13 Things Every C# Developer Should Know

By Eric Lippert and Jon Jarboe

1. DEVELOPMENT PROCESS

The development process is where bugs and defects start. Take advantage of tools that help you avoid or find these problems before you release:

Coding Standards
Consistent use of a standard can lead to more maintainable code, especially in code bases written and maintained by multiple developers or teams. Tools such as FxCop, StyleCop, and ReSharper are commonly used to enforce coding standards.

Developers: Carefully consider violations and analysis results before suppressing them. They identify problems in code paths that are less unusual than you expect.

Code Review
Code review and pair programming are common practices that task developers with deliberately reviewing source code written by others. Others will hopefully recognize mistakes made by the author, such as coding or implementation bugs.

Code review is a valuable practice, but it is fallible by nature of relying on humans and can be difficult to scale.

Static Analysis
Static analysis tools analyze your code without running it, looking for problems like violations of coding standards or the existence of defects without requiring you to write test cases. It is effective at finding problems, but you need to choose tools that identify valuable problems without too many false positives. C# static analysis tools include Coverity, CAT.NET, and Visual Studio Code Analysis.

Dynamic Analysis
Dynamic analysis tools analyze your code while it is running, helping you look for defects such as security vulnerabilities, performance and concurrency problems. It analyzes the code in the context of the runtime environment, so its effectiveness is limited by the testing workload. Visual Studio provides a number of dynamic analysis tools, including the Concurrency Visualizer, IntelliTrace, and Profiling Tools.

Managers/Team Leads: Leverage development best practices to avoid common pitfalls. Carefully consider available tools to ensure they are compatible with your needs and culture. Commit your team to keeping the diagnostic noise level manageable.

Testing
There are many types of tests, such as: unit tests, system integration tests, performance tests, penetration tests. In the development phase, most tests are written by developers or testers to verify the application meets its requirements.

Tests are effective only to the extent that they exercise the right code. It can be challenging to maintain development velocity while implementing both functionality and tests.

Development Best Practices
Invest the time to identify and configure tools to find problems you care about, without creating extra work for developers. Run analysis tools and tests frequently and automatically, to ensure developers address problems while the code is still fresh in mind.

Address all diagnostic output—whether it's compiler warnings, standards violations, defects identified through static analysis, or testing failures—as quickly as possible. If interesting new diagnostics get lost in a sea of "don't cares" or ignored diagnostics, the effort of reviewing results will increase until developers no longer bother.

Adopting these best practices helps improve the quality, security, and maintainability of your code as well as the consistency and productivity of developers and predictability of releases.

CONCERN	TOOL	IMPACT
Consistency, Maintainability	Coding standards, static analysis, code review	Consistent spacing, naming, and formatting improve readability and make it easier for developers to write and maintain code.
Correctness	Code review, static analysis, dynamic analysis, testing	Code needs to not only be syntactically valid, but it must behave as the developer intends and meet project requirements.
Functionality	Testing	Tests verify that code meets requirements such as correctness, scalability, robustness, and security.
Security	Coding standards, code review, static analysis, dynamic analysis, testing	Security is a very complex problem; any weakness or defect can potentially be exploited.
Developer productivity	Coding standards, static analysis, testing	Developers implement code changes more quickly when they have tools to identify mistakes.
Release predictability	Coding standards, code review, static analysis, dynamic analysis, testing	Streamline late-phase activity and minimize fix cycles by addressing defects and problems early.

2. TYPE GOTCHAS

One of C#'s major strengths is its flexible type system; type safety helps catch errors early. By enforcing strict type rules, the compiler is able to help you maintain sane coding practices.

The C# language and .NET framework provide a rich collection of types to accommodate the most common needs. Most developers have a good understanding of the common types and their uses, but there are some common misunderstandings and misuses.

More information about the .NET Framework Class Library can be found in the MSDN library.

Understand and use the standard interfaces
Certain interfaces relate to idiomatic C# features. For example, IDisposable allows the use of common resource-handling idioms such as the "using" keyword. Understanding when to use the interfaces will enable you to write idiomatic C# code that is easier to maintain.

Avoid ICloneable—the designers never made it clear whether a cloned object was intended to be a deep or shallow copy. Since there is no standard for correct behavior of a cloned object, there's no ability to meaningfully use the interface as a contract.

Structures
Try to avoid writing to structs. Treating them as immutable prevents confusion, and is much safer in shared memory scenarios like multithreaded applications. Instead, use initializers when you create structs and create new instances if you need to change the values.

Understand which standard types/methods are immutable and return new values (for example, string, DateTime), versus those that are mutable (List.Enumerator).

Strings
Strings may be null, so use the convenience functions when appropriate. Evaluating (s.Length==0) may throw a NullReferenceException, while String.IsNullOrEmpty(s) and String.IsNullOrWhitespace(s) gracefully handle null.

Flagged enumerations
Enumerated types and constant values help to make the code more readable by replacing magic numbers with identifiers that expose the meaning of the value.

If you find that you need to create a collection of enums, a flagged enum might be a simpler choice:

```
[Flag]
public enum Tag {
  None   =0x0,
  Tip    =0x1,
  Example=0x2
}
public class Snippet {
  public Tag Tag {get;set;};
}
```

This enables you to easily have multiple tags for a snippet:

```
snippet.Tag = Tag.Tip | Tag.Example
```

This can improve data encapsulation, since you don't have to worry about exposing an internal collection via the Tag property getter.

Equality comparisons
There are two types of equality:

1. Reference equality, which means that two references refer to the same object.
2. Value equality, which means that two referentially distinct objects should be considered as equal.

Moreover, C# provides multiple ways to test for equality. The most common techniques are to use:

- The == and != operators
- The virtual Equals method inherited from Object
- The static Object.Equals method
- The Equals method of the IEquatable<T> interface
- The static Object.ReferenceEquals method

It can be difficult to know whether reference or value equality is intended. Review the MSDN equality topic to ensure your comparison works as expected: http://msdn.microsoft.com/en-us/library/dd183752.aspx

If you override Equals, don't forget IEquatable<T>, GetHashCode(), and so on as described in MSDN.

Beware the impact of untyped containers on overloads. Consider the comparison "myArrayList[0] == myString". The array list element is of compile-time type "object," so reference equality is used. The C# compiler will warn you about this potential error, but there are many similar situations where the compiler does not warn about unexpected reference equality.

3. CLASS GOTCHAS

Encapsulate your data
Classes are responsible for managing data properly. For performance reasons they often cache partial results or otherwise make assumptions about the consistency of their internal data. Making data publicly accessible compromises your ability to cache or make assumptions—with potential impacts on performance, security, and concurrency. For example, exposing mutable members, like generic collections and arrays, allows users to modify those structures without your knowledge.

Properties
Properties enable you to control exactly how users can interact with your object, beyond what you can control through access modifiers. Specifically, properties enable you to control what happens on read and write.

Properties enable you to establish a stable API while overriding data access logic in the getters and setters, or to provide a data binding source.

Never throw an exception from a property getter, and avoid modifying object state. Such desires imply a need for a method rather than a property getter.

For more information about properties, see MSDN's property design topic: http://msdn.microsoft.com/en-us/library/ms229006(v=vs.120).aspx

Be careful with getters that have side effects. Developers are conditioned to believe that member access is a trivial operation, so they often forget to consider side effects during code reviews.

Object initializers
You can set properties on a newly created object from within the creation expression itself. To create a new object of class C with specified values for the Foo and Bar properties:

```
new C {Foo=blah, Bar=blam}
```

You can also create instances of anonymous types with specific property names:

```
var myAwesomeObject = new {Name="Foo", Size=10};
```

Initializers execute before the constructor body runs, ensuring that fields are initialized before entering the constructor. Because the constructor has not yet run, a field initializer may not refer to "this"

in any way.

Over-specifying input parameters
To help prevent proliferation of specialized methods, try to use the least specific type needed by the method. For example, consider a method that iterates over a List<Bar>:

```
public void Foo(List<Bar> bars)
{
  foreach(var b in bars)
  {
    // do something with the bar...
  }
}
```

This code should work perfectly well for other IEnumerable<Bar> collections, but by specifying List<Bar> for the parameter, you require that the collection be a List. Choose the least specific type (IEnumerable<T>, ICollection<T>, and so on) for the parameter to ensure maximal usefulness of the method.

4. GENERICS

Generics are a powerful way of defining type-independent structures and algorithms that can enforce type safety.

Use generics collections such as List<T> instead of untyped collections such as ArrayList to improve both type safety and performance.

When implementing a generic type, you can use the "default" keyword to get the default value for a type whose default value cannot be hardcoded into the implementation. Specifically, numeric types have a default value of 0; reference and nullable value types have a default value of null.

```
T t = default(T);
```

5. CONVERSIONS AND CASTS

There are two types of conversions. Explicit conversions must be invoked by the developer, and implicit conversions are applied by the compiler based on context.

Constant 0 is implicitly convertible to enum. When you're trying to call a method that takes a number, you may end up calling a method that takes an enum.

CAST	DESCRIPTION
Tree tree = (Tree)obj;	Use this when you expect that obj will only ever be of type Tree. If obj is not a Tree, an InvalidCast exception will be raised.
Tree tree = obj as Tree;	Use this when you anticipate that obj may or may not be a Tree. If obj is not a Tree, a null value will be assigned to tree. Always follow an "as" cast with conditional logic to properly handle the case where null is returned. Only use this style of conversion when necessary, since it necessitates conditional handling of the return value. This extra code creates opportunities for more bugs and makes the code more difficult to read and debug.

Casting usually means one of two things:

1. You know that the runtime type of an expression will be more specific than the compiler can deduce. The cast instructs the compiler to treat the expression as the more specific type. The compiler will generate code that throws an exception if your assumption was incorrect. For example, a conversion from object to string.
2. You know that there is a value of a completely different type associated with the value of the expression. The cast instructs the compiler to generate code that produces this associated value, or throws an exception if there is none. For example, a conversion from double to integer.

Both kinds of casts are red flags. The first kind of cast raises the question, "why exactly is it that the developer knows something that the compiler doesn't?" If you are in that situation, try to change the program so that the compiler can successfully deduce the correct type. If you think that perhaps the runtime type of an object is of a more specific type than the compile time type, then you can use the "is" or "as" operators.

The second kind of cast raises the question, "why isn't the operation being done in the target data type in the first place?" If you need a result of type int, it might make more sense to use an int than a double.

For additional thoughts see: http://blogs.msdn.com/b/ericlippert/archive/tags/cast+operator/

In cases where an explicit conversion is the right thing to do, improve readability, debug-ability, and testability by using the appropriate operator.

6. EXCEPTIONS

Exceptions are not conditions
Exceptions should generally not be used to control program flow; they represent unexpected circumstances at runtime from which you may not be able to recover. If you anticipate a circumstance that you should handle, proactively check for the circumstance rather than waiting for an exception to fire.

To gracefully convert unreliably formatted strings to numbers, use the TryParse() method; rather than throwing an exception, it returns a Boolean indicating whether the parsing was successful.

Use care with exception handling scope
Write code inside catch and finally blocks carefully. Control might be entering these blocks due to an unexpected exception; code that you expected to have executed already might have been skipped by the exception. For example:

```
Frobber originalFrobber = null;
try {
  originalFrobber = this.GetCurrentFrobber();
  this.UseTemporaryFrobber();
  this.frobSomeBlobs();
}
finally {
  this.ResetFrobber(originalFrobber);
}
```

If GetCurrentFrobber() throws an exception, then originalFrobber is still null when the finally block is executed; if GetCurrentFrobber cannot throw, then why is it inside a try block?

Handle exceptions judiciously
Only catch specific exceptions that you are prepared to handle, and only for the specific section of code where you expect it to arise. Avoid the temptation to handle all exceptions or instances of the root class Exception unless your intention is simply to log and re-throw the exception. Certain exceptions may leave the application in a state where it is better to crash without further damage than to futilely try to recover and inflict damage. Your attempts to recover may inadvertently make matters worse.

There are some nuances around handling fatal exceptions, especially concerning how the execution of finally blocks can impact exception safety and the debugger. For more info, see: http://incrediblejourneysintotheknown.blogspot.com/2009/02/fatal-exceptions-and-why-vbnet-has.html

Use a top-level exception handler to safely handle unexpected situations and expose information to help debug the problem. Use catch blocks sparingly to address specific cases that can be safely handled, and leave the unexpected cases for the top-level handler.

If you do catch an exception, do something with it. Swallowing exceptions only makes problems harder to recognize and debug.

Wrapping exceptions in a custom exception is especially useful for code that exposes a public API. Exceptions are part of the visible interface of a method, which should be controlled along with parameters and return values. Methods that propagate many exceptions are extremely difficult to integrate into robust, maintainable solutions.

Throwing and rethrowing exceptions
When you wish to handle a caught exception at a higher level, maintaining the original exception state and stack can be a great debugging aid. Carefully balance debugging and security considerations.

Good options include simply rethrowing the exception:
```
throw;
```

or using the exception as the InnerException in a new throw:
```
throw new CustomException(…, ex);
```

Do not explicitly rethrow the caught exception like this:
```
throw e;
```

That will reset the exception state to the current line and impede debugging.

Some exceptions originate outside the context of your code. Rather than using a catch block you may need to add handlers for events like ThreadException or UnhandledException. For example, Windows Forms exceptions are raised in the context of the form handler thread.

Atomicity (data integrity)
Exceptions must not impact the integrity of your data model. You need to ensure that your object is in a consistent state—that any assumptions made by the class implementation will not be violated. Otherwise, by "recovering" you may only enable your code to get confused and cause further damage later.

Consider methods that modify several private fields in sequence. If an exception is thrown in the middle of this sequence of modifications, your object may not be in a valid state. Try working out the new values before actually updating the fields, so that you can safely update all of the fields in an exception-safe manner.

Assignment of values of certain types—including bool, 32 bit or smaller numeric types and references—to a variable is guaranteed to be atomic. No such guarantee is made for larger types such as double, long and decimal. Consider always using lock statements when modifying variables shared by multiple threads.

7. EVENTS

Events and delegates work together to provide a means for classes to notify users when something interesting happens. The value of an event is the delegate that should be invoked when the event occurs. Events are like fields of delegate type; they are automatically initialized to null when the object is created.

Events are like a field whose value is a "multicast" delegate. That is, a delegate that can invoke other delegates in turn. You can assign a delegate to an event; you can manipulate events via operators like += and -=.

Beware race conditions
If an event is shared between threads, it is possible that another thread will remove all subscribers after you check for null and before you invoke it—throwing a NullReferenceException.

The standard solution is to make a local copy of the event, to be used for the test and the invocation. You still need to be careful that any subscribers removed in the other thread will operate correctly when their delegate is unexpectedly invoked. You can also implement locking to sequence the operations in a way that avoids problems.

```
public event EventHandler SomethingHappened;
private void OnSomethingHappened()
{
  // The event is null until somebody hooks up to it
  // Create our own copy of the event to protect against
another thread removing our subscribers
  EventHandler handler = SomethingHappened;
  if (handler != null)
    handler(this,new EventArgs());
}
```

For more information about events and races see: http://blogs.msdn.com/b/ericlippert/archive/2009/04/29/events-and-races.aspx

Don't forget to unhook event handlers
Subscribing an event handler to an event source creates a reference from the source object to the receiver object of the handler, which can prevent garbage collection of the receiver.

Properly unhooking handlers ensures that you waste neither time calling delegates that no longer work, nor memory storing useless delegates and unreferenced objects.

8. ATTRIBUTES

Attributes provide a means for infusing the metadata for assemblies, classes, and properties with information about their properties. They are most often used to provide information to consumers of the code—like debuggers, testing frameworks, and applications—via refection. You can define attributes for your own use or use predefined attributes like those listed in the table.

ATTRIBUTE	USED WITH	PURPOSE
DebuggerDisplay	Debugger	Debugger display format
DebuggerBrowsable	Debugger	Controls how debugger shows the element (hidden, collapsed, and so on).
InternalsVisibleTo	Member access	Enables classes to expose internal members to specific other classes. With it, testing routines can access protected members and persistence layers can use special private methods.
DefaultValue	Properties	Specifies a default value for a property.

Be very careful with the DebuggerStepThrough attribute—it can make it very difficult to find bugs in methods to which it is applied, since you will not be able to single step or break on them!

9. DEBUGGING

Debugging is an essential part of any development effort. In addition to providing visibility into normally opaque aspects of the runtime environment, the debugger can intrude on the runtime environment and cause the application to behave differently than if

it were run without the debugger.

Getting visibility into the exception stack
To see the current frame's exception state, you can add the expression "$exception" to a Visual Studio Watch window. This variable contains the current exception state, similar to what you would see in a catch block except that you can see it in the debugger without actuallycatching the exception in the code.

Be careful with side effects in accessors
If your properties have side effects, consider whether you should use an attribute or debugger setting to prevent the debugger from automatically calling the getter. For example, your class may have a property like this:

```
private int remainingAccesses = 10;
private string meteredData;
public string MeteredData
{
  get
  {
    if (remainingAccesses-- > 0)
      return meteredData;
    return null;
  }
}
```

The first time you view this object in the debugger, remainingAccesses will show as having a value of 10 and MeteredData will be null. If you hover over remainingAccesses, however, you will see that its value is now 9. The debugger's display of the property value has changed the state of your object.

10. OPTIMIZATION

Plan early, measure constantly, optimize later
Set reasonable performance goals during design. During development, concentrate on correctness rather than micro-optimizations. Measure your performance frequently against your goals. Only if you have failed to meet your goals should you spend valuable time trying to optimize a program.

Always use the most appropriate tools to make empirical measurements of performance, under conditions that are both reproducible and as similar as possible to real-world conditions experienced by the user.

When you measure performance, be careful about what you are actually measuring. When measuring the time taken in a function, does your measurement include function call or looping construct overhead?

There are many myths about certain constructs being faster than others. Don't assume these are true; experiment and measure.

Sometimes inefficient-looking code can actually run faster than efficient-looking code due to CLR optimizations. For example, the CLR optimizes loops that cover an entire array, to avoid the implicit per-element range checking. Developers often compute the length before looping over an array:

```
int[] a_val = int[4000];

int len = a_val.Length;
for (int i = 0; i < len; i++)
   a_val[i] = i;
```

By putting the length in a variable, the CLR might not be able to recognize the pattern and will skip the optimization. The manual optimization has counterintuitively caused worse performance.

Building strings
If you are going to be doing a lot of string concatenation, use the System.Text.StringBuilder object to avoid building many temporary string objects.

Use batch operations with collections
If you need to create and fill a collection of known size, reserve the space when creating the collection to avoid performance and resource problems due to repeated reallocation. You can further improve performance with an AddRange method, like that in List<T>:

```
Persons.AddRange(listBox.Items);
```

11. RESOURCE MANAGEMENT

The garbage collector enables automatic cleanup of memory. Even so, all disposable resources must be disposed properly—especially those not managed by the garbage collector.

COMMON SOURCES OF RESOURCE MANAGEMENT PROBLEMS	
Memory fragmentation	Allocations will fail if there is not a large enough contiguous chunk of virtual address space.
Process limits	Processes often have access to a strict subset of the memory and resources available to the system.
Resource leaks	The garbage collector only manages memory. Other resources need to be managed correctly by the application.
Resource in limbo	Resources that rely on the garbage collector and finalizers will not become available as soon as they are no longer used. In fact, they may never become available.

Use try/finally blocks to ensure resources are properly released, or have your classes implement IDisposable and take advantage of the using statement which is cleaner and safer.

```
using (StreamReader reader=new StreamReader(file))
{
 //your code here
```

Avoid the garbage collector in production code
Instead of interfering with the garbage collector by calling GC.Collect(), focus on properly releasing or disposing resources.

When measuring performance, be careful to have the garbage collector run when you can properly account for its impact.

Avoid writing finalizers
Contrary to popular rumors, your class doesn't need a Finalizer just because it implements IDisposable! You can implement IDisposable to give your class the ability to call Dispose on any owned composite instances, but a finalizer should only be implemented on a class that directly owns unmanaged resources.

Finalizers are primarily useful for calling an interop API to dispose of a Win32 handle, and SafeHandle handles that more easily.

You cannot assume that your finalizer—which always runs on the finalizer thread—can safely interact with other objects. Those other objects might themselves be in the process of being finalized.

12. CONCURRENCY

Concurrency and multithreaded programming are complicated, difficult affairs. Before you add concurrency to your application, make sure you really understand what you are doing—there are a lot of subtleties!

Multithreaded applications are very difficult to reason about, and are susceptible to issues like race conditions and deadlocks that do not generally affect single-threaded applications. Given these risks, you should consider multi-threading as a last resort. If you must have multiple threads, try to minimize the need for synchronization by not sharing memory between threads. If you must synchronize threads, use the highest-level synchronization mechanism that you can. With highest level first, these mechanisms include:

- Async-await/Task Parallel Library/Lazy<T>
- Lock/monitor/AutoResetEvent
- Interlocked/Semaphore
- Volatile fields and explicit barriers

This card is far too small to explain the intricacies of concurrency in C#/.NET. If you want or need to develop an application that utilizes concurrency, please review a detailed document such as O'Reilly's "Concurrency in C# Cookbook".

Using volatile
Marking a field as "volatile" is an advanced feature that is frequently misunderstood even by experts. The C# compiler will ensure that accessing the field has acquire and release semantics; this is not the same as ensuring that all accesses to the field are under a lock. If you do not know what acquire and release semantics are and how they affect CPU-level optimizations then avoid using volatile fields. Instead, use higher-level tools such as the Task Parallel Library or the CancellationToken type.

Leverage thread-safe built-in methods
Standard library types often provide methods that facilitate thread-safe access to objects. For example, Dictionary.TryGetValue(). Using these methods generally make your code cleaner and you don't need to worry about data races like time-of-check-time-of-use scenarios.

Don't lock on "this", strings, or other common, public objects
When implementing classes that will be used in multithreaded contexts, be very careful with your use of locks. Locking on this, string literals, or other public objects prevents encapsulation of your locking state and can lead to deadlock. You need to prevent other code from locking on the same object your implementation uses; your safest bet is a private object member.

13. COMMON MISTAKES TO AVOID

Dereferencing null
Improper use of null is a common source of coding defects that can lead to program crashes and other unexpected behavior. If you try to access a null reference as if it were a valid reference to an object—for example, by accessing a property or method—the runtime will throw a NullReferenceException.

Static and dynamic analysis tools can help you identify potential NullReferenceExceptions before you release your code.

In C#, null references typically result from variables that have not yet referenced an object. Null is a valid value for nullable value types and reference types. For example, Nullable<int>, empty delegates, unsubscribed events, failed "as" conversions, and in numerous other situations.

Every null reference exception is a bug. Instead of catching the NullReferenceException, try to test objects for null before you use them. That also makes the code easier to read by minimizing try/catch blocks.

When reading data from a database table, be aware that missing values could be represented as DBNull objects rather than as null references. Don't expect them to behave like potential null references.

Use of binary numbers for decimal values
Float and double represent binary rationals, not decimal rationals, and necessarily use binary approximations when storing decimal values. From a decimal perspective, these binary approximations have inconsistent rounding and precision—sometimes leading to unexpected results from arithmetic operations. Because floating point arithmetic is often performed in hardware, hardware conditions can unpredictably exacerbate these differences.

Use Decimal when decimal precision really matters—like with financial calculations.

Modifying structs
One common error scenario is to forget that structs are value types—meaning that they are copied and passed by value. For example, say you have code like this:

```
struct P { public int x; public int y; }

void M()
{
  P p = whatever;
  ...
  p.x = something;
  ...
  N(p);
}
```

One day, the maintainer decides to refactor the code into this:

```
void M()
{
  P p = whatever;
  Helper(p);
  N(p);
}
void Helper(P p)
{
  ...
  p.x = something;
  ...
}
```

And now when N(p) is called in M(), p has the wrong value. Calling Helper(p) passes a copy of p, not a reference to p, so the mutation performed in Helper() is lost. Instead, Helper should return a copy of the modified p.

Unexpected arithmetic
The C# compiler protects you from arithmetic overflows of constants, but not necessarily computed values. Use "checked" and "unchecked" keywords to ensure you get the behavior you want with variables.

Neglecting to save return value
Unlike structs, classes are reference types and methods can modify the referenced object in place. However, not all object methods actually modify the referenced object; some return a new object. When developers call the latter, they need to remember to assign the return value to a variable in order to use the modified object. During code reviews, this type of problem often slips under the radar. Some objects, like string, are immutable so methods never modify the object. Even so, developers commonly forget.

For example, consider string.Replace():

```
string label = "My name is Aloysius";
label.Replace("Aloysius", "secret");
Console.Out.WriteLine(label);
```

The code prints "My name is Aloysius" because the Replace method doesn't modify the string.

Don't invalidate iterators/enumerators
Be careful not to modify a collection while iterating over it.

```
List<int> myItems = new List<int>{20,25,9,14,50};

foreach(int item in myItems)
{
   if (item < 10)
   {
      myItems.Remove(item);
      // iterator is now invalid!
      // you'll get an exception on the next iteration
   }
}
```

If you run this code you'll get an exception thrown as soon as it loops around for the next item in the collection.

The correct solution is to use a second list to hold the items you want to delete then iterate that list while you delete:

```
List<int> myItems = new List<int>{20,25,9,14,50};
List<int> toRemove = new List<int>();

foreach(int item in myItems)
{
  if (item < 10)
  {
     toRemove.Add(item);
  }
}

foreach(int item in toRemove)
{
   myItems.Remove(item);
}
```

RemoveAll like this:

```
myInts.RemoveAll(item => (item < 10));
```

Property name mistakes
When implementing properties, be careful that the property name is distinct from the data member used in the class. It is easy to accidentally use the same name and trigger infinite recursion when the property is accessed.

```
// The following code will trigger infinite recursion
private string name;
public string Name
{
   get
   {
      return Name;  // should reference "name" instead.
   }
}
```

Beware when renaming indirect properties too. For example, data binding in WPF specifies the property name as a string. By carelessly changing that property name, you can inadvertently create problems that the compiler cannot protect against.

ABOUT THE AUTHORS

Jon Jarboe has been improving software development tools for over 20 years, in contexts ranging from embedded systems to enterprise applications. He is a passionate advocate for the quality, security, and productivity benefits of disciplined software testing throughout the SDLC. He greatly enjoys his role at Coverity, which provides the opportunity to understand the challenges faced by development teams, to influence the development of tools to address those challenges, and to help others understand how disciplined testing can improve development velocity and agility while simultaneously improving quality, security, and efficiency.

Eric Lippert is an expert in C# and architect at Coverity where he works on the C# analysis team. Prior to joining Coverity in January, 2013, Eric spent 16 years at Microsoft and was most recently a principal developer on the C# compiler team focused on Roslyn and a member of the C# language design team.

RECOMMENDED BOOK

Essential C# 5.0 is a well-organized, no-fluff guide to the latest versions of C# for programmers at all levels of C# experience. Fully updated to reflect new features and programming patterns introduced with C# 5.0 and .NET 4.5, this guide shows you how to write C# code that is simple, powerful, robust, secure, and maintainable. Microsoft MVP Mark Michaelis and C# principal developer Eric Lippert provide comprehensive coverage of the entire language, offering a complete foundation for effective software development.

BUY NOW

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

Version 1.0 $7.95