Objects & Lists

CF1

Gabe Johnson | October 10, 2024

A few topics today

(this is the best part of the trip, the part I really like)

Objects - a nice packaging of data that works together

Classes - a way to create objects like a factory creates sprockets, and tacks on behaviors as well

Lists - a way to put things in sequence and work on them as a group or individually

Objects!

Objects

Let’s start by contrasting with primitives

- A single value (a “primitive” or “scalar”) value is something that has one part:
    - 42
    - “Forty Two”
    - true
- Primitive values are copied when they are passed around

```
function funcA() {
    let x = 42;
    console.log("in funcA, value starts out as", x);
    funcB(x);
    console.log("back in funcA, the value of x is:", x);
}

function funcB(x) {
    console.log("funcB got", x);
    x = 611;
    console.log("funcB just changed the value to", x);
}
```

What do you think this prints when

we call funcA() ?

Stare at it for a sec.

```
function funcA() {
    let x = 42;
    console.log("in funcA, value starts out as", x);
    funcB(x);
    console.log("back in funcA, the value of x is:", x);
}

function funcB(x) {
    console.log("funcB got", x);
    x = 611;
    console.log("funcB just changed the value to", x);
}
```

```
in funcA, value starts out as 42

funcB got 42

funcB just changed the value to 611

back in funcA, the value of x is: 42
```

Objects

They contain multitudes

- An object is used to hold several related values that make sense as a whole
  - E.g. a point in 2D space has an x and y component
  - Your name has parts - first name, last name, middle name, suffix, etc.

```
function movePoint(pt) {
    pt.x = pt.x + 55;
    pt.y = pt.y + 111;
}

let somePoint = {
    x: 123,
    y: 678,
}

console.log("somePoint (before):", somePoint);
movePoint(somePoint);
console.log("somePoint (after):", somePoint);
```

What do you think this prints?

Stare at it for a sec.

```
function movePoint(pt) {
    pt.x = pt.x + 55;
    pt.y = pt.y + 111;
}

let somePoint = {
    x: 123,
    y: 678,
}

console.log("somePoint (before):", somePoint);
movePoint(somePoint);
console.log("somePoint (after):", somePoint);
```

```
somePoint (before): { x: 123, y: 678 }

somePoint (after): { x: 178, y: 789 }
```

```
function movePoint(pt) {
    pt.x = pt.x + 55;
    pt.y = pt.y + 111;
}

let somePoint = {
    x: 123,
    y: 678,
}

console.log("somePoint (before):", somePoint);
movePoint(somePoint);
console.log("somePoint (after):", somePoint);
```

The things we do to the pt parameter are **persistent** because pt refers to the same memory as somePoint down below. This is the case for objects and lists.

We’ll get to lists later on in this deck.

```
somePoint (before): { x: 123, y: 678 }

somePoint (after): { x: 178, y: 789 }
```

OK, so what?

This means you can let functions do work on your object data

- With primitives, if you need to persist a change, use an assignment operator
- With objects, use assignment operator on its fields
- Also means you have to be careful when modifying object data if you don't want to persist those changes

OK, so what?

This means you can let functions do work on your object data

- If you want to work on a true copy of an object you can use the three dots, called the “spread operator” for some reason:

```
const p1 = { firstName: "James", lastName: "Kirk" }

const p2 = { ...p1 };

p2.firstName = "Jim";

console.log(p1, p2);
```

```
{ firstName: 'James', lastName: 'Kirk' }

{ firstName: 'Jim', lastName: 'Kirk' }
```

Classes!

```
class Boid {
    constructor(x, y, rad, color) {
    // TODO
    }

    draw() {
    // TODO
    }

    move() {
    // TODO
    }
}
```

```
// This is how you use create object instances:

sam = new Boid(1 * quarterW, 1 * quarterH, 25, "#f66");
tweety = new Boid(2 * quarterW, 2 * quarterH, 25, "#8f8");
woodstock = new Boid(3 * quarterW, 3 * quarterH, 25, "#44f");
```

```
// This is how you use object instances:

sam.draw();      // draw all three
tweety.draw();
woodstock.draw();

sam.move();      // then move them
tweety.move();
woodstock.move();
```

Classes are bundles of data and behavior

- The data are called fields, properties, values. In object oriented parlance, they are formally called ‘members’.
- The behaviors are functions that are attached to the class and operate on particular objects. In OO parlance, these functions are called ‘methods’.
- In the Boid example, each Boid has:
    - **data:** include its x and y positions, its radius, and a color.
    - **behavior:** *draw* and *move* methods that operate only on the boid on hand
- Using a class to create an object instance is called ‘instantiating’, using a special function called a ‘constructor’

```
woodstock = new Boid(3 * quarterW, 3 * quarterH, 25, "#44f");
                      ————— x ————  —————— y ——   rad  color
```

When the constructor above is used it makes an object instance with the following member fields:

```
{
  x: 300,
  y: 250,
  rad: 25,
  color: “#44f”,
}
```

```
class Boid {
    constructor(x, y, rad, color) { … }

    draw() { … }

    move() { … }
}
```

Because we used a class, we also can use the behaviors that are attached to instances of that class. In this case, draw() and move().

```
woodstock = new Boid(300, 250, 25, “#44f");
woodstock.draw();
woodstock.move();
```

Boid iteration #1 – three Boids begin a trip

Boid iteration #2 – Boids stay on the screen

Boid iteration #3 – now they notice and chase each other

Boid iteration #4 – improved chasing and flocking

Boid iteration #5 – many Boids in a List (next topic!)

Lists!

Lists

Also known as Arrays

- A List is just what it sounds like: an ordered sequence of things
- In languages like Javascript and Python, Lists are easy mode
- Other languages lists are often low-level and harder to use (but they're fast!)
- You can access things by an index (e.g. "what is item #3 in this list?")
- You can also use methods (the OO kind) to manipulate the list (e.g. "push XYZ onto the end of this list")

Anatomy of a list square brackets with a comma separated list of things inside:

[ ]

[thing1, thing2, thing3, andSoOn]

[trailing, comma, isOk,]

```
const someEmptyList = [];

const colors = ['#f00', '#ff0', '#0af'];

const ages = [7, 47, 47, 71, 76];

const points = [{ x: 60, y: 100 }, { x: 170, y: 2 }];

const fleet = [new Ship(60, 100), new Ship(170, 2)];
```

Access list items with a numeric index inside square brackets:

someList[0]

someList[429]

someList[i]

```
const points = [{ x: 60, y: 100 }, { x: 170, y: 2 }];

const myRide = new Ship(points[0].x, points[0].y);

const spot = points[1];
const myOtherRide = new Ship(spot.x, spot.y);
```

If you try to access an element that doesn’t exist, it will give an error at runtime.

List access starts at element zero - think about it as “distance from the beginning”.

If there are two things in a list, then the valid indices are zero and one.

If there are N things in a list, then the valid indices are zero through N-1 inclusive, but *not N.*

```
const points = [{ x: 60, y: 100 }, { x: 170, y: 2 }];

// Next line errors because there isn’t anything in the
// list at index 2.
const myRide = new Ship(points[2].x, points[2].y); // 💣 💥
```

Lists are often used as a structure to do something with every element.

You can use a for-loop to do this in JS. There are other cooler approaches but this is common across many languages:

```
for (let i=0; i < points.length; i++) {
  const pt = points[i];
  // now use it however you want
  circle(pt.x, pt.y, 10);
}
```

Lists have methods and members just like objects.

counter variable i
starts at zero

someList.length gives you
the number of items in it

increment i after each loop

```
for (let i=0; i < points.length; i++) {
   const pt = points[i];
   // now use it however you want
   circle(pt.x, pt.y, 10);
}
```

access this iteration’s list element with the counter variable

Incidentally, notice that the list elements are objects, so we can access its members using the dot notation.

Another more realistic example (from Boids 5)

```
let boids = [ ]; // initialize an empty list
```

counter variable i starts at zero

iterate as long as counter is less than the list length

increment i after each loop

```
for (let i = 0; i < colors.length; i++) {
  const initialX = random(0, width);
  const initialY = random(0, height);
  const radius = random(10, 60);
  boids.push(new Boid(initialX, initialY,
                       radius, colors[i]));
}
```

access this iteration’s list element with the counter variable

our boids list

invoke the ‘push’ method on our list

the thing we’re pushing is a new instance of the Boid class with these parameters.

This was today

✅ **Objects** - a nice packaging of data that works together

✅ **Classes** - a way to create objects like a factory creates sprockets, and tacks on behaviors as well

✅ **Lists** - a way to put things in sequence and work on them as a group or individually

We'll cover all of this again next week - more on how to use lists, and how to break apart ideas and classes into re-usable files.