C# Questions and Terms

# Q&A

## What is C# ?

A modern, Object Oriented programming language developed by Microsoft. Designed for the CLI, which is a runtime execution environment that can be targeted by various high level languages.

## What is a Jagged Array ?

An array of arrays.

int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};

scores[0] is an ary of three integers, scores[1] an ary of four int

## How many ways can you pass parameters to a method ?

* Value
  + A copy of the arguments; changes have no effect on original param value.
* Reference
  + Reference to the address of var; changes effect original value
* Output
  + To return one or more values from method

## Can you return mult values from a C# function ?

Yes, you can return one value from the function, but also use Output params.

## What is the difference between Ref and Output parameters ?

They are similar, but Output params pass data out of the method, whereas Ref params pass in the memory location of the argument.

## What is a namespace ?

It is designed to keep names separate from one another. The class names in one namespace will not conflict with class names in another namespace.

## What is the purpose of a using statement ?

To include a namespace in a program.

## What are values types in c# ?

* Derived from System.ValueType, and can be assigned a value directly.
* The system allocates memory to store the values (***primitive types:*** int, float, bool, char)
* ***Struct*** is a value type (a struct is a cut-down class which doesn’t support inheritance)

## What are reference types in c# ?

* It contains a memory reference to a variable, not the actual stored data
* Examples of built-in reference types are: object, dynamic, and string
* Classes are ref types

## Which object acts as the base class for all types ?

Object type, an alias for System.Object class

## What is boxing and unboxing?

Boxing is the process of converting a value type to an object type.

int i = 123;

// The following line boxes i.

object o = i;

o = 123;

i = (int)o; // unboxing

## Dynamic type vs. Object type

Similar execpt that **object types are checked at compile time**, whereas dynamic types are checks at runtime.

# What is an Object ?

* An object is a class or struc which defines what the type can do
* They are also called instances, and many instances of the same class can be created

# Terms

## Abstract Classes

\*\*\* see “Generics vs. Abstract” below

## Access Modifiers/Specifiers

* Keywords to specify the accessibility of a type and its members
* 5 modifiers:
  + Public
    - Access to members is not restricted
  + Private
    - Access to members is limited within the defined type
    - Members not accessible to any other classes
  + Internal
    - Class members are accessible in all classes within that assembly
  + Protected
    - Members are only visible when inherited by a child; otherwise member and types are not visible to other classes
  + Protected internal
    - A combination of protected and internal
    - Accessible within the same assembly (Internal)
    - Accessible only when inherited (Protected)

## Binary operators

+, -, !, ~, ++, --

## Encapsulation

* The ability to hide behavior and data from its user
* It enables a group of properties and methods to be considered a single unit/object
* It restricts access to the members of a class to prevent things like data corruption
* 5 levels of access using Access Modifiers: public, private, protected, internal, prot.-internal

## Event Delegate

* These are event handlers in c#
* You are passing a ***reference*** to a method
* A delegate is a reference type, but it ***refers to a method*** rather than an object.
* The delegate is “***encapsulating a method***”
  + public delegate int ***FindResult***(object obj1, object obj2);
  + ***FindResult*** will encapsulate any method that defines two object parameters
  + You can encapsulate a member method with that delegate by creating an instance of that delegate, passing in a method that matches the return type and signature.
  + The class that declares the delegate is decoupled from the class that uses the delegate – i.e. FindResult does not need to know what class uses the delegate, only that an ***int*** should be returned

## Function Overloading

* More than one signatures for the same func/method name
* You cannot overload a func only by a different return type

## Function Override

* In a derived class, a method can have the same name as the method in the base class
* Two ways to override:
  + ***Override*** keyword – extends the base class method
  + ***New*** keyword – hides the base class method

## Generics vs. Abstract

Abstract classes/interfaces are used for generalization of common functionality of a group of entities. For Generics, you might have a very similar implementation of something where the only difference is the object type (i.e. creating a List to store elements, but without knowing ahead of time what type of object to store).

### Generics - http://www.tutorialspoint.com/csharp/csharp\_generics.htm

* Used to write a class or method that can work with any data type
* You write the method with substitute parameters for data types –
  + public class MyGenericArray<T>
* The compiler will generate code to handle the specific data type

### Abstract Classes - http://csharp.net-tutorials.com/classes/abstract-classes/

* Used to define a base class in the hierarchy
* You can only have one Abstract Class in your hierarchy (but many Interfaces)
* It can have constructors (unlike an Interface)
* You cannot create an instance of them (throws excep.); you must subclass them first

abstract class ***FourLeggedAnimal***

{

public virtual string Describe()

{

return "Not much is known about this four legged animal!";

}

}

class Dog : ***FourLeggedAnimal***

{

}

\*\*\* The class “***Dog***” can be instantiated in your program, but not “***FourLeggedAnimal***”

## Inheritance

* Inheritance, Encapsulation, and Polymorphism are the three pillars of OOP.
* Inheritance allows you to create new classes that reuse, extend, and modify behavior defined in other classes.
* Cannot be instantiated – only derived; the derived class inherits members from base class
* A derived class can only have one direct base class; however inheritance is transitive (i.e. if C is derived from B, and B is derived from A, then C inherits from both B and A)
* The derived class inherits all members of the base class, except for constructors/deconstruct
* If a base class declares a ***virtual method***, the derived class can override it
* If a base class declares an ***abstract method***, the derived class must override the method in any ***non-abstract*** class. If a derived class itself is abstract, it inherits abstract members without implementing them.

## Interface

* Looks like a class but has no implementation; they only provide declarations
* Cannot be instantiated (just like an Abstract class cannot)
* It contains events, indexers, methods and properties (no member data)
* Interfaces are inherited by classes and structs, which provide the implementation
* Interfaces define a contract
* From MSDN: By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes.
* ***Abstract classes*** serve a similar purpose, but are mostly used when only a few methods are declared in the base class and the derived class will implement the functionality.

public interface ***ITransactions***

{

// interface members

void showTransaction();

double getAmount();

}

public class Transaction : ***ITransactions***

{

// implement methods here

}

## Multiple Inheritance (MI)

* C# classes are allowed to inherit only from one single parent class (***single inheritance***)
* C++ allows to inherit from mult classes

## Object Class

The base class of all classes in the .NET Framework; it is the root of the type hierarchy.

## Polymorphism (many shapes)

* In OOP, it’s referred to as “One interface, multiple functions”.
* Allows for redefining methods in derived classes.

Two types:

* Static Poly.
  + The response to a function is determined at compile time
  + It uses ***early binding (static binding),*** which links a func to an object at compile time
    - Two types of static polymorphism:
      * Method overloading
        + More than one signatures for the same func/method name
        + You cannot overload a func only by a different return type
      * Operator overloading
        + You can redefined/overload most built-in c# operators
* Dynamic Poly.
  + The response is determined at runtime
  + Method overriding (involving inheritance and virtual functions) is ***runtime polymer***.

## Stack vs. Heap

* The CLR allocated memory for objects in two places: stack and heap
* Stack:
  + a simply first-in, last-out memory structure, and highly efficient
  + A method pushes data onto the stack when invoked, and pops when method is completed
* Heap:
  + a random collection of objects
  + can be allocated/deallocated in random order)
  + requires the overhead of memory manager and garbage collector

## Struc

* cut-down classes
* no inheritance or **finalizers** (finalizers try to free resources and perform cleanup)
* defined the same way as classes except with ‘struct’

***struct*** Point{

private int x, y;

public Point(x,y){

this.x = x;

}

}

## Unary Operators

Operate on only one operand in an expression.

* \* (multiplication, indirection)
  + multiplication operator (**\***), which computes the product of its operands
  + the dereference operator, which allows reading and writing to a pointer
* & (address of)
* + (Unary plus)
* - (Unary negation)
* ! (logical negation)
* ~ (one’s complement)
* ++ (prefix increment)
* -- (prefix decrement)

# Data Structures

## Array

* A simple list of objects, all of the same type.
* Contains a specific array size.

## ArrayList

* Dynamic array, meaning it can have any amount of objects and of any type.
* It doubles in size every time it runs out of space, which reduces the amount of element-copying in the long run

## List<>

* Part of generic collections
* It’s a version of the ArrayList, however with the same type

List<int> intList = new List<int>();

intList.Add(45);

## LinkedList<>

* A series of object which have their references indexed (like Arrays)
* Objects link to each other in nodes
* LinkedList nodes have three values:
  + The object’s value
  + Reference to next node
  + Reference to previous node

## Dictionary

HashTable

HashSet

Stack

Queue