Javascript Commands
With Commentary
(for learning and homework; NOT for exams or quizzes)

Embedding Javascript

```
1  <html>
2     <head>
3        ...
4        <script type="text/javascript" src="path/to/script1.js"
             defer></script>
5        <script type="text/javascript" src="path/to/script2.js"
             defer></script>
6      ...
7    </head>
8  <body>
9    ...
10   <script>
11      // Javascript code goes here
12   </script>
13   ...
14 </body>
15 </html>
```

Some helpful things to know about Javascript before we begin:

- Javascript has become a very widely-used programming language. Here we focus on its role in the context of client-side web usage.
- There are many versions of Javascript. This guide focuses on ES6 syntax and conventions.
- Javascript is not Java. However, the syntax of control structures, math notation, and comments is quite similar between the two languages.
- Client-side Javascript is sent along with a web page and is executed by the browser. Therefore, the same Javascript code might perform differently from one browser to another. As browser makers give more attention to published standards, these differences are becoming fewer than in the past.

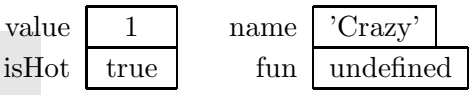
In this example we demonstrate two ways to embed Javascript in a document.

1. On lines 4–5 we link to Javascript code that is stored in an external document.
2. On line 10–12 we directly embed Javascript code on the web page.

There are lots of opinions/options regarding *how* and *where* in a document Javascript code should be embedded. Generally it is considered good form to put all Javascript in external documents and to reference them in the `head` using either `async` or `defer`. I have a slight preference for `defer` because it will execute the code after the page is loaded. Also, if multiple documents are referenced they will be executed in order (which is not guaranteed when using async).

Variables, Types, Comments, and Semicolons

```
1  /* When we declare variables we don't give
2     them a type. We only specify a name. */
3
4  let value, name, isHot, fun
5
6  value = 1         // type is Number
7  name = 'Crazy'    // type is String
8  isHot = true      // type is Boolean
```



It is good form to declare variables using either `let` or `var`. There are lots of resources that explain the nuances of these two declarations. For our purposes we'll prefer `let` in many cases.

Comments are used to provide notes/annotation to be read by other programmers. Comments are ignored by the browser. To create a multi-line comment use `/*` at the beginning and `*/` at the end of the comment block. An single-line comment starts with `//` and goes to the end of the line.

When we *declare* variables (line 4) we announce we plan to use them but they are not given useful values until we initialize them (lines 6–8). *Initializing a variable* means "giving it a value for the first time." Initializing a variable is accomplish through use of the *assignment operator* (i.e., =). The assignment operator takes a value (on the right) and places it into the variable (on the left). A variable that has been declared, but not initialized (e.g., `fun`) contains a special value called `undefined`.

When we assign a value to a variable it takes on a *type* based on the value it is assigned. Common Javascript types are:

Number	a numeric value that can be used in calculations
String	strings; zero or more characters (like words)
Boolean	boolean; either `true` or `false`

A variable's type is not fixed as in some languages. We say that Javascript is a *dynamically typed language*. A varaible's type in Javascript is based on what kind of value it is holding at a particular moment.

Folks used to Java or C++ may notice that the above example does not use semicolons at the end of each line. In Javascript semicolons are mostly optional. One exception to this rule is if you put two separate statements on a single line. In that case you'll need a semicolon to separate the statements. The examples will not use semicolons unless they are needed (much to the chagrin of some people).

Displaying Values

```
let name
name = 'Crazy'
console.log(name)       // write to browser console
alert(name)             // present in annoying pop-up box
document.write(name)    // place on page in current location
```

Output of all three statements (but in 3 different locations):

```
Crazy
```

We typically only display values in this way when debugging a program. Once our code is working we update values on the page by manipulating the DOM (see "The DOM" section later in this document). Some considerations in using these approaches:

console.log This command displays the command in the browser's console which can be accessed by inspecting an element on the page and then clicking the "Console" tab. This method is common for debugging because it silently writes to the console without otherwise interrupting the flow or display of the page. A disadvantage is that the console is cleared when redirected to a new page so it can be hard to track down redirection errors.

alert This is a highly disruptive command that will cause the application to stop execution until the popup with the message is cleared. Sometime this disruptive behavior is desired to prevent a script from continuing past a checkpoint. It also has the benefit that it doesn't require opening a window to the console. On the other hand you must manually clear the popup before the code can continue to run.

document.write The command actually attaches its output directly to the page at the location the command was issued. For this reason it would only make sense if the script code were placed in-line in the body of the document (which is unusual).

```
1 let doMore, num
2 doMore = confirm("Continue?")
3 num = prompt("Enter fav number: ")
4 console.log(doMore)
5 console.log(typeof(doMore))
6 console.log(num)
7 console.log(typeof(num))
```

Of course the output depends on what the user does. Suppose the user clicks OK at the first prompt generated by line 2 and then types 10.2 followed by OK at the second prompt generated by line 3. Expected output (in the console):

```
true
boolean
10.2
string
```

Getting input is another action that we typically handle by grabbing information from a form on the page (via the DOM) rather than by issuing a traditional read command. For quick-and-dirty getting interactive input you can use one of the commands demonstrated above:

confirm This command provides a pop-up with options to click "OK" or "Cancel" in response to the provided text. The command will return true if the OK button is clicked an false if Cancel is clicked.

prompt This command provides a pop-up with a text box for free-form input. When the user enters text into the box and then clicks OK, the data in the form is returned as string.

Notice that the value "10.2" returned by the `prompt` command is stored as a string. Depending on future usage it may require some attention to properly work with that value. See the section called "Arithmetic" below.

Arithmetic

```
1  let a, b, c, d, e, f, g, h
2  a = "5.5"
3  b = 3
4  c = a + b
5  d = b + Number(a)
6  b++
7  e = b / 3
8  f = Math.floor(b / 3)
9  g = b % 2
10 h = Math.random()
```

a	”5.5”
b	4
c	”5.53”
d	8.5
e	1.333333
f	1
g	0
h	0.7697511504757932

Arithmetic in Javascript is fairly simple. You can use *arithmetic operators* ($-$, $+$, $*$, $/$) to perform addition, subtraction, multiplication, and division, respectively. The order of operations works as in algebra. Parentheses can be used to override order of operations. NOTE: There isn't a integer type in Javascript so division works as expected. Shortcuts such as ($++$, $--$, $+=$, etc., work as they do in Java/C++).
Some notable concepts from this example:

- The variable `a` is of type string and therefore when used with the `+` operator on line 4 causes the action to be contatentation of strings rather than numeric addition (hence the surprising result “5.53”).
- To use `a` in numeric calculation we can typecast it to a number as shown on line 5.
- The variable `b` is 4 at the end of the sequence because it was incremented by 1 using the `++` operator on line 6.
- To achieve integer division the `Math.floor()` function can be applied to chop off decimal places as show on line 8.
- The variable `g` ends with a value of 0 because the remainer of 4 when divided by 2 is 0.
- Line 10 illustrates use of the `Math.random()` function which returns a pseudo-random value in the range 0.0000000 to 0.9999999.

There is a fairly extensive library of functions in the `Math` object. See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math`

If Statements

```
let x, y
x = "5.5"
y = 0
if (x == 5.5) {
   console.log("A is true")
}
else {
   console.log("A is false")
}
if (x === 5.5) {
   console.log("B is true")
}
else {
   console.log("B is false")
}
if (x) {
   console.log("C is true")
}
else {
   console.log("C is false")
}
if (y) {
   console.log("D is true")
}
else {
   console.log("D is false")
}
```

Expected output (in console.log):
A is true
B is false
C is true
D is false

The *if-statement* allows you to select one of two alternate actions based on a comparison. After the word "if" you must have an expression inside parentheses. Typically this expression would be a boolean expression (which evaluates to either true of false). If the expression evaluates to true the statements that are indented below the `if` in the first section will happen. Javascript uses braces to define the scope of the steps belonging with the `true` case. In the case the boolean expression evaluates to false, the `else` part will happen. Keep in mind the statements in either section could be if-statements themselves (which is called "nested ifs"). The `else` portion of the statement is optional. If else is omitted and the boolean expression is false then the statements are simply skipped.

Javascript will allow non-boolean expression to appear in the condition. In such a case values that reduce to a number type and equal to 0 are considered false while all other values are interpreted as true.

The examples above illustrate two relational operators (`==` and `===`) The first returns true if the two items it is comparing are equal while the second returns true if the two items are strictly equal. In this example x is a string with the value "5.5" which when compare to a number 5.5 is considered equal (but not strictly equal).

The second two examples show non-boolean expressions with the first evaluating to true and the second to false.

```
1  let isCook, a, b, c, d, e, f, g, h
2  isCool = false
3  a = 7
4  b = -1
5  c = true
6  d = a < b
7  e = a < b || a > 0
8  f = a <= b && a != 0
9  g = !isCool && a == 7
10 h = 0 <= b <= 100  // not what you expect!
```

a	7
b	-1
c	true
d	false
e	true
f	false
g	true
h	true

A *boolean expression* is an expression that evaluates to either `true` or `false`. NOTE: Javascript requires those two constants to be written in lower-case letters and no quotes. Boolean expressions can be built using a combination of variables, relational operators, boolean operators, and constants.

Relational operators (<, <=, >, >=, ==, ! =, ===, ! ==) compare values of the same type and produce a boolean result. NOTE: == compares values for equality and is different than = (which is the assignment operator). For the difference between == and === see the previous section. Also, ! = means "not equal".

Boolean operators (`&&`, `||`, `!`) act on boolean values and produce a boolean result. NOTE: `&&` is true only if both parts are true. The boolean operator `||` is false only if both parts are false. When placed in front of a boolean expression the operator `!` toggles the expression's value. Parentheses can be used to specify the desired order of evaluation of a complex boolean expression.

In this example the resulting value of `h` may be surpising because `b` is not in the range 0 to 100. While this code runs without error it doesn't not evaluate the expression in the way that was probably intended. First, `0 <= b` is evaluated to be `false`. Then `false <= 100` is evaluated to be true (because false is coerced to 0).

```
let one, two, num
let a, b, c, d, e, f, g, h
one = "The quick brown fox"
two = ' jumps.'
num = two.length  // 7
console.log(one + two)
if (one > "ABC") {
   console.log("bigger")
}
a = one[4]
b = one.substr(4,7) // 7 chars starting at 4
c = one.substr(15)  // skip first 15
d = one.substr(-2)  // keep last 2
e = two.replace("jump", "leap")
f = two.replace(/\w+/, "xxx")
g = two.trim()  // remove leading/trailing whitespace
h = two.toUpperCase()
```

num	7
a	"q"
b	"quick b"
c	" fox"
d	"ox"
e	" leaps."
f	" xxx."
g	"jumps."
h	" JUMPS."

Expected output:

```
The quick brown fox jumps.
bigger
```

Strings are used to hold a series of characters. String constants can be specify using either single or double quotes. The `str.length` function returns the length of a string. When the `+` operator is applied to strings it performs *string concatenation*, which is the fancy computer science terms that means "stick them together". To access an individual character in a string you can specify the index in `[]`'s with 0 being the index of the first character. You can also apply Javascript's substr notation to strings to extract a substring. The `.toUpperCase()` and `.toLowerCase()` functions return an upper- and lower-case copy of the string, respectively.

When comparing strings to one another, Javascript performs a lexicographic comparison of the individual characters of the strings moving from the first character to the last, stopping when it finds non-matching characters. So, if comparing "act" and "apple" it will look at the first letter in each word. Since those match it will move to the second letter. Since those are different it will compare those two letters to determine which is "bigger" than the other. Whether one letter is bigger than another is based on the ASCII chart (`http://www.asciitable.com/`. So, Javascript would say that "act" is less than "apple" and that "act" is greater than "APPLE" (because lower and upper case letters have different ASCII values).

The `.replace()` function can match strings or regular expressions. (See section "Regular Expressions below.)

```
1 let str, a, b, c, d, e, f, g
2 str = "There are 53 people with SSN 123-12-1234."
3 a = str.search(/fun/)
4 b = str.search(/re/)
5 c = str.search(/[aeiou]/)
6 d = str.search(/\d/)
7 e = str.search(/\d{3}-\d{2}-\d{4}/)
8 f = str.search(/\s/)
9 g = str.search(/^SSN/)
```

a	-1
b	3
c	2
d	10
e	29
f	5
g	-1

Regular expressions (regex's) are a widely deployed notation for matching patterns in a string of characters. Many tools and languages provide support for regular expressions and in this regard Javascript is not exception. Two places where regex's are use in Javascript are as a search string in the `.search()` function and in the `.replace()` function. We'll focus on the search function for the purposes of these examples. The epressions on lines 2 and 3 show typical usage and results of applying a regex. The regex is delimited using forward slashes (`/`). The pattern specified by the regex is searched for in the string and the function returns the position at which the pattern is first found.

Here are some ways to specify patterns in a regex:

Regex	**Meaning**
`/abc/`	find first `abc`
`/[aeiou]/`	find first vowel; `[]`'s define a character class which means "match anything in this list"
`/[a-zA-Z0-9]/`	find first letter (any case) or digit
`/.\d\w\D\s/`	find any character followed by a digit followed by "word character" followed by non-digit, followed by a white-space character; A period means any character; `\d` is a pre-defined character class for digits; a "word character" (`\w`) can be a letter, a digit, or an underscore
`/\d*x+/`	find 0 or more digits followed by one or more x's
`/\d{3} cows/`	three digits followed by a space followed by the the word "cows"
`/(\w+)@gmail\.com/`	match one or more word characters followed by `@gmail.com`; notice we escaped the period with a backslash so it will match only periods and not any character; the parenthesis are not needed here but can be used with Javascript's `.match()` function to remember and reuse the matched pattern
`/^\d{3} cows$/`	The caret and dollar sign are anchors meaning "starts with" and "ends with" respectively; so this matches strings that start and end with three digits followed by a space followed by the word "cows"

There is a lot more to say about regex's but this should get you started. To practice/learn regex's try: `https://regexone.com`.

```
1  for (let i = 0; i < 3; i++) {
2     console.log(i)
3  }
4
5  let val = 40
6  while (val > 0) {
7     console.log(val)
8     val -= 10
9  }
10
11 let count = 100
12 do {
13    console.log(count)
14    count += 100
15 } while (count <= 400)
```

Expected output:

```
0
1
2
40
30
20
10
100
200
300
400
```

Javascript provides looping structures common to Java/C++.

A *for loop* is commonly used to iterate a fixed number of times. The loop begins with the keyword for followed by three items in parentheses followed by the loop body delimited by braces. The three items in parentheses are: (1) an initialization statement to be performed once before the loop begins, (2) a boolean expression which when true caused the loop body to be executed, and (3) an update statement to be executed each time the loop body completes.

A *while loop* is used to repeat a group of actions and begins with the keyword `while` followed by a boolean expression in parentheses followed by the loop body. The Javascript statements in the loop body are run if the boolean expression evaluates to `true`. When the statements have been executed the boolean expression is revaluated to determine whether or not the statements should be performed again. If the expression evaluates to `false` the statements are skipped. For this reason the boolean expression must contain a variable that gets changed by the statements in the loop (or else suffer the fate of a so-called "infinite loop").

The *do-while* is the same as a while loop except that the loop condition is checked at the bottom of the loop after the body has been executed once.

Arrays

```
let a, b, c, val, str
a = ["zebra", 12, true, 17.3]
a.push("cool")
a[0] = 70
console.log(a)
console.log(a.length)
console.log(a[3])

a.sort()
console.log(a)
val = a.pop()
console.log(val)

b = [20, 30, 40]
for (let elem of b) {
   console.log(elem)
}
str= b.join(",")
console.log(str)
str = "Hi, how are you?"
c = str.split(" ")
console.log(c)
```

Expected output:

```
5
17.3
[12, 17.3, 70, "cool", true]
true
20
30
40
20,30,40
["Hi,", "how", "are", "you?"]
```

Arrays in Javascript are delineated using `[]`'s and can hold values of various types. Each value can be extracted by specifying its index (starting with 0). The `.length` attribute can be used to determine the number of elements in an array. The `.push()` function adds an entry to the end of an array. The `.pop()` function removes the last element of the array and returns it. The `sort()` function re-arranges the list to its natural sorted order.

To visit every elements of an array one can use a traditional for loop using the loop counter as an index. In this example a `for-of` loop cycles through the array having `elem` take on consecutive values from the array. The `.join()` function converts an array to a string by pasting together elements with the specified string. The `split()` function is actually a string method but is demonstrated here because it converts a string to an array by splitting the string on the specified character.

Spread Operator

```
1  // Spread operator on arrays in regular code
2  const nums = [12, 4, 32, 31, 19, 7]
3  console.log(nums)
4  console.log(...nums)
5
6  // Use spread operator to convert array to individual values
       for passing
7  // parameters to functions that require individual values.
8  let ans = Math.min(...nums)
9  console.log("Min is: " + ans)
10
11 // Spread operator on objects in regular code
12 const person = {
13     name: 'Max',
14     age: 29,
15     greet() {
16         console.log('Hi, I am ' + this.name)
17     }
18 }
19
20 // Use spread operator to copy an object (not just its
       reference)
21 const copyOfPerson = {...person}
22 console.log(copyOfPerson)
23
24 // Use of spread operator in parameter list
25 const myMin = (...args) => {
26    let min = args[0]
27    for (let val of args) {
28       if (val < min) {
29          min = val
30       }
31    }
32    return min
33 }
34
35 let ans2 = myMin(6, 20, 9, 34, 3, 22)
36 console.log("Ans: " + ans2)
```

Expected output (to console log):

```
[ 12, 4, 32, 31, 19, 7 ]
12 4 32 31 19 7
Min is: 4
{ name: 'Max', age: 29, greet: [Functio
Ans: 3
```

The spread operator is invoked using three consecutive periods in front of an array or object: `...myVar` The behavior of the operator depends on whether it is invoked in code or in the parameter list of a function. When invoked in code it causes the array or object following it to be broken into its individual pieces. When invoked in a parameter list it causes results in the provided individual parameters to be combined into a single array which can be convenient. In this example we write our own `myMin` function that behaves just like the built-in `min` of the Math library.

```
1  function showSum(a, b) {
2     let c = a + b
3     console.log(c)
4  }
5  function calcSum(a, b) {
6     let c = a + b
7     return c
8  }
9
10 // To use these functions
11 let x = 7.5
12 showSum(x, 1.0)
13 showSum(x * 2.0, x - 2.0)
14 x = calcSum(1.0, 2.2)
15 console.log(x)
```

Expected output:

```
8.5
20.5
3.2
```

Writing a function allows you to give a name to a section of code. The section of code can then be invoked at any time by simply using the name of the function. It is useful to name sections of code to organize a large program or for sections that will need to be invoked multiple times. To define a function start with the keyword `function` followed by the name of the function followed by the (formal) parameter list. Names of functions should be verbs that describe the action of the function. The *parameter list* declares variables that must be filled in order for the function to do its job. In this example the functions specify two parameters named `a` and `b`. The functions must be *called* in order for them to run. This example shows two calls to `showSum()` and one call to `calcSum()`. When calling a function you must specify the *arguments* (or *actual parameters*) that will be used to fill in the formal parameters for that particular call. So, in the first call the parameter `a` takes on the value 7.5 and the parameter `b` takes on the value 1.0. This is an example of *positional arguments* because the values taken left-to-right are plugged in to the parameters left-to-right.

The `calcSum()` function demonstrates a function that returns a value. In function that returns a value you must (1) have a return statement to indicate the value to the returned, and (2) do something with the returned value. In the example, the local variable `c` is returned and the returned value is stored into the variable `x`.

```
1  // Old fashioned JS notation
2  function doubleIt0(val) {
3     return 2 * val
4  }
5
6  // New-fangled JS notation
7  const doubleIt1 = (val) => {
8     return 2 * val
9  }
10
11 // If the function takes only 1 argument then we can leave off
       ()'s.
12 // If the function only has a return statement we can leave
      off {}'s and return.
13 const doubleIt2 = val => 2 * val
14
15 console.log(doubleIt0(14.5))
16 console.log(doubleIt1(14.5))
17 console.log(doubleIt2(14.5))
```

Expected output (in console log):

```
29
29
29
```

Arrow function notation can be confusing if you are used to seeing the `function` keyword and parameter list explicitly provided. Once you are used to it the notation is compact and flexible.

Functions as Parameters

```
1  function doubleIt(x) {
2     return 2 * x
3  }
4
5  let a, b, c, d, e, f, g
6  a = [25, 14, 75, 50, 22]
7  b = a.map(doubleIt)
8  console.log(b)
9  c = a.map(function doubleIt(x) {
10             return 2 * x
11         })
12 d = a.map(function (x) {
13             return 2 * x
14         })
15 e = a.map((x) => { return 2 * x })
16 f = a.map((x) => 2 * x )
17 g = a.map(x => 2 * x)
18 a.forEach((val, i) => console.log(i + " --> " +val))
```

Since we expect `b`, `c`, `d`, `e`, `f`, and `g` to end with the same contents we only display the array `b`. Expected output:

```
[50, 28, 150, 100, 44]
0 --> 25
1 --> 14
2 --> 75
3 --> 50
4 --> 22
```

It is not unusual to pass a function as a parameter to another function. In web programming this is commonly done when a call-back function is needed. The Javascript `array.map()` function is an example of a function that takes another function as a parameter. The `.map()` function is used to apply an action to every element of an array and in doing so, create a new array to store the results.
In the example above we perform the same operation in multiple ways. In all cases the result of calling `.map()` is a new array whose elements are double the values in the original array:

b In the first example we define an external function called `doubleIt` that accepts a single number and returns its value doubled. Then we give the name of the function to the map method. Displaying the new array `b` shows that the function was mapped onto elements of the original array to create the new array.

c This example shows that we can put the function definition as the parameter rather than defining it externally.

d This example shows we can omit the name of the function if we are defining in the parameter list. We call such a function an "anonymous function".

e ES6 allows the use of so called "arrow functions" in which the keyword function is dropped entirely so we keep only the parameter list followed by an arrow followed by the function body.

f–g These final examples show further simplification that can be done in some cases. In particular, if there is only one statement in the function we can omit the braces and return keyword. And, if there is only one parameter we can omit the parentheses. One can see from this concise notation why arrow functions are a popular addition to the language.

The final example shows yet another way to traverse elements of an array by using the `.forEach()` method that takes a function as its parameter. The function accepts two parameters (a value from the array along with that value's index in the array). The function we provide to `.forEach()` in this example uses the shorthand arrow function notation.

```
1  let obj, obj2, str
2  obj = { name: "Fred", age: 10 }
3  obj2 = {
4     name: "Alice",
5     age: 12,
6     display() {
7        return this.name + " is " + this.age + " years old"
8     }
9  }
10
11 console.log(obj.name)
12 console.log(obj['age'])
13 obj.name = "George"
14 console.log(obj.name)
15 str = obj2.display()
16 console.log(str)
```

Expected output:

```
Fred
10
George
Alice is 12 years old
```

An object combines multiple elements of possibly varying types as a single unit. Each element in an object has a property name by which it is accessed. We can access a property using dot notation as on line 11 or by using array-like notation as shown on line 12. There is no concept of private or protected values in an object ... everything is public. Object can also contain functions (methods) as illustrated on line 6–8. Notice that for a function/method inside an object to refer to the object's attributes/properties it requires the use of the keyword `this`. That is, using `name` would be looking for a local variables, but using `this.name` looks for a property of the object.

```
1  class Animal {
2     constructor(name, age) {
3        this.name = name
4        this.age = age
5     }
6     display() {
7        console.log(this.name + " is " + this.age + " years old"
             )
8     }
9  }
10 class Pet extends Animal {
11    constructor(name, age, owner) {
12       super(name,age)
13       this.owner = owner
14    }
15    display() {
16       console.log(this.name + " is owned by " + this.owner)
17    }
18 }
19
20 let wild, pet
21 wild = new Animal("Pyro", 3)
22 wild.display()
23
24 pet = new Pet("Polly", 10, "B.Jones")
25 pet.display()
```

Expected output:

```
Pyro is 3 years old
Polly is owned by B.Jones
```

A class is essentially a prototype of creating objects. ES6 provides a convenient syntax for establishing and using classes. It is conventional for class names to begin with a capital letter as modeled here.

Line 21 shows the syntax for creating a new object based on the pattern/prototype defined by the class definition for `Animal` starting on line 1. The two parameters are passed to the constructor of the class. A constructor is a section of code that is called automatically when a new instance of the class is created. In this case we pass "Pyro" as the name and 3 as the age. In the `Animal` class the constructor establishes two attributes/properties (`name` and `age`) and assigns the values passed to the constructor parameters to them. The class also provides a `.display()` method/function that will write information to the console. As with any object the used of `this` is required in functions/constructors to access properties of the class.

The `Pet` class illustrates the concept of inheritance through the use of the keyword `extends` on line 10. When we say that `Pet` extends `Animal` we immediately inherit (i.e., take on) all of the properties and functions of `Animal`. In this example a `Pet` provides an additional property named `owner`. The constructor is modified from `Animal` to allow 3 parameters. Notice the use of the Java-like notation for calling the constructor of the superclasses on line 12. This statement calls the constructor of the class from which we inherited (which will establish the `name` and `age` properties. This example also demonstrates the possibility of overriding. `Pet` inhertited the `.display()` method from `Animal`, but we have overriden that function with a new one. So, when we call `.display()` on line 25 it calls the overridden function rather than the original.

```
<html>
   <head></head>
   <body>
      <h1>Greetings</h1>
      <div id="mydiv">
         <p>Welcome to this amazing page!</p>
         <p>I hope you like it.</p>
      </div>
      <p id="fun" class="cool neat">This is fun.</p>
      <script>
      // Here we use .innerHTML
         let elem
         elem = document.getElementById("fun")
         elem.style.backgroundColor = "#ccc"
         elem.style.border = "solid 1px black"
         elem.innerHTML = "New paragraph."
         console.log(elem.getAttribute("class"))

      // Here we use .insertAdjacentHTML
        let str = '<p>Great paragraph.</p>'
        elem = document.getElementById("mydiv")
         elem.insertAdjacentHTML('afterbegin',str)
      </script>
   </body>
</html>
```

The code modifies the screen to set the element whose id is "fun" to have a gray background and a border. Also, its contents are changed to hold "New paragraph.". Output to the console is:

```
cool neat
```

When a browser interprets an HTML web page it creates an internal representation of the page as a tree. In this tree the `<html>` element is the root. It has two children: `head` and `body`. Each of those two nodes has children, etc. This mapping of a page to internal memory in the form of a tree is referred to as the Document Object Model (DOM). The DOM provides an API (Application Programming Interface) that allows Javascript to access and manipulate the tree. Anytime the tree is changed there is a corresponding change to what appears on the screen. So, if we remove a branch from the tree that contains an unordered list then that list is not longer part of the page and therefore does not appear on the screen.

Javascript provides a variety of ways to select elements. In this example we show how to select an element by it's id (see line 10). It should be noted that the object returned by the `.getElementById()` function is of type `HTMLElement` which is a large and complex variable with lots of properties and methods. See `https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement` for the full list.

This example demonstrates three properties/methods of an `HTMLElement` object:

style The `style` attribute is itself a large, complex object that gives access to the CSS properties of the element. IMPORTANT: When you modify this `style` property you change the CSS properties of the element and thus cause it to change its appearance on the page (immediately). Notice that the properties in the `style` attribute correspond exactly to CSS property names. The exception to this rule is for CSS properties that contain a dash in the name (e.g., `background-color`). Since a dash cannot be part of a Javascript name the dash is omitted and the first letter of the string following the dash is capitalized (e.g., `backgroundColor`.

innerHTML The `innerHTML` attribute is allows access to the contents of a particular element.

.getAttribute(attr) This method returns the value of the specified attribute for element. There is a corresponding `.setAttribute(attr, val)` method.

Also important to note: When you modify the `.innerHTML` attribute, you are completely replacing all contents inside that element. Sometimes, you simply want to append something rather than replace it. That is where `.insertAdjacentHTML` comes in. This command allows you to add new content to the selected element without removing what is already there. The first parameter must be one of:

- beforebegin: Insert code prior to the selected element (i.e., the new element(s) will be a sibling of the selected element).
- afterbegin: Insert code just inside the selected element, making the new code the first child of the selected element.
- beforeend: New code becomes the last child of the selected element.
- afterend: New code comes after the selected element as its sibling.

```
<html>
   <head></head>
   <body>
      <h1>Greetings</h1>
      <p>Paragraph 1</p>
      <p class="even">Paragraph 2</p>
      <p>Paragraph 3</p>
      <p class="even">Paragraph 4</p>
      <p id="last">Paragraph 5 <span>The end</span></p>
      <script>
         let elem, arr, elem2
         elem = document.querySelector(".even")
         console.log(elem.innerHTML)

         arr = document.querySelectorAll("p")
         console.log(arr.length)

         // give all even paragraphs a grey background
         arr = document.querySelectorAll(".even")
         console.log(arr[1].innerHTML)
         for (elem of arr) {
            elem.style.backgroundColor = "#eee"
         }

         // Alternate way to make a change to all elements
         // that have been selected using querySelectorAll()
         [...document.querySelectorAll('.even')].forEach((
             element) => {
            element.style.color = 'green'
         })

         // search inside of an element
         elem = document.querySelector("#last")
         elem2 = elem.querySelector("span")
         console.log(elem2.innerHTML)

         // there is no span in first paragraph
         elem = document.querySelector("p")
         elem2 = elem.querySelector("span")
         console.log(elem2)
      </script>
   </body>
</html>
```

The even numbered paragraphs will be given a grey background and the console will show these values:

```
Paragraph 2
5
Paragraph 4
The end
null
```

There are a number of ways to select elements in an HTML page. The most flexible of these is to utilize the `.querySelector(sel)` and `.querySelectorAll(sel` methods. These differ from on another in that the former returns the first element that matches the selection while the latter returns an array of elements that match the selection. The parameter (`sel`) can be any valid CSS selector.

Some things to notice in this example:

- `.querySelector()` returns the first matching element only.
- `.querySelectorAll()` returns an array of all matching elements.
- We can select by many CSS selectors (including element, class, and id).
- The `.querySelector()` and `.querySelectorAll()` methods are actually members of the `HTMLElement` class and so can be launched with an already-selected element rather than only with `document`. So, `document.querySelector()` searches the entire HTML document while `elem.querySelector()` only searches in the specified element.
- If there is no matching elements these functions return `null`.
- In the `forEach` example, we convert the array-like result of `.querySelectorAll()` into an array using the spread operator and then re-characterize them as an array using `[]`'s. Then we apply .forEach to cycle through the results.

```
<html>
   <head></head>
   <body>
      <h1>Behold setInterval()</h1>
      <script>
         let colorChanger
         let green = 0

         function changeColor() {
            green = (green + 1) % 256
            document.querySelector("h1").style.color = "rgba
                (0, " + green + ", 0, 1.0)"
            if (green == 0) {
               clearInterval(colorChanger)
            }
         }

         colorChanger = setInterval(changeColor, 10)
      </script>
   </body>
</html>
```

Javascript's `setInterval(recurFunc,millis)` function is an example of a function that accepts another function as a parameter. Calling `setInterval()` will result in the `recurFunc` to be called every `millis` milliseconds. So, the result is a section of code that happens periodically. So, this function highlights two common themes in Javascript programming. First, we are passing a function as a parameter. Second, this function acts *asynchronously*. That is to say, the `setInterval()` function doesn't do some steps and then stop. It keeping running (in the background) while other statements in the program continue to execute. This example uses `setInterval()` to perform an animation in which the text in the `h1` tag cycles from black to green. Upon cycling back around to black we call `clearInterval()` which ends the animation and leaves the `h1` in its final (black) state. If we remove the call to `clearInterval()` then the animation would cycle slowly from black to green and then abruptly return to black and continue this cycle indefinitely.

```
1  <html>
2     <body>
3        Username: <input type="text" name="username"> <br>
4        Password: <input type="password" name="password"
5                         onchange="bewareOfChange()"> <br>
6        <button>Done</button>
7        <script>
8           let textbox, button
9           function bewareOfChange() {
10             alert("A change was made!")
11          }
12          button = document.querySelector('button')
13          button.onclick = () => alert("Done!")
14          textbox = document.querySelector('input[name="
                username"]')
15          textbox.style.color = "#bbb"
16          textbox.value = "Enter username"
17          textbox.addEventListener("focus", function() {
18             textbox.style.color = "#000"
19             textbox.value = ""
20          })
21       </script>
22    </body>
23 </html>
```

Events are actions that occur on a web page. Typically events originate from the action of the user such a moving a mouse, clicking a mouse button, changing a form element, etc. In Javascript we can request that a section of code (i.e., an *event handler*) be executed when a particular event occurs. The process of connecting events with event handlers is called *event registration*. See `https://www.w3schools.com/jsref/dom_obj_event.asp` for a list of events that exist.

The example here demonstrates three ways to perform event registration. The last one mentioned is perhaps the new/preferred way, but there is a lot of existing code that uses the first two methods so it is good to recognize the syntax.

Add attribute to element. Line 5 shows the `change` event being registered for the password box by adding a attribute named `onchange` to the `<input>` element. This is known as *static event registration* and once set in this way cannot be changed via Javascript. This approach also finds disfavor with purists who like to see separation of HTML from CSS/Javascript. In this example we have written a separate, named function and call it when the `change` event occurs.

As with a number of events, the `change` event is not particularly intuitive. According to Javascript a textbox is changed when it takes on a new value and the textbox loses focus (presumably by the user clicking elsewhere).

Assign attribute via code. Lines 12 and 13 shows an event handler for the `click` event being applied to the button. This is an example of *dynamic event registration* which can subsequently be changed via Javascript based on future events. We are essentially giving a function definition to the `onclick` attribute of the button element. We could have written a named function as in the previous case but instead are writing an anonymous function in place. Also note the use of arrow function syntax.

Call .addEventListener(event, func). Lines 17–20 use the `.addEventListener()` method of an element to assign an event handler to the text box for the `focus` event. Here we chose to provide the function definition inline without using arrow function syntax. The code on lines 14–16 sets the contents of the text box to say "Enter username" with light gray letters. When the user brings focus to the text box we empty the box and change the color to black.

```
<html>
   <body>
      <ul id="grocerylist">
         <li><input type="text" value="Eggs" /></li>
         <li><input type="text" value="Bread" /></li>
         <li><input type="text" value="Milk" /></li>
         <li><input type="text" value="Butter" /></li>
         <li><input type="text" value="Cheese" /></li>
      </ul>

      <script>
         var list = document.getElementById("grocerylist")
         list.addEventListener("input", function (event) {
            if (event.target.matches('input[type="text"]')) {
               // call validation function and pass
               // event.target as parameter
            }
         })
      </script>
   </body>
</html>
```

Sometimes you may have a bunch of elements of the same type and you want them to make use of the same handler. This can be done by putting an event handler on a parent element and then in the handler code, check to see if the event happened on one of the items of interest.

In this case we listen for the `input` event on the unordered list. You may observe that a list (ul) isn't a form element an so technically could never trigger the input event. Javascript's default behavior is to propagate an unhandled event to the parent. So, if any of the text boxes within the list generate the input event, that even will "bubble up" (i.e., propagate) to the parent (which is the unordered list).

One cool feature of this approach is that even if we dynamically add text boxes in the list, this handler will still respond to their events!

```
<html>
   <body>
      <form method="post" action="passed.html"
            onsubmit="return formIsValid()">
         Name: <input type="text" name="name">
         <span>Name must have at least 1 character</span><br>
         SSN: <input type="text" name="ssn">
         <span>SSN must be ###-##-####</span><br>
         <input type="submit" value="Done">
      </form>

      <script>
         for (let elem of document.querySelectorAll('span')) {
            elem.style.display="none"
         }
         function formIsValid() {
            let name, ssn, valid
            valid = true
            name = document.querySelector('input[name="name"]'
                )
            if (name.value.search(/^\s*$/) == 0) {
               name.nextElementSibling.style.display="inline"
               valid = false
            }
            else {
               name.nextElementSibling.style.display="none"
            }
            ssn = document.querySelector('input[name="ssn"]')
            if (ssn.value.search(/^\d{3}-\d{2}-\d{4}$/) != 0)
                {
               ssn.nextElementSibling.style.display="inline"
               valid = false
            }
            else {
               ssn.nextElementSibling.style.display="none"
            }
            return valid
         }
      </script>
   </body>
</html>
```

Validating form data on the client side is useful for giving quick feedback to users regarding issues on mechanical form requirements. Some issues to be addressed when validating form data:

- When should the validation occur? (on a per-element basis or when the form is submitted)
- How should the user be notified? (pop-up or by adding content to the page)
- How can we prevent the submit action from occuring when the user clicks a submit button when there is invalid form data?

In this example we choose to validate when the submit button is clicked by attaching an `onsubmit` attribute to the form element. NOTE: Forms are submitted (not form elements). IMPORTANT: The default submit action is to send form elements to the page specified in the `action` attribute. If the form contains invalid data we want to cancel that behavior. This is done by having the function we call return false if an error exists on the form.
We choose to put the error messages right beside the form elements. The code begins by hiding these error messages. Then we define the validation code that is referenced by the `onsubmit` attribute. We require name to have at least one non-space character and we require SSN to take the traditional form. If we find an error we make the error message visible and set a flag accordingly. We return false or true depending on whether there was a validation error.