**What is .NET Framework environment?**

* .NET facilitates the development and execution of distributed applications.
* .NET framework provides:
* Option to use different programming languages
* Safety and tolerability of programs
* Common programming model based on the Windows platform

**Operator**

* Ternary operator -- can replace simple if else or switch.

**Date Types**

* Value Type -- implemented by struct. Default is non-nullable (0).
  + To make non-nullable types nullable, put (?) in front of the type in the declare statement.
  + Use the null-coalescing operator (??) to check if a value is null and assign value (nullable type ternary operator).
* Reference Type -- implemented by class. Default is nullable.

**Date Type Conversion**

* Type Cast -- Does not throw errors
* Convert class -- Throw errors
* Parse/Try Parse methods on type -- ignore errors

**Array ([])**

* cannot increase the size after it is set
* One-dimensional array
* Multi-dimensional array
* Stepped array
  + Stepped arrays are effective in cases where two-dimensional rectangular array has a lot of items that are not used at all

**Method:** perform some form of operation on the data, changing the state of the object.

* Static method -- belongs to class and call using the class name. Only one definition of the method can be invoked
* Instance method -- belongs to instance of a class. Multiple instances of a class can be created, and each instance has its own sperate method.
* Value Parameters -- Creates a copy of the parameter passed, so modifications don't affect variable value being passed.
* Reference Parameters -- Use to ref keyword to pass variable as reference and not as value, which means the value address and not a copy of the value is passed to the method. It also returns a value.
* Out Parameters -- Use when you want a method to return more than one value
* If a parameter is declared with the out modifier, then in the body of the method some value must be assigned to this parameter. Otherwise, there will be a compilation error. If the parameter is declared with the modifier ref, then it is not necessary to assign value to this parameter in the method.
* Parameters Arrays -- The params keyword lets you specify a method parameter that takes a variable number of arguments. Params keyword should be the last one in a method declaration, and only one params keyword is permitted in a method declaration. Passing the parameter is optional. Other you can pass as many elements/values as desire.
* Function chaining -- You can create method that returns instance of the class using return type and out parameter.
* Method signature includes name, parameter type, kind (out) and number of parameters. It doesn't include return type and params modifier

**Class**: defines behaviors and characteristics of any logical entity in a reusable and extensible way. An instance of a class is called object.

* The 4 pillars of OOP are inheritance, encapsulation, abstraction, polymorphism

Constructor

* Constructor can take parameter
* Static constructor is called before instance constructor.
* Use static constructor to initialize static field.
* Static constructor does not allow access modifier. It is called automatically before you called any member of that class
* A construct can inherit another construct by using the semi colon and the “this” keyword
* Doesn't return a value
* A static constructor is called only once whereas instance constructor is called everyone that object is created
* You can use the object initializer syntax to initialize properties

Destructor

* Cleans up the resources used by a class instance
* Don't take any parameters
* Don't have to be called as they are called automatically by the garbage collector

Static Class

* Cannot be instantiated. You access the members of a static class by using the class name itself.
* Contains only static members
* Cannot be instantiated
* Is sealed -- cannot be inherited
* Cannot contain instance constructors
* No static class can have static members
* Constant field operates like static

Fields

* should be set to private and assign value via properties
* Field values can be set via constructor or properties.

Properties

* Used to set value to a field and return a value from a field
* A property with only get assessor is a read only property
* A property with only set assessor is a write only property
* The set assessor has a value field that receives value being passed by client and that value key word should be used in the property
* Languages with no properties like python uses getter and setter methods
* You can use auto-implemented properties if no requirement is needed to set a value to or return a value from a property.
* You can use the refactor feature in C# to encapsulate your fields. Highlight he field and right click and then set refactor

**Inheritance**: one of the principles of object-oriented programming, which enables class to see the program code of another (base) class, supplementing it with its own implementation details.

* Use the base keyword to specify the constructor you want to call
* Parent method is called before child method
* Use new keyword to hide inherited method with same name
* Use base keyword or type cast a child type to parent type or reference base variable and instantiate with child object to get access to hidden method in base class
* Taking advantage of polymorphism is impossible without inheritance.
* Base class elements that are declared with protected, public, internal, and protected internal access modifiers are available.
* All elements of the base class that are declared with the private modifier are not available in the derived class.
* When a class item is protected, it is available within the class in which its declared, as well as in inherited classes. It's not available from class instance.
* When a class item is protected internal, it's available in the inherited within the same assembly
* The allowance of explicit multiple class inheritance could lead ambiguity, which is called diamond problem
* C# achieves multiple class inheritance implicitly by inheriting multiple interfaces and create interface or class reference variable and instantiate those reference variable with class objects that implemented those interfaces

**Polymorphism:** one of the principles of object-oriented programming that provides the ability to a class to have multiple implementations with the same name. There are two types of polymorphism: static (Early Binding) and dynamic (Late Binding). As result of polymorphism method usage can be extended

* Method overriding is one form of polymorphism (virtual - new/override concept)
* Method overloading is another form of polymorphism (creating method with the same name but different method signature)

**Versioning -- allows backward compatibility**

* If the method in the derived class is not preceded by new or override keywords, the compiler will issue a warning and the method will be behave as if the new keyword were present.
* If the method in the derived class is preceded with the new keyword, the method is defined as being independent of the method in the base class.
* If the method in the derived class is preceded with the override keyword, objects of the derived class will call that method instead of the base method.
* In order to apply the override keyword to the method in the derived class, the base class method must be defined virtual.
* The base class method can be called from within the derived class using the base keyword.

**NB** -- a base class variable object can be instantiated with its derived class but a derived class variable object cannot be instantiated with a its base class without explicit conversion at the time of instantiation: parent child relationship.

**Encapsulation:**  the idea of bundling data and methods in one unit such as a class or struct. This concept is also used to hide the internal state of an object from client

**Struct**

* Like classes structs can have fields, properties, constructors and methods
* Struct acts like a sealed class. So it can't be inherited

**Struct vs Class**

* Struct is a value whereas class is a reference type
* Structs are stored on stack, while class reference variables are stored on the stack, but object stored on the heap
* Value types hold their values in memory where they are declared, but reference types hold a reference to an object in memory
* Value types are destroyed immediately after the scope is lost, whereas for reference types only the reference variable is destroyed after the scope is lost. The object is later destroyed by garbage collector.
* When you copy a struct into another struct, a new copy of that struct gets created and modifications on one struct will not affect the values contained by the other struct.
* When you copy a class into another class, we only get a copy of the reference variable. Both the reference variables point to the same (address) object on the heap. So, operations on one variable will affect the values contained by the other reference variable.
* Structs can't have destructors, but classes can have destructors.
* Struts cannot have explicit parameter less constructor whereas a class can
* Structs can't inherit from another class whereas a class can. Both can inherit an interface

**Interfaces**

* Interfaces are created using the interface keyword. Just like classes interfaces also contains properties, methods, delegates or events, but only declarations and no implementations. A compile time error will be thrown for provide implementation for any interface member.
* Interface members are public by default, and they don't allow explicit access modifiers.
* If a class or struct inherits from an interface, it must provide implementation for all interface members. Otherwise, a compile error is thrown.
* Interfaces cannot contain fields
* A class or a struct can inherit from more than one interface at the same time, but a class cannot inherit from more than one class at the same time.
* Interfaces can inherit from other interfaces. A class that inherits this interface must provide implementation for a implementation for all interface members in the entire interface inheritance chain.
* We cannot create an instance of an interface, but an interface reference variable can point to a derived class object.
* One drawback of interface inheritance is that if two interfaces are inherited with bit having the same name for a method declaration, the inheriting class wouldn't know which is being called. To get around that wither rename the one or use the interface name when implementing the method then either:
  + Use type casting when calling the methods, effect convert it variable from a reference class object/variable to interface reference variable.
  + Declare the interface reference variable and instantiate it with the class object.
  + Make one default by implementing it without using the interface name (implicit), and implement the other using the interface name (explicit)
* When a class explicitly implements an interface member, the interface member can no longer be accessed thru class reference variable, but only thru the interface reference variable.
* Cannot use access modifier when implementing interface methods.

**Abstract Class**

* Like interfaces, abstract classes cannot be instantiated. They can only be used as base class for other classes
* Non abstract that inherit classes must provide the implementation for inherited abstract members.
* Abstract class cannot be sealed
* Abstract class contain abstract members, not mandatory.

**Abstract vs Interfaces**

* Abstract classes can have implementations for some of its members (methods), but interfaces can't have implementation for any of its members.
* Interfaces cannot have fields while abstract classes can.
* An interface can inherit another interface only and cannot inherit an abstract class, whereas an abstract class can inherit another abstract class or another interface.
* A class can inherit multiple interfaces at the same time, whereas a class (abstract or otherwise) cannot inherit multiple classes at the same time.
* Abstract classes can have access modifiers while interface members cannot.

**Delegates** -- A type safe function pointer

* Delegates are reference type similar to class
* Use the delegate key word to create a delegate
* Delegate can point to a function with the same signature
* Anytime you want to pass a function as parameter think of delegate
* Lambda expression uses delegate
* A multicast delegate is a delegate that has reference to more than on function. When you invoke a multicast delegate, all the functions the delegate is pointing to are invoked.
* There are 2 approached to create a multicast delegate. Depending on the approach you use:
  + + or += to register a method with the delegate
  + -Or -= to unregister a method with the delegate
* A multicast delegate invokes the methods in the invocation list in the same order in which they are added
* Where do you use multicast delegates? Multicast delegate makes implementation of observer design very simple. Observer pattern is also called as publish/subscribe pattern.
* If the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Similarly, if the delegate has an out parameter, the value of the output parameter, will be the value assigned by the last method.

**Exception Handling**

* An exception is an unforeseen error that occurs when a program is running
* Unhandled exception could provide useful info hackers; shows users error they don't understand, and resources won't get closed out.
* An exception is a class that derives from System.Exception class. The System.Exception class has several useful properties that provide valuable information about the exception.
* Create specific exception first then general exception
* Use the Throw New plus the Exception class to create new exceptions

**Inner Exception**

* The InnerException property returns the Exception instance that caused the current exception
* To retain the original exception, pass it as a parameter to the constructor of the current exception
* Always check if inner exception is not null before accessing any property of the inner exception object, else you may get Null Reference Exception
* To get the type of InnerException use GetType() method.

**Custom Exception**

* Create custom exception if none of the existing dotnet exceptions adequately describes the problem
* To find all the exceptions in .NET go to Debug and click Exceptions
* Best to inherit the Exception so that you don't have to create the exception methods and functionality

**Enums**

* Value type
* If a program uses set of integral numbers, consider replacing them with enums, which makes the program more readable and maintainable.
* Enums are enumerations
* Strongly typed constants. Hence, an explicit cast is needed to convert from enum type to an integral type and vice versa. Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
* The default underlying type of an enum is int
* The default value for first element is zero and gets incremented by 1
* It is possible to customize the underlying type and values.
* Enum keyword (all small letters) is used to create enumerations, whereas Enum class, contains static GetValues() and GetNames() methods which can be used to list Enum underlying type values and Names.

**Types vs Type Members**

* In general classes, structs, enums, interfaces and delegates are called as types and fields, properties, constructors, methods, etc., that normally reside in a type are called type members.
* In C#, there are 5 access modifiers: private, protected, internal, protected internal and public.
* Type members can have all access modifiers whereas types can have only 2 (internal and public) out of the 5.

NB)

* You can use #region to group sections of your code. This will allow you to hide and unhide sections of your code (toggle), making it easier for you to focus on section of the code you want to work on.
* Visual Studio short cut: Edit -- Outlining -- Toggle all outlining

**Access Modifiers**

1. Private members are available only within the containing type
2. Public members are available anywhere. There are no restrictions
3. Protected members are available within the containing type and the type derived from the containing type. Base and this keyword can be used to access protected members in the derived class.
4. Internal members are available within the containing assembly.
5. Protected Internal members are available within containing assembly and from within derived class in any other assembly.

* If an access modifier is not specified for a type, the default will be internal whereas for a type member, the default will be private.

**Attributes**

* Allows you to add declarative information to your program by adding attributes on certain entities such as class, method, properties, etc. This information can then be queried at runtime using reflection
* There are several pre-defined attributes provided by .NET. It is also possible to create your own custom attributes.
  + Obsolete - mark types and types members outdated
  + webMethod - to expose a method as XML webservice method
  + Serializable - indicates that a class can be serialized
* It is possible to customize the attribute using parameters
* An attribute is a class that inherits from System.Attribute base class.

**Reflection** -- Gives the ability to inspect an assembly's metadata (properties, methods, etc.) at runtime. It is used to find all types in an assembly and/or dynamically invoke methods in an assembly.

* Uses of reflection
  + When you drag and drop a button on win forms or asp.net application, the properties window uses reflection to show all the properties of the Button class. So reflection is extensively used by IDE or UI designers.
  + Late binding can be achieved by using reflection. You can use reflection to dynamically create an instance of a type for which we don't have any information at compile time. So, reflection allows you to use code that is not available at compile time.
  + Consider an example where we have two alternate implementations of an interface. You want to allow the user to pick one or the other using a config file. With reflection, you can simply read the name of the class who implementation you want to use from the config file and instantiate an instance of that class. This is another example of late binding using reflection.
* Reflection uses a special class called Type that is located in the reflection namespace. Or use the typeof keyword

**Generics**

* Introduced in C# 2.0. Generics allow us to design classes and methods decoupled from the data types
* Generics classes are extensively used by collection classes available in System.Collections.Genric namespace.
* if you want to build method that takes any type for parameter, use object type when declaring the parameters. In order you want the method to be independent of the data type.
  + If affects performance because object type needs to be unboxed
  + You lose the strongly type advantages
* Instead of using object type use generics to achieve decoupling of method from data type while not suffering from the drawbacks of using object type.

**NB)**

* It's advisable to override the ToString method in custom type with some useful information
* It's advisable to override the Equals method in custom type with logics that check to make sure same types are being compared, type is not null, and a comparison is set up with properties you want use to determine if the custom types are equaled.
* It's advisable to override GetHashCode method when you override Equals method.
* On a null object you cannot call instance method such as ToString, equals without getting an exception
* Convert.ToString() handles null -- returns an empty string; while ToString() doesn't -- throws a null reference exception.
* System.String is immutable. Once created it can't be changed. StringBuilder is mutable and offer better performance than string objects of type System.String when heacy string manipulation is involved.

**Partial Classes**

* Partial classes allow us to split a class into 2 or more files. All these parts are then combined into a single class when the application is compiled. The partial keyword can also be used to split a struct or an interface over two or more files.
* All the parts spread across different files must use the partial keyword.
* All the parts spread across different files must have the same access modifiers.
* If any of the parts declared abstract, then the entire type is considered abstract.
* If any of the parts declared sealed, then the entire type is considered sealed.
* If any of the parts inherits a class, then the entire type inherits that class.
* C# doesn't support multiple class inheritance. Difference parts of the partial class must not specify different base classes.
* Different parts of the partial class can specify different base interfaces.
* Any member declared with partial definition is available to other parts of the partial class.
* Partial methods are private by default
* Partial methods cannot have all the other modifiers such as override etc.
* Signature of the partial method must be the same as the signature of the implementation
* A partial method must be declared within a partial class or struct. A non-partial class or struct cannot include partial methods.
* A partial method can be implemented only once. Trying to implement a partial method more than once will raise a compile time error.

**Indexers**

* Indexers allow instances of a class to be indexed just like arrays
* To create Indexer:
  + Use "this" keyword to create an indexer
  + Just like properties indexers have get and set accessors
  + Indexers can be overloaded

**Optional Parameters**

* There are 4 ways to make method parameters optional
  + Use parameters arrays
  + Method Overloading
  + Specify parameter defaults -- must be the last parameter in a formal parameter list
  + Use OptionalAttribute present in System.Runtime.InteropServices namespace

**Dictionary**

* A dictionary is the collection of key-values pairs
* Dictionary provides fast lookups for values using keys
* Keys must be unique

**List**

* List is of the generic collection classes present in System.Collection.Generic namespace
* Unlike arrays, lists a grow automatically
* The class also provides methods to search, sort, and manipulate lists
* Contains function - checks if an item exists in the list. This method returns true if the item exists.
* Exists function - checks if an item exists in the list based on a condition. This method returns true if the item exists.
* Find function - searches for an element that matches the condition defined by the specified lambda expression and returns the first matching item from the list
* FindLast function - searches for an element that matches the conditions defined by specified lambda expression and returns the last matching item from the list
* FindAll function - return all the items from the list that matches the conditions specified by the lambda expression.
* FindIndex function - returns the index of the first item that matches the condition specified by the lambda expression. There are 2 other overloads of this methods which allows us to specify the range of elements to search within the list.
* FindLastIndex function - returns the index of the last item that matches the condition specified by the lambda expression. There are 2 other overloads of this method that allows us to specify the range of elements to search within the list
* Convert an array to a list use ToList method
* Convert a list to an array use ToArray method
* Convert a list to a dictionary use ToDictionary method
* AddRange - Add method allows you to add one item at a time to the end of the list, whereas AddRange allows you to add another list of items to the end of the list.
* GetRange - Using an item index, we can retrieve only one item at a time from the list. if you want to get a list of items from the list, then use GetRange function. This function expects 2 parameters i.e., the start index in the list and the number of elements to return.
* InsertRange - Insert method allows you to insert a single item into the list at a specified index, whereas InsertRange allows you to insert another list of items to your list at the specified index.
* RemoveRange - Remove function removes only the first matching item from the list. RemoveAt function removes the item at the specified index in the list. RemoveAll function removes all the items that match the specified condition. RemoveRange method removes a range of elements from the list. This function expects 2 parameters, which are the start index in the list and the number of elements to remove. If you want to remove all the elements from the list without specifying any condition, then use clear method.

**Sorting a list of complex types**

* The Sort functionality works for simple types like int, string, char etc. because they implemented the IComparable interface already. Complex types need to implement the Icomparable inteface in order to have that functionality.
* To sort a list of complex types without using LINQ, the complex types have to implement IComparable interface and provide implementation for CompareTo method. CompareTo method returns an
* integer and the meaning of the return value is shown:
  + Greater than zero - the current instance is greater than the object being compared with.
  + Less than zero - the current instance is less than the object being compared with
  + Is zero - the current instance is equal to the object being compared with
* Use comparison delegate with List<T>
  + One of the overloads of the Sort method in List class expects comparison delegate to be passed as an argument: public void Sort(comparison<T> comparison)
    1. Create function whose signature matches the signature of System.Comparison delegate
    2. Create an instance of System.Comparison delegate and then pass the name of the function created in Step 1 as the argument. So, at this point "Comparison" delegate is pointing to our function that contains the logic to compare 2 customer objects.
    3. Pass the delegate instance as an argument to Sort function

**Useful methods of List Collection class**

* TrueForAll - Returns true or false depending on whether every element in the list matches the conditions defined by the specified predicate.
* AsReadOnly - Returns a read-only wrapper for the current collection. Use this method if you don't want the client code to modify the collection i.e., add or remove any elements from the collection. The ReadOnly collection will not have methods to add or remove items from the collection. You can only read items from the collection.
* TrimExcess - Sets the capacity to the actual number of elements in the List, if that number is less a threshold value.

**When to use Dictionary over List**

* Find method of the list class loops through each object in the list until a match is found. So, if you want to lookup a value using a key dictionary is better for performance over list. Hence, use the dictionary when you know the collection will be primarily used for lookups.

**Generic Queue Collection Class**

* A generic FIFO (First In First Out) collection class that is present in System.Collections.Generic namespace.
* To add items to the end of the queue use Enqueue method
* To remove an item at the beginning of the queue use Dequeue method
* To check if an item exists in the queue use Contains method
* Dequeue method removes and returns the item at the beginning of the queue whereas Peek returns the item at the beginning without removing it.

**Generic Stack Collection class**

* Stack is a generic LIFO (Last In First Out) collection class that is present in System.Collections.Genric namespace.
* To insert an item at the top of the stack use Push method
* To remove and return the item that is present at the top of the stack use Pop method
* To check if an item exists in the stack use Contains method
* Pop method removes and returns the item at the top of the stack whereas Peek returns the item at the top without removing it.

**Multithreading in C#**

**What is a Process**

* Process is what the operating system uses to facilitate the execution of a program by providing the resources required. Each process has an unique process ID associated with it. You can view the process within which a program is being executed using windows task manager.

**What is Thread**

* Thread is a lightweight process. A process has at least one thread which is commonly called main thread, which executes the application code. A single process can have multiple threads.
* All threading relates classes are located in System.Threading namespace.

**Advantage of Multi-Threading**

* To maintain a responsive user interface
* To make efficient use of processor time while waiting for I/O operations to compete
* To split large CPU -bound tasks to be processed simultaneously on a machine that has multiple processors/cores

**Disadvantages of Multi-Threading**

* On a single processor/core machine threading can affect performance negatively as there is overhead involved with context switching
* Have to write more lines of codes to accomplish the same task
* Multithreading applications are difficult to write, understand and maintain.

**ThreadStart Delegate**

* To create a THREAD, create an instance of Thread class and to it's constructor pass the name of the function that we want the thread to execute.

**Why a delegate need to be passed as parameter to the Thread class constructor?**

* The purpose of creating a Thread is to execute a function. A delegate is a type safe function pointer, meaning it points to a function that the thread has to execute. In short, all threads reuire an entry point to start execution. Any thread you create will need an explicitly defined entry point i.e. a pointer to the function where they should begin execution. So threads always reuire a delegate.

ParameterizedThreadStart Delegate

* Use ParameterizedThreadStart delegate to pass data to the thread function.

**N.D)** You can also not explicitly create the ThreadStart and ParameterizedThreadStart delegates and the compile implicitly create one.

**Passing data to the Thread Function**

* To pass data to the Thread function in a type safe manner, encapsulate the thread function and the data it needs in a helper function class and use the THreadStart delegate to execute the thread function.

**Retrieving Data from Thread function using Callback Method**

* Create a callback delegate. The actual callback method signature should match the signature of this delegate.

**Thread.Join & Thread.IsAlive Functions**

* Join blocks the current thread and makes it wait until the thread on which join method is invoked completes. Join method also has an overload where we can specify the timeout. If we don't specify the timeout the calling thread waits indefinitely, until the thread on which join() is invoked completes. This overloaded join(int millisecondsTimeout) method returns boolean. True if the thread has terminated otherwise false.
* Join is particularly useful when we need to wait and collect result from a thread execution or if we need to do some clean-up after has completed.
* IsAlive returns booleans. True if the thread is still executing otherwise false.

**What happens if shared resources are not protected from concurrent access in multithreaded program?**

* The output or behavior of the program can become inconsistent if the shared resources are not protected from concurrent access in multithreaded program.

**Which option is better (lock vs Interlock?**

* From a performance perspective using interlocked class is better than using locking.
* The Interlocked class can be used with addition/subtraction (increment, decrement, add, etc.) on and int or long field. The Interlocked class has methods for incrementing, decrementing, adding and reading variables atomically.

**Monitor vs Lock**

* Both monitor class and lock provide a mechanism to synchronize access to objects.
* Lock is the shortcut for Monitor.Enter with try and finally.
* Lock is a shortcut and it's the option for basic usage.
* If you need more control to implement advanced multithreading solutions use TryEnter, Wait, Pulse and Pulse methods, which are found in the Monitor class.

**Resolving Deadlocks**

* Acquiring locks in a specific defined order
* Mutex class
* Monitor.TryEnter Method

**How to find out how many processors are on your machine**

* Use Task Manager and to go Performance. The number of green box is the number of processors on your machine
* Use the following code in any .net application: Environment.ProcessorCount
* In windows command prompt type the following: echo %NUMBER\_OF\_PROCESSORS%

**Anonymous Methods**

* Anonymous method is a method without a name. Introduced in C# 2, they provide us a way of creating delegate instances without having to write a separate method.

Lambda Expression

* Anonymous methods and lambda expressions are very similar. Lambda expressions was introduced in C# 3.
* => is called lambda operator and read as GOES TO. Notice that with a lambda expression you don't have to use the delegate keyword explicitly and don't have to specify the input parameter type explicitly. The parameter type is inferred. Lambda expressions are more convenient to use than anonymous methods. Lambda expressions are particularly helpful in writing LINQ query expressions.

Reference:

* Edx.org
* Microsoft C# Documentation
* KudvenKat YouTube videos
* BestProg.net