Javascript

Contents

Javascript

Javascript

- no types for variables. `var myVar = 'Bob';` (js is *loosely* typed)
  - `const` for constants (can change elements of const arrays/properties of const objects), `let` for variables with *block scope*
- use `===` to test for equality, `==` attempts to convert types.
- arrays can hold any types, *index-valued*. `var myArray = [1, 'Bob', 'Steve', true];`
  - arrays *are* objects, and can hold objects
  - array methods and properties:
    * `myArray.length`
    * `fruits.forEach(someFunction);`
    * `var nums2 = nums1.map(someFunction);` Creates a new array by performing some function on each element.
    * `filter(someFunction)` and `reduce(someFunction)`

        - Using map, filter, and reduce
        - `every(someFunction)` and `some(someFunction)`
        - `fruits.push('Lemon');`
        - `var x = fruits.pop();`
        - `shift()` removes and returns first array element, shifts array, `unshift()` adds element and returns new length.
        - `fruits.join(' * ');` result: `Banana * Apple * Orange`
        - `fruits.splice(pos, num, ... newItems);` Adds newItems starting at pos after deleting num items. (can use to delete)
        - `fruits.concat(...);`
        - `var newFruits = fruits.slice(start, end);`
        - `addNums(...nums);` Use spread operator to split up arrays.
- strings can be surrounded by single or double quotes
    - can use template strings with backticks to embed expressions and format text / data: `my name is ${name}.`
- objects can also hold many values in *key-value* pairs. `var car = { type: 'fiat', model: 500, color: 'white' };`
    - can access properties: `car.model`
    - can use for...in loop: `for (var in object) {...}`

Example constructor:

```
function Person(first, last, age, eye) {
  this.firstName = first;
  ...
  this.eyeColor = eye;
}
```

- everything is an object and can be stored in a variable.
    - javascript *objects* are mutable and passed by reference

Function syntax:

```
function multiply(num1, num2) {
  var result = num1 * num2;
  return result;
}
```

```
/* can assign to variables */
document.querySelector('html').onclick = function() {};

/* arrow-function syntax */
var mult = function(x, y) {
  return x * y;
};

/* function return should be constant */
const mult = (x, y) => { return x * y };
const mult = (x, y) => x * y;
const other = x => x.funct();
const prices = phones.map(phone => phone.price);

/* also allows for lexical binding of this */
var foo = function() {
  var self = this;
  return function(bar) {
    return someFunct(self.property) + bar; /* this keyword would refer to function,
   instead of object */
  };
};

var foo = () => (bar => someFunct(this.property) + bar); /* can use this keyword */
```

Using generators:

```
function* gen() {           /* syntax for generators */
  yield console.log('pear'); /* yield pauses the iterator, and js runs right to left
   */
  yield console.log('banana');
  yield 'apple';            /* will return an object with a value and a done bool */
  var v = yield 'something'; /* can also pass values into generator */
  /* could be used with asynchronous programming... */
}
```

```
var myGen = gen(); /* myGen is an iterator */
myGen.next();
myGen.next();
myGen.next(10);
```

- ecma 6 improvements include spread operator, object literal enhancements, template strings, arrow functions, and generators

jQuery

- jQuery commands start with '$'
  - eg. `var heading = $("#some-id");` (returned in a wrapper, can only use jQuery methods then)
  - then, we can do: `heading.css({position: "relative"});` or `heading.animate({left: 100});`
  - can use filters to refine css selectors, eg. `$("header nav li:first")`, `$("section:not('#contact')")`, or `$("div[class]")`
  - can also use `.prev()`, `.parents()`, `.find(".someClass")` to traverse DOM
  - `.append()`, `.before()`, `.html()`, `.text()` adds and changes contet
  - wrap, unwrap, wrapAll, empty, remove, removeAttr, attr, removeClass, addClass, toggleClass
  - on, off (binding and unbinding events), click (event helper)
    - eg. `myLis.on("click", someCallback);`
    - can check the event object with target, type, pageX properties
    - use event.stopPropagation() to disable recursive behavior
  - safe to use document ready / window load
  - can use animate animate CSS properties with numerical value, fadeOut, fadeIn
    - hide, show, toggle, slideUp, slideDown, sildeToggle
  - can use call back functions to fix animations / fix delays

Node.js

- no document/window object, this in node is a global object
  - console, setInterval, `__dirname`
- function expression, set anonymous function equal to a variable
  - can also pass functions to another (variables can hold anything)

- require modules (other js files, modularizes cold)
  - use exports object to export variables / functions, can have multiple properties, use object literal notation
  - eg. var counter = require('./counter'); require returns whatever is specified to be exported
  - also can require core modules, eg. events, fs for files
- use event emitters to tie a callback function to be fired when some event is emitted
- fs has both synchronous and asynchronous functions, async require a call back function
  - asynchronous functions run in the background
- client - server communication uses different protocols such as https or ftp
  - then uses tcp protocol to send packets over a socket to the client
    - headers are also sent with requests and responses, eg. content-type or status
  - program can listen for requests sent to a particular port number on ip
  - buffer is a temporary storage spot for chunks of data at a time
    - streams of data flow from data source to buffer to client

Server setup:

```
var http = require('http'); /* core module */
var server = http.createServer(function(req, res) {
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type': 'text/plain'}); /* status, object */
  res.end('data');
});

server.listen(3000, '127.0.0.1'); /* have to listen on a port, no routing yet */
console.log('now listening on port 3000');
```

- node.js can read or write data from streams
  - streams can be writeable, readable, or duplex (both)

Using streams:

```
var fs = require('http');
var myReadStream = fs.createReadStream(__dirname + '/readMe.txt', 'utf8'); /* need
```

```
    file encoding type */
var myWriteStream = fs.createWriteStream(__dirname + '/writeMe.txt');
/* stream sends chunks of data at a time */

myReadStream.on('data', function(chunk){
  console.log('new chunk received:');
  console.log(chunk);
  myWriteStream.write(chunk);
})

myReadStream.pipe(myWriteStream); /* alternatively, can use pipes */

var server = http.createServer( (req, res) => {
  /* res.writeHead(200, {'Content-Type': 'text/plain'}); */
  res.writeHead(200, {'Content-Type': 'text/html'}); /* sending html to the client */
  var myReadStream = fs.createReadStream(__dirname + ...);
  myReadStream.pipe(res);
}) /* streams are more performance-efficient */

/* sending json, eg. an api endpoint */
res.writeHead(200, {'Content-Type': 'application/json'});
var someObj = {
  name: 'Jim',
  job: 'Ninja',
  age: 29
};
/* res.end(someObj); not a string! */
res.end(JSON.stringify(someObj));

if (req.url === '/home' || req.url === '/') {
  res.writeHead(200, {'Content-Type': 'text/html'});
  fs.createReadStream(...).pipe(res);
} else if (req.url === '/contact') {
  ...
} else if (req.url === '/api') {
  ... send some json data
```

```
} /* this can be simplified using express */
```

- npm: node package manager, can utilize different pacakges
  - eg. express framework for node
  - package.json file keeps track of npm dependencies
    - `npm init` builds package.json file
    - `npm install ... -save` saves packages into the json
    - `npm install` installs all dependencies
  - nodemon can automatically update / refresh the server on changes
- express: a node package
  - has easy routing system, integrates with template engines, contains middleware framework (can be extended by other packages)

Using express:

```
var express = require('express');
var app = express();

app.set('view engine', 'ejs'); /* express will use ejs as the view engine, in /views
   /...ejs */

/* http methods: get, post, delete, put (type of requests) */
app.get('/', function(req, res) {
  /* res.send('homepage'); no content type specification needed */
  res.sendFile(__dirname + '/index.html');
});
app.get('/contact', function(req, res) {
  ...
});
app.get('/profile/:id', function(req, res) {
  res.send('profile ' + req.params.id);
  var data = {age: 29, job: 'ninja', hobbies: ['eating','fighting']};
  res.render('profile', {person: req.params.name, data: data}) /* render a view! (
    profile.ejs) */
});

app.listen(3000);
```

- templating engines allow us to embed data into html files
- eg. ejs is a lightweight templating engine and a node package

```
<!-- inside the ejs -->
<h1> welcome to the profile of <&= person %></h1>
<p> Age: <%= data.age %></p>
<p> Job: <%= data.job %></p>
<ul>
  <% data.hobbies.forEach(function(item){ %>
    <li><%= item %></li>
  <% }); %>
</ul>
```

- can make partial views / templates to save code (eg. a nav bar across all views)
  - place inside the partials folder inside the views
  - `<% include partials/nav.ejs %>`
- using css within ejs:
  - have to deal with the request for a static files such as css, images, etc
  - can use express *middleware* to deal with this
    - code that runs *between* the request and the response

Using middleware with express:

```
app.use('/assets', function(req, res, next) {
  conseole.log(req.url);
  next(); /* goes to the next middleware */
});

app.use('/assets', express.static('assets')); /* essentially linking the assets
   folder to the server */
/* eg. put css inside the assets */
```

- query strings are additional data added on to url requests
  - in name / value pairs, denoted by ?, separated by &
  - eg. mysite.com/news?page=2&dept=marketing
  - we need to parse the request and pull out the data
  - express parses these query strings for us

```
app.get('/contact', (req, res) => {
console.log(req.query); /* will print an object in name-value pairs of the
   query */
res.render('contact', {qs: req.query});
});
```

- *POST* requests: asks the server to store/accept data in the body of request
  - usually used when submitting forms
  - different when compared with query strings

An example project...

In the html:

```
<!-- using a POST method, action is the URL we are posting to -->
<form id="contact-form" method="POST" action="/contact">
  <label for="who">Who do you want to contact?</label>
  <input type="text" name="who" value="<%= qs.person %>">
  ...
</form>
```

In the js:

```
/* need additional middle-ware, eg. body-parser npm package */
var bodyParser = require('body-parser');
var urlencodeParser = bodyParser.urlencoded({ extended: false});

app.post('/contact', urlencodeParser, function(req, res) {
  console.log(req.body);
  res.render('contact-success', {data: req.body});
});

/* app.js */
var express = require('express');
var app = express();
var todoController = require('./controllers/todoController');
```

```
app.set('view engine', 'ejs');       /* template engine */
app.use(express.static('./public')); /* maps static files to the public folder */

todoController(app); /* fire controllers */

app.listen(3000);
console.log('listening on port 3000');

/* better to split code into different modules / files */
/* MVC structure: models (data), views (template files, ejs), and controls (controls
    app sections, eg. todoController, userController) */

/* controllers/todoController.js handle routes, data, views, etc. */
var mongoose = require('mongoose');
var bodyParser = require('body-parser');

mongoose.connect('mongodb://test:test@...'); /* connect to mongodb */
var todoSchema = new mongoose.Schema({
  item: String
}); /* need a schema, like a blueprint (what kind of info to expect) */

var Todo = mongoose.model('Todo', todoSchema); /* made a model */

/* var itemOne = Todo({item: 'todo1'}).save(function(err){
   if (err) throw err;
   console.log('item saved');
 }); */

var urlencodeParser = ...
/* var data = [{item: 'todo 1'}, {item: 'todo2'}]; */

module.exports = function(app) {
  app.get('/todo', function(req, res) {
    /* get data from mongodb */
    Todo.find({}, function(err, data) {
      if (err) throw err;
```

```
      res.render('todo', {todos: data});
    });
    /* res.render('todo', {todos: data}); */
  });
  app.post('/todo', urlencodedParser, function(req, res) {
    /* get data from view and add it to mongodb */
    var newTodo = Todo(req.body).save(function(err, data) {
      if (err) throw err;
      res.json(data);
    });
    /* data.push(req.body); */
    /* res.json(data); */
  });
  app.delete('/todo/:item', function(req, res) {
    /* delete from mongodb */
    Todo.find({item: req.params.item.replace(/\-/g, " ")}).remove(function(err, data)
    {
      if (err) throw err;
      res.json(data);
    })
    /* data = data.filter(function(todo) {
       return todo.item.replace(/ /g, '-') !== req.params.item;
     });
     res.json(data); */
  });
};
```

In the ejs:

```
<!-- views/todo/ejs -->
<html>
  <head>
    <title>Todo Lists</title>
    <script src=jquery.js ...></script>
    <link href="/assets/styles.css" rel="stylesheet" type="text/css" />
```

```
  </head>
  <body>
    <h1>My Todo List</h1>
    <div id="todo-table">
      <form>
        <input type="text" name="item" placeholder="Add new item..." required />
        <button type="submit">Add Item</button>
      </form>
      <ul>
        <% for(var i = 0; i < todos.length; i++){ %>
          <li><%= todos[i].item %></li>
        <% } %>
      </ul>
    </div>
  </body>
</html>
```

- noSQL is a data base, alternative to SQL, works well with JSON and js
  - used with mongodb, and mongoose npm package

Asynchronous JS

- always has a callback function
  - runs on multiple threads, "asynchronously"
  - flow control can be handled in different ways:
    - callbacks, promises, generators
- AJAX requests:
  - communicate with a server with a http request, no reload the page
  - stands for Async JS and XML (XML is data, can also retrieve in JSON)

Vanilla js vs. jquery:

```
// vanilla js request
var http = new XMLHttpRequest();
http.onreadystatechange = function() {
  if(http.readyState == 4 && http.status == 200) {
    console.log(JSON.parse(http.response)); // 4 different ready states
```

```
  }
};
http.open("GET", "data/tweets.json", true);
http.send();

// jquery alt
$.get("data/tweets.json", function(data) {
  // callback function
  console.log(data);
});
// ex. callback function
var fruits = ["banana", "apple", "pear"];
fruits.forEach(function(val) {
  ...
  // this is a callback function that runs on every val in fruits
  // could also have non-inline callback functions
  // this callback function is called synchronously
});

$.get("data/tweets.json", function() {
  console.log(data); // async callback function
  // can also be non-inline
});

// callback hell:
$.ajax({ // get alternative
  type: "GET",
  url: "data/tweets.json",
  success: function(data){
    console.log(data);
    $.ajax({
      ...
      success: function(data){
        console.log(data);
        ...
      }
```

```
    });
  },
  error: function(jqXHR, textStatus, error) {
    console.log(error);
  }
});

// some cleanup:
function handleError(...) {
  console.log(error);
};
function cbTweets(data) {
  // split up the callbacks into different functions, instead of nested ones
  ...
};
```

- can also use promises
  - promises are objects that represents actions that haven't yet finished
  - promise objects are given before the data is actually retrieved

Using promises:

```
// vanilla js:
function get(url) {
  return new Promise(function(resolve, reject){
  // resolve applies to the .then function, and reject falls to the .catch function
    var xhttp = new XMLHttpRequest();
    xhttp.open("GET", url, true);
    xhttp.onload = function() {
      if (xhttp.status == 200) {
        resolve(JSON.parse(xhttp.response));
      } else {
        reject(xhttp.statusText);
      }
    };
    xhttp.onerror = function() {
```

```
      rejext(xhttp.statusText);
    };
    xhttp.send();
  });
}

var promise = get("data/tweets.json");
promise.then(function(tweets) {
  console.log(tweets);
  return get("data/friends.json");
}).then(function(friends) { // can chain ajax requests
  console.log(friends);
}).catch(function(error) { // can chain
  console.log(error);
});

// jquery promise built-in library:
$.get("data/tweets.json").then(function(tweets) { // returns a promise
  console.log(tweets);
  return $.get("data/friends.json");
}).then(function(friends) {
  console.log(friends);
  return $.get("data/videos.json");
}).then(function(videos) {
  console.log(videos);
});
```

Async/Await:

```
async function init() {
  await createPost(...); // waits for a promise/async process to complete
  getPosts;
}

async function fetchUsers() {
```

```
  const res = await fetch(...); // alternative to .then syntax
  // but you can't await multiple promises... not as flexible as raw promises
  // instead may have to await a promise.all to solve this
  const data = await res.json();
  console.log(data);
}

// can also use Promise.all to chain together promises
const promise1 = Promise.resolve("hello");
const promise2 = 10;
const promise3 = new Promise((reslove, reject) => {
  setTimeout(resolve, 2000, 'Goodbye');
});
Promise.all([promise1, promise2, promise3]).then(values =>
  console.log(values);
); // promise all runs all the promises, and values is an object containing the
    returned object data
```

Async js with generators (functions that can paused):

```
function* gen(){
  var x =  yield 10;
  console.log(x);
}

var myGen = gen();
var y = myGen.next(); // object with value and a done boolean
myGen.next(15);

// with sync js:
genWrap(function*(){
  var tweets = yield $.get("data/tweets.json");
  console.log(tweets);
  var tweets = yield $.get("data/friends.json");
  console.log(friends);
```

```
  var tweets = yield $.get("data/videos.json");
  console.log(videos);
});

function genWrap(generator){
  var gen = generator(); // prepare generator
  function handle(yielded){
    if(!yielded.done){
      yielded.value.then(function(data){
        return handle(gen.next(data));
      })
    }
  }
  return handle(gen.next());
}
```