CSC 470 – Section 3

Topics in Computer Science: Advanced Browser Technologies

Mark F. Russo, Ph.D.

Spring 2016

Lecture 11:

- Multithreading with Web Workers
- ArrayBuffers and Views
- ImageData

Web Workers

- Brings true multithreading to JavaScript
- Ideal for background tasks
- Cannot access the DOM (cannot modify the UI)
- Can access the XHR and other built-in JavaScript objects

- Applications
  - Cryptography
  - Sorting, searching large amounts of data
  - Parallel computation
  - Image Processing
  - …

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Web Workers

- Created as instances of a `Worker` object
- Constructor takes the name of a JavaScript file to load into the Worker

```
var worker = new Worker("upper.js");
```

- The Worker script may be initiated by sending it a message using its `postMessage(…)` method.
- Message data may consist of a wide range of JavaScript data types

```
worker.postMessage(msg);
```

- A `Worker` is notified of received messages as `message` events.
- The global object in a Web Worker can be accessed using the **`self`** keyword

Web Workers

- When a message is sent, a Worker's `self.onmessage` event handler function receives one argument of type `MessageEvent`
- `MessageEvent` objects have several properties…
- The `.data` property of a received `MessageEvent` object contains a `DOMString`, `Blob` or an `ArrayBuffer` containing the data sent
- `Workers` may send messages back to the originating script using its own `postMessage(…)` function invocation

```
// upper.js
// Convert received strings to upper case and return
self.onmessage = function(e) {
  var msg = e.data;
  var msgUpper = msg.toUpperCase();
  self.postMessage(msgUpper);
};
```

Note: `self` is not required here because it is the global object.

https://developer.mozilla.org/en-US/docs/Web/API/MessageEvent

11/upper.html

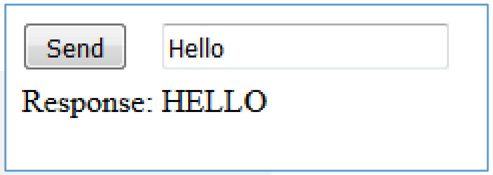
Web Workers

- The main script handles messages returned by a Worker in a manner similar to the way a Worker handles messages
- Workers have a `message` event that handles returned messages by assigning the `onmessage` property to a handler function in the main script
- In both the case of the main script and the Worker script…

`.postMessage(…)` method is used to send a message

`.onmessage` event listener is used to receive a message

Web Worker Example



Response: HELLO

Main Script (upper.html)

```javascript
// Post a message to the worker when the button is clicked
document.getElementById("send").onclick = function(e) {
  document.getElementById('out').value = "";
  var msg = document.getElementById("inp").value;
  console.log("Main script sending " + msg + " to worker");
  worker.postMessage(msg);
};

// Create the web worker object
var worker = new Worker("upper.js");

// Handle any message returned from worker
worker.onmessage = function(e) {
  document.getElementById('out').innerHTML = e.data;
}
```

Send message to Worker

Create Worker

Receive message from Worker

Worker Script (upper.js)

```javascript
// upper.js
// Convert received strings to upper case
self.onmessage = function(e) {
  console.log(self);
  var msg = e.data;
  var msgUpper = msg.toUpperCase();
  self.postMessage(msgUpper);
};
```

Receive message from Main

Send message to Main

11/upper.html

The Web Worker Global Object

- The `self` property references the specialized global scope for each Worker context.
    - A `WorkerGlobalScope` object

- Worker scope includes timers
    - `setInterval(), clearInterval()`
    - `setTimeout(), clearTimeout()`

- Worker scope has its own event queue
- Worker scope has access to the console
- Worker scope has access to the objects:
    - `XMLHttpRequest`, `ImageData`, `OffScreenCanvas`, `WebSocket`, `Worker`, …

Yes, a Worker and spawn new Workers.

https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope

Web Worker Methods

`.postMessage(msg)`

- sends a message to the main thread that spawned it. This accepts a single parameter, which is the data to send to the worker.

`.importScripts(…)`

- to import scripts into a worker scope
- may specify multiple scripts to import, separated by comma

`.close()`

- Discards any tasks queued in the `WorkerGlobalScope`'s event loop, effectively closing this particular scope

`.terminate()`

- Immediately terminates the worker. This does not offer the worker an opportunity to finish its operations; it is simply stopped at once

Web Worker Events

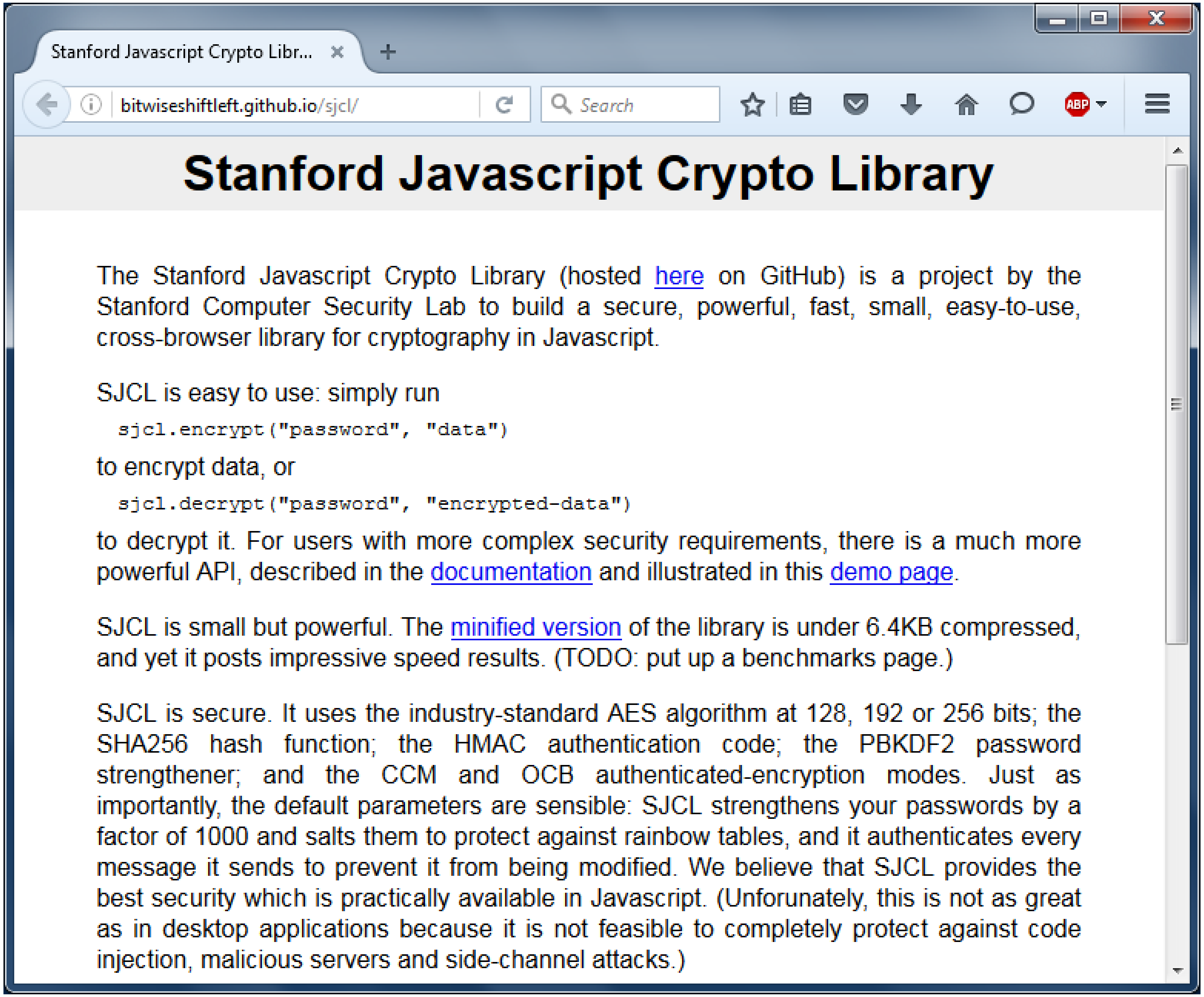
A Worker object may handle multiple events

- `message (.onmessage = …)`
  - Invoked when a Worker receives a message, which is wrapped in a `MessageEvent` object
- `error (.onerror = …)`
  - Called when the error occurs, usually when the script fails to load
- `online (.ononline = …)`
  - Fired when the Worker has gained access to the network
- `offline (.onoffline = …)`
  - Fired when the browser has lost access to the network

https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope

Cracking 4-Digit PINs with Web Workers

- Write a JavaScript program that cracks a 4-digit PIN using brute force
- Start with encrypted data and a decryption algorithm
- Cycle through all PINs (0000 through 9999) until the data are decrypted
- Use multiple Web Workers to speed up the process
- Use the Stanford JavaScript Crypto Library (SJCL) for encryption/decryption


Stanford Javascript Crypto Libr...
bitwiseshiftleft.github.io/sjcl/
Search

Stanford Javascript Crypto Library

The Stanford Javascript Crypto Library (hosted here on GitHub) is a project by the Stanford Computer Security Lab to build a secure, powerful, fast, small, easy-to-use, cross-browser library for cryptography in Javascript.

SJCL is easy to use: simply run

```
sjcl.encrypt("password", "data")
```

to encrypt data, or

```
sjcl.decrypt("password", "encrypted-data")
```

to decrypt it. For users with more complex security requirements, there is a much more powerful API, described in the documentation and illustrated in this demo page.

SJCL is small but powerful. The minified version of the library is under 6.4KB compressed, and yet it posts impressive speed results. (TODO: put up a benchmarks page.)

SJCL is secure. It uses the industry-standard AES algorithm at 128, 192 or 256 bits; the SHA256 hash function; the HMAC authentication code; the PBKDF2 password strengthener; and the CCM and OCB authenticated-encryption modes. Just as importantly, the default parameters are sensible: SJCL strengthens your passwords by a factor of 1000 and salts them to protect against rainbow tables, and it authenticates every message it sends to prevent it from being modified. We believe that SJCL provides the best security which is practically available in Javascript. (Unforunately, this is not as great as in desktop applications because it is not feasible to completely protect against code injection, malicious servers and side-channel attacks.)

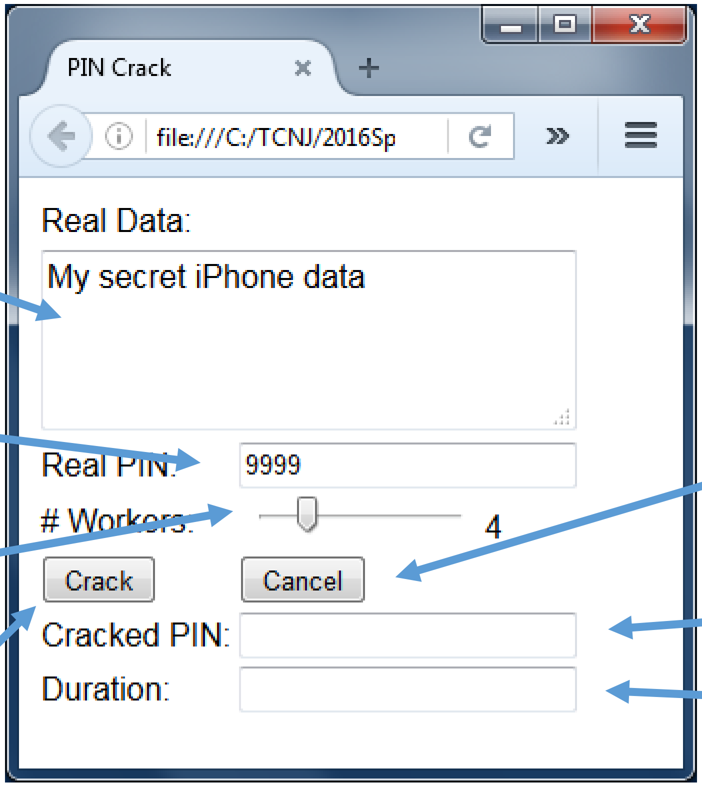
Example: PIN Cracker

Plan

1. Divide the space of all possible PINs into a specified number of equal ranges.
2. Create an array of Worker objects equal to the number of PIN ranges to be searched.
3. Load a script into each Worker that will cycle through all PINs in a range attempting to decrypt the data.
4. Post encrypted data and a different PIN range to each Worker
5. When decryption succeeds, post a message back to main script with found PIN and elapsed time. Terminate all Workers.
6. If no PIN successful, post a null PIN back to main script with elapsed time.

Example: PIN Cracker

User Interface


PIN Crack

file:///C:/TCNJ/2016Sp

Real Data:

My secret iPhone data

Real PIN: 9999

Workers: 4

Crack

Cancel

Cracked PIN:

Duration:

Example: PIN Cracker - 1

```
var el = document.getElementById.bind(document);

// Array of all Worker objects
var workers = [];

// Crack button event
el("bCrack").onclick = function (e) {
    // Clear output widgets and terminate all workers
    el('tPINout').value   = '';
    el('tDuration').value = '';
    workers.forEach(function (w) { w.terminate(); });
    workers = [];

    // Get input PIN
    var PINin = el('tPINin').value;
    if (PINin.length < 4) {
        alert("PIN must be four digits");
        return;
    }

    // Get PIN and encrypt data
    var data = el('tData').value;
    var encrypted = sjcl.encrypt(PINin, data);

    // Get number of workers to create
    var nworkers = Number(el('rNWorkers').value);
    setMsg("Running " + nworkers + " worker(s)");

    // …
```

Example: PIN Cracker - 2

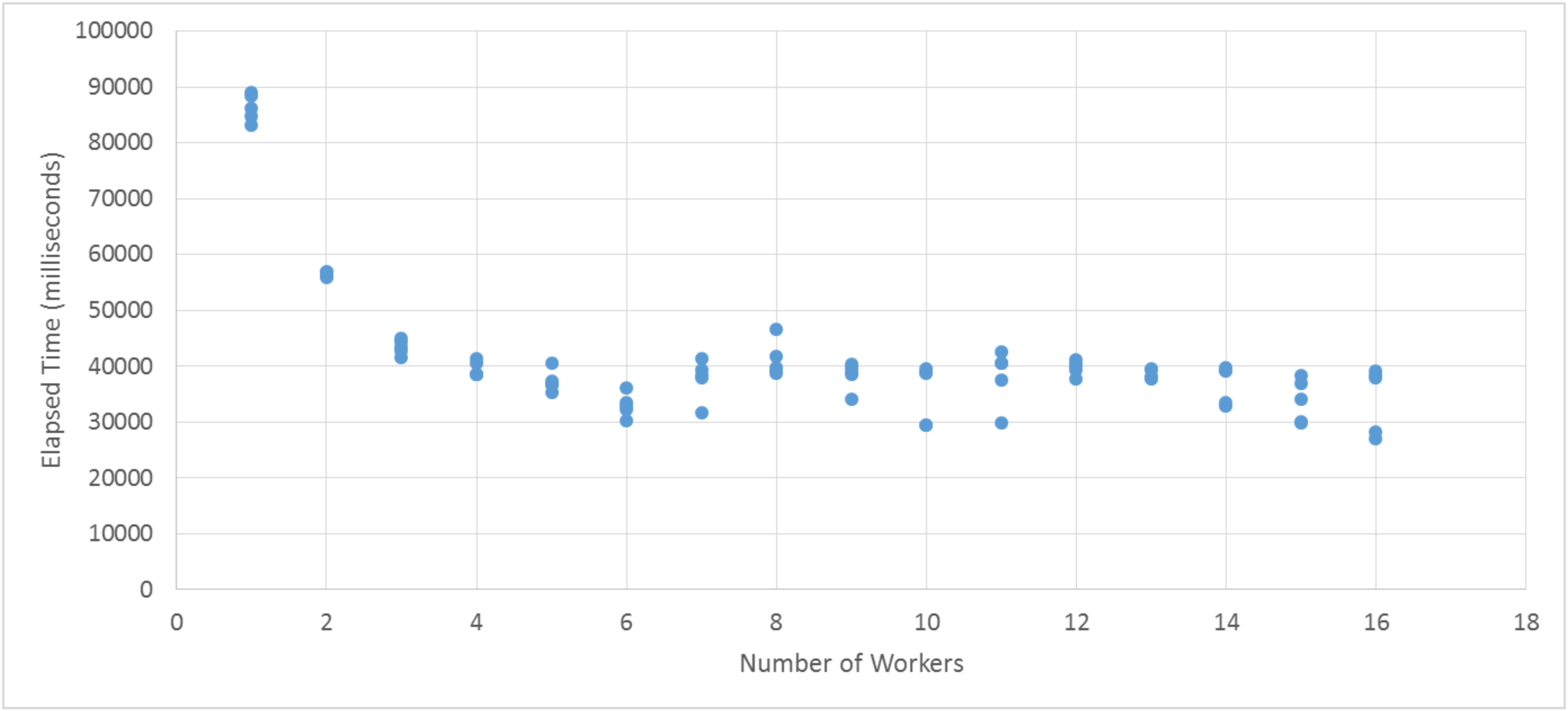
```
//…

// Set up all Workers
for (var i = 0; i < nworkers; i++) {
    // Create Worker object
    var w = new Worker('crack.js');

    // Set message event handler
    w.onmessage = function (e) {
        var PIN     = e.data.PIN;
        var elapsed = e.data.elapsed;
        var name    = e.data.name;

        // If the PIN was found, terminate all the remaining workers
        if (PIN !== null) {
            el('tPINout').value   = PIN;
            el('tDuration').value = elapsed;
            workers.forEach(function(w) { w.terminate(); });
        }

        setMsg('Worker ' + name + ' done');
    };
    // Collect in workers array
    workers.push(w);
}

//…
```

Example: PIN Cracker - 3

```
    //…

    // Pass out all PIN ranges to search
    var delta = Math.ceil(10000 / nworkers);
    var pin = 0;

    for (var i = 0; i < nworkers; i++) {
        // Start of next PIN range
        var first = pin;

        // Kick off each Worker
        var msg = { name      : ('w' + i),
                    encrypted: encrypted,
                    first     : first,
                    count     : delta };
        workers[i].postMessage(msg);

        // Next start PIN
        pin += delta;
    }
};
```

Example: PIN Cracker - 4

- Time to crack PIN vs. number of Workers



Web Workers – A Summary

- To create a Web Worker, use the Worker constructor and pass a script URL:

```
var worker1 = new Worker("work.js");
```

- A Worker's global object is stored in a variable named `self`
- To load additional scripts into the Worker context from within the Worker:

```
importScripts("utilities.js");
```

- To send a message to a Worker from the main script:

```
worker1.postMessage( msg );
```

- A wide variety of data types may be passed to a Worker, but no DOM elements
- To receive messages within a Worker, handle the "message" event:

```
self.onmessage = function(e) {
    var msg = e.data;
    // …
};
```

- To send a message from a Worker back to the parent script:

```
self.postMessage( result );
```

- To receive a message sent by a Worker in the parent script :

```
worker1.onmessage = function(e) {
    var result = e.data;
    // …
};
```

Data Structures

- JavaScript's main data structures are the Array and Object

  … but more advanced computation requires more flexibility

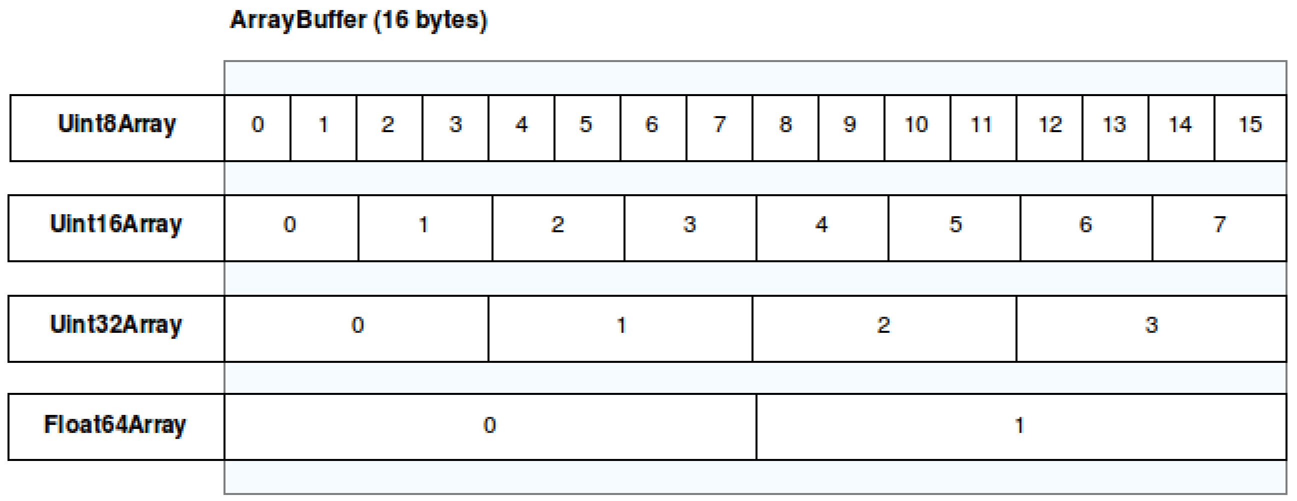
- JavaScript solves this problem with predefined array-like objects
  - provides a mechanism for accessing raw binary data

- JavaScript typed arrays split the implementation into <u>buffers</u> and <u>views</u>

- <u>ArrayBuffer</u> - an object representing a **chunk of data**; it has no format to speak of, and offers no mechanism for accessing its contents
- <u>View</u> – Various objects that turn the data into actual typed arrays

- Useful when dealing with images, audio, video, and other binary data

ArrayBuffer and its Views



ArrayBuffer

- Used to represent a generic, fixed-length binary data buffer
- Cannot be manipulated directly

Views

- Must wrap in a typed array view or a DataView object
- Views offer the buffer in a specific format
- View object methods are used to read and write contents

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

Int8Array Object

- Represents an array of **twos-complement 8-bit signed integers**
- Contents are initialized to 0
- Can access elements using object methods or bracket notation

Constructors

```
new Int8Array(length);
new Int8Array(typedArray);
new Int8Array(object);
new Int8Array(buffer [, byteOffset [, length]]);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int8Array

Int8Array Object Example

```
// Create a raw data buffer
var buf = new ArrayBuffer(64);
console.log(buf.byteLength);        // 64

// 8-bit signed integer view on buffer
var arr8 = new Int8Array(buf);

// Length represents view
console.log(arr8.length);          // 64

// Put/get values
arr8[0] = -2;
console.log(arr8[0]);              // -2
```

Uint8Array Object

- Represents an array of **8-bit unsigned integers**
- Contents are initialized to 0
- Can access elements using the object methods or bracket notation

Constructors

```
new Uint8Array(length);
new Uint8Array(typedArray);
new Uint8Array(object);
new Uint8Array(buffer [, byteOffset [, length]]);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array

Uint8ClampedArray Object

- Represents an array of **8-bit unsigned integers clamped to 0-255**
- Contents are initialized to 0
- Can access elements using the object methods or bracket notation

Constructors

```
new Uint8ClampedArray(length);
new Uint8ClampedArray(typedArray);
new Uint8ClampedArray(object);
new Uint8ClampedArray(buffer [, byteOffset [, length]]);
```

Summary of Typed Array Views

Type	Size in bytes	Description	Web IDL type	Equivalent C type
Int8Array	1	8-bit two's complement signed integer	byte	int8_t
Uint8Array	1	8-bit unsigned integer	octet	uint8_t
Uint8ClampedArray	1	8-bit unsigned integer (clamped)	octet	uint8_t
Int16Array	2	16-bit two's complement signed integer	short	int16_t
Uint16Array	2	16-bit unsigned integer	unsigned short	uint16_t
Int32Array	4	32-bit two's complement signed integer	long	int32_t
Uint32Array	4	32-bit unsigned integer	unsigned long	uint32_t
Float32Array	4	32-bit IEEE floating point number	unrestricted float	float
Float64Array	8	64-bit IEEE floating point number	unrestricted double	double

XMLHttpRequest instances' send() method supports typed arrays and ArrayBuffer objects as an argument

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

Examples

```javascript
// Create a raw data buffer
var buf = new ArrayBuffer(64);
console.log(buf.byteLength);        // 64

// 16 bit view on buffer
var arr16 = new Uint16Array(buf);

// Length represents view
console.log(arr16.length);          // 32

// Create 8-bit unsigned view on buffer
var clamped = new Uint8ClampedArray(buf);
console.log(clamped.length);        // 64

// Attempt to assign values outside clamped range
clamped[0] = 256;
console.log(clamped[0]);            // 255

clamped[0] = -1;
console.log(clamped[0]);            // 0

// Proof that both views access data in common Arraybuffer
clamped[0] = 100
console.log(arr16[0])               // 100
```

DataView Object

- A low-level interface that provides a getter/setter API to read/write arbitrary data to/from a buffer

Constructor

```
new DataView(buffer [, byteOffset [, byteLength]])
```

Properties

`.buffer`

- An existing ArrayBuffer to use as the storage for the new DataView object.

`.byteOffset`

- The offset, in bytes, to the first byte in the specified buffer for the new view to reference. If not specified, the view of the buffer will start with the first byte.

`.byteLength`

- The number of elements in the byte array. If unspecified, length of the view will match the buffer's length.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView

DataView Object Reading Methods

`.getInt8()`

- Gets a signed 8-bit integer (byte) at the specified byte offset from the start of the view.

`.getUint8()`

- Gets an unsigned 8-bit integer (unsigned byte) at the specified byte offset from the start of the view.

`.getInt16()`

- Gets a signed 16-bit integer (short) at the specified byte offset from the start of the view.

`.getUint16()`

- Gets an unsigned 16-bit integer (unsigned short) at the specified byte offset from the start of the view.

`.getInt32()`

- Gets a signed 32-bit integer (long) at the specified byte offset from the start of the view.

`.getUint32()`

- Gets an unsigned 32-bit integer (unsigned long) at the specified byte offset from the start of the view.

`.getFloat32()`

- Gets a signed 32-bit float (float) at the specified byte offset from the start of the view.

`.getFloat64()`

- Gets a signed 64-bit float (double) at the specified byte offset from the start of the view.

DataView Object Writing Methods

`.setInt8()`
- Stores a signed 8-bit integer (byte) value at the specified byte offset from the start of the view.

`.setUint8()`
- Stores an unsigned 8-bit integer (unsigned byte) value at the specified byte offset from the start of the view.

`.setInt16()`
- Stores a signed 16-bit integer (short) value at the specified byte offset from the start of the view.

`.setUint16()`
- Stores an unsigned 16-bit integer (unsigned short) value at the specified byte offset from the start of the view.

`.setInt32()`
- Stores a signed 32-bit integer (long) value at the specified byte offset from the start of the view.

`.setUint32()`
- Stores an unsigned 32-bit integer (unsigned long) value at the specified byte offset from the start of the view.

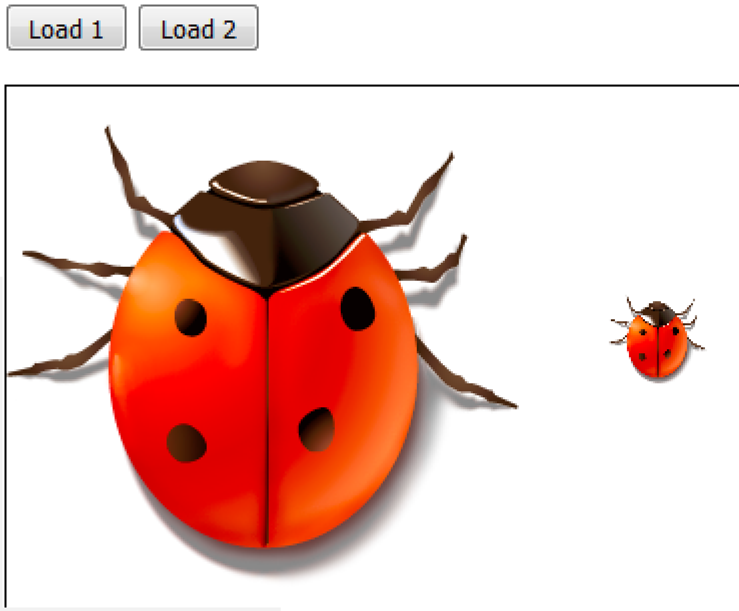
`.setFloat32()`
- Stores a signed 32-bit float (float) value at the specified byte offset from the start of the view.

`.setFloat64()`
- Stores a signed 64-bit float (double) value at the specified byte offset from the start of the view.

DataView Example

```
var b = new ArrayBuffer(16);
var v = new DataView(b, 0);

console.log( b.byteLength );     // 16

v.setUint16(0, 42);
v.getUint16(0);                  // 42

v.setUint16(0, 255);
v.getUint8(0);                   // 0
v.getUint8(1);                   // 255

v.setUint16(0, 256);
v.getUint8(0);                   // 1
v.getUint8(1);                   // 0
```

Recall: Drawing Images



```
// Create and load an image
var img = new Image();
img.onload = function(e) {
  console.log("bug.png loaded");
};
img.src = "bug.png";

// Get canvas and context
var cvs = document.getElementById("cvs");
var ctx = cvs.getContext("2d");

// Handle button clicks
document.getElementById("btn1").onclick = function(e) {
  ctx.drawImage( img, 0, 0 );
};

document.getElementById("btn2").onclick = function(e) {
  ctx.drawImage( img, 300, 100, 50, 50 );
};
```

09/image1.html

ImageData Object

- Represents the underlying pixel data of an area of a `<canvas>` element.
- Created using the `ImageData()` constructor or creator methods on the `CanvasRenderingContext2D` object associated with a canvas:
  - `createImageData()`
  - `getImageData()`
- Can also be used to set a part of the canvas by using `putImageData()`

<u>Properties</u>

`.data`

- A **Uint8ClampedArray** representing a one-dimensional array containing the data in the RGBA order, with integer values between 0 and 255.

`.height`

- An unsigned long representing the actual height, in pixels, of the ImageData.

`.width`

- An unsigned long representing the actual width, in pixels, of the ImageData.

https://developer.mozilla.org/en-US/docs/Web/API/ImageData

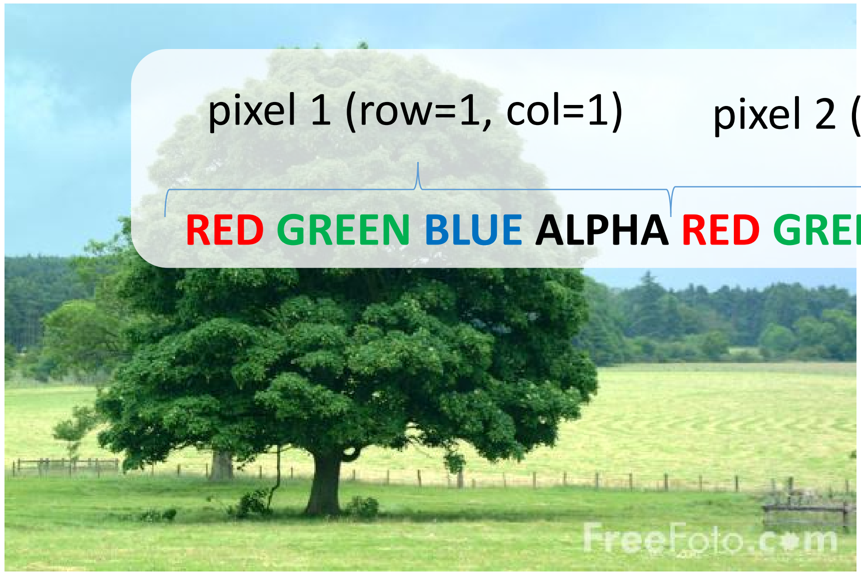
Example: ImageData 1

```
// Load an image into an in-memory Image.
var img = new Image();
img.src = "tree.jpg";

img.onload = function(e) {
    var cvs = el('cvs');
    var w = img.width;
    var h = img.height;

    // Once loaded, resize canvas to match image
    cvs.width = w;
    cvs.height = h;

    // Draw image onto canvas
    var ctx = cvs.getContext('2d');
    ctx.drawImage(img, 0, 0);

    // Get an ImageData object from context
    var idata = ctx.getImageData(0, 0, w, h);

    // Print Uint8ClampedArray length and anticipated length
    console.log( "ImageData length is", idata.data.length );
    console.log( "Expected length is", w*h*4);
};
```

Note. Each pixel is made up of four bytes: red, green, blue and alpha

Example: ImageData 1

- An ImageData's data property is a Uint8ClampedArray
- Each pixel in the image is made up of four bytes in the array
- The length of an ImageData's data (Uint8ClampedArray) is four times the number of pixels in the image
- The four bytes representing a pixel are ordered in the array as:



$$\overbrace{\textbf{RED GREEN BLUE ALPHA}}^{\text{pixel 1 (row=1, col=1)}}\ \overbrace{\textbf{RED GREEN BLUE ALPHA}}^{\text{pixel 2 (row=1, col=2)}}\ \overbrace{\textbf{RED GREEN BLUE ALPHA}}^{\text{pixel 3 (row=1, col=3)}}\ \textbf{...}$$

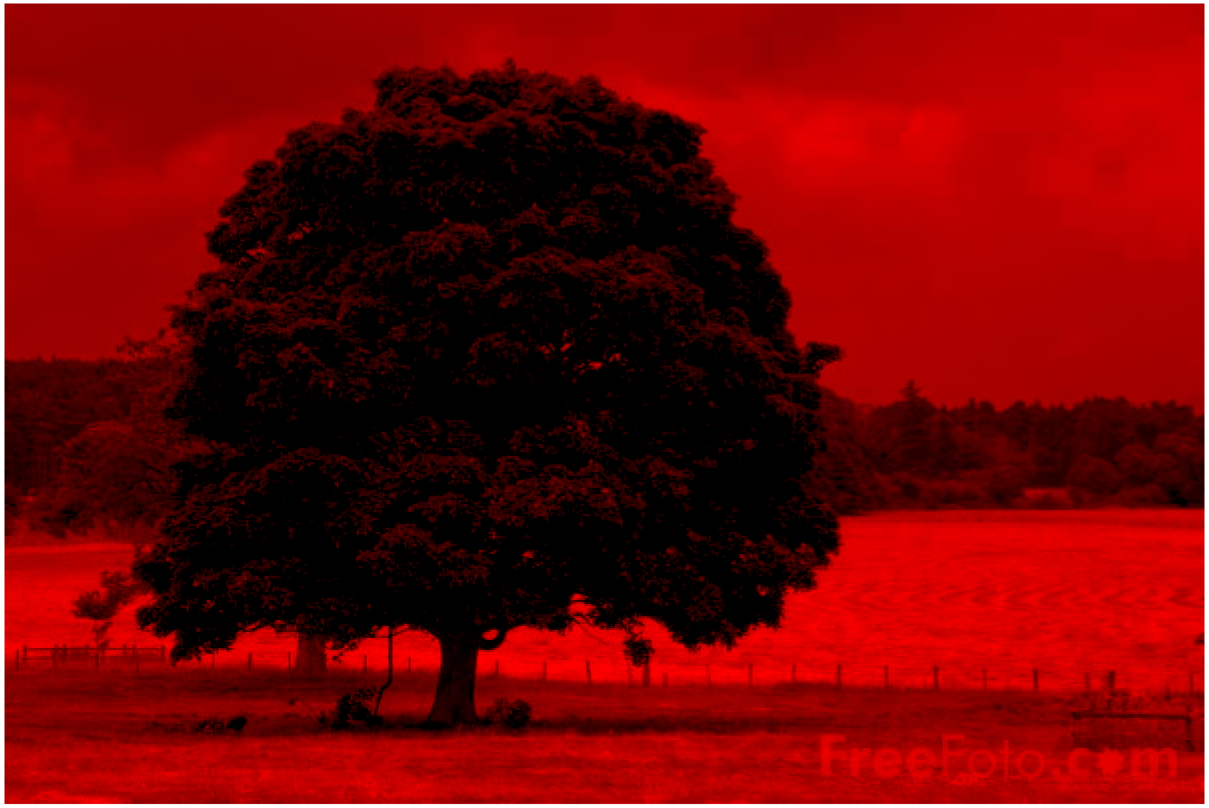
Example: ImageData 2

- Clear blue and green channels, leaving only red

```
var el = document.getElementById.bind(document);
var cvs = el('cvs');
var ctx = cvs.getContext('2d');

// Load an image into an in-memory Image.
var img = new Image();
img.src = "tree.jpg";

img.onload = function(e) {
    // Resize canvas to match image
    cvs.width  = img.width;
    cvs.height = img.height;

    // Draw image onto canvas
    ctx.drawImage(img, 0, 0);
};
```

```
// Remove the green and blue channels
el('bRed').onclick = function(e) {
    var w = cvs.width;
    var h = cvs.height;

    // Get an ImageData object from context
    var idata = ctx.getImageData(0, 0, w, h);

    // Visit all pixel data
    var arr = idata.data;
    for (var i=0; i<arr.length; i+=4) {
        arr[i+1] = 0; // Clear green channel
        arr[i+2] = 0; // Clear blue channel
    }
    // Draw updated image data back to canvas
    ctx.putImageData(idata, 0, 0);
};
```

Example: ImageData 2



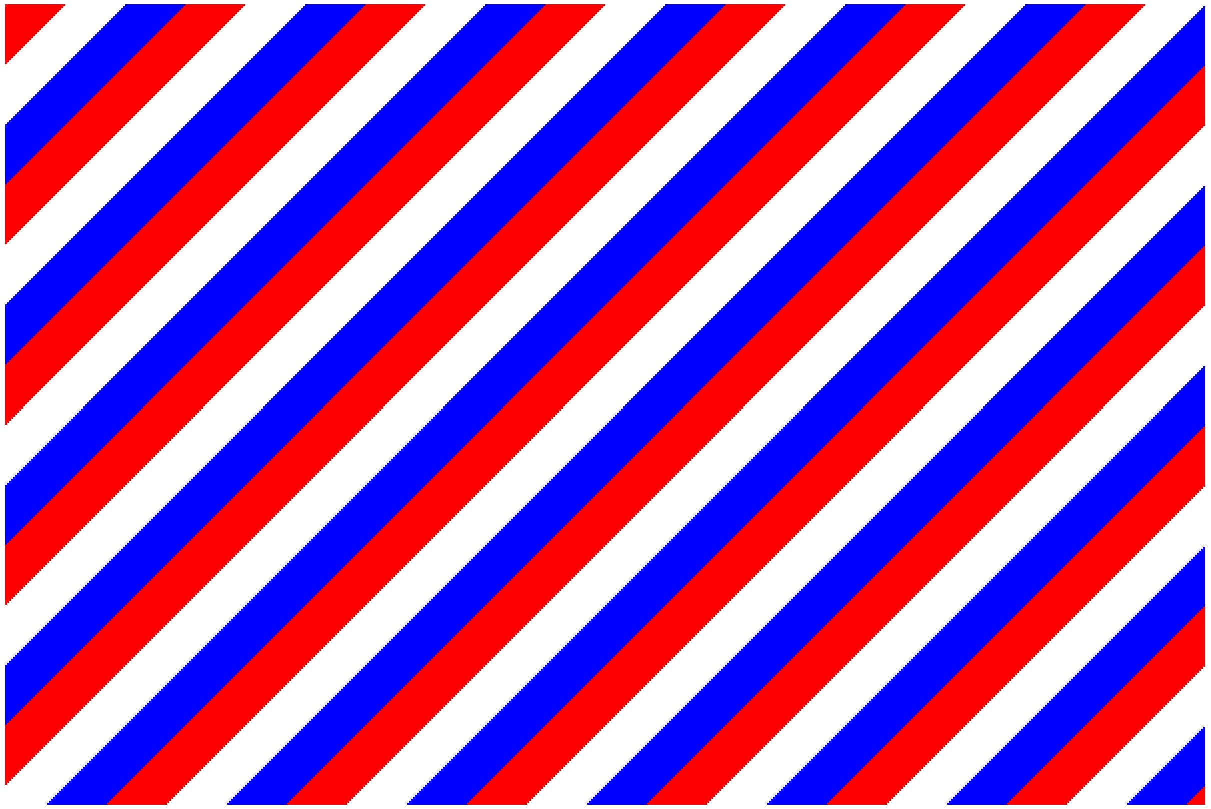
```
var el = document.getElementById.bind(document);
var cvs = el('cvs');
var ctx = cvs.getContext('2d');

// Stripes
el('bStripe').onclick = function(e) {
    var w = cvs.width;
    var h = cvs.height;

    // Create a new ImageData object
    var idata = ctx.createImageData(w, h);

    // Set all pixels
    var arr = idata.data;
    for (var rr=0; rr<h; ++rr) {
        for (var cc=0; cc<w; ++cc) {
            // pixel number
            var i = 4*(rr*w + cc);

            // Compute red, green and blue values
            var j = Math.floor((rr+cc)/30) %  3;
            if (j === 0) {
                r = 255; g = 0; b = 0; a = 255;
            } else if (j === 1 ) {
                r = g = b = a = 255;
            } else if (j === 2) {
                r = g = 0; b = a = 255;
            }

            // Set read, green and blue bytes for pixel i
            arr[i]   = r;
            arr[i+1] = g;
            arr[i+2] = b;
            arr[i+3] = a;
        }
    }
    // Draw updated image data back to canvas
    ctx.putImageData(idata, 0, 0);
};
```

Example: ImageData 3

- A Uint8ClampedArray may be created and populated from scratch
- This may then be saved to an ImageData object and drawn to a Canvas

11/idata3.html

Example: ImageData 3



11/idata3.html

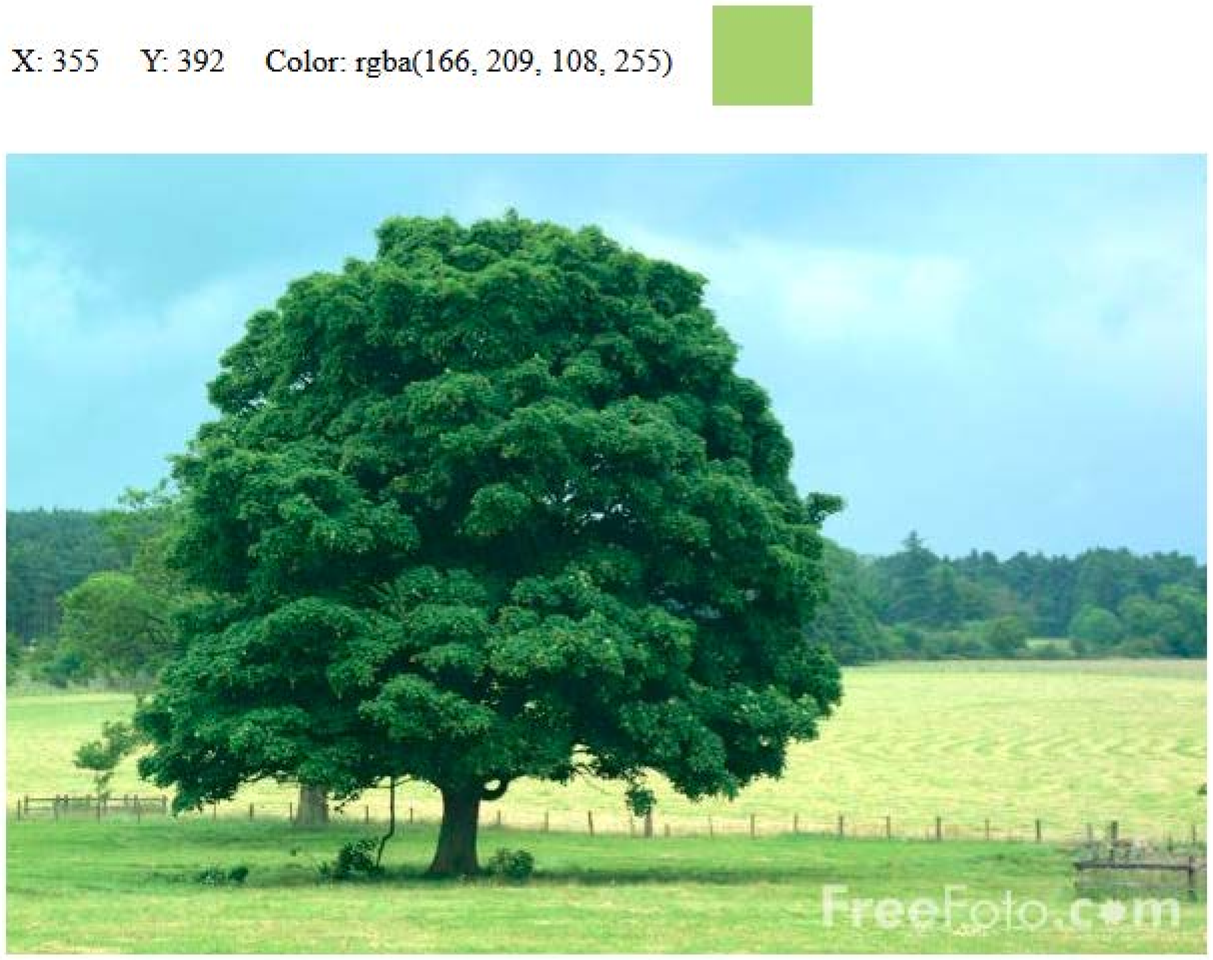
Example: Color Inspector

```javascript
var el = document.getElementById.bind(document);
var cvs = el('cvs');
var ctx = cvs.getContext('2d');
var swatch = el('swatch').getContext('2d');
var idata = null;
var w = 0;
var h = 0;

// Load Image
var img = new Image();
img.src = "minion.png";
//img.src = "tree.jpg";
img.onload = function(e) {
    cvs.width  = img.width;
    cvs.height = img.height;
    ctx.drawImage(img, 0, 0);

    // Set a few globals
    w = cvs.width;
    h = cvs.height;
    idata = ctx.getImageData(0, 0, w, h);
};
```

```javascript
// Handle canvas mousemove event
cvs.onmousemove = function(e) {
    var ob = getMousePos(cvs, e);
    el('x').innerHTML = ob.x;
    el('y').innerHTML = ob.y;

    if (idata) {
        var i = 4*(ob.y*w + ob.x);
        var r = idata.data[i];
        var g = idata.data[i+1];
        var b = idata.data[i+2];
        var a = idata.data[i+3];
        var clr = 'rgba(' + r + ', ' + g + ', '
                  + b + ', ' + a + ')';
        el('clr').innerHTML = clr;
        swatch.fillStyle = clr;
        swatch.fillRect(0, 0, 50, 50);
    }
}

// Get mouse position as an object
function getMousePos(cvs, evt) {
    var rect = cvs.getBoundingClientRect();
    return { x: Math.floor(evt.clientX - rect.left),
             y: Math.floor(evt.clientY - rect.top) };
}
```

11/inspector.html

Example: Color Inspector



```
X: 355     Y: 392     Color: rgba(166, 209, 108, 255)
```

11/inspector.html

Another way to create ImageData

- Create an appropriately sized ArrayBuffer and view as a Uint8ClampedArray
- Replace the ImageData's data using the Uint8ClampedArray's set() method

```
// Stripes
el('bStripe').onclick = function(e) {
    var w = cvs.width;
    var h = cvs.height;

    // Create a new ImageData object
    var idata = ctx.createImageData(w, h);
    //var arr = idata.data;

    // Create a new Uint8ClampedArray of appropriate size
    var arr = new Uint8ClampedArray( new ArrayBuffer(4*w*h) );

    // Set all pixels
    // <snip>

    // Set Uint8ClampedArray in new ImageData
    idata.data.set( arr );

    // Draw updated image data back to canvas
    ctx.putImageData(idata, 0, 0);
};
```

We may calculate image pixel values and save in an Array, but draw at a later time

Accessing Image Pixels - Summary

- `ImageData` object properties include:

`.height`	: `ImageData` height in pixels
`.width`	: `ImageData` width in pixels
`.data`	: the internal array of pixel color data (`Uint8ClampedArray`)

- Use `ctx.getImageData(…)` to get an existing `ImageData` object representing some part of a canvas
- Use `ctx.createImageData(…)` to create a new, empty `ImageData` object
- Use the `Uint8ClampedArray.set(…)` method to replace an entire array of pixel data (`myImageData.data.set(…)`)
- Use `ctx.putImageData(…)` to draw an `ImageData` object to some location on a canvas

Data URLs

- The image on a canvas may be converted to something known as a "Data URL"
- This is a (long) encoded string representing an image from the canvas
- Image format may be specified (image/png, image/jpg, …)

Example:

```
var dataURL = cvs.toDataURL( "image/png" );
```

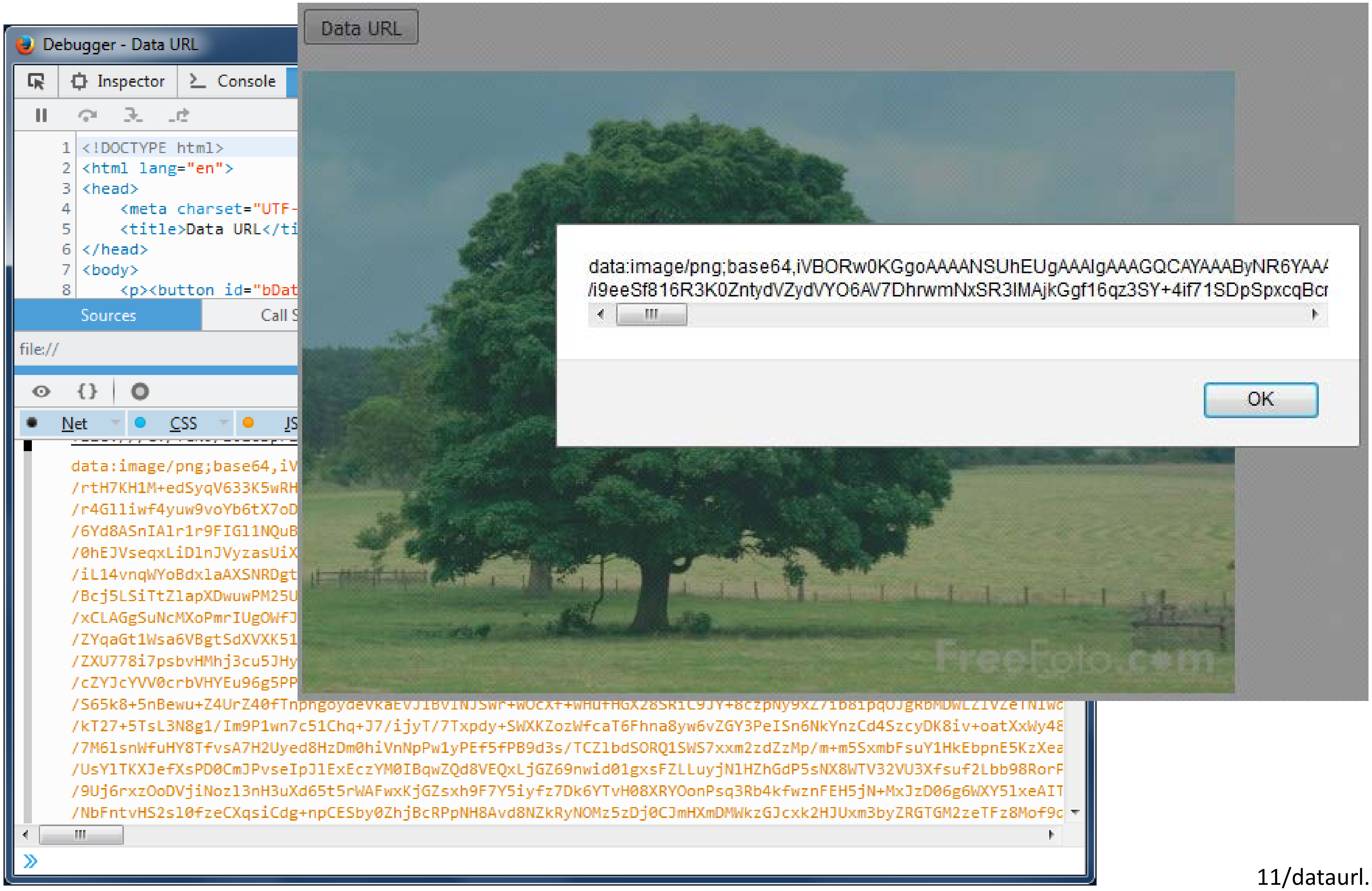
With dataURLs…

- images may be included directly in the text of a web page. No need to load images from separate files
- Image data may be streamed over a network as text or written to a text file. No need for binary buffers
- A snapshot of a canvas may be captured and saved in a variable, and restored at a later time (e.g. to implement an undo stack)

Example: Data URL

```
var el = document.getElementById.bind(document);
var cvs = el('cvs');
var ctx = cvs.getContext('2d');

// Load an image into an in-memory Image.
var img = new Image();
img.src = "tree.jpg";

img.onload = function(e) {
    cvs.width = img.width;
    cvs.height = img.height;
    ctx.drawImage(img, 0, 0);
};

// Remove the green and blue channels
el('bDataURL').onclick = function(e) {
    var durl = cvs.toDataURL("image/jpg");
    console.log(durl);
    alert(durl);
};
```

Example: Data URL



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-
    <title>Data URL</ti
</head>
<body>
    <p><button id="bDat
```

data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAIgAAAGQCAYAAAByNR6YAA
/i9eeSf816R3K0ZntydVZydVYO6AV7DhrwmNxSR3IMAjkGgf16qz3SY+4if71SDpSpxcqBcr

data:image/png;base64,iV
/rtH7KH1M+edSyqV633K5wRH
/r4G1liwf4yuw9voYb6tX7oD
/6Yd8ASnIA1r1r9FIG11NQuB
/0hEJVseqxLiDlnJVyzasUiX
/iL14vnqWYoBdxlaAXSNRDgt
/Bcj5LSiTtZlapXDwuwPM25U
/xCLAGgSuNcMXoPmrIUgOWfJ
/ZYqaGt1Wsa6VBgtSdXVXK51
/ZXU778i7psbvHMhj3cu5JHy
/cZYJcYVV0crbVHYEu96g5PP
/S65k8+5nBewu+Z4UrZ40fTnpngoydevkaEvJ1BvlNJSwr+wOcXf+wHufHGX28SRlC9JY+8czpNy9xZ71b81pqOJgRbMDWLZlVZeTNlwc
/kT27+5TsL3N8g1/Im9P1wn7c51Chq+J7/ijyT/7Txpdy+SWXKZozWfcaT6Fhna8yw6vZGY3PeISn6NkYnzCd4SzcyDK8iv+oatXxWy48
/7M61snWfuHY8TfvsA7H2Uyed8HzDm0hiVnNpPw1yPEf5fPB9d3s/TCZ1bdSORQ1SWS7xxm2zdZzMp/m+m5SxmbFsuY1HkEbpnE5KzXea
/UsY1TKXJefXsPD0CmJPvseIpJ1ExEczYM0IBqwZQd8VEQxLjGZ69nwid01gxsFZLLuyjN1HZhGdP5sNX8WTV32VU3Xfsuf2Lbb98RorF
/9Uj6rxzOoDVjiNoz13nH3uXd65t5rWAFwxKjGZsxh9F7Y5iyfz7Dk6YTvH08XRYOonPsq3Rb4kfwznFEH5jN+MxJzD06g6WXY51xeAIT
/NbFntvHS2sl0fzeCXqsiCdg+npCESby0ZhjBcRPpNH8Avd8NZkRyNOMz5zDj0CJmHXmDMWkzGJcxk2HJUxm3byZRGTGM2zeTFz8Mof9c

11/dataurl.html

Data URLs

- Data URLs may also be loaded directly into an Image object
- Assign the `.src` property of an Image object as you would any other URL

Example:

```
var el = document.getElementById.bind(document);
var cvs = el('cvs');
var ctx = cvs.getContext('2d');

// Load an image into an in-memory Image.
var img = new Image();
img.src = "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAlgAAAGQCAYAAAByNR6YAAAgAElEQVR

img.onload = function(e) {
    cvs.width = img.width;
    cvs.height = img.height;
    ctx.drawImage(img, 0, 0);
};
```

http://www.html5canvastutorials.com/advanced/html5-canvas-load-image-data-url/