Object-Oriented Programming Objectives

Object-Oriented Programming (OOP) is one of the most popular programming paradigms. Additionally, OOP can help beginning engineers learn how to breakdown complex problems.

You will be tested on this material by completing a guided project using the following principles. You will also answer questions about their definitions.

- The three pillars of object-oriented programming
- The SOLID principles
- How to apply the Law of Demeter

Object-Oriented Programming Explained

JavaScript is an object-oriented language. You've already used objects in your programming. For example, when you write

```
const array = [1, 2, 3, 4];
const squares = array.map(x => x * x);
```

the value stored in `array` is an object! The value stored in `squares` is *another* object! There are objects all over the place!

In this article, you're going to learn the theory of object-oriented programming, the features that make it different than other kinds of programming.

- The property of encapsulation.
- The property of inheritance.
- The property of polymorphism.

These features allow us to thoughts in a way that (hopefully) represents the way that we, as humans, think and reason about the world.

Encapsulation: enclose (something) in or as if in a capsule

When you write a class, you do something amazing which revolutionized the organization of computer software source code: you put behavior (the methods) and the data it works on (instance variables, also called *fields*) together. Up to that point, programmers had to deal with code that declared data structures in one file and used in multiple other files all over the code base. Understanding where data got changed became exponentially difficult as the size of software grew.

Knowing where data changes is the most important aspect of software. Every change in the way programmers like yourself organize your code has been about maintainability, about how and where you should put the code that runs the instructions that makes the software works. In a lot of object-oriented languages, like Java, C++, Objective-C, and C#, they tend to either enforce or encourage putting *one class definition per file*. You are not bound by that restriction or convention in JavaScript.

Imagine, if you will, that you want to buy something from a vending machine. You tap your phone against the payment reader (or insert coins into a slot, or swipe a credit card). Once the payment is authorized in any of those forms, you can make a selection and receive your tasty treat. If you were to think about that as a series of steps, you could write them like this.

1. Authorize payment through payment vendor
2. Make selection
3. Retrieve tasty treat

With respect to the concepts of object-oriented programming, the vending machine would be an object. All of the internal workings of the machine, how it communicates wirelessly with a payment vendor, how its internal mechanics thrash about to deliver a beverage or snack to you, all of that is hidden behind a plastic advertisement. All of that behavior is *encapsulated* inside the machine so that you don't have to worry about the details. Imagine a less encapsulated world where you had to perform the following steps to get your tasty treat.

1. Call your payment provider.
2. Specify that you want to spend no more than $2 on your next purchase.

3. Write down a confirmation number for an authorization of up to a $2 payment.
4. Call the vending machine company.
5. Give them your payment authorization confirmation number.
6. Key in a 16-digit authorization number that they tell you.
7. Make selection.
8. Tell the vending company the transaction number and the total amount.
9. Retrieve tasty treat.

Now, instead of hiding all the details about how payments work, the vending machines forces you to have to participate in the payment process. A system such as this relies on people acting as good agents when they report the total amount with the transaction number. A system such as this is inconvenient for the person that has a desire to sate. The vending machine *encapsulates the complexity of its internal mechanisms behind easy-to-use interactions*. The same goes with encapsulation with object-oriented programming.

Say that you wanted to write software that helped gyms manage their business. Something that can happen at the gym would be that you want to register a person as a member for the gym. Code that does that may look something like this.

```
class Gym {
  registerMember(firstName, lastName, email, creditCardInfo) {
    const person = new Person(lastName, firstName, email);
    person.addCardInfo(creditCardInfo);
    this.members.push(person);
  }

  // Lots of other code
}
```

When other code uses this, all it knows about how the gym object works is that there's a method named `registerMember` and if you give it some information, then a person becomes a member of the gym. Imagine that the code inside the `registerMember` function was written like this instead:

```
class Gym {
  registerMember(firstName, lastName, email, creditCardInfo) {
    this.members[this.members.length] = {
      firstName: firstName,
      lastName: lastName,
      email: email,
      creditCardInfo: creditCardInfo,
    };
  }

  // Lots of other code
}
```

The encapsulation of the behavior (adding a new member to the gym) is hidden behind the method name `registerMember`. Any code which uses the `registerMember` method of the `Gym` class doesn't care what the code looks like inside of `Gym`. It still just uses it like this:

```
gym = new Gym();
gym.registerMember('Margaret', 'Hamilton', 'mh@mit.edu', null);
```

Encapsulation puts the behavior and data together behind methods that hide the specific implementation so that code that uses it doesn't need to worry about the details of it.

Inheritance: derived from one's ancestors

In the same way that biology passes traits of a parent organism to its descendants, so does object-oriented programming through its support of**inheritance**. There are a lot of different kinds of inheritance because there are a bunch of different "type systems" that programming languages support. JavaScript, in particular, supports a type of inheritance known as**implementation inheritance**through a mechanism known as **prototypal inheritance**. *Implementation inheritance*means that the data and methods defined on a parent class are available on objects created from classes that inherit from those parent classes. *Prototypal inheritance*means that JavaScript uses prototype objects to make its *implementation inheritance*actually work.

Notes: Here is some terminology for you.

- For the sake of brevity, you should understand that whenever you read the phrase "parent class", that also means "prototype".
- "Super class" is another name for "parent class".
- "Base class" is another name for "parent class".
- Sometimes, you will see "inheritance" referred to as "subtyping".

A built-in example of inheritance

All object in JavaScript share a common parent class, the `Object`parent class. The `Object`class has a method named `toString()`on it. Since all objects in JavaScript are child classes (or grandchild classes or great grandchild classes or great great...), that means that every object in JavaScript has a`toString()`method! If a class doesn't create its own, then it will fall back to its parent class' implementation of `toString()`. If the parent class doesn't have an implementation, and the parent's parent class doesn't have an implementation, it will keep going until it gets to the `Object`class and use that one. (That's some recursion in there. Did you sense that?) Open a terminal, start node, and type the following.

```
> [1, 2, 3].toString();
'1,2,3'
> "some text".toString();
'some text'
> new Date().toString();
'«the current date and time»'
> new Object().toString();
[object Object]
```

You'll note the following:

- The `toString()`method of an array takes the values in the array and turns them into a comma-delimited string, that is, it puts commas between each of the items.
- The `toString()`method of a string does nothing and just returns the string object. (You might remember that strings are primitive types, but strings are special and you can also call methods on them like they are objects.)
- The `toString()`method of a `Date`object returns a long textual representation of the date and time that the `Date`object represents.
- The `toString()`method of `Object`returns the not so helpful "[object Object]".

When you declare a class with no explicit parent class, then JavaScript will make it a child of `Object`.

```
class MyClass {}

// is the same as
class MyClass extends Object {}
```

A custom example of implementation inheritance

Imagine that you have created the following code in a JavaScript file and loaded it into a browser (through a `<script>` tag) or run it with the node command.

```
class Charity {}

class Business {
  toString() { return 'Give us your money.'; }
}

class Restaurant extends Business {
  toString() { return 'Eat at Joe\'s!'; }
}

class AutoRepairShop extends Business {}

class Retail extends Business {
  toString() { return 'Buy some stuff!'; }
}

class ClothingStore extends Retail {}

class PhoneStore extends Retail {
  toString() { return 'Upgrade your perfectly good phone, now!'; }
}

console.log(new PhoneStore().toString());
console.log(new ClothingStore().toString());
console.log(new Restaurant().toString());
console.log(new AutoRepairShop().toString());
console.log(new Charity().toString());
```

What do you think those last four lines will print out? Try running that code to confirm your suspicions. (**Important**: When given the opportunity to try out short snippets of code like the above example, *do not* copy and paste it. This is an opportunity for you to type it into an editor or command line to become familiar with the syntax.)

The three classes `AutoRepairShop`, `Charity`, and `ClothingStore` in the example above do not have their own `toString()` methods. That means that an object of one of those three types can't respond to that method invocation. The JavaScript runtime at that point starts inspecting prototype objects to find a `toString()` method.

- For `AutoRepairShop`, it finds a `toString()` method on its parent class `Business`, and prints "Give us your money.".
- For `ClothingStore`, it finds a `toString()` method on its parent class `Retail`, and prints "Buy some stuff!".
- For `Charity`, it finds a `toString()` method on its implicit parent class `Object`, and prints "object Object".

The nuts and bolts of prototypal inheritance

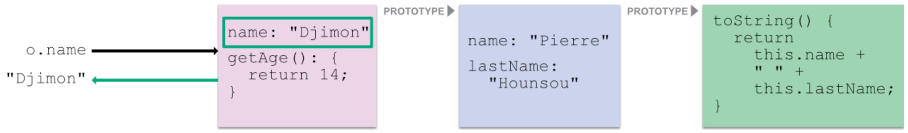
JavaScript is almost singularly unique in its concept of prototype-based inheritance. No other commonly used language in the modern world has this kind of inheritance. (Examples of languages that *do* have prototypal inheritance are Common Lisp, Self, and Io. These languages have niche markets that some people adore. However, you would be hard-pressed to find them in use in most software development shops.)

As you saw in the example from the last section, when you write code that accesses a method (or property) on an object, if JavaScript doesn't find it, it will "delegate" it to its prototype, that is, it will determine if the prototype has that method (or property). If it doesn't find it there, it will delegate it to the object's prototype's prototype until it reaches `Object`. If it doesn't find it

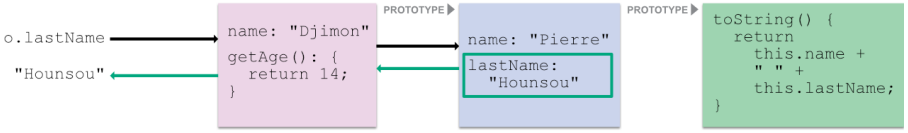
on `Object`, then you'll get an error or `undefined`, depending on the mechanism that you're using.

Consider the following diagrams which show an object with `name` and `getAge` properties. It has a parent object (prototype) that has `name` and `lastName` properties. That parent object has another parent object that has a `toString` property.

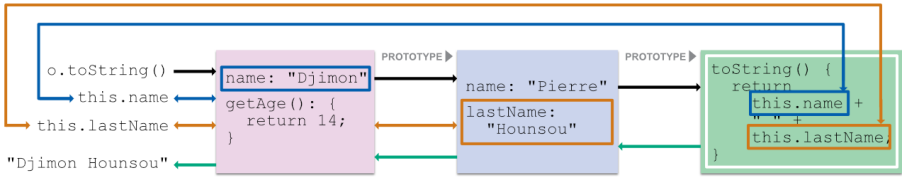
If you write code to get the `name` property of the object, it will look on that object, determine that a `name` property exists, and return the value of that property, "Djimon".



If you write code to get the `lastName` property of the object, it will look at that object, see that there's no `lastName` property on it, and go to its parent object, its prototype object. There, it sees a `lastName` property defined and returns the value of that property, "Hounsou".



If you write code to invoke the `toString()` method of the object, it will look at that object, see that there's no `toString` property on it, and go to its parent object, which also doesn't have a `toString` property. Finally, it will look at the grandparent object (the one in green) and see that it has that `toString()` method on it and invoke it. But, the story doesn't end there.



When the `toString()` method runs, it accesses the `this` property to get `this.name` and `this.lastName`. The `this` property *references the object that the call originally came from*. That's really important.

When JavaScript uses a property (or method) from a prototype that it found through prototypal inheritance, then the `this` property points to the original object on which the first call was made.

In this case, the call to `this.name` in the `toString()` method of the original object's grandparent class acts just like the call to `o.name` from two examples back. The call to `this.lastName` acts the same way.

Try the following code in your JavaScript environment to get a feel for it. Change the assignments in the constructor. Remove assignments in the constructor. See how changes affect the messages printed to the console.

```
class Parent {
  constructor() {
    this.name = 'PARENT';
  }
  toString() {
    return `My name is ${this.name}`;
  }
}

class Child extends Parent {
  constructor() {
    super();
    this.name = 'CHILD';
  }
```

```
}

const parent = new Parent();
console.log(parent.toString());

const child = new Child();
console.log(child.toString());
```

Polymorphism: a cool sounding word to impress your friends

In object-oriented programming, polymorphism means that you can treat a variable as if it were one of its parent types. You've already been doing this in JavaScript through its prototypal inheritance. So, just remember its definition for this course. It's a perennial favored question in interviews:

Polymorphism is the ability to treat an object as if it were an instance of one of its parent classes.

What you've learned

In this reading, you learned about the three pillars of object-oriented programming (encapsulation, inheritance, and polymorphism) and how they relate to JavaScript.

You learned that encapsulation is just all of the details behind an object's methods.

You learned that inheritance is the ability to gain behavior and data from parent classes.

You learned that polymorphism is just a fancy work for being able to use the methods of a parent class on an object of a child class.

The SOLID Principles Explained

The three pillars of object-oriented programming describe *how* classes and objects work. What they don't describe are good practices for *what* should go into a class. That's the difference between object-oriented programming (pillars) and *object-oriented design* (SOLID)! That's where the SOLID Principles come in. "SOLID" is an anagram that stands for

- The Single-Responsibility Principle
- The Open-Close Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

Of these five principles, only two are directly applicable to your use of JavaScript (and Python) in this course. Moreover, two are only truly applicable in languages like Java, C++, and C#. This article spends many words explaining the applicable ones and provides a definition for those that are not so applicable in JavaScript (and Python).

It's really important that you don't stress out over being able to expertly apply these principles any time in the next three years. Being able to apply them takes practice and experience. You're getting introduced to them, now, so that you can have the ideas in your brain. With them in your brain, they can influence how you go about writing classes.

But, don't sit around thinking about the best way to apply these object-oriented design principles. That is **analysis paralysis**! You'll get stuck and not be able to get out of it. Working code is usually better than no code at all. You can write code and, then, think about it, look at it, and start asking yourself whether it follows these principles. That's the best way to start out. Don't worry about applying them to your design *before* you write any code. That will come with practice.

Single-Responsibility Principle

A class should do one thing and do it well.

Robert C. Martin, otherwise known as "Uncle Bob", created the SOLID principles. He explains the Single-Responsibility Principle as, "A class should have only one reason to change." He has also described the way to do it as "Gather together the things that change for the same reasons. Separate those things that change for different reasons."

This principle is about limiting the impact of change.

A class should have the responsibility to have the data and behavior over a single part of the functionality provided by your software, and that responsibility should be entirely encapsulated by the class.

For example, consider the following class.

```
// THIS IS BADLY DESIGNED CODE
class Book {
  constructor (title, author, pages) {
    this.title = title;
    this.author = author;
    this.pages = pages;
    this.currentPage = 0;
  }

  addPage(page) {
    this.pages.push(page);
  }

  getCurrentPage() {
    return this.pages[this.currentPage];
  }

  turnPage() {
    this.currentPage += 1;
```

```
    if (this.currentPage >= this.pages.length) {
      this.currentPage = this.pages.length - 1;
    }
  }

  printText() {
    const firstPage = [this.title + "\n" + this.author];
    return firstPage.concat(this.pages);
  }

  printHTML() {
    const firstPage = [`<h1>${this.title}</h1>`];
    const htmlPages = this.pages.map(x => `<section>${x}</section>`);
    return htmlPages;
  }
}
```

What does the book class know about?

- How to manage the information about a book.
- How to print that book to different formats.

Think about a book that you would hold in your hand. Which of those two "concerns" does the physical book know *nothing* about?

If you answered "printing to different formats", yeah, that's the one. Printing is a different "concern", a different set of responsibilities. Instead of having a single `Book` class that can print itself, following the Single-Responsibility Principle would have you create two classes.

```
class Book {
  constructor (title, author, pages) {
    this.title = title;
    this.author = author;
    this.pages = pages;
    this.currentPage = 0;
  }

  addPage(page) {
    this.pages.push(page);
  }

  getCurrentPage() {
    return this.pages[this.currentPage];
  }

  getPrintableUnits() {
    return this.pages;
  }

  turnPage() {
    this.currentPage += 1;
    if (this.currentPage >= this.pages.length) {
      this.currentPage = this.pages.length - 1;
    }
  }
}

class SimplePrinter {
  constructor(printable) {
    this.printable = printable;
  }

  printText() {
    const firstPage = [this.printable.title + "\n" + this.printable.author];
    return firstPage.concat(this.printable.pages);
  }

  printHTML() {
    const firstPage = [`<h1>${this.printable.title}</h1>`];
    const units = this.printable.getPrintableUnits();
    const htmlPages = units.map(x => `<section>${x}</section>`);
    return firstPage.concat(htmlPages);
  }
}
```

Now, you have two things, a `Book` and a `SimplePrinter`. The `Book` now knows all about being a book. The `Printer` is interested in things that are "printable". In this case, "printable" means having `title` and `author` properties, and a method called `getPrintableUnits()`, that it can use to turn something into text.

What's neat about this is that you can now create any other kind of thing that might get printed and, as long as it has a `title` property, `author` property, and a method called `getPrintableUnits()`, the `SimplePrinter` can handle it! If you now add this `Poem` class to your program, then the `SimplePrinter` can print it, too!

```
class Poem {
  constructor (title, author, lines) {
    this.title = title;
    this.author = author;
    this.lines = lines;
  }

  addLine(line) {
    this.lines.push(line);
  }

  getPrintableUnits() {
    return [this.lines.join('\n')];
  }
}
```

The Liskov Substitution Principle

This principle is named after Barbara Liskov, a computer scientist of some renown from MIT. It has a very mathematical definition that you can now read.

However, if it doesn't make any sense, don't worry. You'll get an easy-to-understand definition after this one.

***Subtype Requirement**: Let $\phi(x)$ be a property provable about objects x of type **T**. Then $\phi(y)$ should be true for objects y of type **S** where **S** is a subtype of **T**.*

Boy, if you're ever asked about the Liskov Substitution Principle in an interview (or a party) and you can rattle off that definition followed by an explanation, you're going to impress the heck out of the interviewer (or your party mates)! What that means is this.

You can substitute child class objects for parent class objects and not cause errors.

All this means is that if you have a class with a method on it, any child class that overrides that method must not do something unexpected. For example, let's say you have the following class in JavaScript.

```
class Dog {
  speak() {
    return "woof!";
  }

  /* other code about dogs */
}
```

Any instance of `Dog` will be able to say "woof!" Now, chihuahuas have a different vocabulary.

```
class Chihuahua extends Dog {
  speak() {
    return "yip!";
  }

  /* other code about chihuahuas */
}
```

What you can't do is create something like the following.

```
class Shitzu extends Dog {
  speak() {
    const fs = require('fs');
    fs.unlink('~/.bash_profile');
    return "YOU'RE PWNED!";
  }

  /* other code about shitzus */
}
```

That would delete the profile for the Bash shell of the person who is running your program! The speak method should have no side effects as defined by the `Dog` class. The fact that a child class, `Shitzu`, will delete files when the `speak` method is called is ludicrous.

The methods that you override in child classes *must* match the intent of the methods found on the parent classes.

The other principles

These other three principles, as mentioned, are important for languages that have *static typing*. That means a variable can have only one kind of thing in it. In JavaScript, you have no such restriction. In JavaScript, you can declare a variable, assign a string to it, then a number, then a date. That's amazing!

```
let value = "Hello, Programmer!";
value = 6.28;
value = new Date();
```

In languages like Java, C++, and C#, you have to specify the kind of data type you want to store in the variable. Then, only that kind (or instances of child classes through polymorphism) can be stored in those variables. The following code in C++ does not compile and reports errors.

```
// THIS IS ERROR-FILLED CODE
std::string value = "Hello, Programmer!";
value = 6.28; // ERROR: Cannot assign a float to
              // a variable of type std::string.
```

In those worlds, these other principles have much more applicability. Again, just memorize their names and definitions.

- **Open-Close Principle** means that a class is *open for extension and closed for modification*. It means that creating new functionality can happen in child classes and not the original class.
- **Interface Segregation Principle** means that method names should be grouped together into granular collections called "interfaces".
- **Dependency Inversion Principle** means that functionality that your class depends on should be provided as parameters to methods rather than using `new` in the class to create a new instance.

What you learned

In this article, you learned a lot about the Single-Responsibility Principle (SRP) and the Liskov Substitution Principle (LSP). You then learned some definitions for the other three SOLID principles. You can start applying the SRP and LSP in all of your code!

Controlling Coupling With The Law Of Demeter

To reduce the complexity of the software that you write, you should try to reduce what is known as the "coupling" of the classes in your source code. In this section, you will learn about the Law of Demeter, a way to help reduce coupling in your software. Then, you will learn about when you can ignore the Law of Demeter altogether.

Coupling

For the purposes of this article, coupling is defined as "the degree of interdependence between two or more classes," not the award-winning British TV show.

To reduce coupling, you must reduce how many other classes must know about to do their job. A caller method is not coupled with all of the inner dependencies. It is only coupled with one object. Using this principle you can also change a class without affecting others. This is an improvement to the way that encapsulation helps you.

The fewer the connections between classes, the less chance there is for the*ripple effect*. Ideally, changing the implementation of a class should not require a change in another class. If it does, that means a class must understand the details of a class on which it doesn't directly depend. This is known as the *principle of locality*.

The formal definition

The definition of the Law of Demeter is as follows.

A method of an object can only invoke the methods (or use the properties) of the following kinds of objects:

- Methods on the object itself
- Any of the objects passed in as parameters to the method
- And object created in the method
- Any values stored in the instance variables of the object
- Any values stored in global variables

Practical advice

The Law of Demeter is more of a guideline than a principle to help reduce coupling between components. The easiest way to comply with the Law of Demeter is *don't use more than one dot (not counting the one after "this")*. For example, the following code breaks the Law of Demeter.

```
// THIS CODE BREAKS THE LAW OF DEMETER
class Airplane {
  constructor() {
    this.engine = new PropEngine();
  }

  takeOff() {
    // this.engine is a value stored in an instance variable
    this.engine.getThrottle().open();
    // two dots^     and     ^
  }
}

class PropEngine {
  constructor() {
    this.throttle = new Throttle();
  }
```

```
  getThrottle() {
    return this.throttle;
  }
}

class Throttle {
  open() {
    return "VROOOOM"!;
  }
}
```

That code breaks the Law of Demeter because it uses two dots to get its work done, the first between `engine` and `getThrottle()` and the second between `getThrottle()` and `open()`.

The `Airplane` class is now coupled to the `PropEngine` class. Assume that the `getThrottle()` method of the `PropEngine` class returns a `Throttle` object that has the `open` method on it. Now, the `Airplane` class need to know about **two classes**, that is, the `PropEngine` class and the `Throttle` class. It should only know about the one that it created, the `PropEngine` class.

Instead, you should change the way the engine works.

```
class Airplane {
  constructor() {
    this.engine = new PropEngine();
  }

  takeOff() {
    this.engine.openThrottle();
  }
}

class PropEngine {
  constructor() {
    this.throttle = new Throttle();
  }

  openThrottle() {
    return this.throttle.open();
  }
}

class Throttle {
  open() {
    return "VROOOOM"!;
  }
}
```

This reduces the *coupling* of `Airplane`. It now only depends on the `PropEngine` class, now.

An important thing to notice here is that the `Airplane` is now "telling" and not "asking". You'll often hear that in object-oriented programming, "tell, don't ask".

In the previous version, the `takeOff` method *asked* for the engine's `Throttle` object using the `getThrottle()` method. Then, it told the `Throttle` object to open. In the better version, the `takeOff` method simple *tells* the engine to `openThrottle` and let's the engine figure out how to do that.

You can't cheat the Law

Separating calls onto different lines *DOES NOT* get around the Law of Demeter. You can't cheat. Here's the earlier code but with a different body for the `takeOff` method.

```
// THIS CODE BREAKS THE LAW OF DEMETER
class Airplane {
  constructor() {
    this.engine = new PropEngine();
  }

  takeOff() {
    const engine = this.engine; // This is the PropEngine
    const throttle = engine.getThrottle(); // This is a Throttle
    throttle.open();
  }
}
```

Sure, it *looks* like there's only one dot per line. But, the Law of Demeter is really about a class knowing about other classes. It still has to know that there is an `open` method on the `Throttle` class to do it's work. So, even though it doesn't break the "one dot" rule, it breaks the stricter definition of the Law of Demeter which is that it is calling a method (`open`) on an object `throttle` that doesn't match any of the five conditions of the definition.

When to ignore the Law of Demeter

When you're working with objects that come from code that you didn't create, you often have to break the Law of Demeter. Hopefully, the other code doesn't change often (or ever)!

For example, here's some code that you might see in a Web application.

```
document
  .getElementById('that-link')
  .addEventListener('click', e => e.preventDefault());
```

That's an obvious violation of the Law of Demeter. Your code must know the details of the HTMLDocumentElement and HTMLLinkElement objects. But, there's no way around it because you *have* to use the API provided by the DOM from the browser.

So, if you don't own the code, and there's no way for you to get the job done without more than one dot, just dot it up!

When else should you ignore the Law of Demeter?

We try very hard to decouple things. The things that are nearly impossible to prevent tight coupling with are the visualizations of our program. The user interface (UI) that people see *has* to know about the structure of the data. Because of that, it is normal to have UIs that break the Law of Demeter all over the place.

UIs are a special thing. They break all kinds of rules because UIs are *not object-oriented concepts*. Their components may be objects, like the *HTMLDivElement* object in the DOM that lets you interact with a `div` element in JavaScript. Using that *thing* is object-oriented. However, the way that you go about bridging between the state of your application to the visualiation is *not* object-oriented.

Go ahead and couple those views to your models. This is just the normal price of software development.

(Lots of very smart people have tried to come up with ways to decouple views and the objects that make up software. No one has really created a satisfactory solution.)

What you've learned

You learned that the Law of Demeter is a way to reduce coupling by following the "one dot" rule, otherwise known as the "don't talk to strangers" rule. You also know, now, that it's practical to break that law when you have to rely on other people's code.