

Lecture 9: Objects in JavaScript

Written by Ben Yan, from materials by Jerry Cain

Objects as Aggregates

JavaScript uses the term **object** in a frustratingly and hilariously imprecise way, just like, say, biweekly in the English language. Though **object** more often refers to the encapsulated data collections we'll see in upcoming lectures on object-oriented programming—for now, **object** will refer to any collection of individual data items, which other languages often call a record, dictionary (in Python), or **aggregate**.

In JavaScript, **objects** are akin to **arrays** in that **several independent pieces of data are bundled into a single data structure**. However, they differ in that the elements in an **object** can be, and often are, of differing type—and are identified by a string name (key), as opposed to by a sequence number (index).

Scrooge and Marley Ben and Jerry

Our first example for **objects** imagines keeping track of the data for employees of *Ben and Jerry* (no relation to the ice cream parlor). This is a light, cheeky modernization of *Scrooge and Marley*, which we introduced in lecture as the notorious bookkeeping company from Charles Dicken's *A Christmas Carol*.

Each employee is identified by a name, a job title, and a salary. Two examples—no surprise, Jerry & Ben—are illustrated below. There's no deeper meaning to this, of course. I don't even know what satire is.

name
Jerry Scrooge Cain
title
Professor
salary
£1000

name
Ben Cratchit Yan
title
Teaching Assistant
salary
£25



In case you're wondering why wages are in British pounds and not adjusted for inflation, I got lazy ☺. Besides, it's not like median wages have increased a whole lot in the 2000s (just ask any millennial...)

Here's how one employee would appear in code, as expressed in **JavaScript Object Notation**, or **JSON**.

```
let professor = {
  name: "Jerry Scrooge Cain",
  title: "Professor",
  salary: 1000,
}
```



Pronounced "Jason", but pls don't spell it that way lol

Under the rules of JSON, you specify an **object** simply by listing its contents as a **sequence of key-value pairs**. The key and the value are separated by a colon, the key-value pairs are separated by commas, and the entire collection is wrapped in a warm hug of curly braces. As such, here's the other employee.

```
let assistant = {
  name: "Ben Cratchit Yan",
  title: "Teaching Assistant",
  salary: 25,
};
```



Selecting Fields from an Object

Given an object, you can select an individual field / value in its memory, using an expression that includes the object name, a dot, and then the name of the corresponding key. For instance,

- `professor.name` returns the string "Jerry Scrooge Cain".
- `assistant.salary` returns the number 25, or Ben's measly income in pounds.

Field selection can also be accomplished with square brackets enclosing the key name **expressed as a string**, e.g., `professor["name"]`. This is often necessary **if the key is a variable that depends on the program's behavior**. A playful example I can think of is a `greet` function that addresses a professor by their `title`, but a student / TA by their `name`, e.g., "Hello Professor!" versus "Hello Ben!"

```
const FACULTY = ["Professor", "Chair", "Dean", "Provost", "Headmaster"];

function greet(employee) {
  let howToAddress = "name"; // key is a variable
  if (FACULTY.includes(employee.title)) {
    howToAddress = "title"; // key has been changed!
  }
  console.log("Hello " + employee[howToAddress] + "!");
}
```

Fields are assignable and modifiable. For instance, `assistant.salary *= 2` doubles Ben's salary, while `assistant.salary = 200` updates it to an entirely new amount. **Tastes like promotion!** 🍷😄

With an existing object, **you can also create a new key-value pair through assignment**. For instance, if we wanted to add a `catchphrase` for the employees of *Ben and Jerry*, it may look something like this:

```
professor.catchphrase = "Does that make sense to everyone? ~does T-pose~"
```

Or with the alternative, but equally hip, syntax,

```
assistant["catchphrase"] = "Jerry, we need to cook! ~makes Benzene ring~"
```

It's quite rare that you'd have to do this, but if you need to delete a field from an object for any reason, you can use the aptly named `delete` operator. For instance, if Ben decides his catchphrase is cringe (and it very much is), and needs to be erased, we could do `delete assistant["catchphrase"]`.



Nested Data Structures

Since arrays can contain values of any type, **the elements of an array can be JavaScript objects** (ooh spiky). This is far from uncommon. For example, the employees at *Ben and Jerry* can be initialized as:

```
let employees = [
  {name: "Jerry Scrooge Cain", title: "Professor", salary: 1000},
  {name: "Ben Cratchit Yan", title: "Teaching Assistant", salary: 25},
  {name: "Eugene Francisco", title: "Section Leader", salary: 10},
];
```

I've added a third person, Eugene, just for kicks. Note that despite the camouflage, **the above structure is very much still an array, and so you can index it to access each individual element**. For instance, `employees[0]` fetches the object corresponding to Jerry, so `employees[0].name` returns his name.

In general, **when you drill into nested structures, think one layer at a time**, like you're slowly peeling at an onion. Keep track of what type the current layer of nesting is, and whether to use indices/keys, e.g.

- `employees` is an array, so we use indices to go deeper, e.g., `employees[0]` for Jerry's object,


```
{name: "Jerry Scrooge Cain", title: "Professor", salary: 1000}
```
- `employees[0]` is an object, so we use keys, e.g., `employees[0].name` for this Jerry's name,


```
"Jerry Scrooge Cain"
```
- `employees[0].name` is a string, so we use indices, e.g., `employees[0].name[0]` to return his first initial, which is "J", and `employees[0].name[14]` to return his last initial, i.e. "C"

I also recommend that your variable names be clear & descriptive, to make it transparent what's stored in each variable. You can even comment the type of each variable for your own memory keeping, e.g.,

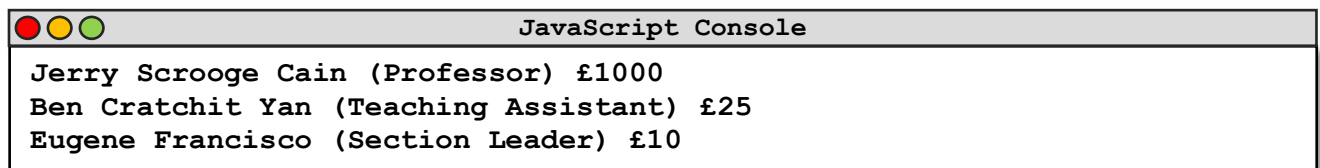
```
let profData = employees[0]; // object with 3 keys: name, title, salary
let profName = profData.name; // string, e.g., "Jerry Scrooge Cain" 😊
```

Print Payroll

The following function prints the payroll for the staff, given an `employees` array as its input argument.

```
function printPayroll(employees) {
  for (let i = 0; i < employees.length; i++) {
    let emp = employees[i];
    console.log(emp.name + " (" + emp.title + ") £" + emp.salary);
  }
}
```

For the example trio of employees listed at the top, `printPayroll(employees)` would display:



```

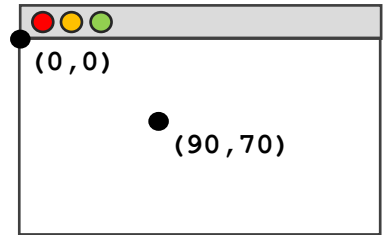
Jerry Scrooge Cain (Professor) £1000
Ben Cratchit Yan (Teaching Assistant) £25
Eugene Francisco (Section Leader) £10
  
```

Point Class and Factory Functions

One data aggregate useful for computer graphics captures **a point in two-dimensional space**, which is composed of an **x** and **y** component.

Points can be readily created in JavaScript with the swift ink of **JSON** object notation. A pair of examples are written below, and are also shown along with their positions in the graphics window to the right.

```
let p1 = { x: 0, y: 0 };
let p2 = { x: 90, y: 70 };
```



The **x** and **y** components of the origin point **p1** can be selected as **p1.x** and **p1.y**, respectively.

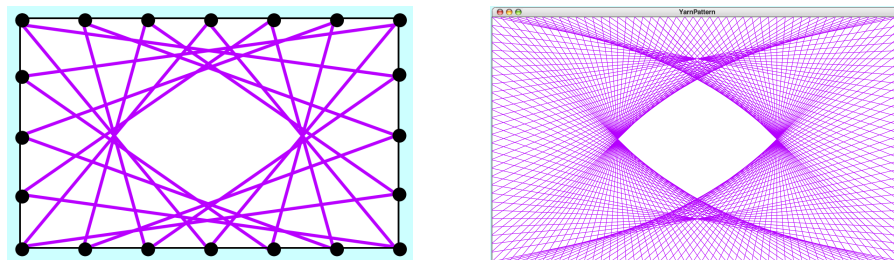
Although **JSON** notation is compact and easy to read, it's often useful to **define a function that creates a JavaScript object** from input parameters. Such functions are called **factories**. For example, this factory function below creates a point-valued object given **x** and **y** coordinates, which default to origin **(0,0)**.

```
function Point(x, y) { // style note: function name is capitalized!
  if (x === undefined) {
    x = 0;
    y = 0;
  }
  return { x: x, y: y }; // e.g., {x: 90, y: 70} for Point(90, 70)
}
```

Note: In the returned object, the **x** on the left is a key name, while the **x** on the right is a specific value.

Yarn Pattern

As an aesthetically pleasing and hopefully inspiring illustration on the use of **Point** objects in graphical applications, we present the **YarnPattern** program, using nothing but straight lines between points to simulate winding a piece of colored yarn around the pegs of a rectangular frame.



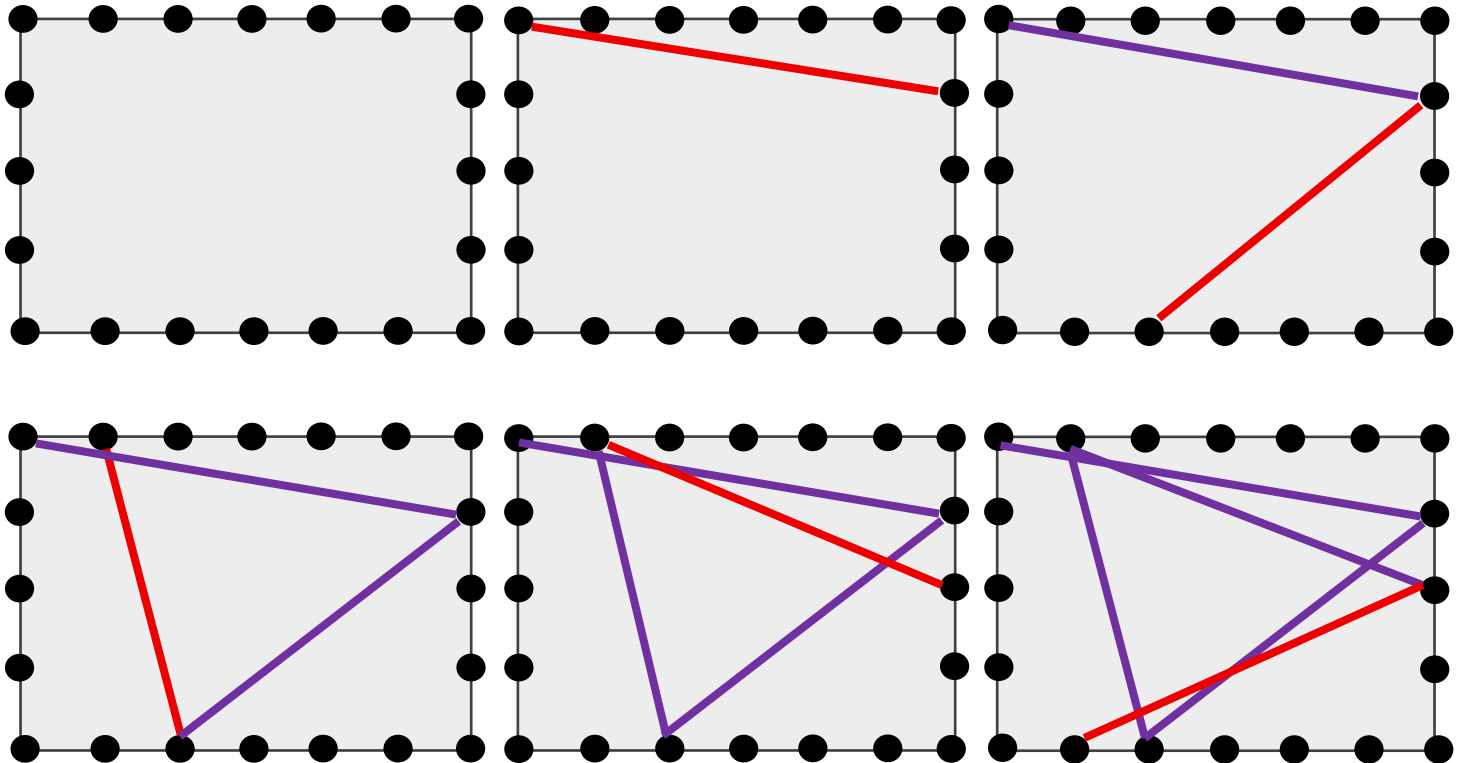
Pictures are shown above for two configurations: **(left)** 6 pegs across, 4 pegs down, each yarn jumping a “delta” of 7 pegs forward, and **(right)** 80 pegs across, 50 pegs down, and each yarn jumping 113 pegs.

We define the following constants, as shown to the right, so that we can easily modify the configuration as we wish, and dynamically create new pictures ☺. None of that Generative AI slop is needed—only our organic, home-grown JavaScript.

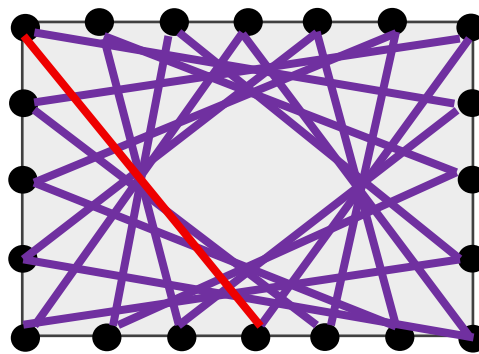
```
const GWINDOW_WIDTH = 1000;
const GWINDOW_HEIGHT = 625;
const N_ACROSS = 80;
const N_DOWN = 50;
const DELTA = 113;
```

Here's the step-by-step process we'll follow to knit this pattern. Take a deep breath ... and here we go!

- Place a set of pegs at regular intervals around the rectangular border of the graphics window.
- Tie a piece of colored yarn around the peg in the upper left corner.
- Stretch the yarn a certain number of pegs (**DELTA**) forward, in clockwise order around the frame, and wrap it around that peg. **The resulting line of yarn is shown in the top middle frame below.**
- Continue stretching forward in leaps of **DELTA** pegs, and bending the yarn into lines, until you close the loop, that is, the yarn has returned to the starting peg in the upper left corner.



The final result glimmers below. The settings are **N_ACROSS** = 6 (number of horizontal pegs **after the first**), **N_DOWN** = 4 (number of vertical pegs **after the first**), **DELTA** = 7. Note the yarn is always a royal purple in color. I just used red to artificially highlight the most recently formed line of yarn ☺.



Creating the Array of Pegs

To begin, we write a function that creates and returns an array of pegs around the perimeter of a rectangle with a specified **width** and **height** in pixels. The upper-left peg is at the origin $(0,0)$, and there are **N_ACROSS** and **N_DOWN** pegs after it in the **x** and **y** dimensions, respectively.

The **x** and **y** spacing between neighboring pegs is given by:

```
let dx = width / nAcross;
let dy = height / nDown;
```

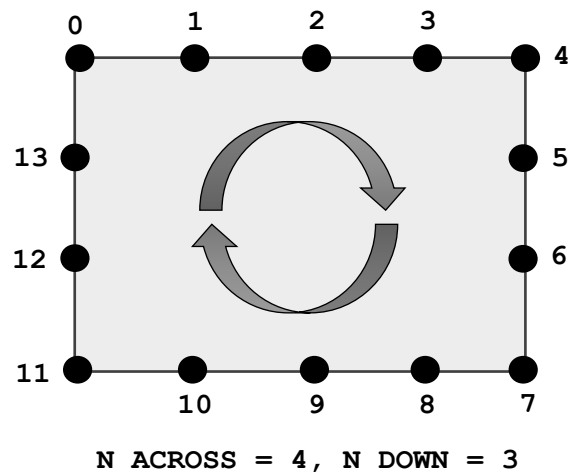
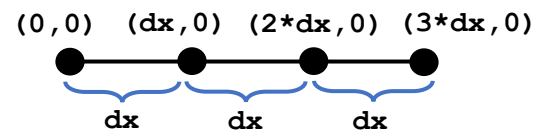
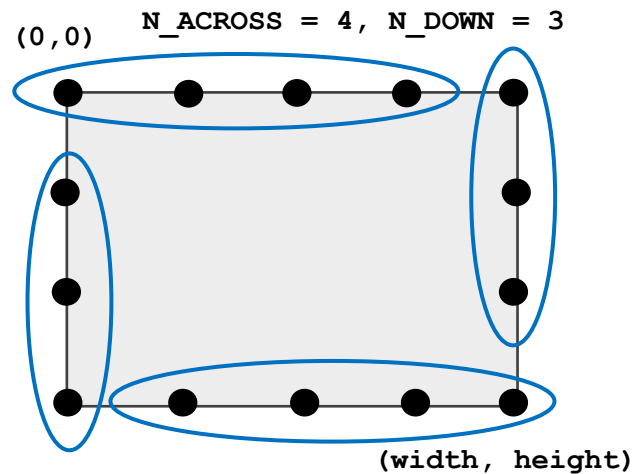
From here, we can initialize an empty array of pegs, and start adding the pegs on the top side (**y** = 0), from left to right.

```
let pegs = []
for (let i = 0; i < nAcross; i++) {
  pegs.push(Point(i * dx, 0));
}
```

Note this loop excludes the rightmost point, which features in the next loop (to add the pegs on the right side). As such, **going in clockwise order around the perimeter**, we can add all pegs in similar fashion.

```
function createPegArray(width, height, nAcross, nDown) {
  let dx = width / nAcross;
  let dy = height / nDown;
  let pegs = [];
  for (let i = 0; i < nAcross; i++) { // pegs on top side, left to right
    pegs.push(Point(i * dx, 0));
  }
  for (let i = 0; i < nDown; i++) { // pegs on right side, top to bottom
    pegs.push(Point(width, i * dy));
  }
  for (let i = nAcross; i > 0; i--) { // pegs on bottom, right to left
    pegs.push(Point(i * dx, height));
  }
  for (let i = nDown; i > 0; i--) { // pegs on left side, bottom to top
    pegs.push(Point(0, i * dy));
  }
  return pegs;
}
```

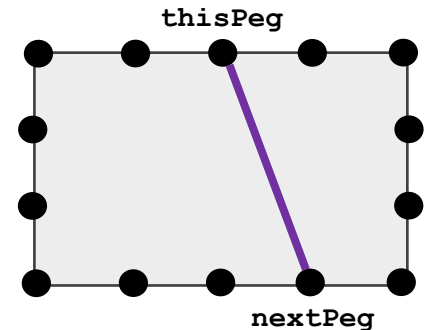
The diagram on the right shows, for the grid example above, the index of each peg in the **pegs** array. Note there are $2 * (N_ACROSS + N_DOWN)$ total pegs.



Winding Yarn Through Two Pegs

We write a helper function that creates a line of magenta purple yarn between two pegs, given their indices, **thisPeg** and **nextPeg**, in an array of **pegs**, and draws it on the graphics window **gw**.

```
function connectWithYarn(gw, pegs, thisPeg, nextPeg)
{
    let p0 = pegs[thisPeg];
    let p1 = pegs[nextPeg];
    let line = GLine(p0.x, p0.y,
                     p1.x, p1.y);
    line.setColor("Magenta");
    gw.add(line);
}
```



Recall that each element in the **pegs** array is a **Point** object, with fields **x** and **y** we can freely access

Winding Yarn Through Entire Frame 🧶

Lastly, we write the **YarnPattern** program itself, using the tools above. We start by creating the graphics window, as well as the **pegs** array, which wraps around the entire perimeter of the window.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let pegs = createPegArray(GWINDOW_WIDTH, GWINDOW_HEIGHT,
                           N_ACROSS, N_DOWN);
```

Next, consider this snippet that moves the yarn **DELTA** pegs ahead of the current index **thisPeg**. The modulo operator **%** is used, so that indices wrap around back to **0** after exceeding **pegs.length**. For instance, if we have **14** pegs, **thisPeg = 13**, **DELTA = 5**, then **nextPeg** is $(13 + 5) \% 14 = 4$.

```
nextPeg = (thisPeg + DELTA) % pegs.length;
connectWithYarn(gw, pegs, thisPeg, nextPeg);
```

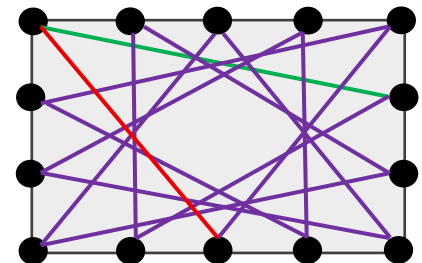
See the diagram on the
previous page for a visual! ↗

We keep doing the above—and replacing **thisPeg** with the index of **nextPeg**, to move the yarn forward for the next iteration—until the yarn returns to the peg at index **0**. Thus, our implementation is:

```
function YarnPattern() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let pegs = createPegArray(GWINDOW_WIDTH,
                              GWINDOW_HEIGHT, N_ACROSS, N_DOWN);
    let thisPeg = 0;
    let nextPeg = -1; // placeholder value

    // weave through the pegs until
    // we get back to where we started
    while (thisPeg !== 0 || nextPeg === -1) {
        nextPeg = (thisPeg + DELTA) % pegs.length;
        connectWithYarn(gw, pegs, thisPeg, nextPeg);
        thisPeg = nextPeg;
    }
}
```

Start of first line!



Start of last line!

In the **while** loop, we include the condition **nextPeg === -1**, so the loop doesn't break right away!

Maps In JavaScript

One of the most important applications of JavaScript objects—besides drawing pretty pictures, of course, as art is the epitome of human expression—is using them to associate **pairs of data values**. These are still **key-value pairs**, but the connotation is the **keys are of the same type and category** (e.g., *colors*, *dorms*, *places*). **The values are generally larger and more complex**, and can even be nested maps themselves.

A classic example is a **dictionary**, and no, not Python dictionaries—but a big, physical, muscular tome like the *Oxford English Dictionary*. The keys are all words; the values are the corresponding definitions.

```
let Dictionary = {
  "syzygy": "3 or more celestial bodies aligned in a straight line",
  "defenestrate": "throw out of a window, or remove from power",
  "cabotage": "not related to sabotage, perhaps surprisingly, but the
     maritime transport of goods or passengers between 2
    places in the same country, by a carrier registered
    in a different country...I'm not making this up.",
  ... more words and definitions - and yes, these are real words ...
}
```

A more contemporary example, and one that befits the digital millennium, is the **World-Wide Web**. The keys are the URLs, and the values are the contents of the corresponding pages the URL transports you to.

Maps are naturally formed in JavaScript through the flesh of object data type. The keys are strings; the values are arbitrary JavaScript values. When you use an object as a map, it's standard to use the square-bracket notation to access values, e.g., `map[key]`, where **key** is expressed as a string variable.

- If the key is defined in the map, the selection `map[key]` returns the value.
- Otherwise, the selection doesn't crash the program, but it does return the constant **undefined**.

Map selections are assignable, so you can set the value associated with a key via `map[key] = value`. For example, if we wanted to include the "Oxford Word of the Year 2023" in our **Dictionary** above,

```
Dictionary["rizz"] = "colloquial noun for style, charm, and charisma 😎"
```

I'm not making this up. I was studying abroad at Oxford when I learned of this, and uh culture shock.

Iterating through Keys in Map

Among the most common operations we perform with map objects is to iterate through their keys. JavaScript supports this operation through a peculiar form of the **for** loop, which is structured as:

```
for (let key in map) {
  let value = map[key];
  ... code to work with the individual key and value ...
}
```

In JavaScript, this **for** loop cycles over keys in any order it chooses. As a note of caution, when you iterate directly over the elements of an array, the syntax is **for (let elem of array)**, not **in** ☹.