Objectives

- Introduction to Javascript

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab10` by typing:

```
$ mkdir lab10
```

Change to this directory by typing:

```
$ cd lab10
```

Exercise: Hello Javascript

Getting the Starting Code for the Javascript Exercises

First clone the repo which has all the starting code for the COMP[29]041 JavaScript labs.

```
$ git clone https://github.com/COMP2041UNSW/js
...
```

Now run the Node Package Manager (NPM)

```
$ cd js
$ 2041 npm install
...
```

For later labs you can use the command below to start up a simple server, this is not needed for this lab but feel free to give it a run if your curious.

```
$ 2041 npm start
...
```

The command will make the web-server available at http://127.0.0.1:8080

Completing Your First Javascript program

Now that you are all set up go into the folder with the first lab exercise

```
$ cd labs/lab10/hello_world
$ gedit hello_world.js & # or your favourite editor
```

And fill in the function in **hello_world.js** to prepend a string passed in with the string "Hello " and return the result.
The file **test.js** contains some driver code to help you test your code.

It will run your **hello** function with the supplied command line argument as input. For example:

```
$ 2041 node test.js world
Hello world
$ 2041 node test.js Andrew!
Hello Andrew!
```

Working from Home

You can complete all the Javascript exercises in your own computer.

You'll first need to install **node** on your computer. You can find installation instructions here: https://nodejs.org/en/download/

If you have problem installing **node** ask in the course forum.

You run the same commands on your own compute,r but without the 2041 prefix

```
$ cd js
$ npm install
....
$ cd labs/lab10/hello_world
$ vim hello_world.js
$ node test.js world
Hello world
$ node test.js Andrew!
Hello Andrew!
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_hello_world
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_hello_world hello_world.js
```

Sample solution for `hello_world.js`

```
// write a program to prepend a string passed by the function
// with the word, with a space ie: "Hello "
// should return the new string.
module.exports = function hello(name) {
   return 'Hello ' + name;
};

// or `Hello ${name}`

// or "Hello ".concat(name)

// probably many other ways.
```

Exercise: Debugging Javascript

Now read the code in **lab10/variables/variables.js**.
It contains a function **doubleIfEven** which takes a single number **n** as argument.

If **n** is even, **doubleIfEven** should return double **n**

If **n** is odd, **doubleIfEven** should return **n**

Also present in the same directory is **test.js** which allows you test **doubleIfEven**

However the code in **variables.js** has bugs:

```
$ cd lab10/variables/variables.js
$ node test.js 3
false // should be 3
$ node test.js 2
2 // should be 4
```

Your task is to fix the bugs.
When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_variables
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_variables variables.js
```

Sample solution for `variables.js`

```
/*
 * This code is broken! Can you figure out why
 * and fix it?
 */

function doubleIfEven(n) {
  let x = n;
  if (even(x)) return double(x);
  return x;
}

function even(a) {
  let x = a;
  if (x%2==0) x = true;
  else x = false;
  return x;
}

function double(a) {
    return a*2;
}

module.exports = doubleIfEven;
```

Exercise: Javascript Sports

Your task is to write a Javascript function **countStats** which given a list of career statistics for a team of rugby players returns the total number of matches they have played and the total number of tries they have scored.
countStats will be given a list in this format:

```
[
    {
        "id": 112814,
        "matches": "123",
        "tries": "11"
    }
]
```

countStats should return an object of the form:

```
{
    "matches": m,
    "tries": t
}
```

Where *m* is the sum of all matches for all games and *t* is the sum of all tries for all games.
For example, given this input

```
[
    {"id": 1,"matches": "123", "tries": "11"},
    {"id": 2,"matches": "1",   "tries": "1"},
    {"id": 3,"matches": "2",   "tries": "5"}
]
```

countStats should return

```
{
    matches: 126,
    tries: 17
}
```

Some code to help you get started is provided in **lab10/count_sport_stats/count_sport_stats.js**.
You are also given example input data in the file **stats.json**.

As usual a **test.js** file is also provided to help you test your code:

```
$ node test.js stats.json
{ matches: 1725, tries: 165 }
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_count_sport_stats
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_count_sport_stats count_sport_stats.js
```

Sample solution for `count_sport_stats.js`

```
// a functional programmming style solution using reduce

function countStats(list) {
   return list.reduce((acc, curr) => ({
      matches: +curr.matches + acc.matches,
      tries: +curr.tries + acc.tries
   }), {
       matches: 0,
       tries: 0
   });
};

module.exports = countStats;
```

Alternative solution for `count_sport_stats.js`

```
// an iterative solution

function countStats(list) {
    var total_matches = 0;
  var total_tries = 0;

    for (var i = 0; i < list.length; i++) {
    // The + is needed to convert the string to an intger
    total_tries += +list[i].tries;
    total_matches += +list[i].matches;
    }

    return {
    matches: total_matches,
    tries: total_tries
  };
}
```

Exercise: Javascript Sports 2

Write a Javascript function **makeTeamList** which takes 3 arguments: a list of career statistics for a team of rugby players, a list of player names, and a list of team names.
makeTeamList's first argument will describe the team with an object in this format:

```
{
    "players": [
        {
            "id": 112814,
            "matches": "123",
            "tries": "11"
        }
    ],
    "team": {
        "id": 10,
        "coach": "John Simmons"
    }
}
```

makeTeamList's second argument will be a list of player names in this format:

```
[
    {
        "id": 112814,
        "name": "Greg Growden"
    }
]
```

makeTeamList's thirds argument will be a list of team names in this format:

```
[
    {
        "id": 10,
        "team": "NSW Waratahs"
    }
]
```

makeTeamList should returns a 'team sheet' that lists the team, coach, players in that order in the following format:

```
[
    "Team Name, coached by CoachName",
    "1. PlayerName",
    "2. PlayerName"
    ....
]
```

Each element should be a string.
The players should be ordered by the number of matches the have played, from most to least.

For example, give these 3 arguments as input

```
{
    "players": [
        {"id": 1,"matches": "123", "tries": "11"},
        {"id": 2,"matches": "1",   "tries": "1"},
        {"id": 3,"matches": "2",   "tries": "5"}
    ],
    "team": {
        "id": 10,
        "coach": "John Simmons"
    }
}
```

```
[
    {"id": 1, "name": "John Fake"},
    {"id": 2, "name": "Jimmy Alsofake"},
    {"id": 3, "name": "Jason Fakest"}
]
```

```
[
    {"id": 10, "team": "Greenbay Packers"}
]
```

makeTeamList should return a list containing exactly these strings:

```
[
    "Greenbay Packers, coached by John Simmons",
    "1. John Fake",
    "2. Jason Fakest",
    "3. Jimmy Alsofake"
]
```

Some code to help you get started is provided in **lab10/join_sport_stats/join_sport_stats.js**.

You are also given example input data in the files **team.json**, **names.json** and **teams.json**.

As usual a **test.js** file is also provided to help you test your code:

```
$ node test.js team.json names.json teams.json
[ 'NSW Warratahs, coached by Barry Cassidy',
  '1. Ronaldo',
  '2. Buffalo Bill',
  '3. Jesse James',
  '4. Cleopatra I',
  '5. Marc Antony' ]
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_join_sport_stats
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_join_sport_stats join_sport_stats.js
```

Sample solution for `join_sport_stats.js`

```
function makeTeamList(teamData, namesData, teamsData) {
    // Take it step by step.
    let { players, team } = teamData;
    const names = namesData;
    const teams = teamsData;

    // do the first join
    const playerMap = [...players, ...names].reduce((map, current) => {
        const { id, ...rest } = current;
        if (map[id]) {
            map[id] = { ...map[id], ...rest}
        } else {
            map[id] = { ...rest };
        }

        return map;
    }, {});

    const teamName = teams.find(t => t.id === team.id).team

    players = Object.values(playerMap)
        .map(({ name, matches }) => ({
            name,
            matches: parseInt(matches)
        }))
        .sort((a, b) => b.matches - a.matches)
        .map((player, index) => `${index + 1}. ${player.name}`);

    return [`${teamName}, coached by ${team.coach}`, ...players];
}

module.exports = makeTeamList;
```

Alternative solution for `join_sport_stats.js`

```
// an iterative solution

function makeTeamList(teamData, namesData, teamsData) {

    const team_name = get_team_name(teamData.team.id, teamsData)

    const team_details = team_name + ", coached by " + teamData.team.coach;
    let team_sheet = [team_details];

    const players = teamData.players.sort((a, b) => b.matches - a.matches);

    for (var i = 0; i < players.length; i++) {
        const player_name = get_player_name(players[i].id, namesData);
        team_sheet.push((i + 1) + ". " + player_name)
    }

    return team_sheet
}

function get_team_name(id, teamsData) {
    for (var i = 0; i < teamsData.length; i++) {
        if (id == teamsData[i].id) {
            return teamsData[i].team;
        }
    }
}

function get_player_name(id, namesData) {
    for (var i = 0; i < namesData.length; i++) {
        if (id == namesData[i].id) {
            return player_name = namesData[i].name;
        }
    }
}

module.exports = makeTeamList;
```

Challenge Exercise: Inheriting a Dog

Your task is to create an Javsacript script constructor function **Dog** which creates a simple object which stores information about a dog.
Your constructor function **Dog** will be given the dog's name and age. It should store this stores this information in the object.

Your object also should have a function **toHumanYears** which returns how old the Dog is in human years (assume 1 dog year == 7 human years).

```
const good_dog = Dog("sam", 9)
console.log(good_dog.name)  // should print sam
console.log(good_dog.age)   // should print 9
console.log(good_dog.toHumanYears())   // should print 63
```

Make your **Dog** constructor function inherit from the provided Animal constructor function given in **lab10/prototype_inheritance/objects.js**

```
console.log(good_dog.__proto__) // should print Animal { makeSound: [Function] }
console.log(good_dog.makeSound() // should print out 'woof'
```

As usual a **test.js** file is also provided to help you test your code:

```
$ node test.js
woof
Animal { makeSound: [Function] }
70
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_prototype_inheritance
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_prototype_inheritance objects.js
```

Sample solution for `objects.js`

```
/*
 * given a name and a age make a Dog object
 * which stores this information
 * and which has a function called
 * toHumanYears which returns how old the
 * Dog is in human years (1 dog year is 7 human years) (not really but lets pretend)
 *
 * const me = Dog("sam",91)
 * me.name should be "sam"
 * me.age should be 91
 *
 * make Dog such that it is inheriting from the provided
 * Animal class
 *
 * me.__proto__ should be Animal
 * me.makeSound() should print out 'woof'
 *
 */

function Animal(age) {
    this.age = age
}

Animal.prototype.makeSound = function() {
    console.log(this.sound)
}

function Dog(name, age) {
    Animal.call(this, age)
    this.name = name
    this.sound = 'woof'
    this.toHumanYears = function() {
        return this.age*7;
    }
}

Dog.prototype = Animal.prototype;

module.exports = Dog;
```

Challenge Exercise: What Have You Drunk?

In **lab10/objects/objects.js** there is part of the code for a **Person** object,
This object tracks how the person spending on drinks in UNIbar.

Finish up the missing functions to keep track of drink orders.

You need to write the function **buyDrink** which is given as argument a drink object, for example:

```
{
     name: "beer",
     cost: 8.50,
     alcohol: true
 }
```

buyDrink add the cost of the drink to the person's tab (total drinks bill) if the person is

1. old enough to drink (over 18 if the drink is alcohol)
2. buying the drink will not push their tab over $1000

If these conditions don't hold the function should do nothing.
You also need to write write a function **getRecipt** which returns a summary of all drinks a person bought, grouped by name, for example:

```
[
    {
        name: beer,
        count: 3,
        total: 25.50
    },
    {
        name: cola,
        count: 1,
        total: 2.50
    }
]
```

Some code to help you get started is provided in **lab10/objects/objects.js**.
As usual a **test.js** file is also provided to help you test your code:

test.js expects three arguments the name and age of the drinker and the name of a json file containing a list of drinks.

For example:

```
$ node test.js Andrew 21 drinks.json
[ { name: 'cola', count: 2, total: 7 },
  { name: 'beer', count: 1, total: 5.5 },
  { name: 'fanta', count: 1, total: 3.5 } ]
$ node test.js  Lisa 21 drinks.json
[ { name: 'cola', count: 2, total: 7 },
  { name: 'fanta', count: 1, total: 3.5 } ]
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_objects
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_objects objects.js
```

Sample solution for `objects.js`

```
/*
 * Fill out the Person prototype
 * function "buyDrink" which given a drink object which looks like:
 * {
 *     name: "beer",
 *     cost: 8.50,
 *     alcohol: true
 * }
 * will add the cost to the person expences if the person
 * is
 *    1. old enough to drink (if the drink is alcohol)
 *    2. buying the drink will not push their tab over $1000
 *
 * in addition write a function "getRecipt" which returns a list as such
 * [
 *    {
 *        name: beer,
 *        count: 3,
 *        total: 25.50
 *    }
 * ]
 *
 * which summaries all drinks a person bought by name in order
 * of when they were bought (duplicate buys are stacked)
 */

function Person(name, age) {
    this.name = name;
    this.age = age;
    this.tab = 0;
    this.history = {};
    this.historyLen = 0;
    this.canDrink = function() {
      return this.age >= 18;
    };
    this.canSpend = function(cost) {
      return this.tab + cost <= 1000;
    }
}

Person.prototype.buyDrink = function(drink) {
  if (!this.canSpend(drink.cost)) return;
  if (drink.alcohol && !this.canDrink()) return;

  if(this.history[drink.name]){
    this.history[drink.name].count++;
    this.history[drink.name].total+=drink.cost;
  } else {
    this.history[drink.name] = {
      name: drink.name,
      count: 1,
      total: drink.cost,
      order: this.historyLen++
    }
  }
  this.tab += drink.cost;
}

Person.prototype.getRecipt = function() {
  let all = Object.keys(this.history);
  all = all.sort((a,b)=>this.history[a].order - this.history[b].order);
  let r = [];
  for(let drink of all) {
    let d = this.history[drink];
    delete d.order;
    r.push(d);
  }
  return r;
}

module.exports = Person;
```

Challenge Exercise: Javascript Piping

Write a function called buildPipe that returns a function which runs all of it's input functions in a pipeline

consider the follwing code

```
let timesTwo = (a) => a*2;
let timesThree = (a) => a*3;
let minusTwo = (a) => a - 2;
let pipeline = buildPipe(timesTwo, timesThree, minusTwo);
```

in his case pipeline(x) is the same as minusTwo(timesThree(timesTwo(x)))

```
 pipeline(6) == 34
```

The framework is given in **lab10/piping/piping.js**

As usual a **test.js** file is also provided to help you test your code:

```
$ node test.js
34
32
```

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest js_piping
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs2041 lab10_js_piping piping.js
```

Sample solution for `piping.js`

```
/*
 * Write a function called buildPipe that returns a function
 * which runs all of it's input functions in a pipeline
 * let timesTwo = (a) => a*2;
 * let timesThree = (a) => a*3;
 * let minusTwo = (a) => a - 2;
 * let pipeline = buildPipe(timesTwo, timesThree, minusTwo);
 *
 * pipeline(6) == 34
 *
 * pipeline(x) in this case is the same as minusTwo(timesThree(timesTwo(x)))
 *
 * test with `node test.js`
 */

function buildPipe(...ops) {
  const _pipe = (a, b) => (arg) => b(a(arg));
  return ops.reduce(_pipe);
}

/* Alternative recursive solution:

function buildPipe(f, ...fs) {
    if (fs.length > 0) return x => buildPipe(...fs)(f(x));
    return x => f(x);
}
*/

module.exports = buildPipe;
```

Submission

When you are finished each exercises make sure you submit your work by running **give**.
You can run **give** multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Saturday 5 January 19:00** to submit your work.

You cannot obtain marks by e-mailing lab work to tutors or lecturers.

You check the files you have submitted here

Automarking will be run by the lecturer several days after the submission deadline for the test, using test cases that you haven't seen: different to the test cases `autotest` runs for you.

(Hint: do your own testing as well as running `autotest`)

After automarking is run by the lecturer you can view it here the resulting mark will also be available via via give's web interface

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 2041 classrun -sturec
```

The lab exercises for each week are worth in total 1 mark.
The best 10 of your 11 lab marks will be summed to give you a mark out of 9. If their sum exceeds 9 - your mark will be capped at 9.

- You can obtain full marks for the labs without completing challenge exercises
- You can miss 1 lab without affecting your mark.