FullStack.Cafe - Kill Your Tech Interview

Q1: What is C#? ☆

Topics: C#

Answer:

C# is the programming language for writing Microsoft .NET applications. C# provides the rapid application development found in Visual Basic with the power of C++. Its syntax is similar to C++ syntax and meets 100% of the requirements of OOPs like the following:

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Q2: What is an `Object`? ☆

Topics: C#

Answer:

According to MSDN, "*a class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically".*

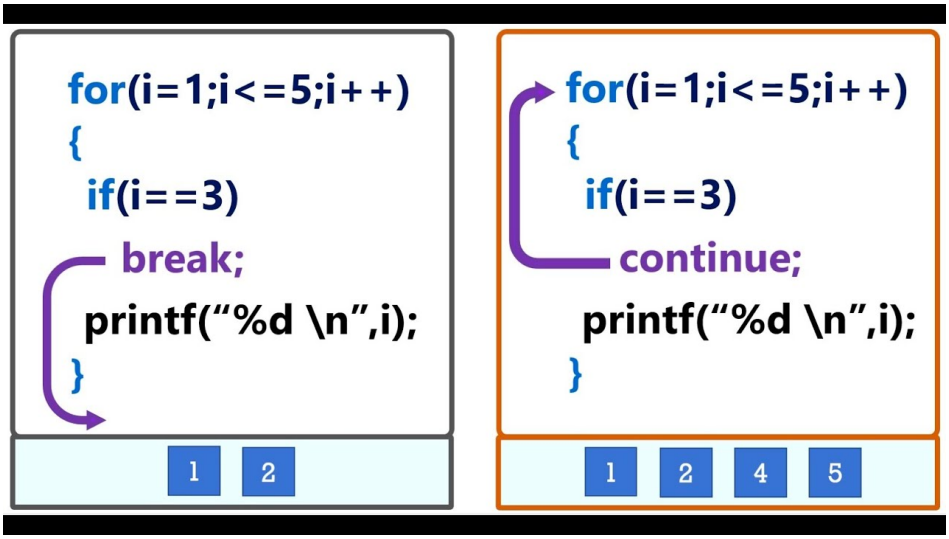
Objects helps us to access the member of a class or struct either they can be fields, methods or properties, by using the dot.

Q3: What is the difference between `continue` and `break` statements in C#? ☆

Topics: C#

Answer:

- using **break** statement, you can **jump out** of a loop
- using **continue** statement, you can **jump over** one iteration and then resume your loop execution



Q4: What are *Property Accessors*? ☆

Topics: C#

Answer:

The **get** and **set** portions or blocks of a property are called **accessors**.

```
class Person
{
  private string name; // field

  public string Name   // property
  {
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

These are useful to restrict the accessibility of a property.

- The **set** accessor specifies that we can assign a value to a private field in a property and without the set accessor property it is like a read-only field.
- By the **get** accessor we can access the value of the private field. A Get accessor specifies that we can access the value of a field publicly.

Q5: What is *Managed* or *Unmanaged* Code? ☆☆

Topics: C#

Answer:

- **Managed Code** - The code, which is developed in .NET framework is known as managed code. This code is directly executed by CLR with the help of managed code execution. Any language that is written in .NET Framework is managed code.
- **Unmanaged Code** - The code, which is developed outside .NET framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications, which, for example, access low - level functions of the

operating system. Background compatibility with the code of VB, ASP and COM are examples of unmanaged code.

Q6: What is the difference between a `Struct` and a `Class` in C#? ☆☆

Topics: C#

Answer:

Class and struct both are the user defined data type but have some major difference:

Struct

- The struct is **value** type in C# and it inherits from `System.Value` Type.
- Struct is usually used for smaller amounts of data.
- Struct can't be inherited to other type.
- A structure can't be abstract.

Class

- The class is **reference** type in C# and it inherits from the `System.Object` Type.
- Classes are usually used for large amounts of data.
- Classes can be inherited to other class.
- A class can be abstract type.
- We can create a default **constructor**.

Q7: Can `this` be used within a `Static` method? ☆☆

Topics: C#

Answer:

We can't use `this` in static method because keyword `this` returns a reference to the **current instance** of the class containing it.

Static methods (or any static member) do not belong to a particular instance. They exist without creating an instance of the class and call with the name of a class not by instance so we can't use this keyword in the body of static Methods, but in case of Extension Methods we can use it as the functions parameters.

Q8: What is the difference between `string` and `StringBuilder` in C#? ☆☆

Topics: C#

Answer:

String

- It's an immutable object that hold string value.
- Performance wise string is slow because its' create a new instance to override or change the previous value.
- `String` belongs to `System` namespace.

StringBuilder

- StringBuilder is a mutable object.
- Performance wise StringBuilder is very fast because it will use same instance of StringBuilder object to perform any operation like insert value in existing string.
- `StringBuilder` belongs to `System.Text.Stringbuilder` namespace.

Q9: What are partial classes? ☆☆

Topics: C#

Answer:

A **partial** class is only use to splits the definition of a class in two or more classes in a same source code file or more than one source files. You can create a class definition in multiple files but it will be compiled as one class at run time and also when you'll create an instance of this class so you can access all the methods from all source file with a same object. Partial classes can be create in the same namespace it's doesn't allowed to create a partial class in different namespace.

Q10: What you understand by *Value types* and *Reference types* in .NET? Provide some comparison. ☆☆

Topics: C# .NET Core

Answer:

In C# data types can be of two types: **Value Types** and **Reference Types**. For a value type, the value is the information itself. For a reference type, the value is a reference which may be null or may be a way of navigating to an object containing the information.

- **Value types** - Holds some value not memory addresses. Example - Struct. A variable's value is stored wherever it is decleared. Local variables live on the stack for example, but when declared inside a class as a member it lives on the heap tightly coupled with the class it is declared in.

- *Advantages:** A value type does not need extra garbage collection. It gets garbage collected together with the instance it lives in. Local variables in methods get cleaned up upon method leave.

- *Drawbacks:**

1. When large set of values are passed to a method the receiving variable actually copies so there are two redundant values in memory.
2. As classes are missed out.it losses all the oop benifits

- **Reference type** - Holds a memory address of a value not value. Example - Class. Stored on heap

- *Advantages:**

1. When you pass a reference variable to a method and it changes it indeed changes the original value whereas in value types a copy of the given variable is taken and that's value is changed.
2. When the size of variable is bigger reference type is good
3. As classes come as a reference type variables, they give reusability, thus benefitting Object-oriented programming

Drawbacks: More work referencing when allocating and dereferences when reading the value.extra overload for garbage collector

Q11: What is *Serialization*? ☆☆

Topics: C#

Answer:

Serialization means** saving the state of your object to secondary memory, such as a file**.

1. Binary serialization (Save your object data into binary format).
2. Soap Serialization (Save your object data into binary format; mainly used in network related communication).
3. XmlSerialization (Save your object data into an XML file).

Q12: What is LINQ in C#? ☆☆

Topics: C# LINQ

Answer:

LINQ stands for Language Integrated Query. LINQ has a great power of querying on any source of data. The data source could be collections of objects, database or XML files. We can easily retrieve data from any object that implements the `IEnumerable<T>` interface.

Q13: Can multiple catch blocks be executed? ☆☆

Topics: C#

Answer:

No, Multiple catch blocks can't be executed. Once the proper catch code executed, the control is transferred to the finally block and then the code that follows the finally block gets executed.

Q14: What are the different types of classes in C#? ☆☆

Topics: C#

Answer:

The different types of class in C# are:

- **Partial class** – Allows its members to be divided or shared with multiple .cs files. It is denoted by the keyword *Partial.*
- **Sealed class** – It is a class which cannot be inherited. To access the members of a sealed class, we need to create the object of the class. It is denoted by the keyword *Sealed*.
- **Abstract class** – It is a class whose object cannot be instantiated. The class can only be inherited. It should contain at least one method. It is denoted by the keyword *abstract.*
- **Static class** – It is a class which does not allow inheritance. The members of the class are also static. It is denoted by the keyword *static*. This keyword tells the compiler to check for any accidental instances of the static class.

Q15: How is *Exception Handling* implemented in C#? ☆☆

Topics: C#

Answer:

Exception handling is done using four keywords in C#:

- **try** – Contains a block of code for which an exception will be checked.
- **catch** – It is a program that catches an exception with the help of exception handler.
- **finally** – It is a block of code written to execute regardless whether an exception is caught or not.
- **throw** – Throws an exception when a problem occurs.

Q16: What is an *Abstract Class*? ☆☆

Topics: C#

Answer:

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events.

An **Abstract class** is a class which is denoted by `abstract` keyword and can be used only as a **Base** class.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings. The `sealed` modifier prevents a class from being inherited and the `abstract` modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Q17: What is *namespace* in C#? ☆☆

Topics: C#

Answer:

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

- NET uses namespaces to organize its many classes.
- Declaring your own namespaces can help you control the scope of class and method names in larger programming projects.

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

Q18: What are *Reference Types* in C#? ☆☆

Topics: C#

Answer:

The **reference types** do not contain the actual data stored in a variable, but they contain a *reference* to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of built-in reference types are: object, dynamic, and string.

Q19: What are *Nullable types* in C#? ☆☆

Topics: C#

Answer:

A type is said to be **nullable** if it can be assigned a value or can be assigned `null`, which means the type has no value whatsoever. By default, all reference types, such as String, are nullable, but all value types, such as Int32, are not.

For example, you can store any value from `-2,147,483,648` to `2,147,483,647` or `null` in a `Nullable<Int32>` variable. Similarly, you can assign `true`, `false`, or `null` in a `Nullable<bool>` variable.

Q20: Why to use `finally` block in C#? ☆☆

Topics: C#

Answer:

Finally block will be executed *irrespective of exception*. So while executing the code in try block when exception is occurred, control is returned to catch block and at last `finally` block will be executed. So closing connection

to database / releasing the file handlers can be kept in `finally` block.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Boxing* and *Unboxing*? ☆☆

Topics: C# .NET Core

Answer:

Boxing and **Unboxing** both are used for type conversion but have some difference:

- **Boxing** - Boxing is the process of converting a value type data type to the object or to any interface data type which is implemented by this value type. When the CLR boxes a value means when CLR is converting a value type to Object Type, it wraps the value inside a `System.Object` and stores it on the heap area in application domain.

- **Unboxing** - Unboxing is also a process which is used to extract the value type from the object or any implemented interface type. Boxing may be done implicitly, but unboxing have to be explicit by code.

The concept of boxing and unboxing underlines the C# unified view of the type system in which a value of any type can be treated as an object.

Q2: What is enum in C#? ☆☆

Topics: C#

Answer:

An **enum** is a value type with a set of related named constants often referred to as an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type, which is user defined. An enum is used to create numeric constants in .NET framework. All the members of enum are of enum type. Their must be a numeric value for each enum type.

Some points about enum

- Enums are enumerated data type in C#.
- Enums are strongly typed constant. They are strongly typed, i.e. an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
- Enumerations (enums) make your code much more readable and understandable.
- Enum values are fixed. Enum can be displayed as a string and processed as an integer.
- The default type is int, and the approved types are byte, sbyte, short, ushort, uint, long, and ulong.
- Every enum type automatically derives from System.Enum and thus we can use System.Enum methods on enums.
- Enums are value types and are created on the stack and not on the heap.

Q3: What are *generics* in C#? ☆☆

Topics: C#

Answer:

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics **allow you to write a class or method that can work with any data type**.

Consider:

```
class DataStore<T>
{
    public T Data { get; set; }
}

DataStore<string> store = new DataStore<string>();
```

Q4: In how many ways you can pass parameters to a method? ☆☆

Topics: C#

Answer:

There are three ways that parameters can be passed to a method:

- **Value parameters** − This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- **Reference parameters** − This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
- **Output parameters** − This method helps in returning more than one value.

Q5: What are dynamic type variables in C#? ☆☆

Topics: C#

Answer:

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Q6: What is the difference between *Interface* and *Abstract Class*? ☆☆☆

Topics: C# OOP

Answer:

There are some differences between **Abstract Class** and **Interface** which are listed below:

- interfaces can have no state or implementation
- a class that implements an interface must provide an implementation of all the methods of that interface
- abstract classes may contain state (data members) and/or implementation (methods)
- abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
- interfaces may be multiple-inherited, abstract classes may not (this is probably the key concrete reason for interfaces to exist separately from abtract classes - they permit an implementation of multiple inheritance that removes many of the problems of general MI).

Consider using abstract classes if :

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

Consider using interfaces if :

1. You expect that unrelated classes would implement your interface. For example,many unrelated objects can implement `Serializable` interface.
2. You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

Q7: What is the difference between `ref` and `out` keywords? ☆☆☆

Topics: C#

Answer:

- `ref` tells the compiler that the object is initialized before entering the function, while
- `out` tells the compiler that the object will be initialized inside the function.

So while `ref` is two-ways, `out` is out-only.

Q8: What is *Extension Method* in C# and how to use them? ☆☆☆

Topics: C#

Answer:

Extension methods enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. An extension method is a special kind of static method, but they are called as if they were instance methods on the extended type.

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Q9: Is there a difference between `throw` and `throw ex`? ☆☆☆

Topics: C#

Answer:

Yes, there is a difference:

- When you do `throw ex`, that thrown exception becomes the "original" one. So all previous stack trace will not be there.
- If you do `throw`, the exception just goes *down the line* and you'll get the full stack trace.

```
static void Main(string[] args)
{
    try
    {
        Method2();
    }
    catch (Exception ex)
    {
        Console.Write(ex.StackTrace.ToString());
        Console.ReadKey();
    }
}

private static void Method2()
{
    try
    {
        Method1();
    }
    catch (Exception ex)
    {
        //throw ex resets the stack trace Coming from Method 1 and propogates it to the caller(Main)
        throw ex;
    }
}

private static void Method1()
{
    try
    {
        throw new Exception("Inside Method1");
    }
    catch (Exception)
    {
        throw;
    }
}
```

Q10: What is the difference between Equality Operator (`==`) and `Equals()` Method in C#? ☆☆☆

Topics: C#

Answer:

The `==` Operator (usually means the same as `ReferenceEquals`, could be overrided) compares the reference identity while the `Equals()` (`virtual Equals()`) method compares if two objects are equivalent.

Q11: What is *Virtual Method* in C#? ☆☆☆

Topics: C#

Answer:

A **virtual method** is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived class. When a virtual method is invoked, the run-time type of the object is checked for an overriding member.

```
public virtual double Area()
{
    return x * y; // The implementation of a virtual member can be changed by an overriding member in a
derived class.
}
```

1. By default, methods are non-virtual. We can't override a non-virtual method.
2. We can't use the virtual modifier with the static, abstract, private or override modifiers.

Q12: What are the uses of `using` in C# ☆☆☆

Topics: C#

Answer:

The reason for the `using` statement is to ensure that the **object is disposed** (call `IDisposable`) as soon as it goes out of scope, and it doesn't require explicit code to ensure that this happens.

The .NET CLR converts:

```
using (MyResource myRes = new MyResource())
{
    myRes.DoSomething();
}
```

to:

```
{ // Limits scope of myRes
    MyResource myRes= new MyResource();
    try
    {
        myRes.DoSomething();
    }
    finally
    {
        // Check for a null resource.
        if (myRes != null)
            // Call the object's Dispose method.
            ((IDisposable)myRes).Dispose();
    }
}
```

Q13: Explain *Anonymous type* in C# ☆☆☆

Topics: C#

Answer:

Anonymous types allow us to create a new type without defining them. This is way to defining read only properties into a single object without having to define type explicitly. Here Type is generating by the compiler and it is accessible only for the current block of code. The type of properties is also inferred by the compiler.

Consider:

```
var anonymousData = new
{
     ForeName = "Jignesh",
     SurName = "Trivedi"
};

Console.WriteLine("First Name : " + anonymousData.ForeName);
```

Q14: What is difference between `constant` and `readonly`? ☆☆☆

Topics: C#

Answer:

Apart from the apparent difference of

- having to declare the value at the time of a definition for a `const` VS `readonly` values can be computed dynamically but need to be assigned before the constructor exits.. after that it is frozen.
- 'const's are implicitly `static`. You use a `ClassName.ConstantName` notation to access them.

There is a subtle difference. Consider a class defined in `AssemblyA`.

```
public class Const_V_Readonly
{
  public const int I_CONST_VALUE = 2;
  public readonly int I_RO_VALUE;
  public Const_V_Readonly()
  {
     I_RO_VALUE = 3;
  }
}
```

`AssemblyB` references `AssemblyA` and uses these values in code. When this is compiled,

- in the case of the `const` value, it is like a find-replace, the value 2 is 'baked into' the `AssemblyB`'s IL. This means that if tomorrow I'll update `I_CONST_VALUE` to 20 in the future. *`AssemblyB` would still have 2 till I recompile it*.
- in the case of the `readonly` value, it is like a `ref` to a memory location. The value is not baked into `AssemblyB`'s IL. This means that if the memory location is updated, `AssemblyB` gets the new value without recompilation. So if `I_RO_VALUE` is updated to 30, you only need to build `AssemblyA`. All clients do not need to be recompiled.

Remember: If you reference a constant from another assembly, its value will be compiled right into the calling assembly. That way when you update the constant in the referenced assembly it won't change in the calling assembly!

Q15: Why can't you specify the accessibility modifier for methods inside the Interface? ☆☆☆

Topics: C#

Answer:

In an interface, we have **virtual methods** that do not have method definition. All the methods are there to be **overridden in the derived class**. That's why they all are **public**.

Q16: Explain *Code Compilation* in C# ☆☆☆

Topics: C#

Answer:

There are four steps in code compilation which include:

- Compiling the source code into Managed code by C# compiler.
- Combining the newly created code into assemblies.
- Loading the Common Language Runtime(CLR).
- Executing the assembly by CLR.

Q17: What is the difference between *Virtual* method and *Abstract* method? ☆☆☆

Topics: C# OOP

Answer:

- A **Virtual method** must always have a default implementation. However, it can be overridden in the derived class, though not mandatory. It can be overridden using *override* keyword.
- An **Abstract method** does not have an implementation. It resides in the abstract class. It is mandatory that the derived class implements the abstract method. An *override* keyword is not necessary here though it can be used.

Q18: What is the difference between *dynamic type* variables and *object type* variables? ☆☆☆

Topics: C#

Answer:

- The object type is an alias for `System.Object` in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from `System.Object`. You can assign values of any type to variables of type object.

- The dynamic type indicates that use of the variable and references to its members **bypass compile-time type checking**. Instead, these operations are resolved at run time. The dynamic type simplifies access to COM APIs such as the Office Automation APIs, to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

- Type dynamic behaves like type object in most circumstances. In particular, any non-null expression can be converted to the dynamic type. The dynamic type differs from object in that operations that contain expressions of type dynamic are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type dynamic are compiled into variables of type object. Therefore, type dynamic exists only at compile time, not at run time.

Assume we have the following method:

```
public static void ConsoleWrite(string inputArg)
{
    Console.WriteLine(inputArg);
}
```

Object: the following code has compile error unless cast object to string:

```
public static void Main(string[] args)
{
    object obj = "String Sample";
    ConsoleWrite(obj);// compile error
    ConsoleWrite((string)obj); // correct
    Console.ReadKey();
}
```

dynamic: the following code compiles successfully but if it contains a value except `string` it throws Runtime error

```
public static void Main(string[] args)
{
    dynamic dyn = "String Sample";
    ConsoleWrite(dyn); // correct
    dyn = 1;
    ConsoleWrite(dyn);// Runtime Error
    Console.ReadKey();
}
```

Q19: What is scope of a *Internal* member variable of a C# class? ☆☆☆

Topics: C#

Answer:

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the *current assembly*. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

You can use it for utility or helper classes/methods that you would like to access from many other classes within the same assembly, but that you want to ensure code in other assemblies can't access.

Q20: What is an *anonymous function* in C#? ☆☆☆

Topics: C#

Answer:

An **anonymous function** is an "inline" statement or expression that can be used wherever a delegate type is expected. You can use it to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

There are two kinds of anonymous functions:

- Lambda Expressions
- Anonymous Methods

Consider:

```
// Original delegate syntax required
// initialization with a named method.
TestDelegate testDelA = new TestDelegate(M);

// C# 2.0: A delegate can be initialized with
// inline code, called an "anonymous method." This
// method takes a string as an input parameter.
TestDelegate testDelB = delegate(string s) {
 Console.WriteLine(s);
};

// C# 3.0. A delegate can be initialized with
// a lambda expression. The lambda also takes a string
// as an input parameter (x). The type of x is inferred by the compiler.
TestDelegate testDelC = (x) => {
 Console.WriteLine(x);
};

// Invoke the delegates.
testDelA("Hello. My name is M and I write lines.");
testDelB("That's nothing. I'm anonymous and ");
testDelC("I'm a famous author.");
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Is there a way to catch *multiple* exceptions at once and without code duplication? ☆☆☆

Topics: .NET Core C#

Problem:

Consider:

```
try
{
    WebId = new Guid(queryString["web"]);
}
catch (FormatException)
{
    WebId = Guid.Empty;
}
catch (OverflowException)
{
    WebId = Guid.Empty;
}
```

Is there a way to catch both exceptions and only call the `WebId = Guid.Empty` call once?

Solution:

Catch `System.Exception` and switch on the types:

```
catch (Exception ex)
{
    if (ex is FormatException || ex is OverflowException)
    {
        WebId = Guid.Empty;
        return;
    }

    throw;
}
```

Q2: Why to use of the `IDisposable` interface? ☆☆☆

Topics: .NET Core C#

Answer:

The "primary" use of the `IDisposable` interface is **to clean up unmanaged resource**s. Note the purpose of the Dispose pattern is to provide a mechanism to clean up both *managed* and *unmanaged* resources and when that occurs depends on how the Dispose method is being called.

Q3: Explain the difference between `Task` and `Thread` in .NET ☆☆☆

Topics: .NET Core C#

Answer:

- **Thread** represents an actual OS-level thread, with its own stack and kernel resources. Thread allows the highest degree of control; you can `Abort()` or `Suspend()` or `Resume()` a thread, you can observe its state, and you can set thread-level properties like the stack size, apartment state, or culture. `ThreadPool` is a wrapper around a pool of threads maintained by the CLR.

- The **Task class** from the Task Parallel Library offers the best of both worlds. Like the `ThreadPool`, a task does not create its own OS thread. Instead, tasks are executed by a `TaskScheduler`; the default scheduler simply runs on the ThreadPool. Unlike the ThreadPool, Task also allows you to find out when it finishes, and (via the generic Task) to return a result.

Q4: What is `sealed` Class in C#? ☆☆☆

Topics: C#

Answer:

- Once a class is defined as a `sealed` class, the class cannot be inherited.
- **Structs** are also `sealed`.

Q5: What is the difference between *overloading* and *overriding*? ☆☆☆

Topics: C#

Answer:

- **Overloading** is when you have multiple methods in the same scope, with the same name but different signatures.

```
//Overloading
public class test
{
    public void getStuff(int id)
    {}
    public void getStuff(string name)
    {}
}
```

- **Overriding** is a principle that allows you to change the functionality of a method in a child class.

```
//Overriding
public class test
{
        public virtual void getStuff(int id)
        {
            //Get stuff default location
        }
}

public class test2 : test
{
        public override void getStuff(int id)
        {
```

```
            //base.getStuff(id);
            //or - Get stuff new location
        }
}
```

Q6: What is *Reflection* in C#.Net? ☆☆☆

Topics: C#

Answer:

Reflection is the ability to query and interact with the type system in a dynamic way. Generally speaking Reflection allows you access to metadata about objects. For instance you can load a DLL and determine if it contains an implementation of an interface. You could use this to discover dll's that support functionality at runtime. Use could use this to extend an application without a recompilation and without having to restart it.

Q7: What is a *Destructor* in C# and when shall I create one? ☆☆☆

Topics: C#

Answer:

A **Destructor** is used to clean up the memory and free the resources. But in C# this is done by the garbage collector on its own. `System.GC.Collect()` is called internally for cleaning up. The answer to second question is "almost never".

Typically one only creates a destructor when your class is holding on to some expensive unmanaged resource that must be cleaned up when the object goes away. It is better to use the **disposable pattern** to ensure that the resource is cleaned up. A destructor is then essentially an assurance that if the consumer of your object forgets to dispose it, the resource still gets cleaned up eventually.

If you make a destructor *be extremely careful* and *understand how the garbage collector works*. Destructors are *really weird*:

- They don't run on your thread; they run on their own thread. Don't cause deadlocks!
- An unhandled exception thrown from a destructor is bad news. It's on its own thread; who is going to catch it?
- A destructor may be called on an object *after* the constructor starts but *before* the constructor finishes. A properly written destructor will not rely on invariants established in the constructor.
- A destructor can "resurrect" an object, making a dead object alive again. That's really weird. Don't do it.
- A destructor might never run; you can't rely on the object ever being scheduled for finalization. It *probably* will be, but that's not a guarantee.

Q8: How *encapsulation* is implemented in C#? ☆☆☆

Topics: C#

Answer:

Encapsulation is implemented by using **access specifiers**. An access specifier defines the scope and visibility of a class member.

- **Public** access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

- **Private** access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.
- **Protected** access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance.

Q9: What is the use of Null Coalescing Operator (`??`) in C#? ☆☆☆

Topics: C#

Answer:

The **??** operator is called the **null-coalescing operator** and is used to define a default value for nullable value types or reference types. It returns the left-hand operand if the operand is not null; otherwise, it returns the right operand.

```
string name = null;
string myname = name ?? "Laxmi";
Console.WriteLine(myname);
```

Q10: What is *lambda expressions* in C#? ☆☆☆

Topics: C#

Answer:

A **lambda expression** is an anonymous function that you can use to create delegates or *expression tree* types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

In the following example, the lambda expression `x => x * x`, which specifies a parameter that's named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Q11: How can you prevent a class from *overriding* in C#? ☆☆☆

Topics: OOP C#

Answer:

You can prevent a class from overriding in C# by using the `sealed` keyword.

Q12: What is difference between *late binding* and *early binding* in C#? ☆☆☆☆

Topics: C#

Answer:

- In **Compile time polymorphism** or **Early Binding** we will use multiple methods with same name but different type of parameter or may be the number or parameter because of this we can perform different-different tasks with same method name in the same class which is also known as Method **overloading**.
- **Run time polymorphism** also known as **late binding**, in Run Time polymorphism or Late Binding we can do use same method names with same signatures means same type or same number of parameters but not in same class because compiler doesn't allowed that at compile time so we can use in derived class that bind at run time when a child class or derived class object will instantiated that's way we says that Late Binding also known as Method **overriding**.

Q13: What is the difference between `is` and `as` operators in C#? ☆☆☆☆

Topics: C#

Answer:

- The `is` operator checks if an object can be cast to a specific type.

```
if (someObject is StringBuilder) ...
```

- The `as` operator attempts to cast an object to a specific type, and returns null if it fails.

```
StringBuilder b = someObject as StringBuilder;
if (b != null) ...
```

Q14: What are pointer types in C#? ☆☆☆☆

Topics: C#

Answer:

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

```
char* cptr;  int* iptr;
```

Q15: What is scope of a Protected Internal member variable of a C# class? ☆☆☆☆

Topics: C#

Answer:

The **protected internal** access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

Q16: Can you create a function in C# which can accept varying number of arguments? ☆☆☆☆

Topics: C#

Answer:

By using the `params` keyword, you can specify a method parameter that takes a variable number of arguments. No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration. The declared type of the params parameter must be a single-dimensional array.

```
public static void UseParams(params int[] list) {
 for (int i = 0; i < list.Length; i++) {
  Console.Write(list[i] + " ");
 }
 Console.WriteLine();
}

public static void UseParams2(params object[] list) {
 for (int i = 0; i < list.Length; i++) {
  Console.Write(list[i] + " ");
 }
 Console.WriteLine();
}

// usage
UseParams(1, 2, 3, 4);
UseParams2(1, 'a', "test");
```

Q17: Is *operator overloading* supported in C#? ☆☆☆☆

Topics: C#

Answer:

A user-defined type can overload a predefined C# operator. That is, a type can provide the custom implementation of an operation in case one or both of the operands are of that type.

```
public static Box operator+ (Box b, Box c) {
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

The above function implements the addition operator (`+`) for a user-defined class `Box`. It adds the attributes of two Box objects and returns the resultant Box object.

Q18: Can *Multiple Inheritance* implemented in C# ? ☆☆☆☆

Topics: C#

Answer:

In C#, derived classes can inherit from one base class only. If you want to inherit from multiple base classes, use interface.

Q19: Test if a Number belongs to the Fibonacci Series ☆☆☆☆

Topics: Fibonacci Series C#

Answer:

A positive integer `ω` is a Fibonacci number if and only one of:

- $5\omega^2 + 4$ and
- $5\omega^2 - 4$

is a *perfect square* or a *square number*.

A **square number** is the result when a number has been multiplied by itself.

Note: it involves squaring a Fibonacci number, frequently resulting in a massive product. There are less than 80 Fibonacci numbers that can even be held in a standard 64-bit integer.

Implementation:

CS

```
bool isFibonacci(int  w)
{
       double X1 = 5 * Math.Pow(w, 2) + 4;
       double X2 = 5 * Math.Pow(w, 2) - 4;

       long X1_sqrt = (long)Math.Sqrt(X1);
       long X2_sqrt = (long)Math.Sqrt(X2);

       return (X1_sqrt*X1_sqrt == X1) || (X2_sqrt*X2_sqrt == X2) ;
}
```

PY

```
from math import *

phi = 1.61803399
sqrt5 = sqrt(5)

def F(n):
    return int((phi**n - (1-phi)**n) /sqrt5)

def isFibonacci(z):
    return F(int(floor(log(sqrt5*z,phi)+0.5))) == z
```

Q20: Explain what is Ternary Search? ☆☆☆☆

Topics: Searching C#

Answer:

Like linear search and binary search, **ternary search** is a searching technique that is used to determine the position of a specific value in an array. In binary search, the sorted array is divided into two parts while in ternary search, it is divided into 3 parts and then you determine in which part the element exists.

Ternary search, like binary search, is a divide-and-conquer algorithm. It is mandatory for the array (in which you will search for an element) to be sorted before you begin the search. In a ternary search, there is a possibility (33.33% chance, actually) that you can eliminate `2/3` of the list, but there is an even greater chance (66.66%) that you will only eliminate `1/3` of the list.

Ternary search has a lesser number of *iterations* than Binary Search but Ternary search also leads to more *comparisons*.

Average number of comparisons:

```
in ternary search = ((1/3)*1 + (2/3)*2) * ln(n)/ln(3) ~ 1.517*ln(n)
in binary search  =                   1 * ln(n)/ln(2) ~ 1.443*ln(n).
```

Worst number of comparisons:

```
in ternary search = 2 * ln(n)/ln(3) ~ 1.820*ln(n)
in binary search  = 1 * ln(n)/ln(2) ~ 1.443*ln(n).
```

Implementation:

cs

```
int ternary_search(int l,int r, int x)
{
    if(r>=l)
    {
        int mid1 = l + (r-l)/3;
        int mid2 = r -  (r-l)/3;
        if(ar[mid1] == x)
            return mid1;
        if(ar[mid2] == x)
            return mid2;
        if(x<ar[mid1])
            return ternary_search(l,mid1-1,x);
        else if(x>ar[mid2])
            return ternary_search(mid2+1,r,x);
        else
            return ternary_search(mid1+1,mid2-1,x);

    }
    return -1;
}
```

Java

```
public static int ternarySearch(int[] A, int x)
{
    int left = 0, right = A.length\ - 1;
    while (left <= right)
    {
        int leftMid = left + (right - left) / 3;
        int rightMid = right - (right - left) / 3;
        // int leftMid = (2*left + right) / 3;
        // int rightMid = (left + 2*right) / 3;
        if (A[leftMid] == x) {
```

```
            return leftMid;
        } else if (A[rightMid] == x) {
            return rightMid;
        } else if (A[leftMid] > x) {
            right = leftMid - 1;
        } else if (A[rightMid] < x) {
            left = rightMid + 1;
        } else {
            left = leftMid + 1;
            right = rightMid - 1;
        }

    }

    return -1;
}
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Explain how does Asynchronous tasks `Async/Await` work in .NET? ☆☆☆☆

Topics: .NET Core C#

Problem:

Consider:

```
private async Task<bool> TestFunction()
{
  var x = await DoesSomethingExists();
  var y = await DoesSomethingElseExists();
  return y;
}
```

Does the second `await` statement get executed right away or after the first `await` returns?

Solution:

`await` pauses *the method* until the operation completes. So the second `await` would get executed after the first `await` returns.

The purpose of `await` is that it will return the current thread to the thread pool while the awaited operation runs off and does whatever.

This is particularly useful in high-performance environments, say a web server, where a given request is processed on a given thread from the overall thread pool. If we don't await, then the given thread processing the request (and all it's resources) remains "in use" while the db / service call completes. This might take a couple of seconds or more especially for external service calls.

Q2: What is *Marshalling* and why do we need it? ☆☆☆☆

Topics: C#

Answer:

Because different languages and environments have different calling conventions, different layout conventions, different sizes of primitives (cf. `char` in C# and `char` in C), different object creation/destruction conventions, and different design guidelines. You need a way to get the stuff out of managed land an into somewhere where unmanaged land can see and understand it and vice versa. That's what **marshalling** is for.

Marshaling is the process between managed code and unmanaged code; It is one of the most important services offered by the **CLR**.

Q3: What is the difference between `dispose` and `finalize` methods in C#? ☆☆☆☆

Topics: C#

Answer:

Finalizer and Dispose both are used for same task like to free unmanaged resources but have some differences see.

Finalize:

- Finalize used to free unmanaged resources those are not in use like files, database connections in application domain and more, held by an object before that object is destroyed.
- In the Internal process it is called by Garbage Collector and can't called manual by user code or any service.
- Finalize belongs to System.Object class.
- Implement it when you have unmanaged resources in your code, and make sure that these resources are freed when the Garbage collection happens.

Dispose:

- Dispose is also used to free unmanaged resources those are not in use like files, database connections in Application domain at any time.
- Dispose explicitly it is called by manual user code.
- If we need to dispose method so must implement that class by IDisposable interface.
- It belongs to IDisposable interface.
- Implement this when you are writing a custom class that will be used by other users.

Q4: What is the *Constructor Chaining* in C#? ☆☆☆☆

Topics: C#

Answer:

Constructor Chaining is an approach where a constructor calls another constructor in the same or base class. This is very handy when we have a class that defines multiple constructors.

Consider:

```
class Foo {
    private int id;
    private string name;

    public Foo() : this(0, "") {
    }

    public Foo(int id) : this(id, "") {
    }

    public Foo(string name) : this(0, name) {
    }

    public Foo(int id, string name) {
        this.id = id;
        this.name = name;
    }
}
```

Q5: What is *Indexer* in C#? ☆☆☆☆

Topics: C#

Answer:

Indexers are a syntactic convenience that enable you to create a class, struct, or interface that client applications can access as an array.

To declare an indexer on a class or struct, use the `this` keyword:

```
// Indexer declaration
public int this[int index] {
// get and set accessors
}
```

You can then access the instance of this class using the array access operator `[ ]`. Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

Consider:

```
class IndexedNames {
    private string[] namelist = new string[size];

    public string this[int index] {
        get {
            string tmp;

            if (index >= 0 && index <= size - 1) {
                tmp = namelist[index];
            } else {
                tmp = "";
            }

            return (tmp);
        }
        set {
            if (index >= 0 && index <= size - 1) {
                namelist[index] = value;
            }
        }
    }
}
```

Q6: When to use `ArrayList` over `array[]` in C#? ☆☆☆☆

Topics: C#

Answer:

- **Arrays** are strongly typed, and work well as parameters. If you know the length of your collection and it is fixed, you should use an array.
- **ArrayLists** are not strongly typed, every Insertion or Retrial will need a cast to get back to your original type. If you need a method to take a list of a specific type, ArrayLists fall short because you could pass in an ArrayList containing any type. ArrayLists use a dynamically expanding array internally, so there is also a hit to expand the size of the internal array when it hits its capacity.

What you really want to use is a generic list like List. This has all the advantages of Array and ArrayLists. It is strongly typed and it supports a variable length of items.

Q7: What are the different ways a method can be overloaded? ☆☆☆☆

Topics: C#

Answer:

Methods can be overloaded using different data types for parameter, different order of parameters, and different number of parameters.

Q8: What is the use of conditional preprocessor directive in C#? ☆☆☆☆

Topics: C#

Answer:

You can use the `#if` directive to create a conditional directive. Conditional directives are useful for testing a symbol or symbols to check if they evaluate to true. If they do evaluate to true, the compiler evaluates all the code between the `#if` and the next directive.

Q9: What is the difference between `System.ApplicationException` class and `System.SystemException` class? ☆☆☆☆

Topics: C#

Answer:

- The `System.ApplicationException` class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.
- The `System.SystemException` class is the base class for all predefined system exception.

Q10: What's the difference between `StackOverflowError` and `OutOfMemoryError`? ☆☆☆☆

Topics: C#

Answer:

- **OutOfMemoryError** is related to Heap. To avoid: Make sure un-necessary objects are available for GC
- **StackOverflowError** is related to stack. To avoid: Make sure method calls are ending (not in an infinite loop)

Q11: When would you use delegates in C#? ☆☆☆☆

Topics: C#

Answer:

Now that we have lambda expressions and anonymous methods in C#, I use delegates much more. In C# 1, where you always had to have a separate method to implement the logic, using a delegate often didn't make sense. These days I use delegates for:

- Event handlers (for GUI and more)
- Starting threads
- Callbacks (e.g. for async APIs)
- LINQ and similar (List.Find etc)
- Anywhere else where I want to effectively apply "template" code with some specialized logic inside (where the delegate provides the specialization)

Q12: Why to use `lock` statement in C#? ☆☆☆☆

Topics: C#

Answer:

The `lock` keyword ensures that one thread does not enter a critical section of code while another thread is in the critical section. If another thread tries to enter a locked code, it will wait, block, until the object is released.

The `lock` keyword calls `Enter` at the start of the block and `Exit` at the end of the block. `lock` keyword actually handles `Monitor` class at back end.

For example:

```
private static readonly Object obj = new Object();

lock (obj)
{
    // critical section
}
```

Q13: What is the `yield` keyword used for in C#? ☆☆☆☆

Topics: C#

Problem:

Consider:

```
IEnumerable<object> FilteredList()
{
    foreach( object item in FullList )
    {
        if( IsItemInPartialList( item )
            yield return item;
    }
}
```

Could you explain what does the `yield` keyword do there?

Solution:

The function returns an object that implements the `IEnumerable` interface. If a calling function starts foreach-ing over this object the function is called again until it "yields". This is syntactic sugar introduced in C# 2.0.

`Yield` has two great uses:

- It helps to provide custom iteration without creating temp collections.
- It helps to do stateful iteration.

The advantage of using `yield` is that if the function consuming your data simply needs the first item of the collection, the rest of the items won't be created.

Another example:

```
public void Consumer()
{
    foreach(int i in Integers())
    {
        Console.WriteLine(i.ToString());
    }
}

public IEnumerable<int> Integers()
{
    yield return 1;
    yield return 2;
    yield return 4;
    yield return 8;
    yield return 16;
    yield return 16777216;
}
```

When you step through the example you'll find the first call to Integers() returns 1. The second call returns 2 and the line "yield return 1" is not executed again.

Q14: What interface should your data structure implement to make the `Where` method work? ☆☆☆☆

Topics: C#

Answer:

Implementing `IEnumerable` makes using `foreach` and `where` *possible*.

Q15: `IEnumerable` vs `List` - What to Use? How do they work? ☆☆☆☆

Topics: C#

Answer:

`IEnumerable` describes behavior, while List is an implementation of that behavior. When you use `IEnumerable`, you give the compiler a chance to defer work until later, possibly optimizing along the way. If you use `ToList()` you force the compiler to reify the results right away.

Whenever you "stacking" LINQ expressions, you should use `IEnumerable`, because by only specifying the behavior gives LINQ a chance to defer evaluation and possibly optimize the program.

Q16: What is the difference between `Func<string,string>` and `delegate`? ☆☆☆☆

Topics: C#

Problem:

Consider:

```
Func<string, string> convertMethod = lambda; // A
public delegate string convertMethod(string value); // B
```

Are they both delegates? What's the difference?

Solution:

The first is declaring a generic delegate variable and assigning a value to it, the second is just defining a delegate type. Both `Func<string,string>` and `delegate string convertMethod(string)` would be capable of holding the same method definitions whether they be methods, anonymous methods, or lambda expressions.

Consider:

```
public static class Program
{
    // you can define your own delegate for a nice meaningful name, but the
    // generic delegates (Func, Action, Predicate) are all defined already
    public delegate string ConvertedMethod(string value);

    public static void Main()
    {
        // both work fine for taking methods, lambdas, etc.
        Func<string, string> convertedMethod = s => s + ", Hello!";
        ConvertedMethod convertedMethod2 = s => s + ", Hello!";
    }
}
```

Q17: Explain the difference between `Select` and `Where` ☆☆☆☆

Topics: C# LINQ

Problem:

Consider:

```
ContextSet().Select(x=> x.FirstName == "John") // A
ContextSet().Where(x=> x.FirstName == "John") // B
```

When should I use `.Select` vs `.Where`?

Solution:

- **Select** is a projection, so what you get is the expression `x=> x.FirstName == "John"` evaluated for each element in ContextSet() on the server. i.e. lots of true/false values (the same number as your original list). If you look the select will return something like `IEnumerable<bool>` (because the type of `x=> x.FirstName == "John"` is a bool). Use `Select` when you want to keep all results, but change their type (project them).
- **Where** filters the results, returning an enumerable of the original type (no projection). Use `Where` when you want to filter your results, keeping the original type

Q18: Explain what is *Short-Circuit Evaluation* in C# ☆☆☆☆

Topics: C#

Answer:

Short-circuit evaluation is a tricky method for evaluating logical operators AND and OR. In this method, the whole expression can be evaluated to `true` or `false` without evaluating all sub expressions. Usually short-circuiting evaluates the right side only if the left side doesn't already determine the result.

Consider:

```
if(myObj != null && myObj.SomeString != null)
```

Although C# (and some other .NET languages) behave this way, it is a property of the language, not the CLR.

Q19: What is the best practice to have best performance using Lazy objects? ☆☆☆☆

Topics: C#

Answer:

You typically use it when you want to instantiate something the first time its actually used. This delays the cost of creating it till if/when it's needed instead of always incurring the cost.

Usually this is preferable when the object may or may not be used and *the cost of constructing it is non-trivial*.

For example `Lazy<T>` makes implementing lazy, thread-safe **singletons** easy:

```
public sealed class Singleton
{
    // Because Singleton's constructor is private, we must explicitly
    // give the Lazy<Singleton> a delegate for creating the Singleton.
    static readonly Lazy<Singleton> instanceHolder =
        new Lazy<Singleton>(() => new Singleton());

    Singleton()
    {
        // Explicit private constructor to prevent default public constructor.
        ...
    }

    public static Singleton Instance => instanceHolder.Value;
}
```

Q20: What happens when we *Box* or *Unbox* Nullable types? ☆☆☆☆

Topics: C#

Answer:

- When a nullable type is boxed, the common language runtime automatically boxes the underlying value of the Nullable object, not the Nullable object itself. That is, if the HasValue property is `true`, the contents of the Value property is boxed. If the `HasValue` property is `false`, `null` is boxed.

- When the underlying value of a nullable type is unboxed, the common language runtime creates a new Nullable structure initialized to the underlying value.

FullStack.Cafe - Kill Your Tech Interview

Q1: Can you explain the difference between `Interface`, `abstract` class, `sealed` class, `static` class and `partial` class in C#? ☆☆☆☆

Topics: C#

Answer:

- **abstract class** Should be used when there is a IS-A relationship and no instances should be allowed to be created from that abstract class. Example: An Animal is an abstract base class where specific animals can be derived from, i.e. Horse, Pig etc. By making Animal abstract it is not allowed to create an Animal instance.

- **interface** An interface should be used to implement functionality in a class. Suppose we want a horse to be able to Jump, an interface IJumping can be created. By adding this interface to Horse, all methods in IJumping should be implemented. In IJumping itself only the declarations (e.g. StartJump and EndJump are defined), in Horse the implementations of these two methods should be added.

- **sealed class** By making Horse sealed, it is not possible to inherit from it, e.g. making classes like Pony or WorkHorse which you like to be inheriting from Horse.

- **static class** Mostly used for 'utility' functions. Suppose you need some method to calculate the average of some numbers to be used in the Horse class, but you don't want to put that in Horse since it is unrelated and it also is not related to animals, you can create a class to have this kind of methods. You don't need an instance of such a utility class.

- **partial class** A partial class is nothing more than splitting the file of a class into multiple smaller files. A reason to do this might be to share only part of the source code to others. If the reason is that the file gets too big, think about splitting the class in smaller classes first.

Q2: How to solve *Circular Reference*? ☆☆☆☆

Topics: C# OOP

Problem:

How do you solve circular reference problems like `Class A` has `Class B` as one of its properties, while `Class B` has `Class A` as one of its properties?

Solution:

First I would tell you needs to rethink your design. Circular references like you describe are often a code smell of a design flaw. In most cases when I've had to have two things reference each other, I've created an interface to remove the circular reference. For example:

BEFORE

```
public class Foo
{
    Bar myBar;
}

public class Bar
```

```
{
    Foo myFoo;
}
```

Dependency graph:

```
Foo     Bar
 ^       ^
 |       |
Bar     Foo
```

Foo depends on Bar, but Bar also depends on Foo. If they are in separate assemblies, you will have problems building, particularly if you do a clean rebuild.

AFTER

```
public interface IBar
{
}

public class Foo
{
    IBar myBar;
}

public class Bar : IBar
{
    Foo myFoo;
}
```

Dependency graph:

```
Foo, IBar     IBar
    ^          ^
    |          |
   Bar        Foo
```

Both Foo and Bar depend on IBar. There is no circular dependency, and if IBar is placed in its own assembly, Foo and Bar being in separate assemblies will no longer be an issue.

Q3: What are the differences between `IEnumerable` and `IQueryable`?

☆☆☆☆☆

Topics: C#

Answer:

- The `IEnumerable<>` interface indicates that something can be enumerated across--in other words, you can do a foreach loop on it.
- The `IQueryable<>` interface indicates that something has some kind of backing query provider that's capable of looking at Expressions that are given to it, and translate them into some kind of query.

What IQueryable has that IEnumerable doesn't are two properties in particular—one that points to a **query provider** (e.g., a LINQ to SQL provider) and another one pointing to a **query expression** representing the IQueryable object as a runtime-traversable abstract syntax tree that can be understood by the given query

provider (for the most part, you can't give a LINQ to SQL expression to a LINQ to Entities provider without an exception being thrown).

Q4: What is *deep* or *shallow* copy concept in C#? ☆☆☆☆☆

Topics: C#

Answer:

- **Shallow Copy** is about copying an object's value type fields into the target object and the object's reference types are copied as references into the target object but not the referenced object itself. It copies the types bit by bit. The result is that both instances are cloned and the original will refer to the same object.
- **Deep Copy** is used to make a complete deep copy of the internal reference types, for this we need to configure the object returned by `MemberwiseClone()`.

In other words a deep copy occurs when an object is copied along with the objects to which it refers.

Q5: What's the difference between the `System.Array.CopyTo()` and `System.Array.Clone()`? ☆☆☆☆☆

Topics: C#

Answer:

- **Clone** - Method creates a shallow copy of an array. A shallow copy of an Array copies only the elements of the Array, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new Array point to the same objects that the references in the original Array point to.
- **CopyTo** - The Copy static method of the Array class copies a section of an array to another array. The CopyTo method copies all the elements of an array to another one-dimension array. The code listed in Listing 9 copies contents of an integer array to an array of object types.

Q6: What is *Multicast Delegate* in C#? ☆☆☆☆☆

Topics: C#

Answer:

Delegate can invoke only one method reference has been encapsulated into the delegate. It is possible for certain delegate to hold and invoke multiple methods. Such delegate called **multicast delegate**. Multicast delegate also know as combinable delegate, must satisfy the following conditions:

- The return type of the delegate must be void. None of the parameters of the delegate type can be delegate type can be declared as output parameters using out keywords.
- Multicast delegate instance is created by combining two delegates, the invocation list is formed by concatenating the invocation list of two operand of the addition operation. Delegates are invoked in the order they are added.

Actually all delegates in C# are MulticastDelegates, even if they only have a single method as target. (Even anonymous functions and lambdas are MulticastDelegates even though they by definition have only single target.)

Consider:

```
ublic partial class MainPage : PhoneApplicationPage
{
    public delegate void MyDelegate(int a, int b);
    // Constructor
    public MainPage()
    {
        InitializeComponent();

        // Multicast delegate
        MyDelegate myDel = new MyDelegate(AddNumbers);
        myDel += new MyDelegate(MultiplyNumbers);
        myDel(10, 20);
    }

    public void AddNumbers(int x, int y)
    {
        int sum = x + y;
        MessageBox.Show(sum.ToString());
    }

    public void MultiplyNumbers(int x, int y)
    {
        int mul = x * y;
        MessageBox.Show(mul.ToString());
    }

}
```

Q7: What is the method `MemberwiseClone()` doing? ☆☆☆☆☆

Topics: C#

Answer:

The **MemberwiseClone()** method creates a *shallow copy* by creating a new object, and then copying the nonstatic fields of the current object to the new object.

- If a field is a *value type*, a bit-by-bit copy of the field is performed.
- If a field is a *reference type*, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object.

Consider:

```
public class Person
{
    public int Age;
    public string Name;
    public IdInfo IdInfo;

    public Person ShallowCopy()
    {
       return (Person) this.MemberwiseClone();
    }

    public Person DeepCopy()
    {
       Person other = (Person) this.MemberwiseClone();
       other.IdInfo = new IdInfo(IdInfo.IdNumber);
       other.Name = String.Copy(Name);
       return other;
    }
}
```

Q8: List some different ways for equality check in .NET ☆☆☆☆☆

Topics: C#

Answer:

- The `ReferenceEquals()` method - checks if two reference type variables(classes, not structs) are referred to the same memory address.
- The `virtual Equals()` method. (`System.Object`) - checks if two objects are equivalent.
- The `static Equals()` method - is used to handle problems when there is a `null` value in the check.
- The `Equals` method from `IEquatable` interface.
- The comparison operator `==` - usually means the same as ReferenceEquals, it checks if two variables point to the same memory address. The gotcha is that this operator can be override to perform other types of checks. In strings, for instance, it checks if two different instances are equivalent.

Q9: What is *jagged array* in C# and when to prefer jagged arrays over multi-dimensional arrays? ☆☆☆☆☆

Topics: C#

Answer:

A jagged array is an array-of-arrays, so an `int[][]` is an array of `int[]`, each of which can be of different lengths and occupy their own block in memory. A multidimensional array (`int[,]`) is a single block of memory (essentially a matrix). The whole point of a jagged array is that the "nested" arrays n**eedn't be of uniform size**.

You could have

```
int[][] jaggedArray = new int[5][];
jaggedArray[0] = new[] {1, 2, 3}; // 3 item array
jaggedArray[1] = new int[10];     // 10 item array
// etc.
```

It's a *set* of related arrays.

A multidimensional array, on the other hand, is more of a cohesive grouping, like a box, table, cube, etc., where there are no irregular lengths. That is to say

```
int i = array[1,10];
int j = array[2,10]; // 10 will be available at 2 if available at 1
```

Also you can't create a `MyClass[10][20]` because each sub-array has to be initialized separately, as they are separate objects:

```
MyClass[][] abc = new MyClass[10][];

for (int i=0; i<abc.Length; i++) {
    abc[i] = new MyClass[20];
}
```

A `MyClass[10,20]` is ok, because it is initializing a single object as a matrix with 10 rows and 20 columns.

Q10: What are *Circular References* in C#? ☆☆☆☆☆

Topics: C#

Answer:

Circular reference is situation in which two or more resources are interdependent on each other causes the lock condition and make the resources unusable. Another example is Class A has class B as one of its properties, while Class B has Class A as one of its properties.

Consider:

```
public class Foo
{
    Bar myBar;
}

public class Bar
{
    Foo myFoo;
}
```

Q11: Could you explain the difference between `destructor`, `dispose` and `finalize` method? ☆☆☆☆☆

Topics: C#

Answer:

In C# terms, a destructor and finalizer are basically interchangeable concepts, and should be used to release **unmanaged** resources when a type is collected, for example external handles. It is ***very*** rare that you need to write a finalizer.

The problem with that is that GC is non-deterministic, so the `Dispose()` method (via `IDisposable`) makes it possible to support *deterministic* cleanup. This is unrelated to garbage collection, and allows the caller to release any resources *sooner*. It is also suitable for use with *managed* resources (in addition to unmanaged), for example if you have a type that *encapsulates* (say) a database connection, you might want disposing of the type to release the connection too.

Q12: Why Abstract class can not be sealed or static? ☆☆☆☆☆

Topics: C#

Answer:

`Sealed` is a modifier which if applied to a class make it non-inheritable and if applied to virtual methods or properties makes them non-ovveridable. `abstract` class makes sense when you want all derived classes to implement same part of the logic. Because a `sealed` class cannot be inherited, it cannot be used as base class and by consequence, an `abstract` class cannot use the sealed modifier.

It's also important to mention that structs are implicitly sealed.

Q13: What is a *preprocessor directives* in C#? ☆☆☆☆☆

Topics: C#

Answer:

The **preprocessor directives** give instruction to the compiler to preprocess the information before actual compilation starts. Generally, the optional/conditional compilation symbols will be provided by the build script.

Consider:

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

I would actually recommend using the Conditional Attribute instead of inline #if statements.

```
[Conditional("DEBUG")]
private void DeleteTempProcessFiles()
{
}
```

Not only is this cleaner and easier to read since you don't end up having #if, #else within your code. This style is less prone to errors either during normal code edits and well as logic flow errors.

Q14: What is the use of `static` constructors? ☆☆☆☆☆

Topics: C#

Answer:

A **static constructor** is useful for initializing any static fields associated with a type (or any other per-type operations) - useful in particular for reading required configuration data into readonly fields, etc.

It is run automatically by the runtime the first time it is needed (the exact rules there are complicated (see "beforefieldinit"), and changed subtly between CLR2 and CLR4). Unless you abuse reflection, it is guaranteed to run at most once (even if two threads arrive at the same time).

You can't overload it.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Q15: What are the benefits of a *Deferred Execution* in LINQ? ☆☆☆☆☆

Topics: C# LINQ

Answer:

In LINQ, queries have two different behaviors of execution: **immediate** and **deferred**.

Deferred execution means that the evaluation of an expression is delayed until its realized value is actually required. It greatly improves performance by avoiding unnecessary execution.

Consider:

```
var results = collection.Select(item => item.Foo).Where(foo => foo < 3).ToList();
```

With deferred execution, the above iterates your collection one time, and each time an item is requested during the iteration, performs the map operation, filters, then uses the results to build the list.

If you were to make LINQ fully execute each time, each operation (Select / Where) would have to iterate through the entire sequence. This would make chained operations very inefficient.

Q16: What is the difference between *Lambdas* and *Delegates*? ☆☆☆☆☆

Topics: C#

Answer:

They are actually two very different things.

- **Delegate** is actually the name for a variable that holds a reference to a method or a lambda, and a lambda is a method without a permanent name.

  ```
  delegate Int32 BinaryIntOp(Int32 x, Int32 y);
  ```

- **Lambdas** are very much like other methods, except for a couple subtle differences:
  - A normal method is defined in a "statement" and tied to a permanent name, whereas a lambda is defined "on the fly" in an "expression" and has no permanent name.
  - Lambdas can be used with .NET expression trees, whereas methods cannot.

A lambda might be defined like this:

```
BinaryIntOp sumOfSquares = (a, b) => a*a + b*b;
```

Q17: Could you explain the difference between `Func` vs. `Action` vs. `Predicate`? ☆☆☆☆☆

Topics: C#

Answer:

- **Predicate**: essentially `Func<T, bool>`; asks the question "does the specified argument satisfy the condition represented by the delegate?" Used in things like `List.FindAll`.

- **Action**: Perform an action given the arguments. Very general purpose. Not used much in LINQ as it implies side-effects, basically.
- **Func**: Used extensively in LINQ, usually to transform the argument, e.g. by projecting a complex structure to one property.

Q18: Can you add extension methods to an existing static class? ☆☆☆☆☆

Topics: C#

Answer:

No. Extension methods require an instance variable (value) for an object. You can however, write a static wrapper class. Also extension methods is just syntactic sugar.

Consider:

```
public static string SomeStringExtension(this string s)
{
   //whatever..
}
// When you then call it
myString.SomeStringExtension();
// the compiler just turns it into:
ExtensionClass.SomeStringExtension(myString);
```

Q19: What is the `volatile` keyword used for? ☆☆☆☆☆

Topics: C#

Answer:

In C# `volatile` tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself (such as the operating system, the hardware, or a concurrently executing thread). The compiler will then avoid any optimisations that may result in problems if the variable changes "outside of its control".

Or in more simple terms:

Sometimes, the compiler will optimize a field and use a register to store it. If thread 1 does a write to the field and another thread accesses it, since the update was stored in a register (and not memory), the 2nd thread would get stale data.

You can think of the volatile keyword as saying to the compiler "I want you to store this value in memory". This guarantees that the 2nd thread retrieves the latest value.

Q20: Explain what is *Weak Reference* in C#? ☆☆☆☆☆

Topics: C#

Answer:

The garbage collector cannot collect an object in use by an application while the application's code can reach that object. The application is said to have a strong reference to the object.

A **weak reference** permits the garbage collector to collect the object while still allowing the application to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist.

Weak references are useful for objects that use a lot of memory, but can be recreated easily if they are reclaimed by garbage collection.

FullStack.Cafe - Kill Your Tech Interview

Q1: Filter out the first 3 even numbers from the list using LINQ ☆☆

Topics: C# LINQ

Answer:

```
var evenNumbers = List
   .Where(x => x % 2 ==0)
   .Take(3)
```

Q2: Implement a Queue using two Stacks ☆☆

Topics: Queues Stacks Java C# JavaScript

Problem:

Suppose we have two stacks and no other temporary variable. Is to possible to "construct" a queue data structure using only the two stacks?

Solution:

Keep two stacks, let's call them `inbox` and `outbox`.

Enqueue:

- Push the new element onto `inbox`

Dequeue:

- If `outbox` is empty, refill it by popping **each** element from `inbox` and pushing it onto `outbox`
- Pop and return the top element from `outbox`

Complexity Analysis:

Time Complexity: O(1) **Space Complexity**: O(1)

In the worst case scenario when outbox stack is empty, the algorithm pops n elements from inbox stack and pushes n elements to outbox, where n is the queue size. This gives `2*n` operations, which is `O(n)`. But when outbox stack is not empty the algorithm has `O(1)` time complexity that gives amortised `O(1)`.

Implementation:

CS

```
public class Queue<T> where T : class
{
    private Stack<T> input = new Stack<T>();
    private Stack<T> output = new Stack<T>();
```

```
    public void Enqueue(T t)
    {
        input.Push(t);
    }

    public T Dequeue()
    {
        if (output.Count == 0)
        {
            while (input.Count != 0)
            {
                output.Push(input.Pop());
            }
        }

        return output.Pop();
    }
}
```

Java

```
public class Queue<E>
{
    private Stack<E> inbox = new Stack<E>();
    private Stack<E> outbox = new Stack<E>();

    public void queue(E item) {
        inbox.push(item);
    }

    public E dequeue() {
        if (outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
               outbox.push(inbox.pop());
            }
        }
        return outbox.pop();
    }

}
```

PY

```
class Queue(object):
    def __init__(self):
        self.instack=[]
        self.outstack=[]

    def enqueue(self,element):
        self.instack.append(element)

    def dequeue(self):
        if not self.outstack:
            while self.instack:
                self.outstack.append(self.instack.pop())
        return self.outstack.pop()
```

JS

```
import { Stack } from "./Stack";

class QueueUsingTwoStacks {
  constructor() {
```

```
    this.stack1 = new Stack();
    this.stack2 = new Stack();
  }

  enqueue(data) {
    this.stack1.push(data);
  }

  dequeue() {
    //if both stacks are empty, return undefined
    if (this.stack1.size() === 0 && this.stack2.size() === 0)
      return undefined;

    //if stack2 is empty, pop all elements from stack1 to stack2 till stack1 is empty
    if (this.stack2.size() === 0) {
      while (this.stack1.size() !== 0) {
        this.stack2.push(this.stack1.pop());
      }
    }

    //pop and return the element from stack 2
    return this.stack2.pop();
  }
}

export { QueueUsingTwoStacks };
```

Q3: Can you return multiple values from a function in C#? Provide some examples. ☆☆☆

Topics: C#

Answer:

There are several ways.

Use **ref / out parameters**. A return statement can be used for returning only one value from a function. However, using output parameters, you can return two values from a function.

Consider:

```
private static void Add_Multiply(int a, int b, ref int add, ref int multiply)
{
    add = a + b;
    multiply = a * b;
}
```

or

```
private static void Add_Multiply(int a, int b, out int add, out int multiply)
{
    add = a + b;
    multiply = a * b;
}
```

Another way is to use `<Tuple>`:

```
private static Tuple<int, int> Add_Multiply(int a, int b)
{
    var tuple = new Tuple<int, int>(a + b, a * b);
```

```
    return tuple;
}
```

Now that C# 7 has been released, you can use the **new included Tuples syntax**:

```
(string, string, string) LookupName(long id) // tuple return type
{
    ... // retrieve first, middle and last from data storage
    return (first, middle, last); // tuple literal
}
```

which could then be used like this:

```
var names = LookupName(id);
WriteLine($"found {names.Item1} {names.Item3}.");
```

Q4: Given an array of ints, write a C# method to total all the values that are even numbers. ☆☆☆

Topics: C#

Answer:

```
static long TotalAllEvenNumbers(int[] intArray) {
  return intArray.Where(i => i % 2 == 0).Sum(i => (long)i);
}
```

or

```
static long TotalAllEvenNumbers(int[] intArray) {
  return (from i in intArray where i % 2 == 0 select (long)i).Sum();
}
```

Q5: What is the output of the program below? Explain. ☆☆☆

Topics: C#

Problem:

Consider:

```
delegate void Printer();

static void Main()
{
        List<Printer> printers = new List<Printer>();
        int i=0;
        for(; i < 10; i++)
        {
            printers.Add(delegate { Console.WriteLine(i); });
        }
```

```
        foreach (var printer in printers)
        {
            printer();
        }
}
```

Solution:

This program will output the number 10 ten times.

Here's why: The delegate is added in the for loop and "reference" (or perhaps "pointer" would be a better choice of words) to `i` is stored, rather than the value itself. Therefore, after we exit the loop, the variable `i` has been set to 10, so by the time each delegate is invoked, the value passed to all of them is 10.

Q6: Refactor the code ☆☆☆

Topics: C#

Problem:

```
class ClassA
{
  public ClassA() { }

  public ClassA(int pValue) {  }
}

// client program
class Program
{
  static void Main(string[] args)
  {
    ClassA refA = new ClassA();
  }
}
```

Is there a way to modify `ClassA` so that you can you call the constructor with parameters, when the Main method is called, without creating any other new instances of the `ClassA`?

Solution:

The `this` keyword is used to call other constructors, to initialize the class object. The following shows the implementation:

```
class ClassA
{
  public ClassA() : this(10)
  { }

  public ClassA(int pValue)
  {  }
}
```

Q7: Reverse the ordering of words in a String ☆☆☆

Topics: Strings C#

Problem:

I have this string

```
s1 = "My name is X Y Z"
```

and I want to reverse the order of the words so that

```
s1 = "Z Y X is name My"
```

Is it possible to do it in-place (without using additional data structures) and with the time complexity being `O(n)`?

Solution:

Reverse the entire string, then reverse the letters of each individual word.

After the first pass the string will be

```
s1 = "Z Y X si eman yM"
```

and after the second pass it will be

```
s1 = "Z Y X is name My"
```

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

Reversing the entire string is easily done in `O(n)`, and reversing each word in a string is also easy to do in `O(n)`. `O(n)+O(n) = O(n)`. Space complexity is `O(1)` as reversal done *in-place*.

Implementation:

CS

```
static char[] ReverseAllWords(char[] in_text)
{
    int lindex = 0;
    int rindex = in_text.Length - 1;
    if (rindex > 1)
    {
        //reverse complete phrase
        in_text = ReverseString(in_text, 0, rindex);

        //reverse each word in resultant reversed phrase
        for (rindex = 0; rindex <= in_text.Length; rindex++)
        {
            if (rindex == in_text.Length || in_text[rindex] == ' ')
            {
                in_text = ReverseString(in_text, lindex, rindex - 1);
                lindex = rindex + 1;
            }
```

```
            }
        }
        return in_text;
    }

    static char[] ReverseString(char[] intext, int lindex, int rindex)
    {
        char tempc;
        while (lindex < rindex)
        {
            tempc = intext[lindex];
            intext[lindex++] = intext[rindex];
            intext[rindex--] = tempc;
        }
        return intext;
    }
```

Q8: How to check if two Strings (words) are *Anagrams*? ☆☆☆

Topics: Strings Data Structures Java C#

Problem:

Explain what is **space** complexity of that solution?

Solution:

Two words are anagrams of each other if they contain the same number of characters and the same characters. There are two solutions to mention:

1. You should only need to **sort** the characters in lexicographic order, and determine if all the characters in one string are equal to and in the same order as all of the characters in the other string. If you sort either array, the solution becomes `O(n log n)`.
2. **Hashmap** approach where key - letter and value - frequency of letter,

- for first string populate the hashmap `O(n)`
- for second string decrement count and remove element from hashmap `O(n)`
- if hashmap is empty, the string is **anagram** otherwise not.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

The major space cost in your code is the hashmap which may contain a frequency counter for each lower case letter from `a` to `z`. So in this case, the space complexity is supposed to be constant `O(26)`.

To make it more general, the space complexity of this problem is related to the size of alphabet for your input string. If the input string only contains ASCII characters, then for the worst case you will have `O(256)` as your space cost. If the alphabet grows to the UNICODE set, then the space complexity will be much worse in the worst scenario.

So in general its `O(size of alphabet)`.

Implementation:

cs

```
public static bool AreAnagrams(string s1, string s2)
{
  if (s1 == null) throw new ArgumentNullException("s1");
  if (s2 == null) throw new ArgumentNullException("s2");

  var chars = new Dictionary<char, int>();
  foreach (char c in s1)
  {
      if (!chars.ContainsKey(c))
          chars[c] = 0;
      chars[c]++;
  }
  foreach (char c in s2)
  {
      if (!chars.ContainsKey(c))
          return false;
      chars[c]--;
  }

  return chars.Values.All(i => i == 0);
}
```

Java

```
// sorting approach
private static boolean isAnagram(String a, String b) {
        char[] ca = a.toCharArray();
        char[] cb = b.toCharArray();

        Arrays.sort(ca);
        Arrays.sort(cb);
        return Arrays.equals(ca, cb);
    }
```

PY

```
# Array size 256 since ASCII 256 unique values
def areAnagram(a, b):
    if len(a) != len(b): return False
    count1 = [0] * 256
    count2 = [0] * 256
    for i in a:count1[ord(i)] += 1
    for i in b:count2[ord(i)] += 1

    for i in range(256):
        if(count1[i] != count2[i]):return False

    return True


str1 = "Giniiii"
str2 = "Protijayi"
print(areAnagram(str1, str2))
```

Q9: Explain how does the Sentinel Search work? ☆☆☆

Topics: Searching C#

Answer:

The idea behind **linear search** is to compare the search item with the elements in the list one by one (using a loop) and stop as soon as we get the first copy of the search element in the list. Now considering the worst case in which the search element does not exist in the list of size `N` then the **Simple Linear Search** will take a total of `2N+1` comparisons (`N` comparisons against every element in the search list and `N+1` comparisons to test against the end of the loop condition).

```
    for (int i = 0; i < length; i++) { // N+1 comparisons
        if (array[i] == elementToSearch) { // N comparisons
            return i; // I found the position of the element requested
        }
    }
```

The idea of **Sentinel Linear Search** is to reduce the number of comparisons required to find an element in a list. Here we replace the last element of the list with the search element itself and run a **while loop** to see if there exists any copy of the search element in the list and quit the loop as soon as we find the search element. This will reduce one comparison in each iteration.

```
    while(a[i] != element) // N+2 comparisons worst case
        i++;
```

In that case the **while loop** makes only one comparison in each iteration and it is sure that it will terminate since the last element of the list is the search element itself. So in the worst case (if the search element does not exists in the list) then there will be at most `N+2` comparisons (`N` comparisons in the while loop and `2` comparisons in the if condition). Which is better than (`2N+1`) comparisons as found in **Simple Linear Search**.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

Take note that both the algorithms (Simple Linear and Sentinel) have time complexity of `O(n)`.

Implementation:

cs

```
/// <summary>
/// Gets the index of first founded item
/// </summary>
/// <typeparam name="T">The type of array</typeparam>
/// <param name="array">Input array</param>
/// <param name="value">Value of searching item</param>
/// <param name="equalityComparer">Custom comparer</param>
/// <returns>The index of founded item or -1 if not found</returns>
public static int BetterLinearSearch < T > (this IEnumerable < T > array, T value, IEqualityComparer < T
> equalityComparer = null) {
  if (equalityComparer == null) equalityComparer = new DefaultEqualityComparer < T > ();

  int arrayLength = array.Count();

  T lastItem = array.Last();
  array = array.SetElement(arrayLength - 1, value);
  int index = 0;
  while (!equalityComparer.Equals(array.ElementAt(index), value))
    index++;

  if (index < arrayLength - 1) return index;

  if (equalityComparer.Equals(lastItem, value)) return arrayLength - 1;
```

```
  return - 1;
}
```

PY

```
def sentinelSearch(ar,n,l):
    # ar : array
    # n : item to be searched
    # l : size of array
    last = ar[l-1] # saving last element in other variable
    ar[l-1] = n # assigning last element as required
    i = 0
    while ar[i]!=n:
        i+=1
    ar[l-1] = last
    if (i<l-1) or n==ar[l-1]:
        print('Item found at',i)
    else:
        print('Item not Found')
```

Q10: Can you do *Iterative Pre-order Traversal* of a *Binary Tree* without *Recursion*? ☆☆☆

Topics: Binary Tree Data Structures C#

Problem:

Given a binary tree, return the preorder traversal of its nodes' values. Can you do it without recursion?

Solution:

Let's use **Stack**:

1. Set current node as `root`
2. Check if current node is `null` then return
3. Store current node value in the container
4. Put `right` on stack
5. Put `left` on stack
6. While stack is *not empty*
   i. `pop` from stack into current node. Note, the `left` will be on top of stack, and will be taken first
   ii. Repeat from step 2

Implementation:

CS

```
/**
* @param root: A Tree
* @return: Preorder in ArrayList which contains node values.
*/
public List<Integer> preorderTraversal(TreeNode root) {
  if (root == null) {
    return new ArrayList<>();
  }

  List<Integer> rst = new ArrayList<>();
  Stack<TreeNode> stack = new Stack<>();
```

```
    stack.push(root);

    while (!stack.empty()) {
      TreeNode node = stack.pop();
      rst.add(node.val);
      if (node.right != null) {
        stack.push(node.right);
      }
      if (node.left != null) {
        stack.push(node.left);
      }
    }

    return rst;
  }
```

PY

```
def preorder_iterative(root, result):
    '''
    Pre-order iterative traversal. The nodes' values are
    appended to the result list in traversal order
    '''
    if not root:
        return

    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()

        result.append(node.value)

        if node.right: stack.append(node.right)
        if node.left: stack.append(node.left)
```

Q11: Is the comparison of time and null in the if statement below valid or not? Why or why not? ☆☆☆☆

Topics: C#

Problem:

Consider:

```
static DateTime time;
/* ... */
if (time == null)
{
    /* do something */
}
```

Solution:

One might think that, since a `DateTime` variable can never be null (it is automatically initialized to Jan 1, 0001), the compiler would complain when a `DateTime` variable is compared to `null`. However, due to type coercion, the compiler does allow it, which can potentially lead to headfakes and pull-out-your-hair bugs.

Specifically, the `==` operator will cast its operands to different allowable types in order to get a common type on both sides, which it can then compare. That is why something like this will give you the result you expect (as

opposed to failing or behaving unexpectedly because the operands are of different types):

```
double x = 5.0;
int y = 5;
Console.WriteLine(x == y);  // outputs true
```

However, this can sometimes result in unexpected behavior, as is the case with the comparison of a `DateTime` variable and `null`. In such a case, both the `DateTime` variable and the `null` literal can be cast to `Nullable<DateTime>`. Therefore it is legal to compare the two values, even though the result will *always* be false.

Q12: Binet's formula: How to calculate Fibonacci numbers without Recursion or Iteration? ☆☆☆☆

Topics: Fibonacci Series Data Structures C# JavaScript

Answer:

There is actually a simple mathematical formula called **Binet's formula** for computing the nth Fibonacci number, which does not require the calculation of the preceding numbers:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

It features the **Golden Ratio (φ)**:

```
φ = (1+sqrt(5))/2
```

or

$$x = \frac{1+\sqrt{(5)}}{2} = 1.618034...$$

The iterative approach scales linearly, while the formula approach is basically constant time.

Complexity Analysis:

Time Complexity: O(log n) **Space Complexity**: O(log n)

Assuming that the primitive mathematical operations (+, -, * and /) are `O(1)` you can use this result to compute the `n` th Fibonacci number in `O(log n)` time (`O(log n)` because of the exponentiation in the formula).

Implementation:

cs

```
static double inverseSqrt5 = 1 / Math.Sqrt(5);
static double phi = (1 + Math.Sqrt(5)) / 2;
/* should use
   const double inverseSqrt5 = 0.44721359549995793928183473374626
   const double phi = 1.61803398874989484820458683436563
*/

static int Fibonacci(int n) {
    return (int)Math.Floor(Math.Pow(phi, n) * inverseSqrt5 + 0.5);
}
```

JS

```
const sqrt = Math.sqrt;
const pow = Math.pow;

const fibCalc = n => Math.round(
  (1 / sqrt(5)) *
  (
    pow(((1 + sqrt(5)) / 2), n) -
    pow(((1 - sqrt(5)) / 2), n)
  )
);
```

Java

```
 private static long fibonacci(int n) {
     double pha = pow(1 + sqrt(5), n);
     double phb = pow(1 - sqrt(5), n);
     double div = pow(2, n) * sqrt(5);

     return (long)((pha - phb) / div);
 }
```

PY

```
def fib_formula(n):
    golden_ratio = (1 + math.sqrt(5)) / 2
    val = (golden_ratio**n - (1 - golden_ratio)**n) / math.sqrt(5)
    return int(round(val))
```

Q13: Explain what is *Fibonacci Search* technique? ☆☆☆☆

Topics: Divide & Conquer Fibonacci Series Searching Data Structures C# JavaScript

Answer:

Fibonacci search is a search algorithm based on **divide and conquer** principle that can find an element in the given **sorted array** with the help of Fibonacci series in $O(log\ n)$ time complexity.

Compared to binary search where the sorted array is divided into two equal-sized parts, one of which is examined further, Fibonacci search divides the array into two *unequal parts* that have sizes that are *consecutive Fibonacci numbers*.

Some facts to note:

- Fibonacci search has the advantage that one only needs addition and subtraction to calculate the indices of the accessed array elements, while classical binary search needs bit-shift, division or multiplication, operations that were less common at the time Fibonacci search was first published
- Binary search works by dividing the seek area in equal parts `1:1`. Fibonacci search can divide it into parts approaching `1:1.618` while using the simpler operations.
- On average, Fibonacci search requires 4% more comparisons than binary search
- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful. Thus Fibonacci search may have the advantage over binary search in slightly reducing the average time needed to access a non-uniform access memory storage.

Let the length of given array be `length` and the element to be searched be `key`.

Steps of Fibonacci Search:

1. Find the smallest Fibonacci number `F(n)` that `F(n) - 1 >= length`

2. `F(n) = F(n-1) + F(n-2)` => `F(n) - 1 = [ F(n-1) - 1 ] + (1) + [ F(n-2) - 1 ]`, where `(1)` here is for the middle item (the **m** in below figure).

```
// |<---------             p: F(n) - 1             --------->|
// |<---    q: F(n-1) - 1    --->|   |<--  r: F(n-2) - 1  -->|
// +----+---+--------------------+---+---+---+---+
// |    | k |                    | m |   | k |   |
// +----+---+--------------------+---+---+---+---+
//                                ^
```

3. If `key < m`, then search key in `q` = `F(n-1) - 1`:

```
// |<---    p: F(n-1) - 1    --->|
// |<----- q ----->|   |<-- r -->|
// +----+---+------+---+---------+
// |    | k |      | m |         |
// +----+---+------+---+---------+
```

   for that set `p = F(n-1) - 1, q = F(n-2) - 1, r = F(n-3) - 1` and repeat the process. Here Fibonacci numbers go *backward* **once**. These indicate elimination of approximately one-third of the remaining array.

4. If `key > m`, then repeat then search key in `r = F(n-2) - 1`:

```
// |<--  p: F(n-2) - 1  -->|
// |<--- q --->|   |<- r ->|
// +---+---+---+
// |   | k |   | m
// +---+---+---+
//               ^
```

   For that, set `p = F(n-2) - 1, q = F(n-3) - 1, r = F(n-4) - 1` and repeat the process. Here Fibonacci numbers go *backward* **twice**, indicating elimination of approximately two-third of the remaining array.

Complexity Analysis:

Time Complexity: O(log n) **Space Complexity**: O(log n)

Fibonacci search has an average- and worst-case complexity of $O(\log n)$. The worst case will occur when we have our target in the larger (2/3) fraction of the array, as we proceed to find it. In other words, we are

eliminating the smaller (1/3) fraction of the array every time. We call once for n, then for (2/3) n, then for (4/9) n and henceforth.

$$fib(n) = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right] \sim c*1.62^n$$

$$for\ n \sim c*1.62^n\ we\ make\ O(n')\ comparisons.\ We,\ thus,\ need\ O(\log(n))\ comparisons.$$

Implementation:

cs

```
#include <algorithm>
#include <cassert>
#include "search.h"

typedef struct FibonacciValues
{
  // Fibonacci number: F(n) = F(n-1) + F(n-2)
  int Fn_2; // F(n-2)
  int Fn_1; // F(n-1)
  int Fn;   // F(n)

  FibonacciValues(int fn_2, int fn_1, int fn)
    : Fn_2(fn_2)
    , Fn_1(fn_1)
    , Fn(fn)
  {
    assert(Fn == Fn_1 + Fn_2);
  }

  ~FibonacciValues()
  {
  }

  void forward()
  {
    Fn_2 = Fn_1;
    Fn_1 = Fn;
    Fn = Fn_1 + Fn_2;
    assert(Fn == Fn_1 + Fn_2);
  }

  void backward()
  {
    Fn = Fn_1;
    Fn_1 = Fn_2;
    Fn_2 = Fn - Fn_1;
    assert(Fn == Fn_1 + Fn_2);
  }

} FibonacciValues;

// Returns the smallest Fibonacci number that is greater than or equal to k.
FibonacciValues GetFibonacci(int k)
{
  FibonacciValues fib(0, 1, 1);
  while (fib.Fn < k) {
    fib.forward();
  }
  return fib;
}

// Notice that the current middle might exceed the array!
```

```
int fibonacciSearch(int list[], const unsigned int length, int key)
{
  assert(list && length);

  int left = 0, right = length - 1, middle;
  // Find a fibonacci number F(n) that F(n) - 1 >= length
  FibonacciValues fib = GetFibonacci(length + 1);

  while (left <= right) {
    // Avoid the middle is over the array.
    middle = std::min(left + (fib.Fn_1 - 1), right);

    if (list[middle] == key) {
      return middle;
    } else if (list[middle] > key) {
      right = middle - 1;
      fib.backward();
    } else { // list[middle] < key
      left = middle + 1;
      fib.backward(); fib.backward();
    }
  }

  return NOT_FOUND;
}
```

JS

```
function fib(n) {
    if (n <= 0)
        return 0;
    if (n <= 2)
        return 1;
    return fib(n-1) + fib(n-2);
}

function smallest_greater_eq_fib(n) {
    let f = fib(0),
        cu = 0;
    while (f < n)
        f = fib(++cu);

    return cu;
}

async function fibonacciSearch(a, k, l, r, display) {
    let f = smallest_greater_eq_fib(r-l+1);

    while (f >= 0) {
        i = Math.min(l+fib(f-1), r-1);
        i = Math.max(0, i);

        refresh(glob_comp, a[i]);

        display(a, i, l, r);
        await sleep(glob_sleep_time);

        if (a[i]==k) {
            glob_comp++;
            refresh(glob_comp, a[i]);

            return new Promise(resolve => resolve(i));
        } else if (k < a[l+fib(f-1)]) {
            glob_comp+=2;
            refresh(glob_comp, a[i]);

            r = i;
            f-=1;
        } else {
            glob_comp+=2;
            refresh(glob_comp, a[i]);
```

```
            l = i;
            f-=2;
        }
    }

    return new Promise(resolve => resolve(-1));
}
```

Java

```
public class Fibonacci_Search
{
    static int kk = -1, nn = -1;
    static int fib[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
            377, 610, 98, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368,
            75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309,
            3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986,
            102334155, 165580141 };

    static int fibsearch(int a[], int n, long x)
    {
        int inf = 0, pos, k;
        if (nn != n)
        {
            k = 0;
            while (fib[k] < n)
                k++;
            kk = k;
            nn = n;
        }
        else
            k = kk;

        while (k > 0)
        {
            pos = inf + fib[--k];
            if ((pos >= n) || (x < a[pos]))
                ;
            else if (x > a[pos])
            {
                inf = pos + 1;
                k--;
            }
            else
                return pos;
        }
        return -1;
    }
}
```

PY

```
def FibonacciSearch(lys, val):
    fibM_minus_2 = 0
    fibM_minus_1 = 1
    fibM = fibM_minus_1 + fibM_minus_2
    while (fibM < len(lys)):
        fibM_minus_2 = fibM_minus_1
        fibM_minus_1 = fibM
        fibM = fibM_minus_1 + fibM_minus_2
    index = -1;
    while (fibM > 1):
        i = min(index + fibM_minus_2, (len(lys)-1))
        if (lys[i] < val):
            fibM = fibM_minus_1
            fibM_minus_1 = fibM_minus_2
```

```
            fibM_minus_2 = fibM - fibM_minus_1
            index = i
        elif (lys[i] > val):
            fibM = fibM_minus_2
            fibM_minus_1 = fibM_minus_1 - fibM_minus_2
            fibM_minus_2 = fibM - fibM_minus_1
        else :
            return i
    if(fibM_minus_1 and index < (len(lys)-1) and lys[index+1] == val):
        return index+1;
    return -1
```

Q14: Find Merge (Intersection) Point of Two Linked Lists ☆☆☆☆

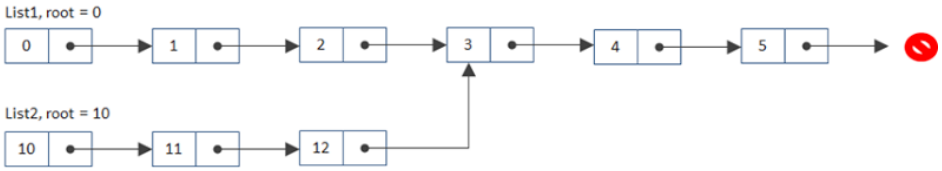
Topics: Linked Lists C#

Problem:

Say, there are two lists of different lengths, **merging at a point**; how do we know where the merging point is?

Conditions:

1. We don't know the length



Solution:

This arguably violates the "parse each list only once" condition, but **implement the tortoise and hare algorithm** (used to find the merge point and cycle length of a cyclic list) so you start at List 1, and when you reach the NULL at the end you pretend it's a pointer to the beginning of List 2, thus creating the appearance of a cyclic list. The algorithm will then tell you exactly how far down List 1 the merge is.

Algorithm steps:

- Make an interating pointer like this:
  - it goes forward every time till the end, and then
  - jumps to the beginning of the opposite list, and so on.
- Create two of these, pointing to two heads.
- Advance each of the pointers by 1 every time, until they meet in intersection point (`IP`). This will happen after either one or two passes.

To understand it count the number of nodes traveled from `head1-> tail1 -> head2 -> intersection point` and `head2 -> tail2-> head1 -> intersection point`. Both will be equal (Draw diff types of linked lists to verify this). Reason is both pointers have to travel same distances `head1-> IP + head2->IP` before reaching `IP` again. So by the time it reaches `IP`, both pointers will be equal and we have the merging point.

Implementation:

CS

```
int FindMergeNode(Node headA, Node headB) {
    Node currentA = headA;
```

```
    Node currentB = headB;

    //Do till the two nodes are the same
    while(currentA != currentB){
        //If you reached the end of one list start at the beginning of the other one
        //currentA
        if(currentA.next == null){
            currentA = headB;
        }else{
            currentA = currentA.next;
        }
        //currentB
        if(currentB.next == null){
            currentB = headA;
        }else{
            currentB = currentB.next;
        }
    }
    return currentB.data;
}
```

Java

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    //boundary check
    if(headA == null || headB == null) return null;

    ListNode a = headA;
    ListNode b = headB;

    //if a & b have different len, then we will stop the loop after second iteration
    while( a != b){
        //for the end of first iteration, we just reset the pointer to the head of another linkedlist
        a = a == null? headB : a.next;
        b = b == null? headA : b.next;
    }

    return a;
}
```

PY

```
# the idea is if you switch head, the possible difference between length would be countered.
# On the second traversal, they either hit or miss.
# if they meet, pa or pb would be the node we are looking for,
# if they didn't meet, they will hit the end at the same iteration, pa == pb == None, return either one
of them is the same,None

class Solution:
    # @param two ListNodes
    # @return the intersected ListNode
    def getIntersectionNode(self, headA, headB):
        if headA is None or headB is None:
            return None

        pa = headA # 2 pointers
        pb = headB

        while pa is not pb:
            # if either pointer hits the end, switch head and continue the second traversal,
            # if not hit the end, just move on to next
            pa = headB if pa is None else pa.next
            pb = headA if pb is None else pb.next

        return pa # only 2 ways to get out of the loop, they meet or the both hit the end=None
```

Q15: Explain when to use `Finalize` vs `Dispose`? ☆☆☆☆☆

Topics: .NET Core C#

Answer:

- The **finalizer method** is called when your object is garbage collected and you have no guarantee when this will happen (you can force it, but it will hurt performance).

- The **Dispose method**, on the other hand, is meant to be called by the code that created your class so that you can clean up and release any resources you have acquired (unmanaged data, database connections, file handles, etc) the moment the code is done with your object.

The standard practice is to implement `IDisposable` and `Dispose` so that you can use your object in a `using` statement such as `using(var foo = new MyObject()) { }`. And in your finalizer, you call `Dispose`, just in case the calling code forgot to dispose of you.

Q16: Explain the difference between `IQueryable`, `ICollection`, `IList` & `IDictionary` interfaces? ☆☆☆☆☆

Topics: C#

Answer:

All of these interfaces inherit from **IEnumerable**. That interface basically lets you use the class in a `foreach` statement (in C#).

- **ICollection** is the most basic of the interfaces you listed. It's an enumerable interface that supports a Count and that's about it.
- **IList** is everything that ICollection is, but it also supports adding and removing items, retrieving items by index, etc. It's the most commonly-used interface for "lists of objects".
- **IQueryable** is an enumerable interface that supports LINQ. You can always create an IQueryable from an IList and use LINQ to Objects, but you also find IQueryable used for deferred execution of SQL statements in LINQ to SQL and LINQ to Entities.
- **IDictionary** is a different animal in the sense that it is a mapping of unique keys to values. It is also enumerable in that you can enumerate the key/value pairs, but otherwise it serves a different purpose than the others you listed.

Q17: in C#, when should we use abstract classes instead of interfaces with extension methods? ☆☆☆☆☆

Topics: C#

Answer:

- **Abstract** classes work by *inheritance*. Being just special base classes, they model some is-a-relationship.

- **Interfaces** on the other hand are a different story. They don't use inheritance but provide *polymorphism* (which can be implemented with inheritance too). They don't model an is-a relationship, but more of a it does support.

Applying extension methods to an interface is useful for applying common behaviour across classes that may share only a common interface.

Q18: Implement the `Where` method in C#. Explain. ☆☆☆☆☆

Topics: C#

Answer:

Consider:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> items, Predicate< T> prodicate)
{
  foreach(var item in items)
  {
      if (predicate(item))
      {
           // for lazy/deffer execution plus avoid temp collection defined
           yield return item;
      }
  }
}
```

The `yield` keyword actually does quite a lot here. It creates a state machine "under the covers" that remembers where you were on each additional cycle of the function and picks up from there.

The function returns an object that implements the `IEnumerable<T>` interface. If a calling function starts `foreach`ing over this object, the function is called again until it "yields" result based on some `predicate`. This is syntactic sugar introduced in **C# 2.0**. In earlier versions you had to create your own `IEnumerable` and `IEnumerator` objects to do stuff like this.

Q19: You have defined a *destructor* in a class that you have developed by using the C#, but the destructor *never executed*. Why? ☆☆☆☆☆

Topics: OOP C#

Answer:

The runtime environment automatically invokes the destructor of a class to release the resources that are occupied by variables and methods of an object. However, in C#, programmers cannot control the timing for invoking destructors, as Garbage Collector is only responsible for releasing the resources used by an object. Garbage Collector automatically gets information about unreferenced objects from .NET's runtime environment and then invokes the *Finalize()* method.

Although, it is not preferable to force Garbage Collector to perform garbage collection and retrieve all inaccessible memory, programmers can use the *Collect()* method of the Garbage Collector class to forcefully execute Garbage Collector.

Q20: Why doesn't C# allow *static methods* to implement an *interface*? ☆☆☆☆☆

Topics: OOP C#

Answer:

My (simplified) technical reason is that static methods are not in the vtable, and the call site is chosen at compile time. It's the same reason you can't have override or virtual static members. For more details, you'd need a CS

grad or compiler wonk - of which I'm neither.

You pass an interface to someone, they need to know how to call a method. An interface is just a virtual method table (vtable). Your static class doesn't have that. The caller wouldn't know how to call a method. (Before i read this answer i thought C# was just being pedantic. Now i realize it's a technical limitation, imposed by what an interface is). Other people will talk down to you about how it's a bad design. It's not a bad design - it's a technical limitation.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is the output of the short program below? Explain your answer. ☆☆☆☆

Topics: C#

Problem:

Consider:

```
class Program {
  static String location;
  static DateTime time;

  static void Main() {
    Console.WriteLine(location == null ? "location is null" : location);
    Console.WriteLine(time == null ? "time is null" : time.ToString());
  }
}
```

Solution:

The output will be:

```
location is null
1/1/0001 12:00:00 AM
```

Although both variables are uninitialized, `String` is a reference type and `DateTime` is a value type. As a value type, an unitialized `DateTime` variable is set to a default value of midnight of 1/1/1 (yup, that's the year 1 A.D.), *not* `null`.

Q2: What is the output of the program below? Explain your answer. ☆☆☆☆

Topics: C#

Problem:

Consider:

```
class Program {
  private static string result;

  static void Main() {
    SaySomething();
    Console.WriteLine(result);
  }

  static async Task<string> SaySomething() {
    await Task.Delay(5);
    result = "Hello world!";
```

```
    return “Something”;
  }
}
```

Also, would the answer change if we were to replace `await Task.Delay(5);` with `Thread.Sleep(5)`? Why or why not?

Solution:

The answer to the first part of the question (i.e., the version of the code with `await Task.Delay(5);`) is that the program will just output a blank line (*not* “Hello world!”). This is because `result` will still be uninitialized when `Console.WriteLine` is called.

Most procedural and object-oriented programmers expect a function to execute from beginning to end, or to a `return` statement, before returning to the calling function. This is not the case with C# `async` functions. They only execute up until the first `await` statement, then return to the caller. The function called by `await` (in this case `Task.Delay`) is executed asynchronously, and the line after the `await` statement isn’t signaled to execute until `Task.Delay` completes (in 5 milliseconds). However, within that time, control has already returned to the caller, which executes the `Console.WriteLine` statement on a string that hasn’t yet been initialized.

Calling `await Task.Delay(5)` lets the current thread continue what it is doing, and if it’s done (pending any awaits), returns it to the thread pool. This is the primary benefit of the async/await mechanism. It allows the CLR to service more requests with less threads in the thread pool.

Asynchronous programming has become a lot more common, with the prevalence of devices which perform over-the-network service requests or database requests for many activities. C# has some excellent programming constructs which greatly ease the task of programming asynchronous methods, and a programmer who is aware of them will produce better programs.

With regard to the second part of the question, if `await Task.Delay(5);` was replaced with `Thread.Sleep(5)`, the program would output `Hello world!`. An `async` method *without* at least one `await` statement in it operates just like a synchronous method; that is, it will execute from beginning to end, or until it encounters a `return` statement. Calling `Thread.Sleep()` simply blocks the currently running thread, so the `Thread.Sleep(5)` call just adds 5 milliseconds to the execution time of the `SaySomething()` method.

Q3: What is the output of the program below? ☆☆☆☆

Topics: C#

Problem:

Consider:

```
public class TestStatic {
 public static int TestValue;

 public TestStatic() {
  if (TestValue == 0) {
   TestValue = 5;
  }
 }
 static TestStatic() {
  if (TestValue == 0) {
   TestValue = 10;
  }

 }

 public void Print() {
  if (TestValue == 5) {
```

```
    TestValue = 6;
   }
   Console.WriteLine("TestValue : " + TestValue);

 }
}

public void Main(string[] args) {

 TestStatic t = new TestStatic();
 t.Print();
}
```

Solution:

`TestValue : 10`

The static constructor of a class is called before any instance of the class is created. The static constructor called here initializes the `TestValue` variable first.

Q4: Is relying on && short-circuiting safe in .NET? ☆☆☆☆

Topics: C#

Problem:

Assume **myObj** is `null`. Is it safe to write this?

```
if(myObj != null && myObj.SomeString != null)
```

Solution:

Yes. In C# && and || are **short-circuiting** and thus evaluates the right side only if the left side doesn't already determine the result.

In C# 6.0 you can do this too:

```
if(myObj?.SomeString != null)
```

Which is the same thing as above.

Q5: Calculate the circumference of the circle ☆☆☆☆☆

Topics: C#

Problem:

Given an instance circle of the following class:

```
public sealed class Circle {
  private double radius;

  public double Calculate(Func<double, double> op) {
```

```
    return op(radius);
  }
}

write code to calculate the circumference of the circle, without modifying the `Circle` class itself.
```

Solution:

The preferred answer would be of the form:

```
circle.Calculate(r => 2 * Math.PI * r);
```

Since we don't have access to the *private* `radius` field of the object, we tell the object itself to calculate the circumference, by passing it the calculation function inline.

A lot of C# programmers shy away from (or don't understand) function-valued parameters. While in this case the example is a little contrived, the purpose is to see if the applicant understands how to formulate a call to `Calculate` which matches the method's definition.

Alternatively, a valid (though less elegant) solution would be to retrieve the radius value itself from the object and then perform the calculation with the result:

```
var radius = circle.Calculate(r => r);
var circumference = 2 * Math.PI * radius;
```

Either way works. The main thing we're looking for here is to see that the candidate is familiar with, and understands how to invoke, the `Calculate` method.