JavaScript

PIC 40A, UCLA

JavaScript

JavaScript is a programming language that is used heavily in web development. Most websites that enable interactivity in responding to a user’s inputs use JavaScript to manage events. It also has many uses in generating more complicated webpages that would be a huge hassle to write with raw HTML.

Most JavaScript runs through a web browser, in a **window**. But JavaScript can also function as a standalone language, such as the case of **NodeJS** where it can run on the server.

JavaScript

JavaScript is a **high level** programming language. It is high level in that many elements that are present and may require attention in languages like C or C++ - including pointers, memory management, variable typing, etc. - are abstracted away.

It is an **interpreted language**: code that is written in JavaScript is parsed by another program, a JavaScript engine in the case of JavaScript, and this other program determines which instructions to carry out. This happens every time the program is run. This differs from a **compiled language**, such as C++, whereby a program can be converted to very efficient machine instructions and run ever after.

Fun fact: the Chrome V8 JavaScript Engine is written in C++!

JavaScript

JavaScript is also a **scripting language.** The term refers to the programs being written as a series of commands/statements that are not compiled, which goes hand-in-hand with it being an interpreted language.

ECMAScript is an international standard for scripting languages. JavaScript is a language that conforms to these standards, but it is not the only such language. The current **ECMA 6 Specifications can be found by clicking here**.

Writing JavaScript

JavaScript can be included in the **head** of a document. For most of the early examples, we will make use of the JavaScript debugging tools in the web browsers and can use an HTML document structure:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Demos</title>
  <script src="some_file.js" defer ></script>
</head>
<body>
</body>
</html>
```

Then code can be written in that **some_file.js** file. Of course the **src** can be a full URL as well, not just a relative path.

Writing JavaScript

There are many other ways to include JavaScript. The approach on the previous slide is, by current standards, the best, for reasons we shall elaborate upon.

JavaScript should be included inside of a **<script>...</script>** pair of tags, or it should be provided as an external resource with **src**.

In HTML4, the **script** elements required **type="text/javascript"**, but this is not required in HTML5.

Writing JavaScript

There is a difference between **src** and **href**. Notice the patterns

```
<a href="...">...</a>
<link href="..." />

<img src="..." />
<script src="...">...</script>
```

In general with **href**, a link is established to the other resource but the page continues to parse the HTML (or the user may be redirected in the case of anchors). With **src**, the resource is embedded in the current HTML document (this can slow down loading).

Writing JavaScript

The old school fix was to include the **script** tags just before the closing body tag.

With HTML5, two new boolean attributes were introduced, **async** and **defer.** Both allow for the HTML to be parsed concurrently with downloading the scripts. With **async**, if multiple scripts are included, the second may be fully loaded before the first; with **defer**, if multiple scripts are present, they are loaded in the order in the code.

JavaScript Overview

In order to be productive with JavaScript, it is important to understand it as a language in its own right, rather than as a bunch of scripts that come out of nowhere. Roughly speaking, amidst seeing some applications, we will consider:

- data types: primitives vs object
- type coercion
- arrays, functions, and other objects
- variables, copies, and references
- Document Object Model
- hoisting (initialization)
- this and closures
- control flow
- Math and Date objects
- events, timers, and asynchronous JavaScript
- cookies
- object oriented design
- libraries (jQuery)

Data Types

In JavaScript there are 6 **primitive types**:

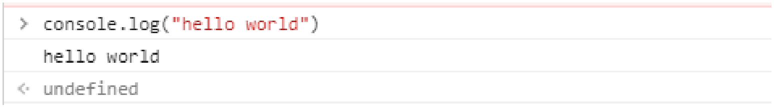
- **undefined**
- **null**
- **Boolean**
- **Number**
- **String**
- **Symbol**

Everything else, including **Arrays**, functions, and classes are **objects.**

All functions are objects in JavaScript. Think of this like the lambda expressions in C++: they are callable objects/classes.

Directly Writing JavaScript

To explore JavaScript code without focusing upon a website, through most web browsers, one can *inspect* an empty tab and go to the *console* view.



```
> console.log("hello world")
  hello world
<· undefined
```

console.log is a useful way to print things for debugging. In this *console* option, some browsers will append extra *undefined* statements indicating that a function/statement does not return a value.

console.log is like std::cout $<<$ in C++.

Data Types: undefined

In JavaScript, a variable can be defined with the **var**, **let**, or **const** keywords.

```
let x;
```

This says that **x** is a variable. The first time a variable comes into existence if it is not assigned a value, the JavaScript engine gives it the value **undefined**. So **undefined** is a value signifying it has not been set yet.

Remark: JavaScript will not give an error if you use a variable that is undefined. Your code will just behave badly...

In C++ terms this undefined value corresponds to when a variable has been defined but not explicitly initialized, often erroneously called "declared but not defined" by the uninitiated.

Data Types: null

If a variable should not have a value after it has already been in use, the programmer may choose to set the variable to **null**:

```
x = null; // now x is null
```

Yes, JavaScript comments are similar to C++ comments with // for single line and /* **...** */ for multi-line comments**.**

There is only one value that is null. This is like nullptr in C++ that is the only instance of std::nullptr_t.

Data Types: Boolean

These correspond to the typical **true** and **false** values of other programming languages. They are useful in control flow.

```
x = true;
x = false;
x = !false; // now true
```

! is logical negation.

Data Types: Number

In JavaScript there is only one data type for numbers, be they integers or floating type. The **Number** data type corresponds to a 64-bit floating point number.

```
let y = 1234567890; // y is a Number
let z = 3.14159265; // z is a Number
```

Data Types: Number

The arithmetic operations for Numbers behave the same in JavaScript as in C++ with: **+, -, *, /, %, +=, -=, *=, /=, %=**, assignment **=**, and prefix/postfix **++** and **- -.** In JavaScript, ** is the exponentiation operator.

```
let x = 3, y = 3;
y = y + 1; // y is now 4, returns value 4
x *= y; // x is now 12, returns value 12
++x; // x is now 13, returns value 13
y- -; // y is now 3, returns value 4

3**4; // 81: 3 to power of 4
```

Note: we can define multiple variables on a single line.

Data Types: String

The **String** datatype should be used to store textual information. A **String** can be declared with either single ’ or double **"** quotes.

```
let msg1 = "hello";
let msg2 = ’world’;
let msg3 = msg1 + " " + msg2; // will be string "hello world"
msg3 += ’!’; // will be string "hello world!"
```

As with C++, the + (and +=) can be used for string concatenation.

Data Types: String

JavaScript supports escape characters:

n for a new line,
t for a tab,
\\ for a backslash,
\’ for a single quote,
" for a double quote,
b for a backspace.

Data Types: String

In JavaScript, **String**s are immutable. The **String** contents themselves cannot be modified (but of course reassignment is allowed).

```
// msg3 was "hello world!"
msg3[0] = 'H'; // does nothing
```

Strings are indexed from 0.

Data Types: String

Here are a few useful things we can do within strings:

```
"hello".length; // extract length property, value is 5

let message = "hello world";
let pos = message.indexOf("world"); // 6, index of "world"
```

length is a property, so we do not call a function upon it.
indexOf returns the index where a substring is found, -1 if not found.

Data Types: String

```
let x = message.substring(0,5); // "hello"
```

substring takes a required first argument for the starting index and an optional past-the-end index as the second argument (by default will result in the rest of the string).

substr also exists and behaves the same as in C++ but it is now deprecated.

Remark: many other functions exist with similar functionalities, but we won't go into them.

Data Types: Symbol

The newest type introduced in the ECMA 6 Standard is **Symbol**. It effectively makes something that is unique in value. It can be used to allow functions to mark objects without interfering with program logic and to give **private**-like features to objects.

```
let x = Symbol("a"); // x and y are constructed the same way
let y = Symbol("a"); // but they are distinct and not equal
```

Alert, Confirm, Prompt

In JavaScript, we can display a message to the screen with an alert box. The syntax is:

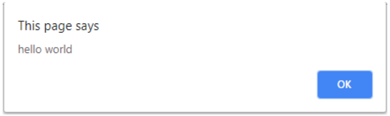
alert(some_message);

where **some_message** could be a string or some other variable.

With a script

```
let msg = 'hello world';
alert(msg);
```

we are greeted with an alert box:


This page says

hello world

Alert, Confirm, Prompt

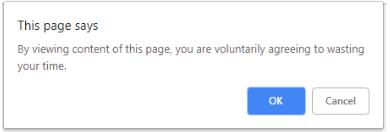
In JavaScript, the **confirm** function makes the user read a message and click **OK** or **Cancel**. It returns the value **true** if they clicked "OK" and false otherwise.

They cannot view other parts of the page until they have clicked OK/Cancel.

With a script

```
let decision = confirm("By viewing content of this page, you are voluntarily agreeing to
    wasting your time.");
```

we are greeted with a confirmation box:



This page says

By viewing content of this page, you are voluntarily agreeing to wasting your time.

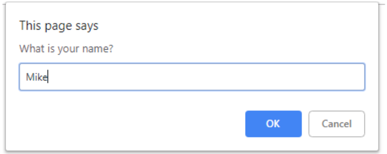
decision will be **true** or **false**.

Alert, Confirm, Prompt

The **prompt** function prompts a user for a piece of information. It takes arguments of the prompt text, a default value of the response, and it returns their answer.
With a script

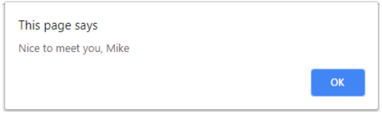
```
let name = prompt("What is your name?", "");
alert("Nice to meet you, " + name);
```

we can be greeted and welcomed


This page says

What is your name?

Mike


This page says

Nice to meet you, Mike

Data Types: Weakly Typed

JavaScript is weakly typed. Variables can be transformed from one type to another without any problems.

```
let x = 14; // x is a Number
x = ’foo’; // now it is a String
x = false; // now a Boolean
```

Data Types: Coercion

If JavaScript needs to operate on two values of different type, it may apply **coercion** to convert one data type into the other. This can be confusing!

Some of the conventions are familiar:

```
let x = 7, y = true;
x += y; // true converted to 1, x now 8
```

When needed, **Boolean** can be converted to **Number** with **true** becoming 1 and **false** becoming 0.

Data Types: Coercion

JavaScript does more, however...

The following values all are converted to **false** during coercion:

- 0 (the Number)
- ” or "" (the empty String)
- null
- undefined
- NaN (flag when a Number is not valid such as from division by 0)

Everything else is converted to **true** during coercion including things like:

- ’0’
- "false"
- [] (an empty array)
- {} (an empty object)

Data Types: Coercion

As a few examples of these coercions, consider a simple if-statement. The syntax is the same as in C++:

```
if("false") {
   console.log("hello"');
}

let x = 0/0; // so NaN

if(x) {
   console.log("hello again”);
}
```

Only the first statement prints.

Data Types: Coercion

This also makes "math" somewhat confusing:

```
3 * "7"; // === 21
"3" * "7"; // === 21
"12" - 9; // === 3
5 / "4"; // === 1.25
5 / "4 and some extra stuff"; // === NaN
4 + "3"; // === "43"
1 + 2 + "3"; // === "33"
1 + "2" + 3; // === "123"
```

When using **+**, JavaScript will concatenate Numbers and Strings. For **-**, *****, /, it will convert the String to a Number if it can or return **NaN** (not a number).

As with all programming languages, operators have precedence and associativity (direction). The **+** is a left-to-right operator so 1+2+"3" is first seen as (1+2)+"3"; whereas 1+"2"+3 is first seen as (1+"2")+3.

Data Types: Coercion

Note the **===** not **==**. In JavaScript both exist but the former is more strict.

```
false == 0; // true: false can be made into Number 0
false === 0; // false: Boolean and Number
```

Data Types: Coercion

Other rules exist, too, but we won’t get into them. It is important to be aware of the consequences of being weakly typed and allowing so many coercions!

The two most efficient ways to convert a Number to a String:

```
let n = -3;
let x = n + ""; // add empty string
let y = n.toString(); // call toString method
```

Simple ways to convert a String to a Number:

```
let z = x * 1; // just mutiply by 1... or subtract 0... or divide by 1
let w = Number(y); // invoke the Number function
```

Control Flow

We'll go through a series of basic control flow structures before looking into objects. Note that in JavaScript, we have the following logical operators:

|| for or
&& for and
! for logical negation.

Control Flow: if, else I

The syntax for **if** and **else** is the same as for C++:

```
if(statement) {
   // what to do
}

if(statement) {
   // what to do here
}
else {
   // what to do here
}

if(statement1) {
   // what to do here
}
else if(statement2) {
```

Control Flow: if, else II

```
   // what to do here
} else {
   // what to do here
}
```

As in C++, grammatically, for if/else and loops, the braces are not required if only a single statement follows, *but they should always be included for code robustness*.

Every nontrivial branch of control flow should have its own documentation.

Control Flow: if, else

Here is a simple example of using **if**/**else**:

```
let x = 277;

if( (x % 2) === 0) { // no remainder when divided by 2
  console.log("even number");
} else { // then has a remainder and is odd
  console.log("odd numer");
}
```

Control Flow: if, else

Beware coercsions!

```
let x = "0";

if( x ) { // if x is true...
  console.log("gets printed!");
}
```

Data Types: Objects Behaviour

In JavaScript, anything which is not a primitive type is an object. As a general overview of object behaviour: an object behaves like a mixture between a C++ pointer and C++ reference.

During assignment, initialization, or passing objects to functions, objects behave like pointers being copied: the assigned-from and assigned-to variables, for example, will be *the same object*.

When objects are being used, such as accessing its value, accessing its properties or member functions, *objects behave like references*: there is no need to “dereference a pointer”

Remark: while the internal implementation could vary from engine to engine, it probably isn’t wrong to think that JavaScript stores objects as pointers to their underlying data and primitive types as their values. Then JavaScript passes these pointers and primitive types around by value everywhere.

Data Types: Objects - Arrays

An object is any data that is not a primitive type. We first consider an **Array**.

In JavaScript, arrays are specified with **[...]** square brackets. They can then contain a (possibly empty) comma separated list of values *of any type!!!*.

Some useful methods and properties of an array include:

- Subscript operator [], indexed from 0, returning the variable at the given index
- **concat**: joins two or more arrays and returns a copy of the joined arrays
- **pop**: removes the last element of the array and returns it
- **push**: adds one or more elements to the array and returns the new length
- **length**: a number corresponding to the array length

Data Types: Objects - Arrays

```
let a = [2,4,6];
let b = [false,'hello',26];
let c = [];
let d = [a,111.111]; // okay: d is an array

d[0].pop(); // now a is [2,4], and d is [ [2,4], 111.111]
```

Warning: in this context, **d** is storing a reference to **a**!

Data Types: Objects - Arrays

```
a = [1,2,false];
b = ['cat','dog','fish'];
c = [];
a[2] = true; // now a is [1,2,true]
d = a.concat(b,c); // d is [1,2,true,'cat','dog','fish'];
c.push('hi','bye'); // returns 2, the new size, and c is ['hi','bye']
a.pop(); // returns true and now a is [1,2]
a.length; // 2, note this is not a member function! no parentheses
```

Remarks: functions such as **concat** and **push** can take more than one argument! And **length** is a property of the array so it does not get invoked as a function.

Data Types: Objects - Arrays

The **concat** function only makes a **shallow copy**!

```
let u = [1,2,3];
let v = [4,5,6];
let w = [u,v,111];
let x = u.concat(w); // x === [ 1, 2, 3, ref to u, ref to v, 111]
x[4].push(7);
v[v.length-1] === 7; // true
```

Control Flow: for

The **for** loop is nearly identical to C++. It follows:

```
for(initialization statement(s); condition; steps after body) {
   // ...
}
```

```
let arr = ["hello", " world"];
let msg = "";
for (let i = 0; i < arr.length; ++i) { // go through each word
   // and add it to the HTML
   msg += arr[i];
}

console.log(msg); // logs "hello world"
```

Control Flow: while and do

While and do also behave the same as C++, with format

```
while(condition) {
   // stuff to do
}

do {
   // stuff
} while(condition);
```

Control Flow: break and continue

There are also **break** statements to break out of the immediately enclosing loop and **continue** statements to skip the remaining steps of a loop iteration before moving to the next iteration.

Control Flow: switch

JavaScript has switch statements, too. The syntax is similar to C++:

```
switch(value){
   case first_value:
      // do stuff
      break;
   case second_value:
      // do stuff
      break;
   default:
      // do stuff
}
```

Switch statements use strict equality **===** to test their cases. Unlike C++, the values being tested need not be of integer type.

Warning: beware to include the **break** statements at the end of each non-default case!

Data Types: Objects - Functions

Functions in JavaScript are objects. They can appear as either **function declarations** or **function expressions**.

A **function declaration** uses the **function** keyword to create a function object of a given name:

```
function add(a,b) {
   return a+b;
}
```

The function above takes in two parameters, called **a** and **b**, and returns the two values with + acting between them.

Functions that return values use a **return** statement. For functions that do not return values, the **return;** is optional.

Data Types: Objects - Functions

As in other programming languages, function documentation is important. The format we adopt is illustrated below:

```javascript
/**
This function adds its two inputs.

@param {Number} a the first number.
@param {Number} b the second number.

@return {Number} the sum.
*/
function add(a,b){
  return a+b;
}
```

All function documentations begin with /** and end with */. They begin with a description.
Then for each input, there is a list
@param {ITS DATA TYPE IN BRACES} its_name a_desciption.
Then for the output, if there is one,
@return {ITS DATA TYPE IN BRACES} a_descrption.

Since JavaScript is loosely typed, listing the expected input/output types is even more important!

Data Types: Objects - Functions

JavaScript also suppots default arguments for functions. As with C++, these arguments can be supplied from right to left, and function parameters are assigned left to right.

```
function f(a,b=3) {
   return a+b;
}

function g(a=9,b=3) {
   return a+b;
}

f(1,8); // 9
f(7); // 10
g(4,4); // 8
g(4); // 7
g(); // 12;
```

Data Types: Objects - Functions

Due to the weakly typed nature of the language, functions **cannot be overloaded** like other languages.

In addition to this, each function is, in C++ terms, "variadic", accepting arbitrary numbers of arguments. Implicitly there is an **arguments** variable local to a function that behaves *like an array* storing all the input arguments. It isn’t an actual array.

We can even choose to lump the final arguments of a function into a single array variable.

Data Types: Objects - Functions

For example:

```javascript
function foo(a,b,...c){
  console.log(a); // 3
  console.log(b); // 4
  console.log(c); // ['hello', 'world']
  console.log(arguments.length); // 4
  console.log(arguments[0]); // 3
  console.log(arguments[1]); // 4
  console.log(arguments[2]); // hello
  console.log(arguments[3]); // world
}

foo(3,4,'hello','world');
```

In C++, this would be kind of like:

```cpp
template<typename A, typename B, typename ... Types>
auto foo(A&& a, B&& b, Types&&... c){
  // do stuff
}
```

Data Types: Objects - Functions

A **function expression**, called **anonymous functions** or **lambdas** in other languages, is one that evaluates to a function but is not named. We can choose to store the function in a variable if we want.

```
let add2 = function(a,b) {
   return a+b;
};
```

The right-hand-side of the above code is a function expression. It evaluates to a function that we refer to as **add2**.

In either case, we get behaviour we expect:

```
add(3,4); // 7
add2(3,4) // 7
```

We created a variable that stores a function, similar to if we had written **let foo = 7;** where we create a variable that stores a number.

Data Types: Objects - Functions

With ECMAScript 6, the **arrow notation** was also introduced for functions. Here are two examples:

```
let double = (x) => 2*x;

let sum_consec = (a,b) => {
   let s = 0;
   while(a <= b) { s += (a++); }
   return s;
};

double(3); // 6
sum_sonsec(1,10); // 55
```

Parameters (0 or more) go in parentheses. There is then an **=>**, followed by the return value. If there is just a value, it can be written directly. For more complicated functions with multiple statements, the body is put in braces.

Data Types: Objects - Functions

JavaScript uses a lot of function expressions because functions can be created for "one off" events where a function is needed on the fly. They also arise in **immediately invoked function expressions (IIFEs)**. We'll see these later, but here's a baby example:

```
(function(a,b){ return a+b; })(3,4); // evaluates to 7
```

The expression in the parentheses is a function expression which can receive two inputs, which is fed the values 3 and 4.

Data Types: Objects - Functions

In C++ terms, function expressions are like lambdas. Our add2 function could be thought of as:

```
auto add2 = [](double a, double b)
  ->double{return a+b;};
```

Data Types: Objects - Functions

We can also use anonymous functions to sort arrays. Arrays have a sort function that by default sorts elements *alphabetically!*. It turns numbers to strings and sorts that way.

```
let arr = [2,100,199];
arr.sort(); // [100,199,2] - - clearly not good
```

But we can provide a binary compare function, being negative if the first argument is "less than" the second, 0 if they are "equal", and positive if the first is "greater than" the second. This also allows arbitrary sorting.

Data Types: Objects - Functions

We change our sorting:

```
arr.sort( function(x,y) { return x - y; }); // [2,100,199]
arr.sort( function(x,y) { return y-x; } ); // [199,100,2]
```

In C++, it works the same. If values is any random access container like a std::vector<int>, say, we can have items sorted in decreasing order:

```
std::sort(std::begin(values), std::end(values),
  [](int x, int y)->bool{
    return y>x;
});
```

Data Types: Objects - Functions

Another use of anonymous functions is in the **map** function of arrays. With the **map** function, we can specify a function to be used to transform all elements of an array.

```
let nums = [1,2,3,4];

num = nums.map( (x) => (x**3) ); // now nums is [1,8,27,64]
```

Each value is cubed. The **map** function returns a transformed array but does not modify the original, hence the assignment involved.

Data Types: Objects - Functions

We use the term **execution context** to describe the environment in which code is being run. This could be the **global execution context**, which is where everything lives by default, or it could be a **function execution context**, which is the same thing as the global, but restricted to the body of the function.

Data Types: Objects - General

In general an object in JavaScript uses the brace syntax. Members (variables/functions) and their values appear in **name: value** pairs, separated by commas.

```
let x = { }; // x is an empty object

let pic40a = { // has a time (Number), room (String), and function
   start_hour: 9,
   room: ’MS 2000’,
   chant: function() { console.log("Go Bruins!"); }
};
```

console.log is a function that logs things. The outputs can be viewed by looking for the console output with web browser development tools. The outputs are not rendered to the webpage.

Data Types: Objects - General

We can now use **pic40a** as a class object.

```
let end_hour = pic40a.start_hour + 1;
pic40a.chant(); // logs "Go Bruins!"
```

Members are accessed with the **.** syntax.

In C++ terms, pic40a is an instance of a class with all public members. It has member variables start_hour and room, of type int and std::string, respectively. It has a member function chant that can display the message.

Data Types: Objects - General

Remember how **x** was an empty object? Well, in JavaScript, more members can be added.

```
x.i = 7; // x now has a member i of value 7
x.foo = function() { }; /* x now has a member foo, a function, that does
   nothing */

x["p"] = false; // x now has a member p of value false
```

Members can be added through the **.** operator or through the subscript operator with a string argument. Members can even be accessed through the subscript.

The most common convention is to use **.** - and that's probably good advice to avoid confusion.

Data Types: Objects - General

Functions are just objects that can be called with **()**. A function internally stores a reference to the instructions to follow when called. But it can store more...

```
function f(name) {
   console.log('hello ' + name);
}

f.a = 7;
```

The variable **f** can be called as in **f("Alice")** but also has a member called **a** with value 7... Yes, this is a little weird, but this comes down to all non-primitives being objects.

Data Types: Objects - General

The **typeof** function returns the type of its argument: for primitive types (except null) it returns their type, for all other inputs it returns "object".

```
typeof(true); // "boolean"
typeof(888); // "number"

let x;
typeof(x); // "undefined"

x = null;
typeof(x); // "object"

typeof({}); // "object"
```

The peculairity with **null** relates historically to how the language was developed and the need to preserve backwards compatability for programs that relied on this quirk before it could be remedied.

Data Types: Comparison

Here we look at the two types of equality and inequality comparisons.

In general we can say **A == B** if, after a possible coercion, **A** and **B** are equal; and **A != B** if, after a possible coercion, **A** and **B** are not equal. This gray area, which can get a lot more complicated, is often termed **truthy** and **falsy**.

In JavaScript there are also **===** and **!==**. We write **A === B** if **A** and **B** are equal and of the same type; and **A !== B** if either **A** and **B** differ in type or differ in value but have the same type.

Remark: it is almost always ill-advised to use **==** or **!=**. Just use **===** and **!==**.

Data Types: Comparison

JavaScript supports the usual comparison operators as C++: **<** (less than), **<=** (less than or equal), **>** (greater than), and **>=** (greater than or equal).

When comparing a string and a number, the string is converted to a number.

```
3 < "10"; // true
"500" <= 500; // true
```

When comparing two strings, the comparison is done lexicographically, i.e. like a dictionary, as in C++:

```
"apple" >= "coffee"; // false
"zebra" < "goldfish"; // false
```

Data Types: Comparison

With the **==** operator, the rules are a lot more complicated. Generally, the JS engine will try to convert one or both of the arguments to a number.

null and **undefined** are equal to each other and themselves under **==** but otherwise are **!=** to other types.

```
undefined == null; // true
"1" != true; // becomes 1 != 1, which is false
null == false; // false
```

Just use the **===** and **!==** and you’ll be safe.

Data Types: Comparison

In JavaScript, **==** and **===** for objects test for “deep equality,”, i.e., tests if two objects store the same pointer value.

```
{} == {}; // false: both empty object but different addresses

let a = {}, b = {};
a == b; // false

let c = a;
a == c; // true
```

Variables

In JavaScript, a variable can be defined with the **var**, **let**, **const** keywords - even none (should be avoided).

In modern JS, only **let** and **const** should be used. In a nutshell:

- **var**: defines a variable either globally if it is defined globally or throughout an entire **function execution context** if defined within a function. **var** gives no warning if a variable is redefined with the **var** keyword.
- **let**: defines a variable only within the given scope, i.e. set of braces { }, or globally if it is defined in the global execution context. An error is generated if a variable defined again after its first **let**.
- **const**: does the same thing as **let** but also **protects against assignment** and for primitive types this includes any modifications such as **++**, ***=**, etc.
- defining a variable without a keyword could either define a new global variable or modify a global variable.

Variables

Recall once again how objects should really be thought of as pointers to underlying data, until we try to access their properties or member functions. For an object, **const** is describing the pointer, not the data! Making an object's data immutable requires more work.

```
const obj = { foo: 'bar'};
obj.foo = 19; // okay: the pointer did not change!

obj = { }; // ERROR: assigning to const
```

In C++ terms, this would be kind of like:

```
const auto p = std::make_shared<Object>();
p->mutator(); // okay

p = q; // ERROR: pointer is const
```

Variables

Remark: when used in global contexts, **let** and **const** do not add properties to the global **window** object but **var** does.

```
// in the global execution context

x = 5; // now window.x === 5

var y = 6; // now window.y === 6

const z = 43; // there is z === 43 but no window.z
```

Variables

Remark: in JavaScript many variables are **global**, meaning they can be accessed everywhere in the program, but there are various means of **encapsulation** and avoiding naming conflicts by wrapping variables up inside of objects or using **IIFE**s.

Valid variable names begin with either a letter or $ or _, then contain letters, digits, $'s, or _'s. Variable names are **case sensitive**.

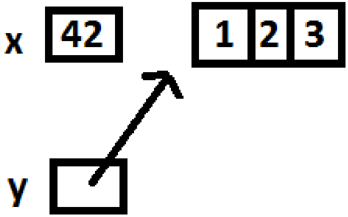
Variables

Any variable that is a primitive type can be thought of as storing its corresponding value.

```
let x = 42;
```

On the other hand, a variable that is an object *only stores a pointer* to the object: the value is stored elsewhere in memory.

```
let y = [1,2,3]; // an Array - all Arrays are objects
```



Passing by Value vs Reference

The preceding facts have serious ramifications. When variables are reassigned, it is only the data stored within the variable itself, not necessarily its value, that gets reassigned.

```
let x = 42;
let y = x; // y is also 42
x = 27; // y is still 42

let foo = [1,2,3,4]; // foo references/points to [1,2,3,4]
let bar = foo; // so does bar
bar.pop(); // foo and bar BOTH reference/point to [1,2,3]
```

Above, **foo** and **bar** both wind up referencing the same block of memory with the array values. By changing either through a member function, the array values in memory are modified, so **foo** and **bar** are changed!

Passing by Value vs Reference

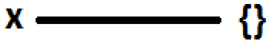
But during assignment, new objects could be created...

```
let x = { };
let y = x; // both reference the empty object

x = { snack: "walnuts and dates" };
```

In the case above, **y** still references the empty object in the end but **x** references the object with **snack**.

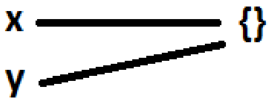
Passing by Value vs Reference

```
let x = {};
```



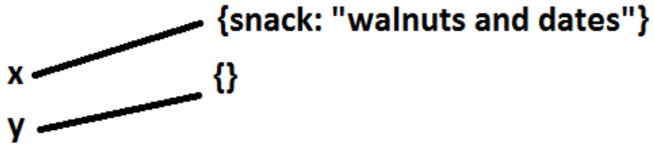
Passing by Value vs Reference

```
let y = x;
```



Passing by Value vs Reference

```
x = { snack: "walnuts and dates" };
```



Passing by Value vs Reference

Functions behave no differently in this regard.

```
function foo(param1, param2) {
   // do stuff
   // maybe return stuff
}

foo(x,y);
```

is effectively the same thing as:

```
let param1=x, param2=y;
// do stuff
// maybe get a value
```

Document Object Model

Before focusing more upon the JavaScript language, we can look at a few applications of the topics so far. We will study the **Document Object Model (DOM)**. This is the organizational scheme used by web browsers to place and style elements and their content. Most of JavaScript interactivity pertains to modifying the DOM.

In the DOM, everything is represented as a tree with nodes. **document** refers to the overall document storing everything.

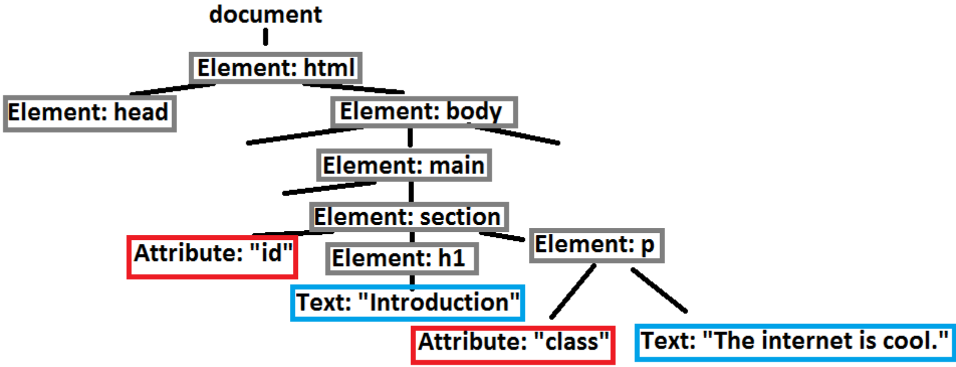
A **node** can be an **element node** (such as p, section, b, etc.), a **text node** (the actual text content), a **comment node**, or an **attribute node** (such as an href, src, etc.).

Danger: any white space is considered a text node!

Document Object Model

Consider

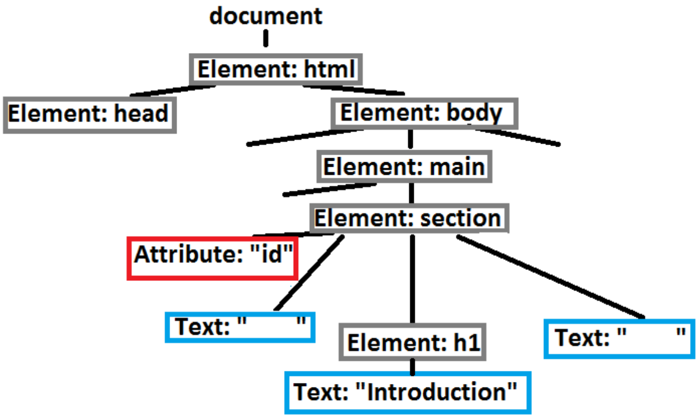
```
<section id="intro"><h1>Introduction</h1><p class="first">The internet is cool.</p></section
>
```



Document Object Model

We added some white space...

```
<section id="intro">
  <h1>Introduction</h1>
</section>
```



Document Object Model

When JavaScript runs, there is a global variable, **document**, and we can retrieve elements of the DOM by invoking member functions upon it.

The **getElementById** method retrieves the element of the given ID.

The **getElementsByName** method retrieves all elements with a given **name** attribute as an array.

The **getElementsByTagName** method retrieves all the elements of a given type and returns them as an array. An argument of "*" can be used to retrieve all elements.

The **getElementsByClassName** retrieves all elements of a given class value and returns them as an array.

Document Object Model

innerHTML is a member of each element object and the HTML within an element can be (re)set by setting this member.

An element’s attribute can be (re)set with the **setAttribute** member function taking the attribute and value as arguments.

We can also retrieve the value of an attribute with the **getAttribute** function, accepting an argument of what attribute value to find.

Document Object Model

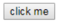
Warning: writing **document.getElementById("some_input").getAttribute("value")** will only give the value originally placed in the HTML.

To get the "live" value, instead write **document.getElementById("some_input").value**.

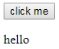
Document Object Model

We now look at how we can change the page display in response to a user clicking. Here is our desired behaviour:

Before clicking the button:



After clicking the button:



Click Events

When events such as clicks take place, we often wish to set **callback functions** — functions to invoke when the event happens.

Responding to click events is extremely common in JavaScript. There are a few ways to accomplish this task. We'll look at a few, starting with the least desirable and moving to the most desirable and professional.

Inline: A button can be given a function to invoke as an **onclick** attribute directly in HTML. This can be difficult to maintain and should be avoided.

Setting onclick with JS: With JavaScript, we can add an **onclick** property with a suitable function call. This approach, however, can lead to glitches if multiple processes are to be added in response to a click. Only one **onclick** value can be given per object.

We can delete **onclick** events by setting the **onclick** property to **null.**

Click Events

Adding an event listener with JS: the **addEventListener** function can listen for events. Multiple event listeners for the same type of event can be added.

The syntax to add a click event calling **f** to **e** is:

```
e.addEventListener( "click", f );
```

To remove that event listener, we need to specify the event type and the same callback with **removeEventListener**:

```
e.removeEventListener( "click", f );
```

Click Events

This is the **inline** approach:

inline.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Inline</title>
  <script src="demo_inline.js" defer></script>
</head>
<body>
  <main>
    <!-- onclick is listed here -->
    <input type="button" value = "click me" onclick="change();" />
    <p id="first">
    </p>
  </main>
</body>
</html>
```

demo_inline.js

```javascript
function change(){
  document.getElementById("first").innerHTML = "hello";
}
```

Click Events

This is the **onclick** approach:

onclick.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS onclick</title>
  <script src="demo_onclick.js" defer></script>
</head>
<body>
  <main>
    <!-- no onclick here -->
    <input type="button" value = "click me" id = "theButton" />
    <p id="first">
    </p>
  </main>
</body>
</html>
```

demo_onclick.js

```javascript
function change(){
  document.getElementById("first").innerHTML = "hello";
}

document..getElementById("theButton").onclick = change;
```

Click Events

This is the **listener** approach:

listener.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Listener</title>
  <script src="demo_listener.js" defer></script>
</head>
<body>
  <main>
    <!-- no onclick here -->
    <input type="button" value = "click me" id = "theButton" />
    <p id="first">
    </p>
  </main>
</body>
</html>
```

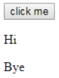
demo_listener.js

```javascript
function change(){
  document.getElementById("first").innerHTML = "hello";
}

document..getElementById("theButton").addEventListener( "click", change );
```

Document Object Model

Now we consider an example that will change the CSS of a paragraph.

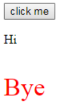
Before clicking the button:



Hi

Bye

After clicking the button:



Hi

Bye

Document Object Model I

index_css.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS CSS</title>
  <link rel="stylesheet" href="style.css" />
  <script src="demo_css.js" defer></script>
</head>
<body>
  <main>
    <input type="button" value = "click me" id= "theButton" />
    <p>
      Hi
    </p>
    <p>
      Bye
    </p>
  </main>
</body>
</html>
```

style.css

```css
.special{
  font-size: 2em;
  color: red;
}
```

Document Object Model II

demo_css.js

```javascript
(function(){
  const b = document.getElementById("theButton"); // button
  const p = document.getElementsByTagName("p")[1]; // second paragraph

  b.addEventListener( "click", () => {
    p.setAttribute("class", "special");
  });
}
)();
```

Document Object Model

The **getElementsByTagName** returns an array of all the **p** elements. We then set the **class** attribute of the second paragraph to **special**, which the CSS styles.

The event listener is added through an IIFE and we used the arrow notation for the callback.

Remark: there is some magic going in with how, for example, how the identifier **p** will no longer exist after the IIFE is invoked but the event will still work. This is called a **closure** — more later.

Document Object Model

We can also set the CSS directly. Given an element node **obj**:

obj.style.property = "value";

where **property** is the CSS property being set and **value** is the new value.

In general CSS properties that used to have - between words become camel-cased: **font-family** becomes **fontFamily**, for example.

Document Object Model I

Consider:

index_css2.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Demos</title>
  <script src="demo_css2.js" defer></script>
</head>
<body>
  <main>
    <input type="button" value = "click me" id = "theButton" />
    <p>
      Hi
    </p>
    <p>
      Bye
    </p>
  </main>
</body>
</html>
```

Document Object Model II

demo_css2.js

```
(function(){
  const p = document.getElementsByTagName("p")[1]; // second paragraph
  const b = document.getElementById("theButton"); // button

  p.addEventListener( "click", () => {
    p.style.fontSize = "2em";
    p.style.color = "red";
  });
})();
```

When clicked, the font size of the second paragraph will double and the font will be red.

Document Object Model

Each node also stores references to some other nodes as members:

- **parentNode** (its parent)
- **children** (all element children as array)
- **firstChild** (an element's first child)
- **firstElementChild** (an elements first element child)
- **lastChild** (an element's last child)
- **lastElementChild** (an elements last element child)
- **nextSibling** (next node that is not a descendent)
- **nextElementSibling** (next element that is not a descendent)
- **previousSibling** (previous node that is not a parent)
- **previousElementSibling** (previous element that is not a parent)

In the preceding example:

```
// references the main
document.getElementsByTagName("p")[0].parentNode;
```

Document Object Model

If **e** is a reference to a radio button or checkbox, **e.checked** is a boolean value. It is true/false when the button has/hasn’t been selected.

The selection can also be set by assigning to **e.checked**.

Document Object Model I

We can build a simple countdown printout with HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Demos</title>
  <script src="counter.js" defer></script>
</head>
<body>
  <main>
    <p id="counter">
    </p>
  </main>
</body>
</html>
```

and **counter.js**

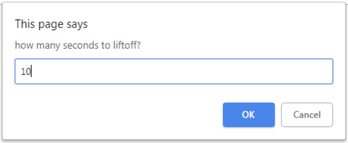
```javascript
/**
This function prints a countdown message for how many seconds until liftoff

@param {Number} time the number of seconds
*/
function run(time){
  let line = "";
  while(time>0){ // until the time is 0
    if(time !== 1){ // if not 1 second, seconds is plural
      line = (time--) + " seconds<br/>";
    }
    else{ // if 1 second, use singular
      line = (time--) + " second<br/>";
```

Document Object Model II

```
    }
    document.getElementById("counter").innerHTML += line;
  }
    document.getElementById("counter").innerHTML += "liftoff!";
}

let input = prompt("how many seconds to liftoff?", "");
run(input*1); // so Number
```

we have a simple page

Before:

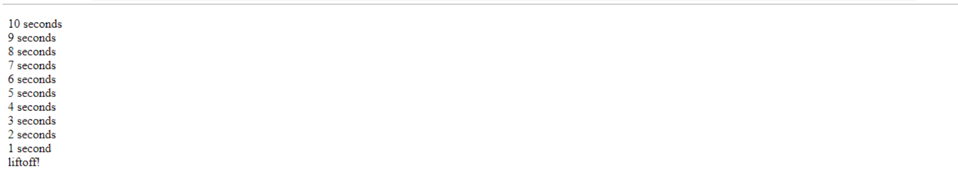

This page says

how many seconds to liftoff?

10

Document Object Model III

After:



```
10 seconds
9 seconds
8 seconds
7 seconds
6 seconds
5 seconds
4 seconds
3 seconds
2 seconds
1 second
liftoff!
```

Query Strings to Prevent Caching

To prevent a web browser from caching files, we can append a **query string** to the end of the file we are including. Then we can update the number each time we update the JavaScript file, just to force the browser to download the file again. This amounts to

```
<script src="demos.js?v=1" defer></script>
<script src="demos.js?v=2" defer></script>
<script src="demos.js?v=3" defer></script>
```
etc.

The **?v=1** is an example of a query string. In this context, it has no effect on the file being downloaded but the browser “thinks” it is different. A query string begins with a **?** and includes **name=value** pairs.

JavaScript Hoisting

Before a JavaScript program begins to run, the JavaScript engine does something called **hoisting.** It secretly moves declarations made with **var** and functions to the top of the execution context.

```
var x = 11;

print(x);
print(y);

var y = 22;

print(y);

function print(a){
  console.log(a);
}
```

The logged output is:

```
11
undefined
22
```

JavaScript Initialization (Hoisting)

Within each execution context, during the **hoisting** phase, all variables declared with **var** are collected and given the value **undefined.** All **function declarations** are collected and their instructions are moved to the code memory, which is accessible throughout the entire execution context. Then the program runs.

So, in the preceding example:

1. The engine sets **x** and **y** to **undefined.** *
2. The engine creates instructions to run when **print** is called and **print** references those instructions. *
3. **x** is set to **11**.
4. **print** can print that value.
5. **print** has to print **y**, but that value is still **undefined**.
6. **y** is set to **22**.
7. **print** can print y as **22** now.

*: it might not be specified which of (1) or (2) happens first, but they both happen before the other steps.

JavaScript Initialization (Hoisting)

The *identical concept* applies to all execution contexts. If we ran the same code inside an IIFE, we would get the same output.

```
(function(){
  var x = 11;

  print(x);
  print(y);

  var y = 22;

  print(y);

  function print(a){
    console.log(a);
  }
}
)();
```

JavaScript Initialization (Hoisting)

This is a reason why **let** and **const** are better. You can actually get errors, not strange bugs, when you use such variables before they have been initialized.

```
(function(){
  let x = 11;

  print(x);
  print(y);

  let y = 22;

  print(y);

  function print(a){
    console.log(a);
  }
}
)();
```

We get an error because we used **y** before it was initialized. **y** did not get hoisted.

Garbage Collection

In lower-level languages, memory management is often the responsibility of the programmer. In languages like C++, stack memory is managed with a last-in-first-out policy and heap memory management requires the programmer, either directly (by using **new** and **delete expressions** or lower level operations) or indirectly (through using containers of the Standard Library that manage dynamic memory), ensure no memory is being wasted, holding up resources, but not being used.

JavaScript and other higher level langauges like Python and Java have what is called **garbage collection.** This process happens outside of the programmer's control.

Garbage Collection

Periodically, the JavaScript engine runs a garbage collection cycle. It looks for variables and objects that cannot be reached from the global **window** object. Any variables/objects that cannot be reached are destroyed and the memory is freed up.

We consider the code snippet below with 3 places where garbage collection could take place.

```
let person = {
  name: "Peter",
  fav_nums: [17,22];
};

// place 1

(function(){
  let str = person.name;
  let nums = person.fav_nums;
  // place 2
})();

person.fav_nums = [1,2,3];
// place 3
```

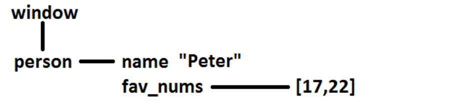
Garbage Collection

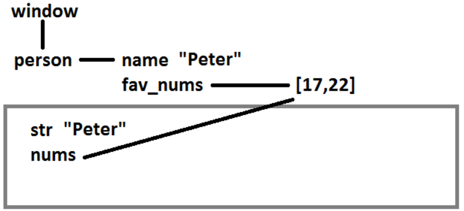
We have the three panels diagramming the memory structure at different points. At first, **person** is the only object. It has a string and array as members. Recall that arrays are objects, which reference their values.

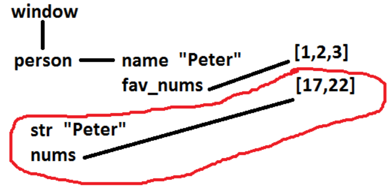
Within the function, we create the new variables **str** and **nums** but since the function is in-use, we don't destroy them.

After the last line of code, **str** and **nums** cannot be reached. And the values originally referenced by **person.fav_nums** cannot be reached. So garbage collection could free them, circled in red.

Garbage Collection







noscript

JavaScript is wonderful to use but some browsers may not support it or else they may not allow its use. We can use the **noscript** tag to display a message in that case.

```html
<!-- DOCTYPE, html, head stuff -->
<body>
<noscript>
Your browser does not support JavaScript. Parts of this page may not work for you as
     intended.
</noscript>
<!-- other stuff -->
</body>
</html>
```

Closures

A **closure** is a way for a function object to preserve its environment, even when those environment values cannot be reached by other parts of a program.

Below we write a function called **validator** that **returns a function** that checks if a username and password are correct.

```
function register(correct_username, correct_password){
  return function(user, pass){
    return( user === correct_username && pass === correct_password);
  };
}

const joe = register('joejoe', '111aaa');
const jane = register('jane_login', 'zyx');

joe('joejoe', '111aaa'); // true
jane('joejoe', '111aaa'); // false
jane('jane_login', 'xyz'); // false
```

Closures

There are some conceptual hurdles here. First of all, remember that functions are objects. It's perfectly legit to return a function object from **validator**. That function object returns a Boolean value.

More fundamentally, though, notice that **correct_username** and **correct_password** are local variables to **validator**. As identifiers for objects, they cease to exist after the function call. Yet the anonymous function returned is supposed to use both of these values for its logic. So how does that work?...

The answer is **closures**. Although **correct_username** and **correct_password** cannot be referenced outside of **validator**, the function returned by **validator** keeps a reference to the spot in memory where those values are stored. So everything just works.

this

In JavaScript, **this** is a reserved keyword that can mean different things, depending where it appears. It always references an object, possibly the global **window** object, possibly another object.

When it appears globally, it references **window**.

```
this.foo = "bar"; // adds foo property to window, value = bar
console.log(this); // same as console.log(window)

var x; // adds x property to window
let y; // does not add y property to window
```

When a variable is declared globally with **var**, it is added to the **window** object. This does not happen with **let**.

this

When **this** is used within an **arrow function expression**, it means the **this** of its enclosing scope. Note that being inside the body of an object *does not* introduce a new scope.

When **this** is used within a traditional **function expression** or **function declaration**, it means *either*:

- the **object of which it is a member** when that function is itself a member function/constructor or
- **window** in all other contexts.

We will look at constructors later.

this

```
var x = 10; // window.x is 10

let obj={
  x: 42,
  p: function() { return this.x; }, // this is obj
  q: function() {
    return () => this.x; // this is the this of the member function, i.e., obj
  },
  r: function(){
    // this is window: anonymous function here not member function
    return function() { return this.x;}
  },
  // this is window: obj does not bring in new scope
  s: () => { return this.x; }
}

function t() { return this.x;}

console.log(obj.p());
console.log(obj.q()());
console.log(obj.r()());
console.log(obj.s());
console.log(t());
```

this

The code outputs:

```
42
42
10
10
10
```

Since **obj.p** is a member function, **this.x** within it is **obj.x.**

The arrow function expression within **q** will use the **this** of the member function to which it belongs, i.e. **obj**.

The function expression *within* **r** is not a member function so **this.x** is **window.x**.

The member function **s** uses arrow notation and **this** is **window.**

Within the free function, **t**, **this** is **window**.

this

Since objects can be thought of as pointers, a workaround many JS developers use when they want to reference member variables within a non-member function is to make a reference to **this** first and then use the reference.

```
let obj={
  x: 42,
  r: function(){
    let self = this;
    return function() { return self.x;}
  }
}

console.log(obj.r()());
```

The code above outputs:

```
42
```

this

When a callback is added through **addEventListener**, the callback is added as a “member function” so the same semantics as the previous example apply.

```
window.x= 1;

function fun(){
  console.log(this.x);
}

// add x to a button
document.getElementById("button").x = 10;

document.getElementById("button").addEventListener("click", fun);
document.getElementById("button").addEventListener("click", ()=>{console.log(this.x);});
document.getElementById("button").addEventListener("click", function(){console.log(this.x)
      ;});

// when clicked:
// 10
// 1
// 10
```

fun1 and the function expression act as member functions and **this** will be the button object. The arrow function behaves like **obj.s** where **this** is **window**.

The Scope Chain

The global environment, the body of each function, and the contents within braces {} form what are called **lexical environments**. When a variable name is used, the local environment is searched first for that variable; if it is not found, JavaScript looks for the variable in the enclosing environment; if not found there, it goes to the enclosing environment, etc., etc. etc. If the variable is not found at all then we have an error.

The Scope Chain

```
let a = 7;
let b = 70;
let c = 700;

function foo(){
  let b = 8;
  let c = 80;
  function bar(){
    let c = 9;
    console.log(c); // finds the c in bar
    console.log(b); // finds the b in foo
    console.log(a); // finds the global a
    console.log(x); // finds the hoisted x
    console.log(w); // will not find w
  }
  if(true){ // execute body since true is true :)
    let w = 10; // only local with let and no hoisting!
    var x = 11; // hoisting
  }
  bar();
}

foo(); // prints 9, 8, 7, 11, then gives error
```

let and const vs var in Loops

The variables **let** and **const** behave very differently in control flow than **var**.

When **var** is used, there is the usual hoisting so the loop variables created with **var** gain global/function execution context and all closures based on this hoisted value.

When **let** and **const** are used, those variables are only local to the lexical scope, i.e., set of braces { }. In loops with **let** and **var**, for each iteration, all closures are based on a **copy** of the loop variable.

let and const vs var in Loops I

For example, the code:

```
let fun = null;

for(var i=0; i < 2; ++i) {
   fun = () => i;
}
```

let and const vs var in Loops II

amounts to:

```
var i; // i is hoisted to entire execution context

let fun = null;
{
   i = 0;
   fun = () => i;
}
++i; // now i is 1
{
   fun = () => i;
}
++i; // now i is 2
```

So **fun** returns **2**, the value of **i**!

let and const vs var in Loops I

The code:

```
let fun = null;

for(let i=0; i < 2; ++i) {
    fun = () => i;
}
```

let and const vs var in Loops II

amounts to:

```
let fun = null;
{
    let i = 0; // only copies of i are used within the loop body!
    {
        let icopy = i;
        fun = () => icopy;
    }
    ++i; // i is now 1
    {
        let icopy = i;
        fun = () => icopy;
    }
    ++i; // i is now 2
} // i ceases to exist outside of loop scope
```

let and const vs var in Loops III

So **fun** returns **1** based on the closure of **icopy**.

Syntax

The JavaScript syntax can be quite forgiving.

In JavaScript, a semicolon is not required to end a statement. Because the language is interpreted, when the JavaScript engine encounters a new line in the code, it may/may not insert a semicolon, depending on whether it thinks one belongs.

When, as the program is parsed from left to right, a token (called the offending token) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true: The offending token is separated from the previous token by at least one LineTerminator [...,] The offending token is } [... or] The previous token is) and the inserted semicolon would then be parsed as the terminating semicolon of a do-while statement ~ECMAScript Language Specification

Syntax

```
function get_best_number_ever(){
  return
  22;
}

console.log( get_best_number_ever() ); // undefined
```

With this code, a semicolon is inserted after the return statement.

Remark: just as a programming practice, it is best to include the semicolons to make statement intents clearer. Seldom does the semicolon insertion actually lead to a bug.

More DOM Manipulations I

JavaScript allows us to create new elements and text nodes with **createElement** and **createTextNode**.

```
const new_p = document.createElement("p"); // make a new p element

// new text node
const p_words = document.createTextNode("paragraph text");
```

The **appendChild** function adds a new child node to a given node.

```
const some_article = document.getElementById("the_article");
some_article.appendChild(new_p);
new_p.appendChild(p_words);
```

More DOM Manipulations I

There are also **insertBefore** to insert one node (first argument) before another (second argument) and **replaceChild** to overwrite the node specified as second argument by the node specified by the first argument.

```
const newer_p = document.createElement("p");
const more_words = document.createTextNode("words that come
before");

some_article.insertBefore(newer_p, new_p);
newer_p.appendChild(more_words);
```

More DOM Manipulation I

Here’s an example using this new syntax.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Demos</title>
  <script src="append.js" defer></script>
</head>
<body>
  <main>
    <input type="button" value = "click me" id="theButton" />
    <div></div>
  </main>
</body>
</html>
```

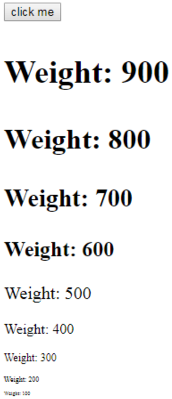
and **append.js**

```javascript
function change(){
  // get the only one
  const div_node = document.getElementsByTagName("div")[0];

  const scale = 400;

  // i represents font weights and scaled font sizes
  for(let i=100; i <= 900; i+=100){ // let it range from 100 to 900
    const new_p = document.createElement("p");
    const label = document.createTextNode("Weight: " + i);

    new_p.appendChild(label);
    new_p.style.fontWeight = i;
```

More DOM Manipulation II

```
      new_p.style.fontSize = (i/scale) + "em";

      const first_child = div_node.firstChild;

      if(first_child === null){ // if div has no children
        div_node.appendChild(new_p); // add the child
      }
      else{ // if it has children
        div_node.insertBefore(new_p, first_child); // make this new first
      }
    }
  }

document.getElementById("theButton").addEventListener("click", change);
```

More DOM Manipulation III



```
Weight: 900
Weight: 800
Weight: 700
Weight: 600
Weight: 500
Weight: 400
Weight: 300
Weight: 200
Weight: 100
```

Control Flow: for ... of

For iterable containers such as arrays, strings, etc., we can iterate through the containers with the **for ... of** loop.

```
let arr = [2,'hello',3.14];
for (const e of arr) { // for every element e of arr
   console.log(e);
}
```

We go through each element of the array, call it **e** (or some other name) temporarily, and log it.

In C++, it is the same thing as the range-for loops:

```
// suppose container is a set/vector/array/etc.
for( const auto& e : container) {
  std::cout << e << '\n';
}
```

Control Flow: for ... in

Warning: there is also a **for... in** loop in JavaScript, but it is not suitable for iterating over values of a container. Do not confuse the two or strange things can happen.

for ... in goes through all enumerables (generally the properties) of an object; **for ... of** goes through iterables.

Control Flow: for ... in

```
const arr = ['foo', 'bar'];
arr.x = 9;

// prints iterables
for(const i of arr) {
    console.log(i); // prints 'foo'and 'bar'
}

// prints the enumerables
for(const i in arr) {
    console.log(i + "--"+ arr[i] ); // prints '0--foo', '1--bar', 'x--9'
}
```

new

In JavaScript, the **new** keyword can be used in conjunction with a function call. In such cases, we are invoking the function as a constructor to create a new object.

Unless **new** is used, the function may not return an object.

Sets and Maps

JavaScript has **Set**s, **Map**s, and other structures. They are similar conceptually to those in C++, having unique values and keys, respectively, but these containers are not sorted by value/key: they are sorted by the insertion order.

They are worth looking up if needed. Here’s just a preview of some of their syntax/functions:

```
// start empty (or put array in parentheses to give values)
const s = new Set();

s.add(3);
s.add('hello'); // stores two elements now
s.has(7); // false, not in the set
s.delete(3); // now 3 is gone
s.size; // 1
```

Sets and Maps

Remark: it is a little different for containers with key-value pairs. The **for ... of** would take the form:

for(let [key_name, value_name] of the_map) { ... }

Sets and Maps

```
// m has ’a’==>9, ’b’==>11 as key==>value pairs
const m = new Map();
m.set(’a’,9); // add key of ’a’ with value 9
m.set(’b’,11);

m.has(’c’); // false: no ’c’ key
m.get(’a’); // 9, get the value
m.delete(’a’); // ’a’==9 gone
m.size; // 1

for( const [key, val] of m ) { // print everything
   console.log( key + ': ' + val );
}
```

The loop prints:

```
b:  11
```

Sets and Maps

Remark: for a **map**, we do not use the square brackets operator to add elements. That will add a property (since maps are objects) but it will not be added to the map values that can be searched through, etc.

Math

Math is a global object with many useful member functions. The following functions are found within **Math**:

- **abs** (absolute value)
- **sin, cos, tan** (sine, cosine, tangent)
- **exp, log** (natural exponential, natural logarithm)
- **pow** (raises first argument to power of second argument)
- **random** (takes no arguments, returns value on $[0, 1)$ - 0 included but 1 not)
- **round** (rounds to nearest integer)
- **ceil, floor** (integer ceiling and floor functions)

For example:

```
let x = Math.random(); // will be between 0 and 1
let die_roll = 1 + Math.floor(6*x); // will be 1,2,3,4,5, or 6
```

Functions as Constructors

In programming, a **constructor** is a function that makes an object.

When a function is called with the **new** keyword, it becomes a constructor for an object. An empty object is created and passed to the function with the special name of **this**.

```
function Person(name) {
   this.name = name;
}

const joe = new Person("Joe"); // joe.name === "Joe"
```

Within the function, because **new** was used, there is an object called **this** that starts empty and has a **name** property added. This object is returned.

Date

In JavaScript, there are **Date** objects that can be constructed by invoking the **Date** constructor.

Internally, JavaScript stores dates in milliseconds since January 1, 1970, 00:00. Here are a few ways to construct dates:

```
let now = new Date(); // no arguments, get current date+time
let start = new Date(0); // one argument = number of milliseconds
let new_year = new Date(2022,0,1);
```

By supplying more than one argument, we specify the **year**, **month**, **day**, **hour**, **minute**, **second**, and number of extra **milliseconds**, with defaulted values of 0 supplied.

Months are indexed mod 12 from 0 = January to 11 = December!

Date

Dates can also be constructed with an International Organization for Standardization (oddly abbreviated ISO) date: "yyyy-mm-dd":

There are also getting and setting methods:

- **setDate** sets day value
- **setMonth** sets month value (0-11 for Jan-Dec!)
- **setFullYear** sets year value
- **setHours** sets hours
- **setMinutes** sets minutes
- **setSeconds** sets seconds
- **setMilliseconds** sets milliseconds
- **setTime** sets milliseconds since Jan 1, 1970, 00:00

Likewise, there are **getDate**, etc., versions.

Date

```
let solstice = new Date("2022-05-21"); // 05 is June

solstice.setHours(2); // at 2 am
solstice.getFullYear(); // 2022
```

Other Useful Classes

Remark: primitives like string, number, and boolean are not objects. There are, however String, Number, and Boolean wrapper objects. These are classes that wrap around the primitives and give them different functionality.

Most of the time, we should avoid using these wrapper objects because they can make code more error prone and less efficient. But for conceptual completeness, code lines like those below implicitly created objects for us:

```
"hello".length; // makes a String objects from "hello" so it has a length

let a = "12";
let b = Number(a);

let c = b.toString(); // make b into a Number object to call function
```

Timers

With JavaScript, we can set a process to run/be repeated after/every some time in milliseconds. For this, we use **setTimeout** and **setInterval**, respectively.

Both of these functions take arguments of a function object for the process to run and the time interval in milliseconds. More arguments can also be given. They both return an id of the event.

There are also **clearTimeout** and **clearInterval** functions that accept an event id and cancel the process(es).

Timers I

With HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>JS Demos</title>
  <script src="resize.js" defer></script>
</head>
<body>
  <main>
    <p id = "text">
      This text will change in size every so often.
    </p>
    <form>
      <fieldset>
        <label for="sec">Time in seconds: </label>
        <input type="text" id="sec" value="" />
      </fieldset>
      <fieldset>
        <input type="button" value="start" id="start_button" />
        <input type="button" value="stop" id = "stop_button" />
      </fieldset>
    </form>
  </main>
</body>
</html>
```

and **demo.js**

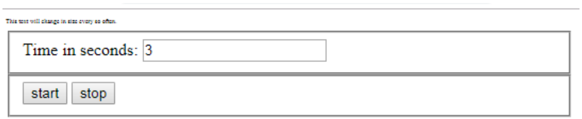
Timers II

```
// params acts as a namespace for properties
const params = {
  event_id: null,
  millisec_per_sec: 1000,
  scale_range: 5
};

function start_changing(){
 if(params.event_id){ // do not double-up on event timers
   clearTimeout(params.event_id);
 }

  const interval = document.getElementById("sec").value * params.millisec_per_sec;

  // does the resizing
  function to_repeat(){
    const scale = Math.random()*params.scale_range;
    document.getElementById("text").style.fontSize = scale + "em";
    call_back();
  }

  function call_back(){
    // call the resizing function
    params.event_id = setTimeout(to_repeat, interval);
  }

  call_back(); // start the process
}

function stop_changing(){
  if(params.event_id){ // make sure event has been set, null becomes false
    clearTimeout(params.event_id);
```

Timers III

```
      params.event_id = null;
    }
  }

document.getElementById("start_button").addEventListener( "click", start_changing);
document.getElementById("stop_button").addEventListener( "click", stop_changing);
```

we have a page where the size of the text changes to a random scaled value from 0-5 with a period in seconds specified by the user.



This text will change in size every so often.

Time in seconds: 3

Timers

Due to the possibility of a function call causing an error or time-intensive processes being called more often than necessary (when they should, ideally, wait for another process), **setTimeout** is considered better practice than **setInterval**.

One can always bring about the desired effect by a function calling itself or another function that calls it using **setTimeout**.

Note how we avoided multiple global variables by encompassing the variables in a single object called **params** that serves as a namespace.

Synchronous vs Asynchronous

The term **synchronous** refers to carrying out one task at a time, in the order received, whereas **asynchronous** refers to carrying out multiple tasks/processes at once.

Fundamentally, JavaScript is synchronous. Each time it processes an event, it loads all the necessary functions onto the call stack (how programs keep track of what to do next) and runs those processes to completion, leaving the call stack empty. Then it can process another event. It happens one at a time!

However, the language has features that give it asynchronous behaviour, too.

Synchronous vs Asynchronous

Functions such as **setTimeout** and **setInterval** are given a **callback** function and a time delay/period. Their callback functions are only invoked when the call stack is empty and when the timer has queued up the callback. This means that the time interval we specifiy is a minimum time to wait for/between callbacks, not the precise time.

Consider the HTML paragraph

```html
<p id="msg"></p>
```

with a JavaScript program as follows:

```javascript
(function(){

  setTimeout( function(){
    document.getElementById("msg").innerHTML += "one<br/>";
    }, 0);

  for(let i=1; i <= 1000; ++i){
    // do nothing, just waste time...
  }

  document.getElementById("msg").innerHTML += "two<br/>";

})();

document.getElementById("msg").innerHTML += "three<br/>";
```

Synchronous vs Asynchronous

At the end of the day, the paragraph will store:

```
two
three
one
```

Even though we specified a timeout of 0 seconds with a request of adding "one" to the paragraph, followed by a long loop, and then later requested to add the "two".

Within the IIFE, the **setTimeout** function was called, and it made a record of the callback function and the time.

At this point, the call stack is not empty. So JavaScript runs the loop and also adds "two" to the paragraph. Even after the IIFE, the call stack is still not empty so the "three" gets added.

Finally, the call stack is empty so the callback function runs, adding "one".

Events

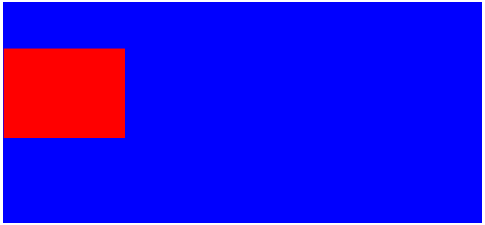
In JavaScript we can specify what events are to take place when the page has loaded by giving a value to the **window.onload** variable (or the **load** event). We can set a callback function to be called once the page has been rendered, the images are in place, etc.

Here is a simple JavaScript program:

```
window.addEventListener("load", function(){
  alert("hello");
});
```

Simple Animation

We consider the work of a simple animation to move a red box from the left to the right of the screen. In the animation, the box will move smoothly from the leftmost part of the screen to the rightmost part.



Simple Animation I

The HTML and CSS provide the basic structure so that the JS code can focus only upon creating and moving the box.

index.html

```html
<!DOCTYPE html>
<html>
<head>
  <title>Animate</title>
  <link rel="stylesheet" href="style.css" type="text/css" />
  <script src="animate.js" defer></script>
</head>
<body>
</body>
</html>
```

Simple Animation II

style.css

```css
body{
  width: 100%;
  height: 500px;
  background-color: blue;
  position: absolute; /* for div positioning */
}

div{
  width: 25%;
  height: 250px; /* half body height */
  position: absolute;
  top: 125px; /* so centered vertically */
  background-color: red;
}
```

Simple Animation III

With the JavaScript, the animation only runs until the box reaches the right. At each interval, we step forward by 1% of the page width.

The value **box.left** represents the left position of the div as it is being animated and we update **box.left**, which in turn is used to set the actual left position.

Simple Animation IV

animate.js

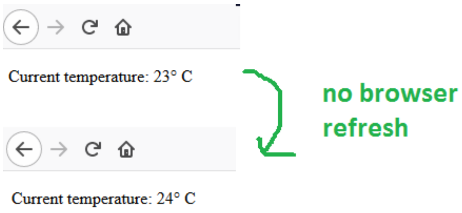
```
// represents/stores all the info for the animated box
const box = {
  width: 25, // %
  left: 0, // %
  end_pos: 100, // %
  dx: 1, // % move by
  element: null, // the div
  wait: 20, // ms delay
  event_id: null  // event id
};

function animate(){
  if(box.left + box.width <= box.end_pos){ // within window
    box.element.style.left = box.left + "%"; // display

    // update position
    box.left += box.dx;
    box.event_id = setTimeout( animate, box.wait );
  }
}

window.addEventListener("load",
  ()=>{
   box.element = document.createElement("div");
   document.getElementsByTagName("body")[0].appendChild(box.element);
     animate();
  });
```

AJAX

Asynchronous Java Script and XML (AJAX) is a way of extracting information from files and other resources while a user is on a page. As the desired data are updated, this can be reflected on the page they visit, without a need for them to refresh.

We'll consider a very simple case study. Imagine we have a website that is supposed to tell the visitors the current temperature. There is a text file that gets updated periodically with the temperature. We want the visitors to see the temperature displayed on the webpage update when/if that text file gets updated...

AJAX


Current temperature: 23° C

Current temperature: 24° C

AJAX I

Here is our HTML (super simple):

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>AJAX Demo</title>
  <script src="ajax.js" defer></script>
</head>
<body>
  <main>
    <p id="msg">
      Current temperature: <span id="temp"></span>&deg; C
    </p>
  </main>
</body>
</html>
```

---

and the JavaScript file:

```javascript
/**
This function creates an object to manage updates to the temperature file
@param {string} file the name of the file
@param {string} element the id of the element to update
@return an object that tracks the file name, element id, current temperature, and length
for the timeout to wait
*/
function Temperature(file,element){
  this.file_name = file;
  this.element_id = element;
  this.temp = null;
  this.interval = 5000; // so 5 sec intervals
```

AJAX II

```
}

/**
This function performs an AJAX call to update part of the HTML page
@param {object} updater a Temperature object
*/
function read_text(updater){
  const xhttp = new XMLHttpRequest(); // object to do ajax with

  // when the operation is complete (readyState 4) and it is successful ( status 200)
  xhttp.onreadystatechange = function() {
    if (this.readyState === 4 && this.status === 200) {
      // change HTML to store what came back
      set_value(updater.element_id, this.responseText);
      setTimeout(read_text, updater.interval, updater);
    }
  };

  // set GET request to read from the file, doing it 'asynchronously'
  xhttp.open("GET", updater.file_name + "?v=" + Math.random() , true);
  xhttp.send(); // do it!
}

/**
This function changes the value of the text within an element
@param {string} id the id of an element
@param {string} value the value to place inside
*/
function set_value(id, value){
  document.getElementById(id).innerHTML = value;
}
```

AJAX III

```
// when the window loads
window.addEventListener("load", function(){
  const updater = new Temperature("temp.txt","temp");
  read_text(updater); // begin reading/checking file
});
```

Here's what we did: we created a global object, **Temperature**, effectively serving as a namespace, to store the timer's event ID, the file to read from, the ID of the element we want to change, and the current temperature.

When the page loads, we perform two processes: we make a **Temperature** variable to store relevant data and set **read_text** as a callback.

read_text sets itself as a callback so that it will be repeated. Both **setInterval** and **setTimeout** can be given third, fourth, ... arguments. The extra arguments beyond the second are used as inputs to the callback function.

AJAX IV

The **set_value** function just looks up an element by ID and inserts the text directly there.

The bulk of the work happens in the **read_text** function. For that we need to discuss the **XMLHttpRequest** object type...

AJAX

From what we see in the example, an **XMLHttpRequest** object has a member, **onreadystatechange**, a callback, that gets called whenever the ready state changes.

When the state changes, provided retrieving the data was successful, we update **data.temp** with the response of the request, stored in the **responseText** member of the **XMLHttpRequest** object.

The **status** member will be 200 if successful.
The **readyState** will be 4 when it is done (3 = in progress, 2 = send has been called, 1 = open has been called, 0 = open not yet called).

AJAX

We call the **open** function to ask the object to perform a **get** request (recall there are different types of requests) from the file **data.file**, and we want the process to occur asynchronously (so other processes can go on). The query string in the file name prevents the browser from caching the results of an old file.

Calling **send** sends the request.

More Events

The syntax for adding event listeners is:

element.addEventListener("event", callback);

where "event" is the event with common values "click" (when clicked), "mouseover" (when the mouse moves over the element), "mouseout" (when the mouse moves off the element); and **callback** is a function object.

More Events

To listen for keys that are pressed, we add an event listener to the **window.document** variable (or just "document"). With events of this type, the event is sent to the callback function for processing.

```
document.addEventListener('keydown',
  function(event){
    alert(event.key + " was pressed");
  }
);
```

The key pressed will be stored as a member variable **key.**

Danger: keydown occurs when any key is pressed down, **keypress** occurs when a character key is pressed down, and **keyup** occurs when a key is released. The arrow keys (up/down/left/right) are only triggered with **keydown**!

Cookies

A **cookie** is a small text file your browser stores when you visit a webpage. When you next visit the page, you may be "remembered" and find your name can still be automatically filled in, etc.

Cookies are created with the syntax:

document.cookie = "name=value; expires=Date of expiry; path=relative path";

where **name** is the name of the cookie and **value** is its value, **expires** specifies when the cookie is forgotten, and **path** specifies the relative directories in the website where the cookie is valid.

Cookies

By default, **expires** is set to when the browser is closed. By default **path** is set to refer to the current folder of the webpage..

If "/" is given for a path, the cookie is valid throughout the entire website; if "/folder" is given, it is valid in "/folder" and all subfolders, etc.

The current working directory with file can be obtained from **window.location.pathname.** Using this as the path ensures the cookie is only valid for the specific page, not even to other pages within the same directory.

Cookies

Multiple cookie name-value pairs can be stored within the **document.cookie**: it is not a regular string! Each time we assign to it, it could store more and more information.

To delete a cookie, we assign to the **document.cookie** a value for name (can be anything or blank) and an **Date** object for expiry in the past.

Warning: do not ever use **Date(0)** for an expiry time. Some browsers use this as the default expiry (when the browser closes). The cookie may not expire!!!

Cookies

In working with cookies, the **split** method of **String** is quite useful. A string object can be broken into an array at a designated splitting character.

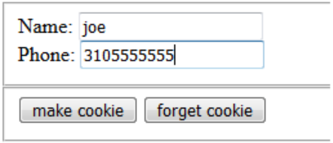
```
let str = "name=John; phone=3105555555;";
let arr = str.split(';'); // arr == ["name=John", " phone=3105555555", ""];
```

Cookies

We will consider a simple case study: a form that can track a user's name and phone number. There are buttons that can toggle between remembering the information and forgetting it.

The user can enter their name and phone and ask for this to be remembered.

When the page loads, if the cookie is being remembered, the name and phone fields get filled in automatically. If the cookie has expired (after 1 minute) or the user asks for the information to be forgotten, the next reloading of the page will find those fields empty.



Name: joe
Phone: 3105555555

Cookies I

Here is our HTML:

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Cookie Demo</title>
  <script src="cookies.js" defer></script>
</head>
<body>
  <main>
    <form>
      <fieldset>
        <label for="name">Name: </label> <input type="text" id="name" /> <br/>
        <label for="name">Phone: </label> <input type="tel" id="phone" /> <br/>
      </fieldset>
      <fieldset>
        <input type="button" id = "make" value="make cookie" />
        <input type="button" id = "forget" value="forget cookie" />
      </fieldset>
    </form>
  </main>
</body>
</html>
```

and here is our JavaScript:

Cookies II

cookies.js

```
/**
This function makes a cookie out of the user data.
It expires in one minute.
*/
function make_cookie(){
  const user_name = document.getElementById("name").value;
  const cookie_name = "name=" + user_name + ";";
  const user_phone = document.getElementById("phone").value;
  const cookie_phone = "phone=" + user_phone + ";";

  const expires = new Date();
  expires.setMinutes(expires.getMinutes()+1);
  const cookie_expires = "expires=" + expires.toUTCString() + ";";

  const cookie_path = "path=" + window.location.pathname;

  document.cookie = cookie_name + cookie_expires + cookie_path;
  document.cookie = cookie_phone + cookie_expires + cookie_path;
}

/**
The user info is removed from the cookie.
*/
function forget_cookie(){
  let past = new Date(1); // 1 ms since the "beginning of time"
  past = past.toUTCString();
  document.cookie = "name=; expires=" + past + "; path=" + window.location.pathname;
  document.cookie = "phone=; expires=" + past + "; path=" + window.location.pathname;
}

```

Cookies III

```
window.addEventListener("load", function(){
  document.getElementById("make").addEventListener( "click", make_cookie );
  document.getElementById("forget").addEventListener( "click", forget_cookie );

  const fields = ["name", "phone"];
  const cookie = document.cookie.split(';');

  for(const field of fields){ // go through each field to populate
    for(const part of cookie){ // look at all parts of the cookie
      let pieces = part.split('='); // split that, the right part

      if(pieces.length===2){ // so enough parts

        while(pieces[0][0] === ' '){ // while whitespace at start
          pieces[0] = pieces[0].substring(1); // remove it!
        }

        if(pieces[0] === field){ // if field found within part
          document.getElementById(field).value = pieces[1];
          break;
        }
      }
    }
  }
});
```

Cookies

Before panicking about security, it is important to note that most websites **do not** store passwords in a cookie!

After a user logs in, a cookie with an identification number attribute may be given to that user. Valid id numbers such that a user should still be "logged in" are stored in a database that the outside world should not have access to.

Warning: some browsers such as **Chrome** strictly interpret **Date**s as UTC Strings. As an extra step we had to convert the value of **expires** to a UTC String time otherwise it will not work in Chrome.

Redirect

If we want to redirect someone, we simply set the **window.location** property:

To redirect to external link, provide full URL with **http:**

```
window.location = "http://www.math.ucla.edu";
```

To redirect to relative directory/page:

```
window.location = "imgs/cat.png";
```

Redirect I

The **window.opener** property is a reference to the window that opened **window**. This is how the evil redirections can take place. Consider:
index.html

```html
<!DOCTYPE html>
<html>
<head>
  <title>Original Page</title>
</head>
<body>
  <p><a href="other.html" target="_blank" rel="opener">Go to other page</a>.</p>
</body>
</html>
```

other.html

```html
<!DOCTYPE html>
<html>
<head>
  <script src="to_evil.js" defer></script>
  <title>Other Page</title>
</head>
<body>
  <p>Welcome to other page.</p>
</body>
</html>
```

Redirect II

to_evil.js

```
window.opener.location = "evil.html";
```

evil.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Evil Page</title>
</head>
<body>
  <p>Bad things have happened!</p>
</body>
</html>
```

It seems for new browsers, the default is that anchors open with **rel="noopener"** so this dangerous redirection cannot happen. But if an anchor is opened with **rel="opener"** then in the example, a user left-clicking on the link on **index.html** will have **other.html** open in a new tab with the **index.html** tab being redirected to **evil.html**.

JavaScript Coding

Before some the final topic, we will look at a number of important style points for JavaScript programming. Many of these points apply to all programming languages, but this list concisely summarizes "good practice".

For readability...

- Comment code.
- Comment every branch of control flow.
- Document functions in the prescribed manner.
- Use descriptive variable names.
- Avoid "magic numbers" and define those values as variables.
- Space out the code nicely.

JavaScript Coding

For robustness and efficiency:

- Always use the semicolons.
- Include braces for all control flow.
- Use === and !==, not == or !=.
- Use an object to serve as a namespace instead of polluting the global environment.
- Use an IIFE to avoid polluting the global environment.
- Use **const** and **let** instead of **var** to avoid hoisting variables to the lexical scope.
- Avoid objects such as **new Number**, **new Boolean**, etc.: use the primitives instead.
- Debug code! Ensure it works.
- Use **defer** for the script tags.

Object Oriented Programming

JavaScript can be implemented as an object oriented language. Here we’ll look at more specifics of creating constructors and classes.

JavaScript adheres to **prototypal inheritance**, which is different to the **classical inheritance** of languages like C++, Java, and Python.

With classical inheritance, we have the notion of a class, which is more abstract and distinct from instances of a class.

In prototypal inheritance, there are no such thing as "classes": everything is an object (even functions, remember!). All objects have a prototype object that in turn can reference another object as its prototype. An object has access to all of its own properties, plus any that can be reached by looking to its prototype, or its prototype’s prototype, etc.

Remark: it really is "prototypal inheritance", not "prototypical inheritance"!

Object Oriented Programming

Recall our example of writing a **Person** constructor.

```
function Person(name) {
   this.name = name;
}

const joe = new Person("Joe");
```

We will now consider adding methods.

Object Oriented Programming

Member functions could be handled in a similar way as adding properties to an object by having a **this.function_name = some_function_object;** in the constructor. But that would be inefficient and not make use of the power of prototypal inheritance.

Instead, we add a property to the object’s prototype. Remember that **Person** is an object...

```
Person.prototype.get_name = function() {
   return this.name;
};
```

Now we can extract the **name** of **joe** via **joe.get_name();**.

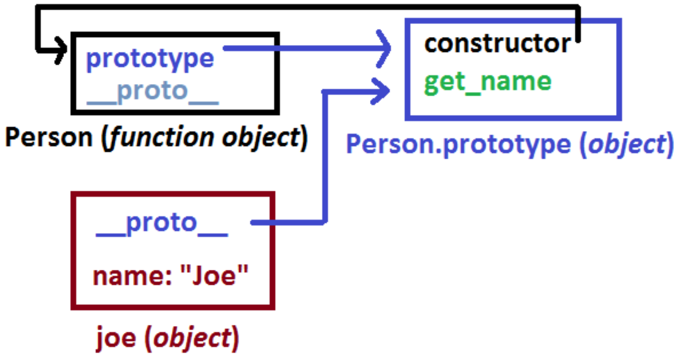
Object Oriented Programming

All objects have a **__proto__** property and it corresponds to the prototype of the constructor that created the object. Strictly speaking the **__proto__** property is inherited and works as a getter (to come).

All function objects also have a **prototype** property, which is an object that references the function through a **constructor** property.

Prototypal inheritance applies to lookup through the prototype of objects.

Object Oriented Programming



Object Oriented Programming

To allow for an object **B** to inherit from **A**, i.e., have as its prototype, the prototype of object **A**, we use:

B.prototype = Object.create(A.prototype);

This creates an empty object whose prototype is that of **A**'s prototype, and this object serves as the prototype for **B**. This should take place before any prototype values of **B** are added.

This will make the B's prototype's constructor to be **A** so we overwrite this with:

B.prototype.constructor = B;

Object Oriented Programming

Within the **B** constructor function, to call the **A** constructor, i.e., to add **A** properties to the **this** of **B**, we write

A.call(this, list of arguments);

We’ll look at an example with Squares and ColouredSquares...

Object Oriented Programming I

We'll write the JavaScript "equivalent" of the C++ classes below:

```
struct Square{
 double length;
 Square(double _length) : length(_length) {} // (A)
 double get_area() const {return length*length;} //(B)
};

struct ColouredSquare : public Square{ // (C)
 std::string colour;
 ColouredSquare(double _length, std::string _colour):
  Square(_length), colour(std::move(_colour)){} // (D)
 void display() const { // (E)
  std::cout << colour + " square of area "
   << get_area() << '\n';
 }
};
```

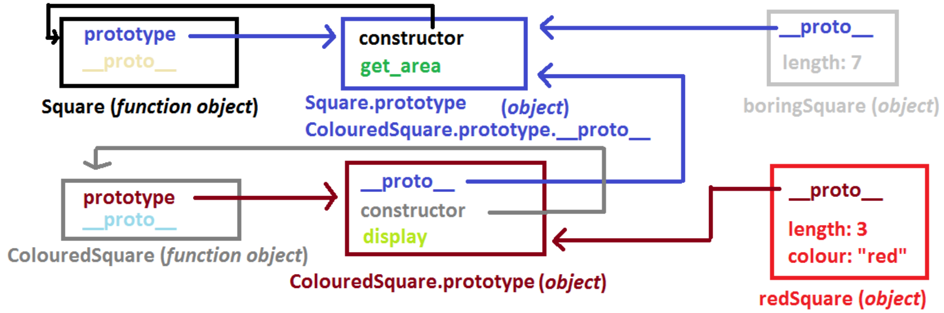
Object Oriented Programming I

Note the correspondences between the marked points (A)-(E) in the two languages.

```javascript
function Square(_length){ // (A): define Square constructor
  this.length = _length; // add property
}

function ColouredSquare(_length, _colour){ // (D): define ColouredSquare constructor
  Square.call(this,_length); // initialize Square properties
  this.colour = _colour; // add ColouredSquare properties
}

Square.prototype.get_area = function(){ // (B): define member function
  return this.length**2;
};

ColouredSquare.prototype = Object.create(Square.prototype); // (C): inherit Square methods
ColouredSquare.prototype.constructor = ColouredSquare; // constructor is ColouredSquare

ColouredSquare.prototype.display = function(){ // (E): add ColouredSquare member function
  console.log(this.colour + " square of area " + this.get_area());
};

let boringSquare = new Square(7);
console.log(boringSquare.get_area()); // 49
let redSquare = new ColouredSquare(3,"red");
redSquare.display(); // "red square of area 9"
```

Object Oriented Programming I

Here’s a diagram of what we have done.



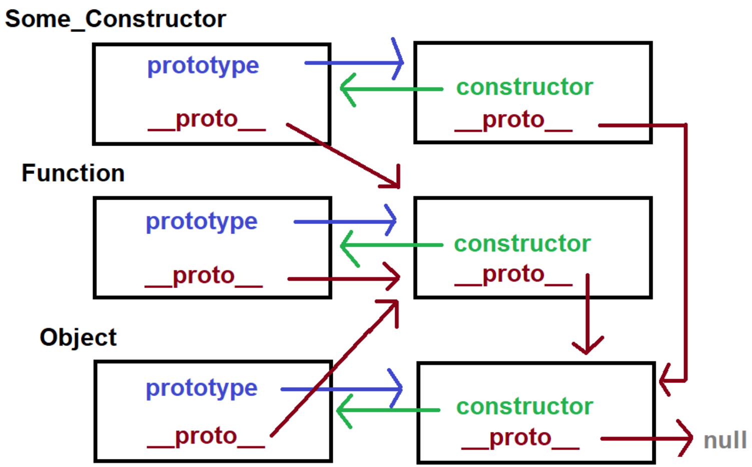
When we call **redSquare.display()**, such a property is not part of **redSquare** so JS looks to its prototype object, which does. When we call **redSquare.get_area()**, such a property is not part of **redSquare** so JS looks to its prototype object. The prototype does not have this method so JS looks at its prototype object which does have **get_area**.

Object Oriented Programming I

In effect, the **__proto__** of an object is the object in which to begin searching for a property if it is not found on the original object itself. The **prototype** is an object that constructor objects use to add functionality to the objects they create.

Here is a picture of the “top" part of the inheritance chain:

Object Oriented Programming II



Object Oriented Programming

Remark 1: the **__proto__** is a little dated. Originally not all browsers supported it and its use was discouraged. Then, it was brought into ECMAScript but other approaches are preferred. In general **Object.getPrototypeOf(o)** is the same as **o.__proto__**.

We can likely view **__proto__** as a property that always returns **Object.getPrototypeOf(o)** when accessing **o.__proto__**.

Remark 2: actually, **__proto__** is not stored on each object and is somehow available through inheritance.

Object Oriented Programming

Remark 3: the ES6 did introduce the **class** keyword in JavaScript giving programmers the syntactic sugar of writing classes like other programming languages. But it does not change what is going on "under the hood". The preceding presentation of inheritance was done in an effort to keep the concept as clear as possible, without glossing over those important details.

Remark 4: lookup does not apply to properties within or derived from the **prototype** property of a function object — only its **__proto__** (or equivalent).

Object Oriented Programming I

Remark 5: updating the **constructor** of derived classes is good practice (so should be done) but the cases where it is useful are more edge-cases. Consider:

```
function Cat(name){
  this.name = name;
}

Cat.prototype.copy = function() {
  return new this.constructor( this.name ); // weird way to call constructor
};

function Kitten(name){
  Cat.call(this, name);
}

Kitten.prototype = Object.create(Cat.prototype);

const k1 = new Kitten("Puffy");

k1 instanceof Kitten; // true

const k2 = k1.copy();
k2 instanceof Kitten // false, is a Cat!

Kitten.prototype.constructor = Kitten;

const k3 = k1.copy();
k3 instanceof Kitten; // true
```

Object Oriented Programming II

The **instanceof** syntax in JS can be used to check if a given entity is an instance of a particular object type.

Because of how the prototype chain is structured when **k2** is made, **k2** is really only a **Cat** but not a **Kitten**.

Aside: it can be useful to define a **copy** function because this way we can save users of our classes the pain of copying an object manually if they want a copy.

Object Oriented Programming I

Here we implement the preceding classes with ES6 syntax.

```
class Square{
  constructor(_length) { this.length = _length; }
  get_area() { return this.length**2; }
}

class ColouredSquare extends Square{
  constructor(_length, _colour) {
    super(_length);
    this.colour = _colour;
  }

  display() {
    console.log(this.colour + " square of area " + this.get_area());
  }
}
```

The **constructor** keyword serves as the constructor function. The **extends** does the inheritance. The **super** invokes the constructor of the base class. The member functions do not have a **function** keyword.

Object Oriented Programming I

Here we implement the preceding classes with ES6 syntax **and with privacy.** With ES6, private variables must be listed within the class and prefixed by **#**.

```
class Square{
 #length;
  constructor(_length) { this.#length = _length; }
  get_area() { return this.#length ** 2; }
}

class ColouredSquare extends Square{
  #colour;
  constructor(_length, _colour) {
    super(_length);
    this.#colour = _colour;
  }

  display() {
    console.log(this.#colour + " square of area " + this.get_area());
  }
}
```

The **#** is part of the variable name. Except for its listing in the class interface, we still need to use **this.** to access these members.

Object Oriented Programming I

JavaScript also gives us the ability to set customized **getters** and **setters.** Both have the syntax is if they were a property of an object but can be used to retrieve values or to assign in a special way.

In the example below, a **Journal** class stores an array of strings called **#entries.** But we can access or write to **#entries** through a property **text.**

```
class Journal{
 #entries;
  constructor() { this.#entries = []; }
  get text() { return this.#entries; }
  set text(more) { this.#entries.push(more); }
}

const j = new Journal();

// as setter
j.text = "Another day in quarantine.";
j.text = "Yet another day in quarantine.";

// as getter
let all = j.text; // [ "Another day in quarantine.", "Yet another day in quarantine." ]
```

Constness I

We have seen that the **const** keyword only prevents assignment to an entity. It does not prevent its members from being modified or assigned to as seen in **foo** below:

```
const foo = {
  x: 1,
  y: 2,
  z: '3'
};

// no error
foo.z = 3;

// 6, not '33'
console.log(foo.x + foo.y + foo.z);
```

Constness II

Object has a method called **freeze.** It can be fed an input argument, call it **i.** It then makes **i** immutable (effectively making all members of **i** **const**). It also returns **i.** This immutability applies only to the members themselves, not to any “descendants."

Changing an immutable value may lead to an error (when mutating member functions are called) or even have no effect.

```javascript
const u = { a: 7, b: [1,2,3] };
const v = Object.freeze( u );

console.log(u===v); // true

u.a = 42; // u.a still 7
v.b = [100,101]; // u.b still as before
v.b.pop(); // u.b now has [1,2]

const arr = Object.freeze([true, true,false, true]);
arr.push(true); // ERROR
arr[0] = false; // has no effect
```

Constness III

If we want, it is possible to “deep freeze" an object an prevent any of its members from being modified. Here we give one implementation. Note that **getOwnPropertyNames** returns an array of *all enumerable and non-enumerable* properties of an object.

The **forEach** function of an array runs through each key in the array and invokes a function upon each of them.

Constness IV

```
function deep_freeze(o){
  Object.getOwnPropertyNames(o).forEach(
    // p is a property of o
    (p) => {
      // access value through o[0]
      const v = o[p];
      // value must be an object (but typeof(null) is object)
      if(typeof(v)==="object" && v){
        deep_freeze(v); // freeze recursively
      }
    }
  );
  // all of o's properties are frozen so now freeze o
  Object.freeze(o);
}

const foo = {
  x: 1,
  y: [1,2,3]
};

deep_freeze(foo);

foo.x = 2; // foo.x still 1
foo.y = [7]; // foo.y still [1,2,3]
foo.y.push(8); // ERROR
```

Using Symbols

Programmers have found various uses for Symbols but according to the ECMAScript Language Specification,

The Symbol type is the set of all non-String values that may be used as the key of an Object property.

Thus, its main use is centered around object properties. One use in particular allows for some properties to be hidden from a **for ... in** loop as symbols do not show up in this context. Even if a variable is not private, it can still be hidden from enumerations if it is a symbol.

Using Symbols

Technically not all symbols are unique. If a symbol is made with the **Symbol.for** function taking a string, all symbols with that same string will be equal.

```
Symbol("A") === Symbol("A"); // false

Symbol.for("B") === Symbol.for("B"); // true
```

Internally a symbol could be implemented as an integer type with many possible values with each symbol getting its own value (except with the **Symbol.for** case).

Using Symbols

In the next example we consider an **Catalogue** class where as items are added to it, they are added as properties with a price. We also track the number of items.

When **Catalogue** objects are printed, we only want to see the items and their price, not the total number of items.

Using Symbols I

```
class Catalogue {
  constructor() {
    this[Symbol.for("count")] = 0;
  }

  add(item, price){
    this[item] = price;
    ++this[Symbol.for("count")];
  }

  print() {
    for(const i in this){
      console.log(i + ": $" + this[i]);
    }
  }

  get num_items() {
    return this[Symbol.for("count")];
  }
}

const store = new Catalogue();
store.add("piano", 500);
store.add("guitar", 350);
store.add("music theory book", 175);

store.print();
console.log( store.num_items);
```

Using Symbols II

The non-symbol properties are printed with the **for... in** loop.

```
piano: $500
guitar: $350
music theory book: $175
3
```

Using Symbols

Symbols can still be found through **Object.getOwnPropertySymbols**.

```
const o = { };
o[Symbol("a")] = 9;

// prints 9
for(const i of Object.getOwnPropertySymbols(o) ) {
   console.log(o[i]);
}
```

Visitor Location

By making use of AJAX and Application Programming Interfaces (**API**s), we can use JavaScript to obtain information about visitors to our website such as their location.

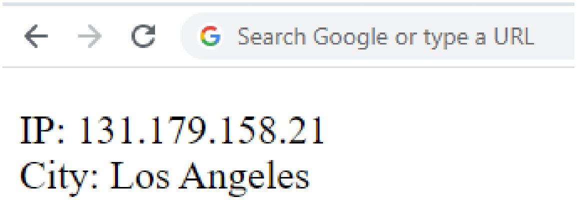
In this example, we will make use of the free APIs, https://api.ipify.org/ and https://ipapi.co/ to display a visitor's IP address and City to the webpage. These APIs do not require API codes and can return plain text (among other options).

Visitor Location I

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <script src="ip.js" defer></script>
  <title>IP Lookup</title>
</head>
<body>
  <p id="visitor"></p>
</body>
</html>
```

The page will display the visitor's IP address and location in the paragraph.



IP: 131.179.158.21
City: Los Angeles

Visitor Location II

ip.js

```javascript
function print_ip(ip,p){
  const text = document.createTextNode("IP:  " + ip);
  const br = document.createElement("br");
  p.appendChild(text);
  p.appendChild(br);
}

function print_city(city,p){
  const text = document.createTextNode("City:  " + city);
  p.appendChild(text);
}

function locate_user(p){
  const xhttp = new XMLHttpRequest(); // object to do ajax with

  xhttp.onreadystatechange = function() {
    if (this.readyState === 4 && this.status === 200) {
      const ip = this.responseText;
      print_ip(ip,p);
      read_location(ip,p);
    }
  };

  // using this URL retrives the IP address of the user
  xhttp.open("GET", 'https://api.ipify.org/' , true);
  xhttp.send(); // do it!
}

function read_location(ip,p){
  const xhttp = new XMLHttpRequest(); // object to do ajax with
```

Visitor Location III

```

  xhttp.onreadystatechange = function() {
    if (this.readyState === 4 && this.status === 200) {
      const city = this.responseText;
      print_city(city,p);
    }
  };

  // with the /city, we access the city of the IP address
  const url = "https://ipapi.co/" + ip + "/city/";

  xhttp.open("GET", url , true);
  xhttp.send(); // do it!
}

// pass the paragraph for editing/updating
locate_user(document.getElementById('visitor'));
```

jQuery

A useful JavaScript library is **jQuery.** It makes a lot of operations and syntax from regular "pure" JavaScript a lot shorter and simpler. In addition, it makes writing functional code across web browsers simpler because it manages "edge cases" like what to do for Internet Explorer...

It is JavaScript, however!

This is a very brief overview of some of its features since JavaScript has already been introduced.

jQuery

To access jQuery, one needs to link to the library:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>JQuery Demo</title>

  <script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
   defer></script>
  <!-- other stuff -->
```

This may take some searching for "most current JQuery library link". It is best to choose the "minified" version so it can be downloaded faster.

jQuery - Selectors

In jQuery, there is a friendly object called **$.** Here are some things we can do with it:

$('foo'): references ALL elements with tag of **foo**, not just an awkward array

$('#bar'): references ALL elements (should just be one) with id **bar**

$('.baz'): references ALL elements of the class **baz**

$("quuz[name='blarg']"): references ALL quuz elements with name **blarg**

jQuery - Events

$(function_object): invokes the function **function_object** when the document has loaded, much like specifying the **window.onload.**

We can add click event listeners:

$('#id').click(function_object);

or what to do as a mouse moves over and off an element with the **hover**:

$('#id').hover(function_move_in, function_move_out);

jQuery - Edits

If **e** is a node (or group of them) then with JQuery:

e.html(); // returns the html an element
e.html('blah'); // sets the html within all to blah

e.attr('foo'); // returns the value of the foo attribute of element
e.attr('foo','bar'); // sets the value of the foo attribute to bar for all

e.css('property'); // returns value of given CSS property of element
e.css('property', 'value'); // sets the value of the CSS property for all

e.val(); // gives the value in e if it is an input field, say
e.val('foo'); // sets the value in e to foo

e.eq(n) // returns a reference to the element at index n of e

jQuery Example I

With the HTML file

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>jQuery Demo</title>
  <script src="https://code.jquery.com/jquery-3.3.1.min.js" defer></script>
  <script src="demos.js" defer></script>
</head>
<body>
  <main id="the_main">
    <p id="p1">first</p>
    <p id="p2">second</p>
  </main>
</body>
</html>
```

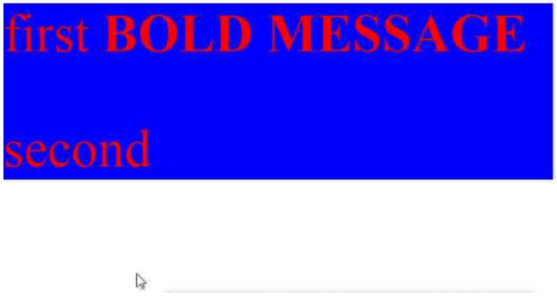
and JavaScript file

```javascript
$(
function(){
  $('p').css('color','red'); // all p's are red
  $('#the_main').css('background-color','blue'); // main background color

  const p1 = $('#p1'); // insert more into the p1
  p1.html(p1.html() + " <b>BOLD MESSAGE</b>");

  $('#the_main').hover(
    function(){ // when the mouse moves over
      $('#p2').html("mousing over");
    },
```

jQuery Example II

```
      function(){ // when the mouse moves out
        $('#p2').html("mousing out");
      }
    );
  }
);
```

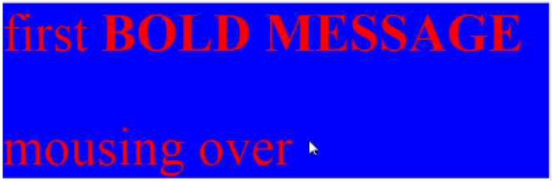
we have the simple functionality illustrated:



first **BOLD MESSAGE**

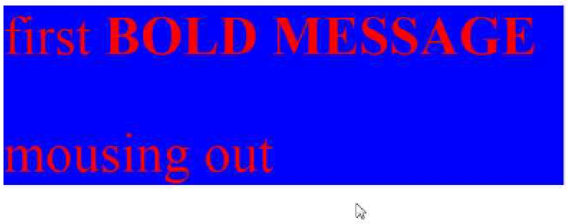
second

jQuery Example III


first **BOLD MESSAGE**

mousing over

jQuery Example IV


first **BOLD MESSAGE**

mousing out

jQuery - AJAX

It even makes AJAX calls easier:

```
$.ajax({
  url : "file.txt",
  dataType: "text",
  success : function (data) {
    $("#my_element").text(data);
  }
});
```

We call the **ajax** function of **$** and pass it an object.

The object has a **url** property for where to look for data, a **dataType** property for what data it will receive, and a **success** property, a callback, for what to do when there is success. In this case, it sets the HTML of the element of ID **my_element** to the text within the file.

Aside: String Interpolation

By enclosing a string in **backtics** (tick mark on tilde key), prepending a variable with **$** and enclosing it in braces, a variable can be interpolated as a string.

```
const x = 111;
const y = `The value of x is ${x}.`;
alert(y); // alerts: "The value of x is 111."
```

PHP will do similar things when we study it.

Aside: Regular Expressions

JavaScript supports regular expressions. A regular expression object is created within /'s. They have a **test** function that accepts a string to determine if it meets the condition.

```
const reg = /\d{3,}/;
reg.test('14'); // false
reg.test('55555555'); // true
```

Aside: Canvas I

HTML5 supports a **canvas** element that works well with JavaScript. It allows for drawing and animations.

Consider HTML:

```html
<!-- ... -->
<body>
  <section id="canvas_area">
    <canvas id="drawing"></canvas>
  </section>
</body>
<!-- ... -->
```

Then with JS:

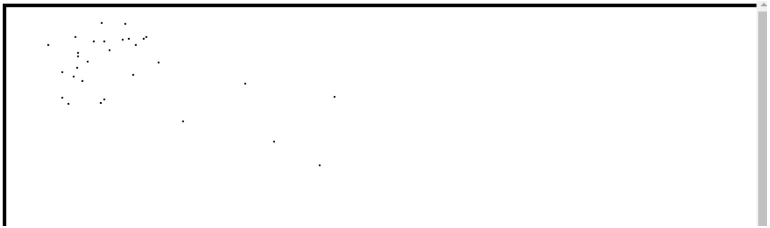
```javascript
function ClickPicture(){
  // set up for drawing
  this.can = document.getElementById('drawing');
  this.can.width = window.innerWidth;
  this.can.height = window.innerHeight;
  this.can.style.border = "4px solid black";

  // a drawing context
  this.ctx = this.can.getContext("2d");

  this.start();
}

ClickPicture.prototype.start = function () {
```

Aside: Canvas II

```
  const self = this;

  // when its canvas is clicked
  this.can.addEventListener('click',
    function(e){ // call the event 'e'
      const rect = self.can.getBoundingClientRect();

      // find coordinates from mouse and draw
      const xpos = e.clientX - rect.left;
      const ypos = e.clientY - rect.top;

      const width = 2;

      self.ctx.fillRect(xpos,ypos,width,width);
    }
  );
}

window.addEventListener( "load", function(){
  const drawing = new ClickPicture();
  });
```

Aside: Canvas III

we have an HTML page that responds to user's clicks, drawing small boxes where the user clicks.



Aside: Canvas IV

With each click, a 2px $\times$ 2px rectangle is drawn at based on the location of the user's mouse click.

The **getBoundingClientRect** function returns a **DOMRect** object that has information such as the size and position of an element (in this case the canvas) relative to the viewport. Accessing **left** and **top** tells the position of the canvas' boundaries.

The **getContext("2d")** returns a **CanvasRenderingContext2D** object that represents a context in which 2-dimensional graphics can be displayed. When this object has the **fillRect** function invoked upon it, we specify a rectangle to be drawn at a given **x** and **y** position with a given **width** and **height** — in respective order.

Remark: orientation is such that we move *down* as **y** increases.

Summary

- JavaScript is a high level programming language, interpreted by a JavaScript engine, that has loose typing and automatic memory management.
- There are 6 primitive types: **undefined**, **null**, **Boolean**, **Number**, **String**, and **Symbol.**
- Arrays, functions, and objects are all “objects".
- Functions always accept objects by reference and fundamental types by value.
- The keywords **let** and **const** define variables and preserve scope whereas **var** does hoisting.
- When **+** involves strings, concatenation is done and other math operations convert arguments to numbers.
- **alert**, **confirm**, and **prompt** can open boxes/inputs for the user.
- One needs to be aware of coercions, **truthy**, and **falsey**.

Summary

- The **Document Object Model** maps the html representation of a page to a set of objects and functions that can be manipulated with JS.
- The **this** keyword means **window** when used globally or directly in the body of a free function; as a member function it means the implicit parameter.
- With **for ... of** JS has range-based for loop capabilities.
- Using **setTimeout**, **seteInterval**,**clearTimeout** and **clearInterval**, one can set up timed events/processes. Times are in milliseconds.
- JS is a synchonous language.
- **AJAX** calls can be used to read/write data while the JS engine continues to run.
- Through **document.cookie** one can set/erase cookies with the **name**, **expires**, and **path** properties.
- JS uses **prototypal** inheritance.
- There are many libraries such as **jQuery** to make work with JavaScript simpler.

Exercises I

1. What is a closure?
2. In JS, how do the behaviours of f and g differ below?

```
function foo(a){
  let ret = null;
  for(var i=0; i < a; ++i){
    ret = function() { return i; }
  }
  return ret;
}

function goo(a){
  let ret = null;
  for(let i=0; i < a; ++i){
    ret = function() { return i; }
  }
  return ret;
```

Exercises II

```
}

let f = foo(2);
let g = goo(2);
```

3. These questions relate to cookies.
   - 3.1 When retrieving cookie information from JS, what format does the cookie string appear in?
   - 3.2 Write code to split the string into an array of name=value pairs and print all such pairs.
   - 3.3 Write code that will create a cookie with "user=Bob" in the "/" path that will expire in 1 year.

Exercises III

4. Write a page that...
   a when the user first visits, they receive an alert “Welcome!” After this, when they visit the page, they are greeted with “Welcome back. You have visited this page [N] times before.” where [N] is the number of previous visits. The “memory” of the page, i.e., how long it remembers a specific visitor should be 1 minute, i.e., if the visitor hasn’t been to the page for more than 1 minute, they are forgotten and greeted with “Welcome” again.
   b does the same as (a) but this time the message is displayed on the page and the text is blue when the number of previous visits is even and green when the number of previous visits is odd.

   Cookies may be useful.
5. What is x?

```
const x = "5" * 3 - "6";
```

6. What is y after the code below?

```
const y = [9, 12, 8];
y.sort();
```

Exercises IV

7. What happens when the ’new’ keyword is used in JS?
8. Write a line of code to redirect a user to **http://www.math.ucla.edu**.
9. Draw a sketch of prototypal inheritance in terms of object structure in memory at the lines 3, 4, and 5 below.

```
1   function Base(i) { this.i = i; }
2   function Derived(i) { Base.call(this); }
3   Derived.prototype = Object.create( Base.prototyp
4   Derived.prototype.constructor = Derived;
5   let x = new Derived();
```

10. Determine the printout

Exercises V

```
i = 5;

function foo() {
  let i = 7;

  ++this.i;

  console.log(i);
}

console.log(i);
```

Exercises VI

11. Determine x and y after the code below.

```
let x = 4;
let y = { foo: 3 }

(function(a,b){
  ++a;
  ++b.foo;
}
)(x,y);
```

12. Write a webpage where a user can enter an **IP** address as a text input. When the press a “Location” button, the city associated to that IP address should be alerted.
13. Redo the **Temperature** example with **jQuery**.
14. Using the pre-ES6 syntax, write an **Employee** class that
    - stores the person’s name, email address, salary, and department;
    - has a constructor to set these fields;

Exercises VII

- has a **setSalary** function to change the salary; and
- has a **print** function to print these fields.

Then, write a **Trainee** class extending **Employee** that also

- stores the person's supervisor's name;
- has a constructor to set all the fields; and
- has a **print** function to print the name, email address, salary, department, and supervisor.

15. Repeat the previous exercise with the **class** syntax of ES6.
16. What does the **const** declaration mean in JS? Is the code below legal? If not, on which lines do problems arise?

```
1   const u = 42;
2   const v = new Date();
3   u++;
4   v.setHours( v.getHours() + 7 );
```

Exercises VIII

17. Write a page that begins by using **promt** to ask the user to enter a regular expression.
    The page should then repeatedly ask the user to enter a name with **prompt**, stopping only when they enter a blank name.
    It should then display all the names in a table with thick, solid borders around all fields. The rows of the table should be of the form:
    [Number] | **[Name]**
    where [Number] starts at 1 and increases by 1 each row and the names appear in the same order that the user entered them.
    Additionally: the background colour for any row where the corresponding name satisfies the regular expression should be yellow.

Exercises IX

18. Write a webpage that, every 10 seconds, displays the current *UTC Time* on the webpage in the format: YYYY-MM-DD at HH:MM:SS, i.e. a 4 digit year, 2 digit month, 2 digit day, 2 digit hour, 2 digit minute, and 2 digit second. For example: “2021-02-08 at 08:25:04”. The font color of the time should alternate between blue and orange, i.e., it first displays as blue. Then orange. Then blue, etc. *Do not worry about the client’s local time!*
19. Write a webpage using the HTML5 canvas. It should, very gradually, draw your name - one small square at a time.
20. Explain the difference between **const**, **Object.freeze**, and performing a deep freeze on an object.
21. Write a function that will freeze only **Array** members (and “descendant" members) of its input object. The freezing should be deep, though: it should go deeper than the direct members of the object. *Hint:* arrays have **Array** as their **constructor**.