API design principle: Reading a property or adding an event handler should not alter observable behavior

devblogs.microsoft.com/oldnewthing/20231016-00

October 16, 2023

Raymond Chen

On of the Windows Runtime API design principles is that reading a property[1] or adding an empty event handler should not affect the API's proper usage. It is legal for the implementation to optimize based on whether a property was accessed or whether a handler is registered,[2] but the optimization should not affect overall correctness.

Here are examples of bad behavior we want to avoid:

> If you read the `Widget.Stream` property, you must call the `Close` method on the returned stream.

> If you add a handler for the `FancyReady` event, then the `PlainReady` event is not raised.

> The `MischiefDetected` event handler must call `MischiefManaged` before returning.

The reason for the "reading a property should not affect proper usage" guideline is that many debuggers will "helpfully" dump the properties of an object. In the case of the `Stream` property above, if reading the `Stream` creates an obligation to `Close` it, then each time you hover over a widget or log it to the console, the debugger will read the `Stream` property and show it on the screen. The debugger doesn't know the special rule about having to `Close` the stream, so the stream will go unclosed, and you have a memory leak.

Even worse, that stream may be associated with an open file handle, so now you leaked a file handle, and the effects of a leaked file handle can be quite severe. Debugging is hard enough. Don't create a situation where *a bug is introduced by the presence of a debugger*. "Yeah, we can't run this program under the debugger to figure out what is going wrong, because once we run it under the debugger, it crashes with a sharing violation."

It is also common, especially when learning how to use a new feature, to add handlers to every event, where all the handler does is log a message like "FancyReady received" followed by the values of all of the event arguments. This lets developers see the event flow and gain a better mental model of how the feature works. But if adding an event handler changes the feature's proper behavior, you create a version of the Heisenberg Uncertainty Principle: Attempting to observe the system changes its behavior.

And you definitely don't want to put people into a position where they throw up their hands in frustration and say, "I don't understand. Once we connect a debugger or turn on logging, the program crashes even before we get to the problem we're trying to solve. This problem is un-debuggable."

[1] You are allowed to raise an exception from a property access if the situation calls for it.

[2] You are allowed to require that a handler be registered for an event. That doesn't violate the principle, because you're saying that omitting the handler is was never proper API usage to begin with. (In C/C++ terms, it is "undefined behavior".) It does mean that if the developer adds a dummy handler that just logs information but does no work, they might inadvertently "fix" their program. In the case of improper usage, you should pass a custom message to `RoOriginateError` to remind the developer why the operation failed. "You must register a MuffinReady handler before you can Bake()."