115

CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - II

Sub:	C# with .NET							Code:	10CS761
Date:	04/11/2016	Duration:	90 mins	Max Marks:	50	Sem:	7	Branch:	CSE/ISE

Answer Any FIVE FULL Questions

#	Questions	Marks	OBE	
			CO	RBT
1	List different methods of **Sytem.Exception** class? With program explain the use of **try, catch and finally** keyword?	10	CO2	L1
2	With program illustrate **generic exception, finally block, multiple catch and inner exception**?	10	CO2	L3
3	With program illustrate the use of **interface as parameter and return values**?	10	CO4	L3
4	Difference between **interface and abstract class**? Write a note on **IEnumerable and IEnumerator** interfaces?	10	CO4	L2
5	Write program explain **ICloneable, IComparer and IComparable** interfaces?	10	CO4	L5
6	Write a note on basics of **object lifetime and object generations**?	10	CO2	L2
7	With Program explain **finalizable and disposable** objects in object lifetime?	10	CO2	L3
8	Write a note on delegate concept?	10	CO4	L2

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1:	Explain structure of C# application.	-	-	1	2	2	-	-	1	-	-	-	2
CO2:	Explain C# Language Fundamentals, the role of exception handling and the basics of Garbage Collection process	1	1	2	2	-	-	-	-	-	-	-	-
CO3:	Develop simple C# applications using Object Oriented features.	-	1	1	1	-	-	-	-	-	-	-	3
CO4:	Explain Interfaces, Delegates and Events using simple programs.	-	1	1	-	-	-	-	-	-	-	1	2
CO5:	Exploring System. Collections and namespaces to build custom container	1	1	1	-	2	-	-	-	-	-	-	1
CO6:	Design and Develop Mobile Application using .NET Assembles	1	1	2	2	2	-	-	-	-	-	-	2

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*

1. List different methods of **Sytem.Exception** class? With program explain the use of **try, catch and finally** keyword?

System.Exception Property	**Meaning in Life**
Data	This property retrieves a collection of key/value pairs (represented by an object implementing IDictionary) that provides additional, programmer-defined information about the exception. By default, this collection is empty (e.g., null).
HelpLink	This property returns a URL to a help file or website describing the error in full detail.
InnerException	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur. The previous exception(s) are recorded by passing them into the constructor of the most current exception.
Message	This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter.
Source	This property returns the name of the assembly that threw the Exception.
StackTrace	This read-only property contains a string that identifies the sequence of calls that triggered the exception. As you might guess, this property is very useful during debugging if you wish to dump the error to an external error log.
TargetSite	This read-only property returns a MethodBase type, which describes numerous details about the method that threw the exception (invoking ToString() will identify the method by name).
InnerException	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur.

2. With program illustrate **generic exception, finally block, multiple catch and inner exception**?

```
static void Main(string[] args)
{
Console.WriteLine("***** Handling Multiple Exceptions *****\n");
Car myCar = new Car("Rusty", 90);
myCar.CrankTunes(true);
try
{
// Speed up car logic.
}
catch(CarIsDeadException e)
{
// Process CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
// Process ArgumentOutOfRangeException.
}
catch(Exception e)
{
// Process any other Exception.
}
finally
{
// This will always occur. Exception or not.
myCar.CrankTunes(false);
}
Console.WriteLine();
}
```

3. With program illustrate the use of **interface as parameter and return values**?

Explicit interface implementation can also be very helpful whenever you are implementing a number of interfaces that happen to contain identical members. For example, assume you wish to create a class that implements all the following new interface types:

```
public interface
{
void Draw();
}
public interface IDrawToPrinter
{
void Draw();
}
```

If you wish to build a class named SuperImage that supports basic rendering (IDraw), 3D rendering (IDraw3D), as well as printing services (IDrawToPrinter), the only way to provide unique implementa

implementation:

```
// Not deriving from Shape, but still injecting a name clash.
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
void IDraw.Draw()
{ /* Basic drawing logic. */ }
void IDrawToPrinter.Draw()
{ /* Printer logic. */ }
void IDraw3D.Draw()
{ /* 3D rendering logic. */ }
}
```

4. Difference between **interface and abstract class**? Write a note on **IEnumerable and IEnumerator** interfaces?

Consider an example below to explain the use of IEnumerable and IEnumerator:

```
//Car is a container of car objects……
Public class Car
{
Private Car[] CarArray;
//Create some car objects upon startup…….
Public Cars()
{
CarArray = new[4];
CarArray[0] = new Car("FeeFee", 200, 0);
CarArray[1] = new Car("Clunker", 300, 0);
CarArray[2] = new Car("Zippy", 30,0);
}
}
```

Below method is defined by the IEnumerable interface type:

```
Public interface IEnumerable
{
IEnumerable GetEnumerator();
}
//IEnumerable defines a single method
Public IEnumerable GetEnumerator ()
{
///Ok, now what……?
}
```

Now, Given that IEnumerable.GetEnumerator() returens an IEnumerator interface, you may update the cars type as shown below:

```
//Getting closer….
Public class cars: IEnumerable, IEnumerator
{
……
//Implementing an IEnumerable……
Public IEnumerator GetEnumerator()
```

```
{
Return(IEnumerator) this;
}
}
```

5. With program explain ICloneable, IComparer and IComparable interfaces?

System.Objectdefines a member named MemberwiseClone (). This method Object users do not call this method directly (as it is protected); however, a given object may call this method itself during the *cloning* process. To illustrate, assume you have a class named Point:

```
// A class named Point.
public class Point
{
// Public for easy access.
public int x, y;
public Point(int x, int y) { this.x = x; this.y = y;}
public Point(){}
// Override Object.ToString().
public override string ToString()
{ return string.Format("X = {0}; Y = {1}", x, y ); }
}
```

Given what you already know about reference types and value types, you are aware that if you assign one reference variable to another, you have two references pointing to the same object in memory. Thus, the following assignment operation Point object on the heap; modifications using either reference affect the same object on the heap:

```
static void Main(string[] args)
{
// Two references
Point p1 = new Point(50, 50);
Point p2 = p1;
p2.x = 0;
Console.WriteLine(p1);
Console.WriteLine(p2);
}
```

When you wish to equip your custom types to support the ability to return an identical copy of itself to the caller, you may implement the standard ICloneable interface. This type defines a single method named Clone():

```
public interface ICloneable
{
Object Clone();
}
```

The System.IComparable interface specifies a behavior that allows an object to be sorted based on some specified key. Here is the formal definition:

```
// This interface allows an object to specify its
// relationship between other like objects.
public interface IComparable{
int CompareTo(object o);
}
{
...
// IComparable implementation.
int IComparable.CompareTo(object obj){
Car temp = (Car)obj;
if(this.carID > temp.carID)
return 1;
if(this.carID < temp.carID)
return -1;
else
return 0;
}
}
```

6. Write a note on basics of object lifetime and object generations?

In C# with .NET programming language, programmers never directly deallocate an object from memory. Instead, .NET objects are allocated onto a region of memory termed "*Managed Heap",*

where they will be automatically deallocated by the runtime at "some time" in the feature. The garbage collector removes an object from the heap when it is *unreachable* by any part of your code base.

When the CLR is attempting to locate unreachable objects, is does *not* literally examine each and every object placed on the managed heap. Obviously, doing so would involve considerable time, especially in larger (i.e., real-world) applications. To help optimize the process, each object on the heap is assigned to a specific "generation."The idea behind generations is simple: The longer an object has existed on to stay there. For example, the object implementing Main() will be in memory until the program terminates. Conversely, objects that have been recently placed on the heap are likely to be unreachable rather quickly (such as an object created within a method scope). Given these assumptions, each object on the heap belongs to one of the following generations:

- *Generation 0*: Identifies a newly allocated object that has never been marked for Collection.
- *Generation 1*: Identifies an object that has survived a garbage collection (i.e., it was marked for collection, but was not removed due to the fact that the sufficient heap space was acquired)
- *Generation 2*: Identifies an object that has survived more than one sweep of the garbage collector The garbage collector will investigate all generation 0 objects first. If marking and sweeping these objects results in the required amount of free memory, any surviving objects are promoted to generation 1. To illustrate how an object's generation affects the collection process.

7. With Program explain finalizable and disposable objects in object lifetime?

The role of the Finalize () method is to ensure that a .NET object can clean up unmanaged resources when garbage collected. Thus, if you are building a type that does not make use of unmanaged entities (by far the most common case), finalization is of little use. In fact, if at all possible, you should design your types to avoid supporting finalize () method for the very simple reason that finalization takes time.

- When you allocate an object onto the managed heap, the runtime automatically determines whether your object supports a custom Finalize () method. If so, the object is

marked as *finalizable* and a pointer to this object is stored on an internal queue named the *finalization queue*. The finalization queue is a table maintained by the garbage collector that points to each and every object that must be finalized before it is removed from the heap.

- When the garbage collector determines it is time to free an object from memory, it examines each entry on the finalization queue, and copies the object off the heap to yet another managed structure termed the *finalization reachable* table (often abbreviated as freachable, and pronounced “eff-reachable”).
- At this point, a separate thread is spawned to invoke the Finalize () method for each object on the freachable table *at the next garbage collection.* Given this, it will take at very least *two* garbage collections to truly finalize an object. The bottom line is that while finalization of an object does ensure an object can clean up unmanaged resources, it is still nondeterministic in nature.

Disposable objects
The Main Objective of Disposable Object:

It is an optional, standard way to provide "a method that releases allocated unmanaged resources".
So what does that actually mean? Well, if you're using unmanaged resources,and you keep those resources allocated throughout the lifetime of your object then you should implement IDisposable.Therefore using that the Developers who use this class can then call Dispose if they wish to free these resources early, before your object falls out of scope.
Before doing that,the user should define a class that inherits dispose() method,which is done as follows:
e.g. **class MyDisposableClass:IDisposable**

After this you need to define a method and inside it keep the resource(s) which you want to use regularly without having to call it everytime.

Important Points:

1)The pattern for disposing an object, referred to as a dispose pattern, imposes order on the lifetime of an object. The dispose pattern is used only for objects that access unmanaged resources. This is because the garbage collector is very efficient at reclaiming unused managed objects.

2)Using a Disposable object a Dispose method could be callable multiple times without throwing an exception.

3)Using Dispose() method we dont need to call Close() method explicitly.
Furthermore the main function of using block is to call the dispose() method.

Simple e.g showing the use of Disposable object:

```
using System;
using System.IO;

namespace Disposable
{
class Disp : IDisposable
{
public void Dispose()
{
Console.WriteLine("I am disposed \n");
}
}
public class MyDisposableClass
{


static void Main(string[] args)
{
using (Disp d = new Disp())
{
Console.WriteLine("Inside using block\n");
}
Console.WriteLine("Outside using block\n");
}
}
}
```

Its outputs:

```
Inside using block
I'm disposed
outside using block
using System;
using System.Collections.Generic;
using System.Text;

class Program {
   static void Main(string[] args) {
      using (MyResourceWrapper rw = new MyResourceWrapper()) {
      }
```

```
      MyResourceWrapper rw2 = new MyResourceWrapper();
      for (int i = 0; i < 10; i++)
         rw2.Dispose();
   }
}

public class MyResourceWrapper : IDisposable {
   public void Dispose() {
      Console.WriteLine("In Dispose() method!");
   }
}
```

8. Write a note on delegates?

Historically speaking, the Windows API makes frequent use of C-style function pointers to create entities termed *callback functions* or simply *callbacks*. Using callbacks, programmers were able to configure one function to report back to (call back) another function in the application. The problem with standard C-style callback functions is that they represent little more than a raw address in memory. Ideally, callbacks could be configured to include additional type-safe information such as the number of (and types of) parameters and the return value (if any) of the method pointed to. Sadly, this is not the case in traditional callback functions, and, as you may suspect, can therefore be a frequent source of bugs, hard crashes, and other runtime disasters. Nevertheless, callbacks are useful entities. In the .NET Framework, callbacks are still possible, and their functionality is accomplished in a much safer and more object oriented manner using *delegates*. In essence, a delegate is a type-safe object that points to another method (or possibly multiple methods) in the application, which can be invoked at a later time. Specifically speaking, a delegate type maintains three important pieces of information:

• The *name* of the method on which it makes calls

• The *arguments* (if any) of this method

• The *return value* (if any) of this method

Defining a Delegate in C#

When you want to create a delegate in C#, you make use of the delegate keyword. The name of your define the delegate to match the signature of the method it will point to. For example,

assume you wish to build a delegate named BinaryOp returns an integer and takes two integers as input parameters: **// This delegate can point to any method,**

// taking two integers and returning an

// integer.

public delegate int BinaryOp(int x, int y);

When the C# compiler processes delegate types, it automatically generates a sealed ing from System.MulticastDelegate. This class (in conjunction with its base class, System.Delegate) provides the necessary infrastructure for the delegate to hold onto the list of methods to be invoked BinaryOp delegate using ildasm.exe