# Traversing Nested Objects

[(https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-traversing-nested-objects)](https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-traversing-nested-objects) [(https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-traversing-nested-objects/issues/new)](https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-traversing-nested-objects/issues/new)

## Learning Goals

- Revisit why nested objects are useful
- Practice accessing inner properties
- **Bonus**: Use recursion to iterate over nested objects and arrays

## Introduction

You've just been onboarded to the dev team working on Flatbook, the world's premier Flatiron School-based social network. Here at Flatbook, we have some pretty complex data-modeling needs. For instance, think about the breadth of information we might want to display on each user's profile page:

- First name
- Last name
- Employer
  - Company name
  - Job title
- Friends
  - First name
  - Last name
  - Employer
  - Company name
  - Job title
- Projects
  - Title

- Description

We can already start to see some problems with trying to fit all of this into a *shallow* (non-nested) JavaScript object:

```
const userInfo = {
  firstName: "Avi",
  lastName: "Flombaum",
  companyName: "Flatbook Labs",
  jobTitle: "Developer Apprentice",
  friend1firstName: "Nancy",
  friend1lastName: "Burgess",
  friend1companyName: "Flatbook Labs",
  friend1jobTitle: "Developer Apprentice",
  friend2firstName: "Corinna",
  friend2lastName: "Jackson",
  friend2companyName: "Flatbook Labs",
  friend2jobTitle: "Senior Developer",
  project1title: "Flatbook",
  project1description:
    "The premier Flatiron School-based social network in the world.",
  project2title: "Scuber",
  project2description:
    "A burgeoning startup helping busy parents transport their children to and from all of their activities on scooter
};
```

Goodness, that's messy. It would be a nightmare to keep the object updated. If Avi un-friends Nancy, do we shift Corinna's info into the `friend1...` slots and delete the `friend2...` properties, or do we leave Corinna as `friend2...` and delete the `friend1...` properties? There are no good answers. Except...

## Objects in Objects

Recall from the lesson on objects that the values in an object can be *anything*, including another object. If we reorganize the above object a bit, it becomes significantly easier to read and update:

```
const userInfo = {
  firstName: "Avi",
  lastName: "Flombaum",
  company: {
    name: "Flatbook Labs",
    jobTitle: "Developer Apprentice",
  },
  friends: [
    {
      firstName: "Nancy",
      lastName: "Burgess",
      company: {
        name: "Flatbook Labs",
        jobTitle: "Developer Apprentice",
      },
    },
    {
      firstName: "Corinna",
      lastName: "Jackson",
      company: {
        name: "Flatbook Labs",
        jobTitle: "Lead Developer",
      },
    },
  ],
  projects: [
    {
      title: "Flatbook",
      description:
        "The premier Flatiron School-based social network in the world.",
    },
    {
      title: "Scuber",
      description:
        "A burgeoning startup helping busy parents transport their children to and from all of their activities on scc
    },
  ],
```

We've pared the sixteen messy properties in our first attempt down to a svelte five: `firstName` , `lastName` , `company` , `friends` , and `projects` . `company` points at another object, and both `friends` and `projects` point to arrays of objects. Let's practice accessing some of those beautifully nested data points. Copy `userInfo` into the [replit (https://replit.com/languages/javascript)](https://replit.com/languages/javascript) code window and follow along. Once you click run, you can check the values of the variable's properties in the console window.

To review, for a property at the top level of our object, we can grab a value using dot notation:

```
userInfo.lastName;
//=> "Flombaum"
```

If the property we're accessing is nested inside another object, we just append the additional key(s):

```
userInfo.company.jobTitle;
//=> "Developer Apprentice"
```

If the property is nested inside an array, we need to specify the index in the array for the object that we want. To get the first name of Avi's first friend and the title of his second project:

```
userInfo.friends[0].firstName;
//=> "Nancy"

userInfo.projects[1].title;
//=> "Scuber"
```

It's worth spending some time getting comfortable with nested data structures — you will see a lot of them as you proceed through the curriculum and in your career as a developer. Create your own in the REPL and practice accessing various pieces of data.

## Arrays in arrays

Working with nested arrays isn't all that different from nested objects. Simply replace the named properties of nested objects with indexes of nested arrays. Let's review with an example. Be sure to follow along in the REPL.

**Top Tip:** You may have discovered that, when you're working in the embedded terminal in VS Code or the terminal application on your computer, you can repeat the last command you ran by pressing the "up" arrow key. The same is true in the REPL console! Give it a try.

Copy the following code into the REPL's code window:

```
const letters = ["a", ["b", ["c", ["d", ["e"]], "f"]]];
```

Given the above nested array, how would we get the letter `'e'` ? First, we'd need the second element in `letters` , `letters[1]` :

```
letters[1];
//=> ["b", ["c", ["d", ["e"]], "f"]]
```

Then we'd need the second element of that element, so `letters[1][1]` :

```
letters[1][1];
//=> ["c", ["d", ["e"]], "f"]
```

Then the second element of **that** element, `letters[1][1][1]` :

```
letters[1][1][1];
//=> ["d", ["e"]]
```

And the second element of ***that*** element, `letters[1][1][1][1]` :

```
letters[1][1][1][1];
//=> ["e"]
```

Finally, we want the first element in that final nested array, `letters[1][1][1][1][0]` :

```
letters[1][1][1][1][0];
//=> "e"
```

Whew! That's a lot to keep track of. Just remember that each lookup (each set of square brackets) "drills down" into each successive nested array.

## Bonus: Iterating over nested objects and arrays

> **Note**: From here on out, this lesson gets pretty abstract! If you're feeling confident with arrays and objects and are curious how to write some abstract code to iterate over nested objects, continue on; otherwise, feel free to continue to the next lesson.

Our initial shallow object had a lot of drawbacks, but one advantage of it is that it was very easy to iterate over all of the information:

```
const userInfo = {
  firstName: "Avi",
  lastName: "Flombaum",
  companyName: "Flatbook Labs",
  jobTitle: "Developer Apprentice",
  friend1firstName: "Nancy",
  friend1lastName: "Burgess",
  friend1companyName: "Flatbook Labs",
  friend1jobTitle: "Developer Apprentice",
  friend2firstName: "Corinna",
  friend2lastName: "Jackson",
  friend2companyName: "Flatbook Labs",
  friend2jobTitle: "Senior Developer",
  project1title: "Flatbook",
  project1description:
    "The premier Flatiron School-based social network in the world.",
  project2title: "Scuber",
  project2description:
    "A burgeoning startup helping busy parents transport their children to and from all of their activities on scooter
};

function shallowIterator(target) {
  for (const key in target) {
    console.log(target[key]);
  }
}

shallowIterator(userInfo);
// LOG: Avi
// LOG: Flombaum
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Nancy
// LOG: Burgess
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Corinna
// LOG: Jackson
// LOG: Flatbook Labs
// LOG: Senior Developer
// LOG: Flatbook
// LOG: The premier Flatiron School-based social network in the world.
// LOG: Scuber
```

We can also use it with arrays:

```
const primes = [2, 3, 5, 7, 11];

shallowIterator(primes);
// LOG: 2
// LOG: 3
// LOG: 5
// LOG: 7
// LOG: 11
```

> **Note**: our `shallowIterator()` function uses `for...in` to iterate through the object that's passed to it. We learned in the previous lesson that `for...in` is not the best iterator to use with arrays. Because we're not currently working in the browser (and therefore cross-browser consistency isn't an issue), we can safely ignore that problem for the moment. Since this example is fairly complicated, we'll work through the process using `for...in` first then, once we've got that working, build a modification that will handle arrays appropriately.

Unfortunately, as you may be able to guess from its name, our `shallowIterator()` function can't handle nested collections:

```
const numbers = [1, [2, [4, [5, [6]], 3]]];

shallowIterator(numbers);
// LOG: 1
// LOG: [2, [4, [5, [6]], 3]]
```

It's trained to iterate over the passed-in array's elements or object's properties, but our function has no concept of *depth*. When it tries to iterate over the above nested `numbers` array, it sees only two elements at the top level of the array: the number `1` and **another** array, `[2, [4, [5, [6]], 3]]` . It `console.log()` s out both of those elements and calls it a day, never realizing that we also want it to print out the elements inside the nested array.

It behaves similarly with objects. If we passed the nested version of `userInfo` to it, the values at the top level of the object ("Avi" and "Flombaum") would be logged correctly, but for the `company` key, the object it points to would be logged, and, for `friends` and `projects` , arrays of objects would be logged. Try it out for yourself in the REPL.

Let's modify our function so that if it encounters a nested object or array, it will additionally print out all of the data contained therein. We can do this using an `if` condition and the `typeof` operator:

```
function shallowIterator(target) {
  for (const key in target) {
    if (typeof target[key] === "object") {
      for (const nestedKey in target[key]) {
        console.log(target[key][nestedKey]);
      }
    } else {
      console.log(target[key]);
    }
  }
}

shallowIterator(numbers);
// LOG: 1
// LOG: 2
// LOG: [4, [5, [6]], 3]
```

Now we've gone two levels deep, which gets us a bit closer to our goal. However, there are two pretty clear drawbacks to this strategy:

1. We'll have to add a new `for...in` statement for every level of nesting, quickly ballooning our function out to an unmanageable size.
2. Since we need to add a separate `for...in` statement for each additional level, we'll have to know exactly what the target structure looks like ahead of time and update our function accordingly. That's a lot of repetitive, error-prone work, and it results in a function that can only be used for data with that particular structure.

There has to be another way!

## Recursion

Lucky for us, there **is** another way: recursion. It's one of the more powerful concepts in programming, but it's also pretty hard to grasp at first. **Don't sweat it if it doesn't click immediately**. We'll introduce the concept here but come back to it periodically throughout the rest of the

JavaScript material. Essentially, **a recursive function is a function that calls itself**.

Let's take a look at a better way to write our `shallowIterator()` to take advantage of recursion:

```
function deepIterator(target) {
  if (typeof target === "object") {
    for (const key in target) {
      deepIterator(target[key]);
    }
  } else {
    console.log(target);
  }
}
```

When we invoke `deepIterator()` with an argument, the function first checks if the argument is an object (recall that the `typeof` operator returns `"object"` for arrays as well). If the argument **isn't** an object, `deepIterator()` simply `console.log()` s out the argument and exits. However, if the argument **is** an object, we iterate over the properties (or elements) in the object, passing each to `deepIterator()` and **re-invoking the function**. That's recursion!

Let's see it in action:

```
const numbers = [1, [2, [4, [5, [6]], 3]]];

deepIterator(numbers);
// LOG: 1
// LOG: 2
// LOG: 4
// LOG: 5
// LOG: 6
// LOG: 3
```

To help us see what's going on here let's use a REPL. Go ahead and copy the following code into **[replit (https://replit.com/languages/javascript)](https://replit.com/languages/javascript)**:

```
function deepIterator(target) {
    console.log("Argument: ", target);
    if (typeof target === 'object') {
      for (const key in target) {
        deepIterator(target[key]);
      }
    } else {
      console.log("Logged value: ", target);
    }
  }

  const numbers = [1, [2, [4, [5, [6]], 3]]];

  deepIterator(numbers);
```

Notice that we've added a `console.log()` at the top of the function that will log whatever argument was passed to our function. We've also added a label to the second `console.log()` so you can see the values that are getting logged from the `else` statement. If you run the code, you will see an "Argument" logged for each time the function is called. You will also see a "Logged value" for each time the code in the `else` executes. Referring to the output of the `console.log()` s, step through the function for each element to trace what's happening.

Our function also works with combinations of nested objects and arrays. Replace the existing code in replit with the following and run it (we've gone back to a single `console.log()` inside our function to make the output easier to read):

```
function deepIterator(target) {
    if (typeof target === 'object') {
      for (const key in target) {
        deepIterator(target[key]);
      }
    } else {
      console.log(target);
    }
  }
```

```
const userInfo = {
  firstName: "Avi",
  lastName: "Flombaum",
  company: {
    name: "Flatbook Labs",
    jobTitle: "Developer Apprentice",
  },
  friends: [
    {
      firstName: "Nancy",
      lastName: "Burgess",
      company: {
        name: "Flatbook Labs",
        jobTitle: "Developer Apprentice",
      },
    },
    {
      firstName: "Corinna",
      lastName: "Jackson",
      company: {
        name: "Flatbook Labs",
        jobTitle: "Lead Developer",
      },
    },
  ],
  projects: [
    {
      title: "Flatbook",
      description:
        "The premier Flatiron School-based social network in the world.",
    },
    {
      title: "Scuber",
      description:
        "A burgeoning startup helping busy parents transport their children to and from all of their activities on scc
    },
  ],
};

deepIterator(userInfo);
// LOG: Avi
// LOG: Flombaum
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Nancy
// LOG: Burgess
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Corinna
// LOG: Jackson
// LOG: Flatbook Labs
// LOG: Lead Developer
// LOG: Flatbook
// LOG: The premier Flatiron School-based social network in the world.
// LOG: Scuber
```

To keep track of how many times our function is recursively invoking itself, it might be helpful to use a counter variable:

```
let counter = 0;

function deepIterator(target) {
  counter++;

  if (typeof target === "object") {
      for (const key in target) {
        deepIterator(target[key]);
      }
    } else {
      console.log(target);
    }
  }

  deepIterator(userInfo);
  // LOG: Avi
  // LOG: Flombaum
  // LOG: Flatbook Labs
  // LOG: Developer Apprentice
  // LOG: Nancy
  // LOG: Burgess
  // LOG: Flatbook Labs
  // LOG: Developer Apprentice
  // LOG: Corinna
  // LOG: Jackson
  // LOG: Flatbook Labs
  // LOG: Lead Developer
  // LOG: Flatbook
  // LOG: The premier Flatiron School-based social network in the world.
  // LOG: Scuber
  // LOG: A burgeoning startup helping busy parents transport their children to and from all of their activities on scoo
```

If you check the value of `counter` after running the code, you should see that `deepIterator()` was called a total of 26 times: we invoked it once, and it invoked itself 25 additional times! If we look closely at our nested `userInfo` object, we can see that it contains two arrays, seven nested objects, and sixteen key-value pairs where the value is a string. Add those all up (2 + 7 + 16), and you get our 25 recursive invocations!

**Reminder**: You can check the value of `counter` by either wrapping it in a `console.log()` in the code window or checking it in the console window after running the code.

## Modifying our Program to Better Handle Arrays

In our `deepIterator()` function, we're using an `if` statement to evaluate the argument that's passed in. We do one thing if `target` is an object and something else if it's not. Now we want to modify our function to handle one more situation: when `target` is an array. Doing this is a simple matter of adding an `else if` to our `if` statement; its code block will execute **if** `target` is an array.

We can determine whether a variable is an array using an Array Static Method (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#static_methods), `Array.isArray()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray). Let's use that in our new condition. We'll also add a new `console.log()` to verify that our code is correctly detecting the arrays (we've commented out the final `console.log()` for now):

```
function deepIterator(target) {
  if (typeof target === "object") {
    for (const key in target) {
      deepIterator(target[key]);
    }
  } else if (Array.isArray(target)) {
    console.log("We found an array");
    // iterate through the array
  } else {
    // console.log(target);
  }
}

deepIterator(userInfo);
```

Now if we run the code, we see ...nothing. For some reason, our code isn't detecting the arrays. Remember that **arrays are objects**, so our `if` condition returns true for arrays as well as objects and the `else if` never executes. We can fix this by reversing the order of our conditions. Let's also put the second `console.log()` back in, and try running the code again.

```
function deepIterator(target) {
  if (Array.isArray(target)) {
    // iterate through the array
    console.log("We found an array");
  } else if (typeof target === "object") {
    for (const key in target) {
      deepIterator(target[key]);
    }
  } else {
    console.log(target);
  }
}

deepIterator(userInfo);
// LOG: Avi
// LOG: Flombaum
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: We found an array
// LOG: We found an array
```

Much better! The logs are working for the primitive values and the non-array object, so now we just need to code the body of our new `if` statement, using `for...of` :

```
function deepIterator(target) {
  if (Array.isArray(target)) {
    for (const element of target) {
      deepIterator(element);
    }
  } else if (typeof target === "object") {
    for (const key in target) {
      deepIterator(target[key]);
    }
  } else {
    console.log(target);
    }
  }

deepIterator(userInfo);
// LOG: Avi
// LOG: Flombaum
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Nancy
// LOG: Burgess
// LOG: Flatbook Labs
// LOG: Developer Apprentice
// LOG: Corinna
// LOG: Jackson
// LOG: Flatbook Labs
// LOG: Lead Developer
// LOG: Flatbook
// LOG: The premier Flatiron School-based social network in the world.
// LOG: Scuber
```

Whew!

## Conclusion

This is very advanced stuff, and you should absolutely not get discouraged if it doesn't click at first. Create some other nested data structures and traverse over them with `shallowIterator()` and `deepIterator()` , noting the limitations of the former. Use the debugging tools available to you to get a handle on what's happening at each step of the process.

You got this!

## Resources

- **MDN: Recursion (JavaScript) (https://docs.microsoft.com/en-us/scripting/javascript/advanced/recursion-javascript)**
- **freeCodeCamp: Recursion in JavaScript (https://medium.freecodecamp.org/recursion-in-javascript-1608032c7a1f)**
- **JavaScript.info: Debugging in Chrome (https://javascript.info/debugging-chrome)**