How to create a raw Transform stream(stream.Transform class) in node.js?

Read about why you should use stream in stream-handbook.

Streams were originally designed to make processing I/O in Node more manageable and efficient. Streams are essentially `EventEmitters` that can represent a readable and/or writable source of data. Just like a stream of liquid, the data flows to/from.

By default streams only support dealing with `Strings` and `Buffers`. However, you can make use of object streams in our own code by using the `objectMode` option.

When in `objectMode`, streams can push `Strings` and `Buffers` as well as any other JavaScript object. Another major difference is that when in **`objectMode`**, the internal buffering algorithm counts objects rather than bytes. This means if we have a Transform stream (**Node's core `stream.Transform` `class`**) with the `highWaterMark` option set to 5, the stream will only buffer a maximum of 5 objects internally.

Analysis of Sensor Data Problem

Within IoT devices there is large flood of data (from sensors, becons and wearable) generated which needs to be processed and analyzed. Wouldn't it be helpful to programmatically get statistics or locate patterns in raw data streams and provide a translation to further analysis. For many sensor data formats, in order to do that, we need to parse the data stream and transform it to simple representation for further analysis.

The beauty of Node streams is we don't have to do this all in memory (sensor data can be huge) and we can process data as soon its been read. Streams also will work from any I/O source (file system, network).

Using the new stream APIs, we can create a reusable I/O component that transforms our data into individual lines for further processing.

Lets assume for this assignment that sensor data from devices has only ASCII characters/strings and each string should be separated by ',' or '.'(ASCII characters) for distinction.[Refer to contents of input-sensor.txt on canvas]

In our problem, the input and output are actually the same data. However, this data is transformed into meaningful representation for further processing down the road (such as collecting statistics or searching).

"Transforms sit in the middle of a pipeline and are both readable and writeable:"

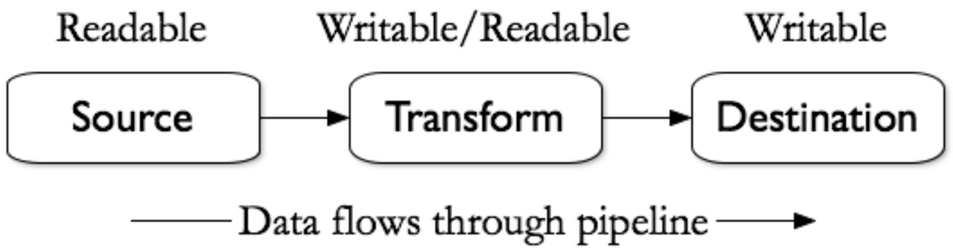

Figure1: Transform pipeline (Source: https://strongloop.com/strongblog/practical-examples-of-the-new-node-js-streams-api/)

Transform stream links the input to the output. The transform stream also provides a "_flush" method, which gives us an easy hook into the end of the input stream and feels much cleaner than having to bind to the "finish" event internally.

The class to extend and the method(s) to implement depend on the sort of stream class you are writing:

Use-case	Class	Method(s) to implement
Read only	Readable	_read
Writing only	Writable	_write, _writev
Reading and writing	Duplex	_read, _write, _writev
Operate on written data, then read the result	Transform	_transform, _flush

To set up our assignment3, we need to include the stream module and instantiate a new Transform stream in object mode and implement prototype _transform and _flush methods. What is object mode ? objectMode allows a consumer to get at each value that is pushed from the stream. Without objectMode, the stream defaults to pushing out chunks of data. As the name suggests, objectMode is not just for string values, like in our problem, but for any object in JavaScript ({ “my”: [“json”, “record”]}).

```
/*
A simple node.js that uses streams to transform the data and it teaches command line
application development.

References:


 + https://strongloop.com/strongblog/practical-examples-of-the-new-node-js-streams-api/

 + For prototyping and inheritence model
   refer to intermediate.js on canvas and http://book.mixu.net/node/ch6.html

 + https://nodesource.com/blog/understanding-object-streams/


 + commander.js
   - https://github.com/visionmedia/commander.js
   - http://tjholowaychuk.com/post/9103188408/commander-js-nodejs-command-line-
interfaces-made-easy

 */




var Transform = require('stream').Transform;


var util = require( "util" ).inherits;


…….//other modules such as fs, commander, underscore etc  are loaded

// For Node 0.8 users
if (!Transform) {
  Transform = require('readable-stream/transform');
}

//Constructor logic includes Internal state PatternMatch needs to consider because it has
```

```
to parse chunks that get transformed

function PatternMatch(…
…..

//Switching on object mode so when stream reads sensordata it emits single pattern match.
   Transform.call(
        this,
        {
            objectMode: true
        }
    );
………

}

// Extend the Transform class.
// --
// NOTE: This only extends the class methods - not the internal properties. As such we
// have to make sure to call the Transform constructor(above).


inherits(PatternMatch, Transform);





// Transform classes require that we implement a single method called _transform and
//optionally implement a method called _flush.  You assignment will implement both.

PatternMatch.prototype._transform = function (chunk, encoding, getNextChunk){


….

}

//After stream has been read and transformed, the _flush method is called. It is a great
//place to push values to output stream and clean up existing data

PatternMatch.prototype._flush = function (flushCompleted) {


.....




}

//That wraps up our little patternMatch module.
```

```
//Program module is for taking command line arguments
program
        .option('-p, --pattern <pattern>', 'Input Pattern such as . /n ,')
         .parse(process.argv);

// Create an input stream from the file system.
var inputStream = fileSystem.createReadStream( "input-sensor.txt" );
```

```
// Create a Regular Expression stream that will run through the input and find matches
// for the given pattern - "words".
var patternStream = inputStream.pipe( new PatternMatch(...));

// Read matches from the stream.

patternStream.on(….
```