Interface and abstract class difference

#### Difference between Abstract Class and Interface

|  |  |
| --- | --- |
| Abstract Class | Interface |
| It contains both declaration and definition part. | It contains only a declaration part. |
| Multiple inheritance is not achieved by abstract class. | Multiple inheritance is achieved by interface. |
| It contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). | It does not contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). |
| It can contain static members. | It does not contain static members. |
| It can contain different types of access modifiers like public, private, protected etc. | It only contains public access modifier because everything in the interface is public. |
| The performance of an abstract class is fast. | The performance of interface is slow because it requires time to search actual method in the corresponding class. |
| It is used to implement the core identity of class. | It is used to implement peripheral abilities of class. |
| A class can only use one abstract class. | A class can use multiple interface. |
| If many implementations are of the same kind and use common behavior, then it is superior to use abstract class. | If many implementations only share methods, then it is superior to use Interface. |
| Abstract class can contain methods, fields, constants, etc. | Interface can only contains methods, properties, indexers, events. |
| It can be fully, partially or not implemented. | It should be fully implemented. |

Virtual and abstract class

**Virtual methods have an implementation and provide the derived classes with the option of overriding it. Abstract methods do not provide an implementation and force the derived classes to override the method** . So, abstract methods have no actual code in them, and subclasses HAVE TO override the method.

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

# Volatile keyword Volatile Class

# The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time. The compiler, the runtime system, and even hardware may rearrange reads and writes to memory locations for performance reasons. Fields that are declared volatile are excluded from certain kinds of optimizations. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution. For more information, see the Volatile class.

Contains methods for performing volatile memory operations.

C#Copy

public static class Volatile

Inheritance

[Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-6.0)

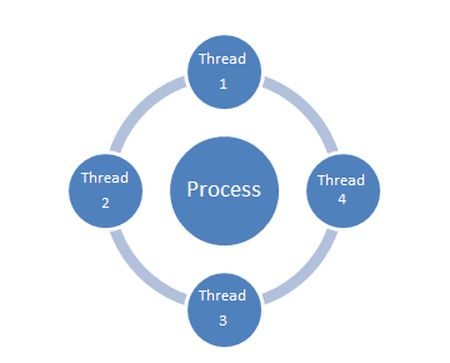
Volatile

How can active multithreading in c# and how can we check its not colabrated

## **Multithreading Overview**

A thread is an independent stream of instructions in a program. A thread is similar to a sequential program. However, a thread itself is not a program, it can't run on its own, instead it runs within a program's context.

The real usage of a thread is not about a single sequential thread, but rather using multiple threads in a single program. Multiple threads running at the same time and performing various tasks is referred as Multithreading. A thread is considered to be a lightweight process because it runs within the context of a program and takes advantage of resources allocated for that program.



We can use **use join () method of thread class** . To ensure three threads execute you need to start the last one first eg T3 and then call join methods in reverse order eg T3 calls T2. join, and T2 calls T1. join.

# Async and await,

Suppose we are using two methods as Method1 and Method2 respectively, and both the methods are not dependent on each other, and Method1 takes a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for the completion of this method, and then it will execute Method2. Thus, it will be a time-intensive process even though both methods are not depending on each other.

We can run all the methods parallelly by using simple thread programming, but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming, but if we use the async and await keywords, we will get the solutions in much less code.

Also, we are going to see more examples, and if any third Method, as Method3 has a dependency of method1, then it will wait for the completion of Method1 with the help of await keyword.

Async and await in C# are the code markers, which marks code positions from where the control should resume after a task completes.

Let’s start with practical examples for understanding the programming concept.

## **Code examples of C# async await**

We are going to take a console application for our demonstration.

**Example 1**

In this example, we are going to take two methods, which are not dependent on each other.

**Code sample**

class Program

{

static void Main(string[] args)

{

Method1();

Method2();

Console.ReadKey();

}

public static async Task Method1()

{

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

// Do something

Task.Delay(100).Wait();

}

});

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

// Do something

Task.Delay(100).Wait();

}

}

}

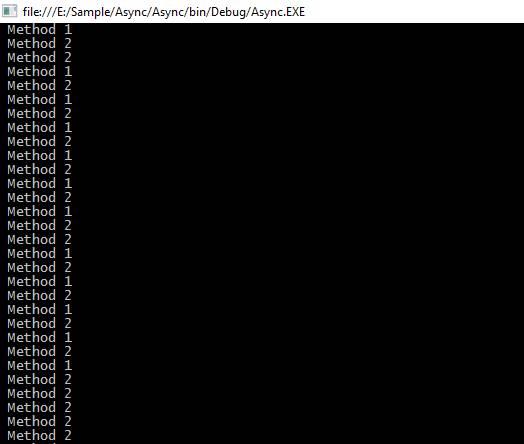
C#

Copy

In the code given above, Method1 and Method2 are not dependent on each other and we are calling from the Main method.

Here, we can clearly see Method1, and Method2 are not waiting for each other.

**Output**

****

Now, coming to the second example, suppose we have Method3, which is dependent on Method1

**Example 2**

In this example, Method1 is returning the total length as an integer value and we are passing a parameter as a length in a Method3, which is coming from Method1.

Here, we have to use await keyword before passing a parameter in Method3 and for it, we have to use the async keyword from the calling method.

If we are using C# 7 or less, then we cannot use async keyword in the Main method for the console Application because it will give the error below.



We are going to create a new method as callMethod and in this method, we are going to call our all Methods as Method1, Method2, and Method3, respectively.

**Code sample** **C# 7**

class Program

{

static void Main(string[] args)

{

callMethod();

Console.ReadKey();

}

public static async void callMethod()

{

Task<int> task = Method1();

Method2();

int count = await task;

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

count += 1;

}

});

return count;

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

}

}

public static void Method3(int count)

{

Console.WriteLine("Total count is " + count);

}

}

C#

Copy

**Code sample C# 9**

class Program

{

static async Task Main(string[] args)

{

await callMethod();

Console.ReadKey();

}

public static async Task callMethod()

{

Method2();

var count = await Method1();

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

count += 1;

}

});

return count;

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

}

}

public static void Method3(int count)

{

Console.WriteLine("Total count is " + count);

}

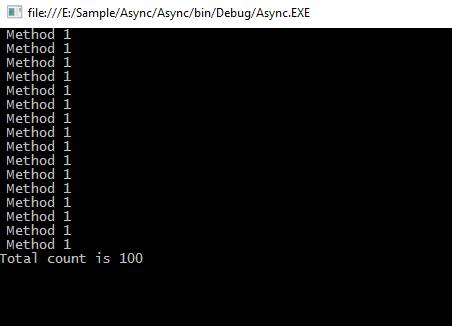
}

C#

Copy

In the code given above, Method3 requires one parameter, which is the return type of Method1. Here, await keyword is playing a vital role for waiting of Method1 task completion.

**Output**

****

# A==b can we override

# You can't. Operators aren't overridden, they're overloaded. That means the implementation to be used is entirely decided at compile-time.

# Opps concepts four piller

# Abstraction inheritance polymorphism encapsuklation

# done

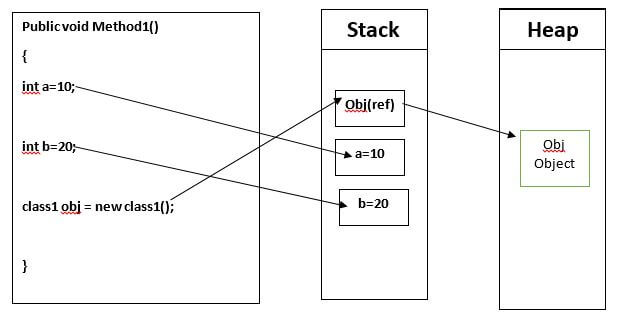
# Stack and heap

## **Difference between Stack and Heap Memory in C#**

|  |  |  |
| --- | --- | --- |
| Category | Stack Memory | Heap Memory |
| What is Stack & Heap? | It is an array of memory.  It is a LIFO (Last In First Out) data structure.  In it data can be added to and deleted only from the top of it. | It is an area of memory where chunks are allocated to store certain kinds of data objects.  In it data can be stored and removed in any order. |
| How Memory is Manages? | https://csharpcorner-mindcrackerinc.netdna-ssl.com/article/stack-vs-heap-memory-c-sharp/Images/stack1.gif | https://csharpcorner-mindcrackerinc.netdna-ssl.com/article/stack-vs-heap-memory-c-sharp/Images/heap1.gif |
| Practical Scenario | https://csharpcorner-mindcrackerinc.netdna-ssl.com/article/stack-vs-heap-memory-c-sharp/Images/ex11.gif  Value of variable storing in stack | https://csharpcorner-mindcrackerinc.netdna-ssl.com/article/stack-vs-heap-memory-c-sharp/Images/ex122.gif  Value of variable storing in heap |
| What goes on Stack & Heap? | "Things" declared with the following list of type declarations are Value Types  (because they are from System.ValueType):  bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong, ushort | "Things" declared with following list of type declarations are Reference Types  (and inherit from System.Object... except, of course, for object which is the System.Object object):  class, interface, delegate, object, string |
| Memory Allocation | Memory allocation is Static | Memory allocation is Dynamic |
| How is it Stored? | It is stored Directly | It is stored indirectly |
| Is Variable Resized? | Variables can’t be Resized | Variables can be Resized |
| Access Speed | Its access is fast | Its access is Slow |
| How is Block Allocated? | Its block allocation is reserved in LIFO.  Most recently reserved block is always the next block to be freed. | Its block allocation is free and done at any time |
| Visibility or Accessibility | It can be visible/accessible only to the Owner Thread | It can be visible/accessible to all the threads |
| In Recursion Calls? | In recursion calls memory filled up quickly | In recursion calls memory filled up slowly |
| Used By? | It can be used by one thread of execution | It can be used by all the parts of the application |
| StackOverflowException | .NET Runtime throws exception “StackOverflowException” when stack space is exhausted | - |
| When wiped off? | Local variables get wiped off once they lose the scope | - |
| Contains | It contains values for Integral Types, Primitive Types and References to the Objects | - |
| Garbage Collector | - | It is a special thread created by .NET runtime to monitor allocations of heap space.  It only collects heap memory since objects are only created in heap |

## **Summary**

Now, I believe you will be able to know the key difference between Stack and Heap Memory in C#.



# <https://www.c-sharpcorner.com/uploadfile/c210df/how-memory-is-managed-by-stack-and-heap/>

# Function values./variables in heap

It is a little more complicated than that and the fact that the stack and heap are used are really implementation details. It makes more sense to talk about lifetime of data. Short lived data will be stored on the stack (or in registers). Long lived data is stored on the heap.

Instances of reference types are always considered long lived, so they go on the heap. Value types can be both. Local value types are typically stored on the stack, but if something extends the lifetime of such a variable beyond the scope of the function, storing it on the stack wouldn't make sense. This happens for captured variables and these will be stored on the heap even if they are value types.

# Ref and out

# Method overidding.overloading,

olymorphism means “Many Forms”. In Polymorphism, poly means “Many” and morph means “Forms.” Polymorphism is one of the main pillars in Object Oriented Programming. It allows you to create multiple methods with the same name but different signatures in the same class. The same name methods can also be in derived classes.

There are two types of Polymorphism,

1. Method Overloading
2. Method Overriding

In this article, I will explain method overloading and method overriding concept in C#. I will try to demonstrate step by step differences between these.

## **Method Overloading**

Method Overloading is a type of polymorphism. It has several names like “Compile Time Polymorphism” or “Static Polymorphism” and sometimes it is called “Early Binding”.

Method Overloading means creating multiple methods in a class with same names but different signatures (Parameters). It permits a class, struct, or interface to declare multiple methods with the same name with unique signatures.

Compiler automatically calls required method to check number of parameters and their type which are passed into that method.

1. **using** System;
2. **namespace** DemoCsharp
3. {
4. **class** Program
5. {
6. **public** **int** Add(**int** num1, **int** num2)
7. {
8. **return** (num1 + num2);
9. }
10. **public** **int** Add(**int** num1, **int** num2, **int** num3)
11. {
12. **return** (num1 + num2 + num3);
13. }
14. **public** **float** Add(**float** num1, **float** num2)
15. {
16. **return** (num1 + num2);
17. }
18. **public** **string** Add(**string** value1, **string** value2)
19. {
20. **return** (value1 + " " + value2);
21. }
22. **static** **void** Main(**string**[] args)
23. {
24. Program objProgram = **new** Program();
25. Console.WriteLine("Add with two int parameter :" + objProgram.Add(3, 2));
26. Console.WriteLine("Add with three int parameter :" + objProgram.Add(3, 2, 8));
27. Console.WriteLine("Add with two float parameter :" + objProgram.Add(3 f, 22 f));
28. Console.WriteLine("Add with two string parameter :" + objProgram.Add("hello", "world"));
29. Console.ReadLine();
30. }
31. }
32. }

In the above example, you can see that there are four methods with same name but type of parameters or number of parameters is different. When you call Add(4,5), complier automatically calls the method which has two integer parameters and when you call Add(“hello”,”world”), complier calls the method which has two string parameters. So basically in method overloading complier checks which method should be called at the time of compilation.

**Note:**Changing the return type of method does not make the method overloaded. You cannot create method overloaded vary only by return type.

## **Method Overriding**

Method Overriding is a type of polymorphism. It has several names like “Run Time Polymorphism” or “Dynamic Polymorphism” and sometime it is called “Late Binding”.

Method Overriding means having two methods with same name and same signatures [parameters], one should be in the base class and other method should be in a derived class [child class]. You can override the functionality of a base class method to create a same name method with same signature in a derived class. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding.

1. **using** System;
2. **namespace** DemoCsharp
3. {
4. **class** BaseClass
5. {
6. **public** **virtual** **int** Add(**int** num1, **int** num2)
7. {
8. **return** (num1 + num2);
9. }
10. }
11. **class** ChildClass: BaseClass
12. {
13. **public** **override** **int** Add(**int** num1, **int** num2)
14. {
15. **if** (num1 <= 0 || num2 <= 0)
16. {
17. Console.WriteLine("Values could not be less than zero or equals to zero");
18. Console.WriteLine("Enter First value : ");
19. num1 = Convert.ToInt32(Console.ReadLine());
20. Console.WriteLine("Enter First value : ");
21. num2 = Convert.ToInt32(Console.ReadLine());
22. }
23. **return** (num1 + num2);
24. }
25. }
26. **class** Program
27. {
28. **static** **void** Main(**string**[] args)
29. {
30. BaseClass baseClassObj;
31. baseClassObj = **new** BaseClass();
32. Console.WriteLine("Base class method Add :" + baseClassObj.Add(-3, 8));
33. baseClassObj = **new** ChildClass();
34. Console.WriteLine("Child class method Add :" + baseClassObj.Add(-2, 2));
35. Console.ReadLine();
36. }
37. }
38. }

In the above example, I have created two same name methods in the BaseClass as well as in the ChildClass. When you call the BaseClass Add method with less than zero value as parameters then it adds successfully. But when you call the ChildClass Add method with less than zero value then it checks for negative value. And the passing values are negative then it asks for new value.

So, here it is clear that we can modify the base class methods in derived classes.

Points to be remembered,

1. Method cannot be private.
2. Only abstract or virtual method can be overridden.
3. Which method should be called is decided at run time.

## **Conclusion**

So, today we learned what Polymorphism is in OOP and what are the differences between method overloading and method overriding.

# Virtual use in overidding

# When a virtual method is invoked, the run-time type of the object is checked for an overriding member . The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member. By default, methods are non-virtual.

# Memory allocation

**C# Memory Management - Part 1**

In this article, I want to mention how memory management is done in .NET environment. I will try to keep it simple and short so that people with different levels of knowledge and experience can benefit from this. I believe this topic is very important in writing good-quality code and besides, it is popular in programming interviews.

I am planning to write this in three parts in order to make it easy to read. This part will include the topics:

* Stack and heap
* Value types and reference types

If you are ready, let’s start :)

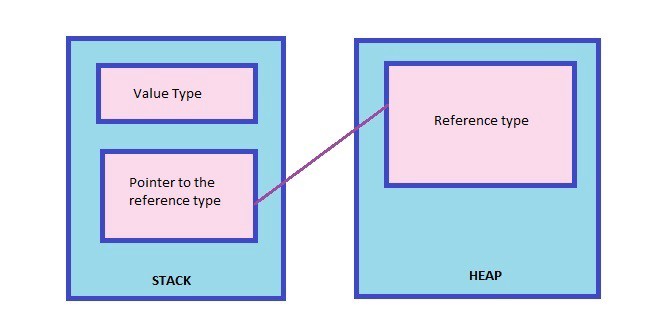
**Stack and Heap**

Stack and heap are portions of the memory. The ***Common Language Runtime (CLR)*** allocates memory for objects in these parts.

**Stack** is a simple LIFO(last-in-first-out) structure. Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and its allocation is done when the program is compiled. When a method is invoked, the CLR bookmarks the top of the stack. The method then pushes data onto the stack as it executes. When the method completes, the CLR resets the stack to its previous bookmark, popping all the method’s memory allocations is one simple operation.

**Heap** can be viewed as a random jumble of objects. It allows objects to be allocated or deallocated in a random order. Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory. The heap requires the overhead of a garbage collector to keep things in order.

Value type variables are stored in the stack and reference type variables are stored in the heap. *(This is a very general statement based on the information*[*here*](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype?redirectedfrom=MSDN&view=netframework-4.7.2)*.)*



**Value Type and Reference Type**

A **value type** holds the data within its own memory location.

Value types => *bool, byte, char, decimal, double, float, int, long, uint, ulong, ushort, enum, struct*

A **reference type** contains a pointer to another memory location that holds the real data.

Reference types => *class, interface, delegate, string, object, dynamic, arrays*

If you assign a value type variable to another variable, the value is copied directly.

Here is an example for a value type variable copy:

Output:

https://miro.medium.com/max/596/1*L-f0zqAgarOunxo5HMVmMw.png

As you see in the output, both variables work independently.

If you assign a reference type variable to another, as reference type variables represent the address of the variable, the reference is copied and both variables point to the same location of the heap.

Below is an example of reference variable copy:

Output:

https://miro.medium.com/max/562/1*Rp3NhczvU0pVCaiJOBN0eg.png

Have you noticed how Alice lost her name :) As these types of copies can result in unexpected behaviours in our programs, we should be careful when we copy reference type variables.

Now we have a general idea about the stack, heap and value/reference type variables. Additionally, I want to provide two very helpful resources which explain in detail how variables are stored in the memory.

First one is [Jon Skeet](https://stackoverflow.com/users/22656/jon-skeet)‘s [blog post](http://jonskeet.uk/csharp/memory.html). Below is a citation from his post:

*The memory slot for a variable is stored on either the stack or the heap. It depends on the context in which it is declared:*

*· Each local variable (i.e. one declared in a method) is stored on the stack. That includes reference type variables — the variable itself is on the stack, but remember that the value of a reference type variable is only a reference (or null), not the object itself. Method parameters count as local variables too, but if they are declared with the ref modifier, they don’t get their own slot, but share a slot with the variable used in the calling code.*

*· Instance variables for a reference type are always on the heap. That’s where the object itself “lives”.*

*· Instance variables for a value type are stored in the same context as the variable that declares the value type. The memory slot for the instance effectively contains the slots for each field within the instance. That means (given the previous two points) that a struct variable declared within a method will always be on the stack, whereas a struct variable which is an instance field of a class will be on the heap.*

*· Every static variable is stored on the heap, regardless of whether it’s declared within a reference type or a value type. There is only one slot in total no matter how many instances are created.*

Second is Wallace Kelly’s YouTube video. I strongly recommend you watch this video as he demonstrates the topic in a perfect way. After watching that, you will see how everything fall into place in your mind :)

# Garbge collection

# What is garbage collection in C #? The garbage collector (GC) manages the allocation and release of memory . The garbage collector serves as an automatic memory manager. You do not need to know how to allocate and release memory or manage the lifetime of the objects that use that memory

# Access modifier

C# has the following access modifiers:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| public | The code is accessible for all classes |
| private | The code is only accessible within the same class |
| protected | The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about [inheritance](https://www.w3schools.com/cs/cs_inheritance.asp) in a later chapter |
| internal | The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter |

# What is the difference between thread and process?

# In performance which is better

Here, are the important differences between Process and Thread

| **Parameter** | **Process** | **Thread** |
| --- | --- | --- |
| Definition | Process means a program is in execution. | Thread means a segment of a process. |
| Lightweight | The process is not Lightweight. | Threads are Lightweight. |
| Termination time | The process takes more time to terminate. | The thread takes less time to terminate. |
| Creation time | It takes more time for creation. | It takes less time for creation. |
| Communication | Communication between processes needs more time compared to thread. | Communication between threads requires less time compared to processes. |
| Context switching time | It takes more time for context switching. | It takes less time for context switching. |
| Resource | Process consume more resources. | Thread consume fewer resources. |
| Treatment by OS | Different process are tread separately by OS. | All the level peer threads are treated as a single task by OS. |
| Memory | The process is mostly isolated. | Threads share memory. |
| Sharing | It does not share data | Threads share data with each other. |

# What is a garbage collector? What is GAC?

# The Global Assembly Cache (GAC) is a folder in the Windows directory to store the. NET assemblies that are specifically designated to be shared by all applications executed on a system

# Dispose and tupple

# The Tuple <T> class was introduced in. NET Framework 4.0. A tuple is a data structure that contains a sequence of elements of different data types . It can be used where you want to have a data structure to hold an object with properties, but you do not want to create a separate type for it.

This piece of code will show you some memory management in classes.  
  
When we give application to client, we don't know how will he use that whether he will call dispose or not?  
  
It may be missing at client side so our application should be smart enough to free used resources, here is code that will give the idea to implement finalizer and dispose in same class to take care of resource cleaning.

using System;

namespace disposeConsole

{

class ResourceManagement : IDisposable

{

public ResourceManagement()

{}

private bool IsDisposed = false;

public void Free()

{

if (IsDisposed)

throw new System.ObjectDisposedException("Object Name");

}

//Call Dispose to free resources explicitly

public void Dispose()

{

//Pass true in dispose method to clean managed resources too and say GC to skip finalize in next line.

Dispose(true);

//If dispose is called already then say GC to skip finalize on this instance.

GC.SuppressFinalize(this);

}

~ResourceManagement()

{

//Pass false as param because no need to free managed resources when you call finalize it will be done

//by GC itself as its work of finalize to manage managed resources.

Dispose(false);

}

//Implement dispose to free resources

protected virtual void Dispose(bool disposedStatus)

{

if (!IsDisposed)

{

IsDisposed = true;

// Released unmanaged Resources

if (disposedStatus)

{

// Released managed Resources

}

}

}

}

}

C#

Copy

# What are stack,heap,value, reference types, boxing and unboxing? .

# Explain opps,Abstraction, encapsulation, inheritance and polymorphism? .................................. 31

# How is abstract class different from aninterface? ................................................................. 32

# What are the different types of polymorphism?

# Function call by refrence and call by value

# Resp API

# How can GC know this object is deleted

# Construtur and it type

# Static and private what is this,.how can we use

# Constutor calling

# Inheritance

# Static data member % classes

# Method overriffin od overloading scenario,

# How can we create objects

# Difference between a thread and a process

**Thread and a Process**

The processes and threads are independent sequences of execution, the typical difference is that threads run in a shared memory space, while processes run in separate memory spaces.

A process has a self contained execution environment that means it has a complete, private set of basic run time resources purticularly each process has its own memory space. Threads exist within a process and every process has at least one thread.

Each process provides the resources needed to execute a program. Each process is started with a single thread, known as the primary thread. A process can have multiple threads in addition to the primary thread.

On a multiprocessor system, multiple processes can be executed in parallel. Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.

Threads have direct access to the data segment of its process but a processes have their own copy of the data segment of the parent process.

Changes to the main thread may affect the behavior of the other threads of the process while changes to the parent process does not affect child processes.

Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process.

**What is multithreading ?**

In .NET languages you can write applications that perform multiple tasks simultaneously. Tasks with the potential of holding up other tasks can execute on separate threads is known as multithreading.