Difference between "throw" and "throw ex" in .NET

Exception handling seems to be a common problem for .NET developers, particularly younger developers. We pretty much all know that you should wrap operations that have the potential for failing in a try/catch block if you are interested in being able to do something about the error that occurred. I'm not going to talk about the rules and guidelines for using exception handling. Instead I'm going to focus on a particular aspect of exception handling, which I tend to call exception bubbling.

Exception bubbling means that even though you are catching the exception and doing something with it, you want that exception to "bubble" up from your code to the calling code so it has a chance to do something with that exception. This is a fairly common scenario, but it has the potential to cause some major problems when you are debugging.

I'm sure most of the exception bubbling code you've seen looks similar to this

```
try
{
    // do some operation that can fail
}
catch (Exception ex)
{
    // do some local cleanup
    throw ex;
}
```

This code looks perfectly reasonable and does the job. It properly catches the exception, does some local cleanup and then bubbles the exception up the chain. (A side note here is that you really shouldn't catch a general exception like this. I'm doing this for simplicity in the examples, but you should be catching specific exceptions and only those that you can do something about.)

However, how many of you have seen code that looks like this

```
try
{
    // do some operation that can fail
}
catch (Exception ex)
{
    // do some local cleanup
    throw;
}
```

There is a subtle difference between these two calls that won't be apparent until you are trying to debug the problem. That difference is in the stack trace information that gets sent with the exception.

In the first case, the stack trace is truncated below the method that failed. What this means is that when you look at the stack trace, it will look as if the exception originated in your code. This isn't always the case, particularly if you are bubbling up a CLR generated exception (like a SqlException). This is a problem known as "breaking the stack", because you no longer have the full stack trace information. This happens because you are in essence creating a new exception to throw.

By using "throw" by itself, you preserve the stack trace information. You can confirm this by looking at the IL generated for these two code blocks. This makes the difference very obvious since in the first example the IL instruction called is "throw" while in the second the instruction is called "rethrow".

Before you run and change all of your code, there are still places where "throw ex" is appropriate. There are times when you want to add information to the exception that was caught or change it into a more meaningful exception. In these instances you actually want a new exception to be thrown. Again, there are two ways you can do this. The most common way that I have seen is

```
try
{
    // do some operation that can fail
}
catch (Exception ex)
{
    // do some local cleanup
    throw new ApplicationException("operation failed!");
}
```

However, this still suffers the problem of breaking the stack. Here you are generating a completely new exception and loosing any of the stack trace information from the original exception. What you really want to do is

```
try
{
    // do some operation that can fail
}
catch (Exception ex)
{
    // do some local cleanup
    throw new ApplicationException("operation failed!", ex);
}
```

By passing the original exception to the ApplicationException you are preserving the original exception, and it's stack trace information, as the inner exception to your ApplicationException.

To wrap everything up

1. Only catch exceptions if they are important to you and you need to do some sort of cleanup as a result.
2. If you need to bubble an exception up the chain, use "throw" by itself.
3. If you need to add information to the exception or repackage it, always pass the original exception as the inner exception.