# CIL.JS

A CIL (also known as MSIL) interpreter.

The goal of this project is to build a fully working implementation of ECMA-335 6th Edition running in JavaScript, that is also usable by other JavaScript. While JSIL compiles IL to JavaScript, the goal here is to take the raw bytes of a DLL or EXE and execute the IL in the browser (so no “compiling” really, hence an interpreter).

The project is to be in ES3 unless there is no other choice (see threading below). This is to allow the broadest usage cases.

## Memory Model

While ECMA-335 does not give a memory manager type generally a Garbage Collector is implemented. Because of limitations of JavaScript a garbage collector makes no sense. Instead Reference Counting is to be used.

All “heap” memory access is asynchronous. This is to allow threads to access memory which is to only actually exist on the main thread. Other threads will have to make asynchronous calls to the main thread requesting memory operations. This is flawed and would allow for easy deadlocking but is done because of the limitations of JavaScript’s threading model.

## Threads

JavaScript historically does not have threads and in fact ECMAScript does not have threads. Most modern browsers though have implemented an API called Workers which allows for a threading system to be used. This threading system is limited in that the only connection between threads is via the main thread and via asynchronous calls. This means that memory cannot easily be shared, see memory model for what we do instead. This also means that each thread must load the entire CIL.JS in memory and load its own copy of the assemblies into its memory space, so that we limit cross thread dependencies.

While the Worker API is not ES3 compatible since there is no other way to do threading in JavaScript it is allowed, BUT CIL.JS will be designed to work perfectly fine single threaded if Workers are not present.

## Interop with JavaScript

One all the assemblies have been loaded into CIL.JS is will create dummy wrapper objects that “normal” JavaScript can consume of the publicly visible objects. Because of the need to be ES3 compatible we will work like IL and these wrapper objects will only have methods and fields.

The methods will have “marshaling” logic to convert wrapper objects and JavaScript primitives into the appropriate IL memory object, allocating on the stack, in the heap, or getting a pointer to the heap in the case of a wrapper object. The methods will take as their first parameter a function that is called once the method finishes. If the method has a return value it is “marshaled” back to JavaScript primitives or a JavaScript primitive and then given as the first parameter to that function. If there is no return value then the function is just called with no parameter.