# On the proper care and feeding of the default_overload Windows Runtime attribute

devblogs.microsoft.com/oldnewthing/20210528-00

May 28, 2021

Raymond Chen

The Windows Runtime provides a language-independent way of expressing a programming interface. There are many parts to this, but today we'll look at method overloading. (Documentation on overloading Windows Runtime methods.)

Windows Runtime methods may be freely overloaded by arity: That is, you can define multiple methods with the same name, as long as each one takes a different number of arguments. For usability purposes, it is recommended that you add new parameters to the end, and that the behavior of the shorter-parameter-list function be equivalent to calling the longer-parameter-list function with (scenario-specific) natural defaults for the missing parameters.

```
runtimeclass Widget
{
  void Start();
  void Start(StartMode mode);
  void Start(StartMode mode, Widget parent);
}
```

In this example, the `Start` can be called with two parameters to specify a start mode and a parent widget. If you omit the parent widget, then it defaults to null. And if you omit the mode, then it starts in some scenario-specific default mode.

Here's another example:

```
runtimeclass DeviceInformation
{
  static DeviceWatcher CreateWatcher();
  static DeviceWatcher CreateWatcher(DeviceClass deviceClass);
  static DeviceWatcher CreateWatcher(String aqsFilter);
  static DeviceWatcher CreateWatcher(String aqsFilter,
                                     IIterable<String> additionalProperties);
  static DeviceWatcher CreateWatcher(String aqsFilter,
                                     IIterable<String> additionalProperties,
                                     DeviceInformationKind kind);
}
```

There are five overloads, and aside from the mysterious `DeviceClass` overload, they follow the recommended pattern of having each new overload be an extension of the previous overload.

If you decide to have multiple overloads of a method with the same number of parameters, you get a compiler error:

```
The 1-parameter overloads of DeviceInformation.CreateWatcher must have exactly one
method specified as the default overload by decorating it with Windows.Foundation.
Metadata.DefaultOverloadAttribute.
```

What is going on here?

Some programming languages are dynamically-typed. JavaScript and Python are two examples. For these languages, the selection of the overload considers only the number of parameters and not their types. This means that a JavaScript call to

```
DeviceInformation.createWatcher(v);
```

is ambiguous: Should `v` be coerced to a `DeviceClass` or to a `String`?

To resolve the ambiguity, you are required to apply the `[default_overload]` attribute to one of the methods. But how do you choose which one?

If you mark one overload as the default, then all other overloads with the same arity become unavailable to dynamically-typed languages. Therefore, the default overload should be chosen so that the functionality of the lost overloads is still available by other means, typically by using one of the other still-present overloads.

In our example, the functionality of the `String` overload can be obtained by calling the `String, IIterable<String>` overload and passing an empty list of properties. On the other hand, the `DeviceClass` overload is the only way to create a `DeviceWatcher` that is filtered to a `DeviceClass`.

This makes the `DeviceClass` method the clear winner. Indicate that by applying the `default_overload` attribute:

```
runtimeclass DeviceInformation
{
  static DeviceWatcher CreateWatcher();
  [default_overload]
  static DeviceWatcher CreateWatcher(DeviceClass deviceClass);
  static DeviceWatcher CreateWatcher(String aqsFilter);
  static DeviceWatcher CreateWatcher(String aqsFilter,
                                     IIterable<String> additionalProperties);
  static DeviceWatcher CreateWatcher(String aqsFilter,
                                     IIterable<String> additionalProperties,
                                     DeviceInformationKind kind);
}
```

What if there is no winner because all of the conflicting overloads provide unique functionality not available from other overloads?

```
runtimeclass Widget
{
    static Widget Create(Uri source);
    static Widget Create(IInputStream source);
}
```

Our hypothetical `Widget` object can be created from a Uri, or created from an input stream. If we mark one as the default overload, then we completely lose access to the other one.

To solve this problem, give the two versions different names so that they aren't overloaded.

```
runtimeclass Widget
{
    static Widget CreateFromUri(Uri source);
    static Widget CreateFromStream(IInputStream source);
}
```

Now both creation patterns are available to all languages.

Raymond Chen

**Follow**