<https://www.codeproject.com/Articles/38069/Memory-Management-in-NET>

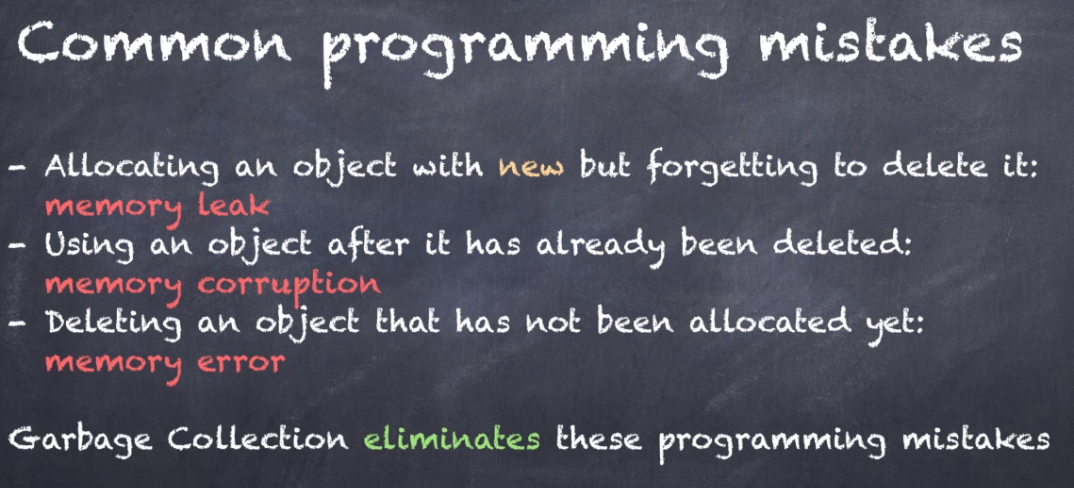
[**https://www.c-sharpcorner.com/article/memory-management-in-net/**](https://www.c-sharpcorner.com/article/memory-management-in-net/)**=**

<https://www.jetbrains.com/help/dotmemory/NET_Memory_Management_Concepts.html>

<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/latency>

<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/weak-references>

<https://www.dotnettricks.com/learn/netframework/difference-between-finalize-and-dispose-method>

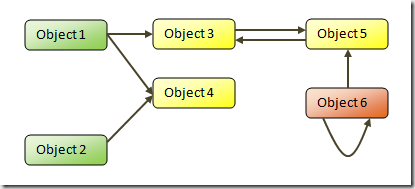




**Mark and Sweep Algorithm**

### Recap of definition

1. ***Object***: This is a unit of storage on the heap
2. ***Object/Reference graph*:** This is the directional graph of objects in memory. A typical sample is below. The nodes are objects in memory and the edges (arrows) are references one object holds to another.

[](https://msdnshared.blob.core.windows.net/media/TNBlogsFS/BlogFileStorage/blogs_msdn/abhinaba/WindowsLiveWriter/BackToBasicsMarkandSweepGarbageCollectio_FF09/image_4.png) 

1. ***Roots*:** These are the set of nodes in the object graph from which the references start. These are typically references held in registers, local variable on the stack or global variables. The green nodes in the diagram above are roots.
2. ***Unreachable object*:** These are nodes in the graph which have no edge referencing them. The Orange node in the diagram above is an unreachable object. This is the node that the GC needs to clean/free because it is not reachable from any node and is hence garbage memory.

### Mark-sweep GC:

In this scheme memory is not freed the moment they become garbage. Actually no subsystem is used to keep track of memory as they are being used.

When the system starts running out of memory (or some other such trigger) the GC is fired. It first enumerates all the roots and then starts visiting the objects referenced by them recursively (essentially travelling the nodes in the memory graph). When it reaches an object it marks it with a special flag indicating that the object is reachable and hence not garbage. At the end of this mark phase it gets into the sweep phase. Any object in memory that is not marked by this time is garbage and the system disposes it.

### Advantage and disadvantages

The primary advantage of mark-sweep is that it handles cyclic references naturally. Moreover, no additional overheads are added while normal execution (e.g. allocation, pointer manipulations). Combined with compaction it ensures good locality of reference and reduced fragmentation and hence optimal subsequent allocations.

However, mark-sweep pauses useful execution and walks entire memory marking and sweeping it. Hence it adds large freezes which is un-acceptable in most interactive systems. However, incremental variations of Mark-sweep are available which works around this limitation. Mark-sweep also starts thrashing when memory fills up as it fails to clean up memory and it gets triggered again and again as memory approaches exhaustion. Self-tuning GC helps in this scenario.

**When is the GC RUN?**

At the face of it, it seems that GC should be run when memory allocation fails. However, when memory allocation fails there is a high chance that GC will fail as well because it would need some working memory to function which won’t be available. **So in real world systems GC is fired under a variety of conditions like:**

1. System is low on memory (enough for a GC to run but soon allocations are going to fail)

The system has low physical memory. This is detected by either the low memory notification from the OS or low memory indicated by the host.

1. The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
2. The [GC.Collect](https://docs.microsoft.com/en-us/dotnet/api/system.gc.collect) method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

This obviously varies from system to system and is carefully tuned to match the scenarios being supported.

## **The managed heap:**

## **https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals**

After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.

There is a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.

**How GC Works:** <https://www.codeproject.com/Articles/1095402/Garbage-Collection-and-Csharp>

<https://www.red-gate.com/products/dotnet-development/ants-memory-profiler/learning-memory-management/memory-management-fundamentals> -for Generations

<https://www.red-gate.com/products/dotnet-development/ants-memory-profiler/learning-memory-management/understanding-dotnet-interoperability>

<https://www.red-gate.com/products/dotnet-development/ants-memory-profiler/learning-memory-management/avoidingautomaticgc>

http://joeduffyblog.com/2005/04/08/dg-update-dispose-finalization-and-resource-management/

GC works on **managed heap**, which is nothing but a block of memory to store objects, when garbage collection process is put in motion, it checks for dead objects and the objects which are no longer used, then it compacts the space of live object and tries to free more memory.

The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap. There are three generations of objects on the heap:

* **0 Generation (Zero)**: This generation holds short-lived objects, e.g., Temporary objects. GC initiates garbage collection process frequently in this generation.

Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection.

Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

* **1 Generation (One)**: This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
* **2 Generation (Two)**: This generation holds long-lived objects like a static and global variable that needs to be persisted for a certain amount of time. Objects which are not collected in generation Zero, are then moved to generation 1, such objects are known as *survivors*, similarly objects which are not collected in generation One, are then moved to generation 2 and from there onwards objects remain in the same generation.

**Survival and promotions**

Objects that are not reclaimed in a garbage collection are known as survivors, and are promoted to the next generation. Objects that survive a generation 0 garbage collection are promoted to generation 1; objects that survive a generation 1 garbage collection are promoted to generation 2; and objects that survive a generation 2 garbage collection remain in generation 2.

When the garbage collector detects that the survival rate is high in a generation, it increases the threshold of allocations for that generation, so the next collection gets a substantial size of reclaimed memory. The CLR continually balances two priorities: not letting an application's working set get too big and not letting the garbage collection take too much time.

## **What happens during a garbage collection?**

A garbage collection has the following phases:

* A marking phase that finds and creates a list of all live objects.
* A relocating phase that updates the references to the objects that will be compacted.
* A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

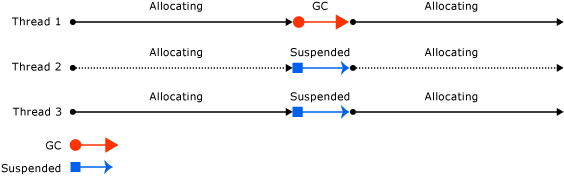
Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different segment, because they are promoted to generation 2.

**The garbage collector uses the following information to determine whether objects are live:**

* **Stack roots**. Stack variables provided by the just-in-time (JIT) compiler and stack walker.
* **Garbage collection handles**. Handles that point to managed objects and that can be allocated by user code or by the common language runtime.
* **Static data**. Static objects in application domains that could be referencing other objects. Each application domain keeps track of its static objects.

Before a garbage collection starts, all managed threads are suspended except for the thread that triggered the garbage collection.

The following illustration shows a thread that triggers a garbage collection and causes the other threads to be suspended.



## **What is Managed and Unmanaged Objects/Resources?**

**Managed code:**

* **Managed objects** are created, managed and under scope of CLR, pure .NET code managed by runtime, anything that lies within .NET scope and under .NET framework classes such as string, int, bool variables are referred to as managed code.
* Means that compile to Intermediate Language - IL, and execute under the management of Common Language Runtime - CLR. Ex: C#, VB …

**Unmanaged Code:**

* **Unmanaged objects** are created outside the control of .NET libraries and are not managed by CLR, example of such unmanaged code is COM objects, file streams, connection objects, Interop objects. (Basically, third party libraries that are referred in .NET code.)
* Means that compile directly to machine code, and directly executed by OS. Ex: C++, Win32.

## **Clean Up Unmanaged Resources:**

<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged>

<https://docs.microsoft.com/en-us/visualstudio/code-quality/ca1816-call-gc-suppressfinalize-correctly?view=vs-2015>

https://stackoverflow.com/questions/2605412/why-should-we-call-suppressfinalize-when-we-dont-have-a-destructor#

The most common types of unmanaged resource are objects that wrap operating system resources, such as files, windows, network connections, or database connections. Although the garbage collector is able to track the lifetime of an object that encapsulates an unmanaged resource, it doesn't know how to release and clean up the unmanaged resource.

When we create unmanaged objects, GC is unable to clear them and we need to release such objects explicitly when we finished using them. Mostly unmanaged objects are wrapped/hide around operating system resources like file streams, database connections, network related instances, handles to different classes, registries, pointers etc. GC is responsible to track the life time of all managed and unmanaged objects but still GC is not aware of releasing unmanaged resources.

**There are different ways to cleanup unmanaged resources:**

* Implement IDisposable interface and Dispose method
* 'using' block is also used to clean unmanaged resources

<https://stackoverflow.com/questions/35426983/why-do-we-need-idisposable-objects>

<https://www.codeproject.com/Articles/413887/Understanding-and-Implementing-IDisposable-Interfa>

<https://blog.stephencleary.com/2009/08/first-rule-of-implementing-idisposable.html>

<https://dailydotnettips.com/benefit-of-using-in-dispose-for-net-objects-why-and-when/>

If your types use unmanaged resources, you should do the following:

* Implement the [dispose pattern](https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern). This requires that you provide an [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) implementation to enable the deterministic release of unmanaged resources. A consumer of your type calls [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) when the object (and the resources it uses) is no longer needed. The [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method immediately releases the unmanaged resources.
* Provide for your unmanaged resources to be released in the event that a consumer of your type forgets to call [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose). There are two ways to do this:
  + Use a safe handle to wrap your unmanaged resource. This is the recommended technique. Safe handles are derived from the [System.Runtime.InteropServices.SafeHandle](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.safehandle) class and include a robust [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize) method. When you use a safe handle, you simply implement the [IDisposable](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable) interface and call your safe handle's [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.safehandle.dispose) method in your [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) implementation. The safe handle's finalizer is called automatically by the garbage collector if its [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method is not called.

—or—

* + Override the [Object.Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize) method. Finalization enables the non-deterministic release of unmanaged resources when the consumer of a type fails to call [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) to dispose of them deterministically. However, because object finalization can be a complex and error-prone operation, we recommend that you use a safe handle instead of providing your own finalizer.

Consumers of your type can then call your [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) implementation directly to free memory used by unmanaged resources. When you properly implement a [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method, either your safe handle's [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize) method or your own override of the [Object.Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize) method becomes a safeguard to clean up resources in the event that the [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method is not called.

**Object.Finalize:**   
Provides for object finalization if unmanaged resources are not released by the [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method.

Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.

The following example verifies that the [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) method is called when an object that overrides [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) is destroyed. The [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class.

[Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as [GC.ReRegisterForFinalize](https://docs.microsoft.com/en-us/dotnet/api/system.gc.reregisterforfinalize?view=netframework-4.7.2) and the [GC.SuppressFinalize](https://docs.microsoft.com/en-us/dotnet/api/system.gc.suppressfinalize?view=netframework-4.7.2) method has not been subsequently called.

[**GC.SuppressFinalize**](https://docs.microsoft.com/en-us/dotnet/api/system.gc.suppressfinalize):  
Suppresses finalization. This method is customarily called from a Dispose method to prevent a finalizer from executing.

Requests that the common language runtime not call the finalizer for the specified object.

This method sets a bit in the object header of obj, which the runtime checks when calling finalizers. A finalizer, which is represented by the [Object.Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) method, is used to release unmanaged resources before an object is garbage-collected. If obj does not have a finalizer, the call to the [SuppressFinalize](https://docs.microsoft.com/en-us/dotnet/api/system.gc.suppressfinalize?view=netframework-4.7.2) method has no effect.

Objects that implement the [IDisposable](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.7.2) interface can call this method from the object's [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose?view=netframework-4.7.2) implementation to prevent the garbage collector from calling [Object.Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netframework-4.7.2) on an object that does not require it. Typically, this is done to prevent the finalizer from releasing unmanaged resources that have already been freed by the [IDisposable.Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose?view=netframework-4.7.2) implementation.

**Benefits of GC:** In the common language runtime (CLR), the garbage collector serves as an automatic memory manager. It provides the following benefits:

* Enables you to develop your application without having to free memory.
* Allocates objects on the managed heap efficiently.
* Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
* Provides memory safety by making sure that an object cannot use the content of another object.

**Types in C#:**

**Var type:**

Var is used to declare implicitly typed local variable means it tells the compiler to figure out the type of the variable at compilation time. A var variable must be initialized at the time of declaration.

### Valid var statements

1. var str = "1";
2. var num = 0;
3. string s = "string";
4. var s2 = s;
5. s2 = null;
6. string s3 = null;
7. var s4 = s3;

### At compile time, the above var statements are compiled to IL, like this:

1. string str = "1";
2. int num = 0;
3. string s2 = s;
4. string s4 = s3;

### The compile-time type value of var variable cannot be null but the runtime value can be null.

1. *// invalid var statements*
2. var v; *//need to initialize*
3. var num = null; *// can’t be null at compile time*

### Once var variable is initialized its data type became fixed to the type of the initial data.

1. *// invalid var statements*
2. var v2 = "str12";
3. v2 = 3; *// int value can’t be assign to implicitly type string variable v2*

**Anonymous Type:**

An anonymous type is a simple class generated by the compiler within IL to store a set of values. Var data type and new keyword is used to create an anonymous type.

1. var emp = new { Name = "Deepak", Address = "Noida", Salary=21000 };

### At compile time, the compiler will create an anonymous type, as follows:

1. class \_\_Anonymous1
2. {
3. private string name;
4. private string address;
5. int salary; public string Name
6. {
7. get{return name; }
8. set { name=value }
9. }
10. public string Address
11. {
12. get{ return address; }
13. set{ address=value; }
14. }
15. public int Salary
16. {
17. get{ return salary; }
18. set{ salary=value; }
19. }
20. }

### The anonymous type is very useful when you want to shape the result in your desired form like this:

1. var result =from book in Books
2. where book.Price > 200
3. orderby book.IssueDate descending
4. select new
5. {
6. Name = book.Name,
7. IssueNumber = "#" + book.Issue
8. };

### In above example, I change the name of the “Issue” field of Book table to “IssueNumber” and add # before value to get desired output.

### Dynamic type:

### It avoids compile time type checking. A dynamic type escapes type checking at compile time; instead, it resolves type at run time.

### Example:

dynamic dynamicVariable = 1;

The compiler compiles dynamic types into object types in most cases. The above statement would be compiled as:

object dynamicVariable = 1;

Console.WriteLine(dynamicVariable.GetType().ToString());

**Output**: System.Int32

## **Methods and Properties of Dynamic Type:**

If you assign class object to the dynamic type then the compiler would not check for correct methods and properties name of a dynamic type that holds the custom class object.

dynamic dynamicStudent = new Student();

dynamicStudent.FakeMethod();

At run time dynamicStudent will become Student type and if FakeMethod is not present in Student class than exception will be thrown.

The actual type of dynamic would resolve at runtime. A dynamic type changes its type at runtime based on the value of the expression to the right of the "=" operator.

## **Dynamic Type as a Method Parameter:**

A method can have dynamic type parameters so that it can accept any type of parameter at run time.

http://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember:

1. The dynamic types are resolved at runtime instead of compile time.
2. The compiler skips the type checking for dynamic type. So it doesn't give any error about dynamic types at compile time.
3. The dynamic types do not have intellisense support in visual studio.
4. A method can have parameters of the dynamic type.
5. An exception is thrown at runtime if a method or property is not compatible

### C# Keywords:

## **Ref:**

The ref keyword is used to pass an argument as a reference. This means that when value of that parameter is changed in the method, it gets reflected in the calling method. An argument that is passed using a ref keyword must be initialized in the calling method before it is passed to the called method.

**Out:**

The out keyword is also used to pass an argument like ref keyword, but the argument can be passed without assigning any value to it. An argument that is passed using an out keyword must be initialized in the called method before it returns back to calling method.

**Example:**

1. public class Example
2. {
3. public static void Main() *//calling method*
4. {
5. int val1 = 0; *//must be initialized*
6. int val2; *//optional*
8. Example1(ref val1);
9. Console.WriteLine(val1); *// val1=1*
11. Example2(out val2);
12. Console.WriteLine(val2); *// val2=2*
13. }
15. static void Example1(ref int value) *//called method*
16. {
17. value = 1;
18. }
19. static void Example2(out int value) *//called method*
20. {
21. value = 2; *//must be initialized*
22. }
23. }
25. */\* Output*
26. *1*
27. *2*
28. *\*/*

**Note:**

1. Do not be confused with the concept of passing by reference and the concept of reference type. These two concepts are not the same.
2. A value type or a reference type can be passed to method parameter by using ref keyword. There is no boxing of a value type when it is passed by reference.
3. Properties cannot be passed to ref or out parameters since internally they are functions and not members/variables.

## **Ref and out in method overloading:**

Both ref and out cannot be used in method overloading simultaneously. However, ref and out are treated differently at run-time but they are treated same at compile time (CLR doesn't differentiates between the two while it created IL for ref and out). Hence methods cannot be overloaded when one method takes a ref parameter and other method takes an out parameter. The following two methods are identical in terms of compilation.

### Incorrect:

1. class MyClass
2. {
3. public void Method(out int a) *// compiler error “cannot define overloaded”*
4. {
5. *// method that differ only on ref and out"*
6. }
7. public void Method(ref int a)
8. {
9. *// method that differ only on ref and out"*
10. }
11. }

### Correct:

1. class MyClass
2. {
3. public void Method(int a)
4. {
6. }
7. public void Method(out int a)
8. {
9. *// method differ in signature.*
10. }
11. }

## **Constant:**

Constant fields or local variables must be assigned a value at the time of declaration and after that they cannot be modified. By default constant are static, hence cannot define it as static.

1. public const int X = 10;

A const field is a compile-time constant. A constant field or local variable can be initialized with a constant expression which must be fully evaluated at compile time.

1. void Calculate(int Z)
2. {
3. const int X = 10, X1 = 50;
4. const int Y = X + X1; *//no error, since its evaluated a compile time*
5. const int Y1 = X + Z; *//gives error, since its evaluated at run time*
6. }

You can apply const keyword to built-in value types (byte, short, int, long, char, float, double, decimal, bool), enum, a string literal, or a reference type which can be assigned with a value null.

1. const MyClass obj1 = null;*//no error, since its evaluated a compile time*
2. const MyClass obj2 = new MyClass();*//gives error, since its evaluated at run time*

Constants can be marked as public, private, protected, internal, or protected internal.

Use the const modifier when you sure that the value a field or local variable would not be changed.

### ReadOnly:

### A readonly field can be initialized either at the time of declaration or with in the constructor of same class. Therefore, readonly fields can be used for run-time constants.

1. class MyClass
2. {
3. readonly int X = 10; *// initialized at the time of declaration*
4. readonly int X1;
6. public MyClass(int x1)
7. {
8. X1 = x1; *// initialized at run time*
9. }
10. }

### It is not static by default. Readonly keyword can be apply to value type and reference type (which initialized by using the new keyword) both. Also, delegate and event could not be readonly. Use the readonly modifier when you want to make a field constant at run time.

### Safe type casting with Is Operator:

### The IS operator checks whether the type of a given object is compatible with the new object type. It returns Boolean type value: true if given object is compatible with new one, else false. In this way IS operator help you to do safe type casting.

1. Object obj = new Object(); *// Creates a new Object obj*
2. *// checking compatibility of obj object with other type*
3. Boolean b1 = (obj is Object); *// b1 is set to true.*
4. Boolean b2 = (obj is Employee); *//The cast fails: no exception thrown, but b2=false.*
5. *//we can also use it*
6. if (obj is Employee)
7. {
8. Employee emp = (Employee) obj;
9. *// TO DO:*
10. }

### If reference of object is null, IS operator will return false since there is no object available to check its type.

### Safe type casting with AS Operator:

### The AS operator also checks whether the type of a given object is compatible with the new object type. It returns non-null if given object is compatible with new one, else null. In this way AS operator help you to do safe type casting. The above code can be re-written by using AS operator in a better way.

1. Object obj = new Object(); *// Creates a new Object obj*
2. *// checking compatibility of obj object with other type*
3. Employee emp = obj as Employee; *// The cast fails: no exception is thrown, but emp is set to null.*
4. if (emp != null)
5. {
6. *// TO:DO*
7. }

### Note:

### If the reference of the given object is null, the AS operator will return NULL since there is no object available to check its type.

### AS operator performs only reference conversions, nullable conversions, and boxing conversions. This operator cannot perform other conversions like as user-defined conversions.

1. Object obj = new Object();
2. Employee emp = obj as Employee; *// try to Cast obj to an Employee*
3. *//the above cast fails: no exception is thrown, but emp is set to null.*
4. emp.ToString(); *// Accessing emp throws a NullReferenceException.*

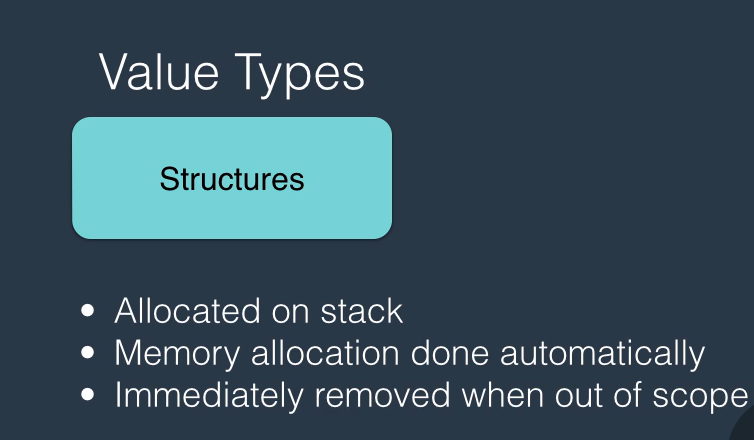
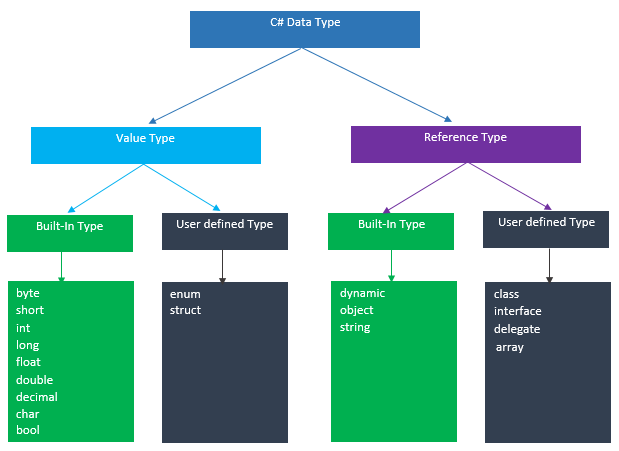
### C# Datatype

### Data Type refers to the type of data that can be stored in a variable. It also specifies how much memory would be allocated to a variable and the operations that can be performed on that.

C# is rich in data type which is broadly divided into two categories.

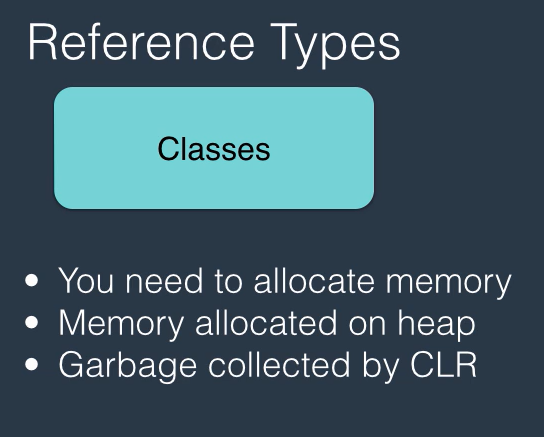
* Value Type
* Reference Type

**Value Type:**

A value type variable store actual value. Also, value types are stored in a stack. Values types are of two types - built-in and user-defined. Value types are derived from System.ValueTy

**Reference Type:**

A reference type variable stores a reference to the actual value. It means reference type contains a pointer to another memory location that holds the actual data. Also, reference types are stored in a heap. Reference types are of two types - built-in and user-defined. Reference types are derived from System.Object.



### String Hacks:

### Convert char array to string: ToString will not work, gives System[] something..

### new string(strArr)

### string.Join("", strArr)

### Array Hacks:

### Remove array elements:

### By using LINQ(Where, FindAll).

### By converting Array to List and then use Remove method of array and then covert back to Array.