Built-in Modules

aka JavaScript Standard Library

Status update in prep for stage 2

https://github.com/tc39/proposal-built-in-modules

Champions: Michael Saboff, Mattijs Hoitink & Mark S. Miller

Acknowledgement

- I want to thank various people who have provided feedback and suggestions to work through current issues:

  Mattijs Hoitink, Keith Miller, Mark S. Miller, Jordan Harband, Shu-Yu Guo, Devin Rousso, Kris Kowal & Chip Morningstar

Agenda

- Refresher
- Two stage 2 blockers from June/July 2019 Plenaries:
  - Need the ability to synchronously import from traditional scripts.
  - The ability to shim / polyfill before the main app runs.
- Proposed design to resolve the blockers.

Built In Modules Conceptually

Host

ExecutionContext

```
import * from “js:Complex”;
  •
  •
  •
let i = new Complex(0, 1);
```

Internal Built In Module Map

ModuleSpecifier	*ModuleRecord*
js:Collections	BuiltInCollectionsRecord
js:BTree	BuiltInBTreeRecord
js:Complex	BuiltInComplexRecord
.	
.	
.	
.	
.	
.	

Two Blocking Issues

1. Need the ability to import built in modules from classic scripts before continuing execution (synchronous import).

2. Must be able to Shim / Polyfill Built In Modules similar to how global properties are shimmed.
   - Shim before the main application code loads.
   - Provide for a missing implementation.
   - Change or redact component functionality.
   - Shim multiple times in a deterministic ordered manner.
     e.g. shim B builds on shim A

Imports from Traditional Scripts

- We don’t want to orphan traditional scripts when adding built in modules.

- The `import` keyword is not available to scripts.
  - Modules are evaluated asynchronously (and the end of the event loop) while scripts are evaluated synchronously.
  - We currently have `import()` API, but it is async.
    More involved ergonomics that an `import` statement.
  - We propose adding a synchronous method to import built in modules.

Shimming

- Built in modules need to be “shimmable”.
- If required, shimming needs to happen before the “main app” code runs.
- Shims can be applied to prior shims.
- It is a high want that the shim setup code be able to lock down the resulting shimmed modules.

Built In Module Object

Add a new `BuiltInModule` object with the following methods:

- `hasModule(moduleSpecifier)`- Returns true / false based on whether there is a moduleSpecifier exists in the built in module map with `ModuleSpecifier` **key**.
- `getModule(moduleSpecifier)` - Returns the exports for the `ModuleSpecifier` **key** from the built in module map.
- `setModule(moduleSpecifier, exports)` - Adds or replaces the exports for the `ModuleSpecifier` **key** in the built in module map.
  - Entries can be added and updated until frozen.
- `freezeModules()` - Freezes the internal built in module map.

Built In Module Namespaces

- Namespaces separate modules by functionality.
  - Namespaces are denoted by a prefix, e.g. `js:`
  - There will likely be several namespaces.
  - Other standards bodies / organizations will likely want their functionality in a dedicated namespace, e.g. TC-53, OpenJS Foundation, …
  - Organizations can create and utilize a private namespace of their choosing for their own purposes.

Built In Module Namespaces

- TC-39 is responsible for the content of the `js:` namespace (core language).
  - Outside parties are free to propose additions to the `js:` namespace.
  - TC-39 uses a staging process to approve modules for the `js:` namespace. Could be the current one or a modification thereof.
- Other bodies can create their own namespaces.
  - Likely namespaces could be `web:`, `node:`, `device:` & `blockchain:`.

Questions?

Thank you!