Interview Questions You Should Know – C#

What are Namespaces?

Namespaces organize code within the .NET Framework so that code files can locate types. By default, the namespace for a project matches the project name, and each subfolder within the directory creates a new namespace branch (separated by periods). If we want to use a type from a different namespace we need to add a **using statement** to the top of the file or **fully qualify** the namespace.

```
using System.Data.SqlClient;
            …
SqlConnection myconnection;
            OR
// fully qualified
System.Data.SqlClient.SqlConnection myconnection;
```

What is a constructor?

A constructor is a **function member** that is executed when an **instance** of the class is created in memory. The constructor must have the same name as the class, and can be **overloaded** to take parameters. Constructors are most often used as "setup" methods to ensure a class is created in a valid state for use. If you specify parameters in a constructor, the class cannot be **instantiated** (created) in memory without providing those parameters.

```
class Rectangle
{
    public double Length { get; set; }
    public double Width { get; set; }

    // Constructor has no return type and shares the name of the class
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }
}
```

Strings are immutable in C#, what does that mean?

Immutable means the string values cannot be changed once they are created. Any modification to a string value destroys the old memory location and creates a brand new one. This can be inefficient, so when we are making many modifications to a string we should use the System.Text.StringBuilder class.

What is the GAC?

The GAC is the Global Assembly Cache. Shared assemblies live in the GAC, which allows multiple applications to share DLLs instead of having to deploy them with each application. The GAC supports versioning. (Note, as a junior you won't be doing this, but you should know it exists).

What is overloading?

Overloading is when the same method name is used with different signatures. A **signature** of a method is the return type and all of the types of its parameters. You can re-use a method name as long as the signature is different.

```
public int Sum(int x, int y)
{
    return x + y;
}

public double Sum(double x, double y)
{
    return x + y;
}

public int Sum(int x, int y, int z)
{
    return x + y + z;
}
```

What is inheritance?

Inheritance is when one class receives data and function members from a **base** (parent) class. It is a pillar of object oriented development and allows code reuse. We can only inherit from one class in C#, but we can build an **inheritance chain** of many classes to simulate multiple inheritance. We specify inheritance in C# by putting a colon (:) after the class name.

```
class Animal
{
    public string Type { get; set; }
}

// Dog inherits from Animal, thus it gets all public and protected members, like type
class Dog : Animal
{
    public Dog()
    {
        Type = "Canine";
    }
}
```

With inheritance, we can use the **abstract** keyword to force children to implement a function member using the **override** keyword. If you declare any function member as abstract, the base class must also be marked as abstract. An abstract class cannot be directly instantiated.

```
abstract class Animal
{
    public string Type { get; set; }
    public abstract string Greet();
}
```

```
class Dog : Animal
{
    public Dog()
    {
    Type = "Canine";
    }

    // Dog inherits from Animal, thus it must override abstract method Greet()
    public override string Greet()
    {
        return "Bark!";
    }
}
```

Base classes can also provide a default implementation using the **virtual** keyword. This allows child classes to choose to either use the base class member or override it with their own implementation.

```
class Animal
{
    public virtual string Greet()
    {
        return "Hello!"; // default
    }
}

class Human : Animal
{
    // Greet() will return Hello!
}

class Dog : Animal
{
    // Dogs bark instead of saying Hello
    public override string Greet()
    {
        return "Bark!";
    }
}
```

If we want to prevent a class from being inherited, we can use the **sealed** keyword, although this is not common since the point of object oriented programming is to extend classes.

```
sealed class Human : Animal
{
// human cannot be inherited from
}
```

What is an interface?

An interface is a group of **function members** that provide no implementation. Like inheritance, interfaces can be added to a class after the colon (:). There are a few key differences between interfaces and inheritance:

1. Interfaces cannot provide default implementation (no shared code)
2. You can **implement** multiple interfaces, but you can only inherit from one class
3. You do not put accessors (public, private, etc) on interface methods

```
// Animal as an interface instead of inheritance
interface Animal
{
    string Type { get; set; }
    string Greet();
}
```

Where does a console application start execution?

The main() method

How do you use backslashes in strings?

You can either prefix the string with the @ sign or **escape** the slash by doubling it up:

```
string s1 = @"This string has a carriage return\n";
string s2 = "This is an escaped slash \\";
```

What is the difference between a reference type and a value type?

Value types are loaded onto the **stack** in memory and reference types are loaded onto the **heap**. When you pass a value type to another method a copy of it is created, and changes do not affect the original. A reference type only the memory **pointer** is passed, so changes will affect all of the pointers since they all point to the same data.

What does static mean?

A static is only loaded into memory one time. It cannot reference instance (non-static) members. Static variables are often used for information that can be shared across many classes (configuration settings, common values like Pi, etc.). Static methods are typically used for general purpose methods (Math.Pow, Factories, Formatters, etc.)

What is the difference between a struct and a class?

A struct is a value type and cannot be inherited. It is stored on the stack and in some cases provides better performance. In most non-high performance applications developers default to using classes.

What is polymorphism?

Polymorphism in C# is when multiple classes through inheritance or interfaces can be used by referencing the type of the base class or interface instead of the **concrete instance**. This allows our code to treat different types the same way as long as they implement or inherit from the same class or interface.

```
interface ICalculateArea
{
    double GetArea();
}

class Square : ICalculateArea
{
    public double Side { get; set; }

    public double GetArea()
    {
        return Side*Side;
    }
}

class Circle : ICalculateArea
{
    public double Radius { get; set; }

    public double GetArea()
    {
        return Math.PI*Radius*Radius;
    }
}

// in our program we can do this to display any shape's area without
// knowing the concrete type (square, circle)
void DisplayArea(ICalculateArea shape)
{
    Console.WriteLine(shape.GetArea());
}
```

What are access modifiers, name them?

Access modifiers determine who can access your types and their members:

1. **Public** – Anyone can access
2. **Private** – Only the class itself can access it
3. **Protected** – Only the class and its children (via inheritance) can access it
4. **Internal** – Only members of the same assembly (project) can access it
5. **Protected Internal** – Only the class's children in the same assembly can access it

What is an array?

An array is a data structure that contains a number of **elements** of the same type. Arrays are declared with the type name followed by square brackets. Once an array receives a size, the size cannot be changed, you must declare a new array with a larger size and copy the elements into the new array.

Arrays have an **index** for accessing individual elements, and it always starts with 0. Arrays can tell you how big they are by using the **length** property, which is the total number of elements. Remember to subtract 1 from length to use indexes!

```
int[] arr1 = new int[2]; // new array with 2 elements
arr1[0] = 5; // set first element (index 0)
arr1[1] = 10; // set second element (index 1)
```

What are the conditional statements in C#?

The **if statement** defines a code block that only executes if a **condition** is met. The condition must always **evaluate** to a **bool** (true or false).

```
int x = 10;

if (x > 5)
{
    Console.WriteLine("x is bigger than 5");
}
else
{
    Console.WriteLine("x is less than 5");
}
```

Conditions can be complex by using **and** && and **or** || modifiers to chain conditions together. In an and condition *all conditions must be true*. In an or condition *only one condition must be true.*

The **switch** statement is the other conditional statement. Switches declare a test variable, and then evaluate multiple **cases** for matches. They are a more efficient way of writing if…else if…else if…else logic. A switch statement may have a **default** case at the end. Multiple case statements can also **fall through** to a code block. We must **break** or **return** after each case block.

```
switch (x)
{
    case 0:
        Console.WriteLine("x is zero");
        break;
    case 1: // fall through
    case 2:
    case 3:
    case 4:
        Console.WriteLine("x is less than 5");
        break;
    default:
        Console.WriteLine("x is 5 or greater");
        break;
}
```

What are loops, name them?

Loops define code blocks that will be executed until a condition is met.

A **for loop** executes a pre-determined number of times, it is most often used when we know exactly how many times it will need to loop (ex: when we know the size of an array, or want to run a process a certain number of times). For loops must declare an **iterator** variable, a **condition**, and specify an **increment** to the iterator.

```
// print 1 to 10
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine(i);
}
```

A **while loop** will only start if the starting condition is true and will continue until a condition is met. This is often used to run batches like processing data of an unknown size until end of file or end of result set is reached.

```
int x = 11;
// print 1 to 10
// x started at 11, this will be skipped
while (x < 10)
{
    Console.WriteLine(x);
    x++;
}
```

A **do…while** loop unlike the while loop is guaranteed to run one time because the condition is at the bottom.

```
int x = 11;
// we started at 11, but it will print 11, increment x, then exit
do
{
    Console.WriteLine(x);
    x++;
} while (x < 10);
```

A **for…each** loop will take any **collection** (arrays, lists, dictionaries, etc) and loop one time for each item in the collection.

```
List<int> ints = new List<int> { 1,2,3,4,5,6,7};

foreach (int i in ints)
{
    Console.WriteLine(i);
}
```

What is a collection?

A collection is a special class for data storage and retrieval that is intended to store multiple items of the same type. There are many types of collections in .NET:

1. **List<T>** - Is a list of items, we can Add() and Remove() items from the list among other things
2. **Stack<T>** - Is a stack of items that are stored in Last-In First-Out. We can Push() and Pop() stack items or Peek() to see the current.
3. **Queue<T>** - Is a group of items stored in First-In First-Out format. We can Enqueue() and Dequeue() items or Peek() to see the current.
4. **Dictionary<key, value>**- A dictionary is a **key value pair**. When we add items to a dictionary, the key must be unique. It is great for keeping counts of duplicate items, etc.
5. **HashTable<key, value>** - Very much like a dictionary, except the key must be a string.

```
// using a dictionary to count duplicates
List<int> ints = new List<int> {10, 10, 10, 20, 30, 30, 40};
Dictionary<int, int> counts = new Dictionary<int, int>();

foreach (int i in ints)
{
    if (counts.ContainsKey(i))
        counts[i] += 1; // already exists, increment count
    else
        counts.Add(i, 1); // add to dictionary, count = 1
}

foreach (int key in counts.Keys)
{
    Console.WriteLine("{0} appeared {1} times", key, counts[key]);
}
```

How do I use code in a different project or assembly?

To use code from a different project or assembly, the targeted code must be public. Provided it is, you can add a reference to it in your current project then place a using statement to the namespace the targeted code is declared in.

For referencing libraries from other developers we can often us **NuGet** to download and reference packages automatically.

How do you handle errors in an application?

If an error is not handled, the program will crash. To prevent crashing we use the try…catch block. We place code that may fail in the try part of the block, and error handling code in the catch block. Optionally we can also have a finally block that will always be executed regardless of errors and is most often used for cleanup (closing files, etc.).

If we put a catch block in a program, we must be sure to log or display the error. If we discard errors without notification or logging this is called “swallowing” errors and it makes applications more difficult

to debug! Typically exception handling will be done in the business layer or the User Interface (sometimes both). It is typically considered bad practice to throw exceptions across boundaries like Services or between a web server and a client.

```
// open file pointer
var sw = File.AppendText("log.txt");

try
{
    sw.WriteLine("Hello!");
    throw new Exception("oops");  // force error
}
catch (Exception ex)
{
   // inform user
   Console.WriteLine("An error occurred: {0}", ex.Message);
}
finally
{
    // cleanup
    sw.Close();
}
```