C# Assignment

1.) Do we define global variable in camel Case or in Pascal Case?

Ans:

for constant variables

In C#, constant variables are typically defined using PascalCase, not camelCase. The convention for naming constant variables follows the same PascalCase convention as other class-level members like fields, properties, and methods. Using PascalCase helps distinguish constant variables from other variables and makes your code more readable and consistent with C# naming conventions.

For variables-

While it's technically possible to define global variables using camelCase (the first letter in lowercase with subsequent words capitalized), it is not a common or recommended practice in C#. Using camelCase for global variables could lead to confusion and make it harder to distinguish them from local variables or fields.

In summary, it's best practice to define global variables in Pascal Case to maintain code clarity and consistency with C# naming conventions.

In practice, most C# developers tend to use PascalCase for global variables because it aligns with the convention for naming classes and properties, making it easier to distinguish global variables from local variables and fields.

2.) Difference between Variables and Properties?

Ans: - variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Example-

int i, j;

char c, ch;

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called Fields. Properties are an extension of fields and are accessed using the same syntax. They use accessors through which the values of the private fields can be read, written or manipulated.
Properties do not name the storage locations. Instead, they have accessors that read, write, or compute their values.
For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

3.) Difference between Functions and Subroutine?

Ans: - Functions and subroutines are both essential programming constructs used to group and organize code for reusability and modularity, but they have some key differences in terms of their behavior and how they are used.

Subroutines and functions have the following similarities:

- They can be internal or external.
  - Internal
    - Can pass information by using common variables
    - Can protect variables with the procedure instruction
    - Can pass information by using arguments.
  - External
    - Must pass information by using arguments
    - Can use the ARG instruction or the ARG built-in function to receive arguments.
- They use the RETURN instruction to return to the caller.

Subroutines:

- Subroutines, also known as procedures or void methods, do not return a value. They perform a sequence of actions or operations but do not produce a result that can be used directly.

- Subroutines are used when you want to group a set of statements together to perform an action or task without the need for returning a value. They are often used for their side effects, such as modifying object states or performing I/O operations.
- In C#, subroutines are commonly referred to as "methods" or "void methods." Methods in C# can be defined without a return value (void), and they are used to group a set of statements together to perform a specific action or task.

Functions:

Functions are designed to return a value to the caller. They perform a specific task or computation and provide a result back to the calling code.

Functions are used when you want to perform a specific task or calculation and obtain a result that you can use or manipulate. You call a function, and it returns a value.

Example

```
int add (int a, int b)
{
  sum = a + b;
   return sum;
 }
```

4.) All datatypes in c#?

Ans: - A data type specifies the size and type of variable values

1. Integral Types:

- `sbyte`: Signed 8-bit integer.
- `byte`: Unsigned 8-bit integer.
- `short`: Signed 16-bit integer.
- `ushort`: Unsigned 16-bit integer.
- `int`: Signed 32-bit integer.
- `uint`: Unsigned 32-bit integer.

- `long`: Signed 64-bit integer.

- `ulong`: Unsigned 64-bit integer.

Example-

```
int integerNumber = 42;

byte byteValue = 255;

long longNumber = 123456789012345;
```

2. Floating-Point Types: - `float`: Single-precision floating-point number.

- `double`: Double-precision floating-point number.

Example-

```
float floatValue = 3.14f;

double doubleValue = 2.71828;

decimal decimalValue = 123.456m;
```

4. Character Types: - `char`: A single Unicode character.

Example-

```
char charValue = 'A';
```

5. Boolean Type: - `bool`: Represents a Boolean value (true or false).

Example-

```
bool isTrue = true;

bool isFalse = false;
```

6. String Type: - `string`: Represents a sequence of characters.

Example

```
string stringValue = "Hello, C#!";
```

Object type- object represents the base type for all other data types.it is reference type.

Example

```
object anyObject = "This can hold any type of object.";
```

7. Enumeration Types: - `enum`: A user-defined data type consisting of named constants.

```
enum DaysOfWeek
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
DaysOfWeek today = DaysOfWeek.Monday;
```

8. Nullable Value Types: - `Nullable<T>` or `T? `: Allows value types to have a null value.

```
int? nullableInt = null;
```

9. Reference Types:

- `object`: The base class for all C# types.
- `dynamic`: Represents an object whose operations are resolved at runtime.
- `string`: Yes, it's also a reference type, but it's worth mentioning twice for its significance.

```
class Person
```

```
{
    public string Name;
    public int Age;
}
Person person1 = new Person { Name = "John", Age = 30 };
```

10. Arrays: - Arrays are used to store collections of elements of the same type.

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

11. User-Defined Types: - You can create your own custom data types using classes and structures.

12.Pointer Types (Unsafe Context Only): - C# allows the use of pointers in unsafe code blocks to work with memory directly.

13. Delegate Types: - Used to declare and work with delegates, which are used for event handling and callbacks.

14. Function Types (Delegates): - You can define custom delegate types for method signatures.

15. Value Tuple Types: - Introduced in C# 7.0, they allow you to create lightweight, unnamed tuples.

16. Anonymous Types: - Generated by the compiler for query expressions.

17. Custom Structs and Classes: - You can define your own custom structures and classes to represent complex data.

5.) Conditions in c#?

Ans: - In C#, conditions are used to make decisions in your code, and they control the flow of execution based on whether a specified condition is true or false. Conditions are typically used within control structures like if statements, switch statements, and loops. Here are some common ways to express conditions in C#:

1. Comparison Operators: - You can use comparison operators to compare values. Common comparison operators include:

- `==` (equal to)
- `! =` (not equal to)
- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)

Example:

```
int a = 12;

int b = 5;

if (x > y)

{

   // Code to execute if the condition is true

}
```

2. Logical Operators: - Logical operators allow you to combine multiple conditions. Common logical operators include:

- `&&` (logical AND)
- `||` (logical OR)
- `! ` (logical NOT)

Example:

```
int age = 25;
```

```
bool isStudent = true;
if (age >= 18 && isStudent)
{
    // Code to grant a student discount
}
```

3. Ternary Operator (Conditional Operator): - The ternary operator (`? : `) allows you to create concise conditional expressions.

Example:

```
int result = (x > y)? x: y;
```

4. Conditional Statements (if, else if, else): - The `if` statement allows you to execute code conditionally based on a Boolean expression. You can also use `else if` and `else` to specify alternative conditions.

Example:

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

5. Switch Statement: - The `switch` statement allows you to select one of many code blocks to be executed.

Example:

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

6. Pattern Matching (C# 7.0 and later): - Pattern matching allows you to match values against patterns and extract information from them.

Example:

```
object obj = "Hello";
if (obj is string str)
{
    Console. WriteLine ($"The length of the string is {str. Length}");
}
```

7. Nullable Types and Null Conditional Operator (?.): - You can use `Nullable<T>` to represent nullable value types and the `? ` operator to conditionally access members of objects that may be null.

Example:

```
int? nullableValue = null;
int length = nullableValue? ToString (). Length?? 0;
```

Conditions are essential for making decisions and controlling the flow of your C# programs. They allow your code to respond to different scenarios and user inputs.

6.) -12 and +12 time zone exist? If yes than it will meet or not?

Ans: - Yes, there are time zones that are 12 hours ahead and 12 hours behind Coordinated Universal Time (UTC). The time zone that is 12 hours ahead of UTC is often referred to as UTC+12, while the time zone that is 12 hours behind UTC is often referred to as UTC-12.

UTC+12 is observed in several places, including some islands in the Pacific Ocean, such as Fiji, Wallis and Futuna, and Tuvalu. UTC-12 is not a commonly used time zone and is not observed in many places, but it does exist.

If you were to compare a location in UTC+12 with a location in UTC-12, there would be a 24-hour time difference between them. In other words, when it is noon (12:00 PM) in a UTC+12 time zone, it would be midnight (12:00 AM) in a UTC-12 time zone.

7.) Program using bitwise operators, datatypes, conversion, statements, conditional statement?

Ans: - using System;

```
class Program
{
   static void Main()
   {
      int res;
      Console.WriteLine("Enter the first number");
      int n1 = Convert.ToInt32(Console.ReadLine());

      Console.WriteLine("Enter the second number");
      int n2 = Convert.ToInt32(Console.ReadLine());

      res = n1 + n2;
      Console.WriteLine("Sum " + res);

      res = n1 - n2;
      Console.WriteLine("Difference " + res);

      res = n1 * n2;
      Console.WriteLine("Multiplication " + res);

      res = n1 & n2;
```

```
        Console.WriteLine("Bitwise AND " + res);

        res = n1 | n2;
        Console.WriteLine("Bitwise OR " + res);

        res = n1 ^ n2;
        Console.WriteLine("Bitwise XOR " + res);

        res = ~n1;
        Console.WriteLine("Bitwise Complement " + res);

        res = n1 << 2;
        Console.WriteLine("Bitwise Left Shift " + res);

        res = n1 >> 2;
        Console.WriteLine("Bitwise Right Shift " + res);


        if (n2 != 0)
        {
            res = n1 / n2;
            Console.WriteLine("Quotient " + res);
        }
        else
        {
            Console.WriteLine("Cannot divide by zero");
        }
    }
}
```

8.) Loops in c#?

Ans: - Loops: - Loops in programming are structures that allow you to repeatedly execute a block of code as long as a certain condition is true. They are essential for automating repetitive tasks and iterating over collections of data. In C#, there are several types of loops, including:

For loop-

```
for (int i = 0; i < 5; i++)

{

   Console.WriteLine(i);

}
```

While loop-

```
int j = 0;

while (j < 5)

{

   Console.WriteLine(j);

   j++;

}
```

Do-while loop-

```
int k = 0;

do

{

   Console.WriteLine(k);

   k++;

} while (k < 5);
```

For each loop- Specially used for iterating over elements in arrays or collections.

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

```
foreach (int num in numbers)

{

   Console.WriteLine(num);

}
```

Breaks: - A "break" is a keyword or statement that is used to abruptly exit or terminate a loop or switch statement before it has completed all its iterations or cases.

The use of "break" allows you to prematurely exit from a loop or switch statement based on a certain condition or when a specific criterion is met.

Continue-Skips the rest of the code inside the loop for the cureent iteration and moves to the next iteration.

9.) **Functions: -** A function is a self-contained block of code that performs a specific task or set of tasks. Functions are designed to be reusable and modular, allowing you to break down a program into smaller, more manageable parts.
functions are called methods, and they play a crucial role in structuring and organizing code.

```
access_modifier return_type MethodName(parameter_list)
{
   // method body
}
```

Example-

```
public int AddNumbers(int num1, int num2)
{
   return num1 + num2;
}
```

Calling a function

```
int result = AddNumbers(5, 7);
```

void methods-

```
public void DisplayMessage(string message)
{
   Console.WriteLine(message);
}
```

A method with void doesn't return a value.

10.) **Pointers:** -Pointers in C# are a powerful feature of the language, but they are not as commonly used or as explicit as in languages like C or C++. In C#, pointers are primarily used within "unsafe" code blocks, which allow you to work with memory addresses directly.

Unsafe code:

Pointers in C# are generally used within a block of code marked as unsafe. This is done to indicate that the code inside the block might use pointers and perform operations that are not subject to some of the usual safety checks.

```
unsafe
{
   // Unsafe code here
}
```

Declaration

```
int* ptr;
```

Initialization

```
int a = 10;
int* ptr = &a;
```

The & operator is used to get the memory address of a variable.

11.) Difference between ref and out.

Ref keyword- when using ref, the variable must be initialized before it passed to the method.

```
void ModifyValue(ref int x)

{

   x = x * 2;

}

int number = 5;

ModifyValue(ref number);
```

Bidirectional-parameter is bidirectional

Input requirement- the variable passed as ref must be initialized before the method is called.because method might read its initial value.

Out keyword-

No initialization- The variable with out doesn't need to be initialized before it is passed to the method. The method is expected to initialize it.

```
void InitializeAndReturn(out int y)

{

   y = 10;

}


// Usage

int result;

InitializeAndReturn(out result);
```

Output requirement-

The method using out is expected to assign a value to the parameter before it exits. The initial value of the variable passed as out is not considered.

Output-Only

It indicates that the parameter is used for output purposes, and the method is not expected to read its initial value.

12.) How much arguments we can pass into main method ?

The Main method is the entry point of a console application.

```
static void Main(string[] args)
{
   // Code here
}
```

The args parameter is an array of strings, representing the command-line arguments passed to the program when it is executed.
You can technically pass as many command-line arguments as you want. The args array will contain all the arguments passed to the program, and you can access them using array indexing.

```
static void Main(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine($"Argument {i + 1}: {args[i]}");
    }
}
```