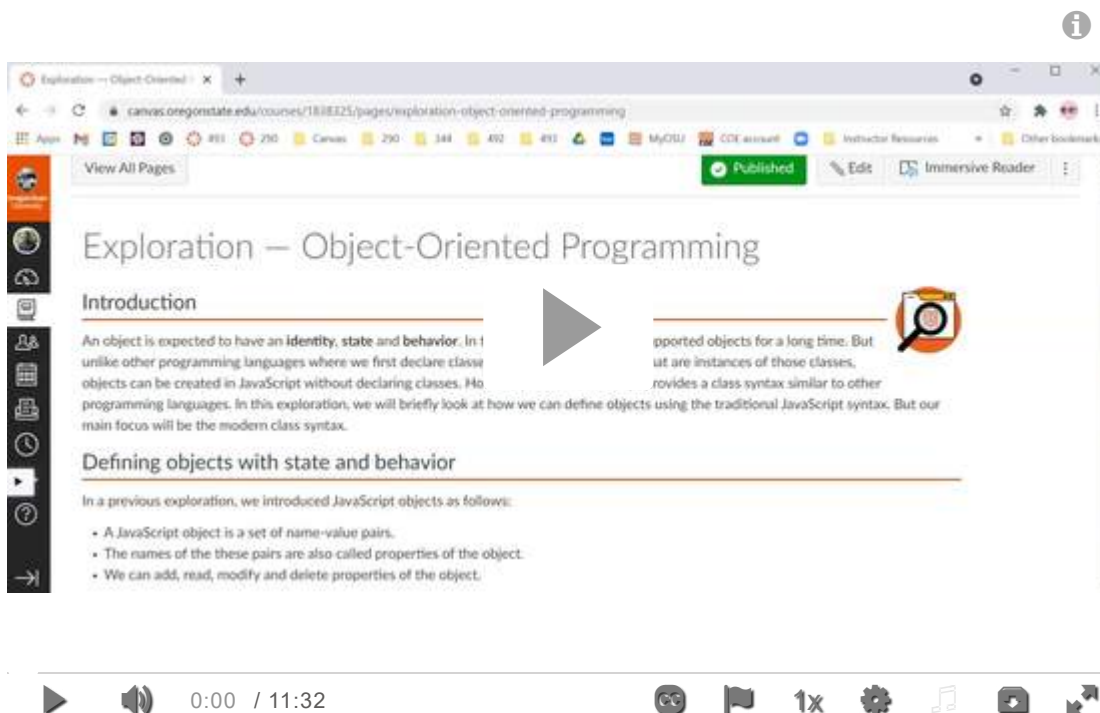


Exploration — Object-Oriented Programming

Introduction



An object is expected to have an **identity**, **state** and **behavior**. In that sense, JavaScript has supported objects for a long time. But unlike other programming languages where we first declare classes and then create objects that are instances of those classes, objects can be created in JavaScript without declaring classes. However, modern JavaScript provides a class syntax similar to other programming languages. In this exploration, we will briefly look at how we can define objects using the traditional JavaScript syntax. But our main focus will be the modern class syntax.



Defining objects with state and behavior

In a previous exploration, we introduced JavaScript objects as follows:

- A JavaScript object is a set of name-value pairs.
- The names of the these pairs are also called properties of the object.
- We can add, read, modify and delete properties of the object.

Let us see if JavaScript objects have identity, state and behavior. We use the following two objects from a previous example to examine this question:

```
let s1 = { company: 'Splunk', symbol: 'SPLK', price: '137.55' };  
let s2 = { company: 'Microsoft', symbol: 'MSFT', price: '232.04' };
```

- Identity
 - The two objects are distinct and have their own identity.
 - `s1===s2` returns false.
- State
 - Each object has its own state which is in the properties, `company`, `symbol`, `price`.
- Behavior
 - Functions are first-class value in JavaScript.
 - We can add behavior to an object by adding properties whose values are functions.

Example

In the following example, we defined a function `totalPrice`. We then added a property `totalPrice`, whose value is a function, to the objects `s1` and `s2`. In the function `totalPrice`, the variable `this` refers to the object which appears to the left of the `.`.

[▶ Run](#)[open in @replit](#)

index.js ×

```
1 'use strict';  
2  
3 // s1 is an object with 3 properties  
4 let s1 = { company: 'Splunk', symbol: 'SPLK', price: 137.55 };  
5  
6 // s2 is an object with 3 properties  
7 let s2 = { company: 'Microsoft', symbol: 'MSFT', price: 232.04 };  
8  
9 // We define a function and assign it to the variable totalPrice
```

Console Shell



Using prototypes to create objects (optional)

Note: The following sections are optional. When developing applications in modern JavaScript, we must use the **class** syntax which is discussed in the following sections of this exploration. Read this section to get a general understanding of the **prototype** mechanism so that you are aware of it when you come across JavaScript code that uses it. However, we don't require a deep understanding of this section. If you just skim this section, it will not negatively impact your understanding of the rest of the course material.

In the above example, we had to add the property `totalPrice` separately to each of the objects `s1` and `s2`. When defining these two objects, we also had to list the other properties of these objects separately, even though both the objects have exactly the same properties. When we have many objects which have the same properties, it is very useful if we could declare the common properties once and then create many objects with those common properties. JavaScript has supported a concept called **prototypes** for a long time that can be used to achieve this goal.

A prototype is a special object that collects properties common to multiple objects. Every JavaScript object has an internal property for its prototype. We can leverage prototypes to create objects with the same properties as follows:


- a. Define a prototype which has all the common function-valued properties (i.e., all properties whose value is a function).
- b. Define a function that
 1. Takes arguments for all the common properties related to state,
 2. Creates an object with these properties and sets the values of the properties to its arguments,
 3. Sets the prototype of this object to the prototype we declared in Step 1, and
 4. Returns the object create in Step b.

Example

Let's say we want to create stock objects that have the properties `company`, `symbol`, `price` and `totalPrice`, where the 1st three properties model the state of the object and `totalPrice` is a function-valued property.

In the following example, we define

1. An object `stockPrototype` which has the function-value property `totalPrice` and will be used as the prototype object for stock object.
2. A function `createStock` which
 - Creates a new stock object,
 - Sets the value of properties to its arguments,
 - Sets the prototype to `stockPrototype`,
 - Returns the new object.

 Runopen in  index.js × 
 1 'use strict';
2
 3 // A prototype for stock objects
const stockPrototype = {
4 ▼ totalPrice: function (count) {
5 ▼ return this.price * count;
}
6 }


Console Shell



Defining Classes (optional)









JavaScript supports defining classes using the keyword `class`.


- A class can have at most one constructor which is (not surprisingly) named `constructor`.
- If a class is declared without a `constructor`, then it automatically gets a constructor with an empty body.
- We create instances of a class by using the keyword `new` followed by the name of the class.

Example

In the following example, we define a class `Stock`.

- The constructor function takes 3 arguments `company`, `symbol`, and `price`.
- The class also has a method `totalPrice`.
- We create new instances of the class by calling the constructor with the keyword `new`, i.e., `new Stock(...)` and passing the values of `company`, `symbol` and `price` as arguments to the constructor.

 Runopen in  replit index.js x 
 1 'use strict';
2
 3  /**
4 * Class Stock
5 */
 6  class Stock {
7  /**
8 *
9 * @param {string} company




Console Shell



Static Methods and Fields (optional)

Static methods and fields [_\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static\)_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static) are properties of a class, and not of individual objects. When a value needs to be shared by all instances of a class, we should define it as a static property. Static properties are defined using the keyword `static`. These properties are accessed by prefixing them with the name of the class. An actual example of a static method that we have already used is **Object.keys()** [_\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys\)_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys) where `keys()` is the static method defined on the class `Object`.

Example

In the following example, the class `Stock` includes a static field `DISCOUNT_PCT` and a static method `discountedPrice`. By convention, static field names use all upper case.

 Runopen in  replit index.js × 
 1 'use strict';
2
3  /**
4 * Class Stock
5 */
6  class Stock {
7 // A static field
8 static DISCOUNT_PCT = 2.5;





Console Shell


Defining subclasses (optional)

JavaScript supports the key object-oriented concept of defining subclasses. We can use the `extend` keyword to define a new class, called the subclass, that extends an existing class, called the superclass. The subclass inherits the properties, fields as well as methods, of the superclass. The subclass can add its own properties. The subclass can also **override** any inherited method by providing its own definition of that inherited method.

Example

In the following example, we define two classes, named `FinancialInstrument` and `Stock`. The class `Stock` extends `FinancialInstrument`. `Stock` adds a property named `exchange` and also overrides the inherited method `totalPrice`. Note that the subclass constructor method must call the superclass's constructor.

 Runopen in  replit index.js x 
   

```
1 'use strict';  
2  
3 /**  
4  * FinancialInstrument  
5  */  
6 class FinancialInstrument {  
7     /**  
8     *  
9     * @param {string} company
```


Console Shell


Exercise (optional)

Use the class syntax to define a class `Point2Dim` which models a point in 2 dimensions.

- Define a constructor that takes 2 parameters, the x-coordinate and the y-coordinate of the point.
- Provide a method `distanceFrom` which takes one parameter, an object of the class `Point2Dim`.
 - This method should return the Euclidean distance in 2 dimensions between the point passed as the argument and the point on which the method is called.







Now define another class `Point3Dim` which models a point in 3 dimensions.

- This class should extend the class `Point2Dim`.
- Override the function `distanceFrom` in `Point3Dim` to return the Euclidean distance in 3 dimensions between the point passed as an argument and the point on which the method is called.

Hints:

- You can find the formulas to compute the Euclidean Distance between two points in [this Wikipedia article](https://en.wikipedia.org/wiki/Euclidean_distance) [_\(https://en.wikipedia.org/wiki/Euclidean_distance\)_](https://en.wikipedia.org/wiki/Euclidean_distance).

- JavaScript [built-in Math functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) [_ \(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) are described on MDN. You can use functions described there to compute powers and square-root.
- You can use the following replit as the starting point of your code

 Runopen in  index.js × 
 



```
1  'use strict';  
2  
3  /**  
4   * Models a point in 2 dimensions  
5   */  
5  class Point2Dim {  
6    /**  
7     *  
8     * @param {number} x The x-coordinate of the point
```

Console Shell

Summary

The textbook specifies Golden Rule 4 as follows:

Golden Rule 4: Understand prototypes, but use modern syntax for classes, constructors, and methods.

This exploration should have given you an understanding of why this rule is important, and also provide you the tools to apply this rule in your JavaScript coding.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- MDN has a detailed discussion of [Classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) [_ \(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) which also includes some of the less important topics that we have skipped in this exploration.