

.NET Best Practices



Satisfying Customers Beyond Expectations

Trinay Technology Solutions

08

Table of Contents

Managed Heap and Garbage Collection Overview	3
Memory Allocation.....	3
De-allocation of Memory.....	3
Object Finalization.....	4
Generations in GC.....	5
Minimize Boxing and Unboxing	6
Finalize() Method Implementation	7
Implement the Standard Dispose Pattern.....	8
Utilize <code>using</code> and <code>TRY/finally</code> for Resource Cleanup.....	10
Always Use Properties Instead of Accessible Data Members.....	12
Prefer <code>readonly</code> to <code>const</code>	14
Prefer the <code>is</code> or <code>as</code> Operators to Casts	15
Always Provide <code>ToString()</code>	16
Ensure that 0 is a valid State for Value Types.....	17
Consider overriding the <code>Equals()</code> method for value types.....	18
Understand the Relationships Among <code>ReferenceEquals()</code> , <code>static Equals()</code> , <code>instance Equals()</code> , and <code>operator==</code>	18
Avoid pre-allocating memory.....	19
Prefer Variable Initializers to Assignment Statements.....	19
Initialize Static Class Members with Static Constructors.....	21
Utilize Constructor Chaining.....	21
Prefer Small, Simple Functions	22
Avoid inefficient string operations	22
Prefer Single Large Assemblies Rather Than Multiple Smaller Assemblies.....	22
Locking and Synchronization guidelines.....	22
Minimize Thread Creation.....	23
Do not set local variables to null.....	23
Miscellaneous	23
Multithreading.....	29
Serialization	31
Remoting.....	31
Security	32

Managed Heap and Garbage Collection Overview

Memory Allocation

1. .NET components aren't allocated out of raw memory maintained by operating system.
2. Each physical process that hosts .NET, has pre-allocated special heap called the managed heap.
3. Managed heap is directly handled by CLR.

When the CLR is loaded, two initial heap segments are allocated—one for small objects and one for large objects, which I will refer to as the small object heap (SOH) and the large object heap (LOH), respectively.

Allocation requests are then satisfied by putting managed objects on one of these managed heap segments. If the object is less than 85,000 bytes, it will be put on a SOH segment; otherwise it'll be on a LOH segment. Segments are committed (in smaller chunks) as more and more objects are allocated onto them.

4. Each thread has its own stack, but heaps are typically shared by threads in a process.
5. .NET allocates memory off the managed heap.
6. .NET maintains a pointer to the next available address in the managed heap.
7. When a new object is created, it allocates the required space for the object and advances the pointer.

Note:

In unmanaged environments such as C++, objects are allocated off the native operating system heap. The operating system manages its memory by using a linked list of available blocks of memory. Each time the operating system has to allocate memory, it traverses that list looking for a big enough block.

8. Win32 applications have a limitation of 2 GB memory space.

Note:

- * By definition, a 32-bit processor uses 32 bits to refer to the location of each byte of memory. $2^{32} = 4.2$ billion, which means a memory address that's 32 bits long can only refer to 4.2 billion unique locations (i.e. 4 GB).
- * In the 32-bit Windows world, each application has its own "virtual" 4GB memory space. (This means that each application functions as if it has a flat 4GB of memory, and the system's memory manager keeps track of memory mapping, which applications are using which memory, page file management, and so on.)
- * This 4GB space is evenly divided into two parts, with 2GB dedicated for kernel usage, and 2GB left for application usage. Each application gets its own 2GB, but all applications have to share the same 2GB kernel space.
- * Even when more than 4GB of memory is present, each process still has the normal 2GB virtual address space, and the kernel address space is still 2GB

De-allocation of Memory

9. De-allocation of memory and the destruction of objects are also different in .NET.

Note:

COM uses reference counting for De-allocation of memory. Clients that share an object have to call `AddRef ()` to increment the counter. New COM objects are created with a reference count of one. When a client is done with an object, it calls `Release ()` to decrement the counter. When the reference count reaches zero, the object destroys itself.

10. .NET has a sophisticated garbage-collection mechanism that detects when an object is no longer being used by clients and then destroys it.
11. .NET keeps track of accessible paths to objects in the code.
12. .NET keeps track of each new object it allocates off the managed heap and of the relationship between this object and its clients.
13. .NET updates its graph of objects and adds a reference in the graph to that object from the object that created it.
14. The entity responsible for releasing unused memory is called the garbage collector.
15. When garbage collection is triggered, the garbage collector deems every object in the graphs as garbage.
16. The garbage collector then recursively traverses each graph, going down from the roots, looking for reachable objects.
17. Every time the garbage collector visits an object, it tags it as reachable. When the garbage collector is done traversing the graphs, it knows which objects were reachable and which were not.
18. Reachable objects should be kept alive. Unreachable objects are considered garbage, and therefore destroying them does no harm.
19. Next, the garbage collector scans the managed heap and disposes of the unreachable objects by compacting the heap and overwriting the unreachable objects with reachable one.
20. The garbage collector moves reachable objects down the heap, writing over the garbage, and thus frees more space at the end for new object allocations.
21. All unreachable objects are purged from the graphs.

Note:

Garbage collection is usually triggered in .NET by heap exhaustion, but application shutdown also triggers garbage collection.

Object Finalization

22. .NET objects are never told when they become garbage; they are simply overwritten when the managed heap is compacted. If the object holds expensive resources, how can it dispose of and release these resources? To address this problem, .NET provides object finalization.
23. If the object has specific cleanup to do, it should implement a method called `Finalize ()`, defined as:
Protected void Finalize ();
24. When the garbage collector decides that an object is garbage, it checks the object metadata. If the object implements the `Finalize ()` method, the garbage collector doesn't destroy the object. Instead, the

garbage collector marks the object as reachable (so it will not be overwritten by heap compaction), then moves the object from its original graph to a special queue called the finalization queue.

A separate thread iterates over all the objects in the finalization queue, calling the `Finalize()` method on each and letting the objects do their cleanup.

25. After calling `Finalize()`, the garbage collector removes the object from the queue.

Generations in GC

- * The .NET Garbage Collector defines generations to optimize its work.
- * Any object created since the last garbage collection operation is a generation 0 object.
- * Any object that has survived one GC operation is a generation 1 object.
- * Any object that has survived two or more GC operations is a generation 2 object.
- * The purpose of generations is to separate local variables and objects that stay around for the life of the application.
- * Generation 0 objects are mostly local variables. Member variables and global variables quickly enter generation 1 and eventually enter generation 2.
- * Every GC cycle examines generation 0 objects. Roughly 1 GC out of 10 examines the generation 0 and 1 objects. Roughly 1 GC cycle out of 100 examines all objects.
- * Think about finalization and its cost again: An object that requires finalization might stay in memory for nine GC cycles more than it would if it did not require finalization. If it still has not been finalized, it moves to generation 2. In generation 2, an object lives for an extra 100 GC cycles until the next generation 2 collection.
- * When a generation is collected, all younger generations are also collected.
- * A generation 2 garbage collection is also known as a full garbage collection.
- * From a generation point of view, large objects belong to generation 2 because they are collected only when there is a generation 2 collections.
- * Generations are the logical view of the garbage collector heap.
- * Physically, objects live on managed heap segments.

Minimize Boxing and Unboxing

- * Value types are containers for data.
- * `System.Object` is a single reference type, at the root of the entire object hierarchy.
- * The .NET Framework uses boxing and unboxing to bridge the gap between these two.
- * Boxing places a value type in an untyped reference object to allow the value type to be used where a reference type is expected.
- * Unboxing extracts a copy of that value type from the box.
- * Boxing converts a value type to a reference type.
- * A new reference object, the box, is allocated on the heap.
- * And a copy of the value type is stored inside that reference object.
- * Now, the box contains the copy of the value type object.
- * When you need to retrieve anything from the box, a copy of the value type gets created and returned.
- * A copy of the object goes in the box, and another gets created whenever you access what's in the box.

Even a simple statement such as this performs boxing:

```
Console.WriteLine ("A few numbers :{ 0}, {1}, {2}", 25, 32, 50);
```

In a sense, you have generated this construct:

```
int i =25;  
object o = i;           // box  
Console.WriteLine (o.ToString ());
```

Inside `Write Line`, the following code executes:

```
Object o;  
int i = ( int )o;      // unbox  
String output = i.ToString ( );
```

To avoid this particular penalty, you should convert your types to string instances yourself before you send them to `Write Line`:

```
Console.WriteLine ("A few numbers :{ 0}, {1}, {2}",  
25. ToString (), 32.ToString (), 50.ToString ());
```

Tip:
Minimize Boxing and Unboxing to avoid temporary copies of objects

Finalize () Method Implementation

There is much more to object finalization than meets the eye. In particular, you should note that calling `Finalize()` is nondeterministic in time. This may postpone the release of resources the object holds and threaten the scalability and performance of the application.

Here are a number of points to be kept in mind when implementing the `Finalize()` method:

- * When you implement `Finalize()`, it's important to call your base class's `Finalize()` method as well, to give the base class a chance to perform its cleanup:

```
Protected void Finalize ( )
{
    /* Object cleanup here */
    base.Finalize ( );
}
```

- * Make sure to define `Finalize()` as a protected method. Avoid defining `Finalize()` as a private method, because that prevents your subclasses from calling your `Finalize()` method.
- * Avoid making blocking calls, because you'll prevent finalization of all other objects in the queue until your blocking call returns.
- * Finalization must not rely on a specific order (e.g., Object A should release its resources only after Object B does). The two objects may be added to the finalization queue in any order.
- * It's important to call the base-class implementation of `Finalize()` even in the face of exceptions. You do so by placing the call in a try/finally statement:

```
Protected virtual void Finalize ( )
{
    try
    {
        /* Object cleanup here */
    }
    finally
    {
        base.Finalize( );
    }
}
```

Note:

Because these points are generic enough to apply to every class, the C# compiler has built-in support for generating template `Finalize()` code. In C#, you don't need to provide a `Finalize()` method; instead, you provide a C# destructor. The compiler converts the destructor definition to a `Finalize()` method, surrounding it in an exception-handling statement and calling your base class's `Finalize()` method automatically on your behalf.

Classes with finalizers require a minimum of two garbage collection cycles to be reclaimed. This prolongs the use of memory and can contribute to memory pressure. When the garbage collector encounters an unused object that requires finalization, it moves it to the "ready-to-be-finalized" list. Cleanup of the object's memory is deferred until after the single specialized finalizer thread can execute the registered

finalizer method on the object. After the finalizer runs, the object is removed from the queue.

Tip:

- * Implement Finalize() method only when you have un-managed resources to release.
- * Implementing Finalize() un-necessarily causes issues as objects stay in memory for longer duration.
- * If you Implement Finalize, Implement IDisposable

Implement the Standard Dispose Pattern

- * A standard pattern is used throughout the .NET Framework for disposing nonmemory resources
- * The standard dispose pattern disposes resources in deterministic way.
- * The standard dispose idiom frees your unmanaged resources using the IDisposable interface when clients remember, and it uses the finalizer defensively when clients forget.
- * The root base class in the class hierarchy should implement the IDisposable interface to free resources. This type should also add a finalizer as a defensive mechanism.
- * Your class must have a finalizer if it uses nonmemory resources. The only way you can guarantee that nonmemory resources get freed properly is to create a finalizer. So create one.

Note:

When the Garbage Collector runs, it immediately removes from memory any garbage objects that do not have finalizers. All objects that have finalizers remain in memory. These objects are added to a finalization queue, and the Garbage Collector spawns a new thread to run the finalizers on those objects. After the finalizer thread has finished its work, the garbage objects can be removed from memory. Objects that need finalization stay in memory for far longer than objects without a finalizer. The performance penalty associated with finalization can be avoided by implementing standard Dispose pattern.

- * The implementation of your IDisposable.Dispose() method is responsible for four tasks:
 - o Freeing all unmanaged resources.
 - o Freeing all managed resources (this includes unhooking events).
 - o Setting a state flag to indicate that the object has been disposed. You need to check this state and throw Object Disposed exceptions in your public methods, if any get called after disposing of an object.
 - o Suppressing finalization. You call GC.SuppressFinalize (this) to accomplish this task.
- * If derived classes override finalize or add their own implementation of IDisposable, those methods must call the base class; otherwise, the base class doesn't clean up properly.

- * finalize and Dispose share some of the same responsibilities.
- * Virtual void Dispose(bool is Disposing); This overloaded method does the work necessary to support both `finalize` and `Dispose`, and because it is virtual, it provides an entry point for all derived classes.
- * Derived classes can override this method, provide the proper implementation to clean up their resources, and call the base class version. You clean up managed and unmanaged resources when `is Disposing` is True; clean up only unmanaged resources when `is Disposing` is false. In both cases, call the base class's `Dispose (bool)` method to let it clean up its own resources.
- * Here is a short sample that shows the framework of code you supply when you implement this pattern. The `MyResourceHog` class shows the code to implement `IDisposable`, a finalizer, and create the virtual `Dispose` method:

```

public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    Private bool _already Disposed = false;

    // finalizer:
    // Call the virtual Dispose method.
    ~MyResourceHog ()
    {
        Dispose (false );
    }

    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose( true );
        GC.SuppressFinalize( true );
    }

    // Virtual Dispose method
    protected virtual void Dispose( bool isDisposing )
    {
        // Don't dispose more than once.
        if ( _alreadyDisposed )
            return;
        if ( isDisposing )
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.
        // Set disposed flag:
        _alreadyDisposed = true;
    }
}

```

```
}
```

If a derived class needs to perform additional cleanup, it implements the protected Dispose method:

```
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool _disposed = false;

    protected override void Dispose( bool isDisposing )
    {
        // Don't dispose more than once.
        if ( _disposed )
            return;
        if ( isDisposing )
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.

        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose( isDisposing );

        // Set derived class disposed flag:
        _disposed = true;
    }
}
```

Tip:

- * If an object implements Close() or Dispose() methods, it does so because it holds an expensive, shared, native resource that should be released as soon as possible. So, do not forget to call Close() or Dispose() on classes that support it.
- * Suppress Finalization in Dispose() method
- * If your class inherits from a disposable class, then make sure that it calls the base class's Dispose.
- * Also, if you have any member variables that implement IDisposable, call Dispose on them, too.

Utilize using and try/finally for Resource Cleanup

Types that use unmanaged system resources should be explicitly released using the `Dispose()` method of the `IDisposable` interface. The best way to ensure that `Dispose()` always gets called is to utilize the `using` statement or a `try/finally` block.

Suppose you wrote this code:

```
public void ExecuteCommand( string connString, string commandString )
{
    SqlConnection myConnection = new SqlConnection( connString );
    SqlCommand mySqlCommand = new SqlCommand( commandString,
        myConnection );

    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
}
```

Two disposable objects are not properly cleaned up in this example: `SqlConnection` and `SqlCommand`. Both of these objects remain in memory until their finalizers are called. (Both of these classes inherit their finalizer from `System.ComponentModel.Component`.)

You fix this problem by calling `Dispose` when you are finished with the command and the connection:

```
public void ExecuteCommand( string connString, string commandString )
{
    SqlConnection myConnection = new SqlConnection( connString );
    SqlCommand mySqlCommand = new SqlCommand( commandString,
        myConnection );

    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();

    mySqlCommand.Dispose( );
    myConnection.Dispose( );
}
```

That's fine, unless any exceptions get thrown while the SQL command executes. In that case, your calls to `Dispose()` never happen. The `using` statement ensures that `Dispose()` is called. You allocate an object inside a `using` statement, and the C# compiler generates a `try/finally` block around each object:

```
public void ExecuteCommand( string connString, string commandString )
{
    using ( SqlConnection myConnection = new
        SqlConnection( connString ))
    {
        using ( SqlCommand mySqlCommand = new
            SqlCommand( commandString,
                myConnection ))
        {
            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
    }
}
```

```
}  
}  
}
```

Whenever you use one `Disposable` object in a function, the `using` clause is the simplest method to use to ensure that objects get disposed of properly. The `using` statement generates a `TRY/finally` block around the object being allocated. These two blocks generate exactly the same IL:

```
SqlConnection myConnection = null;
```

```
// Example Using clause:  
using ( myConnection = new SqlConnection( connString ))  
{  
    myConnection.Open();  
}
```

```
// example Try / Catch block:  
try {  
    myConnection = new SqlConnection( connString );  
    myConnection.Open();  
}  
finally {  
    myConnection.Dispose( );  
}
```

Tip:

- * The `using` statement works only if the compile-time type supports the `IDisposable` interface. whereas `finally` blocks can be used for any type of cleanup operations.
- * A quick defensive `as` clause is all you need to safely dispose of objects that might or might not implement `IDisposable`:

```
// The correct fix.  
// Object may or may not support IDisposable.  
object obj = Factory.CreateResource( );  
using ( obj as IDisposable )  
    Console.WriteLine( obj.ToString( ));
```

- * Every `using` statement creates a new nested `TRY/finally` block.
- * `Dispose()` does not remove objects from memory. It is a hook to let objects release unmanaged resources. That means you can get into trouble by disposing of objects that are still in use. Do not dispose of objects that are still being referenced elsewhere in your program.

Always Use Properties Instead of Accessible Data Members

- * Properties let you expose data members as part of your public interface and still provide the encapsulation you want in an object-oriented environment.
- * Properties enable you to create an interface that acts like data access but still has all the benefits of a function. Client code accesses properties as though they are accessing public variables.
- * Any validations on data members could be made in one place by implementing properties.

```
public class Customer
{
    private string _name;
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (( value == null ) ||
                ( value.Length == 0 ))
                throw new ArgumentException( "Name cannot be blank",
                    "Name" );
            _name = value;
        }
    }

    // ...
}
```

- * Because properties are implemented with methods, adding multithreaded support is easier. Simply enhance the implementation of the `get` and `set` methods to provide synchronized access to the data:

```
public string Name
{
    get
    {
        lock( this )
        {
            return _name;
        }
    }
    set
    {
        lock( this )
        {
            _name = value;
        }
    }
}
```

```
}  
}
```

- * The property syntax extends beyond simple data fields. If your type should contain indexed items as part of its interface, you can use indexers (which are parameterized properties). It's a useful way to create a property that returns the items in a sequence:

```
public int this [ int index ]  
{  
    get  
    {  
        return _theValues [ index ] ;  
    }  
    set  
    {  
        _theValues[ index ] = value;  
    }  
}
```

```
// Accessing an indexer:  
int val = MyObject[ i ];
```

Prefer `readonly` to `const`

- * C# has two different versions of constants: compile-time constants and runtime constants.
- * Compile-time constants are slightly faster, but far less flexible, than runtime constants.
- * Reserve the compile-time constants for when performance is critical and the value of the constant will never change over time.
- * You declare runtime constants with the `readonly` keyword. Compile-time constants are declared with the `const` keyword:

```
// Compile time constant:  
public const int _Millennium = 2000;  
  
// Runtime constant:  
public static readonly int _ThisYear = 2004
```

- * The differences in the behavior of compile-time and runtime constants follow from how they are accessed. A compile-time constant is replaced with the value of that constant in your object code. This construct:

```
if ( myDateTime.Year == _Millennium )  
  
compiles to the same IL as if you had written this:  
if ( myDateTime.Year == 2000 )
```

- * Compile-time constants can be used only for primitive types (built-in integral and floating-point types), enums, or strings.
- * You cannot initialize a compile-time constant using the `new` operator, even when the type being initialized is a value type.
- * Compile-time constants are limited to numbers and strings.
- * Runtime constants can be of any type.
- * You must initialize runtime constants in a constructor, or you can use an initializer.
- * The final advantage of using `const` over `readonly` is performance: Known constant values can generate slightly more efficient code than the variable accesses necessary for `readonly` values.

Prefer the `is` or `as` Operators to Casts

Take a look at an example. You write a piece of code that needs to convert an arbitrary object into an instance of `MyType`. You could write it this way:

```
object o = Factory.GetObject( );

// Version one:
MyType t = o as MyType;

if ( t != null )
{
    // work with t, it's a MyType.
} else
{
    // report the failure.
}
```

Or, you could write this:

```
object o = Factory.GetObject( );

// Version two:
try {
    MyType t;
    t = ( MyType ) o;
    if ( t != null )
    {
        // work with T, it's a MyType.
    } else
    {
```

```
// Report a null reference failure.
}
} catch
{
// report the conversion failure.
}
```

The first version is simpler and easier to read. It does not have the try/catch clause, so you avoid both the overhead and the code.

Notice that the cast version must check null in addition to catching exceptions. null can be converted to any reference type using a cast, but the as operator returns null when used on a null reference. So, with casts, you need to check null and catch exceptions. Using as, you simply check the returned reference against null.

Remember that user-defined conversion operators operate only on the compile-time type of an object, not on the runtime type. It does not matter that a conversion between the runtime type of o2 and MyType exists. The compiler just doesn't know or care. This statement has different behavior, depending on the declared type of st:

```
t = ( MyType ) st;
```

This next statement returns the same result, no matter what the declared type of st is. So, you should prefer as to casts; it's more consistent. In fact, if the types are not related by inheritance, but a user-defined conversion operator exists, the following statement will generate a compiler error:

```
t = st as MyType;
```

The as operator does not work on value types. This statement won't compile:

```
object o = Factory.GetValue( );
int i = o as int; // Does not compile.
```

That's because ints are value types and can never be null. What value of int should be stored in i if o is not an integer? Any value you pick might also be a valid integer. Therefore, as can't be used. You're stuck with a cast.

But you're not necessarily stuck with the behaviors of casts. You can use the is statement to remove the chance of exceptions or conversions:

```
object o = Factory.GetValue( );
int i = 0;
if ( o is int )
    i = ( int ) o;
```

If o is some other type that can be converted to an int, such as a double, the is operator returns false. The is operator always returns false for null arguments.

The is operator should be used only when you cannot convert the type using as.

Always Provide ToString()

- * This string representation of a type can be used to easily display information about an object to users.
- * The string representation can also be useful for debugging.
- * Every type that you create should provide a reasonable override of this method.
- * The `System.Object` version returns the name of the type.
- * Every type you create should override `ToString()` to provide the most common textual representation of the type.

```
public class Customer
{
    private string    _name;
    private decimal  _revenue;
    private string    _contactPhone;
}
```

- * The inherited version of `Object.ToString()` returns "Customer". That is never useful to anyone.
- * Your override of `Object.ToString()` should return the textual representation most likely to be used by clients of that class. In the `Customer` example, that's the name:

```
public override string ToString()
{
    return _name;
}
```

- * Anytime the .NET FCL wants to get the string representation of a customer, your customer type supplies that customer's name.
- * Consider using `IFormattable.ToString()` for more complex implementations.

Ensure that 0 is a valid State for Value Types

- * The default .NET system initialization sets all objects to all 0s.
- * Never create an enum that does not include 0 as a valid choice.

```
public enum Planet
{
    // Explicitly assign values.
    // Default starts at 0 otherwise.
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Neptune = 7,
    Uranus = 8,
    Pluto = 9
}
```

- * `Planet sphere = new Planet();`

- * sphere is 0, which is not a valid value.

Consider overriding the `Equals()` method for value types

- * The `Equals` method is provided by `System.Object`.
- * To use the standard implementation of `Equals`, your value type must be boxed and passed as an instance of the reference type `System.ValueType`.
- * The `Equals` method then uses reflection to perform the comparison.
- * To avoid this overhead, better override `Equals` method.

Ex:

```
public struct Rectangle{
    public double Length;
    public double Breadth;
    public override bool Equals (object ob) {
        if(ob is Rectangle)
            return Equals((Rectangle)ob);
        else
            return false;
    }
    private bool Equals(Rectangle rect) {
        return this.Length == rect.Length && this.Breadth==rect.Breadth;
    }
}
```

Understand the Relationships Among `ReferenceEquals()`, `static Equals()`, `instance Equals()`, and `operator==`

- * C# provides four different functions that determine whether two different objects are "equal":

```
public static bool ReferenceEquals
    ( object left, object right );
public static bool Equals
    ( object left, object right );
public virtual bool Equals( object right);
public static bool operator==( MyClass left, MyClass right );
```

- * Two variables of a reference type are equal if they refer to the same object, referred to as object identity.
- * Two variables of a value type are equal if they are the same type and they contain the same contents.
- * `Object.ReferenceEquals()` returns `True` if two variables refer to the same object that is, the two variables have the same object identity.

- * Whether the types being compared are reference types or value types, this method always tests object identity, not object contents. Yes, that means that `ReferenceEquals()` always returns `false` when you use it to test equality for value types. Even when you compare a value type to itself,

`ReferenceEquals()` returns `false`. This is due to boxing.

```
int i = 5;
int j = 5;
if ( Object.ReferenceEquals( i, j ))
    Console.WriteLine( "Never happens." );
else
    Console.WriteLine( "Always happens." );
```

```
if ( Object.ReferenceEquals( i, i ))
    Console.WriteLine( "Never happens." );
else
    Console.WriteLine( "Always happens." );
```

- * You'll never redefine `Object.ReferenceEquals()` because it does exactly what it is supposed to do: test the object identity of two different variables.
- * `static Object.Equals()`. This method tests whether two variables are equal when you don't know the runtime type of the two arguments.
- * As with `ReferenceEquals()`, you'll never redefine the static `Object.Equals()` method because it already does exactly what it needs to do: determines whether two objects are the same when you don't know the runtime type.

```
public static bool Equals( object left, object right )
{
    // Check object identity
    if (left == right )
        return true;
    // both null references handled above
    if ((left == null) || (right == null))
        return false;
    return left.Equals (right);
}
```

Avoid pre-allocating memory

Allocation of managed memory is a quick operation and the garbage collector has been optimized for extremely fast allocations. The main reason for preallocating memory in unmanaged code is to speed up the allocation process. This is not an issue for managed code.

Prefer Variable Initializers to Assignment Statements

Classes often have more than one constructor. In such cases, it easy for a member variables and constructors to get out of synch. The best way to make sure this doesn't happen is to initialize variables where you declare them instead of in the body of every constructor.

Utilize the initializer syntax for both static and instance variables:

```
public class MyClass
{
    // declare the collection, and initialize it.
    private ArrayList _coll = new ArrayList( );
}
```

Regardless of the number of constructors you eventually add to the `MyClass` type, `_coll` will be initialized properly.

The compiler generates code at the beginning of each constructor to execute all the initializers you have defined for your instance member variables.

The initializers are added to the compiler-generated default constructor. So, even if you don't have constructor defined explicitly, the variable is already initialized with the help of initializers.

Using initializers is the simplest way to avoid uninitialized variables in your types.

You should not use initializer when:

1. you are initializing the object to 0, or null. The default system initialization sets everything to 0 for you before any of your code executes. The system-generated 0 initialization is done at a very low level using the CPU instructions to set the entire block of memory to 0. Any extra 0 initialization on your part is redundant and becomes inefficient.
2. same initialization doesn't happen on all constructors.

```
public class MyClass
{
    // declare the collection, and initialize it.
    private ArrayList _coll = new ArrayList( );

    MyClass( )
    {
    }

    MyClass( int size )
    {
        _coll = new ArrayList( size );
    }
}
```

When you create a new `MyClass`, specifying the size of the collection, you create two array lists.

One is immediately garbage. The constructor body creates the second array list.

3. Initialization requires exception handling. You cannot wrap the initializers in a `TRY` block.

Initialize Static Class Members with Static Constructors

- * You should initialize static member variables in a type before you create any instances of that type.
- * C# lets you use static initializers and a static constructor for this purpose.
- * A static constructor is a special function that executes before any other methods, variables, or properties defined in that class are accessed.
- * As with instance initialization, you can use the initializer syntax as an alternative to the static constructor. If you simply need to allocate a static member, use the initializer syntax. When you have more complicated logic to initialize static member variables, create a static constructor.
- * As with instance initializers, the static initializers are called before any static constructors are called. And, yes, your static initializers execute before the base class's static constructor.

Utilize Constructor Chaining

- * When your class is having multiple constructors defined, member initialization gets repeated among all the constructors. C++ programmers would factor the common algorithms into a private helper method. Initializing read-only variables is not possible with this approach as it can be done only in the constructor.
- * When you find that multiple constructors contain the same logic, factor that logic into a common constructor.
- * With constructor chaining, you'll get the benefits of avoiding code duplication, and constructor initializers generate much more efficient object code.
- * Constructor initializers allow one constructor to call another constructor. This example shows a simple usage:

```
public class MyClass
{
    // collection of data
    private ArrayList _coll;
    // Name of the instance:
    private string _name;

    public MyClass() :
        this( 0, "" )
    {
    }

    public MyClass( int initialCount ) :
        this( initialCount, "" )
    {
    }

    public MyClass( int initialCount, string name )
```

```
{
    _coll = ( initialCount > 0 ) ?
        new ArrayList( initialCount ) :
        new ArrayList();
    _name = name;
}
```

Prefer Small, Simple Functions

- * Translating your C# code into machine-executable code is a two-step process.
- * The C# compiler generates IL that gets delivered in assemblies.
- * The JIT compiler generates machine code for each method, as needed.
- * Small functions make it much easier for the JIT compiler to amortize that cost.

Avoid inefficient string operations

- * Avoid inefficient string concatenation.
- * Use + when the number of appends is known.
- * Use StringBuilder when the number of appends is unknown.
- * Treat StringBuilder as an accumulator.
- * Use the overloaded Compare method for case insensitive string comparisons.

Prefer Single Large Assemblies Rather Than Multiple Smaller Assemblies

Overheads associated with smaller assemblies:

- * The cost of loading metadata for smaller assemblies
- * JIT Compile Time
- * Security Checks

Locking and Synchronization guidelines

- * Acquire locks late and release them early
- * Avoid locking and synchronization unless required
- * Use granular locks to reduce contention.

- * Avoid excessive fine grained locks
- * Use the fine-grained lock(C#) statement instead of Synchronized
- * Avoid locking "this"

Minimize Thread Creation

Threads use both managed and unmanaged resources and are expensive to initialize and may result in the processor spending most of its time performing thread switches; it also places increased pressure on the garbage collector to cleanup resources. use the ThreadPool when you need threads.

Do not set local variables to null.

JIT compiler can statically determine that the variable is no longer referenced and there is no need to explicitly set it to null. And do not forget to set Unneeded Member variables to Null before making long-running calls

Miscellaneous

1. Avoid putting multiple classes in a single file.
2. A single file should contribute types to only a single namespace. Avoid having multiple namespaces in the same file.
3. Avoid files with more than 500 lines (excluding machine-generated code).
4. Avoid methods with more than 25 lines.
5. Avoid methods with more than five arguments. Use structures for passing multiple arguments.
6. Lines should not exceed 80 characters.
7. Do not manually edit any machine-generated code.
 - a. If modifying machine-generated code, modify the format and style to match this coding standard.
 - b. Use partial classes whenever possible to factor out the maintained portions.
8. Avoid comments that explain the obvious. Code should be self-explanatory. Good code with readable variable and method names should not require comments.
9. Document only operational assumptions, algorithm insights, and so on.
10. Avoid method-level documentation.
 - a. Use extensive external documentation for API documentation.
 - b. Use method-level comments only as tool tips for other developers.

11. With the exception of zero and one, never hardcode a numeric value; always declare a constant instead.
12. Use the const directive only on natural constants, such as the number of days of the week.
13. Avoid using const on read-only variables. For that, use the readonly directive:

```
public class MyClass
{
    public const int DaysInWeek = 7;
    public readonly int Number;
    public MyClass(int someValue)
    {
        Number = someValue;
    }
}
```

14. Assert every assumption. On average, every fifth line is an assertion:

```
using System.Diagnostics;

object GetObject( )
{...}

object someObject = GetObject( );
Debug.Assert(someObject != null);
```

15. Every line of code should be walked through in a "white box" testing manner.
16. Catch only exceptions for which you have explicit handling.
17. In a catch statement that throws an exception, always throw the original exception (or another exception constructed from the original exception) to maintain the stack location of the original error:

```
catch(Exception exception)
{
    MessageBox.Show(exception.Message);
    throw; //Same as throw exception;
}
```

18. Avoid error code as method return values.
19. Avoid defining custom exception classes.
20. When defining custom exceptions:
 - a. Derive the custom exception from Exception.
 - b. Provide custom serialization.
21. Avoid multiple Main() methods in a single assembly.
22. Make only the most necessary types public; mark others as internal.
23. Avoid friend assemblies, as they increase interassembly coupling.
24. Avoid code that relies on an assembly running from a particular location.
25. Minimize code in application assemblies (i.e., EXE client assemblies). Use class libraries instead to contain business logic.

26. Avoid providing explicit values for enums:

```
//Correct
public enum Color
{
    Red,Green,Blue
}
//Avoid
public enum Color
{
    Red = 1,Green = 2,Blue = 3
}
```

27. Avoid specifying a type for an enum:

```
//Avoid
public enum Color : long
{
    Red,Green,Blue
}
```

28. Always use a curly brace scope in an if statement, even if it contains a single statement.

29. Avoid using the trinary conditional operator.

30. Avoid function calls in Boolean conditional statements. Assign into local variables and check on them:

```
bool IsEverythingOK( )
{...}
//Avoid:
if(IsEverythingOK( ))
{...}
//Correct:
bool ok = IsEverythingOK( );
if(ok)
{...}
```

31. Always use zero-based arrays.

32. Always explicitly initialize an array of reference types:

```
public class MyClass
{}
const int ArrraySize = 100;
MyClass[] array = new MyClass[ArrraySize];
for(int index = 0; index < array.Length; index++)
{
    array[index] = new MyClass( );
}
```

33. Do not provide public or protected member variables. Use properties instead.

34. Avoid using the new inheritance qualifier. Use override instead.

35. Always mark public and protected methods as virtual in a non-sealed class.

36. Never use unsafe code, except when using interop.

37. Avoid explicit casting. Use the as operator to defensively cast to a type:

```
Dog dog = new GermanShepherd( );
GermanShepherd shepherd = dog asGermanShepherd;
```

```
if(shepherd != null)
{...}
```

38. Always check a delegate for null before invoking it.
39. Do not provide public event member variables. Use event accessors instead.
40. Avoid defining event-handling delegates. Use `GenericEventHandler` instead.
41. Avoid raising events explicitly. Use `EventsHelper` to publish events defensively.
42. Always use interfaces.
43. Classes and interfaces should have at least a 2:1 ratio of methods to properties.
44. Avoid interfaces with one member.
45. Strive to have three to five members per interface.
46. Do not have more than 20 members per interface. The practical limit is probably 12.
47. Avoid events as interface members.
48. When using abstract classes, offer an interface as well.
49. Expose interfaces on class hierarchies.
50. Prefer using explicit interface implementation.
51. Never assume a type supports an interface. Defensively query for that interface:

```
SomeType obj1;
IMyInterface obj2;

/* Some code to initialize obj1, then: */
obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
    obj2.Method1( );
}
else
{
    //Handle error in expected interface
}
```

52. Never hardcode strings that will be presented to end users. Use resources instead.
53. Never hardcode strings that might change based on deployment, such as connection strings.
54. Use `String.Empty` instead of `""`:

```
//Avoid
string name = "";

//Correct
string name = String.Empty;
```

55. When building a long string, use `StringBuilder`, not `string`.
56. Avoid providing methods on structures.

- a. Parameterized constructors are encouraged.
 - b. You can overload operators.
57. Always provide a static constructor when providing static member variables.
58. Do not use late-binding invocation when early binding is possible.
59. Use application logging and tracing.
60. Never use goto, except in a switch statement fall-through.
61. Always have a default case in a switch statement that asserts:

```
int number = SomeMethod( );
switch(number)
{
    case 1:
        Trace.WriteLine("Case 1:");
        break;
    case 2:
        Trace.WriteLine("Case 2:");
        break;
    default:
        Debug.Assert(false);
        break;
}
```

62. Do not use the this reference unless invoking another constructor from within a constructor:

```
//Example of proper use of 'this'
public class MyClass
{
    public MyClass(string message)
    {}
    public MyClass( ) : this("Hello")
    {}
}
```

63. Do not use the base word to access base class members unless you wish to resolve a conflict with a subclass member of the same name or when invoking a base class constructor:

```
//Example of proper use of 'base'
public class Dog
{
    public Dog(string name)
    {}
    virtual public void Bark(int howLong)
    {}
}
public class GermanShepherd : Dog
{
    public GermanShepherd(string name) : base(name)
    {}
    override public void Bark(int howLong)
```

```

    {
        base.Bark(howLong);
    }
}

```

64. Do not use `GC.AddMemoryPressure()`.

65. Do not rely on `HandleCollector`.

66. Implement `Dispose()` and `Finalize()` methods based on the template in Chapter 4.

67. Always run code unchecked by default (for the sake of performance), but explicitly in checked mode for overflow- or underflow-prone operations:

```

int CalcPower(int number,int power)
{
    int result = 1;
    for(int count = 1;count <= power;count++)
    {
        checked
        {
            result *= number;
        }
    }
    return result;
}

```

68. Avoid explicit code exclusion of method calls (`#if...#endif`). Use conditional methods instead:

```

public class MyClass
{
    [Conditional("MySpecialCondition")]
    public void MyMethod( )
    {}
}

```

69. Avoid casting to and from `System.Object` in code that uses generics. Use constraints or the `as` operator instead:

```

class SomeClass
{}
//Avoid:
class MyClass<T>
{
    void SomeMethod(T t)
    {
        object temp = t;
        SomeClass obj = (SomeClass)temp;
    }
}
//Correct:
class MyClass<T> where T : SomeClass
{
    void SomeMethod(T t)
    {
        SomeClass obj = t;
    }
}

```

```
}
```

70. Do not define constraints in generic interfaces. Interface-level constraints can often be replaced by strong typing:

```
public class Customer
{...}
//Avoid:
public interface IList<T> where T : Customer
{...}
//Correct:
public interface ICustomerList : IList<Customer>
{...}
```

71. Do not define method-specific constraints in interfaces.
72. If a class or a method offers both generic and non-generic flavors, always prefer using the generics flavor.
73. When implementing a generic interface that derived from an equivalent non-generic interface (such as `IEnumerable<T>`), use explicit interface implementation on all methods, and implement the non-generic methods by delegating to the generic ones:

```
class MyCollection<T> : IEnumerable<T>
{
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {...}
    IEnumerator IEnumerable.GetEnumerator()
    {
        IEnumerable<T> enumerable = this;
        return enumerable.GetEnumerator();
    }
}
```

Multithreading

74. Use synchronization domains. Avoid manual synchronization, because that often leads to deadlocks and race conditions.
75. Never call outside your synchronization domain.
76. Manage asynchronous call completion on a callback method. Do not wait, poll, or block for completion.
77. Always name your threads:

```
Thread currentThread = Thread.CurrentThread;
string threadName = "Main UI Thread";
currentThread.Name = threadName;
```

The name is traced in the debugger Threads window, making debug sessions more productive.

78. Do not call `Suspend()` or `Resume()` on a thread.
79. Do not call `Thread.Sleep()`, except in the following conditions:
80. `Thread.Sleep(0)` is an acceptable optimization technique to force a context switch.
81. `Thread.Sleep()` is acceptable in testing or simulation code.
82. Do not call `Thread.SpinWait()`.

83. Do not call `Thread.Abort()` to terminate threads. Use a synchronization object instead to signal the thread to terminate.
84. Avoid explicitly setting the thread priority to control execution. You can set the thread priority based on task semantics (such as `ThreadPriority.BelowNormal` for a screensaver).
85. Do not read the value of the `ThreadState` property. Use `Thread.IsAlive()` to determine whether the thread is dead or alive.
86. Do not rely on setting the thread type to background thread for application shutdown. Use a watchdog or other monitoring entity to deterministically kill threads.
87. Do not use the thread local storage unless thread affinity is guaranteed.
88. Do not call `Thread.MemoryBarrier()`.
89. Never call `Thread.Join()` without checking that you are not joining your own thread:

```
void WaitForThreadToDie(Thread thread)
{
    Debug.Assert(Thread.CurrentThread.ManagedThreadId != thread.ManagedThreadId);
    thread.Join( );
}
```

90. Always use the `lock()` statement rather than explicit `Monitor` manipulation.
91. Always encapsulate the `lock()` statement inside the object it protects:

```
public class MyClass
{
    public void DoSomething( )
    {
        lock(this)
        {...}
    }
}
```

92. You can use `synchronized` methods instead of writing the `lock()` statement yourself.
93. Avoid fragmented locking.
94. Avoid using a `Monitor` to wait or pulse objects. Use manual or auto-reset events instead.
95. Do not use volatile variables. Lock your object or fields instead to guarantee deterministic and thread-safe access. Do not use `Thread.VolatileRead()`, `Thread.VolatileWrite()`, or the volatile modifier.
96. Avoid increasing the maximum number of threads in the thread pool.
97. Never stack `lock()` statements, because that does not provide atomic locking:

```
MyClass obj1 = new MyClass( );
MyClass obj2 = new MyClass( );
MyClass obj3 = new MyClass( );

//Do not stack lock statements
lock(obj1)
lock(obj2)
lock(obj3)
{
```

```
obj1.DoSomething( );
obj2.DoSomething( );
obj3.DoSomething( );
}
```

Use `WaitHandle.WaitAll()` instead.

Serialization

- 98. Prefer the binary formatter.
- 99. Mark serialization event-handling methods as private.
- 100. Use the generic `IGenericFormatter` interface.
- 101. Always mark non-sealed classes as serializable.
- 102. When implementing `IDeserializationCallback` on a non-sealed class, make sure to do so in a way that allows subclasses to call the base class implementation of `OnDeserialization()`.
- 103. Always mark unserializable member variables as non-serializable.
- 104. Always mark delegates on a serialized class as non-serializable fields:

```
[Serializable]
public class MyClass
{
    [field:NonSerialized]
    public event EventHandler MyEvent;
}
```

Remoting

- 105. Prefer administrative configuration to programmatic configuration.
- 106. Always implement `IDisposable` on single-call objects.
- 107. Always prefer a TCP channel and a binary format when using remoting, unless a firewall is present.
- 108. Always provide a null lease for a singleton object:

```
public class MySingleton : MarshalByRefObject
{
    public override object InitializeLifetimeService( )
    {
        return null;
    }
}
```

- 109. Always provide a sponsor for a client-activated object. The sponsor should return the initial lease time.
- 110. Always unregister the sponsor on client application shutdown.
- 111. Always put remote objects in class libraries.
- 112. Avoid using `SoapSuds.exe`.

113. Avoid hosting in IIS.

114. Avoid using uni-directional channels.

115. Always load a remoting configuration file in Main(), even if the file is empty and the application does not use remoting:

```
static void Main( )
{
    RemotingConfigurationEx.Configure( );
    /* Rest of Main( ) */
}
```

116. Avoid using Activator.GetObject() and Activator.CreateInstance() for remote object activation. Use new instead.

117. Always register port 0 on the client side, to allow callbacks.

118. Always elevate type filtering to Full on both client and host, to allow callbacks.

Security

119. Always demand your own strong name on assemblies and components that are private to the application, but are public (so that only you can use them):

```
public class PublicKeys
{
    public const string MyCompany = "1234567894800000940000000602000000240000"+
        "52534131000400000100010007D1FA57C4AED9F0"+
        "A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C83"+
        "4C99921EB23BE79AD9D5DCC1DD9AD23613210290"+
        "0B723CF980957FC4E177108FC607774F29E8320E"+
        "92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99"+
        "285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF"+
        "0FC4963D261C8A12436518206DC093344D5AD293";
}

[StrongNameIdentityPermission(SecurityAction.LinkDemand,
    PublicKey = PublicKeys.MyCompany)]
public class MyClass
{...}
```

120. Apply encryption and security protection on application configuration files.

121. When importing an interop method, assert unmanaged code permission and demand appropriate permission instead:

```
[DllImport("user32",EntryPoint="MessageBoxA")]
private static extern int Show(IntPtr handle,string text,string caption,
    int msgType);
[SecurityPermission(SecurityAction.Assert,UnmanagedCode = true)]
[UIPermission(SecurityAction.Demand,
    Window = UIPermissionWindow.SafeTopLevelWindows)]
public static void Show(string text,string caption)
{
    Show(IntPtr.Zero,text,caption,0);
}
```

122. Do not suppress unmanaged code access via the SuppressUnmanagedCodeSecurity attribute.
123. Do not use the /unsafe switch of TlbImp.exe. Wrap the RCW in managed code so that you can assert and demand permissions declaratively on the wrapper.
124. On server machines, deploy a code access security policy that grants only Microsoft, ECMA, and self (identified by a strong name) full trust. Code originating from anywhere else is implicitly granted nothing.
125. On client machines, deploy a security policy that grants client application only the permissions to execute, to call back the server, and to potentially display user interface. When not using ClickOnce, client application should be identified by a strong name in the code groups.
126. To counter a luring attack, always refuse at the assembly level all permissions not required to perform the task at hand:

```
[assembly:UIPermission(SecurityAction.RequestRefuse,  
    Window=UIPermissionWindow.AllWindows)]
```

127. Always set the principal policy in every Main() method to Windows:

```
public class MyClass  
{  
    static void Main( )  
    {  
        AppDomain currentDomain = AppDomain.CurrentDomain;  
        currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);  
    }  
    //other methods  
}
```

128. Never assert a permission without demanding a different permission in its place.