
Best practices

You could spend almost unlimited amounts of time optimizing your code for better performance or memory consumption if you had nothing else to think about. However, other concerns, such as maintainability, security and productivity, *must* be considered. Often the best practice for one goal is not as applicable for another; indeed, in many cases, the opposite is true. The question that you clearly need to consider is "What is the most efficient method for optimizing my code, taking these competing goals into account?" The list below outlines some sensible steps that should be taken to ensure well-behaved code in most common situations, and at minimal cost to other concerns. You may not be familiar with some of these terms or concepts now, but you will be, by the end of this section.

- Use the `IDisposable` interface with the Disposable pattern to release unmanaged resources, and suppress finalization in the Dispose method if no other finalization is required.
- Use the `using` statement to define the scope of a disposable object, unless doing so could cause an exception.
- Use `StringBuilder` when the number of string concatenations is unknown or contained within a loop.
- Initialize the capacity of a `StringBuilder` to a reasonable value, if possible.
- Use **lazy initialization** to defer the cost of instantiating members of a class if the members are not needed when the object is created.
- Use a `struct` to represent any immutable, single value whose instance size is 16 bytes or smaller and which will be infrequently boxed.
- Override the `GetHashCode` and `Equals` methods of a `struct.`
- Mark all fields in a `struct` as `readonly.`
- Use the `readonly` keyword on class fields that should be immutable.
- Implement `INotifyPropertyChanged` on component classes that may be used by binding clients.
- Avoid maintaining **large object graphs** in memory.
- Avoid using **unsigned number types** (except for byte).
- Prefer **CLS-compliant** types.
- Be aware of **variable capture** when using anonymous functions.
- Do not pass delegates to `IQueryable` extension methods.
- Avoid excessive grouping or aggregate functions in LINQ queries.
- Use **static methods** when it is unnecessary to access or mutate state.
- Initialize collections (such as `List<T>`) with the target size, if it is known.
- Consider using content instead of embedded resources for larger artifacts.
- In an IIS hosted application, set the `//compilation/system.web/compilation/ debug` attribute to `false` before releasing to production.
- Remove event handlers before disposing of an object.