JavaScript Advanced Objects

JavaScript destructuring assignment shorthand syntax

The JavaScript *destructuring assignment* is a shorthand syntax that allows object properties to be extracted into specific variable values.
It uses a pair of curly braces (`{}`) with property names on the left-hand side of an assignment to extract values from objects. The number of variables can be less than the total properties of an object.

```
const rubiksCubeFacts = {
  possiblePermutations: '43,252,003,274,489,856,000',
  invented: '1974',
  largestCube: '17x17x17'
};
const {possiblePermutations, invented, largestCube} =
rubiksCubeFacts;
console.log(possiblePermutations); //
'43,252,003,274,489,856,000'
console.log(invented); // '1974'
console.log(largestCube); // '17x17x17'
```

shorthand property name syntax for object creation

The *shorthand property name* syntax in JavaScript allows creating objects without explicitly specifying the property names (ie. explicitly declaring the value after the key). In this process, an object is created where the property names of that object match variables which already exist in that context. Shorthand property names populate an object with a key matching the identifier and a value matching the identifier's value.

```
const activity = 'Surfing';
const beach = { activity };
console.log(beach); // { activity: 'Surfing' }
```

`this` Keyword

The reserved keyword `this` refers to a method's calling object, and it can be used to access properties belonging to that object.
Here, using the `this` keyword inside the object function to refer to the `cat` object and access its `name` property.

```
const cat = {
  name: 'Pipey',
  age: 8,
  whatName() {
    return this.name
  }
};

console.log(cat.whatName());
// Output: Pipey
```

javascript function this

Every JavaScript function or method has a `this` context. For a function defined inside of an object, `this` will refer to that object itself. For a function defined outside of an object, `this` will refer to the global object (`window` in a browser, `global` in Node.js).

```
const restaurant = {
  numCustomers: 45,
  seatCapacity: 100,
  availableSeats() {
    // this refers to the restaurant object
    // and it's used to access its properties
    return this.seatCapacity - this.numCustomers;
  }
}
```

JavaScript Arrow Function this Scope

JavaScript arrow functions do not have their own this context, but use the this of the surrounding lexical context. Thus, they are generally a poor choice for writing object methods.

Consider the example code:

loggerA is a property that uses arrow notation to define the function. Since data does not exist in the global context, accessing this.data returns undefined .

loggerB uses method syntax. Since this refers to the enclosing object, the value of the data property is accessed as expected, returning "abc" .

```
const myObj = {
    data: 'abc',
    loggerA: () => { console.log(this.data); },
    loggerB() { console.log(this.data); },
};

myObj.loggerA();    // undefined
myObj.loggerB();    // 'abc'
```

getters and setters intercept property access

JavaScript getter and setter methods are helpful in part because they offer a way to intercept property access and assignment, and allow for additional actions to be performed before these changes go into effect.

```
const myCat = {
  _name: 'Snickers',
  get name(){
    return this._name
  },
  set name(newName){
    //Verify that newName is a non-empty string before
setting as name property
    if (typeof newName === 'string' && newName.length > 0){
      this._name = newName;
    } else {
      console.log("ERROR: name must be a non-empty string");
    }
  }
}
```

javascript factory functions

A JavaScript function that returns an object is known as a *factory function*. Factory functions often accept parameters in order to customize the returned object.

```
// A factory function that accepts 'name',
// 'age', and 'breed' parameters to return
// a customized dog object.
const dogFactory = (name, age, breed) => {
  return {
    name: name,
    age: age,
    breed: breed,
    bark() {
      console.log('Woof!');
    }
  };
};
```

javascript getters and setters restricted

JavaScript object properties are not private or protected. Since JavaScript objects are passed by reference, there is no way to fully prevent incorrect interactions with object properties.
One way to implement more restricted interactions with object properties is to use *getter* and *setter* methods.
Typically, the internal value is stored as a property with an identifier that matches the *getter* and *setter* method names, but begins with an underscore (_).

```
const myCat = {
  _name: 'Dottie',
  get name() {
    return this._name;
  },
  set name(newName) {
    this._name = newName;
  }
};

// Reference invokes the getter
console.log(myCat.name);

// Assignment invokes the setter
myCat.name = 'Yankee';
```

Print