Chapter 53: Binary Data

Section 53.1: Getting binary representation of an image file

This example is inspired by this question.

We'll assume you know how to load a file using the File API.

```
// preliminary code to handle getting local file and finally printing to console
// the results of our function ArrayBufferToBinary().
var file = // get handle to local file.
var reader = new FileReader();
reader.onload = function(event) {
    var data = event.target.result;
    console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //gets an ArrayBuffer of the file
```

Now we perform the actual conversion of the file data into 1's and 0's using a `DataView`:

```
function ArrayBufferToBinary(buffer) {
   // Convert an array buffer to a string bit-representation: 0 1 1 0 0 0...
   var dataView = new DataView(buffer);
   var response = "", offset = (8/8);
   for(var i = 0; i < dataView.byteLength; i += offset) {
       response += dataView.getInt8(i).toString(2);
   }
   return response;
}
```

`DataViews` let you read/write numeric data; `getInt8` converts the data from the byte position - here `0`, the value passed in - in the `ArrayBuffer` to signed 8-bit integer representation, and `toString(2)` converts the 8-bit integer to binary representation format (i.e. a string of 1's and 0's).

Files are saved as bytes. The 'magic' offset value is obtained by noting we are taking files stored as bytes i.e. as 8-bit integers and reading it in 8-bit integer representation. If we were trying to read our byte-saved (i.e. 8 bits) files to 32-bit integers, we would note that 32/8 = 4 is the number of byte spaces, which is our byte offset value.

For this task, `DataViews` are overkill. They are typically used in cases where endianness or heterogeneity of data are encountered (e.g. in reading PDF files, which have headers encoded in different bases and we would like to meaningfully extract that value). Because we just want a textual representation, we do not care about heterogeneity as there is never a need to

A much better - and shorter - solution can be found using an `UInt8Array` typed array, which treats the entire `ArrayBuffer` as composed of unsigned 8-bit integers:

```
function ArrayBufferToBinary(buffer) {
    var uint8 = new Uint8Array(buffer);
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Section 53.2: Converting between Blobs and ArrayBuffers

JavaScript has two primary ways to represent binary data in the browser. ArrayBuffers/TypedArrays contain mutable (though still fixed-length) binary data which you can directly manipulate. Blobs contain immutable binary

data which can only be accessed through the asynchronous File interface.

Convert a `Blob` to an `ArrayBuffer` (asynchronous)

```
var blob = new Blob(["\x01\x02\x03\x04"]),
    fileReader = new FileReader(),
    array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

Version ≥ 6

Convert a `Blob` to an `ArrayBuffer` using a `Promise` (asynchronous)

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
    console.log("Array contains", array.byteLength, "bytes.");
});
```

Convert an `ArrayBuffer` or typed array to a `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

Section 53.3: Manipulating ArrayBuffers with DataViews

DataViews provide methods to read and write individual values from an ArrayBuffer, instead of viewing the entire thing as an array of a single type. Here we set two bytes individually then interpret them together as a 16-bit unsigned integer, first big-endian then little-endian.

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);

view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);

console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true));  // 511
```

Section 53.4: Creating a TypedArray from a Base64 string

```
var data =
   'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACN' +
   'byblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHx' +
   'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==';
```

```
var characters = atob(data);

var array = new Uint8Array(characters.length);

for (var i = 0; i < characters.length; i++) {
  array[i] = characters.charCodeAt(i);
}
```

Section 53.5: Using TypedArrays

TypedArrays are a set of types providing different views into fixed-length mutable binary ArrayBuffers. For the most part, they act like Arrays that coerce all assigned values to a given numeric type. You can pass an ArrayBuffer instance to a TypedArray constructor to create a new view of its data.

```
var buffer = new ArrayBuffer(8);
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);

console.log(byteView);  // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]
```

ArrayBuffers can be copied using the `.slice(...)` method, either directly or through a TypedArray view.

```
var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView);  // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]
```

Section 53.6: Iterating through an arrayBuffer

For a convenient way to iterate through an arrayBuffer, you can create a simple iterator that implements the `DataView` methods under the hood:

```
var ArrayBufferCursor = function() {
  var ArrayBufferCursor = function(arrayBuffer) {
    this.dataview = new DataView(arrayBuffer, 0);
    this.size = arrayBuffer.byteLength;
    this.index = 0;
  }

  ArrayBufferCursor.prototype.next = function(type) {
    switch(type) {
      case 'Uint8':
        var result = this.dataview.getUint8(this.index);
        this.index += 1;
        return result;
      case 'Int16':
        var result = this.dataview.getInt16(this.index, true);
        this.index += 2;
        return result;
      case 'Uint16':
        var result = this.dataview.getUint16(this.index, true);
```

```
        this.index += 2;
        return result;
      case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
      case 'Uint32':
        var result = this.dataview.getUint32(this.index, true);
        this.index += 4;
        return result;
      case 'Float':
      case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
      case 'Double':
      case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
        this.index += 8;
        return result;
      default:
        throw new Error("Unknown datatype");
    }
  };

  ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
  }

  return ArrayBufferCursor;
});
```

You can then create an iterator like this:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

You can use the `hasNext` to check if there's still items

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

You can use the `next` method to take the next value:

```
var nextValue = cursor.next('Float');
```

With such an iterator, writing your own parser to process binary data becomes pretty easy.