Introduction to JavaScript and the DOM
Student Notes

Getting Going

There are several options for writing / executing JavaScript code for practical exercises:

- It's highly recommended to install and use a good IDE that supports JavaScript and HTML directly. JetBrains WebStorm is an excellent choice that has a 30 day free trial and flexible licensing if you decide to continue with it. Other IDEs are, of course, available. If you work in an organization with other JavaScript programmers, it's likely a good choice to use the same tools they do.
- Enter code fragments directly in the Chrome browser's console.
- Use an online editor such as http://jsfiddle.net
- Execute a script in node.js using:
  `node <filename>`
- Load a script into a webpage:
  `<script> console.log("Hello JavaScript World!"); </script>`
  or
  `<script src='relative path to script.js'></script>`
  Note: *Do not omit* the closing script tag!

JavaScript / API Documentation Sources

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

https://262.ecma-international.org/

http://devdocs.io/

Variable Declaration

Variables, as opposed to objects, are entirely untyped and can be used for multiple different types in the lifetime of a single variable.

Variables are generally references to objects, and the object knows its capabilities (methods and fields). Type is essentially the aggregate of capabilities. Runtime errors arise if a type is asked to perform an action that it does not support. Note that some primitive types exist.

`[ var | let | const ] <identifier> [ = <expression> ]`

Four variable declarations forms exist, "naked" (i.e. a variable is used for the first time without declaration), `var`, `let`, and `const`. Note that `const` is used *instead of* `let`, it does *not* modify `let`. The first two forms are older, so will be seen in many examples. The use of `let` and `const`, however, is greatly preferred. In modern coding, immutable data is generally preferred, and `const` is therefore a stronger choice.

The naked or undeclared variables style is considered archaic and strongly discouraged because such variables are created and placed in the global scope, possibly overwriting another definition! This has been the source of many hard to track down bugs in the past.

The `var` declaration form creates *function scoped* variables. The two-pass interpreter causes the declaration to be “hoisted” but any associated initialization statements are *not* moved. This creates curious effects where a variable can be in scope, but uninitialized.

The `let` and `const` forms create “block scoped” variables (this is a newer and preferred form).

A `const` declaration *must be initialized* and creates a constant value. Note that if the value is a reference, `const` does *not* prevent mutation of the target object.

Identifiers should start with a letter, dollar, or underscore, combinations of letters and numbers may follow. Generally, reserve dollar and underscore for special purposes.

Basic Coding Conventions

- `"use strict";`
  should be placed at the of every file (or inside the function literal if used). This causes JavaScript to change in ways that reduce the likelihood of “silent” errors.
  (Note: If you're working with a transpiler, this is typically added to it's output by that tool.)
- Function and variable names are `camelCaseWithInitialLower`
- "Constants" are `ALL_CAPS_WITH_UNDERSCORE`
- K&R style braces (i.e. “{” goes at the *end* of the line, not the start of the next line)
- Functions having `InitialCapsCamelCase` should be reserved for “function constructors”
- Although a source of never ending contention, It’s probably a good idea to terminate statements with semicolons consistently to avoid ambiguity, and surprises, particularly when team members might include less-than top-flight experts (i.e. any real team!)

Literal formats

Decimal, octal, and hexadecimal literals for number type follow the conventions of C, C++, Java etc. Binary format e.g. `0b0101010` is also supported.

Text strings may be enclosed in *either* single or double quotes (allowing strings containing the “other” quote type). Text can include C style escape sequences, such as \n.

Character, boolean, and bigint literals

`'\u56CD'`	Unicode character literals may be used in sourcecode (e.g. representing characters not printable in this machine) or in textual literals.
`true` / `false`	boolean
`10n`	biting (note bigint cannot be mixed with regular number in arithmetic expressions)

Regular expression literals

`/[a-z]+/i`	/ … / … defines a regular expression literal, matching the pattern defined between the slashes, with modifiers that follow. (This example matches one or more of 'a' through 'z', as a case insensitive match.

Array and object literals

`const an_array =` `  [ 1, 3, 5, 7 ];`	Array literal, used here in initialization.The literal is valid in its own right, it is not restricted to initialization.

`let an_object = {`
`  name: "Fred",`
`  age: 42`
`};`
	Object literal, used here in initialization. As above, the literal is valid in its own right.

`{`
`  "name" : "Fred",`
`  "age" : 42`
`}`
	JSON literal, also legal in source code (note the double quotes around the name and age "key" parts. These are required, and must be double quotes, not single as is permitted for other string literals.

Primitives and Objects

JavaScript provides several primitive types/values:

`null`

`undefined`

`boolean`

`number` (64 bit floating point numeric, there is no integer specific arithmetic in JavaScript)

`string` (character sequence)

symbol

bigint

Note that `Boolean`, `Number`, and `String` can exist as either objects or primitives, the uppercase names describe the object types.

Generally, create *primitives* using literal formats, or using factory methods (not the new operator) like this:

```
const ninetyNine = Number('99');
```

If the “constructor form” is used for data that has a primitive form:

```
const ninetyNine = new Number('99');
```

strict-equality comparisons will fail. That is:

`99 === Number(99)` is `true`, but

`99 === new Number(99)` is `false`.

Arrays and Array Elements

Arrays are objects, normally indexed starting from zero. They know their own length, and report this with the length field. They can be dynamically extended by adding new elements at non-existent subscripts, and elements can be removed using the `delete` operator. Elements with numeric subscripts do not have to be at consecutive subscripts–the length is reported from zero to the highest positive integral subscript. Subscripts do not have to be numeric, nor positive, although common usage assumes that the subscripts are consecutive, positive, integers.

Object members / fields

Objects are essentially maps / associative arrays. Field names are keys in the map. Object field access may be as:

```
anObject.x = 99;
```

or:

```
anObject["keys can be expressions"] = 99;
```

Key / value pairs can be added and deleted in the object dynamically. there is no concept of a compile time class that constrains the fields in an object.

Object key types are *not* limited to textual types

Essential Operators

Many C/C++/Java operators are essentially the same in JavaScript:

`+`, `-`, `*`, `/` , `%`, `++`, `--`, `[<index/key>]`, `&&`, `||`, `!`, `? :`

+	String concatenation, with conversion to string (via a method called `toString`)
`obj[<expr>]`	Object element access
`new X(<args>)`	Construct an instance of X using (optional) arguments. Note that the form `X()` does not construct, it merely invokes a function.
`delete x`	Delete the field / variable
`typeof(obj)`	Returns a text representation of the operand's data type. Might be: object, string, number, boolean, function, undefined

Notes:

- Beware of type coercions with mismatched arguments, particularly with +.
- Precedence is broadly normal; parentheses recommended!

Comparison Operators

<, <=, >=, >	Less, less or equal, greater or equal, greater, behave as expected, and produce a boolean
`===`	Strict equals (does not coerce types)
`!==`	Strict not-equal (does not coerce types)

Notes:

- == and != are also equality comparison operators, but are *generally not recommended*, because they perform type coercions that can lead to unexpected results. This relates to the idea of "truthy" and "falsy" values, which can be similarly surprising.
- Equality tests test values of *expressions* (often variables) which are likely to be references, *not* the objects those references point at. Primitives, however, behave as expected.
- JavaScript does not standardize a function name for a “semantic equivalence” test.

Flow Control Constructs

Basic flow controls are as C/C++/Java and the syntax is likely to be largely unsurprising:

`if` / `else`

`switch` / `case` / `break` / `default`

`while`, `do while`, `for`, `for ... in`, `for ... of`

Flow Control Structures with Examples:

The `if` statement does not require an else-if type keyword, because curly brace blocks are used. Simply use `else if` instead. Parentheses are required around the test condition. Although the curly brace block is not required, it's highly recommended in all cases, and necessary if more than a single subordinate statement is used.

```
const x = 10;
if (x > 50) {
    console.log("it's large");
} else if (x > 20) {
    console.log("it's medium");
} else {
    console.log("it's small");
}
```

The `switch` / `case` statement provides for multi-way selection:

```
const x = 10;
switch(x) {
    case 9: case 10: case 11:
        console.log("nine, ten, or eleven");
        break;
```

```
    case 100:
        console.log("it's a hundred");
        break;
    default:
        console.log("It's something else");
        break;
}
```

The match specified by a `case` must be an exact match for the argument of `switch`, this structure is not intended to work with expressions that test the value of the `switch` (use `if`/`else` for that) instead, the `switch` argument will be matched with the expression in the `case`.

The `break` statement is syntactically optional; however if omitted, execution flow will continue to the statements of the following case, which can be useful for alternation (see `case 9: case 10: case 11:` in the above example) but is far more likely to be an error in the majority of situations. Take care not to omit it by mistake.

Unusual for `switch` statements of this type expressions of `case` *need not* be constant. This allows the possibility of multiple matches. The first match in code order (top to bottom) will prevail.

The `while` loop is very conventional with a test enclosed in parentheses following the while keyword:

```
let x = 0;
while (x <= 4) {
    console.log("x is " + x);
    x++;
}
```

JavaScript provides a `do` / `while` statement that definitely executes at least once:

```
let x = 0;
do {
    console.log("x is " + x);
} while (x !== 0);
```

There are three variations of loop using the keyword for. One is a close mimic of the C/C++ for loop. It has three elements in the parentheses, initialization, test, and preparation for the next iteration. The initialization and preparation parts can have multiple elements separated by commas, though it's far more common to declare and initialize a single variable, and update that

same variable in preparation for the next iteration. Here's an example that uses two variables in the loop:

```
for (let x = 0, y = 10; x <= 4; x++, y--) {
    console.log("x is " + x + " and y is " + y);
}
```

Iterating over objects can be done with the `for` / `in` construction. This loop form extracts the keys from an object one at a time and runs the body of the loop with each extracted key:

```
const person = { name: "Fred", age: 42 };
for (const field in person) {
  console.log("person[" + field + "] -> " + person[field]);
}
```

Note that this iterates *keys*, not values and because of this, this loop form is generally inappropriate for arrays. First, it iterates the keys, not the values, and second an array might contain elements with *keys other than regular numeric subscripts*–any elements added to the array, or `Array.prototype` will generally be enumerated too which is not likely to be what's wanted.

A relatively newer feature of JavaScript is the `for` / `of` construction. This iterates directly over the *values* stored in arrays (and some other types known as “iterators”. Iterators are not discussed further here).

```
const arr = ["hello", "bonjour", "guten Tag", "你好"];
arr[10.4] = "ten, ish!!";
arr['weird'] = "a weird value";
for (const value of arr) {
    console.log("> " + value);
}
```

Note that this example prints the values `"hello"`, `"bonjour"`, `"guten Tag"`, and `"你好"`, but not `"ten, ish!!"`, nor `"a weird value"` which are not contiguous positive integer indexes. By contrast the for / in construction would enumerate all the keys, `0`, `1`, `2`, `3`, `10.4`, and `weird`.

If loops are used in modern code it's better style to avoid nesting them, keep them as simple as possible, however, even though their use might be questionable, JavaScript provides `break` and `continue` which can be used in nested loops with labels. The `break` keyword jumps out of the enclosing loop, or a named loop if a label is used (`outer` in this example). The

`continue` keyword abandons the remainder of the body of the current iteration and jumps either to the test condition, or to the prepare for next iteration part if it's a C/C++ style `for` loop:

```
let x = 15;

outer: while (true) {
  for (var i = 0; i < 10; i++) {
    if (i % 3 === 0) continue; // restarts with i++
    console.log(i);
    if (x-- === i) break outer; // abandons while (true)
  }
}
```

Another point of modern style is that, increasingly, explicit loops are being replaced with "monadic" features of arrays, namely, map, filter, flatMap, and forEach. These will not be discussed further at this point.

Truthy and Falsy Values

JavaScript attributes a boolean-like "truthy or falsy" meaning to any value. For example, a non-zero number is treated as true, while zero is interpreted as false. Similarly, an empty string is taken as false, while a non-empty string is treated as true. This is sometimes used as a shortcut, but can also cause quite troublesome bugs. It might be safer to make specific comparisons to ensure the code's meaning is unambiguous. These tests are equivalent:

```
if (x) console.log("true!");
if (x != 0) console.log("true!");
```

Nullish coalescing and chaining

Historically, people have used the automatic coercion to truthy/falsy values to provide easy ways to create default or fallback values. This can be dangerous, since empty string, or zero, and similar, should typically not be replaced. Instead, the "nullish coalescing operator" which is the double question mark "`??`" can be used:

```
let x; // currently undefined
console.log(x ?? "Useful");  // prints "Useful"
console.log(x || "from ||"); // prints "from ||"
x = 0;
console.log(x ?? "Useful");  // prints "0"
```

```
console.log(x || "from ||"); // prints "from ||"
```

This operator is short-circuiting; that is, if the left side is neither null, nor undefined, then the right side is not evaluated.

The `?.` (query-dot) operator provides for conditional dereferencing of object elements. If the regular dot operator is used to attempt to extract a non-existent element of an object, this causes an exception. Using the `?.` operator instead, the result is simply `undefined`. This can be combined with the `??` operator to provide a fallback:

```
let t = {
    value: "A value"
};
console.log(t.value?.toUpperCase()); // A VALUE
console.log(t.nonValue?.toUpperCase()); // undefined
console.log(t.nonValue?.toUpperCase() ?? "fallback value");
// fallback value
```

Functions in JavaScript

JavaScript gives the appearance of functions as top-level, stand alone, features of the language.

Declaring A Function

The fundamental form of function declaration looks like this:

```
function <identifier> ( [<arguments>] ) {
  <code for body of function>
}
```

The function name must be a legal identifier. The function may use the `return` keyword to return a value to the caller, and/or to exit before execution reaches the bottom of the function. If no return keyword is used, or if it is used without an expression to return, a function returns the special value `undefined`.

An argument list for a function is optional. If present, its form is a comma separated list of identifiers that form local variables initialized with copies of the caller's value. For example:

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

The invocation of a function does *not* have to match the supplied argument list:

- Missing arguments result in parameter variable having the special value `undefined`

- All arguments are picked up in an array-like object called `arguments`. This is true whether or not an explicit argument list is provided in the declaration of the function. Access the elements of `arguments` using an integer numeric subscript, just like an array, and use `arguments.length` to determine how many were provided.
- Functions are objects. They can be treated like any other object, and can have properties added to them (including other functions!)

Examples:

```
function makeMessage(name, isFormal, handle) {
  return "Greetings " + (isFormal ? handle : "") + " " + name;
}
```

may be invoked as:

```
console.log(makeMessage("Freda", true, "Ms."));
```

Variable Length Argument Lists

The `arguments` variable already mentioned supports variable argument lists directly.

Default Argument Values

Default values for function arguments may be explicitly provided in the declaration:

```
function doDefaultStuff(arg = "def-val"){}
```

In older versions of JavaScript (pre-ES6), defaults may be created manually in code. This was commonly done by comparing the argument's value with the special value `undefined` and using the ternary operator to conditionally assign a new value if :

```
function handMadeDefault(arg1) {
  arg1 = (arg1 === undefined) ? "default" : arg1;
```

Questionable Legacy Defaulting

Sometimes, this trick using the short-circuit 'or' operator is seen:

```
function doStuff(arg){
  arg = arg || "def-val";
```

However, this is *generally unsafe* because it will replace any value that is “falsey”, including numeric zero, empty string, and `false` itself with the default value.

Destructuring Assignments

JavaScript supports assignment from structures (arrays and objects) directly to multiple variables in a single operation.

From an array, declaring and initializing `x` and `y` from the first and third element of the array:

```
const [x, , y] = ['first', 'second', 'third'];
```

Similar effects can be achieved using objects:

```
const ob = {
  inA : "a value for inA",
  inC : "all at sea",
  inB : "2b | !2b",
  inD : "dee de dee deee!"
};
const { inA: newA, inB: wasB } = ob;
```

The above example results in the variable `newA` having the value `"a value for inA"` and the variable `wasB` having the value `"2b | !2b"`. Notice that the “input” field name is on the left of the colon in the assignment structure, and the newly created variable is on the right of the colon.

If the new variable is to have the same name as the field in the source structure, then a simpler form can be used:

```
const { inA, inC } = ob;
```

This example declares *local* variables `inA` and `inC`, initializing them from `ob.inA` and `ob.inC`.

Spread / Rest Operator

The `...` operator can be used to assign values to or from an array. It provides for variable length argument lists collected into a true array:

```
function addressMessage(message, ...names) {
  for (const n of names) // process each name
```

When the above function is called with many arguments, the first argument would be assigned to `message`, and the remaining arguments are used to form an array referred to by `names`. In this syntax, `...` is called the “rest” operator.

This `...` operator can be used “in reverse”--in which case, it’s called the “spread” operator. When used with arrays it takes an array and “spreads” it out over many receiving elements of another array:

```
const a1 = [1, 2, 3];

const a2 = [...a1, 4, 5, 6];
```

results in `a2` being an array of values `[1, 2, 3, 4, 5, 6]`.

Spread can also spread elements of an array into individual arguments for a function:

```
function values(a, b, c) {}

const a1 = [1, 2, 3];

values(...a1);
```

Which invokes the function called values with the arguments a=1, b=2, c=3.

Spread can also be used in the invocation of a function that uses a rest operator in its argument list.

Declaring A Method In A Literal Object

Traditional syntax:

```
var myObj = {
  firstName: "Sheila",
  lastName: "Smith",
  getName: function(formal) {
    if (formal) {
      return "Ms. " + this.firstName + " " + this.lastName;
    } else {
      return this.firstName;
    }
  }
};
```

Note: *Always* refer to fields of an object using an instance prefix, such as `this`. If the prefix is omitted, you'll be referring to values in the current execution context, not an object. This can result in pushing values into the global execution context by mistake.

ECMAScript 6 provided a new syntax for declaring a function in an object literal:

```
var myObj = {
```

```
  firstName: "Sheila",
  lastName: "Smith",
  getName(formal) {
    if (formal) {
      return "Ms. " + this.firstName + " " + this.lastName;
    } else {
      return this.firstName;
    }
  }
};
```

Using either of the declaration syntaxes, these invocations are valid:

```
console.log('Welcome ' + myObj.getName(true));
console.log('Hi ' + myObj.getName(false));
```

Note: Methods on objects that will have multiple instances should typically be provided via the prototype mechanism, rather than being copied into each literal individually.

Functions As Data

In JavaScript, a function is a legitimate value for a variable, function argument, or `return`. A function is actually just a special, executable, kind of object. This provides syntactic support for “functional programming” style.

Anonymous Function Declarations

A function is an expression (of type function) and does not need a name. This is similar to an object, which doesn't have an intrinsic name, only variables that refer to it, and its own identity.

An anonymous function can be declared in an *expression context* (but not a statement context).

```
var aFunction = function(a,b) {
  console.log('a is ' + a + ' b is ' + b);
};
```

Notice this fails if not assigned to a variable (because this is a statement context, not an expression context):

```
function(a) { console.log(a);}; // FAILS
```

A function argument is an expression context too, so this works:

```
myButton.addEventListener(
  'click',
  function(e) { console.log('click!'); }
);
```

Arrow Functions (Lambda Expressions)

Function expressions have two simplified forms. For a function requiring a block body (usually more than one statement):

```
(a,b) => {
  console.log('a is ' + a + ', b is ' + b);
}
```

and for a function that simply returns an expression:

```
(a,b) => a + b
```

In either case, parentheses are optional around the argument list if that argument list has exactly one element, so this is also valid:

```
a => console.log('a is ' + a)
```

Avoid using arrow functions to define member functions in object literals (they do not attach to the `this` object in the normal way).

Sometimes we must define a function that takes more arguments than we need to use. If these arguments are later in the formal parameter list, we can simply leave them out, but if they precede something we need, we can either use a variable and ignore it in the body of the arrow function, or we can use an underscore as a placeholder, like this:

```
(a,b) => b * b // ignore a
(_,b) => b * b // use underscore as a dummy parameter/placeholder
```

Closures

Functions defined inside other functions may be passed as return values (or parts of composite return values). When this happens, the inner function retains access to the variables of the enclosing function scope even after the enclosing function has completed execution. This behavior is typically called a *closure*.

```
function getTestDividesBy(val) {
  return function (v) {
    // this function retains access to val from the
```

```
    // call to the enclosing function
    return v % val === 0;
  }
}
var dividesByTwo = getTestDividesBy(2); // val is 2 for this function
var dividesByThree = getTestDividesBy(3); // val is 3 for this one
for (var i = 0; i < 10; i++) {
  console.log(i + ' divides by 2? ' + dividesByTwo(i));
  console.log(i + ' divides by 3? ' + dividesByThree(i));
}
```

Closures are often used to create “private” data space that is not accessible directly as fields of objects.

Warning: Although a nested anonymous function retains access to variables in the enclosing scope, the value of `this` in the enclosing scope is not usable. If needed, this should be explicitly assigned to a new local variable, and that local variable should be used instead of `this` in the nested function. Note that this warning does not apply to nested arrow functions.

Immediately Invoked Function Expression

Scripts often need to create a storage “namespace” to work in, so as to avoid cluttering global space and risking collision with other scripts also using that namespace. Each function invocation can create a closure, and a closure creates a perfectly valid and functional namespace.

To completely avoid cluttering the global space, we must avoid naming the function, or using a variable to refer to it. This is handled by creating an IIFE:

```
(function() {
  // variables created in this function invocation persist as long
  // as any reference to them exists. Functions created here can\
  // be attached to other components, e.g. as event listeners
}());
```

Or, as a more concrete example:

```
<input type="text" id="myInput">
<script type="text/javascript">
  (function() {
```

```
    let myInput = document.getElementById("myInput");
    myInput.addEventListener('keyup',
      function(e){
        if (e.keyCode === 13) {
          console.log('line reads: ' + myInput.value);
        }
      }
    );
  }()); // notice this completes the declaration of the function
        // and invokes it immediately. Variable myInput is only
        // visible inside this IIFE, avoiding naming collisions.
 </script>
```

Note that the IIFE must be a function expression, and it *must* be surrounded with parentheses, otherwise it will be treated as a syntax error in the attempt to create a function statement.

Creating Objects With Common Features

The object literal form creates a single object instance, and if called repeatedly will create not simply a new object, but also the function part will be duplicated each time, which is wasteful. One way to avoid this is to define the functions that are common to the instances by placing them in a *prototype* for the objects. This is typically done using the “build from prototype” behavior:

```
const prototype = {
  name: 'unset',
  toString() {
    return 'My name is ' + this.name;
  }
};
const newObject = Object.create(prototype);
// create name in newObject (prototype is unchanged)
newObject.name = "Albert";
console.log(newObject.toString()); // My name is Albert
```

In this example, `newObject` is created and "inherits" everything in the `prototype` object. The new object delegates to the prototype when those features are requested, but it can also redefine the fields, or the behaviors, in itself.

Note: the construction behavior is commonly wrapped in a factory function of some sort.

Static Fields and Functions

Prior to JavaScript ES6, the effect of static fields and functions was achieved simply by adding them to the constructor function (remember that functions are objects and can have members in their own right). Since ES6, a keyword `static` can be added and creates the same structural effect, but with a cleaner syntax.

```
class Date {
  static MONTHSINYEAR = 12;
  static getMonthsInYear() {
    return Date.MONTHSINYEAR;
}
```

allows: `console.log(Date.MONTHSINYEAR);` and `console.log(Date.getMonthsInYear());`

Control Of Fields

JavaScript ES 6 omits the notion of private data in classes. Strong access protection can be implemented using closures instead of objects. JavaScript ES2020 does provide for private data in objects, although this is not done using a keyword. Data elements that are to be inaccessible from outside the class definition must be declared in the class, and are both declared and accessed using the # prefix:

```
class Date {
  #day; #month; #year;  // declare #day, #month, #year as private
  constructor (d, m, y) {
    this.#day = d;
    this.#month = m;
    this.#year = y;
  }

  toString() {
```

```
    return "Date, " + this.#day + " / " + this.#month
      + " / " + this.#year;
  }

  get day() {
    return this.#day;
  }
}
```

This hash-based private feature can be used with instance and static elements, both fields and methods.

Object Methods Controlling Fields

`Object.defineProperty` (and `Object.defineProperties`) can make a field immutable, and/or "hidden" (but *not* inaccessible), or alternatively be defined in terms of accessor and/or mutator functions.

```
Object.defineProperty(<target-object>, <key>, <descriptor>);
```

The <descriptor> is an *object* that may contain these fields:

`configurable`	The field may be deleted, or its type altered
`enumerable`	The field is reported in for in loops
`value`	The field's value
`writable`	The field may be assigned
`get`	The accessor function for the field
`set`	The mutator function for the field

Note that value and/or writable may not be combined with set and/or get. That is, *either* specify storage and an optional write-protection, *or* specify access/mutate functions for the field.

Example:

```
const newObject = {
```

```
  name: "unset",
  toString() { return "My name is " + this.name; }
};
newObject.name = "Albert"; // succeeds
// Now make the name field read-only
Object.defineProperty(newObject, "name", { writable: false });
newObject.name = "Alberto"; // fails
```

Controlling the Controls

We can also prevent changes to both the properties and their configuration.

`Object.freeze(<target>)` prevents any changes to the object <target> from this point forward.

`Object.seal(<target>)` prevents any changes in the set of properties, without altering the writability of the existing fields.

Handling Error Conditions

JavaScript provides an exception mechanism that uses `try`, `catch`, and `finally`. Any type of data can be thrown, but an `Error` object is most appropriate generally. `Error` has a constructor that takes a message-string as an argument. The message should describe the problem in some way, and is available later as the `message` member of the Error object.

A single `catch` block may be provided, which receives anything thrown in the `try` block. The type of what was thrown may be tested with `typeof` or `instanceof`.

A `finally` block can be used for cleanup operations, and if present, is invoked for all flows (successful, unsuccessful-caught, unsuccessful-uncaught).

For example, if the method `dodgy()` might throw an `Error`, then this might be appropriate:

```
try {
  dodgy();
} catch (e) {
  if (e instanceof Error) {
    console.log("dodgy broke, and reports: " + e.message);
  } else {
    console.log("problem of unexpected type caught!?");
} finally {
```

```
  console.log("doing cleanup");
}
```

Problems are reported to callers using the `throw` keyword, with the data to be thrown as an argument:

```
function dodgy() {
  if (Math.random() > 0.5) {
    throw new Error('randomly, that broke');
  }
}
```

JavaScript Inheritance

Inheritance is radically different in JavaScript compared with C++/Java etc.

There is no "class-as-a-template" concept—and no compile-time class inheritance—associated with an object. Instead, if the object itself does not define a requested field or method, the runtime system checks the *prototype* of the object. If the item is still not found, the prototype of the prototype is checked. This proceeds transitively all the way up to the prototype of Object.

The prototype, and its contents, is dynamically variable (by assignment at runtime) and is a per-object feature, not a class oriented thing. This behavior is much closer to the tradition of the *Strategy Pattern* (and therefore more flexible, and probably preferable to the more familiar compile-time inheritance modes, although it is quite likely to be unfamiliar).

Read/write access to the prototype of any object is generally available through the functions `Object.getPrototypeOf()` and `Object.setPrototypeOf()`. In most environments, a field `__proto__` (note there are *two* underscores at each end of the name) is also available. Replacing an object's prototype can be slow, so although it's very powerful, this approach should probably not be used gratuitously.

For a long time, variations in the feature set of JavaScript and its object between different browsers created great difficulties for front-end developers. A common technique used by libraries was to add new features to the prototype of key features of the browser (for example, the document object). In this way, missing features could be added dynamically where they were found missing. This approach has become much less important since the implementation of JavaScript and its key object types is far better standardized today. However, you might still see this approach used to add a library's features to a core aspect of JavaScript. The terms "polyfill" and "shim" are sometimes used to describe this technique.

Function Constructors

A function constructor is a function that has a prototype field on the function object itself. When invoked following the `new` keyword, an object is created. That object has its prototype set to the prototype field of the function, and then that object is passed into a call to the function itself as the `this` value. If the function does not explicitly return anything then the `this` value, after any changes made by the function itself, is returned to the caller.

```
function Thing(color) { // function constructor
  if (color != undefined) {
    this.color = color; // 'this' is the object created by new
  }
}
Thing.prototype = {
  color: "white",
  toString() {
    return 'a ' + this.color + ' Thing';
  }
}
let aThing = new Thing("blue");
```

Note: Although a function constructor is intended to be called using the keyword `new`, it is in fact a regular method, and can in fact be called as a standalone method. It isn't immediately obvious how this could be useful, but some advanced techniques might make use of the feature.

The `instanceof` Test

Objects created from a Function constructor can participate meaningfully in `instanceof` tests. The `instanceof` operator takes two operands, the first is an object, the second is a function. If the prototype of the function that created the object is a prototype of the object, then this returns true.

```
aThing instanceof Thing → true
```

Inheritance With Prototypes

The object that is used as the prototype for a function constructor likely has its own prototype. This creates an “inheritance chain”, as the search for fields and functions will proceed up the chain until it reaches the Object prototype at the top.

Class Syntax

ECMAScript 6 introduced a syntax that for templating of objects using a style based on that of class-based object oriented languages, such as C++, Java, C#, and others. Note that this feature is mostly “syntactic sugar”; that is, it does not change the underlying prototype-based mechanism of the language, nor does it really create a compile-time class concept in the way that other languages understand them.

```
class Person {
  constructor(name) {
    Object.defineProperty(this, "iname", {value: name});
    // this.iname = name; // alternative for simple assignment
  }
  get name() { // creates an accessor method
    return this.iname;
  }
  set name(name) {
    this.name = name;
  }
  toString() {
    return "Person{name: " + this.iname + "}";
  }
}
```

Notes:

- The constructor function is referred to as `constructor()`. It differs from a traditional constructor function in that it can *only* be invoked using `new`.
- Methods are defined in a different syntax, without the use of the keyword `function`.
- Accessor methods may be labeled `get`, in which case, they are invoked as if reading a field, for example:
  `const p = new Person("Fred");`

`console.log(p.name);`
rather than:
`console.log(p.name());`

- Mutator methods may be labeled `set`, in which case, they are invoked as if assigning a field, for example:
  `p.name = "Frederick";`
  rather than:
  `p.name("Frederick");`

Class Inheritance

The class syntax also supports a single implementation inheritance model, this is comparable with languages like C++ and Java from a syntactic perspective, but not so comparable from a semantic perspective.

```
class Employee extends Person {
  constructor(name, position) {
    super(name);
    Object.defineProperty(this, "_position", { value: position });
  }
  get name() { return super.name + " who is a " + this._position; }
  get position() { return this._position; }
  toString() {
    return "Employee{" + super.toString()
      + ", position: " + this._position + "}";
  }
}
```

Notes:

- In a constructor `super(xxx)` passes values to the parent class constructor.
- Methods, including `get`/`set` property methods, may be overridden.
- Parent class fields and behaviors can be accessed using the `super.xxx` and `super(xxx)` syntaxes.
- Despite the syntax, this is still a prototypal inheritance mechanism.

Functional Programming Concepts

Function expressions are commonly passed as arguments into functions and/or returned from functions. A function can create and return a new function derived from an argument function.

Behavior as an argument

Passing behavior as an argument might be likened to passing a sub-recipe; it provides a little supporting behavior to help with the larger behavior built into an operation. Perhaps the most commonly recognized example is that of sorting. The basic mechanism of sorting is an algorithm (for example a quicksort, a merge sort, or even a bubble sort) that has been coded to operate on a particular data structure (perhaps an array). However, for flexibility, the "ordering behavior" should be provided as an argument. This can be achieved with any function, but is a very common use for arrow functions:

```
const names = [ "Fred", "Jim", "Sheila" ];

names.sort((a,b) => b.length - a.length);
```

Using `bind` To Derive Behavior

"Partial Application" is one example of a functional programming technique that can create a new, derived, function from an existing one. The prototype of any function defines a `bind()` method which allows creation of a new function that invokes the existing one using some predefined arguments. The resulting function takes fewer arguments as a result:

```
const add = (a,b) => a + b;

const addFive = add.bind(null, 5);

console.log(addFive(9));
```

In this example, the `bind` method creates a new function (referred to by `addFive`, which is equivalent to:

```
function addFive(b) { return add(5, b); }
```

The first argument to `bind` is the `this` context object that will appear as the implicit `this` argument in the invoked behavior. In our example, it's `null`, as we have no need of it.

Mix-ins

Mix-In is a term that represents the idea of adding behavior (or fields) to an object "from below", rather than the more familiar adding "from above" that is implied by inheritance. JavaScript provides two tools that provide practical approaches to this.

Mix-Ins “By Hand”

If an object (or its prototype) is not sealed, then new fields and functions can be written to the object (or its prototype) at runtime. This can be used to create a “mix-in” effect.

The method `Object.assign(<target>, ...sources)` duplicates enumerable, own, fields from each of sources into <target> (and returns that target object).

Note that this mix-in mechanism modifies an existing object (or possibly the prototype shared by many objects).

Mix-Ins From Class Expressions

Class definitions are actually expressions, in much the same way that functions are. And like functions, they can also be anonymous. This allows an interesting mechanism for creating a mix-in.

```
const Nameable = (x) => class extends x {
  get name() {
    return 'Call me: ' + this.name;
  }
};
```

Notes:

- Nameable is a function that takes a class expression as a parameter (that's the x parameter in the arrow function) and returns a new, anonymous, class expression.
- The returned class expression defines a subclass of the provided class (x), which has been extended by the addition of the features provided in the class body that follows; in this example a name accessor function.
- This mix-in approach creates a new class from which instances can be created.

Usage

To use this, define the class that includes the mix-in like this:

```
class NamePerson extends Nameable(Person){};
const np = new NamePerson();
```

Note that in this case, the `NamePerson` may be given additional features in the class body (between the curly braces).

Alternatively, the new class can be left anonymous:

```
const np = new (Nameable(person))();
```

JavaScript Asynchrony

JavaScript is essentially single-threaded, but in many environments (notably in the browser) it is “event driven”. This means that the code runs from top to bottom performing initialization and setup, and attaching “call-back handlers” to objects that can trigger an event that occurs later (such as UI input devices, and network requests). Each callback handler is a function expression, and the infrastructure (the event loop) invokes the function when the triggering event arises.

Example, log a message on the console 2 seconds after setup:

```
setTimeout(() => console.log("Hello!"), 2000);
```

Given this, the system will configure the specified behavior, but continue executing. After two seconds, a triggering event will be submitted, and when the event processor has completed setup processing, and processing of any earlier events, then the event handler will invoke the anonymous function, printing the message.

If the action is to be performed repeatedly, the `setInterval` method is used the same way, but the behavior is performed again and again, until and unless canceled.

Canceling a Timeout or Interval

If a timeout or interval timer should be canceled, this can be performed using the `clearInterval` method. Both `setTimeout` and `setInterval` return an identifying value which is passed as the argument to clearInterval to stop the triggering of future invocations of the behavior.

Callbacks are very common in JavaScript code, but can quickly become messy due to excessive levels of nesting. The “Promises API” was developed to help clean up programs with non-trivial asynchronous requirements.

Promises

JavaScript's libraries provide a “promise”. A promise allows client software to make a request for an operation to be performed by the operating system, such as an HTTP request, without blocking the execution of JavaScript waiting for the response (which would cause the entire user interface to freeze up).

Using a promise the caller makes a request and provides two callback methods, one is called after the request completes successfully, the other will be called if the request fails.

Promises allow for a sequence of such operations, passing data down the sequence in an easy-to-read, easy-to-code, and easy-to-modify form.

Handling a Promise

A Promise is generally handled through its `then()` method. The `then()` method takes two arguments, both of which are functions. If the Promise completes successfully (“resolves”), then the first function will be invoked with data from the operation that was running in background. If the Promise completes unsuccessfully (“rejects”), then the second function is invoked. Assuming the `doStuff()` function returns a promise:

```
doStuff().then(
  (data) => console.log("success: " + data),
  (errordata) => console.log("failure: " + errordata)
);
```

Success-only or Failure-only

It's not essential to have methods for handling both success and failure provided as arguments to a `then()` function call. If a call to `then()` is not provided for the particular result state–that is, if the operation's outcome is failure, but no failure handler is provided–then that particular `then()` is simply skipped. This makes most sense when `then()` calls are chained.

Chaining `then()` Calls

Promises allow the chaining of `then()` processing. For example, assuming operation() returns a promise:

```
operation()
  .then(success1, failure1)
  .then(success2)
  .catch(recovery3) // convenience form for then(undefined,
recovery3)
  .then(success 4, failure4);
```

The processing function, as appropriate for the success or failure of the previous step, is called with the data or error from that previous step.

Each processing function can return a data item, or another Promise.

If another promise is returned, then it will complete asynchronously either as success (resolve) or failure (reject). If a processing function returns simple data, it is taken to be a success response.

Each step in the chain processes with either the success or failure handler, based on the prior step's result.

If a failure handler returns a promise that resolves, or a simple object, then the subsequent handler will execute on the success path. This facilitates retry type behavior.

Promise Example with the Fetch API

The fetch API provides a promise-based mechanism for making HTTP request. Here's an example that assumes the HTML page has an `<li id="namelist">` element and that the fetched document is JSON format, representing an array of objects, where each object has a field called name:

```
const target = document.getElementById("namelist");

fetch("http://myserver.com/people")
    .then(x => x.json(), e => console.log("error: " + e))
    .then(x => x.map(y => y.name)
        .forEach(n => {
            const content = document.createTextNode(n);
            const item = document.createElement("li");
            item.appendChild(content);
            target.appendChild(item);
        }
    ), e => console.log("error: " + e));
```

Using Promises With Callback APIs

Modern APIs for long-running or asynchronous operations probably return promises directly, but those that do not can be interfaced to the Promise, as in this example:

```
function delay2secs(operation) {
  return new Promise(
   (resolve, reject) => setTimeout(() => resolve(operation()), 2000)
  );
}
```

Note that the call to `resolve(operation())` invokes operation, which is assumed to be internally asynchronous, perhaps built using callbacks, and passes the result of that computation back to the Promise through the resolve function that's an argument of the arrow function. In its turn he promise will pass that data into the success (first) function of the then handler. If the reject method is called, the same basic behavior results, except that the failure (second) method of the then handler is invoked.

Basic API Reference

`String` Methods

Given: `const s = "hello";`

Then:

`s.length → 5`

`s.charAt(4) → "o"`

`s.codePointAt(0) → 104`

`String.fromCodePoint(65, 66, 67) → "ABC"`

Note: 65 is the UTF-8 code for the letter 'A', 66 is 'B', and 104 is 'h'

`s.indexOf('ll') → 2`

Note: return of -1 implies not found

`s.lastIndexOf('l') → 3`

`s.substr(3,2) → "lo"`

Notes:

- Second argument is count of characters to include
- Second argument is optional, all remaining characters will be included if this is omitted
- First argument can be negative, which measures from end of string

`s.toUpperCase() → "HELLO"` (also `toLowerCase`)

Compare Strings lexically using > and <

Split a string based on a regular expression:

```
"hello there how are you".split(/[^a-z]+/i)
```

→ `["hello", "there", "how", "are", "you"]`

Regular expression matching with capturing uses unescaped parens to indicate groups. Matched groups are listed in a returned array. Element zero of that array is the overall match:

```
"1234 And this".match(/([0-9]+) ([a-z]+).*/i)
```

→ `["1234 And this", "1234", "And"]`

Array Methods

Arrays have zero-based integer-sequence index behavior, but can have any object type as an index too. The API includes:

`[1,2,3][0]` → `1`

`[1,2,3].length` → `3`

`[1,2,3].concat([9,8,7])` → `[ 1, 2, 3, 9, 8, 7 ]`

`[1,2,3].fill(0)` → `ar1` now contains `[0,0,0]`

`[1,2,3].indexOf(2)` → `1`

`[1,2,3].join(" : ")` → `string value "1 : 2 : 3"`

Given:

```
const ar1 = [1,2,3,4];
```

And assuming each operation is applied to the *original value* of `ar1`

`ar1.pop()` → `4`, ar1 is modified to `[1,2,3]`

`ar1.push(9)` → `5`–the new length, `ar1` is modified to `[1,2,3,4,9]`

`ar1.shift()` → `1`, `ar1` is modified to `[2,3,4]`

`ar1.sort((a,b) => b - a)` → `[4,3,2,1]`, `ar1` is now `[4,3,2,1]`

If the comparison function is omitted, items are sorted in the natural order.

If a comparison function is provided as an argument to sort, that comparison function must take two arguments of the type in the array, and return the "difference" between them (as if first minus second). The difference value must be numeric, and is considered as positive, negative or zero. The actual value is irrelevant, only the sign matters.

Note, this sort operation *modifies* the original list *and* returns a reference to it.

Array Monad-Like Methods

Arrays also support some functional programming behaviors, e.g.

`[1,2,3,4].reduce((a,b) => a+b)` → `10`

`[1,2,3,4].filter(a => a % 2 == 0)` → `[2,4]`

`[1,2,3,4].map(a => a * 2)` → `[2,4,6,8]`

```
[1,2,3,4].flatMap(x => new Array(x).fill().map((_, i) => i))
```

→ `[0,0,1,0,1,2,0,1,2,3]`

```
[1,2,3,4].forEach(e => console.log(e));
```

Note that `filter`, `map`, `flatMap`, and `forEach` can take a two-argument function, in which case the second argument is the index of the array element being processed. For example:

`[0,0,0,0].map((a, b) => b)` → `[0,1,2,3]`

Also note: if `fill` is called with no arguments, it will fill with undefined, that's fine in the flatMap example, since we immediately map the value to something more interesting based on its position in the sequence.

Generating Visible Output

- In node.js and most browsers: `console.log("some text");`
- Into the body of an HTML page: `document.write("<p>This is a paragraph");`
- Also in HTML pages:

```
<div id='myOutput'></div>
<script type="text/javascript">
     document.getElementById("myOutput").innerHTML = 'Some output';
</script>
```

Reading User Input

- In HTML pages:

```
<input type="text" id="myIn">
<script>
  const myIn = document.getElementById("myIn");
  myIn.addEventListener("change",
    e => console.log("read: " + myIn.value));
</script>
```

Note: Input on an HTML page is “event driven”. That means that the act of typing causes the calling of the function. You cannot have your program call for input when it wants. In effect the cause and effect roles are reversed. The "change" event type is triggered when enter is pressed and a different value is in the text field, not on every keystroke, nor even every press of enter.

Generating Random Numbers

Generate a random number x such that 0 <= x < 1.0

```
const x = Math.random();
```

Convert String And Other Types

- Any Object Type → String: `myObject.toString()`
- Any Object Type → String: `"" + value`
- String to number types: `const x = +"3.2"`

Get Date/Time Now

```
const d = new Date();
const day = d.getDate();
const dow = d.getDay(); // 0 = Sunday
const month = d.getMonth();
const year = d.getFullYear();
```

`Set` and `Map` Classes

A set is a data structure that rejects duplicate elements and provides fast lookup to determine if a given item is in the set. JavaScript's APIs provide a `Set` class. It's key methods include: `size`, `add`, `clear`, `delete`, `entries`, `forEach`, `has`, `values`

A `Set` can be created empty, or using an iterable (such as an array) to initialize the starting contents. Duplicates in the iterable will be dropped from the created set.

Equality in a Set is approximately the same as determined by `===`. Handling of some corner cases (such as `NaN`) are modified to be more practical.

A map is a data structure that is comparable to a database with a primary key and a single value stored against that key. Objects in JavaScript have long been used as maps, since they provide the key features. However there are several disadvantages to this approach and its preferable to use the `Map` class that's part of JavaScript's core APIs. The `Map` class includes these key features: `size`, `clear`, `delete`, `entries`, `forEach`, `get`, `has`, `keys`, `set`

A Map can be iterated using `for of`, in which case it yields a series of elements each of which is a key/value pair in a two-element array.

A Map is an object too, and consequently the normal object subscript access mechanism works with Maps too. However, this form of interaction does not interact with the map's storage, only with that of the object. Therefore it's important to avoid using this by mistake.

Map Example

```
const names = new Map();
names.set("Fred", "Jones");
names.set("Alice", "Smith");
console.log(names);
console.log(names.size);
console.log(names.has("Fred")); // true
```

```
names["Fred"] = "Wait, what?"; // writes to the object structure
console.log(names.get("Fred")); // -> Jones
console.log(names.Fred);        // -> Wait, what?
console.log(names["Fred"]);     // -> Wait, what?
for (const x of names) {
    console.log(x[0] + " : " + x[1]);
}
```

AJAX – Interacting With the Server For Data and Updates

Using the XMLHttpRequest API

Sending an HTTP request and handling successful response from the server may be performed like this:

```
const req = new XMLHttpRequest();
req.open("GET", "path/to/server/data");
req.setRequestHeader("Accept", "application/json");
req.addEventListener("load", function(e) {
  if (req.status === 200) {
    console.log(req.responseText);
    console.log("headers: " + req.getAllResponseHeaders());
  }
}
req.send();
```

Notes:

- The data format received from a request should depend on the `Accept` header, but `XMLHttpRequest` simply treats the data as text.
- Requests that carry entity data from client to server, such as POST and PUT requests, may be sent. Place the entity data into the argument of the send method call.

JSON

JSON conversions may be performed as:

```
const data = JSON.parse(jsonTextData); //
```
data now contains an object

```
const text = JSON.stringify(data);
```

Avoid using eval to parse JSON, since this also executes arbitrary code embedded in the text, which is entirely unsafe. The JSON methods ignore functions and other code.

```
const object = {
  name: "Fred",
  age: 42,
  toString() {
      return "Name: " + this.name + ", age: " + this.age;
  }
};
const jText = JSON.stringify(object);
console.log(jText);
const obj2 = JSON.parse(jText);
console.log(obj2);
const code = "console.log('hello dangerous world')";
eval(code);
```

The Document Object Model

When used in the browser, JavaScript relies heavily on the Document Object Model, or DOM. The DOM may be thought of as the objects that describe the structure and content of a web document and which allow interaction with that document. It's also fair to think of the DOM in terms of the API those objects provide. Elements in the DOM describe essentially all aspects of the presented web page, including the structure, the presentation styling, the text and images of the page, and also user interactions. The essential behaviors provided by the DOM include:

- Finding elements in the page
- Adding and removing elements
- Changing details of elements, such as the text presented
- Attaching behaviors to be invoked in response to a user interaction

Getting Started

Before any JavaScript code can run, it must be loaded and then executed. The relative timing of loading and initializing the page relative to the launching of the JavaScript can cause issues and must be given some attention, though only the basics are addressed here.

The first observation is that if a page contains multiple scripts, they run in order from top to bottom of the page. Things declared before a script runs (whether code or DOM elements) will be accessible to the script when it runs. Here's an example:

```
<html>
    <head>
        <script>showIt(document.getElementById("theThing"));</script>
        <script>
            function showIt(x) { console.log("The item is " + x); }
            showIt(document.getElementById("theThing"));
        </script>
    </head>
    <body>
        <script>showIt(document.getElementById("theThing"));</script>
        <div id="theThing"></div>
        <script>showIt(document.getElementById("theThing"));</script>
    </body>
</html>
```

Notice that in this example, there are *four* scripts. The first fails with an error because the function `showIt()` is not yet defined. The second prints `null` because the `div` with the id `theThing` has not been defined yet. Note that the function `showIt()` is "hoisted" which means that the declaration is treated as if it were at the top of the file. Consequently, in a single script, a function can be called before its actual declaration. The third script also prints `null` because it's still before the declaration of the `div`. The fourth script actually prints `[object HTMLDivElement]` indicating that `theThing` now exists.

From this example, it's clear that it's generally better to declare our scripts at the end of the HTML, just before the closing `</body>` tag.

Events Fired Upon Loading

In addition to controlling script execution by the order they're included in the html page, two events can be helpful in that they're fired at specific times. One is attached to the document, and the other to the window:

```
document.addEventListener("DOMContentLoaded", <handler>, false);
window.addEventListener("load", <handler>, false);
```

The first of these is generally more useful. It fires, calling the function denoted by <handler>, when the structure of the document has been fully resolved. The second is fired after all the resources are loaded too. That's things like css files and images. Generally, the first is sufficient for most purposes, with the second–because it might be significantly delayed–being reserved for those particular functions that must know the exact details, such as dimensions, of all elements such as images which otherwise load later in the process.

Finding Elements In the DOM

There are many ways to find elements in the DOM, and historically many libraries (notably jQuery) have been used to help. In modern JavaScript, there are sufficient techniques that are fully standardized that external library help is generally not required. Some of the key methods are:

`document.getElementById(<the id>)`

In an ideal world, every element in a web page would have a unique ID that's easy to predict. That's not always the case, particularly in pages built by many different individuals on a team, and perhaps built from page fragments that are assembled at page-delivery time. However, if an ID is known, this method allows direct access to it. If more than one element does in fact have the same ID, the result might be unpredictable, but the method returns a single item. Specify `<the id>` as a simple string.

`document.getElementsByClassName(<class...>)`

Many elements on a page can share a class name, and any one element can have several classes attributed to it. This method gets all the elements that carry a particular class or classes. The returned value is an `Iterable` giving access to the items, and allowing iteration using `for of`. Specify the required class as a string. If multiple classes are required to be matched, simply provide a space separated sequence in that string.

`document.getElementsByName (name=...)`

Similar to CSS classes, elements can be given a name attribute. The specification for names (unlike IDs) does not require them to be be unique, hence this method returns an Iterable. However, if spaces are used in a name, they are part of the name, not a separator between multiple names.

`document.getElementsByTagName ('p' etc...)`

The tag name of an element is the particular type of HTML element, for example, a paragraph <p> or a division . This method gives an Iterator of all the elements in the DOM that have been constructed at the point where the script executes.

`document.querySelector(css-selector)`

`document.querySelectorAll(css-selector)`

A CSS selector is a powerful mechanism for identifying an element or elements. They're used extensively with the CSS styling system. These two methods allow their use for identifying DOM elements in code. CSS selectors are specified as strings.

CSS Selectors

There are many CSS selectors, the main ones used in locating elements in the DOM are listed here:

- An unadorned text item is taken as a tag type specification, e.g. `"p"` represents all paragraph tags `<p>`.
- If the text is preceded with a period, then it taken as a classname, e.g. `".numeric"` matches elements like these `<p class="numeric">`, `<div class="currency numeric">`
- If the text is preceded with a hash/pound symbol, it is taken as an ID, so `"#inp123"` matches an element of this form `<input id="inp123">`
- Selectors that are surrounded by square brackets refer to attributes. Some examples are:
    - `[attr]` matches an element that has this particular attribute `attr`, but without regard to the value of that attribute. For example, `[attr]` matches `<p attr="123">`. Note that the attribute can be a core attribute (such as `class`, `name`, and `id`), or can be a user-defined attribute.
    - `[attr=value]` an element that has the attribute `attr` with the particular `value`
    - `[attr~=value]` matches an element that has `attr` with value as one whitespace separated element. For example `[class~="numeric"]` matches elements like these `<p class="numeric">`, `<div class="currency numeric">`

Documentation on the full set of attribute-based selectors can be found at: `https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors`

These four methods are also provided by the `Element` class: `getElementsByClassName()`, `getElementsByTagName()`, `querySelector()`, `querySelectorAll()`. When used on an element, these search only the contents of that particular element, not the entire document.

Combining CSS Selectors

CSS selectors can be used in combinations. Here are some examples:

`p, li` – individually, these match the tags of type `p` and `li`, but when separated with a comma, this creates an "or" relationship, so both paragraphs and list items will be matched.

`p.blue` – this example matches paragraphs that have a class attribute with the value blue in their list. Note that it's important that this example joins the `p` and `.blue` parts *without any spaces*. Adding a space would change the meaning and fail to match what we want.

`div p` – this example matches paragraphs that are located inside a `div` tag. Note that the containment does not have to be at the very next level, so a paragraph contained in a `span` that's contained inside a `div` would still match. Note that this use of space to indicate containment explains why the previous example must not have a space.

`div + p` – this example matches paragraphs that immediately follow a `div`. Note that a paragraph must come *after* the `div` closes, not "after the text `<div>`".

Finding Nodes Contained In Nodes

The Node class gives access to contents by index position. Some key API features on the Node class are:

`childNodes` – this member is a live, read-only, list of the child elements

`firstChild` – this member is a read-only reference to the first element in the node

`lastChild` – this member a read-only reference to the last element in the node

`nextSibling` – this member is a read-only reference to the next element in the node that contains this element

Elements and Nodes

Most, but not all, nodes in a DOM will be Element objects and frequently, we're only interested in Elements specifically. The Element class also provides some navigation methods that are specific to finding other elements. Notice that the names of these read-only attributes all include the word Element:

`childElementCount`

`firstElementChild`

`lastElementChild`

`nextElementSibling`

`previousElementSibling`

Adding and removing elements

Once an element in the DOM has been located, we can interact with it. If it's a container-type element, we can add and remove the elements within it. For example, if it's a `<div>` we can put paragraphs `<p>`, spans `<span>`, and unnumbered lists `<ul>` in it. If it's an unnumbered list, we can add list items `<li>` to it. We can also delete anything that we choose from such a container.

Using `innerHTML`

Sometimes the easiest way to create new elements inside another item is simply to set the innerHTML attribute. This approach rewrites the entire content of the element so it can be used to remove the contents too. However, it's critical to avoid using this to insert anything that isn't absolutely trustworthy because what's inserted will be interpreted by the browser with all the privileges of your site:

```
const target = document.getElementById("here");

target.innerHTML = "<p>This is simple";
```

Working With Elements

The base type for all the structural parts of the DOM is the `Node` class, but most of the time, we'll be working with one of the subclasses of this, notably `Document` and `Element`.

To add an element to the DOM, it must first be created. Elements belong to a particular hierarchy of nodes, and should generally be created in that hierarchy. Most of the time, this means the creation will be performed by the document itself. We can create elements for tags, or text elements, for example:

```
const target = document.getElementById("here");

const para = document.createElement("p");

const text = document.createTextNode("Hello, this is a new text
node!");

para.appendChild(text);

target.appendChild(para);
```

Elements can be removed from their container using `removeChild`. Assuming `list` is a <ul> element with more than two items:

```
const item = list.childNodes[1];

list.removeChild(item); // removes the second item from the list
```

An item can be replaced in a single operation:

```
list.replaceChild(newItem, oldItem);
```

Elements can be inserted before a given existing element in the container. Assuming again that `list` is a `<ul>` with some items in it:

```
const item = document.createElement("li");
text = document.createTextNode("Alfred");
item.appendChild(text);
const beforeHere = list.children[1];
list.insertBefore(item, beforeHere);
```

Changing Elements

Nodes can have contents added, removed, and replaced, but additionally the details of a given element can be altered. Attributes are the features of an html element such as class, id, name etc.. If an attribute name is known, the value can be read or modified:

```
const theId = myDiv.getAttribute("id");
myDiv.setAttribute("class", "selected");
```

Importantly, setting an attribute, particularly the class, might change the presentation. That is, if a style sheet has a particular presentation rule for the class .selected then if this is added to, or removed from, the values in the class specification of an element (remember that a class value is a space-separated lists of class types), then the presentation will immediately be updated to reflect this change.

Changing Classes

Adding, removing, and toggling individual elements of the space-separated class attribute value is such a common operation that this is directly supported by the API of the Element. Obtain access to the classList feature of the Element and then the methods `add`, `remove`, and `toggle` may be used directly:

```
target.classList.toggle("red")
```

Using Events

If something happens at an unpredictable time, and in particular if that time is a result of external influences such as user interaction, or the operation of a disk or network, it is inappropriate to write JavaScript code that waits until the occurrence. Instead the occurrence should create and send an event, and our code processes the even in the "event processing loop".

Generally JavaScript code runs from top to bottom configuring the environment as it needs, and then, invisible in our code, execution enters the event processing loop. All the timeouts, user interactions, and more actually put a data item, called an event, into a queue. At any given moment the execution engine is either processing something, or waiting for a new event to arrive in an otherwise empty queue, or it's pulling the oldest event off the queue and is about to process it.

We need to attach our behaviors for things like button presses and other user interactions to these events, and we need to know how to determine which event type will arrive, and what supporting information it will carry, for the events we must handle.

Adding Code To Process an Event

For user interactions, we will attach our processing code to an element of the DOM. Assuming button is a button element in the user interface, we can attack a click handler like this.

```
btn.addEventListener("click", e => console.log("Ouch!"));
```

Multiple handlers can be added, even with the same type attached to the same element. If a handler is to be removed, this can be achieved with the removeEventListener call:

```
btn.removeEventListener("click", handler);
```

Critical Warning About Processing Time

JavaScript and browsers are essentially single threaded. The system creates a degree of asynchrony by using the event model. However, it's critical to understand that when any JavaScript code is executing, *nothing else will happen*. This means two things:

1) The initialization phase of our scripts–that is, the time they take to run from top to bottom–must be kept short. The browser will appear to be frozen until these scripts are fully completed. Any "ongoing" processing *must be triggered by events*.
2) The time spent executing an event handler must also be short. Just as with initialization, while the event handler is running, nothing else can happen and the browser window will appear to be frozen.

The bottom line is that we must ensure our event processing code completes quickly; if it takes more than a few tenths of a second, the user will quickly become anxious that something has gone wrong–and remember, their machine might not have nearly as much power as yours does.

Event Types

There are many different event types, and any given event source might define its own. The only reliable method for determining what events might arise is to check documentation. Events, like many other aspects of the DOM are not always consistent between browsers, so it's a good

idea to check the compatibility of what you're intending to use. However, most of the core behaviors used regularly have events that are available in all modern browsers. Some examples are:

`click`, `focus`, `blur`, `keydown`, `keyup`, `mousedown`, `mouseup`, `mouseenter`, `mouseleave`, `mousemove`

Event Data

Event handlers are called with an event object that provides additional information on the details of what happened. This information usually includes the target Element that received the event, and often much more such as the X/Y coordinates of a mouse event, or the key that was pressed, in each case, the exact contents of the event object are specific to the type of event that was received. More details on this can be found in the MDN documentation at:

https://developer.mozilla.org/en-US/docs/Web/API/Event#interfaces_based_on_event

Event Delivery

In the simplest interpretation, an event is delivered to the callback handler/handlers for that event type that are registered on the object whenever that event happens at the object. However, the objects receiving events, which are often DOM elements, usually have a hierarchical container relationship. That is it could be that the document contains a `<div>` that contains a `<span>` that contains a `<button>`. So, if an event, for example a `click`, is triggered on the button, then it's also in the "screen area" of everything else. And, in general, if we add an event handler to all three of these elements, they will all receive the event. The delivery will, in general, be to the button first, then to the span that contains it, then to the div that contains that.

In this model, we think of the container as the "top" of a structure, and the nested elements as being "below" one another. So, the button would be at the bottom of this. The flow of the event from the bottom (most specific) element to the top (the outermost container) is called "bubbling".

In a case like that just described, the event's `target` field will refer to the button, even when the event is delivered to the `<span>` or `<div>`. The field `currentTarget` indicates the element to which the event is currently being delivered.

However, the whole story is a bit more complex. Events actually start at the largest container, and are available for processing in what's called the "capture" phase. After the event reaches it's target, it's passed outward again from the target, up through the containers. This is the bubbling phase just mentioned.

We can control an event registration to determine whether it attaches to the capture phase or the bubbling phase. The simplest way to do this is with a third, boolean, parameter passed with the addEventListener call. The first of these adds an event listener for the capture phase, the

second for the bubbling phase. Note that leaving out the third argument results in attaching the event to the bubbling phase.

```
div.addEventListener("click", e => console.log("capture div"), true);
div.addEventListener("click", e => console.log("bubbling div"));
```

Not all events support the bubbling phase and they will be attached to the capture phase by default. The documentation will indicate which events this applies to.

The event phase can be determined from a field of the event object called `eventPhase`, and will be the values 1, 2, or 3 indicating capturing, at the intended target, and bubbling, respectively. Note that the intended target can receive the event twice if it's registered for both capturing and bubbling.

Stopping Propagation

The event object has a method on it `stopPropagation()` . If this method is called inside an event handler, there will be no further processing of the event. If the handler that does this is in the capture phase, there will be no more capturing and no bubbling at all. If the `stopPropagation()` occurs during the bubbling phase, then it will simply not propagate through bubbling to any more containers.

This can be particularly useful in situations where a contained element wants to process an event, such as a click, but the container also wants to process that with a different effect. By ensuring that the event does not propagate in the bubbling phase, the container will only receive the event if it occurred in the boundaries of the container element, but not within the inner element.

Preventing Default Behavior

Some elements, for example, the submit button of a form, have a default behavior of their own. Sometimes, we need to prevent this behavior from happening. By calling the method `preventDefault()` on the event object, an event handler can achieve this end.

Testing JavaScript

There are many ways of testing JavaScript in a browser. Broadly, it can be done from a framework that simulates a browser, or internally to the browser code itself.

The former is commonly performed with software such as Selenium Web Driver, while the Mocha framework can be run on the page itself in a browser. Of course, embedding tests in the page requires that those tests be removed before deployment in production but mocha can be invoked with only four lines added at the end of the HTML body, so the cleanup can be done fairly simply by removing those lines.

Unit Testing Principles

A well designed unit test should have certain characteristics:

1) it should test just one thing, so it's easy to understand what went wrong in the event it fails
2) it should have no dependencies or side-effects. It's important that it shouldn't matter what order tests run in, and in particular not matter if a test is removed from the suite. Tests that exhibit such dependencies cause great difficulties in practice since the source of a failure (or an incorrect pass) is often hidden in another test's execution.
3) The test should be written in three steps:
    a) set up the pre-conditions
    b) execute the thing that is to be tested
    c) verify the results that should be found

    These three steps are sometimes referred to as "given, when, then" steps. If implemented clearly, they should form a kind of documentation of one aspect of what the thing being tested does.

The Mocha Framework

Because our tests should follow the three steps noted above, it's usual to use a framework to help with the repetitive aspects. While there are several frameworks to choose from, Mocha is one such and we'll introduce that here.

Using `describe()` and `it()`

Mocha tries to make your tests read reasonably naturally, two key functions, `describe()` and `it()` are used to help with this. Here's a very simple example:

In the HTML file, at the end of the `<body>`:

```
<div id="mocha">

<script
src="https://cdnjs.cloudflare.com/ajax/libs/mocha/8.0.1/mocha.min.js"
></script>

<script src="scripts/myscript.js"></script>
```

The `<div id="mocha">` defines a place on the web page that will display the results of the tests. The first script is the Mocha framework, you might load this from the main originating provider, or you might keep a local copy on your server, the URL above loads from the main provider. The second script (`scripts/tests.js`) is the file that triggers the tests. In this case, we've chosen to name the tests file `tests.js`, but this is not important.

Here's a minimal example of what `tests.js` might look like to illustrate the key parts:

```
mocha.setup('bdd'); // set up the mocha framework
describe('test', function() {
  it('passes', function() {
    const pi = 3.141;
    console.log("Unit diameter circle area is " + pi * 0.5 * 0.5);
    // this test is considered to pass because nothing is thrown
  });

  it('fails', function() {
    // this fails because it throws an Error
    throw new Error("That broke");
  });
});
mocha.run(); // trigger the running of the tests.
```

Notice that the `describe()` function takes a descriptive text string. This should be fairly verbose, its mission is to describe what's being tested. The second argument to `describe()` is another function. This function (and the functions passed to the `it()` method calls) are usually anonymous functions, but they should not be arrow functions. The reason for avoiding arrow functions is that an arrow function does not get the correct value for the `this` context object, and some operations (not illustrated in the above example will fail in that situation).

The argument function to `describe()` calls the `it()` method as many times as needed for the individual tests that should be run. Each invocation should take another descriptive text string, and another (non-arrow) function. This function describes the actual test to be performed.

If the function provided to the `it()` method returns normally (that is, does not throw anything) then it is considered to be a successful test. If it throws an `Error`, it's considered a failure. In fact, if the function throws anything, it will be considered a failed test, but it's expected that the code should specifically throw an instance of `Error` to indicate this.

The final step in the code is the call to `mocha.run()`, this launches the tests. Note that the output will be presented in the `<div id="mocha">`. This can be formatted with an appropriate style sheet, but is readable without any styling.

Setup and Teardown

With this style of testing, as much as possible each test should run independently of all the others. This means that there's likely to be a lot of repeated code to set up the starting point for each test. Rather than have to repeat this, the `'bdd'` mode of Mocha provides easy hooks for triggering specific methods to run before and after both the suite of tests, and each individual test. These are the methods `before`, `after`, `beforeEach`, and `afterEach`. These methods do not take a description string, but instead simply take a function that will be invoked as the name suggests. For example this `describe()` function:

```
describe('test', function() {
    before(function() {console.log("before all tests");});
    after(function() {console.log("after all tests");});
    beforeEach(function() {console.log("before each test");});
    afterEach(function() {console.log("after each test");});

    it('passes', function() {
      console.log("Success!");
    });

    it('fails', function () {
      console.log("Failure!");
      throw new Error("That broke");
    });
});
```

Will create this output on the console:

```
before all tests
before each test
Success!
after each test
before each test
Failure!
after each test
```

```
after all tests
```

Assertion Frameworks

In addition to using a test framework, it's common to use an "assertion framework" too. This is simply code that allows clean expressions of tests. For example, it's common to want to validate that a certain calculated result has a particular value, or if it's floating point, a value that's very close to the expected value (to allow for rounding errors). Although this sounds like a simple problem, it can get quite repetitive to provide the exact tests required. Consequently there are many frameworks that express these tests. As an example, the "Chai" assertion framework would allow a check to see if a computed result has the value 5 using this code:

```
chai.expect(x).to.equal(5);
```

To load Chai into the browser, add this script tag, probably right after the loading of Mocha:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/chai/4.2.0/chai.min.js">
</script>
```