CAB230 Web Computing
Prac 1: Elementary JavaScript

In this unit, we will begin our exploration of JavaScript fairly gently. Some of you will dash through this prac very quickly, while others will find most of it new. But if you already know something of the language, at least skim through the document and use it as an opportunity to revise and to make sure that you haven't missed anything.

For those new to the language, your experience will depend on what you have seen before. If you've come from another modern programming language, many of the types will be familiar to you. Those of you who have some background in C or C++ or C# or Java will be more familiar with the general syntax of the language. This seems comforting, but the comfort can be deceptive. JavaScript is much less strongly typed than its C-like cousins.

There are subtleties in almost every part of JavaScript, and the types present many of the more common issues. This prac will cover the basics - you can explore more by clicking the link in the header of each section which will take you to the Mozilla Developer Network. The documentation there is dense, but complete.
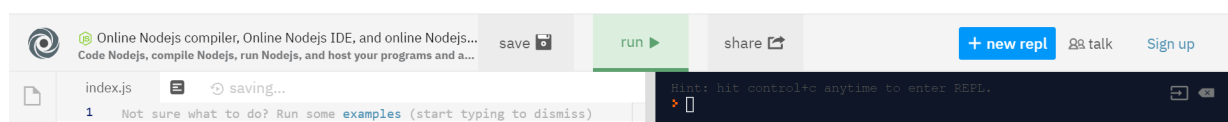
There will not be too much actionable code in this lesson - it's mostly getting a foundation. You will get to have more fun after you've learnt the basics.
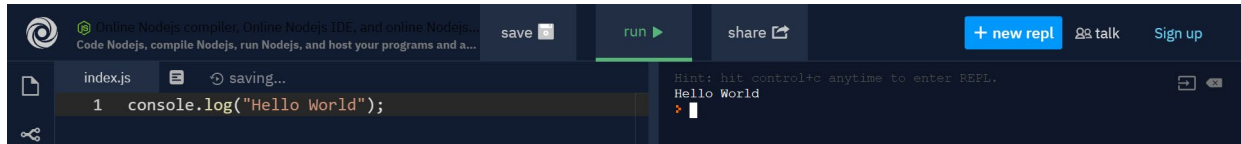
Getting Started.

When experimenting with JavaScript, it is best to use a REPL – or *Read-Eval-Print Loop* – a code window of some kind that sits on top of a JavaScript interpreter or compiler, usually node.js. We will teach you a lot about node later in the semester, and there will be instructions on BB if you wish to install it on your own machine. But here we will use a service called REPL.it. This provides free access to an online node.js REPL (along with many other languages), which has the advantage of some intellisense support. You can go to this link for direct access to a Node.JS repl. If there are issues loading this one – sometimes you may hit capacity limitations, try the Babel REPL.

All the following exercises can be completed either with Node.JS directly installed, or using a REPL.it REPL. Both will be referred to as the REPL. Any code blocks seen below can be entered directly. Note that only the last item in the list will be printed out on the screen. Semicolons (`;`) are generally *not* required in JavaScript (except in some specific cases) due to Automatic Semicolon Insertion. Semicolons will be used in examples for explicit clarity and as a result of our personal preferences.

As a quick example, I am going to work through a few simple statements and a loop in the REPL.it Node REPL. We begin with a blank window (the version shown is v10.16.0, and others will be used later. Currently, node is at v14.16.0, but all recent versions are fine for these purposes). You may find the output difficult to read in some cases, so run the exercises for yourselves.

I am going to print a straightforward Hello World message, and then surround it with a loop to make it print out ten times. All trivial, but the key is that the system helps you and displays the results. *Don't just read it* – however trivial it is, the point is to see it work. Similar results can be achieved in the node REPL and in a number of editors which provide REPL environments. For clarity, I use the settings menu to change to a dark layout and increase the font size. We first use the `console.log` output function. Click on run to execute the code:

We know that `console.log` is a function, so let's create a variable called `result` to hold the return value. We again print using the `console.log` output function:
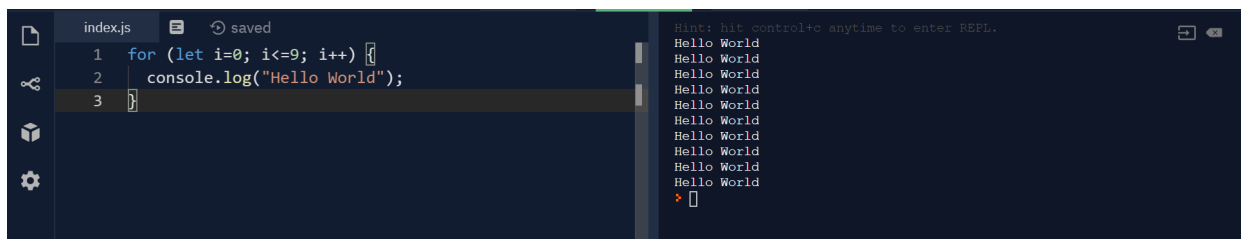

We see the value of the statement is undefined – it is there, though you may have to expand the image to see it. The log function does not return a value.

Using a syntax familiar from the lecture - and from other C-like languages if we know one - we next encase the Hello World log statement in a for loop:
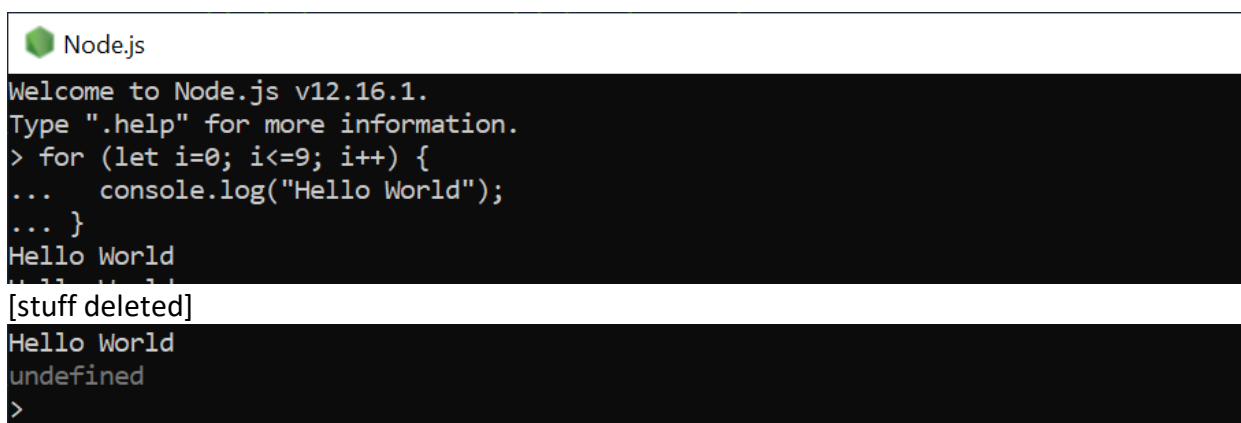

Don't worry about the use of let for now – we will look at this later in more detail. We see that "Hello World" appears multiple times as expected. That is what for loops are supposed to do. But if we run the same code in a local node window, which has different display settings, we see that the REPL offers "undefined" as the result.

[stuff deleted]

If we edit the code to remove the function, so that we have an empty loop executing ten times, we still have the same result. The main point at the moment is that you are doing nothing wrong. We will cover this below, but if the code statement or function does not have an associated value, then its value is `undefined`. Some REPLs show this explicitly and it can be irritating or worrisome to people learning JS. There are subtleties that aren't important to us at this stage. The full guide may be found at the link below – don't read it now unless you are an enthusiast, but the basic story is this:

A method or statement … returns undefined if the variable that is being evaluated does not have an assigned value. A function returns undefined if a value was not `returned`.

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

```
> for (let i=0; i<=9; i++) {
... }
undefined
>
```

For now, we will move on and look at some of the fundamentals of the language.

Basic Types.

These types (also known as 'primitive types') are the standard building blocks for JavaScript. As noted, JS is weakly-typed, which means that extra care may be required. You should take a look at the examples below, and if in doubt, type them into the REPL and see what happens.

Numbers

JavaScript has no difference between `int` and `float`. There are also no size-specific variants that exist in other languages, such as `float64` or `int32`.

```
1;        // a number
1.2;      // also a number
-1.2;     // a negative number
6.023e23  // scientific notation
```

You can perform mathematics on numbers, too.

```
1 / 3;
3 * 10;
6 + 4;
6 + 5 * 4;       // correct order is?
(6 + 5) * 4;     // parentheses clarify order
```

All arithmetic is JavaScript is "safe" - you will not get any errors performing operations. You may get some 'special' values, but you will not get a crash.

The 'common' special values are:
- NaN: 'Not a Number'. This is the result of any operation that doesn't result in a valid number, such as `1/0`.

- `Infinity`: The largest possible number expressible in JavaScript - there is a complete loss of precision at this point. `Infinity + 1 - 2` will not get you a 'real' number again. This is the same as overflow in other languages.
- `-Infinity`: The same as `Infinity`, but when dealing with negative numbers.

You can see other special values here, but you likely won't need to use these for a long time.

All numbers in JavaScript are 64-bit binary floating point. This results in a couple of 'gotchas' to keep in mind (but don't worry too much about them at this stage). For example, before running this in your REPL, think about what you expect to happen. But make sure you *run* these.

```
0.1 + 0.2 == 0.3; // boolean output
```

Note that this time the value is not undefined. Then go ahead and try the following:

```
// floating-point is *hard* - or at least tricky
0.1 + 0.2 == 0.30000000000000004;
```

Booleans

As with most languages, true and false are the Boolean values in JavaScript. They require no special constructor:

```
true;   // a boolean
false;  // also a boolean
```

As shown above, they may result from the use of the `==` operator:

```
1 == 1; // true
1 == 2; // false
```

They are also the result of using other operators:

```
1 > 1;   // is 1 greater than 1?
1 >= 1;  // is 1 greater than or equal to 1?
2 < 1;   // is 2 less than 1?
2 <= 1;  // is 2 less than or equal to 1?
```

But as noted in the lecture, Douglas Crockford is quick to point out that JS equality comparisons can be troublesome. Here we return to his discussion of the equality comparison operators `===` and `!==` and their 'evil twins' `==` and `!=`. As Crockford points out, the operators with three characters (*strict* equality and *strict* inequality) behave properly even when the types are different, while the evil twins (*abstract or loose* equality and *abstract or loose* inequality) do not necessarily behave as one might expect.

Here again are some of the comparisons from Crockford's examples. Try them out in the REPL to make sure that they are correctly stated, and then fix them by using `===` instead:

```
'' == 0;   // true
0 == '';   // true
0 == '0';  // true
```
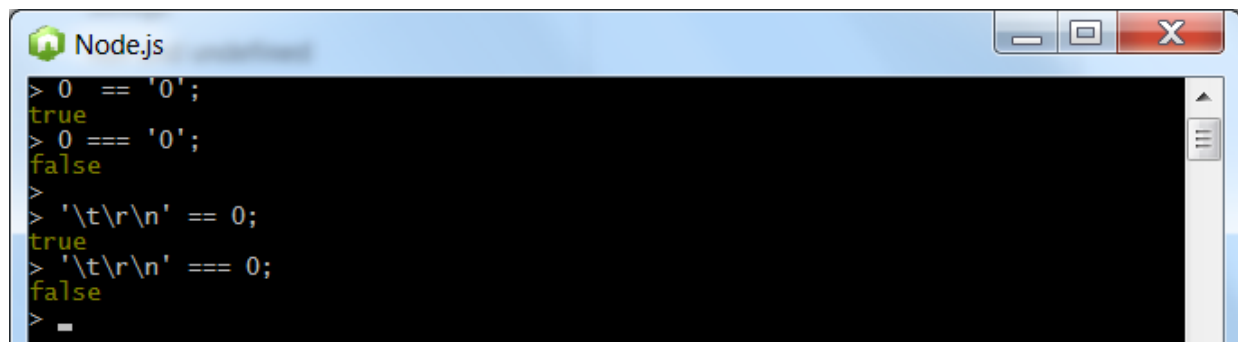
```
false == 'false';  // false
false == '0';       // true

false == undefined; // false
false == null;      // false
null == undefined;  // true

'\t\r\n' == 0;  // true
```

As an example, here is the node REPL showing a couple of these comparisons:

JavaScript of course also supports the usual logical operators, AND, OR and NOT. The syntax is simple and corresponds to that used in most C-like languages:

- && - AND or conjunction
- || - OR or disjunction
- ! – NOT or negation

Obviously we can work with logical expressions involving Boolean variables, but here we will focus only on the literal values. Here are some examples to try out in the REPL:

```
false && true;  // false
false || true;  // true
!false && true; // true
true && false || true;  // Order of operations?
```

Usually we will avoid the sort of confusion which may result in the last case by inserting parentheses to make our intentions clear to the machine and to any human readers.

Strings

Strings in JavaScript have, confusingly, 3 declaration syntaxes:

```
'I am a string';
"I am a string";
`I am also a string`;
```

There is no difference between a single-quote (') and a double-quote (") when creating a string - you just have to use the same type at the start and end of the text. This can be helpful if you want to use some kind of quote within your string, without having to use an escape mechanism:

```
"I will 'use' single-quotes";
'I will "use" double-quotes';
```

Strings declared this way can also only be a single line - if you want a 'line-break', you must put in the escape code for a line break: \n.

```
// the following code will *not* work
"This is an
 invalid  string"

// This is the correct way
"This is a\nvalid string"
```

Strings created with a back-tick are slightly different. They are technically known as a 'template literal' and allow for interpolation (and some other functionality which we won't need for a while). This allows you to insert other values into your strings.

```
`3 + 1 = ${3 + 1}`; // will print "3 + 1 = 4"
```

The interpolation is started with a ${, and ended with a }. These braces must contain an *expression* - so you can't declare a variable, but pretty much everything we've written so far is an *expression.* So, for those paying close attention, this *does* mean you can put a template literal inside the interpolation of another template literal.

Template literals can also contain newlines:

```
`You can put as
many
newlines as you like`;
```

Go ahead and try this in the REPL and see how it comes out. Note the difference from the other string delimiters.

Regular JavaScript strings may also be joined together with another string or with values of another type using the concatenation operator:

```
'I have ' + 5 + ' apples.' + ' How many do you have?';
```

Once more, you should try this in the REPL and see how it works out.

null and undefined

These are two types you need to know about, but won't make as much sense until we get to variable declarations and functions. They make even less sense when you try and compare them without using 'strict equals'.

- `null` is unlike the primitive types we have seen before. It represents the lack of a value, as it does in most other languages.
- `undefined` is slightly different - it represents a value that has not been defined.
- `null` must be explicitly assigned or returned.
- `undefined` is what you get when you do *not* explicitly assign or return.

We will cover these in more depth when looking at variables and functions, and their use will

become pretty clear through repetition.

Variables and Expressions

We have noted previously that JavaScript is weakly-typed, and that this causes problems for us in a variety of contexts. We have seen this emerge when using the loose equality comparison above, but the problems are especially troublesome when we mix types in an expression. As usual, an expression is a fragment of code which produces a value. When the types of the variables and values in the expression are the same, JS tends to behave itself. When we mix them up, JS tends to behave badly. We will now consider a few examples involving mixed types and unusual values.

The first two expressions use the value `null`:

```
8 * null;

8 * 5 - 2 + null;
```

What happens if we instead use `undefined`?

Now consider the cases below, taken or adapted from Eloquent JS:

```
"5" + 1;
"5" - 1;

"5" * 2;
"five" + 2;
"five" * 2;
```

Again try them out. Remember that some operators are overloaded, and that the plus sign has a role in arithmetic, but also in the concatenation of strings. JS makes some sense of the mixed types – when it sees an arithmetic operation, it tries to parse the string to yield a number - but maybe there are times when we would prefer it didn't. The lesson is to make sure that we ***never*** rely on this sort of behaviour.

In the remainder of this prac, we are going to focus on an example based on the Fibonacci Sequence (https://en.wikipedia.org/wiki/Fibonacci_number). As is very well known, each Fibonacci number is obtained as the sum of its two predecessors. We usually define the sequence as follows:

$$F_0 = 0$$
$$F_1 = 1$$

$$F_n = F_{\{n-1\}} + F_{\{n-2\}}$$

So, we quickly see that the sequence runs through:

$$0,1,1,2,3,5,8,13,21,34, \ldots$$

The Fibonacci numbers become very large more rapidly than one might expect, but it is not the same as a factorial or other multiplicative expression. Here we will write some very simple JS code and introduce a basic for loop to calculate the n-th Fibonacci number, F_n. At first, we will do this manually for two iterations. The simplest code looks something like this:

```js
let f_older = 0;
let f_old = 1;
let f_new = f_old + f_older;
console.log(f_new);
```

and we quickly obtain $F_2 = 1$. If you don't see this straight away, please work through it by hand on paper. Don't just assume it is ok.

What we now need to do is to update it all. So we see that:

```js
f_older = f_old;
f_old = f_new;
f_new = f_old + f_older;
console.log(f_new);
```
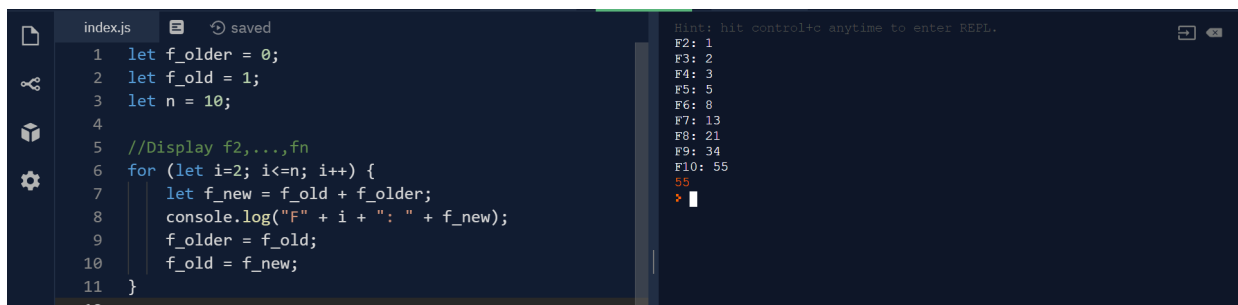
and this time we obtain $F_3 = 2$. Obviously, we need to include this code in a loop. As you have seen in the lecture, for loops in JS are structured in the usual way of C-like languages. The basic programming exercise is an easy one, but here our focus is on the structure:

```js
let f_older = 0;
let f_old = 1;
let n = 10;

//Display f2,...,fn
for (let i=2; i<=n; i++) {
    let f_new = …
    console.log("F" + i + ": " + f_new);
    f_older = …
    f_old = …
}
```

We have left a few gaps for you to fill in. The key issues are the use of the `let` declaration, and the usual for loop structure, with the initialization, the test and the increment, and the use of the concatenation operator in the log function.

Running this in the online REPL yields the following result:

Why are we left with 55 printed in red at the bottom?

Now, once you have the code in place and working, we are going to play with a few things. None of these make any sense at all – we are just going to use this as an example for the scope rules. At the bottom of the for loop, please include the following log statements:

```
console.log("f_old: " + f_old);
console.log("n: " + n);
console.log("i :" + i);
console.log("f_new: " + f_new);
```

Now run the code and see what happens. Remember that modern JS allows block scope. We know from the output above that we can print out i and f_new within the loop body, but we have an issue here.

What if we now remove the let statement in the for loop?

```
for (i=2; i<=n; i++) {
```

What about if we replace the let keyword with var?

```
for (var i=2; i<=n; i++) {
```

A few comments. First, just because this seems to be better, doesn't mean that it is. The let keyword helps us limit the scope of a variable to the region of the code in which we want it to work for us. If the variable is undeclared, then its scope is global, which means it can cause a great deal of grief elsewhere. The var keyword gives us function scope, but here all of the code lies within the one function, and so there is no difference from global.

One final variation. What if we were to go ahead and declare one of the variables using the const keyword?

```
let f_older = 0;
const f_old = 1;
let n = 10;
```

We quickly see that this will only work once. We may set the value this first time, but any updates will fail. But what happens if we restore this declaration and instead change the declaration in the loop body as follows:

```javascript
for (let i=2; i<=n; i++) {
    const f_new = f_old + f_older;
```

Why is this different?

For those who want one last exercise, take our Fibonacci code above and write a JavaScript function that returns the n-th Fibonacci number. You should need only one parameter.

```javascript
let fibonacci = function(n) {

    ...

    return …
}
```

Test your function with the following calls for $n = 5, 10, 15, 20$ and 50.

```javascript
let num = 10;
console.log("F" + num +": " + fibonacci(num));
```

Other Resources:

https://javascript.info/types
https://eloquentjavascript.net/01_values.html