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" \l "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" \l "PixelPointer)* are stored line-sequentially. The type of the given pixels (*[PixelPointer](https://www.mvtec.com/doc/halcon/13/en/gen_image1.html" \l "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" \l "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:**