Intro to Javascript

Variables and Data Structures

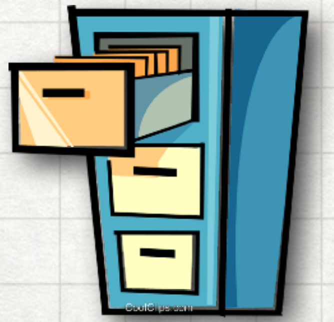
VARIABLES

There is no "free floating" data inside a computer program. All data resides inside containers called ***variables***. When we put data into a variable, we say that we *assign data to a variable*, and the data itself is called the ***value*** of the variable.

We can re-assign new values to an existing variable—hence, the name "variable" itself means *changeable*, *varying*, *shifting*, etc.

The two most basic types of variables in Javascript:

A *scalar variable*, represented by the metaphor of a bucket. Scalar variables can contain zero things or one thing.



A *array variable*, represented by the metaphor of a filing cabinet. The array can have an infinite number of drawers, but each drawer contains zero things or one thing.

VARIABLE NAMES

All variables have *names*, which are just labels. Variable names allow us to use the variable elsewhere in our program. We cannot manipulate or modify or process the data in the variables unless we refer to the variables by its name.

Variable names in Javascript are one-word long and contain alphanumeric characters. *No spaces are allowed!* Variable names can have numbers, but they cannot start with a number. Underscores are acceptable too.

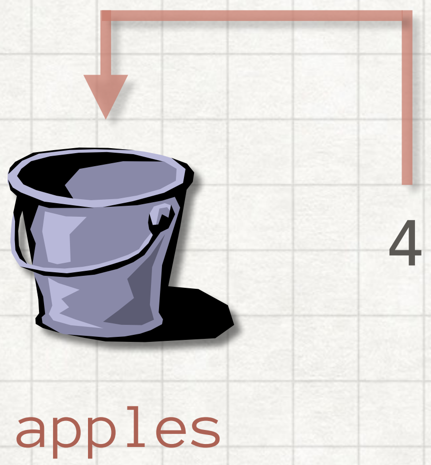
The convention is to use lowercase letters, but uppercase are allowed. *Use descriptive variable names!* Here are some valid variable names:

```
name
id_number
my_favourite_city_in_the_world
cat_2
aCamelCaseVariableName
coffeeshop1234
```

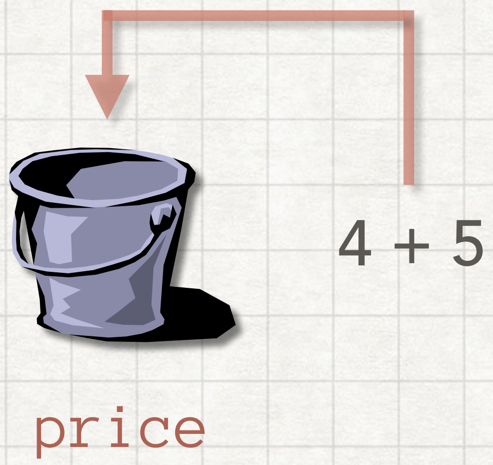
VARIABLE ASSIGNMENT

We assign data to a variable with the = sign. In programming languages, the = sign is a transitive verb that moves data from right to left. It means *figure out what's on the righthand side, no matter how complex, and put it into the variable on the lefthand side.*

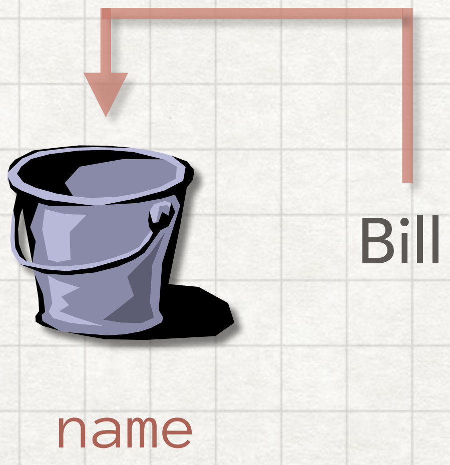
```
apples = 4;
```



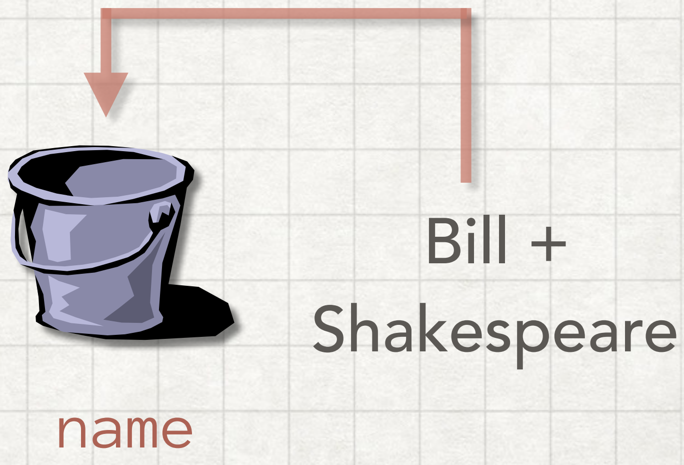
```
price = 4 + 5;
```



```
name = "Bill";
```



```
name = "Bill" +
"Shakespeare";
```



TWO ELEMENTARY DATA TYPES

Computers understand a few types of "raw" or "primitive" data types. The two most elementary are numbers and strings. A *number* is made of digits with an optional sign and decimal point:

```
price = 4.99;
id_number = 7;
tax_rate = 0.05;
temperature = -12.7;
```

A *string* is an *alphanumeric sequence that cannot be changed or simplified*. Strings can contain any letters at all. Javascript uses quotation marks (either singles or doubles—it doesn't matter) to identify the beginning and ending of strings. The quotation marks are metacharacters and are not part of the string itself:

```
name = 'Bill';
cat2 = "Mr. Snuggles";
phone_number = '780-555-1234';
```

VAR

In Javascript, we "declare" a variable with the `var` keyword. This is necessary only the first time we use the variable.*

In Javascript, just like in English, every sentence ends with a semi-colon.†

```
var price = 4.99;
var id_number = 7;
var tax_rate = 0.05;
var temperature = -12.7;
```

```
var first = 'Bill';
var last = 'Shakespeare';
var fullname = first + ' ' + last;
```

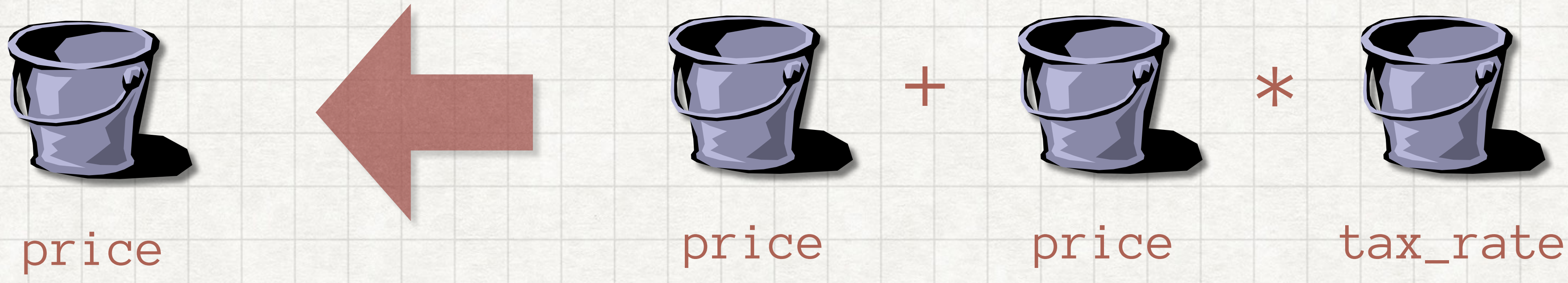
** Omitting `var` doesn't cause an error, but `var` controls a variable's scope, which will become really important to us when we learn functions.*

† Omitting the semi-colon doesn't generate an error, but it's best practice because punctuation helps the Javascript parser interpret our code properly.

OVERWRITING A VARIABLE

It's permissible to reassign a new value to a variable we use on the righthand side. This is called *overwriting a variable* and we will do it all the time.

```
price = price + price * tax_rate;
```



* means multiply

/ means divide

+ means add

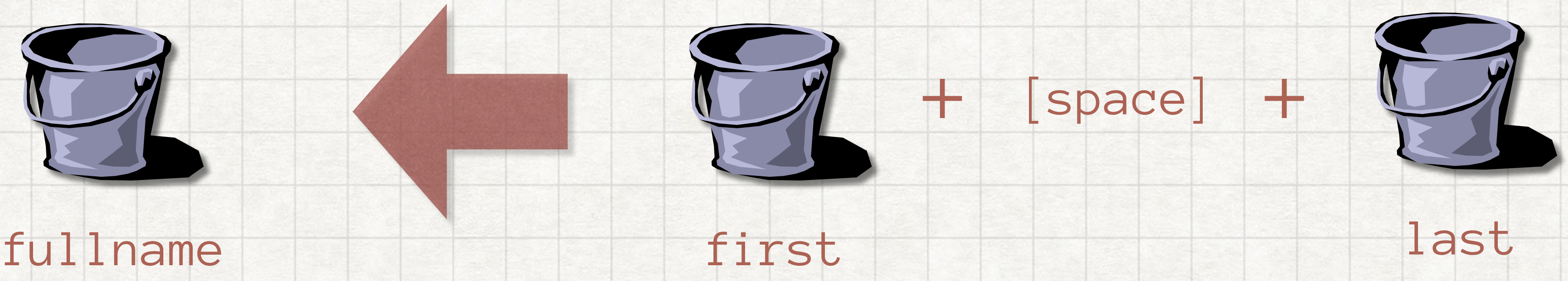
– means subtract

Multiplication and division are always done before addition and subtraction.

CONCATENATION

Javascript doesn't particularly care what's in the variables. Here, we're joining strings together (called *concatenation*):

```
fullname = first + ' ' + last;
```



Notice that the + sign works on both numbers and strings even though adding numbers and concatenating strings are very different actions. Javascript automatically notices the different data types and it behaves appropriately.

Intro to Javascript

Arrays

ARRAYS

A *array* is a collection of data. We'll use the metaphor of a filing cabinet as a way to visualize how arrays work.

The data inside the drawer is called the *value*. The label on each drawer of a array is a number called the *index*. Together, the index and the value comprise an *element*.

We typically start counting indexes from the front and beginning at 0.

values	A	B	C	D	E	F	G	H
indexes	0	1	2	3	4	5	6	7

ARRAYS

We create a new array by putting the values inside square brackets:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
```

values	A	B	C	D	E	F	G	H
indexes	0	1	2	3	4	5	6	7

The square brackets signify that we're building an array. Javascript will add new "drawers" as necessary in order to provide a home for each data value. Javascript will also index the "drawers" with numbers automatically.

ARRAYS

We can also create a new array by telling Javascript to build a new object of type *Array*:

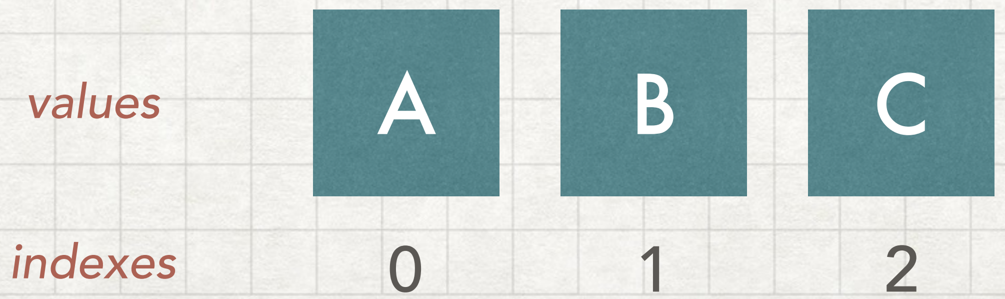
```
letters = new Array;
```

Sometimes you'll see this with parenthetical brackets . . .

```
letters = new Array();
```

. . . because we can send values:

```
letters = new Array('a', 'b', 'c');
```



ARRAYS

Arrays have some interesting properties:

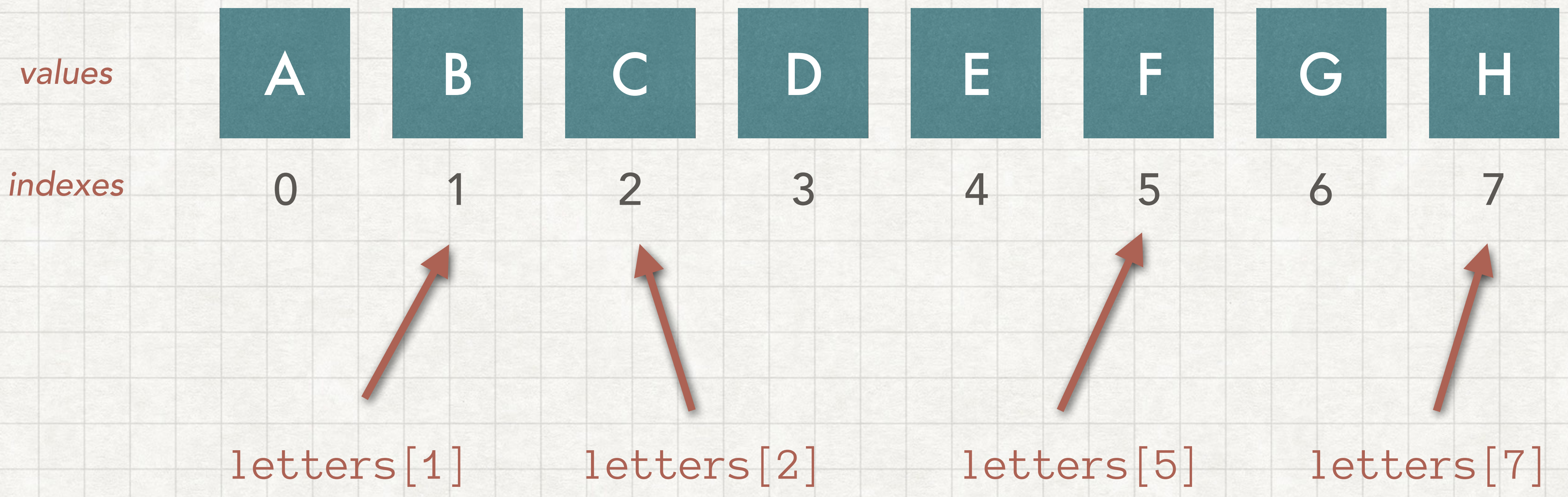
- They can have any number of drawers (even zero!)
- Every drawer contains either zero things or one thing.
- The drawers are always in the same order.
- Each drawer has a label that is a *number* and is called an *index.*
- Drawers are numbered starting at zero.
- Together, an *index* and a *value* comprise an *element*.

For those experienced in Python: Javascript arrays can be indexed directly as they can in Python, but they cannot be sliced. However, Javascript arrays do have a slice() method that allows them to be manipulated like Python lists.

ARRAYS

We refer to any element by putting its index in square brackets after the variable name.

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', ‘H'];
```



```
my_friends_initials = letters[5] + letters[1] + letters[2];
```

Intro to Javascript

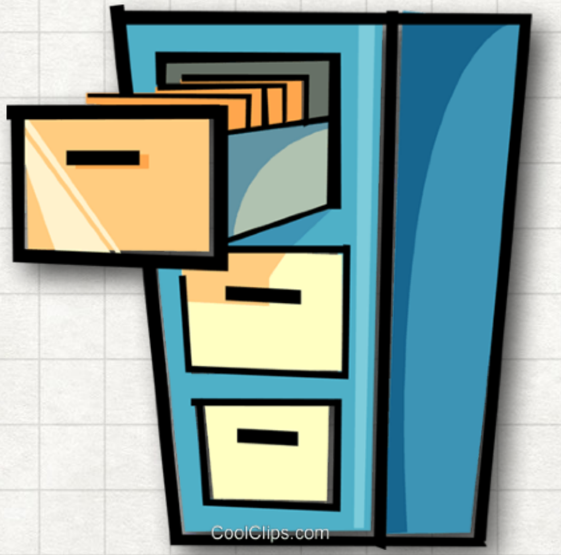
Objects

JAVASCRIPT OBJECTS

Javascript objects are like arrays, but the indexes on the drawers are strings, not numbers. We can put any kind of data inside the drawer, but the labels are always strings.

```
person = {
   'first': 'William',
   'last': 'Shakespeare',
   'birth': 1564,
   'death': 1616,
   'citizenship': 'English'
};
```

Sometimes we call these *key: value* pairs. Each pair creates a new "drawer" in the object. The *key* becomes the label and the *value* becomes the contents of the "drawer." These keys are often called *properties*.



ACCESSING KEY/VALUE PAIRS:

I have two ways to access any of the drawers in the object:

```
person = {
   'first': 'William',
   'last': 'Shakespeare',
   'birth': 1564,
   'death': 1616,
   'citizenship': 'English'
};
```

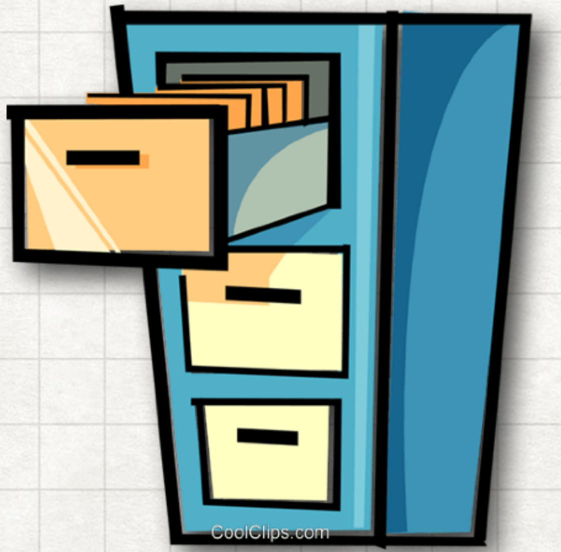
```
      person['first']    or    person.first
       person['last']    or    person.last
      person['birth']    or    person.birth
      person['death']    or    person.death
person['nationality']    or    person.nationality
```

JAVASCRIPT OBJECTS

Here's another example. This is the (over-simplified) structure of a Tweet:

```
tweet = {
 "created_at": "Wed May 30 20:19:24 +0000 1593",
 "id": 1050118621198921728,
 "id_str": "1050118621198921728",
 "text": "William Shakespeare is a fraud!",
 "user": "Kit Marlowe",
 "entities": "#fraud"
};
```

FYI: Christopher Marlowe died mysteriously on 30 May 1593. He was probably murdered. I think he tweeted about the wrong people.



JAVASCRIPT OBJECTS

The differences between an array and an object might seem trivial, but they actually have major implications in Javascript.

- Objects can have any number of drawers (even zero!)
- Every drawer contains either zero things or one thing.
- The drawers are not guaranteed to keep their order.
- Each drawer has a label that is a *string* and is called a *key* or a *property*.
- Later, we'll see that we can put a function inside a drawer.

JSON

This curly bracket, key/value pair data structure `{ key: value }` is called ***Javascript Object Notation***, or ***JSON***. It's probably the most common data interchange format in the world today.

Practically every web service dishes out data in JSON format. Why? Because it's a built-in, inherent data structure in Javascript.

Practically every programming language can output data in JSON format.

KNOW YOUR BRACKETS

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
```

Arrays use square brackets!

```
person = {
    'first': 'William',
    'last': 'Shakespeare',
    'birth': 1564,
    'death': 1616,
    'citizenship': 'English'
};
```

JSON objects use curly brackets!

STRINGS HAVE PROPERTIES

Strings, for example, have a collection of properties. Some give us answers while others perform actions. Properties that perform actions are more technically called *methods* and are names followed by parentheses `()`. We separate the variable name from the method with a dot. When we execute a method, we say that we *call* the method.

```
city = 'Edmonton'
```

`city.length`	8
`city.toUpperCase()`	'EDMONTON'
`city.toLowerCase()`	'edmonton'
`city.startsWith('E')`	True
`city.endsWith('r')`	False
`city.replace('o', 'x')`	'Edmxntxn'
`city.search('m')`	2
`city.substring(2, 5)`	'mon'
`city`	'Edmonton'

Check out the full list: https://www.w3schools.com/jsref/jsref_obj_string.asp

STRINGS AND HTML DISPLAY

Strings can also contain metacharacters like tabs (`\t`) and newlines (`\n`). They are part of a collection of characters colloquially called *whitespace*. I can remove extra whitespace from the beginnings and endings of strings:

```
dirty_string = '\t\n  Hello!   \n'
dirty_string.trim()

'Hello!'
```

However, web browsers ignore whitespace in HTML documents. Browsers clean strings in tags like `<p>` and `<br>`. So you will not always need to clean strings for display in an HTML document, but you will need to clean strings if you're doing text analysis or something.

To retain original string formatting in HTML, use the `<pre>` tag.

ARRAYS HAVE PROPERTIES

In Javascript, arrays are objects too. For more info, check out the full list of array properties and methods: https://www.w3schools.com/jsref/jsref_obj_array.asp

```
letters = ['c', 'b', 'a']
```

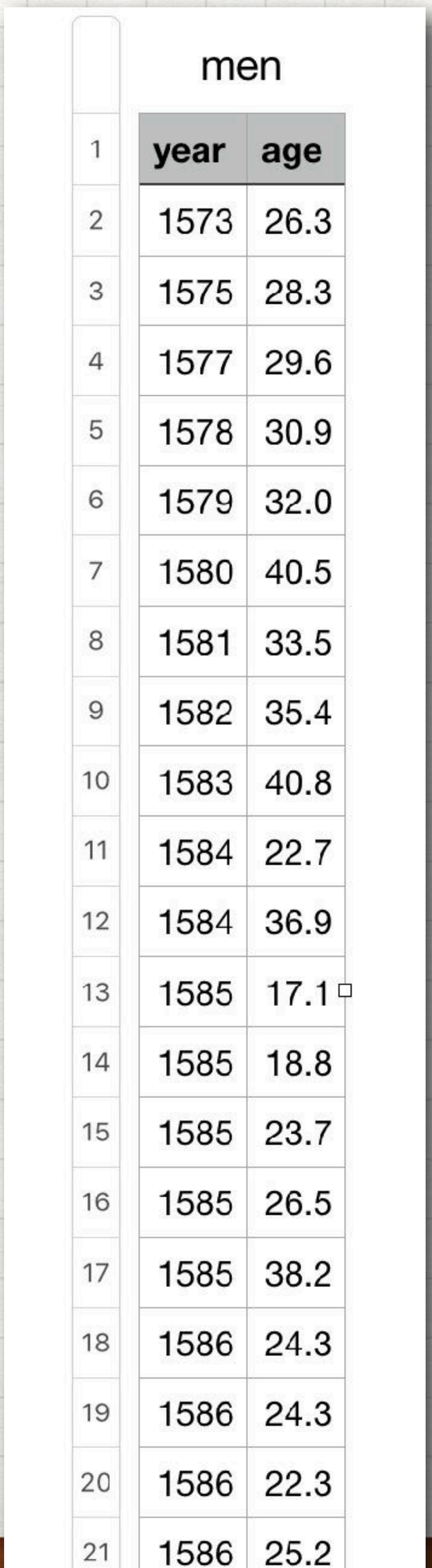
`letters.push('d')`	`['c', 'b', 'a', 'd']`
`letters.sort()`	`['a', 'b', 'c', 'd']`
`letters.pop()`	`'d'`
`letters.indexOf('b')`	`1`
`letters.reverse()`	`['c', 'b', 'a']`
`letters.splice(` *`data`* `)`	*adds/removes array elements*
`letters.forEach()`	*a loop iterator for arrays!*
`letters.join(':')`	`'c:b:a'`
`letters.length`	*3*

*** Note: as with strings,* `length` *is also a property of an array. No parentheses!*

Intro to Javascript

More Complex Data Structures

COMPLEX STRUCTURES



men

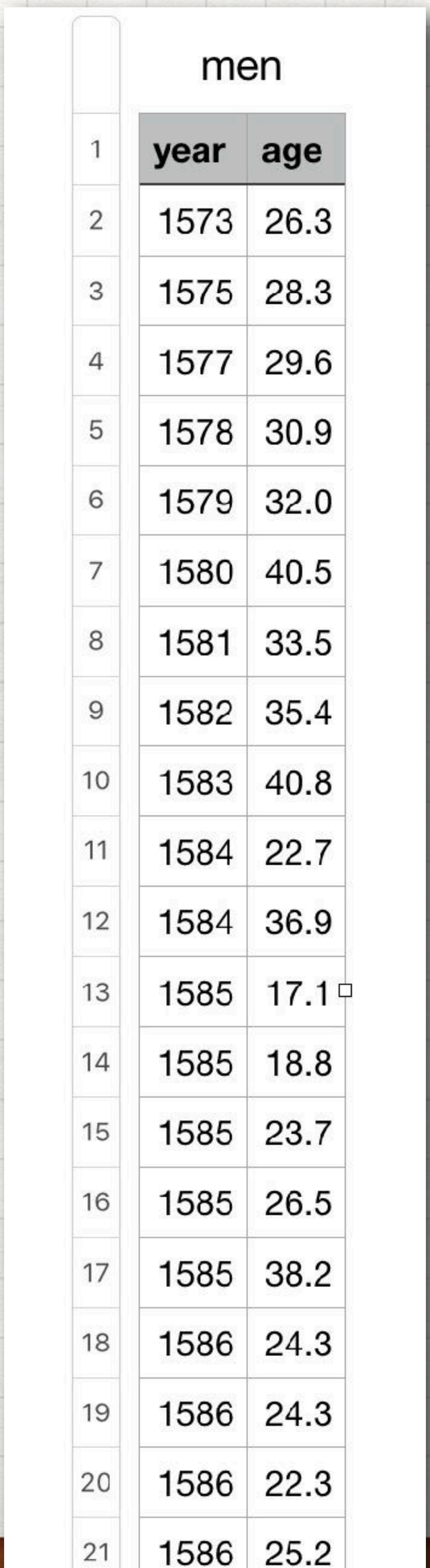
	year	age
1	year	age
2	1573	26.3
3	1575	28.3
4	1577	29.6
5	1578	30.9
6	1579	32.0
7	1580	40.5
8	1581	33.5
9	1582	35.4
10	1583	40.8
11	1584	22.7
12	1584	36.9
13	1585	17.1
14	1585	18.8
15	1585	23.7
16	1585	26.5
17	1585	38.2
18	1586	24.3
19	1586	24.3
20	1586	22.3
21	1586	25.2

Let’s say that I have a spreadsheet that looks like this. My spreadsheet app has assigned sequential row numbers to each row. In Javascript, we’d say it’s a *array*. Here, my spreadsheet app starts at 1. Javascript would start at 0, of course.

The *value* of each element (the content of each drawer) here would be nicely represented by an object of key/value pairs. Each array “drawer” contains a little object “drawer.” We could frame out a array of objects like this:

```
men = [
    { },
    { },
    { }
]
```

COMPLEX STRUCTURES



men

	year	age
1	year	age
2	1573	26.3
3	1575	28.3
4	1577	29.6
5	1578	30.9
6	1579	32.0
7	1580	40.5
8	1581	33.5
9	1582	35.4
10	1583	40.8
11	1584	22.7
12	1584	36.9
13	1585	17.1
14	1585	18.8
15	1585	23.7
16	1585	26.5
17	1585	38.2
18	1586	24.3
19	1586	24.3
20	1586	22.3
21	1586	25.2

And then we can add the data.

```
men = [
   {'year': 1573, 'age': 26.3},
   {'year': 1575, 'age': 28.3},
   {'year': 1577, 'age': 29.6}
]
```

Any given row is just a little object:

```
men[1]  ==> {'year': 1575, 'age': 28.3}
```

To retrieve a “cell,” we access the array index first and then the object key second:

```
men[0]['year'] ==> 1573
men[2]['age'] ==> 29.6
```