C# Basic Notes

**C# - Tuple :-**

* 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 don't want to create a separate type for it.
* The following example creates a tuple with three elements:

Tuple<int, string, string> person = new Tuple <int , string , string>(1, "Steve", "Jobs");

* In the above example, we created an instance of the Tuple that holds a person's record. We specified a type for each element and passed values to the constructor. Specifying the type of each element is cumbersome. C# includes a static helper class Tuple, which returns an instance of the Tuple<T> without specifying each element's type, as shown below.
* var person = Tuple.Create(1, "Steve", "Jobs");
* A tuple can only include a maximum of eight elements. It gives a compiler error when you try to include more than eight elements.
* Learn Tuple in below Link.
* [**https://www.tutorialsteacher.com/csharp/csharp-tuple**](https://www.tutorialsteacher.com/csharp/csharp-tuple)
* A tuple can only include a maximum of eight elements. It gives a compiler error when you try to include more than eight elements.
* var numbers = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8);

## **Accessing Tuple Elements :-**

* A tuple elements can be accessed with Item<elementNumber> properties, e.g., Item1, Item2, Item3, and so on up to Item7 property. The Item1 property returns the first element, Item2 returns the second element, and so on. The last element (the 8th element) will be returned using the Rest property.
* Example: Accessing Tuple Elements

var person = Tuple.Create(1, "Steve", "Jobs");

person.Item1; // returns 1

person.Item2; // returns "Steve"

person.Item3; // returns "Jobs"

var numbers = Tuple.Create("One", 2, 3, "Four", 5, "Six", 7, 8);

numbers.Item1; // returns "One"

numbers.Item2; // returns 2

numbers.Item3; // returns 3

numbers.Item4; // returns "Four"

numbers.Item5; // returns 5

numbers.Item6; // returns "Six"

numbers.Item7; // returns 7

numbers.Rest; // returns (8)

numbers.Rest.Item1; // returns 8

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

A [NetworkInterface](https://docs.microsoft.com/en-us/dotnet/api/system.net.networkinformation.networkinterface?view=net-6.0) array that contains objects that describe the available network interfaces, or an empty array if no interfaces are detected.

**IntPtr:**

* Represents a signed integer where the bit-width is the same as a pointer.

# Object Class:

# Supports all classes in the .NET class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all .NET classes; it is the root of the type hierarchy.

**Remarks:**

Languages typically do not require a class to declare inheritance from [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-6.0) because the inheritance is implicit.

Because all classes in .NET are derived from [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-6.0), every method defined in the [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-6.0) class is available in all objects in the system. Derived classes can and do override some of these methods, including:

* [Equals](https://docs.microsoft.com/en-us/dotnet/api/system.object.equals?view=net-6.0) - Supports comparisons between objects.
* [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=net-6.0) - Performs cleanup operations before an object is automatically reclaimed.
* [GetHashCode](https://docs.microsoft.com/en-us/dotnet/api/system.object.gethashcode?view=net-6.0) - Generates a number corresponding to the value of the object to support the use of a hash table.
* [ToString](https://docs.microsoft.com/en-us/dotnet/api/system.object.tostring?view=net-6.0) - Manufactures a human-readable text string that describes an instance of the class.

# gen\_image1 (Operator)

## **Name**

**gen\_image1** — Create an image from a pointer to the pixels.

## **Signature**

**gen\_image1**( : [*Image*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Image) : [*Type*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Type), [*Width*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Width), [*Height*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Height), [*PixelPointer*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#PixelPointer) : )

## **Description**

The operator gen\_image1 creates an image of the size [*Width*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Width) \* [*Height*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Height). The pixels in [*PixelPointer*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#PixelPointer) are stored line-sequentially. The type of the given pixels ([*PixelPointer*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#PixelPointer)) must correspond to [*Type*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#Type) (see [gen\_image\_const](https://www.mvtec.com/doc/halcon/13/en/gen_image_const.html) for a more detailed description of the pixel types). The storage for the new image is newly created by HALCON. Thus, the storage on the [*PixelPointer*](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html#PixelPointer) can be released after the call. Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

**Learn Types Of Events:**

**Learn all thread life cycle.**

**Project UI Design.**

**Software work:**

* Inspect all camera image and check all part.then it detect ok or not ok part and print there result on screen.
* Check all side and find all thread are ok and no any crash. Then result print ok. Otherwise print not ok.

**GenImage1:**

// Summary:

// Create an image from a pointer to the pixels.

//

// Parameters:

// image:

// Created image with new image matrix.

//

// type:

// Pixel type. Default: "byte"

//

// width:

// Width of image. Default: 512

//

// height:

// Height of image. Default: 512

//

// pixelPointer:

// Pointer to first gray value.

**Var in c#:**

* var can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
* var cannot be used on fields at class scope.

**Comment Shortcut in visual studio.**

**Comment:** Ctr+k+c

**Uncomment:** ctr+k+u

**What is Constructer:**

* A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created.
* Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation.
* It is used to assign initial values to the **data members** of the same class.

**Types of Constructer in c#:**

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Private Constructor
5. Static Constructor

**Difference Between Static and Non static constructer**

* **Static constructer** call only one time when **first** object is created.
* **Non Static / Instance** constructer call every time when object is created.

**Private Constructer:**

* If a constructor is created with private specifier is known as Private Constructor.
* It is not possible for other classes to derive from this class and also it’s not possible to create an instance of this class.

**Points to Remember:**

* It is the implementation of a singleton class pattern.
* use private constructor when we have only static members.
* Using private constructor, prevents the creation of the instances of that class.
* **Example:**

using System;

namespace privateConstructorExample {

public class Geeks {

// declare private Constructor

private Geeks()

{

}

// declare static variable field

public static int count\_geeks;

// declare static method

public static int geeks\_Count()

{

return ++count\_geeks;

}

// Main Method

public static void Main()

{

// Geeks s = new Geeks(); // Error

Geeks.count\_geeks = 99;

// Accessing without any

// instance of the class

Geeks.geeks\_Count();

Console.WriteLine(Geeks.count\_geeks);

// Accessing without any

// instance of the class

Geeks.geeks\_Count();

Console.WriteLine(Geeks.count\_geeks);

}

}

}

**Out Keyword:**

* The ***out*** is a keyword in C# which is used for the passing the arguments to methods as a reference type. It is generally used when a method returns multiple values.

# Int32.TryParse Method:

* Converts the string representation of a number to its 32-bit signed integer equivalent. A return value indicates whether the operation succeeded.
* public static bool TryParse (string? s, out int result);
* A string containing a number to convert.
* **Result**
  + When this method returns, contains the 32-bit signed integer value equivalent of the number contained in s, if the conversion succeeded, or zero if the conversion failed. The conversion fails if the s parameter is null or [Empty](https://docs.microsoft.com/en-us/dotnet/api/system.string.empty?view=net-6.0), is not of the correct format, or represents a number less than [Int32.MinValue](https://docs.microsoft.com/en-us/dotnet/api/system.int32.minvalue?view=net-6.0) or greater than [Int32.MaxValue](https://docs.microsoft.com/en-us/dotnet/api/system.int32.maxvalue?view=net-6.0). This parameter is passed uninitialized; any value originally supplied in result will be overwritten.

**Static and Non Static Class:**

| **Static Class** | **Non-Static Class** |
| --- | --- |
| Static class is defined using static keyword. | Non-Static class is not defined by using static keyword. |
| In static class, you are not allowed to create objects. | In non-static class, you are allowed to create objects using new keyword. |
| The data members of static class can be directly accessed by its class name. | The data members of non-static class is not directly accessed by its class name. |
| Static class always contains static members. | Non-static class may contain both static and non-static methods. |
| Static class does not contain an instance constructor. | Non-static class contains an instance constructor. |
| Static class cannot inherit from another class. | Non-static class can be inherited from another class. |

**Thread in C#:**

* Multi-threading is the most useful feature of C# which allows concurrent programming of two or more parts of the program for maximizing the utilization of the CPU.
* Each part of a program is called Thread. So, in other words, threads are lightweight processes within a process.
* C# supports two types of threads are as follows:

**Single Threaded Application:**

* Operating System support Multi-Tasking.
* So Operating System Create One Process for Every Application.
* **For Example:** In same Time start VLC Player, Notepad, Visual Studio.
* Then this time create three Process, One for VLC, Second for Notepad, and Third Process for Visual Studio.
* **Thread Run** Inside Process.
* **So** When We Run any application then this time Run thread.
* In single Threaded application only single thread Responsible to run all the application.
* In your application multiple method/ functionality are present. But in single threaded application not run all function Simultaneously.
* Every function can Run one by one. Means First Method Process Completed then call second Method or then call Third. And so on.

**Multi-Threaded Application:**

* Multithreading is a feature provided by the operating system that enables your application to have more than one execution path at the same time.
* Technically, multithreaded programming requires a multitasking operating system.
* In Multi-Threading Multiple Thread Are Running.
* So this time one Thread is waiting state or sleeping then this time another thread Run and Complete their work.
* Not wait to complete first thread all work.

**Steps to create a thread in a C# Program:**

1. First of all, import System.Threading namespace, it plays an important role in creating a thread in your program as you have no need to write the fully qualified name of class everytime.

Using System;

Using System.Threading

1. Now, create and initialize the thread object in your main method.

public static void main()

{

Thread thr = new Thread(job1);

}

**Or**

You can also use ThreadStart constructor for initializing a new instance.

public static void main()

{

Thread thr = new Thread(new ThreadStart(job1));

}

1. Now you can call your thread object.

public static void main()

{

Thread thr = new Thread(job1);

thr.Start();

}

**Some commonly used classes in this namespace are:**

|  |  |
| --- | --- |
| Class Name | Description |
| **Mutex** | It is a synchronization primitive that can also be used for IPS (interprocess synchronization). |
| **Monitor** | This class provides a mechanism that access objects in synchronize manner. |
| **Semaphore** | This class is used to limit the number of threads that can access a resource or pool of resources concurrently. |
| **Thread** | This class is used to creates and controls a thread, sets its priority, and gets its status. |
| **ThreadPool** | This class provides a pool of threads that can be used to execute tasks, post work items, process asynchronous I/O, wait on behalf of other threads, and process timers. |
| **ThreadLocal** | This class provides thread-local storage of data. |
| **Timer** | This class provides a mechanism for executing a method on a thread pool thread at specified intervals. You are not allowed to inherit this class. |
| **Volatile** | This class contains methods for performing volatile memory operations. |

**Most Common Instance Member of the System.Threading. Thread class**

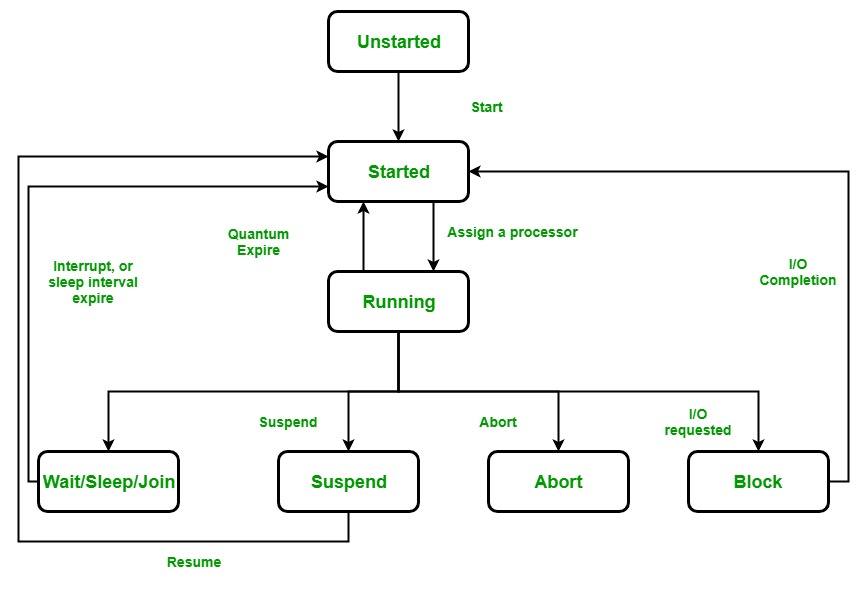
The following are the most common instance members of the System.Threading.Thread class:

* **Name**   
    
  A property of string type used to get/set the friendly name of the thread instance.
* **Priority**   
    
  A property of type System.Threading.ThreadPriority to schedule the priority of threads.
* **IsAlive**   
    
  A Boolean property indicating whether the thread is alive or terminated.
* **ThreadState**   
    
  A property of type System.Threading.ThreadState, used to get the value containing the state of the thread.
* **Start()**Starts the execution of the thread.
* **Abort()**   
    
  Allows the current thread to stop the execution of the thread permanently.
* **Suspend()**   
    
  Pauses the execution of the thread temporarily.
* **Resume()**Resumes the execution of a suspended thread.
* **Join()**  
    
  Make the current thread wait for another thread to finish.

# Lifecycle and States of a Thread in C#

A thread in C# at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

**Flow Chart:**



**Life Cycle of a thread**

1. **Unstarted state:**When an instance of a Thread class is created, it is in the unstarted state, means the thread has not yet started to run when the thread is in this state. Or in other words *Start()* method is not called.

Thread thr = new Thread();

1. **Runnable State:**A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. Or in other words, the *Start()* method is called.
2. **Running State:**A thread that is running. Or in other words, the thread gets the processor.
3. **Not Runnable State:**A thread that is not executable because
   1. Sleep() method is called.
   2. Wait() method is called.
   3. Due to I/O request.
   4. Suspend() method is called.
4. **Dead State:** When the thread completes its task, then thread enters into dead, terminates, abort state.

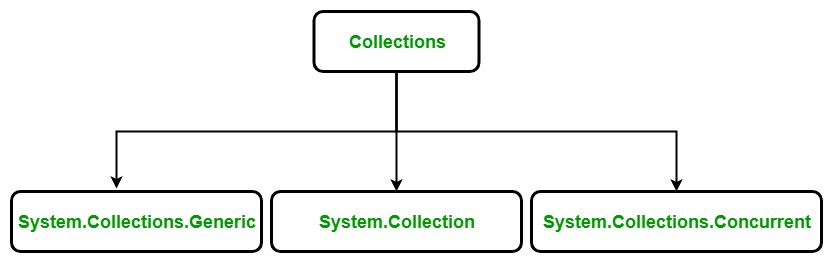
# Collections in C#

# Collections standardize the way of which the objects are handled by your program.

# In other words, it contains a set of classes to contain elements in a generalized manner.

# With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.

**C# divide collection in several classes, some of the common classes are shown below:**



#### **System.Collections.Generic Classes:**

* Generic collection in C# is defined in System.Collection.Generic namespace.
* It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries.
* These collections are type-safe because they are generic means only those items that are type-compatible with the type of the collection can be stored in a generic collection, it eliminates accidental type mismatches.
* Generic collections are defined by the set of interfaces and classes.
* Below table contains the **frequently used classes** of the
* **System.Collections.Generic namespace:**

1. [List<T>](https://www.geeksforgeeks.org/c-list-class/)
2. **Dictionary<TKey,TValue>**
3. **Queue<T>**
4. **SortedList<TKey,TValue>**
5. **Stack<T>**
6. [HashSet<T>](https://www.geeksforgeeks.org/c-sharp-hashset-class/)
7. [LinkedList<T>](https://www.geeksforgeeks.org/c-sharp-linkedlist-class/)

**Explanations Step by Step.**

1. **List<T> Class:**

* **List<T> class** represents the list of objects which can be accessed by index.
* It comes under the **System.Collection.Generic** namespace.
* List class can be used to create a collection of different types like integers, strings etc. List<T> class also provides the methods to search, sort, and manipulate lists.

**Characteristics:**

* It is different from the arrays. A **List<T> can be resized dynamically** but arrays cannot.
* List<T> class can accept null as a valid value for reference types and it also allows duplicate elements.
* If the Count becomes equals to Capacity, then the capacity of the List increased automatically by reallocating the internal array. The existing elements will be copied to the new array before the addition of the new element.
* List<T> class is the generic equivalent of ArrayList class by implementing the IList<T> generic interface.
* This class can use both equality and ordering comparer.
* List<T> class is not sorted by default and elements are accessed by zero-based index.
* For very large List<T> objects, you can increase the **maximum capacity to 2 billion elements** on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.

#### **Constructors**

| Constructor | Description |
| --- | --- |
| **List<T>()** | Initializes a new instance of the List<T> class that is empty and has the default initial capacity. |
| **List<T>(IEnumerable<T>)** | Initializes a new instance of the List<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied. |
| **List<T>(Int32)** | Initializes a new instance of the List<T> class that is empty and has the specified initial capacity. |

|  |
| --- |
| // C# program to create a List<T>  using System;  using System.Collections.Generic;    class Geeks {        // Main Method      public static void Main(String[] args)      {          // Creating a List of integers          List<int> firstlist = new List<int>();          // displaying the number          // of elements of List<T>          Console.WriteLine(firstlist.Count);      }  } |