Embracing the power of the empty set in API design (and applying this principle to selectors and filters)

devblogs.microsoft.com/oldnewthing/20240812-00

August 12, 2024

Raymond Chen

In mathematics, the empty set is a set with no members. This sounds totally useless, but it's actually quite powerful.

Suppose you have a Widget object and a method `GetChildren()` that returns the child widgets. What should you do if the Widget has no children?

Sometimes, teams try to be clever and say, "Oh, if there are no children, then I will just return a null pointer instead of a list of child elements." For example, `IShellFolder::EnumObjects` has the rule that if it returns `S_FALSE`, then it is allowed to return a null pointer instead of an enumerator.

As a result, this code is subtly wrong:

```
ComPtr<IEnumIDList> e;
if (SUCCEEDED(shellFolder->EnumObjects(hwnd, SHCONTF_FOLDERS, &e)) {
    for (CComHeapPtr<IDLIST_RELATIVE> child;
         e->Next(1, &child, nullptr) == S_OK;
         child.Free()) {
        // process the child folder
    }
}
```

The above code is at risk of crashing if there are no child folders: The `IShellFolder` implementation may chose to use the special case carve-out and return `S_FALSE` from the `EnumObjects` method, and set `e = nullptr`. If that happens, then the enumeration loop crashes on a null pointer.

The correct code would have to check for this corner case and skip the enumeration loop. One way is to check for `S_FALSE` explicitly:

```
ComPtr<IEnumIDList> e;
HRESULT hr = shellFolder->EnumObjects(hwnd, SHCONTF_FOLDERS, &e);
if (SUCCEEDED(hr) && (hr != S_FALSE)) {
    for (CComHeapPtr<IDLIST_RELATIVE> child;
         e->Next(1, &child, nullptr) == S_OK;
         child.Free()) {
        // process the child folder
    }
}
```

Another is to check the enumerator object for null.

```
ComPtr<IEnumIDList> e;
if (SUCCEEDED(shellFolder->EnumObjects(hwnd, SHCONTF_FOLDERS, &e)) && e) {
    for (CComHeapPtr<IDLIST_RELATIVE> child;
         e->Next(1, &child, nullptr) == S_OK;
         child.Free()) {
        // process the child folder
    }
}
```

Another is to check the return value for `S_OK` exactly:

```
ComPtr<IEnumIDList> e;
if (shellFolder->EnumObjects(hwnd, SHCONTF_FOLDERS, &e) == S_OK) {
    for (CComHeapPtr<IDLIST_RELATIVE> child;
         e->Next(1, &child, nullptr) == S_OK;
         child.Free()) {
        // process the child folder
    }
}
```

Regardless of how you deal with the issue, it doesn't hide the fact that this special case is an extra bit of hassle, and more importantly, it's the sort of hassle that is *easily overlooked* when writing code.

The obvious-looking code is wrong. The weird-looking code is correct. This is the opposite of what we want: We want the natural code to be the correct one.

The general rule for methods that return collections is therefore that if there are no items to return, then the methods should return empty collections, rather than null references. The empty set is a perfectly valid set, and existing algorithms operate properly on an empty set. For example, you iterate over an empty collection the same way as a nonempty collection.

The empty set also follows mathematical rules when applied to union and intersection. If you have collection of sets that happens to be empty, then mathematically, the union of the elements of the empty set is another empty set. And the intersection of the elements of the empty set is the entire universe.

When applied to functions, this means that if a method takes a set of criteria with the requirement that elements must match at least one of the criteria (a "selector" pattern), then an empty set selects no elements. On the other hand, if the requirement is that elements must match *all* of the criteria (a "filter" pattern), then an empty set selects all elements.[1]

For example, suppose we have a method that finds all widgets of the specified colors:

```
IVectorView<Widget>
    FindWidgetsOfColors(IIterable<Color> colors);
```

What should happen if you pass an empty list of colors?

Well, since you didn't specify any colors, then are no matching widgets, so mathematically speaking, this should return an empty vector. You might also choose to declare that choosing widgets by color must provide at least one color, so calling `FindWidgetsOfColors` with an empty set of colors is allowed to fail with `E_INVALIDARG`. But if you're going to return something, you had better return the empty set. Do not design the function so that passing an empty list of colors returns all the widgets.

Conversely, suppose you have a method that finds widgets subject to filter criteria:

```
enum WidgetFilter
{
    Active,
    Connected,
};

IVectorView<Widget>
    FindWidgetsWithFilter(IIterable<WidgetFilter> filters);
```

What should happen if you pass an empty list of filters?

Mathematically, an empty list of filters is an unfiltered query, and you should return all the widgets. Again, you might decide that a filtered query must provide at least one filter, but if you choose to allow an empty list of filters, it had better return all the widgets.

These powers of the empty set mean that many operations have their natural meanings. For example, if you have a filter set and decide that you want to allow inactive widgets, you can remove `WidgetFilter.Active` from your filters. If `FindWidgetsWithFilter` treated an empty filter list as meaning "Return no widgets at all", then you'd need a special case for the possibility that the `Active` filter was the last filter.

Bonus chatter: Most programming languages understand the power of the empty set. For example, C++ `std::all_of` and JavaScript `Array.prototype.every` return `true` when given an empty collection, whereas C++ `std::any_of` and JavaScript `Array.prototype.some` return `false` when given an empty collection.

[1] This naming distinction between "selector" patterns (where each selector adds to the list of results) and "filter" patterns (where each filter shrinks the list of results) is my own and is not officially part of the Windows API design patterns.

My proposal that was accepted is that filters should be named after what they allow to pass through, not by what they remove. For example, passing the `WidgetFilter.Active` filter means that you want the active widgets. It doesn't mean that you want to filter out the active widgets (and return only the inactive ones). If you want a filter to remove active widgets, then the filter should be named something like `WidgetFilter.Inactive`. For adjectives that don't have an obvious antonym, you can use the prefix `Not`, as in `WidgetFilter.NotBlinking`.

Bonus chatter: We saw another application of this principle some time ago when we looked at what a timeout of zero should mean.