Fun Tricks With Objects & Lists

CF1

Gabe Johnson | October 17, 2024

Where “Object Oriented” Shines

- “OO” for those in the know
- Helps you model things in the world / game / art project
- … at the same time it helps organize your code
- It is **very** possible to go overboard with object orientation (e.g inheritance)
- However if you use it effectively, it can help both your mental model and your code quality

Null & Undefined

Let’s talk about black holes for a sec

- Tony Hoare’s ‘billion dollar mistake’ (interesting article! click!)
- Most data types have a special “nothing” value, often a default:
    - Booleans: false
    - Strings: empty string
    - Integers: zero
- What about objects? What should the default value of a Person object be?

Null & Undefined

Let’s talk about black holes for a sec

```
let p1 = new Person(“Victor”); // initialize p1 to known value
p1.sayHello();                 // probably no problem here
let p2;                        // declare but don’t initialize p2
p2.sayHello();                 // Uh…
```

Null & Undefined

```
> let p2;
<· undefined
> p2.sayHello();
✖ ▶Uncaught                VM78464:1
  TypeError: Cannot read
  properties of undefined
  (reading 'sayHello')
      at <anonymous>:1:4
> 😭
```

This is typically the message you get when accessing fields on a null object in Javascript.

Many other languages have similar behavior. Some languages just crash! (C, C++)

Some enlightened languages take steps to help prevent you from making this mistake (Typescript, Dart).

Some languages are designed to prevent null references entirely (Rust).

Uses of Objects

Example 1: Social App

What are the ‘Things’ we might turn into classes?

- People
- Message Boards
- Messages
- Images
- Tags

Example 2: Space War Game

What are the ‘Things’ we might turn into classes?

- Ships
- Asteroids, Planets, Stars
- Projectiles

Example 3: Public Interactive Art Display

What are the 'Things' we might turn into classes?

- People
- Environmental: Light, Wind, Temperature
- Interactions: Gestures, Presense, Touches
- Display output: Lights, Sounds, Mechanisms

What’s to be gained by objectifying data?

- Mental models.
- We all know what a ‘Person’ is (or do we? 🤔)
- We know what a ‘Person’ can do (think, climb trees, eat cookies…)
- … and the kind of data involved (name, SSN, grade in 8th grade English…)
- Easier to reason about ‘Person’ objects by name than it is to reason about collections of data about people.

Polymorphism

Polymorphism

Fancy Word, Simple Idea

- If several objects have common behaviors you can use them in those terms
- Example: say we have classes for Circle, Rectangle, and Triangle
  - They all have *draw*, *computeArea* and *move* functions
  - So we can think of that set of behavior as an *Interface*
  - We don't need to know which class of thing we're working with, as long as it has that draw/computeArea/move interface, we can use those methods

Polymorphism

Fancy Word, Simple Idea

- In the Boids example, we could create an entirely second (third, fourth, forty seventh, etc) class that has the same methods as the Boid class
- … and we could use them in the same way
- Each class's behavior would likely be different, but the way we use it is the same. For example:
  - "ScaredBoid" that avoids other Boids
  - "FollowerBoid" that attempts to follow another Boid at some distance
  - "PickyBoid" that behaves differently depending on which color Boids are nearby

Polymorphism

Fancy Word, Simple Idea

- In other languages like Typescript, Python, and Java, you can formalize an interface and give it a name. **Here's Typescript**, which is basically Javascript with type safety added:

```
interface Boid {
    draw: VoidFunction;
    move: VoidFunction;
    notice: VoidFunction;
}
```

Polymorphism

Fancy Word, Simple Idea

- Here, VoidFunction means “a function that takes no arguments and returns no value”.
- So draw, move, and notice are expected to all be void functions.

```
interface Boid {
    draw: VoidFunction;
    move: VoidFunction;
    notice: VoidFunction;
}
```

Polymorphism

Once you've defined an interface you can implement it by name

```
class ScaredBoid implements Boid {
    constructor() { ... }
    draw() { ... }
    move() { ... }
    notice(other: Boid) { ... }
}
```

```
class FollowerBoid implements Boid {
    constructor() { ... }
    draw() { ... }
    move() { ... }
    notice(other: Boid) { ... }
}
```

```
class PickyBoid implements Boid {
    constructor() { ... }
    draw() { ... }
    move() { ... }
    notice(other: Boid) { ... }
}
```

Polymorphism

No Typescript needed. Be polymorphic in plain Javascript

```
class Square {
    constructor(sideLength) { this.sideLength = sideLength; }
    getArea() { return this.sideLength * this.sideLength; }
}

class Circle {
    constructor(radius) { this.radius = radius; }
    getArea() { return Math.PI * this.radius * this.radius; }
}
```

Both classes have a method called getArea so we can use them both as ‘area-able’ types. As long as we know *that* we don’t need to know which class they are.

Polymorphism

No Typescript needed. Be polymorphic in plain Javascript

```
const s = new Square(5);
const c = new Circle(5);

const shapes = [s, c];

for (let i = 0; i < shapes.length; i++) {
    console.log(shapes[i].getArea());
}
```

Here we make a couple shapes - things that have a `getArea` method - and iterate through a list of them.

Array Methods

Array Methods

More expressive ways to work with lists

```
for (let i = 0; i < shapes.length; i++) {
    console.log(shapes[i].getArea());
}

shapes.forEach(shape => console.log(shape.getArea()));
```

These two are equivalent.

Javascript lists are technically Array objects and they have lots of useful methods to help you iterate over their data, transform, and query them.

Array Methods - they take a ‘callback’

Looks funky but you’ll learn to love it

```
shapes.forEach(shape => console.log(shape.getArea()));
```

This is a callback. It is a little function that you define in-line (no ‘function’ keyword needed) that will be called for each item in the list.

```
console.log(shapes[0].getArea());
console.log(shapes[1].getArea());
console.log(shapes[2].getArea());
```

These two are equivalent (assuming length = 3).

We can ‘unroll’ the forEach statement by explicitly doing what the callback function does for each subsequent index.

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
myList.filter(f)
```

Returns a new list that only includes items from myList that cause the filter function f to return true.

```
[1, 2, 3, 4, 5, 6].filter(v => v % 1);

//     => [1, 3, 5]
```

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
const raw = ['🥔', '🐮', '🥬', '🐷'];

const isVeg = (v) => {
    if (v === '🥔' || v === '🥬') return true;
    else return false;
}

const veggies = raw.filter(isVeg); // [ '🥔', '🥬' ]
```

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
myList.map(f)
```

Returns a new list that transforms each item in the list to a corresponding result value.

```
[1, 2, 3, 4, 5, 6].map(v => v * v)

//      => [1, 4, 9, 16, 25, 36]
```

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
const raw = ['🥔', '🐮', '🥬', '🐷'];

const cook = (v) => {
    if (v === '🥔') return '🍟';
    if (v === '🐮') return '🍔';
    if (v === '🥬') return '🥗';
    if (v === '🐷') return '🥓';
}

const cooked = raw.map(cook); // ['🍟','🍔','🥗','🥓']
```

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
myList.reduce(f, v0)
```

Reduces the entire list into a single value based on an initial value v0 and a reduce function.

```
[1, 2, 3, 4, 5, 6].reduce((v, sum) => v + sum, 0)

//      => 21
```

A Sophomoric (and memorable) example

Filter, Map, Reduce

```
const cooked = raw.map(cook); // ['🍟','🍔','🥗','🥓']

eat = ((v, a) => '💩');

const eatenAll = cooked.reduce(eat); // '💩'
```

Array Methods Help You Think

- Once you get the hang of using array methods and writing callbacks, it is easy *and expressive* to operate on lists of data
- Arbitrary Example: I have a list of points that could be anywhere, in any order. I only want to consider those with an X range of 10 to 50 and Y in a range of 0 to 100. Of those, what is the sum of the Y values?

I only want to consider those with an X range of 10 to 50 and Y in a range of 0 to 100. Of those, what is the sum of the Y values?

```
const points = [
    { x: -10, y: -10 }, // 🚫 x out of range
    { x: 30, y: 30 },   // ✅
    { x: 40, y: 80 },   // ✅
    { x: 70, y: 70 }    // 🚫 x out of range
];

const sumOfYInZone = points
    .filter(pt => pt.x >= 10 && pt.x <= 50 && pt.y >= 0 && pt.y <= 100)
    .map(pt => pt.y)
    .reduce((v, sum) => sum + v, 0);

console.log("Sum of y value in region is", sumOfYInZone); // 110
```

Filter

```
const points = [
    { x: -10, y: -10 }, // 🚫 x out of range
    { x: 30, y: 30 },   // ✅
    { x: 40, y: 80 },   // ✅
    { x: 70, y: 70 }    // 🚫 x out of range
];

const inRange = points
    .filter(pt => pt.x >= 10 && pt.x <= 50 && pt.y >= 0 && pt.y <= 100);

inRange is now: [
    { x: 30, y: 30 },   // ✅
    { x: 40, y: 80 },   // ✅
];
```

Map

```
const inRange = [
    { x: 30, y: 30 },   // ✅
    { x: 40, y: 80 },   // ✅
];

const onlyY = inRange.map(pt => pt.y);

onlyY is now: [ 30, 80 ]
```

Reduce

```
const onlyY = [ 30, 80 ];

const sumOfY = onlyY.reduce((v, sum) => sum + v, 0);

sumOfY is now: 110
```

Reduce

Two arguments

```
const onlyY = [ 30, 80 ];

const sumOfY = onlyY.reduce((v, sum) => sum + v, 0);

sumOfY is now: 110
```

A reduce function

An initial value

Reduce

The ‘sum’ value will use the initial value on the first round.

The ‘v’ value will be a value from the array. So 30, then 80.

The reduce function returns sum + v each time, and that value is used as the sum in the next round.

```
const onlyY = [ 30, 80 ];

const sumOfY = onlyY.reduce((v, sum) => sum + v, 0);

sumOfY is now: 110
```

A reduce function

An initial value

The original code. Can you follow the chain of array methods now?

```
const points = [
    { x: -10, y: -10 }, // 🚫 x out of range
    { x: 30, y: 30 },   // ✅
    { x: 40, y: 80 },   // ✅
    { x: 70, y: 70 }    // 🚫 x out of range
];

const sumOfYInZone = points
    .filter(pt => pt.x >= 10 && pt.x <= 50 && pt.y >= 0 && pt.y <= 100)
    .map(pt => pt.y)
    .reduce((v, sum) => sum + v, 0);

console.log("Sum of y value in region is", sumOfYInZone); // 110
```

Destructuring

Destructure Objects and Arrays

Shortcuts for accessing object properties and elements of arrays

There are a few flavors of this:

1. On assignment: `const { x, y } = myObject;`
2. On function execution: `({x, y}) => { /* code */ };`
3. To copy: `const voltronAtOrigin = {...myObject, x: 0, y: 0};`

Object Destructuring

On Assignment

```
const badGuy = {
    x: 733,
    y: 394,
    hitpoints: 900,
    defense: 100,
    damage: 200,
};
```

Object Destructuring

On Assignment

```
const badGuy = {
    x: 733,
    y: 394,
    hitpoints: 900,
    defense: 100,
    damage: 200,
};
```

```
const { x, y } = badGuy;
console.log("x and y:", x, y); // x and y: 733 394
```

Array Destructuring

On Assignment

```
const players = ['rock', 'paper', 'scissors'];


const [p1, p2] = players;
console.log('p1 and p2:', p1, p2); // p1 and p2: rock paper
```

Object Destructuring

On Function Execution

This doesn't use destructuring - this is how we've been doing it.

```
// If we write a function that will take an object, we can do it the obvious
// way like this, without destructuring:
function oldSchool(someObject) {
    const x = someObject.x;
    const y = someObject.y;
    // now do stuff with x and y
    console.log('oldSchool has x and y:', x, y);
}

oldSchool(badGuy); // oldSchool has x and y: 733 394
```

Object Destructuring

On Function Execution

Destructuring!

```
// A different and often more expressive way of doing this is to destructure
// the object on entry to the function:
function newFangled({ x, y }) {
    console.log('newFangled has x and y:', x, y);
}



newFangled(badGuy); // newFangled has x and y: 733 394
```

Array Destructuring
On Function Execution

This doesn't use destructuring - this is how we've been doing it.

```
function oldSchoolFromArray(combatants) {
    const p1 = combatants[0];
    const p2 = combatants[1];
    console.log("oldSchoolFromArray:", p1, p2);
}

oldSchoolFromArray(players); // oldSchoolFromArray: rock paper
```

Array Destructuring

On Function Execution

Destructuring!

```
function newFangledFromArray([p1, p2]) {
    console.log("newFangledFromArray:", p1, p2);
}
newFangledFromArray(players); // newFangledFromArray: rock paper
```

Object Copy

With the spread operator (three dots …)

Last, you can use the 'spread' operator to create new objects and arrays.

```
const copyOfBadGuy = { ...badGuy };
console.log("copyOfBadGuy should be the same as the original badGuy:");
console.log('  badGuy:     ', badGuy);
console.log('  copyOfBadGuy', copyOfBadGuy);
```

What will this print?

Object Copy

With the spread operator (three dots …)

Last, you can use the 'spread' operator to create new objects and arrays.

```
const copyOfBadGuy = { ...badGuy };
console.log("copyOfBadGuy should be the same as the original badGuy:");
console.log('  badGuy:     ', badGuy);
console.log('  copyOfBadGuy', copyOfBadGuy);
```

```
copyOfBadGuy should be the same as the original badGuy:
  badGuy:      { x: 733, y: 394, hitpoints: 900, defense: 100, damage: 200 }
  copyOfBadGuy { x: 733, y: 394, hitpoints: 900, defense: 100, damage: 200 }
```

Object Copy

With the spread operator (three dots …)

If there is more than one object, or if there is an object with other key/value pairs, it will combine things from left to right. In the case of objects, it will overwrite values that appear later in the expression.

```
const buffedBadGuy = { ...badGuy, hitPoints: badGuy.hitpoints + 200, defense: 200 };
console.log("buffedBadGuy should have more HP and defense than the original badGuy:");
console.log('  badGuy:     ', badGuy);
console.log('  buffedBadGuy', buffedBadGuy);
```

What will this print?

Object Copy

With the spread operator (three dots ...)

If there is more than one object, or if there is an object with other key/value pairs, it will combine things from left to right. In the case of objects, it will overwrite values that appear later in the expression.

```
const buffedBadGuy = { ...badGuy, hitPoints: badGuy.hitpoints + 200, defense: 200 };
console.log("buffedBadGuy should have more HP and defense than the original badGuy:");
console.log('  badGuy:     ', badGuy);
console.log('  buffedBadGuy', buffedBadGuy);
```

```
buffedBadGuy should have more HP and defense than the original badGuy:
  badGuy:      { x: 733, y: 394, hitpoints: 900, defense: 100, damage: 200 }
  buffedBadGuy { x: 733, y: 394, hitpoints: 1100, defense: 200, damage: 200 }
```

Destructuring to get named parameters

Very useful, very common

You can use this to make 'named' parameters to functions by expecting an object argument, and then destructuring everything by name.

```
function calculateDistanceToOrigin({ x, y }) {
    return Math.sqrt(x * x + y * y);
}
const dist = calculateDistanceToOrigin({ x: 4, y: 3 });
console.log("Distance from (4, 3) to the origin:", dist);
```

What will this print?

Destructuring to get named parameters

Very useful, very common

You can use this to make 'named' parameters to functions by expecting an object argument, and then destructuring everything by name.

```
function calculateDistanceToOrigin({ x, y }) {
    return Math.sqrt(x * x + y * y);
}
const dist = calculateDistanceToOrigin({ x: 4, y: 3 });
console.log("Distance from (4, 3) to the origin:", dist);
```

```
buffedBadGuy should have more HP and defense than the original badGuy:
  badGuy:      { x: 733, y: 394, hitpoints: 900, defense: 100, damage: 200 }
  buffedBadGuy { x: 733, y: 394, hitpoints: 1100, defense: 200, damage: 200 }
```

Last thing:

Watch the Functional Bros video I linked on Canvas.
20 minutes or so. See if you can follow along.