Classes Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Define a constructor function using ES5 syntax.
2. Define a method on the prototype of a constructor function.
3. Declare a class using ES6 syntax.
4. Define an instance method on a class (ES6).
5. Define a static method on a class (ES6).
6. Instantiate an instance of a class using the `new` keyword.
7. Implement inheritance using the ES6 `extends` syntax for an ES6 class.
8. Utilize the `super` keyword in a child class to inherit from a parent class.
9. Utilize `module.exports` and `require` to import and export functions and class from one file to another.

Constructor Function, What's Your Function?

Up until now, you've used object initializer or "literal notation" to create POJOs (plain old JavaScript objects). While this approach to creating objects is convenient (and not to mention easy), it's not an ideal way to define the attributes and behaviors for an object type nor is it an efficient way to create many objects of that type.

In ES2015, JavaScript gained the `class` keyword, giving developers a formal way to create a class definition to specify an object type's attributes and behavior. The class definition is also used to create objects of that specific type.

In this article, you'll learn how constructor functions and prototypes were used, prior to the introduction of the ES2015 `class` keyword, to mimic or imitate classes. Understanding constructor functions and prototypes will not only prepare you for working with legacy code, it'll prepare you to understand how ES2015's classes are really just a syntactic layer of sugar over these language features.

When you finish this article, you should be able to:

- Define a constructor function for an object type that initializes one or more properties;
- Invoke a constructor function using the `new` keyword;
- Use the `instanceof` operator to check if an object is an instance of a specific object type; and
- Define sharable methods on the `prototype` property of a constructor function.

Defining a constructor function

To review, an object created using object initializer or literal notation syntax looks like this:

```
const fellowshipOfTheRing = {
  title: 'The Fellowship of the Ring',
  series: 'The Lord of the Rings',
  author: 'J.R.R. Tolkien'
};
```

While it's not explicitly stated, the above object literal represents a "Book" object type. An object type is defined by its attributes and behaviors. This particular "Book" object type has "title", "series", and "author" attributes which are represented by the object literal's `title`, `series`, and `author` properties.

Behaviors are represented by methods, but this particular object literal doesn't define any methods. We'll see an example of an object type behavior later in this article.

A constructor function in JavaScript handles the creation of an object—it's a "factory" for creating objects of a specific type. Calling a constructor function returns an object with its properties initialized to the provided argument values along with any available methods for operating on the object's data.

Here's an example of a constructor function for the "Book" object type:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}
```

This `Book` constructor function is responsible for creating "Book" objects. If your application had four unique object types, then you'd typically declare four constructor functions—one constructor function for each unique object type.

While the `Book` constructor function uses JavaScript's standard syntax for function declarations, there are a few things specific to constructor functions worth highlighting:

- **The name of the constructor function is capitalized.** Following this convention will help you (and other developers) to correctly identify this function as a constructor function.
- **The function doesn't explicitly return a value.** When invoked with the `new` keyword, constructor functions implicitly return the newly created object. In just a bit, you'll see an example of this.
- **Within the constructor function's body, the `this` keyword references the newly created object.** This allows you to initialize properties on the object.

Invoking a constructor function

Constructor functions are designed to be invoked with the `new` keyword:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

const fellowshipOfTheRing = new Book(
  'The Fellowship of the Ring',
  'The Lord of the Rings',
  'J.R.R. Tolkien');

console.log(fellowshipOfTheRing); // Book { title: 'The Fellowship of the Ring',
```

Four things occur when invoking a constructor function with the `new` keyword:

1. A new empty object is created (i.e. `{}`);
2. The new object's prototype is set to the object referenced by the constructor function's `prototype` property (more about this in just a bit);
3. The constructor function is called and `this` is bound to the new object; and
4. The new object is returned after the constructor function has completed.

Important: *If you return something from a constructor function then you'll break the behavior described in item #4 as the return value will be whatever you're explicitly returning instead of the new object.*

Understanding object type instances

Remember that a constructor function handles the creation of an object—it's a "factory" for creating objects of a specific type. An object created from a constructor function is said to be an **instance** of the object type defined by the constructor function.

In the below example, the `Book` constructor function defines a `Book` object type. Calling the `Book` constructor function with the `new` keyword creates an instance of the `Book` object type:

```
// This constructor function defines
// a `Book` object type.
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

// Use the `new` keyword to create
// three instances of the `Book` object type.

const fellowshipOfTheRing = new Book(
  'The Fellowship of the Ring',
  'The Lord of the Rings',
  'J.R.R. Tolkien');
```

```
const twoTowers = new Book(
  'The Two Towers',
  'The Lord of the Rings',
  'J.R.R. Tolkien');

const returnOfTheKing = new Book(
  'The Return of the King',
  'The Lord of the Rings',
  'J.R.R. Tolkien');

// Logging each instance to the console
// shows that each is a `Book` object type.

console.log(fellowshipOfTheRing); // Book { title: 'The Fellowship of the Ring',
console.log(twoTowers); // Book { title: 'The Two Towers', ... }
console.log(returnOfTheKing); // Book { title: 'The Return of the King', ... }

// Comparing each instance to the others
// shows that each instance is a unique object
// and not equal to the others even though they
// are all `Book` object types.

console.log(fellowshipOfTheRing === twoTowers); // false
console.log(fellowshipOfTheRing === returnOfTheKing); // false
console.log(twoTowers === returnOfTheKing); // false
```

In this example, the `new` keyword is used to create three instances of the `Book` object type, which are referenced by the `fellowshipOfTheRing`, `twoTowers`, and `returnOfTheKing` variables. While each instance is a `Book` object type, they are also unique objects and therefore not equal to each other.

Using the `instanceof` operator to check an object's type

Sometimes it's helpful to know if an object is an instance of a specific type. JavaScript makes this easy to do using the `instanceof` operator:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Use the `instanceof` operator to check if the
// `fellowshipOfTheRing` object is an instance of `Book`.
console.log(fellowshipOfTheRing instanceof Book); // true
```

The `instanceof` operator allows us to confirm that calling the `Book` constructor with the `new` keyword creates an instance of the `Book` object type.

Invoking a constructor function without the `new` keyword

We can use the `instanceof` operator to prevent our constructor functions from being misused.

Invoking a constructor function without the `new` keyword results in one of two unexpected outcomes:

- When running in non-strict mode, `this` will be bound to the global object **not** the newly created object; or

- When running in strict mode, `this` will be `undefined`, which results in a runtime error when attempting to initialize a property on the newly created object using the `this` keyword.

You can write the string below at the top of your file to enable strict mode for an entire script or inside a function body for function-level strict mode.

Strict mode can be enabled by writing the following string:

```
"use strict";
```

Because the second outcome results in an error when calling the constructor function, it's a bit easier to debug than the first outcome. Up until now, we've only seen errors generated by JavaScript. With the `throw` keyword and the `Error` constructor function, we can throw our own custom errors:

```
function Book(title, series, author) {
  if (!(this instanceof Book)) {
    // Throws a custom error when `Book` is called without the `new` keyword.
    throw new Error('Book needs to be called with the `new` keyword.');
  }

  this.title = title;
  this.series = series;
  this.author = author;
}

// Calling the `Book` constructor method with the `new` keyword
// successfully creates a new instance.
const fellowshipOfTheRing = new Book(
  'The Fellowship of the Ring',
  'The Lord of the Rings',
  'J.R.R. Tolkien');

// Calling the `Book` constructor method without the `new` keyword
// throws an error with the message
// "Book needs to be called with the `new` keyword."
const fellowshipOfTheRing = Book(
  'The Fellowship of the Ring',
  'The Lord of the Rings',
  'J.R.R. Tolkien');
```

In this example, an `if` statement has been added to the `Book` constructor function that checks if `this` isn't bound to an instance of the `Book` constructor and throws an error explaining the problem.

Defining sharable methods

When defining the behavior or methods for an object type, avoid the temptation to define the methods within the constructor function:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;

  // For example only!
  // To avoid inefficient use of computer memory
  // don't define methods directly on the newly created object.
  this.getInformation = function() {
    return `${this.title} by ${this.author}`;
  };
}
```

Remember that a method is a function that's associated with a specific object using a property.

Using this approach is inefficient in terms of computer memory usage as each object instance would have its own method definition. If you had a hundred object instances there'd be a hundred method definitions! A better approach is

to define the method once and then share that method definition across all instances of that object type.

Let's explore how prototypes can be used to define sharable methods.

Prototypes and delegation

In JavaScript, a **prototype** is an object that is delegated to when a reference to an object property or method can't be resolved.

For example, if a property or method isn't available on an object, JavaScript will delegate to the object's prototype to see if that object has the requested property or method. If the property or method is found on the prototype, then the action is carried out on the prototype object. The delegation to the prototype happens automatically, so from the caller's perspective it looks as if the original object had the request property or method.

In JavaScript, you can make an object the prototype of another object. When an object is a prototype of another object, it's properties and methods are made available to the other object.

Here's a simple, arbitrary example involving two object literals: `a` and `b`. Object `a` defines a method named `alpha()` and object `b` defines a method named `beta()`:

```
const a = {
 alpha() {
   return 'Alpha';
 }
};

const b = {
 beta() {
   return 'Beta';
 }
};
```

The first time that you attempt to call the `alpha()` and `beta()` methods on object `a`, only the call to the `alpha()` method succeeds as the `beta()` method is only defined on object `b`:

```
console.log(a.alpha()); // Alpha
console.log(a.beta());  // Error: a.beta is not a function
```

When you check the data type of `a` or `b`, you see that they are objects. This means you can access the `alpha()` and `beta()` with property accessors using `dot notation` or `bracket notation`.

```
console.log(typeof a); // Prints 'object'

// Dot notation
a.alpha(); // Alpha

// Bracket notation
a["alpha"](); // Alpha
```

After using the `Object.setPrototypeOf()` method to set object `b` as the prototype of `a`, the call to the `beta()` method on object `a` succeeds:

```
// For example only!
// Calling the `Object.setPrototypeOf()` method can have
// a negative impact on the performance of your application.
Object.setPrototypeOf(a, b);

console.log(a.alpha()); // Alpha
console.log(a.beta()); // Beta
```

Important: The `Object.setPrototypeOf()` method is used in this example for demonstration purposes only. *Calling the `Object.setPrototypeOf()` method can have a negative impact on the performance of your application, so you should generally avoid using it.*

The call to `beta()` method works now because when the method isn't found on object `a`, the call is delegated to object `a`'s prototype which is object `b`. The `beta()` method is found on object `b` and it's successfully called.

Starting with ES2015, you can use the `Object.getPrototypeOf()` method to get an object's prototype. Calling the `Object.getPrototypeOf()` method and passing object `a` allows us to verify that object `a`'s prototype is object `b`:

```
// Use the `Object.getPrototypeOf()` method
// to get the prototype of object `a`.
console.log(Object.getPrototypeOf(a)); // { beta: [Function: beta] }
```

An object's prototype is sometimes referred to in writing using the notation `[[prototype]]`. For example, MDN Web Docs' JavaScript documentation will sometimes refer to an object's prototype as its `[[prototype]]`.

The `__proto__` property

Prior to ES2015 and the addition of the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods, there wasn't an official way to get or set an object's internal `[[prototype]]` object. As a workaround, many browsers (including Google Chrome and Mozilla Firefox) made available a `__proto__` property providing an easy way to access an object's `[[prototype]]`:

```
// For example only!
// The `__proto__` property is deprecated in favor of
// the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods.
console.log(a.__proto__); // { beta: [Function: beta] }
```

While the `__proto__` property is widely supported by browsers and handy to use when debugging, you should never use it in your code as it's deprecated in favor of the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods.

Code that relies upon the deprecated `__proto__` property will unexpectedly stop working if any of the browser vendors decide to remove the property from their implementation of the JavaScript language specification. When the need arises, use the `Object.getPrototypeOf()` method to get an object's prototype.

Instead of having to say "underscore underscore proto underscore underscore" or "double underscore proto double underscore" when referring to the `__proto__` property, developers will sometimes say "dunder proto".

Defining sharable methods on a constructor function's `prototype` property

Let's use what you've learned about prototypes and delegation in JavaScript to define methods for an object type that'll be shared across all of its instances.

Every constructor function has a `prototype` property that represents the object that'll be used as the prototype for instances created by invoking the constructor function with the `new` keyword. We can confirm this by comparing the prototype for an instance created from a constructor function to the constructor function's `prototype` property:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

const twoTowers = new Book(
 'The Two Towers',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Get the prototypes for both `Book` instances.
const fellowshipOfTheRingPrototype = Object.getPrototypeOf(fellowshipOfTheRing);
const twoTowersPrototype = Object.getPrototypeOf(twoTowers);

// Compare the `fellowshipOfTheRing` and `twoTowers` prototypes
// to the `Book` constructor function's `prototype` property.
console.log(fellowshipOfTheRingPrototype === Book.prototype); // true
console.log(twoTowersPrototype === Book.prototype); // true

// Compare the `fellowshipOfTheRing` and `twoTowers` prototypes
// to each other.
console.log(fellowshipOfTheRingPrototype === twoTowersPrototype); // true
```

This example shows that:

- Every instance created by a constructor function has its prototype (i.e. `[[prototype]]`) set to the object referenced by the constructor function's `prototype` property; and
- The object referenced by the constructor function's `prototype` property isn't copied when it's set as an instance's prototype—every instance's prototype references the same object.

This means that any method that we define on the constructor function's `prototype` property will be shared across all instances of that object type:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

// Any method defined on the `Book.prototype` property
// will be shared across all `Book` instances.
Book.prototype.getInformation = function() {
 return `${this.title} by ${this.author}`;
};

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

console.log(fellowshipOfTheRing.getInformation()); // The Fellowship of the Ring
```

When the `getInformation()` method is called, the `fellowshipOfTheRing` object is checked first to see if the method is defined on that object. When the method isn't found, the method call is delegated to the instance's prototype, which is set to the `Book` constructor function's `prototype` property. This time, the `getInformation()` method is found and called.

Notice that we can use the `this` keyword in our shared `getInformation()` method implementation to access properties (or methods) on the instance that we're calling the method on.

The problem with arrow functions

If you're like me, you like the concise syntax of arrow functions. Unfortunately, you can't use arrow functions when defining methods on a constructor function's `prototype` property:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

// For example only!
// Using an arrow function to define a method
// on a constructor function doesn't work as expected
// when using the `this` keyword.
Book.prototype.getInformation = () => `${this.title} by ${this.author}`;

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Oops! Not what we expected.
console.log(fellowshipOfTheRing.getInformation()); // undefined by undefined
```

Remember that arrow functions don't have their own `this` binding—they use the `this` binding from the enclosing lexical scope. This is why the `this` keyword within the `getInformation()` method doesn't work as expected in the above example as it doesn't reference the current instance (the object instance created by the `Book` constructor function).

For more information on arrow functions, the `this` keyword, and lexical scoping, see this page on MDN Web Docs.

This problem is easily avoided—just stick with using the `function` keyword when defining methods on a constructor function's `prototype` property.

Prototypes and the `instanceof` operator

Earlier, you saw an example of how the `instanceof` operator can be used to check if an object is an instance of a specific type:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}

const fellowshipOfTheRing = new Book(
  'The Fellowship of the Ring',
  'The Lord of the Rings',
  'J.R.R. Tolkien');

// Use the `instanceof` operator to check if the
// `fellowshipOfTheRing` object is an instance of `Book`.
console.log(fellowshipOfTheRing instanceof Book); // true
```

The `instanceof` operator uses prototypes to determine if an object is an instance of a specific constructor function. To do that, the `instanceof` operator checks if the prototype of the object on the left side of the operator is set to the `prototype` property of the constructor function on the right side of the operator.

What you learned

In this article, you learned

- how to define a constructor function for an object type that initializes one or more properties;
- how to invoke a constructor function using the `new` keyword;
- how to use the `instanceof` operator to check if an object is an instance of a specific object type; and
- how to define sharable methods on the `prototype` property of a constructor function.

Putting the Class in JavaScript Classes

For years, JavaScript developers used constructor functions and prototypes to mimic classes. Starting with ES2015, support for classes were added to the language, giving developers an official way to define classes.

When you finish this article, you should be able to:

- Define an ES2015 class containing a constructor method that initializes one or more properties;
- Instantiate an instance of a class using the `new` keyword;
- Define instance and static class methods;
- Understand that ES2015 classes are primarily syntactic sugar over constructor functions and prototypes; and
- Use the `instanceof` operator to check if an object is an instance of a specific class.

Defining an ES2015 class

To review, a constructor function in JavaScript handles the creation of an object—it's a "factory" for creating instances of a specific object type. Here's an example of a constructor function for a `Book` object type:

```
function Book(title, series, author) {
  this.title = title;
  this.series = series;
  this.author = author;
}
```

An ES2015 class defines the attributes and behavior for an object type and is used to create instances of that type—just like a constructor function. Classes are defined using the `class` keyword, followed by the name of the class, and a set of curly braces. Here's an example of the above `Book` constructor function rewritten as an ES2015 class:

```
class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }
}
```

Note that you **cannot** use the following syntax inside of classes:

```
// THIS IS BAD CODE. DO NOT COPY. ILLUSTRATIVE USE ONLY.
class MyClass {
    function constructor() {

    }
}
```

or

```
// THIS IS BAD CODE. DO NOT COPY. ILLUSTRATIVE USE ONLY.
class MyClass {
    let constructor = () => {

    }
}
```

Notice that class names, like constructor functions, begin with a capital letter. Following this convention will help you (and other developers) to correctly identify the name as a class.

While not required, the above class definition includes a `constructor` method. Class `constructor` methods are similar to constructor functions in the

following ways:

- `constructor` **methods don't explicitly return a value.** When instantiating class instances with the `new` keyword, `constructor` methods implicitly return the newly created object instance. In just a bit, you'll see an example of this.
- **Within a `constructor` method's body, the `this` keyword references the newly created object instance.** This allows you to initialize properties on the object instance.

Instantiating an instance of a class

To create or instantiate an instance of a class, you use the `new` keyword:

```
class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Output:
// Book {
//  title: 'The Fellowship of the Ring',
//  series: 'The Lord of the Rings',
//  author: 'J.R.R. Tolkien'
// }
console.log(fellowshipOfTheRing);
```

Four things occur when instantiating an instance of a class:

1. A new empty object is created (i.e. `{}`);
2. The new object's prototype is set to the class' `prototype` property value (more about this in just a bit);
3. The `constructor` method is called and `this` is bound to the new object; and
4. The new object is returned after the `constructor` method has completed.

Important: *Just like with constructor functions, if you return something from a `constructor` method then you'll break the behavior described in item #4 as the return value will be whatever you're explicitly returning instead of the new object.*

Attempting to instantiate a class instance without the `new` keyword

You might recall that invoking a constructor function without the `new` keyword produced an unexpected outcome. Unlike constructor functions, attempting to instantiate a class instance without using the `new` keyword results in a runtime error:

```
// This code throws the following runtime error:
// TypeError: Class constructor Book cannot be invoked without 'new'
// Notice the lack of the `new` keyword.
const fellowshipOfTheRing = Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');
```

This default behavior is an example of how ES2015 classes improve upon constructor functions.

Class definitions aren't hoisted

In JavaScript, you can call a function before it's declared:

```
test();

function test() {
  console.log('This works!');
}
```

This behavior is known as hoisting.

Unlike function declarations, class declarations aren't hoisted. The following code will throw an error at runtime:

```
// This code throws the following runtime error:
// ReferenceError: Cannot access 'Book' before initialization
const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }
}
```

This error is easy to avoid: simply get into the habit of declaring your classes**before**you use them.

Defining methods

A class can contain two types of method definitions: instance methods and static methods. So far, when working with constructor functions, you've only seen examples of instance methods.

Defining an instance method

Instance methods, as the name suggests, are invoked on an instance of the class. Instance methods are useful for performing an action on a specific instance.

The syntax for defining a class instance method is the same as the shorthand method syntax for object literals: the method name, the method's parameters wrapped in parentheses, followed by a set of curly braces for the method body. Here's an example of an instance method named `getInformation()`:

```
class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
 }

  getInformation() {
    return `${this.title} by ${this.author}`;
  }
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

console.log(fellowshipOfTheRing.getInformation()); // The Fellowship of the Ring
```

Notice that you can use the `this` keyword within the instance method body to access properties (and methods) on the instance that the method was invoked on.

Instance methods and prototypes

While the code for a class instance method doesn't give any indication of this, instance methods are made available to instances via a shared prototype object. Just like with constructor functions, this approach prevents method definitions from being unnecessarily duplicated across instances, saving on memory utilization.

Because instance methods are actually defined on a shared prototype object, they're sometimes referred to as "prototype" methods.

Defining a static method

Static methods are invoked directly on a class, not on an instance. Attempting to invoke a static method on an instance will result in a runtime error.

The syntax for defining a class static method is the same as an instance method except that static methods start with the `static` keyword. Here's an example of a static method named `getTitles()`:

```
class Book {
 constructor(title, series, author) {
   this.title = title;
   this.series = series;
   this.author = author;
 }
 // Static method that accepts a variable number
 // of Book instances and returns an array of their titles.
 // Notice the use of a rest parameter (...books)
 // to capture the passed parameters as an array of values.
 static getTitles(...books) {
   return books.map((book) => book.title);

 }

 getInformation() {
   return `${this.title} by ${this.author}`;
 }
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

const theTwoTowers = new Book(
 'The Two Towers',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Call the static `Book.getTitles()` method
// to get an array of the book titles.
const bookTitles = Book.getTitles(fellowshipOfTheRing, theTwoTowers);

console.log(bookTitles.join(', ')); // The Fellowship of the Ring, The Two Towers
```

The `getTitles()` static method accepts a variable number of Book instances and returns an array of their titles.

Notice that the method makes use of a rest parameter (`...books`) to capture the passed parameters as an array of values. Using this approach is merely a convenience; the code could be rewritten to require callers to pass in an array of `Book` instances.

Static methods aren't invoked on an instance, so they can't use the `this` keyword to access an instance. You can pass one or more instances into a static method via a method parameter, which is exactly what the above `getTitles()` method does. This allows static methods to perform actions across groups of instances.

Static methods can also be used to perform "utility" actions—actions that are independent of any specific instances but are related to the object type in some way.

Static methods and constructor functions

Static methods aren't unique to ES2015 classes. It's also possible to define static methods when working with constructor functions.

Here's the above example rewritten to use a constructor function:

```
function Book(title, series, author) {
 this.title = title;
 this.series = series;
 this.author = author;
}

// Static methods are defined
// directly on the constructor function.
Book.getTitles = function(...books) {
 return books.map((book) => book.title);
}

// Instance methods are defined
// on the constructor function's `prototype` property.
Book.prototype.getInformation = function() {
 return `${this.title} by ${this.author}`;
};

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

const theTwoTowers = new Book(
 'The Two Towers',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

console.log(fellowshipOfTheRing.getInformation()); // The Fellowship of the Ring

console.log(theTwoTowers.getInformation()); // The Two Towers by J.R.R. Tolkien

// Call the static `Book.getTitles()` method
// to get an array of the book titles.
const bookTitles = Book.getTitles(
 fellowshipOfTheRing, theTwoTowers);

console.log(bookTitles.join(', ')); // The Fellowship of the Ring, The Two Towers
```

Comparing classes to constructor functions

You've already seen how class `constructor`, instance, and static methods behave in a similar fashion to their constructor function counterparts. This is evidence that ES2015 classes are primarily syntactic sugar over constructor functions and prototypes.

"Syntactic sugar" refers to the addition of syntax to a programming language that provides a simpler or more concise way to leverage features that already exist as opposed to adding new features.

We can use the `instanceof` operator to validate how the various elements of a class map to constructor functions and prototypes.

For reference, here's the `Book` class definition that we've been working with in this article:

```
class Book {
 constructor(title, series, author) {
   this.title = title;
   this.series = series;
   this.author = author;
 }

 static getTitles(...books) {
   return books.map((book) => book.title);
 }

 getInformation() {
   return `${this.title} by ${this.author}`;
 }
}
```

First, we can use the `instanceof` operator to verify that the `Book` class is actually a `Function` object, not a special "Class" object or type:

```
console.log(Book instanceof Function); // true
```

We can also use the `instanceof` operator to verify that the `getInformation()` instance method is defined on the underlying `Book` function's `prototype` property:

```
console.log(Book.prototype.getInformation instanceof Function); // true
```

Similarly, we can verify that the `getTitles()` static method is defined on the `Book` function:

```
console.log(Book.getTitles instanceof Function); // true
```

Going even further, we can use the `isPrototypeOf()` method to check if an instance of the `Book` class has its prototype set to the `Book.prototype` property:

```
const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

console.log(Book.prototype.isPrototypeOf(fellowshipOfTheRing)); // true
```

All of this confirms that the `Book` class is simply an alternative way of writing this `Book` constructor function:

```
function Book(title, series, author) {
 this.title = title;
 this.series = series;
 this.author = author;
}

Book.getTitles = function(...books) {
 return books.map((book) => book.title);
}

Book.prototype.getInformation = function() {
 return `${this.title} by ${this.author}`;
};
```

Using the `instanceof` operator to check an object's type

When working with constructor functions, you saw how the `instanceof` operator could be used to check if an object is an instance of a

specific type. This technique also works to check if an object is an instance of a specific class:

```
class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }
}

const fellowshipOfTheRing = new Book(
 'The Fellowship of the Ring',
 'The Lord of the Rings',
 'J.R.R. Tolkien');

// Use the `instanceof` operator to check if the
// `fellowshipOfTheRing` object is an instance of the `Book` class.
console.log(fellowshipOfTheRing instanceof Book); // true
```

Knowing that ES2015 classes are a layer of syntactic sugar over constructor functions and prototypes, this use of the `instanceof` operator probably isn't surprising to you. The `instanceof` operator checks if the prototype of the object on the left side of the operator is set to the `prototype` property of the class on the right side of the operator.

What you learned

In this article, you learned

- how to define an ES2015 class containing a constructor method that initializes one or more properties;
- how to instantiate an instance of a class using the `new` keyword;
- how to define instance and static class methods;
- that ES2015 classes are primarily syntactic sugar over constructor functions and prototypes; and
- how to use the `instanceof` operator to check if an object is an instance of a specific class.

The DNA of JavaScript Inheritance

Classes don't have to be defined and used in isolation from one another. It's possible to base a class—a *child* class—upon another class—the *parent* class—so that the child class can access or inherit properties and methods defined within the parent class.

Basing a class upon another class is commonly known as *inheritance*. Leveraging inheritance gives you a way to share code across classes, preventing code duplication and keeping your code DRY (don't repeat yourself).

When you finish this article, you should be able to:

- Define a parent class;
- Use the `extends` keyword to define a child class that inherits from a parent class;
- Understand that inheritance in JavaScript is implemented using prototypes; and
- Define a method in a child class that overrides a method in the parent class.

Defining a parent class

Imagine that you recently started a new project developing an application to track your local library's catalog of books and movies. You're excited to get started with coding, so you jump right in and define two classes: `Book` and `Movie`.

Defining the `Book` and `Movie` classes

The `Book` class contains `title`, `series`, and `author` properties and a `getInformation()` method. The `getInformation()` method returns a string containing the `title` and `series` property values if the `series` property has a value. Otherwise, it simply returns the `title` property value. Here's what your initial implementation looks like:

```
class Book {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}
```

The `Movie` class contains `title`, `series`, and `director` properties and a `getInformation()` method which behaves just like the `Book.getInformation()` method. Here's your initial implementation:

```
class Movie {
  constructor(title, series, director) {
    this.title = title;
    this.series = series;
    this.director = director;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
```

```
  }
}
```

To help facilitate a quick test, you instantiate an instance of each class and log to the console a call to each instance's `getInformation()` method:

```
const theGrapesOfWrath = new Book('The Grapes of Wrath', null, 'John Steinbeck');
const aNewHope = new Movie('Episode 4: A New Hope', 'Star Wars', 'George Lucas');

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars)
```

To test your code, you use Node.js to execute the JavaScript file that contains your code (`index.js`) from the terminal by running the command `node index.js`. Here's the output in the terminal window:

```
The Grapes of Wrath
Episode 4: A New Hope (Star Wars)
```

Feeling good about the progress that you've made on the project, you decide to take a break and grab a snack and something to drink. Upon your return, you review your code for the `Book` and `Movie` classes and quickly notice that the classes both contain the following members (i.e. properties and methods):

- The `title` and `series` properties; and
- The `getInformation()` method.

Your code isn't very DRY (don't repeat yourself)!

Defining the `CatalogItem` parent class

While this is an arbitrary example, it illustrates a situation that often arises in software development projects. It can be difficult to anticipate when and where code duplication will occur.

While the `Book` and `Movie` classes represent two different types of items found in the library's catalog, they're also both catalog items. That commonality between the classes allows you to leverage inheritance to keep your code DRY.

Inheritance is when a class is based upon another class. When a class inherits from another class, it gets access to its properties and methods.

To use inheritance, you'll create a new `CatalogItem` parent class and move the `title` and `series` properties and the `getInformation()` method into that class. Then you'll update the `Book` and `Movie` classes to inherit from the `CatalogItem` parent class.

Here's what the implementation for the `CatalogItem` parent class looks like:

```
class CatalogItem {
  constructor(title, series) {
    this.title = title;
    this.series = series;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}
```

Inheriting from a class

Now you need to update the `Book` and `Movie` classes to inherit from the `CatalogItem` parent class. To do that, you'll make the following changes to the `Book` class:

- Use the `extends` keyword to indicate that the `Book` class inherits from the `CatalogItem` class;
- Use the `super` keyword to call the `CatalogItem` class's `constructor()` method from within the `Book` class's `constructor()` method; and
- Remove the `getInformation()` method from the `Book` class.

To indicate that the `Book` class inherits from the `CatalogItem` class, add the `extends` keyword after the class name followed by the name of the parent class:

```
class Book extends CatalogItem {
  constructor(title, series, author) {
    this.title = title;
    this.series = series;
    this.author = author;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}
```

*Remember that class declarations aren't hoisted like function declarations, so you need to ensure that the `CatalogItem` class is declared **before** the `Book` and `Movie` classes or a runtime error will be thrown.*

Since the `Book` class defines a `constructor()` method, the `constructor()` method in the `CatalogItem` parent class must be called before attempting to use the `this` keyword to initialize properties within the `Book` class's `constructor()` method. To call the `constructor()` method in the parent class, use the `super` keyword:

```
class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}
```

Notice that the `title` and `series` parameters are passed to the parent class's `constructor()` method by calling the `super` keyword as a method and passing in the `title` and `series` parameters as arguments. Failing to call the parent class's `constructor()` method before attempting to use the `this` keyword would result in a runtime error.

Lastly, since the `getInformation()` is defined in the `CatalogItem` parent class, the `getInformation()` method can be safely removed from the `Book` class:

```
class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
```

```
  }
}
```

That completes the updates to the `Book` class! Now you can turn your attention to the `Movie` class, which needs to be refactored in a similar way.

Time to practice! *To help reinforce your learning, try to make the changes to the `Movie` class on your own. When you're done, compare your code to the code shown below.*

Here's the updated `Movie` class:

```
class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }
}
```

Go ahead and use Node to re-run your application (`node index.js`) to confirm that the output to the terminal window is unchanged. You should see the following output:

```
The Grapes of Wrath
Episode 4: A New Hope (Star Wars)
```

Great job! You've improved your code without breaking the behavior of your application.

For reference, here's the current state of your code:

```
class CatalogItem {
  constructor(title, series) {
    this.title = title;
    this.series = series;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}

class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
  }
}

class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }
}

const theGrapesOfWrath = new Book('The Grapes of Wrath', null, 'John Steinbeck');
const aNewHope = new Movie('Episode 4: A New Hope', 'Star Wars', 'George Lucas');

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars)
```

The `CatalogItem`, `Book`, and `Movie` classes form a simple class hierarchy. More complicated class hierarchies can include as many as a dozen or more classes.

Understanding how `this` works from within a parent class

Reviewing the `CatalogItem` parent class, you'll notice that the `this` keyword is used both in the `constructor()` and `getInformation()` methods:

```
class CatalogItem {
 constructor(title, series) {
   this.title = title;
   this.series = series;
 }

 getInformation() {
   if (this.series) {
     return `${this.title} (${this.series})`;
   } else {
     return this.title;
   }
 }
}
```

Regardless of where the `this` keyword is used, it always references the instance object (the object created using the `new` keyword). This behavior allows the `constructor()` method in a class—child or parent—to initialize properties on the instance object. It also gives access to instance object properties from within any instance method, regardless if the method is defined in a child or parent class.

Understanding how inheritance works in JavaScript

Earlier in this lesson, you saw how the `instanceof` operator and the `Object.getPrototypeOf()` method could be used to confirm that ES2015 classes are primarily *syntactic sugar* over constructor functions and prototypes.

We can use similar debugging techniques to see how class inheritance in JavaScript is implemented using prototypes.

Syntactic sugar refers to the addition of syntax to a programming language that provides a simpler or more concise way to leverage features that already exist as opposed to adding new features.

For reference, here are the `CatalogItem` and `Book` class definitions that we've been working with in this article:

```
class CatalogItem {
  constructor(title, series) {
    this.title = title;
    this.series = series;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}

class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
  }
}
```

To review, we can use the `instanceof` operator to verify that the `CatalogItem` and `Book` classes are actually `Function` objects, not special "Class" objects or types:

```
console.log(Catalogitem instanceof Function); // true
console.log(Book instanceof Function); // true
```

The underlying function for the `Book` class, like a constructor function, has a `prototype` property. The object referenced by the `Book.prototype` property is used to set the prototype (i.e. `[[prototype]]`) for every instance of the `Book` class.

We can verify this using the `Object.getPrototypeOf()` method:

```
// Create an instance of the Book class.
const theGrapesOfWrath = new Book('The Grapes of Wrath', null, 'John Steinbeck');

// Verify that the prototype of the instance
// references the `Book.prototype` object.
console.log(Object.getPrototypeOf(theGrapesOfWrath) === Book.prototype); // true
```

The `Book` class uses the `extends` keyword to inherit from the `CatalogItem` class. This gives instances of the `Book` class access to the `getInformation()` method defined within the `CatalogItem` class. But how is that accomplished?

Inheritance, prototype chains, and delegation

Just like the `Book` class, the underlying function for the `CatalogItem` class has a `prototype` property. Because the `Book` class inherits from the `CatalogItem` class, the object referenced by the `Book.prototype` property will have its `[[prototype]]` set to the `CatalogItem.prototype` property.

Again, we can verify this using the `Object.getPrototypeOf()` method:

```
console.log(Object.getPrototypeOf(Book.prototype) === CatalogItem.prototype); //
```

The relationships between the `Book` instance, the `Book.prototype` property, and the `CatalogItem.prototype` property form a *prototype chain*.

In fact, the prototype chain doesn't end with the `CatalogItem.prototype` property. The object referenced by the `CatalogItem.prototype` property has its `[[prototype]]` set to the `Object.prototype` property, which is the default base prototype for all objects.

Yet again, we can verify this using the `Object.getPrototypeOf()` method:

```
console.log(Object.getPrototypeOf(CatalogItem.prototype) === Object.prototype);
```

Notice that as we move from the bottom of the prototype chain to the top, from the `Book.prototype` object to the `CatalogItem.prototype` object to the `Object.prototype` object, we move from more specialized objects to more generic objects.

Prototype trivia: `Object.prototype` is the `[[prototype]]` for all object literals and the base `[[prototype]]` for any objects created with the `new` keyword.

Remember that a prototype is an object that's delegated to when a property or method can't be found on an object. We just saw that a class hierarchy defines a prototype chain. A prototype chain in turn defines a series of prototype objects that are delegated to, one by one, when a property or method can't be found on an instance object.

For example, we can call the `getInformation()` method on an instance of the `Book` class, like this:

```
console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
```

The following occurs when the `getInformation()` method is invoked:

- JavaScript looks for the `getInformation()` method on the `theGrapesOfWrath` object.
- When the method isn't found, the method call is delegated to the `theGrapesOfWrath` object's `[[prototype]]` (the object referenced by the `Book.prototype` property).
- When the method isn't found (again), the method call is delegated to the `Book.prototype` property object's `[[prototype]]` (the object referenced by the `CatalogItem.prototype` property).
- This time the method is found (yes!) and the method is successfully called with `this` set to the `theGrapesOfWrath` object.

Using this system of delegation, JavaScript provides a mechanism for objects to "inherit" properties and methods from other objects. Because of its use of prototypes, JavaScript's implementation of inheritance is technically known as *prototypal inheritance*.

Overriding a method in a parent class

Defining a method in a parent class to add behavior to all of its descendant classes is useful and helps to keep your code DRY (don't repeat yourself). But what if a specific child class needs to modify the behavior of the parent class method?

For example, what if the `getInformation()` method for the `Movie` class needs to add the director's name to the end of the string that's returned by the method? One way to satisfy this requirement is to override the parent class's `getInformation()` method in the child class.

Method overriding is when a child class provides an implementation of a method that's already defined in a parent class.

Taking advantage of how delegation works in JavaScript, to override a method in a parent class, you can simply define a method in a child class with the same method name:

```
class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }

  getInformation() {
    // TODO Implement this method!
  }
}
```

Now when the `getInformation()` method is called on an instance of the `Movie` class, the method will be found on the instance's `[[prototype]]` (the object referenced by the `Movie.prototype` property). This stops JavaScript from searching any further up the prototype chain, so the `getInformation()` method that's defined in the `CatalogItem` class isn't even considered.

You can think of the `getInformation()` method that's defined in the `Movie` class as "shadowing" or "hiding" the `getInformation()` method that's defined in the `CatalogItem` class.

Now we need to implement the `getInformation()` method in the `Movie` class. We **could** copy and paste the code from the `getInformation()` method in the `CatalogItem` class as a starting point, but we want to keep our code DRY!

What if we could call the `getInformation()` method in the `CatalogItem` class from within the `getInformation()` method in the `Movie` class? Using

the `super` keyword, we can:

```
class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }

  getInformation() {
    let result = super.getInformation();

    if (this.director) {
      result += ` [directed by ${this.director}]`;
    }

    return result;
  }
}
```

In the above implementation of the `getInformation()` method, the `super` keyword is used to reference the `getInformation()` method that's defined in the parent class—the `CatalogItem` class. We then take that result of calling the `getInformation()` method in the parent class and append the `director` property in brackets (i.e. `[]`) as long as the `director` property actually has a value.

Now we've modified the behavior of a parent class method without having to duplicate code between classes!

What you learned

In this article, you learned

- how to define a parent class;
- how to use the `extends` keyword to define a child class that inherits from a parent class;
- that inheritance in JavaScript is implemented using prototypes; and
- how to define a method in a child class that overrides a method in the parent class.

Using Modules in Node.js

Up until now, you've used Node to run a single JavaScript file that contains all of your code. For trivial Node applications, this approach works fine, but for most Node applications, a different approach is required. Instead of a single, monolithic JavaScript file that contains all of your application code, you'll use multiple files, with each file containing a logical unit of code that defines a module.

When you finish this article, you should be able to:

- Add a local module to a Node.js application;
- Use the `module.exports` property to export from a module;
- Use the `require()` function to import from a module;
- Use modules that export a single item; and
- Understand how module loading works in Node.js.

This article only covers using modules in Node.js. *Later on, you'll learn how to use modules with JavaScript that runs in the browser.*

Introducing Node.js modules

In Node.js, each JavaScript file in a project defines a module. A module's content is private by default, preventing content from being unexpectedly accessed by other modules. Content must be explicitly exported from a module so that other modules can import it. You'll learn how to share content between modules later in this article.

Modules defined within your project are known as *local modules*. Ideally, each local module has a single purpose that's focused on implementing a single bit of functionality. Local modules, along with core and third-party modules, are combined to create your application.

Core and third-party modules

Core modules are the native modules contained within Node.js that you can use to perform tasks or to add functionality to your application. Node contains a variety of core modules, including modules for working with file paths (`path`), reading data from a stream one line at a time (`readline`), reading and writing files to the local file system (`fs`), and creating HTTP servers (`http`).

Developers, companies, and organizations that use Node.js also create and publish modules that you can use in your applications. These *third-party modules* are distributed and managed using npm, a popular package manager for Node.js. You'll learn about npm and package managers in a future lesson.

The CommonJS module system

Recent versions of Node.js actually contain two different module systems. A legacy module system known as *CommonJS* and a newer module system known as *ECMAScript Modules* or simply *ES Modules*. Conceptually, CommonJS and ES Modules are similar, but their syntax and implementation details differ.

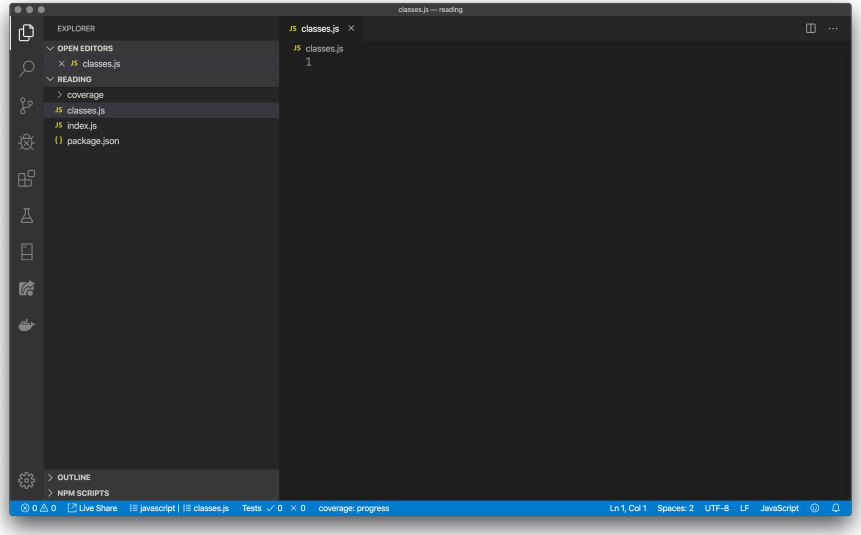
ES Modules will eventually replace CommonJS, but the transition won't happen overnight. Like older versions of JavaScript, CommonJS modules may never disappear completely due to the amount of legacy Node.js code that exists.

You'll start with learning about and using CommonJS modules. In a future lesson, you'll be introduced to ES Modules.

Adding a local module to a Node.js application

To add a local module to a Node application, simply add a new JavaScript file (`.js`) to your project! You can locate the file in the root of the project or within a folder or a nested folder.

Here's a screenshot of adding a `classes` module (`classes.js`) to the root folder of a Node application in Visual Studio Code:



The `classes` module will define the classes for a simple library catalog application, which will be used to track a library's catalog of books and movies.

Here's the code for the `CatalogItem`, `Book`, and `Movie` classes:

classes.js

```
class CatalogItem {
  constructor(title, series) {
    this.title = title;
    this.series = series;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}

class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
  }
}

class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }

  getInformation() {
    let result = super.getInformation();

    if (this.director) {
      result += ` [directed by ${this.director}]`;
    }

    return result;
  }
}
```

The `CatalogItem` class represents an item in the library's catalog. The `CatalogItem` class serves as the parent class to the `Book` and `Movie` classes,

which respectively represent books and movies in the library's catalog.

Code contained within a module is private by default, meaning that it's only accessible to other code contained with that module. If you attempted to reference the `Book` or `Movie` classes in the `index.js` file, you'd get a runtime error.

The `index.js` file is the entry point for the application. A Node application's entry point is the file that's passed to the `node` command (i.e. `node index.js`) when starting an application from the terminal.

Exporting from a module

To make the `Book` and `Movie` classes accessible to other modules in our application, you need to export them.

Each module in Node has access to a `module` object that represents the current module. The `module` object contains a number of properties that provide information about the current module. One of those properties, the `module.exports` property, is used to export items from the module.

To export an item, simply define a property for that item on the `module.exports` object:

classes.js

```
class CatalogItem {
  // Contents removed for brevity.
}

class Book extends CatalogItem {
  // Contents removed for brevity.
}

class Movie extends CatalogItem {
  // Contents removed for brevity.
}

module.exports.Book = Book;
module.exports.Movie = Movie;
```

Node initializes the `module.exports` property to an empty object. If you don't declare and initialize any properties on the `module.exports` object, then nothing will be exported from the module.

The `module.exports` property names don't need to match the class names, but for this specific example, it makes sense to keep the property names consistent with the class names.

Notice that we're intentionally not exporting the `CatalogItem` class. The `CatalogItem` class is the parent class for the `Book` and `Movie` classes and can stay private to this module.

In this example (and the others that follow), we're exporting an ES2015 class, but what you can export from a module isn't restricted to just classes. You can just as easily export a function or an object.

Assigning a new object to the `module.exports` property

Instead of defining properties on the `module.exports` property, you can assign a new object that contains a property for each item that you want to export:

classes.js

```
class CatalogItem {
  // Contents removed for brevity.
}
```

```
class Book extends CatalogItem {
  // Contents removed for brevity.
}

class Movie extends CatalogItem {
  // Contents removed for brevity.
}

module.exports = {
  Book,
  Movie
};
```

Both approaches will look the same to the consumers of the module, so choosing which approach to use is a stylistic choice.

The `exports` shortcut

In addition to the `module.exports` property, Node provides an `exports` variable that's initialized to the `module.exports` property value. Instead of defining properties on the `module.exports` property, you can use the `exports` variable as a shortcut:

classes.js

```
class CatalogItem {
  // Contents removed for brevity.
}

class Book extends CatalogItem {
  // Contents removed for brevity.
}

class Movie extends CatalogItem {
  // Contents removed for brevity.
}

exports.Book = Book;
exports.Movie = Movie;
```

While this is handy, it's important to note that you can't use the `exports` variable if you want to assign a new object to the `module.exports` property:

classes.js

```
class CatalogItem {
  // Contents removed for brevity.
}

class Book extends CatalogItem {
  // Contents removed for brevity.
}

class Movie extends CatalogItem {
  // Contents removed for brevity.
}

// Don't do this!
// Assigning a new value to the `exports` variable
// doesn't update the `module.exports` property.
exports = {
  Book,
  Movie
};
```

To understand why this doesn't work, let's imagine how Node initializes the `module.exports` property and declares and initializes the `exports` variable:

```
// Initialize the `module.exports` property to an empty object.
module.exports = {};
```

```
// Declare and initialize the `exports` variable.
let exports = module.exports;
```

Assigning a new value to the `exports` variable breaks the linkage between the variable and the `module.exports` property:

classes.js

```
// Class definitions removed for brevity.

// Assign a new value to the `exports` variable.
exports = {
  Book,
  Movie
};

// The `module.exports` property still references an empty object.
console.log(module.exports); // {}
```

While the `exports` variable now references your new object, the `module.exports` property still references the empty object that Node assigned to it. This results in nothing being exported from your module.

Just remember to only define properties on the `exports` variable—never assign a new value to it!

Because of this issue, some developers and teams prefer to use the `module.exports` property exclusively and ignore that the `exports` shortcut exists. As long as you understand how to properly use the `exports` variable, it's safe to use in your applications.

Importing from a module

The code for the application's entry point, `index.js`, looks like this:

index.js

```
const theGrapesOfWrath = new Book(
  "The Grapes of Wrath",
  null,
  "John Steinbeck"
);
const aNewHope = new Movie(
  "Episode 4: A New Hope",
  "Star Wars",
  "George Lucas"
);

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars) [dir
```

If you attempted to run your application using the command `node index.js`, you'd receive the following error:

```
ReferenceError: Book is not defined
```

You're attempting to instantiate an instance of the `Book` class but that class is defined in the `classes` module, not the `index` module (the module defined by the `index.js` file).

Each module needs to explicitly state what it needs from other modules by saying "I need this and this to run". When a module needs something from another module, it's said to be dependent on that module. A module's dependencies are the modules that it needs to run.

The `require()` function

The `index` module is dependent upon the `Book` and `Movie` classes, so you need to import them from the `classes` module. To do that, you can use the `require()` function:

index.js

```
// Use the `require()` function to import the `classes` module.
const classes = require("./classes");

// Declare variables for each of the properties
// defined on the `classes` object.
const Book = classes.Book;
const Movie = classes.Movie;

const theGrapesOfWrath = new Book(
  "The Grapes of Wrath",
  null,
  "John Steinbeck"
);
const aNewHope = new Movie(
  "Episode 4: A New Hope",
  "Star Wars",
  "George Lucas"
);

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars) [dir
```

To import from a local module, you pass to the `require()` function a path to the module: `./classes`. The dot in the path means to start in the current folder and look for a module named `classes`. The module name is the name of the file without the `.js` file extension.

You can optionally include the `.js` file extension after the module name, but most of the time it's omitted.

Remember that the `classes` module exports an object (using the `module.exports` property) with two properties, `Book` and `Movie`, which reference the `Book` and `Movie` classes defined within the `classes` module. The object that the `classes` module exports is what's returned from the `require()` function call and captured by the `classes` variable:

index.js

```
const classes = require("./classes");
```

To make it a little easier to reference the `Book` and `Movie` classes, local variables are declared for each:

index.js

```
const Book = classes.Book;
const Movie = classes.Movie;
```

Now if you run your application using the command `node index.js`, you'll see the following output:

```
The Grapes of Wrath
Episode 4: A New Hope (Star Wars) [directed by George Lucas]
```

To review, when a module requires code from another module it becomes dependent on that module. So, in this example, the `index` module has a dependency on the `classes` module—without the `Book` and `Movie` classes this code in the `index` module wouldn't be able to successfully run.

Using destructuring when importing

Instead of declaring a variable for the module that you're importing and then declaring a variable for each individual item that the module exports, you can use destructuring to condense that code to a single statement:

index.js

```
const { Book, Movie } = require("./classes");

const theGrapesOfWrath = new Book(
  "The Grapes of Wrath",
  null,
  "John Steinbeck"
);
const aNewHope = new Movie(
  "Episode 4: A New Hope",
  "Star Wars",
  "George Lucas"
);

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars) [dir
```

Either approach works fine, so this is one of the many stylistic choices you'll make as a developer.

Using single item modules

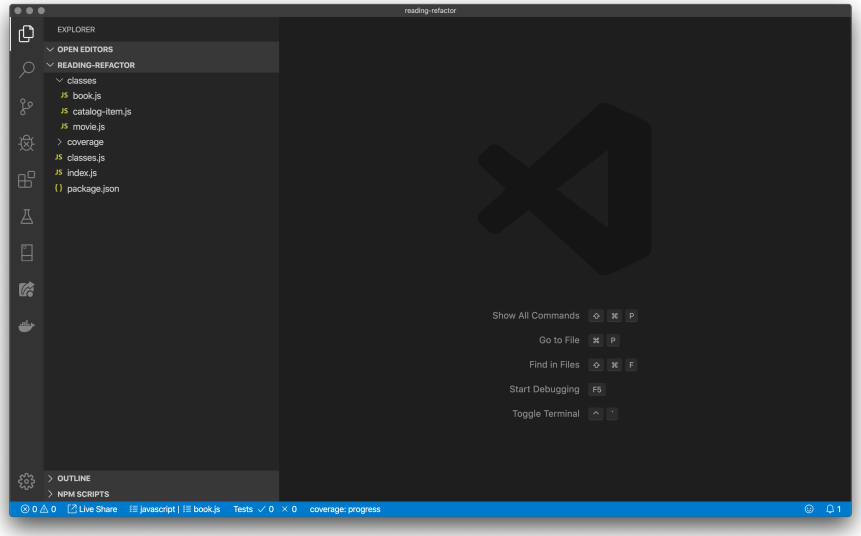
Knowing how best to organize a project using modules is challenging—even for experienced developers. There's also a variety of valid approaches for organizing projects. While there's no single "best" way (each approach has benefits and drawbacks), some developers prefer modules that only export a single item.

Following the convention of a single exported item per module helps to keep modules focused and less likely to become bloated with too much code. This has many advantages including improving the readability and manageability of your code.

Splitting apart the `classes` module

Currently, the `classes` module (defined by the `classes.js` file) defines and exports three classes: `CatalogItem`, `Book`, and `Movie`. Let's split apart the `classes` module so that each class will become its own module.

Start by adding a folder to your project named `classes`. Then add three files to the `classes` folder, one for each class: `catalog-item.js`, `book.js`, and `movie.js`. Here's a screenshot of the project after these changes:



Now you're ready to move the `CatalogItem` class from the `classes` module to the `catalog-item` module. To do that, copy and paste the code for the `CatalogItem` class from one module to the other:

classes/catalog-item.js

```
class CatalogItem {
  constructor(title, series) {
    this.title = title;
    this.series = series;
  }

  getInformation() {
    if (this.series) {
      return `${this.title} (${this.series})`;
    } else {
      return this.title;
    }
  }
}
```

For any module that contains or exports a single item, we can simply assign that item to the `module.exports` property:

classes/catalog-item.js

```
class CatalogItem {
  // Contents removed for brevity.
}

module.exports = CatalogItem;
```

Next, move the `Book` class from the `classes` module to the `book` module, copying and pasting the code for the `Book` class from one module to the other:

classes/book.js

```
class Book extends CatalogItem {
  constructor(title, series, author) {
    super(title, series);
    this.author = author;
  }
}
```

Sharp eyes will notice that the `Book` class inherits from the `CatalogItem` class (using the `extends` keyword). This means that the `book` module has a dependency on the `catalog-item` module.

You can import the `CatalogItem` class using the `require()` function, declaring and initializing a variable for the single item that's exported from the module:

classes/book.js

```
const CatalogItem = require("./catalog-item");

class Book extends CatalogItem {
  // Contents removed for brevity.
}
```

This demonstrates that when importing from a module, you need to be aware if the module exports a single item or multiple items. For local modules, you can review the code for the module you're importing from to determine how the `module.exports` property is being used. For core modules in Node or third-party modules, you'll need to consult the documentation for the module if you're unfamiliar with the module.

Finish up the `book` module by exporting the `Book` class from the module:

classes/book.js

```
const CatalogItem = require("./catalog-item");
```

```
class Book extends CatalogItem {
  // Contents removed for brevity.
}

module.exports = Book;
```

Now you're ready to move the `Movie` class from the `classes` module to the `movie` module. The process and end result, will look a lot like the `book` module:

classes/movie.js

```
const CatalogItem = require("./catalog-item");

class Movie extends CatalogItem {
  constructor(title, series, director) {
    super(title, series);
    this.director = director;
  }

  getInformation() {
    let result = super.getInformation();

    if (this.director) {
      result += ` [directed by ${this.director}]`;
    }

    return result;
  }
}

module.exports = Movie;
```

After moving the classes to their own modules, you can safely remove the `classes` module by deleting the `classes.js` file.

The last change that you need to make is to the `index` module. The `index` module needs to import the `Book` and `Movie` classes from the new modules:

index.js

```
const Book = require("./classes/book");
const Movie = require("./classes/movie");

const theGrapesOfWrath = new Book(
  "The Grapes of Wrath",
  null,
  "John Steinbeck"
);
const aNewHope = new Movie(
  "Episode 4: A New Hope",
  "Star Wars",
  "George Lucas"
);

console.log(theGrapesOfWrath.getInformation()); // The Grapes of Wrath
console.log(aNewHope.getInformation()); // Episode 4: A New Hope (Star Wars) [dir
```

Notice that the `classes` folder name needed to be added to the path that's passed to the `require()` function.

If you run your application again using the command `node index.js`, you'll see the following output (which is unchanged from the previous version of the application):

```
The Grapes of Wrath
Episode 4: A New Hope (Star Wars) [directed by George Lucas]
```

Understanding module loading

In this article, we've focused on creating and using local modules, but that's just one of the available module types. How does Node determine if a module is a local, core, or third-party module?

Module loading logic

When attempting to load a module, Node will examine the identifier passed to the `require()` function to determine if the module is a local, core, or third-party module:

- Local module = identifier starts with `./`, `../`, or `/`
- Node.js core module = identifier matches a core module name
- Third-party module = identifier matches a module in the `node_modules` folder

If the identifier starts with `./`, `../`, or `/`, then Node will attempt to load a local module from the current project using the provided path.

If the identifier passed to the `require()` function isn't a path, then Node will check if the module name matches a core module name.

If the identifier isn't a core module, then Node will attempt to load the module from the `node_modules` folder. The `node_modules` folder is a special folder that the `npm` package manager creates. You'll learn more about npm and the `node_modules` folder in a future lesson.

Module loading process

The first time that a module is imported by another module, Node will load the module, execute the code contained with the module, and return the `module.exports` object to the consuming module. To help improve performance, Node caches modules so that they only need to be loaded and executed once.

An interesting side effect of the module loading process is that code contained within a module is only executed when it's first imported by another module. If a module is never imported by another module—meaning that it's not a dependency of another module—then the code contained within that module won't be executed.

The exception to this rule is the module that's specified as the entry point for your application. Typically that's the `index.js` or `app.js` file located in the root of your project folder. Code in the entry point module is automatically executed by Node when the application is started.

What you learned

In this article, you learned

- how to add a local module to a Node.js application;
- how to use the `module.exports` property to export from a module;
- how to use the `require()` function to import from a module;
- how to use modules that export a single item; and
- how module loading works in Node.js.