OOP and Prototype

# Exercise 01: a Vector type

Write a class Vec that represents a vector in two-dimensional space. It takes xand y parameters (numbers), which it should save to properties of the same name.

Give the Vec prototype two methods, plus and minus, that take another vector as a parameter and return a new vector that has the sum or difference of the two vectors’ (this and the parameter) *x* and *y* values.

Add a getter property length to the prototype that computes the length of the vector—that is, the distance of the point (*x*, *y*) from the origin (0, 0).

// Input??

// Ouput,

// class Vec

// field // x, y

// constructur(x, y)

// method:

// get/stter, legnth

|  |
| --- |
| // Your code here.  console.log(new Vec(1, 2).**plus**(new Vec(2, 3)));  // => Vec{x: 3, y: 5}  console.log(new Vec(1, 2).**minus**(new Vec(2, 3)));  // => Vec{x: -1, y: -1}  console.log(new Vec(3, 4).**length**);  // => 5 |

# Exercise 02: Groups

The standard JavaScript environment provides another data structure called Set. Like an instance of Map, a set holds a collection of values. Unlike Map, it does not associate other values with those—it just tracks which values are part of the set. A value can be part of a set only once—adding it again doesn’t have any effect.

Write a class called Group (since Set is already taken). Like Set, it has add, delete, and has methods. Its constructor creates an empty group, add adds a value to the group (but only if it isn’t already a member), delete removes its argument from the group (if it was a member), and has returns a Boolean value indicating whether its argument is a member of the group.

Use the === operator, or something equivalent such as indexOf, to determine whether two values are the same.

Give the class a static from method that takes an iterable object as argument and creates a group that contains all the values produced by iterating over it.

|  |
| --- |
| class Group {  // Your code here.  }  let group = Group.from([10, 20]);  console.log(group.has(10));  // → true  console.log(group.has(30));  // → false  group.add(10);  group.delete(10);  console.log(group.has(10));  // → false |

# Exercise 03: Iterable groups

Make the Group class from the previous exercise iterable. Refer to the section about the iterator interface earlier in the chapter if you aren’t clear on the exact form of the interface anymore.

If you used an array to represent the group’s members, don’t just return the iterator created by calling the Symbol.iterator method on the array. That would work, but it defeats the purpose of this exercise.

It is okay if your iterator behaves strangely when the group is modified during iteration.

|  |
| --- |
| // Your code here (and the code from the previous exercise)  for (let value of Group.from(["a", "b", "c"])) {  console.log(value);  }  // → a  // → b  // → c |

# Exercise 04: Borrowing a method

Earlier in the chapter I mentioned that an object’s hasOwnProperty can be used as a more robust alternative to the in operator when you want to ignore the prototype’s properties. But what if your map needs to include the word "hasOwnProperty"? You won’t be able to call that method anymore because the object’s own property hides the method value.

Can you think of a way to call hasOwnProperty on an object that has its own property by that name?

|  |
| --- |
| let map = { one: true, two: true }; Object.prototype. hasOwnProperty  // Fix this call  console.log(  Object.prototype.**hasOwnProperty**.call(map, ‘one’);  Map.**hasOwnProperty() => method call**  **hasOwnProperty() => function call**  );  // → true |

Object.prototype.hasOwnProperty = function() {  
 // refer keyword this;  
}