|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Creating Objects**  To create a new, blank (i.e., “empty”) object, you can use object **literal notation**, or the *Object()* **constructor function***.* If you're not familiar with constructor functions, no need to worry! We'll jump into them in-depth in Lesson 3. For now, just know that the following two expressions are equivalent: | | | | | | |
| const myObject = {}; // Using literal notation:  const myObject = new Object(); // Using the Object() constructor function:  While both methods ultimately return an object without properties of its own, the Object() constructor function is a bit slower and more verbose. As such, the recommended way to create new objects in JavaScript is to use literal notation | | | | | | |
| **Modifying Properties**  Keep in mind that data within objects are mutable, meaning that data can be changed. There are a few exceptions to this, but for now, let's see how we can **modify/reassign existing** properties **in an object.**  Consider the following cat object: | | | | | | |
| **original** | **changes** | | | | | **output** |
| const cat = {  age: 2,  name: 'Bailey',  meow: function () {  console.log('Meow!');  },  greet: function (name) {  console.log(`Hello ${name}`);  }  }; | cat.age += 1;  cat.age; // 3  cat.name = 'Bambi';  cat.name; // 'Bambi' | | | | | {  age: 3,  name: 'Bambi',  meow: function () {  console.log('Meow!');  },  greet: function (name) {  console.log(`Hello ${name}`);  }  }; |
|  | | | |  | | |
| **Adding Properties**  Properties can be added to objects simply by specifying the property name, then giving it a value. Let's start off with a blank object, then add two properties: | | | | | | |
| const printer = {};  printer.on = true;  printer.mode = 'black and white'; | | | | | | |
| The above example uses dot notation to add properties, but keep in mind that square bracket notation works just as well:  ***printer['remainingSheets'] = 168;*** | | | | | | |
| Likewise, we can add a method to the printer object in a similar manner. This time, the value of the property is an anonymous (i.e., unnamed) function:  **printer.print = function () {**  **console.log('The printer is printing!');**  **};** | | | | | | |
| Great! The complete printer object now looks like the following:  **{**  **on: true,**  **mode: 'black and white',**  **remainingSheets: 168,**  **print: function () {**  **console.log('The printer is printing!');**  **}**  **}** | | | | | | |
|  | | | | | | |
| **Removing Properties**  Recall that since objects are mutable, not only can we modify existing properties (or even add new ones) -- we can also delete properties from objects.  Say that the printer object above actually doesn't have any modes (i.e., 'black and white', 'color', etc.). We can go ahead and remove that property from printer using the delete operator. | | | | | | |
|  | | | |  | | |
| **Passing an Object**  On the other hand, objects in JavaScript are mutable. If you pass an object into a function, Javascript passes a reference to that object. Let's see what happens if we pass an object into a function and then modify a property: | | | | | | |
| let originalObject = {  favoriteColor: 'red'  };  function setToBlue(object) {  object.favoriteColor = 'blue';  }  setToBlue(originalObject);  originalObject.favoriteColor;// 'blue' | | | | In example, **originalObject** contains a single *property,* **favoriteColor**, which has a value of 'red'. We pass **originalObject** into the **setToBlue()** function and invoke it. After accessing **originalObjec**t's **favoriteColor** property, we see that the value is now 'blue'! | | |
| Consider this iceCreamOriginal object, which shows the amount of ice cream cones each instructor has eaten:  const iceCreamOriginal = {  Andrew: 3,  Richard: 15  }; | | | | | | |
| const iceCreamCopy = iceCreamOriginal;  iceCreamCopy.Richard;  // 15  As expected, the **expression iceCreamCopy.Richard;** returns **15** (i.e., it is the same value as the Richard property in iceCreamOriginal) | | | | | assign a new variable to iceCreamOriginal. We'll then check the value of its Richard property: | |
| let's change the value in the copy, then check the results: | | | | | iceCreamCopy.Richard = 99;  iceCreamCopy.Richard;  // 99  iceCreamOriginal.Richard;  // 99  making changes to the copy (iceCreamCopy) has a direct effect on the original object (iceCreamOriginal) as well. In both objects, the value of the Richard property is now 99. | |
|  | |  | | |  | |
| Comparing an Object with Another Object | | | | | | |
| const parrot = {  group: 'bird',  feathers: true,  chirp: function () {  console.log('Chirp chirp!');  }  };  const pigeon = {  group: 'bird',  feathers: true,  chirp: function () {  console.log('Chirp chirp!');  }  };  Naturally, one might expect the parrot object and pigeon object to be equal. After all, both objects look exactly the same! Let's compare parrot and pigeon to find out:  parrot === pigeon;  // false | | | | | | |
| let's create a new variable, myBird, and assign it to one of the objects above: | | | | const myBird = parrot; | | |
| As we've just learned, myBird not only refers to the same object as parrot -- they are the same object! If we make any updates to myBird's properties, parrot's properties will be updated with exactly the same changes as well. Now, the comparison will return true: | | | | myBird === parrot;  // true | | |
| So since pigeon is not the same object as myBird or parrot, any comparisons between myBird and pigeon will return false: | | | | myBird === pigeon;  // false | | |
|  | |  | | |  | |
|  | | | | | | |
|  | | | |  | | |
|  | |  | | |  | |
|  | | | | | | |
|  | | | |  | | |
|  | | | | | | |
|  | | |  | | | |
|  | | | | |  | |
|  | | | | | | |
|  | | |  | | | |
|  | | | | |  | |
|  | | | | | | |
|  | | | | |  | |
|  | | | | | | |
|  | | | | |  | |
|  | | | | | | |
|  | | |  | | | |
|  | | | | |  | |
|  | | | | |  | |
|  | | | | |  | |
|  | | | | |  | |
|  | | | | |  | |
|  | | | | |  | |
|  | |  | | |  | |

**QUESTION 1 OF 6**

What is true about modifying objects? Select all that apply:

* Removing properties with the delete operator returns true upon successful deletion.
* With a few exceptions, properties in objects are mutable.

Consider the following house object:

let house = {

color: 'green',

numRooms: 4,

numWindows: 8,

forSale: false

};

**2. Write an expression to delete the numWindows property from house.**

🡪delete house.numWindows;

**3. Consider the updated house object from above:**

**let** house = {

color: 'green',

numRooms: 4,

forSale: false

};

Write an expression to add a new hasGarage property to house. Set the value of the hasGarage property to true.

**QUESTION 4 OF 6**

Which of the following is immutable? Select all that apply:

* 8
* 'How are you today?'
* 3.14
* True

**QUESTION 5 OF 6**

Consider the following:

**let** string = 'orange';

**function** **changeToApple**(string) {

string = 'apple';

}

changeToApple(string);

console.log(string);

*// ???*

**What is logged to the console? 🡪 orange**

### QUESTION 6 OF 6

Consider the following object, oven:

**const** oven = {

type: 'clay',

temperature: 400

};

What is the value of oven's temperature property after the following operations?

**const** newOven = oven;

newOven.temperature += 50;

* 450