Going beyond the empty set: Embracing the power of other empty things

devblogs.microsoft.com/oldnewthing/20240923-00

September 23, 2024

The empty set contains nothing. This sounds really silly, but it's actually really nice.

The Windows Runtime has a policy that if a method returns a collection (such as an `IVector`), and the method produces no results, then it should return an empty collection, rather than a null reference. That way, consumers can just iterate over the collection without having to deal with a null test.

For example, suppose you have a method `Widget::GetAssociatedDoodads` which returns an `IVectorView<Doodad>` representing the `Doodad` objects that have been associated with a `Widget` object. If no `Doodad`s have been associated with the `Widget`, then it should return an empty vector, not a null pointer. That allows developers to write the natural-looking code:

```
// C#
foreach (var doodad in widget.GetAssociatedDoodads()) {
    ⟦ process each doodad ⟧
}

// C++/WinRT
for (auto&& doodad : widget.GetAssociatedDoodads()) {
    ⟦ process each doodad ⟧
}

// JavaScript
widget.GetAssociatedDoodads().forEach(doodad =>
{
    ⟦ process each doodad ⟧
});
```

rather than having to insert a null test (which is easily forgotten):

```
// C#
var doodads = widget.GetAssociatedDoodads();
if (doodads != null) { // annoying null test
    foreach (var doodad in widget.GetAssociatedDoodads()) {
        ⟦ process each doodad ⟧
    }
}

// C++/WinRT
auto doodads = widget.GetAssociatedDoodads();
if (doodads) { // annoying null test
    for (auto&& doodad : doodads) {
        ⟦ process each doodad ⟧
    }
}

// JavaScript
var doodads = widget.GetAssociatedDoodads();
if (doodads) { // annoying null test
    doodads.forEach(doodad =>
    {
        ⟦ process each doodad ⟧
    });
}
```

The principle of the empty collection applies to other types of collections, like `IMap<K, V>`, `array`. You can think of strings as collections of characters, and you can think of memory buffers (such as `IBuffer`) as collections of bytes.

An example of a poor design is the `CryptographicBuffer` class. (Sorry, `CryptographicBuffer`, for throwing you under the bus.)

<table>
<tr><th>Method</th><th>Expected Result</th><th>Actual Result</th></tr>
<tr><td>buffer = ConvertStringToBinary("");</td><td rowspan="5">buffer != null
buffer.Length == 0</td><td rowspan="4">buffer == null
buffer.Length /* crashes */</td></tr>
<tr><td>buffer = CreateFromByteArray(new[] {});</td></tr>
<tr><td>buffer = DecodeFromBase64String("");</td></tr>
<tr><td>buffer = DecodeFromHexString("");</td></tr>
<tr><td>buffer = GenerateRandom(0);</td><td>buffer != null
buffer.Length == 0</td></tr>
</table>

If the `ConvertStringToBinary`, `CreateFromByteArray`, `DecodeFromBase64String`, `DecodeFromHexString` are given empty strings or arrays, you expect them to produce an empty buffer, but instead they return *no buffer at all*.

This means that code like this looks correct:

```
// Write the string to a file as UTF-8
var buffer = CryptographicBuffer.ConvertStringToBinary(
        BinaryStringEncoding.Utf8, message);
await FileIO.WriteBufferAsync(storageFile, buffer);
```

but then you discover (probably at a very inconvenient moment) that it crashes if the message is an empty string, because `ConvertStringToBinary` returned `null` (instead of a non-null reference to an empty buffer), and then `WriteBufferAsync` threw an invalid parameter exception because the buffer cannot be null.

On the other hand, if you ask `GenerateRandom` to generate zero random bytes, it correctly gives you an empty buffer, rather than a null pointer. So at least one of the methods in the `CryptographicBuffer` class understands how empty collections work.

As a bonus insult, the `CryptographicBuffer.Compare` method requires that both buffers be non-null, so you can't even do this:

```
// Do it twice and confirm the results are the same
var buffer1 = CryptographicBuffer.ConvertStringToBinary(
        BinaryStringEncoding.Utf8, message);
var buffer2 = CryptographicBuffer.ConvertStringToBinary(
        BinaryStringEncoding.Utf8, message);
if (CryptographicBuffer.Compare(buffer1, buffer2)) {
    // the buffers are equal
}
```

The code crashes if the message is an empty string because `buffer1` and `buffer2` will be `null`, which is not a valid parameter to `CryptographicBuffer.Compare`. It's a bit ironic that the `CryptographicBuffer` can dish out null buffers but can't take them.

Cryptography in general seems to have a hard time with the concept of zero. The `UserDataProtectionManager.ProtectBufferAsync` method, for example, rejects attempts to protect an empty buffer, so if you want to protect a buffer that might be empty, you need to special-case the empty buffer.

```
// This version crashes if the buffer is empty.
static class Protector
{
    static UserDataProtectionManager manager =
        UserDataProtectionManager.TryGetDefault();

    public Task<IBuffer> ProtectBufferAsync(IBuffer buffer)
    {
        if (manager != null) {
            return await manager.ProtectBufferAsync(buffer,
                    UserDataAvailability.AfterFirstUnlock);
        } else {
            // No protection available - leave unprotected.
            return buffer;
        }
    }

    public Task<IBuffer> UnprotectBufferAsync(IBuffer buffer)
    {
        if (manager != null) {
            return await manager.UnProtectBufferAsync(buffer);
        } else {
            // No protection available - it was left unprotected.
            return buffer;
        }
    }
}
```

A naïve way of fixing this is to detect an empty buffer and just skip the `ProtectBufferAsync` call, letting an empty buffer be its own protected buffer. This is a bad idea, however, because a bad guy who sees an empty protected buffer will know that this represents an empty unprotected buffer. If the buffer represents a password, then they will know that the password is blank!

If you choose some sentinel non-empty buffer value to represent a non-empty buffer, you then have to have some way of distinguishing this from a genuine non-empty buffer that happens to match your sentinel. In mathematical terms, your function that converts buffers to non-empty buffers needs to be injective. One way is to append a dummy byte to the buffer, and remove the dummy byte when unprotecting.

```
// C#

// Work around inability to protect empty buffers
// by appending a dummy byte to all buffers.
var paddedBuffer = WindowsRuntimeBuffer.Create(buffer.Length + 1);
paddedBuffer.Length = actualBuffer.Capacity;
buffer.CopyTo(paddedBuffer);
var protectedBuffer = await manager.ProtectBufferAsync(
    paddedBuffer, UserDataAvailability.AfterFirstUnlock);

// Reverse the workaround by removing the dummy byte
// after unprotecting.
var result = await manager.UnprotectBufferAsync(protectedBuffer);
if (result.Status == UserDataBufferUnprotectStatus.Succeeded)
{
    var trimmedBuffer = result.UnprotectedBuffer;
    trimmedBuffer.Length = trimmedBuffer.Length - 1;
    ⟦ do something with the trimmed buffer ⟧
}

// C++

// Work around inability to protect empty buffers
// by appending a dummy byte to all buffers.
auto length = buffer.Length();
auto paddedBuffer = winrt::Buffer(length + 1);
paddedBuffer.Length(length + 1);
memcpy_s(paddedBuffer.data(), length, buffer.data(), length);
auto protectedBuffer = co_await manager.ProtectBufferAsync(
    paddedBuffer, winrt::UserDataAvailability::.AfterFirstUnlock);

// Reverse the workaround by removing the dummy byte
// after unprotecting.
auto result = co_await manager.UnprotectBufferAsync(protectedBuffer);
if (result.Status() == winrt::UserDataBufferUnprotectStatus::Succeeded) {
    auto trimmedBuffer = result.UnprotectedBuffer();
    trimmedBuffer.Length(trimmedBuffer.Length() - 1);
    ⟦ do something with the trimmed buffer ⟧
}
```

The inability to handle zero-byte buffers makes everybody’s life harder.

Zero. It’s a valid number. Please support it.