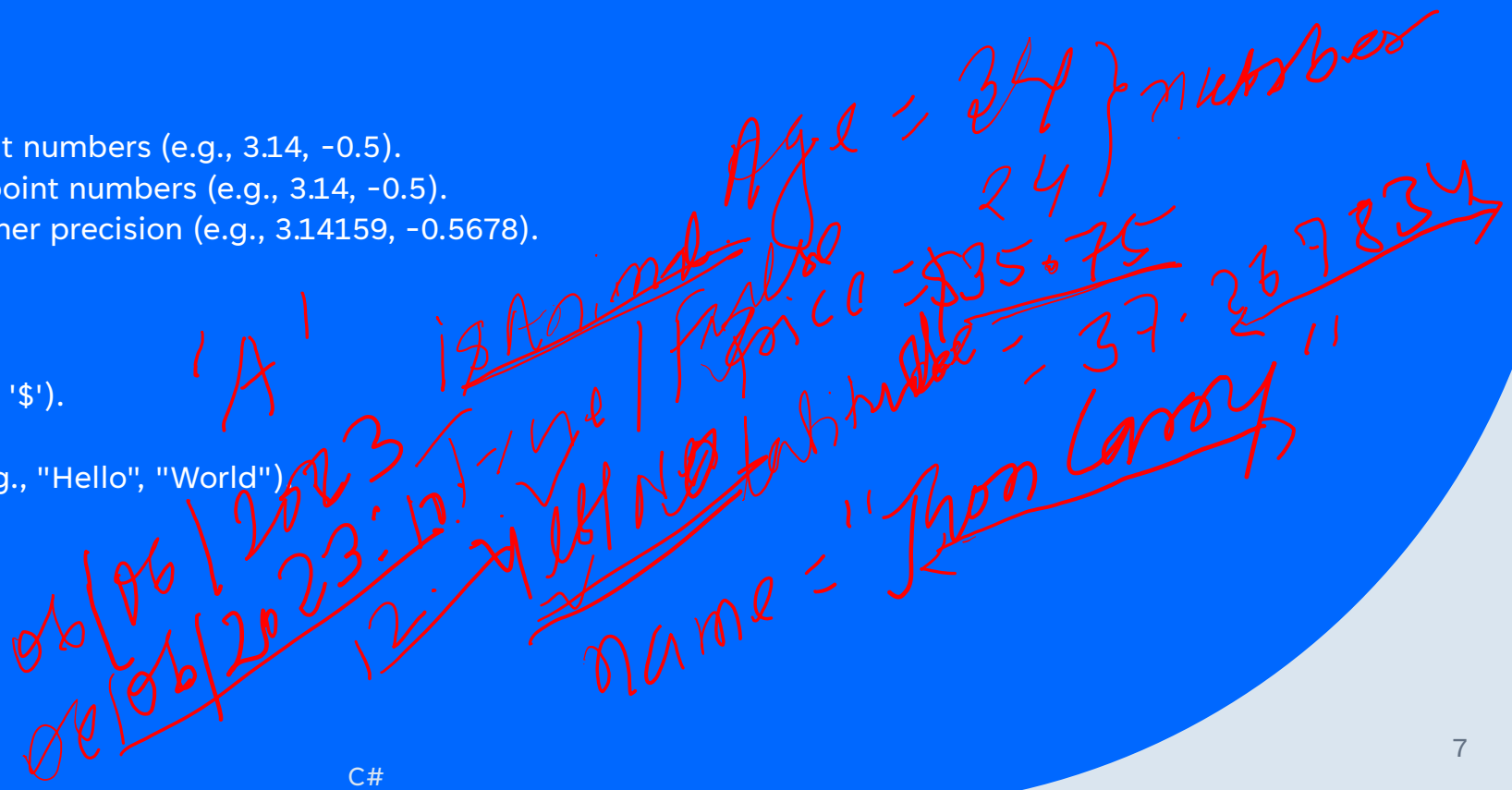
Agenda

- Anonymous Types
- Tuple
- LINQ
- String operation
- Datetime operation
- Input Output (IO)
- Serialization
- Encoding
- Exception Handling

Data Types

In C#, data types define the type of data that a variable can hold. C# provides various built-in data types that you can use to declare variables. Here are some commonly used data types in C#:

Numeric Types:
int: Used to store whole numbers (e.g., 1, 10, -5).
float: Used to store single-precision floating-point numbers (e.g., 3.14, -0.5).
double: Used to store double-precision floating-point numbers (e.g., 3.14, -0.5).
decimal: Used to store decimal numbers with higher precision (e.g., 3.14159, -0.5678).
Boolean Type:
bool: Used to store either true or false values.
Character Types:
char: Used to store a single character (e.g., 'a', '9', '$').
String Type:
string: Used to store a sequence of characters (e.g., "Hello", "World").
Date and Time Types:
DateTime: Used to store dates and times.
TimeSpan: Used to store a duration of time.
DateOnly :Used to store date only.
TimeOnly: Used to store time only.



Data Types

Arrays:
int[], string[], etc.: Used to store collections of values of the same data type.
Object Type:
object: The base type for all other types in C#. It can store any type of value.

Variable

In C#, a variable is a named storage location that holds a value of a specific type. Variables are used to store and manipulate data during the execution of a program. Here's the basic syntax for declaring a variable in C#:

```
<type> <variableName>;
```

In this syntax, <type> represents the data type of the variable, and <variableName> is the name you choose for the variable.

```
int age; // Declaring an integer variable named "age"
```

You can also assign an initial value to a variable at the time of declaration:

```
int age = 25; // Declaring an integer variable named "age" and assigning it the value 25
```

Once you have declared a variable, you can assign new values to it using the assignment operator (=):

```
age = 30; // Assigning a new value (30) to the "age" variable
```

C# also supports the concept of implicit typing using the var keyword. With implicit typing, the compiler infers the type of the variable based on the assigned value:

```
var name = "John"; // Implicitly typed variable "name" with the value "John" (inferred as string)
```

In this example, the compiler automatically determines that the variable name should be of type string based on the assigned value.

It's important to note that C# is a statically typed language, which means that once you declare a variable with a specific type, you cannot change its type later.

Operators

In C#, operators are special symbols or keywords that perform operations on one or more operands and produce a result. C# provides various types of operators that can be classified into different categories based on their functionality. Here are the most commonly used categories of operators in C#:

1. Arithmetic Operators:
   - Addition (+): Adds two operands.
   - Subtraction (-): Subtracts the second operand from the first.
   - Multiplication (*): Multiplies two operands.
   - Division (/): Divides the first operand by the second.
   - Modulus (%): Returns the remainder of the division.
2. Assignment Operators:
   - Assignment (=): Assigns a value to a variable.
   - Compound Assignment Operators (e.g., +=, -=, *=): Performs an operation and assigns the result to the variable.
3. Comparison Operators:
   - Equal to (==): Checks if two operands are equal.
   - Not equal to (!=): Checks if two operands are not equal.
   - Greater than (>): Checks if the left operand is greater than the right operand.
   - Less than (<): Checks if the left operand is less than the right operand.
   - Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
   - Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

Operators

Logical Operators:

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if at least one operand is true.
- Logical NOT (!): Reverses the logical state of the operand.

Increment and Decrement Operators:

- Increment (++): Increases the value of the operand by 1.
- Decrement (--): Decreases the value of the operand by 1.

Bitwise Operators:

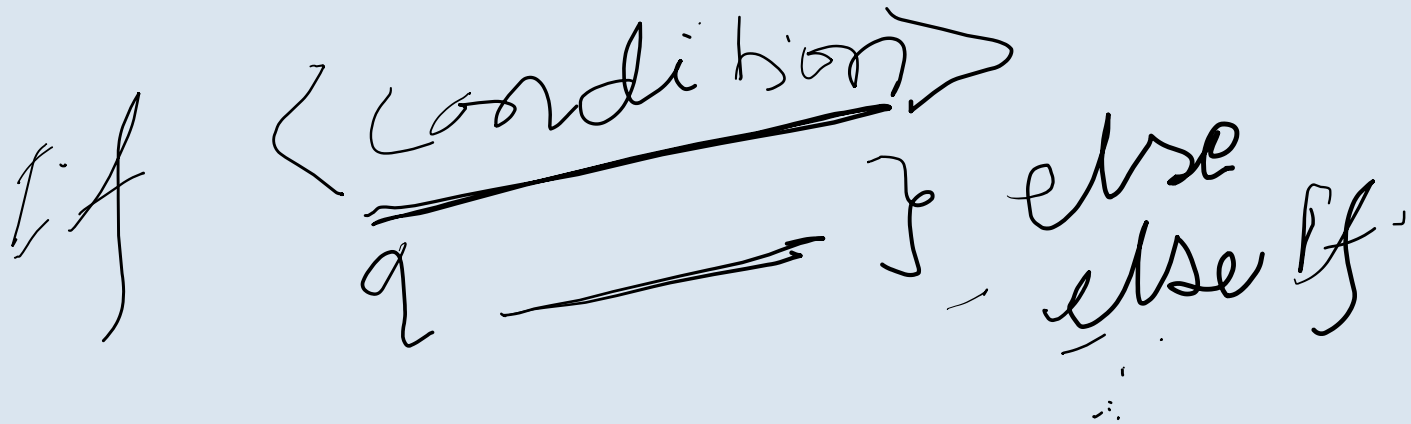
- Bitwise AND (&): Performs a bitwise AND operation on the operands.
- Bitwise OR (|): Performs a bitwise OR operation on the operands.
- Bitwise XOR (^): Performs a bitwise exclusive OR operation on the operands.
- Bitwise NOT (~): Inverts the bits of the operand.
- Left Shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
- Right Shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Conditional Operator:

- Conditional (ternary) operator (?:): Evaluates a condition and returns one of two expressions based on the result.

These are just some of the commonly used operators in C#. There are additional operators available for specific purposes, such as type casting, nullability, and more.

If Loop



In C#, the if statement is used to conditionally execute a block of code based on a specified condition. It allows you to perform different actions depending on whether the condition evaluates to true or false. Here's the basic syntax of the if statement:

```
if (condition)          {          // Code to be executed if the condition is true          }
```

If the condition is true, the code inside the block will be executed. If the condition is false, the block will be skipped, and the program will continue with the next statement after the if block.

Here's an example that demonstrates the usage of the if statement:

```
int number = 10;
if (number > 5)  {     Console.WriteLine("The number is greater than 5.");  }
```

In this example, the condition number > 5 is evaluated. If the condition is true (which it is because number is 10), the code inside the if block will execute, and the message "The number is greater than 5." will be printed to the console.

You can also include an optional else block to specify an alternative code block to execute when the condition is false:

```
int number = 3;
if (number > 5) {    Console.WriteLine("The number is greater than 5."); }
else {    Console.WriteLine("The number is less than or equal to 5.");  }
```

In this case, since number is 3, the condition number > 5 is false, and the code inside the else block will execute instead. The message "The number is less than or equal to 5." will be printed to the console.

If Loop

You can also use the else if clause to specify additional conditions to check:

```
int number = 7;
if (number > 10) {    Console.WriteLine("The number is greater than 10."); }
else if (number > 5) {    Console.WriteLine("The number is greater than 5 but less than or equal to 10.");}
else {    Console.WriteLine("The number is less than or equal to 5.");}
```

In this example, different messages will be printed based on the value of number. If number is greater than 10, the first condition will be true. If it's greater than 5 but less than or equal to 10, the second condition will be true. Otherwise, the code in the else block will be executed.

Loop

In C#, you can create loops using various constructs, such as for, while, do-while, and foreach. These constructs allow you to repeat a block of code multiple times until a specified condition is met. Here are examples of each loop construct:

For loop:

The for loop is commonly used when you know the number of iterations in advance. It consists of an initialization, a condition, and an iterator statement.

```
for (int i = 0; i < 5; i++)
{
  // Code to be executed repeatedly
 Console.WriteLine("Iteration: " + i);
}
```

While loop:

The while loop repeats a block of code while a specified condition is true. The condition is checked before each iteration.

```
int i = 0;
while (i < 5)
{
  // Code to be executed repeatedly
 Console.WriteLine("Iteration: " + i);
 i++;
}
```

Loop

Do-While loop:

The do-while loop is similar to the while loop, but the condition is checked after each iteration. This guarantees that the loop will execute at least once.

```
int i = 0;
do {      // Code to be executed repeatedly
 Console.WriteLine("Iteration: " + i);
 i++;
} while (i < 5);
```

Foreach loop:

The foreach loop is used to iterate over elements in an array or any collection that implements the IEnumerable interface.

```
int[] numbers = { 1, 2, 3, 4, 5 };
foreach (int number in numbers)  {            // Code to be executed for each element
 Console.WriteLine("Number: " + number);
}
```

These loop constructs provide different ways to control the flow of execution and repeat code based on specific conditions. Choose the appropriate loop construct based on your requirements and the nature of the problem you are trying to solve.

class

In C#, a class is a fundamental building block of object-oriented programming (OOP). It is a blueprint or a template that defines the characteristics and behaviors of an object. You can think of a class as a user-defined data type that encapsulates data (attributes or fields) and methods (functions) related to a specific entity or concept.

```
public class Person
{
    // Fields (attributes)
    public string Name;
    public int Age;

    // Methods
    public void SayHello()
    {
        Console.WriteLine("Hello, my name is " + Name + " and I am " + Age + " years old.");
    }
}
```

In the above code, we have defined a class called "Person." It has two fields: Name of type string and Age of type int. We also have a method SayHello() that prints a message to the console, using the values of the Name and Age fields.

class

To create an instance (object) of the Person class, you can do the following:

```
Person person = new Person();
person.Name = "John";
person.Age = 25;
person.SayHello();  // Output: Hello, my name is John and I am 25 years old.
```

In this example, we create a new instance of the Person class using the new keyword. We then assign values to the Name and Age fields of the person object. Finally, we call the SayHello() method, which prints the greeting message to the console using the object's field values.

Classes in C# provide a way to define reusable and modular code. They enable you to create multiple objects with similar attributes and behaviors, making it easier to manage and organize your code.

Abstract class and Interface

An abstract class is a template definition of methods and variables in a specific class, or category of objects. In programming, objects are units of code, and each object is made into a generic class. Abstract classes are classes that contain one or more abstracted behaviors or methods.

Here are some points regarding abstract class.

1.Abstract class can contain abstract members as well as non-abstract members in it.
2.A class can only inherit from one abstract Class.
3.We cannot create object of an abstract class.

Interface

It is also user defined type like a class which only contains abstract members in it. These abstract members should be given the implementation under a child class of an interface. A class can be inherited from a class or from an interface.

Points to remember

1.Interface contains only abstract methods.
2.We cannot create object of an interface.
3.The default scope for a member in Interface is Public. So, no need to use the Public access specifier in the program.

NOTE - In case of multiple inheritance, use Interface.

Static class

The class which is created by using the static modifier is called a static class in C#. A static class can contain only static members. It is not possible to create an instance of a static class. This is because it contains only static members. And we know we can access the static members of a class by using the class name.

Structure

In C#, both structures and classes are used to define types, but they have some fundamental differences in terms of their behavior and usage. Here are the key differences between structures and classes:

Type of Reference:

Structures are value types, which means they are stored directly on the stack or as part of another object. When a structure is assigned to a new variable or passed as a method parameter, a copy of the structure is created.

Classes, on the other hand, are reference types, which means they are stored on the heap and accessed through references. When a class object is assigned to a new variable or passed as a method parameter, the reference to the object is copied, not the entire object itself.

Default Initialization:

Structures are automatically initialized with their default values when they are declared. Numeric types are set to 0, bool types are set to false, and reference types are set to null.

Classes, however, are not automatically initialized. They need to be instantiated using the new keyword before their members can be accessed.

Structure

Inheritance and Polymorphism:
Structures cannot inherit from other structures or classes, and they cannot be the base of a class. They also do not support polymorphism. Classes, on the other hand, can inherit from other classes or interfaces, allowing for the creation of class hierarchies. They also support polymorphism, which means derived classes can be treated as instances of their base class or interface.

Performance Considerations:
Structures are generally more lightweight and faster to allocate and deallocate compared to classes. Since they are value types, they are stored directly in memory and do not require heap allocation. However, copying structures can be more expensive due to their value semantics.
Classes are typically larger in size due to their reference semantics, as they store references to the actual data on the heap. They involve heap allocation and deallocation, which can introduce some overhead. However, copying class references is relatively cheap.

Usage Guidelines:
Structures are often used for small, lightweight, and immutable data types, such as coordinates, points, or simple data containers. They are suitable for scenarios where copying values is preferable or when you need to avoid heap allocation.
Classes are commonly used for more complex data structures, objects with mutable state, or when you need to define behavior through methods and inheritance. They are suitable for scenarios where you want to work with references to the same data or share instances across multiple parts of the code.
It's important to consider these differences when choosing between structures and classes in C#, as they can have implications for memory usage, performance, and the behavior of your code.

Enum

In C#, an enum is a value type that represents a set of named constants. It allows you to define a list of named values, where each value represents a specific constant. This makes it easier to work with a predefined set of related values instead of using arbitrary numbers or strings.
To define an enum in C#, you use the enum keyword followed by the name of the enumeration. Here's an example:

```
enum DaysOfWeek
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

In this example, DaysOfWeek is the name of the enumeration, and it contains seven named constants representing the days of the week.

You can use the enum by referencing its name and the desired constant value. For example:

```
DaysOfWeek today = DaysOfWeek.Wednesday;
```

Enum

You can also assign specific integer values to the enum constants if needed. By default, the first constant is assigned a value of 0, and subsequent constants are assigned values incremented by 1. However, you can explicitly assign values to the constants. Here's an example:

```
enum ErrorCode
{
    None = 0,
    NotFound = 404,
    Unauthorized = 401,
    InternalServerError = 500
}
```

In this case, None is explicitly assigned a value of 0, NotFound is assigned 404, Unauthorized is assigned 401, and so on.

enum types are often used for representing a limited set of options or states in a program. They can improve code readability, maintainability, and type safety by enforcing the use of specific predefined values instead of arbitrary integers or strings.

Array

In C#, an array is a data structure that allows you to store multiple values of the same type in a sequential manner. It provides a convenient way to work with a collection of elements. Here's how you can declare, initialize, and use arrays in C#:

1. Declaring an array:

To declare an array, you need to specify the type of elements it will store, followed by square brackets ([]), and the name of the array. For example, to declare an array of integers, you can use the following syntax:

```
int[] numbers;
```

2. Initializing an array:

Once you've declared an array, you can initialize it with values. There are several ways to initialize an array:

a. Initialize with values directly:

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

b. Initialize with a specified size and default values:

```
int[] numbers = new int[5]; // Creates an array of size 5 with default values (0 for integers)
```

c. Initialize with values using the new keyword:

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

3. Accessing array elements:

Array elements are accessed using zero-based indexing. You can access individual elements by specifying the index within square brackets ([]). For example, to access the third element in the numbers array, you can use the following syntax:

```
int thirdElement = numbers[2]; // Index 2 represents the third element
```

Array

4. Modifying array elements:

You can modify array elements by assigning new values to them. For example:

```
numbers[0] = 10; // Modifying the first element to be 10
```

5. Array length:

The length of an array (the number of elements it can store) can be obtained using the Length property. For example:

```
int arrayLength = numbers.Length;
```

6. Iterating through an array:

You can use loops, such as for or foreach, to iterate through the elements of an array. For example, using a for loop:

```
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine(numbers[i]);
}
```

That's a basic overview of working with arrays in C#. There are many more operations and features you can use with arrays, such as sorting, searching, and multidimensional arrays.

Collection

In C#, a collection is a group of related objects that can be stored and manipulated as a single unit. Collections provide a way to organize and manage groups of items, such as a list of employees, a set of customer records, or a dictionary of key-value pairs. C# provides several built-in collection classes in the .NET Framework, which are commonly used to store and manipulate data. Here are some of the commonly used collection classes in C#:

List<T>: Represents a dynamic-sized list of elements of type T. It allows you to add, remove, and access elements using index-based notation. It provides methods like Add, Remove, Insert, Contains, and more.

Dictionary<TKey, TValue>: Represents a collection of key-value pairs. It allows you to store elements based on a unique key and retrieve the corresponding value efficiently. It provides methods like Add, Remove, ContainsKey, TryGetValue, and more.

HashSet<T>: Represents a set of unique elements. It does not allow duplicate elements and provides methods to add, remove, and check for the existence of an element.

Queue<T>: Represents a first-in, first-out (FIFO) collection of elements. It provides methods like Enqueue, Dequeue, and Peek for adding, removing, and accessing elements, respectively.

Stack<T>: Represents a last-in, first-out (LIFO) collection of elements. It provides methods like Push, Pop, and Peek for adding, removing, and accessing elements, respectively.

Collection

LinkedList<T>: Represents a doubly-linked list of elements. It allows you to efficiently add, remove, and access elements in a sequence.

These are just a few examples of collection classes in C#. Depending on your specific needs, you may also find other collection classes like SortedList, SortedSet, ObservableCollection, etc., useful. Additionally, C# also provides the IEnumerable and ICollection interfaces that can be implemented to create custom collections.

Thank you

Saroj Kumar Jha

srojkrjha@gmail.com