DATA STRUCTURES: OBJECTS AND ARRAYS

Numbers, Booleans and strings are the atoms that data structures are built from. Many types of information require more than one atom. Objects allow us to group values - including - other objects to build more complex structures.

Data Structures

Javascript provides a data type specifically for storing sequences of values. It is called an array.

```
let listOfNumbers = [2, 3, 5, 7, 11];
```

Properties The two main ways to access properties in Javascript are with a dot and with square brackets.

Methods

Both string and array values contain in addition to the length property, a no. of properties that hold function values.

```
var person = {
    name: "John Doe"
    sayHello: function () {
        console.log ("hello")
    }
}
```

→ name is the property of object person, it stored string "John Doe" as value; you can access it via dot notation.

→ sayHello is the method of object, and it is a function. Access it using dot notation; person.sayHello()

This below example demonstrates two methods you can use to manipulate arrays.

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence)
// → [1,2,3,4,5]
```

These somewhat silly names are the traditional terms for operations on a stack. A stack is a data structure that allows you to push values into it and pop them out again.

Call stack

A call stack is a mechanism for an interpreter (like the Javascript interpreter in a web browser) to keep track of its place in a script that calls multiple functions.

OBJECTS

Values of the type object are arbitrary collection of properties. One way to create an object is by using braces as an expression.

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
```

```
console.log(day1.wolf);
// → undefined
console.log(day1.squirrel);
// → false
day1.wolf = false;
console.log(day1.wolf);
// → false
```

Properties whose names aren't valid binding names or valid numbers have to be quoted.

```
let descriptions = {
    work: "Went to work",
    "touched tree": "Touched a tree"
};
```

This means that braces have two meanings in Javascript. At the start of a statement, they start a block of statements. In any other position, they describe an object. Fortunately, it's rarely useful to start a statement with an object in braces, so the ambiguity is not much of a problem.

Reading a property that doesn't exist will give you the value undefined.

It is possible to assign a value to property expression with the = operator. This will replace the property's value if it already existed or create a new property on the object if it didn't.

To find out what properties an object has, you can use the Object.keys function

```
console.log (Object.keys ({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

There's an Object.assign function that copies all properties from one object into another.

```
let objectA = {a: 1, b: 2};
Object.assign (objectA, {b: 3, c: 4});
console.log (objectA);
// → {a: 1, b: 3, c: 4}
```

Arrays are just a kind of object specialized for storing sequences of things. If you evaluate typeof [], it produces "object".

MUTABILITY

→ Numbers, strings and Booleans, are all immutable - it is impossible to change values of those types.

→ Objects work differently. You can change their properties, causing a single object value to have diff. content at diff. times.

When you compare objects with Javascript's == operator, it compares by identity: it will produce true only if both objects 'will return false, even if they have identical properties. There is no "deep" comparison operator built into JavaScript, which compares objects by contents, but it will be possible to write it by yourself.

So, Jacques starts up his Javascript interpreter and sets up the environment he needs to keep his journal.

```
let journal = [];
function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
```

Note that the object added to the journal looks a little odd. Instead of declaring properties like events: events, it just gives a property name. This is shorthand and that means the same thing - if a property name in brace notation isn't followed by a value, its value is taken from the binding with the same name.

```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower"],
         false);
```

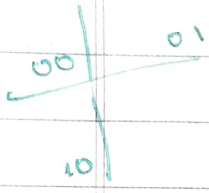
Once ~~has~~ he has enough data points, he intends to use ~~statistics~~ to find out which of these events may be related to the squirrelifications.

To compute the measure of correlation blw two Boolean variables, we can use the phi coefficient (ϕ). This is a formula whose input is a frequency table containing the no. of times the diff. combinations of variables were observed.

Study correlation part again.

To extract a two-by-two table for a specific event from the journal, we must loop over all the entries and tally how many times the event occurs in relation to squirrel transformations.

```
function tableFor(event, journal) {
    let table = [0, 0, 0, 0];
    for (let i = 0; i < journal.length; i++) {
        let entry = journal[i], index = 0;
        if (entry.events.includes(event)) index += 1;
        if (entry.squirrel) index += 2;
        table[index] += 1;
    }
    return table;
}
```




```
console.log(tableFor("pizza", JOURNAL));
```

Arrays have an includes method that checks whether a given value exists in the array. The function uses that to determine whether the event name it is interested in is part of the event list for a given day.

The body in the tableFor figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it's interested in and whether the event happens alongside a squirrel accident. The loop then adds one to the correct box in the table.

We now have the tools to compute individual correlations. The only step remaining is to find a correlation for every type of event that was recorded and see whether anything stands out.

ARRAY LOOPS

In the tableFor function, there's a loop like this:

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // Do something with entry
}
```

This kind of loop is common in classical Javascript - going over arrays one element at a time is something that comes up a lot, and to do that you'd run a counter over the length of the array and pick out each element in turn. There is a simpler way to write such loops in Javascript.

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
```

When a for loop looks like, with the word of after a variable definition, it will loop over the elements of the value given after of. This works not only for arrays but also for strings and some other data structures. We will discuss how it works in chapter 6

THE FINAL ANALYSIS

We need to compute a correlation for every type of event that occurs in the data set. To do that, we first need to find every type of event.

```
function journalEvents(journal) {
  let events = [];
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend"]
```

By going over all the events and adding those that aren't already in there to the events array, the function collects every type of event. Using that, we can see all of the correlations.

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
```

Most correlations seem to lie close to zero. Let's filter the results to show only correlations greater than 0.1 or less than -0.1

```
For (let event of journalEvents (JOURNAL)) {
    let correlation = phi (tableFor (event, JOURNAL));
    if (correlation > 0.1 || correlation < -0.1) {
        console.log (event + ":", correlation);
    }
}

console.log (phi (tableFor ("peanut teeth", JOURNAL)));

// → 1
```

That's a strong result. The phenomenon occurs precisely when Jacques eats peanuts and fails to brush his teeth.

FURTHER ARRAYOLOGY

I want to introduce you to a few more object related concepts. We saw push and pop, which will add and remove elements at the end of an array. The corresponding methods for adding and removing things at the start of an array are called unshift and shift.

```
let todoList = [];
function remember (task) {
    todoList.push (task);
}
function getTask () {
    return todoList.shift ();
}
```

```
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

This program manages a queue of tasks. You add tasks to the end of the queue by calling push. And when you are ready to do something, you call getTask() to get (and remove) the front item from the queue. (shift). The unshift function adds a task to the front.

To search for a specific value, arrays also provide an indexOf method. ~~the method searches~~ found or -1
To search from the end instead of the start, there's a similar method called lastIndexOf

Both indexOf and lastIndexOf take an optional second argument that indicates where to start searching.

Another fundamental array method is slice :

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
```

The concat method can be used to glue arrays together to create a new array, similar to what the + operator does for strings.

The following shows both concat and slice in action.

PTO

```
function remove (array, index) {
    return array.slice (0, index).concat (array.slice (index+1)
                                          );
}

console.log (remove (["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

If you pass to concat an argument that is not an array, that value will be added to the new array as if it were a one-element array.

STRINGS AND THEIR PROPERTIES

We can read properties like length and toUpperCase from string values. But if you try to add a new property, it doesn't stick.

```
Let Kim = "Kim";
Kim.age = 88;
console.log (Kim.age);
// → undefined
```

Values of type string, number, and Boolean are not objects, and though the language doesn't complain if you try to set new properties on them, it doesn't actually store those properties. As mentioned earlier, such values are immutable and cannot be changed.

But these types do have built-in properties. Every string value has a no. of methods. Some very useful ones are slice and indexOf, which resemble the array methods of the same name.

Date: / /

```
console.log ("coconuts".slice (4, 7));
// → nut
console.log ("coconut".indexOf ("u"));
// → 5
```

One difference is that a string's indexOf can search for a string containing more than one character, whereas the corresponding array method looks only for a single element.

```
console.log ("one two three".indexOf ("ee"));
// → 11
```

The trim method removes whitespace (spaces, newlines, tabs and similar characters) from the start and end of a string.

```
console.log ("  okay/n  ".trime());
// → okay
```

The zeroPad function also exists as a method. It is also called padStart and takes the desired length and padding character as arguments.

```
console.log (String(6).padStart(3, "0"));
// → 006
```

You can split a string on every occurence of another string with split and join it again

PTO

A string can be repeated with the repeat method, which creates a new string containing multiple copies of the original string, glued together.

```
console.log ("LA".repeat(3));
// → LALALA
```

We have already seen the string type's length property.

REST PARAMETERS

It can be useful for a function to accept any no. of arguments. For example, Math.max computes the maximum of all the arguments it is given.

To write such a function, you put three dots before the function's last parameter, like this:

```
function max (...numbers) {
    let result = -Infinity;
    for (let number of numbers) {
        if (number > result) result = number;
    }
    return result;
}

console.log (max (4, 1, 9, -2));
```

When such a function is called, the rest parameter is bound to an array containing all further arguments. If there are other parameters before it, their values aren't part of that array. When, as in max, it is the

only parameter, it will hold all arguments.
It is also possible to - max (9, ... numbers);

Square brackets array notation similarly allows the triple-dot operator to spread another array into the new array.

```
let words = ["never", "fully"];
console.log (["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```