Extension API Guidelines

This is a loose collection of guidelines that you should be following when proposing API. The process for adding API is described here: https://github.com/Microsoft/vscode/wiki/Extension-API-process.

Breakage

We DO NOT want to break API. Therefore be careful and conservative when proposing new API. It needs to hold up in the long term. Expose only the minimum but still try to anticipate potential future requests.

Namespaces

The API is structured into different namespaces, like `commands`, `window`, `workspace` etc. Namespaces contain functions, constants, and events. All types (classes, enum, interfaces) are defined in the global, `vscode`, namespace.

JavaScript

The API should have a JavaScript'ish feel. While that is harder to put in rules, it means we use namespaces, properties, functions, and globals instead of object-factories and services. Also take inspiration from popular existing JS API, for instance `window.createStatusBarItem` is like `document.createElement`, the members of `DiagnosticsCollection` are similar to ES6 maps etc.

Global Events

Events aren't defined on the types they occur on but in the best matching namespace. For instance, document changes aren't sent by a document but via the `workspace.onDidChangeTextDocument` event. The event will contain the document in question. This **global event** pattern makes it easier to manage event subscriptions because changes happen less frequently.

Private Events

Private or instance events aren't accessible via globals but exist on objects, e.g., `FileSystemWatcher#onDidCreate`. *Don't* use private events unless the sender of the event is private. The rule of thumb is: 'Objects that can be accessed globally (editors, tasks, terminals, documents, etc)' should not have private events, objects that are private (only known by its creators, like tree views, web views) can send private events'

Event naming

Events follow the `on[Did|Will]VerbSubject` patterns, like `onDidChangeActiveEditor` or `onWillSaveTextDocument`. It doesn't hurt to use explicit names.

Creating Objects

Objects that live in the main thread but can be controlled/instantiated by extensions are declared as interfaces, e.g. `TextDocument` or `StatusBarItem`. When you allow creating such objects your API must follow the `createXYZ(args): XYZ` pattern. Because this is a constructor-replacement, the call must return synchronously.

Shy Objects

Objects the API hands out to extensions should not contain more than what the API defines. Don't expect everyone to read `vscode.d.ts` but also expect folks to use debugging-aided-intellisense, meaning whatever the debugger shows developers will program against. We don't want to appear as making false promises. Prefix your private members with `_` as that is a common rule or, even better, use function-scopes to hide information.

Sync vs. Async

Reading data, like an editor selection, a configuration value, etc. is synchronous. Setting a state that reflects on the main side is asynchronous. Despite updates being async your 'extension host object' should reflect the new state synchronously. This happens when setting an editor selection

```
editor.selection = newSelection

    |
    |
    V

1. On the API object set the value as given
2. Make an async-call to the main side ala `trySetSelection`
3. The async-call returns with the actual selection (it might have changed in the meantime)
4. On the API object set the value again
```

We usually don't expose the fact that setting state is asynchronous. We try to have API that feels sync -`editor.selection` is a getter/setter and not a method.

Data Driven

Whenever possible, you should define a data model and define provider-interfaces. This puts VS Code into control as we can decide when to ask those providers, how to deal with multiple providers etc. The `ReferenceProvider` interface is a good sample for this.

Enrich Data Incrementally

Sometimes it is expensive for a provider to compute parts of its data. For instance, creating a full `CompletionItem` (with all the documentation and symbols resolved) conflicts with being able to compute a large list of them quickly. In those cases, providers should return a lightweight version and offer a `resolve` method that allows extensions to enrich data. The `CodeLensProvider` and `CompletionItemProvider` interfaces are good samples for this.

Cancellation

Calls into a provider should always include a `CancellationToken` as the last parameter. With that, the main thread can signal to the provider that its result won't be needed anymore. When adding new parameters to provider-functions, it is OK to have the token not at the end anymore.

Objects vs. Interfaces

Objects that should be returned by a provider are usually represented by a class that extensions can instantiate, e.g. `CodeLens`. We do that to provide convenience constructors and to be able to populate default values.

Data that we accept in methods calls, i.e., parameter types, like in `registerRenameProvider` or `showQuickPick`, are declared as interfaces. That makes it easy to fulfill the API contract using class-instances or plain object literals.

Strict and Relaxed Data

Data the API returns is strict, e.g. `activeTextEditor` is an editor or `undefined`, but not `null`. On the other side, providers can return relaxed data. We usually accept 4 types: The actual type, like `Hover`, a `Thenable` of that type, `undefined` or `null`. With that we want to make it easy to implement a provider, e.g., if you can compute results synchronous you don't need to wrap things into a promise or if a certain condition isn't met simple return, etc.

Validate Data

Although providers can return 'relaxed' data, you need to verify it. The same is true for arguments etc. Throw validation errors when possible, drop data object when invalid.

Copy Data

Don't send the data that a provider returned over the wire. Often it contains more information than we need and often there are cyclic dependencies. Use the provider data to create objects that your protocol speaks.

Enums

When API-work started only numeric-enums were supported, today TypeScript supports string-or-types and string-enums. Because fewer concepts are better, we stick to numeric-enums.

Strict Null

We define the API with strictNull-checks in mind. That means we use the optional annotation `foo?: number` and `null` or `undefined` in type annotations. For instance, its `activeTextEditor: TextEditor | undefined`. Again, be strict for types we define and relaxed when accepting data.

Undefined is False

The default value of an optional, boolean property is `false`. This is for consistency with JS where undefined never evaluates to `true`

JSDOC

We add JSDoc for all parts of the API. The doc is supported by markdown syntax. When document string-datatypes that end up in the UI, use the phrase 'Human-readable string...'

Optional parameters (`?` vs `| undefined`)

- For implementation, treat omitting a parameter with `?` the same as explicitly passing in `undefined`
- Use `| undefined` when you want to callers to always have to consider the parameter.
- Use `?` when you want to allow callers to omit the parameter.
- Never use `?` and `| undefined` on a parameter. Instead follow the two rules above to decide which version to use .
- If adding a new parameter to an existing function, use `?` as this allows the new signature to be backwards compatible with the old version.
- Do not add an overload to add an optional parameter to the end of the function. Instead use `?`.

Optional properties (`?` vs `| undefined`)

- Do not write code that treats the absence of a property differently than a property being present but set to `undefined`
  - This can sometimes hit you on spreads or iterating through objects, so just something to be aware of

- For readonly properties on interfaces that VS Code exposes to extensions (this include managed objects, as well as the objects passed to events):
  - Use `| undefined` as this makes it clear the property exists but has the value `undefined`.
- For readonly properties on options bag type objects passed from extensions to VS Code:
  - Use `?` when it is ok to omit the property
  - Use `| undefined` when you want the user to have to pass in the property but `undefined` signals that you will fall back to some default
  - Try to avoid `? + | undefined` in most cases. Instead use `?`. Using both `? + | undefined` isn't wrong, but it's often more clear to treat omitting the property as falling back to the default rather than passing in `undefined`
- For unmanaged, writable objects:
  - If using `?`, always also add `| undefined` unless want to allow the property to be omitted during initialization, but never allow users to explicitly set it to `undefined` afterwards. I don't think we have many cases where this will be needed
    - In these cases, you may want to try changing the api to avoid this potential confusion
  - If adding a new property to an unmanaged object, use `?` as this ensures the type is backwards compatible with the old version