Objects and object references

Objects

In JavaScript objects are used to store key to value pair collection of data. You can think of them as dictionary in Python.

You can create an object using brackets `{ }`, and a list of optional properties that the object will have. For example:

```
let exampleObj = {
    age: 30,
    name: "John"
};
```

Access/modify the property using dot notation:

```
console.log(exampleObj.age);


exampleObj.age = 50;
```

To delete a property you can use the `delete` operator

```
delete exampleObj.age;
```

You can also create property that is multi-worded, but they must be quoted:

```
let exampleObj = {
    name: "John",
    age: 30,
    "have computer": true
};
```

Accessing multi-worded must use the square bracket notation, since dot notation require key to be a valid variable identifier (which excludes space, digit or special characters).

Single word property you can either use square bracket notation or dot notation, is up to you.

```
console.log(exampleObj["have computer"]);
```

Empty Object

You can create empty object using either of these syntax:

```
let user = new Object();
let user = {};
```

Accessing object property

With the square bracket notation, you can use a variable to query the property. However, you cannot do the same with dot notation, it doesn't work.

```
let key = "name";


console.log(exampleObj[key]); // Print out John
console.log(example.key); // Undefined, because it tries to find property named key
```

Computed properties

To insert a property name that is from a variable you can use the square bracket.

```
let fruit = prompt("What fruit do you want?");


let bag = {
[fruit]: 5
}


console.log(bag.apple);
```

In this case, if the user entered "apple", then the apple property will be inserted into the `bag` object with value of 5.

Property value shorthand

If the variable you are assigning a property attribute to is the same as the property name, like below:

```
function makeUser(name, age) {
return {
    name: name,
        age: age
    }
}
```

You can just ignore writing the property assignment part and just put the variable name like below:

```
function makeUser(name, age) {
  return {
    name,
        age
    }
}
```

They are both equivalent, but it is a shorthand way of writing the other. If the property name is the same as the variable, then you can just use the variable name as a shorthand.

You can mix and match property shorthand with normal property assignment.

Property name limitations

Variables cannot have keyword names. However it is not true for object property, the name can be whatever even keywords!

```
let obj = {
  for: 1,
    let: 2,
    return: 3
}
```

If you use other types as property name such as `0`, they are automatically converted into String so `"0"`.

```
let obj = {
  0: "test"
};

console.log(obj["0"]);
console.log(obj[0]);
// Print out same thing. The 0 in both the property and the square bracket are converted to
String.
```

Property existence test

You can test if an object has a property via two ways:

```
console.log(user.noSuchProperty === undefined); // If this is true, then it has no
"noSuchProperty"
```

```
"key" in object // If this is true, it exists, false it doesn't
```

The `in` operator is more preferred, because there are cases where comparing to `undefined` will fail, for example, if the property's value is `undefined`.

Looping over keys of object

```
for (let key in object) {
  // Executes the body for each "key"
    // Access the value via object[key]
}
```

Object.keys, values, entries

These method provide a generic way of looping over an object. They are called on the `Object` class because it is meant to be generic, the each individual object can write their own while still having this generic way.

- `Object.keys(obj)`: Returns an array of keys
- `Object.values(obj)`: Returns an array of values
- `Object.entries(obj)`: Returns an array of `[key, value]` pairs

Since `Object` lack `map`, `filter`, and other functions that array supports you can simulate it using `Object.entries` follow by `Object.fromEntries`

```
let prices = {
  banana: 1,
    orange: 2,
    meat: 4,
};

let doublePrice = Object.fromEntries(
  Object.entries(prices).map(entry => [entry[0], entry[1] * 2])
);
```

Object references

When you assign a primitive data type to another variable, it makes a new copy of the original value.

However, if you assign another variable an existing object you are making an alias, it is a reference to the original object. It does not make a new copy. So if you change the attribute of the alias, it will also change the original object!

Reference comparsion

Two objects are equal if they are the same object

```
let a = {};
let b = a; // Making a reference of a

a == b; // This is true, because they are referencing the same object
a === b; // Also true.
```

Duplicating object

The function `Object.assign(dest, ...sources)` will take in a target object, in which to copy all the property to. One or more list of source object whose's property to copy into `dest`.

```
let user = { name: "John" };

let perm1 = { canView: true };
let perm2 = { canEdit: true };

Object.assign(user, perm1, perm2);

console.log(user); // Print out "John", true, true
```

However, `Object.assign()` doesn't support deep cloning, if the object we are copying contain another object, then it will be copying the references to the destination object. In order to do deep cloning you will have to use `structuredClone(object)` function to clone all nested objects.

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = structuredClone(user);
```

```
user.sizes == clone.sizes // false, the size object within user object is cloned as well.
```

Method and "this" keyword

You are able to add methods (functions that is a property of an object) to the object.

```
let user = {
    name: "John",
    age: 30
};

// First way of adding method
user.sayHi = function() {
    console.log(`Hi my name is ${this.name}`);
};

// Second way of adding method
let ricky = {
    name: "Ricky",
    age: 22,

    sayHi: function() {
        console.log(`Hi my name is ${this.name}`);
    }
};
```

A shorthand way of writing method for an object is you can skip out the property name:

```
// Second way of adding method
let ricky = {
    name: "Ricky",
    age: 22,

    sayHi() {
        console.log(`Hi my name is ${this.name}`);
    }
};
```

The `sayHi()` function in both `ricky` object are kind of similar but not fully identical, but the shorter syntax is preferred.

The `this` keyword is used to access the object that the method is invoked upon. Using `this` keyword allow you to access the object's property that it is invoked upon. In the previous example it is used to accessed `ricky`'s name property.

"this" is not bound

Unlike other languages like Java or Python, the `this` keyword can be used in any function actually and doesn't have to be for a method.

You can directly write the following function:

```
function sayHi() {
  console.log(this.name);
}
```

Then you can assign it to be an object's method:

```
let user = {name: "John"};
let admin = {name: "Admin"};

function sayHi() {
  console.log("Hi I am " + this.name);
}

user.f = sayHi;
admin.f = sayHi;
user.f(); // Will say "Hi I am John". this == user
admin.f(); // Will say "Hi I am Admin". this == admin
```

`this` will determine which object it is invoked upon at call-time, which object is before the dot basically.

> Calling the same function without an object will make `this == undefined`. If you call sayHi() directly, in the previous example then `this` will be `undefined`. If you do this in browser, then `this` will be assigned the global object `window`. In Nodejs it will also be a global object. It is expected that you call the function in an object context if it is using `this`.

Arrow function have no "this"

If you reference `this` in arrow function, then it inherit the `this` from the outer "normal" function.

```
let ricky = {
    name: "Ricky",
    age: 22,

    sayHi: function() {
    console.log(`Hi my name is ${this.name}`);
    },

    foo() {
        let bar = (x, y) => {
            console.log(this);
        }
        bar();
    }
};
```

In this case if you invoke `ricky.foo()` then the `this` that is being used in the `bar` arrow function will be referring to `ricky` object, since it is inheriting it from the `foo` normal function.

In addition, if you invoke an arrow function that uses "this" directly without it being nested inside any function at all, "this" will be an empty object.

Object to primitive conversion

JavaScript doesn't allow you to customize how operator work on objects. Languages like Ruby, C++, or Python allows you but not JavaScript!

When you are using operator with +, -, * with objects, those objects are first converted into primitive before carrying out the operations.

So the rules for converting object to primitive is as follows

1. Treating object as a boolean is always true. All objects are `true` in boolean context
2. Using object in numerical context, the behavior can be implemented by overwriting the special method
3. For object in string context, the behavior can also be implemented by overwriting the special method

Hints

To decide which conversion that JavaScript apply to the object, hints are provided. There are total of three hints.

1. "string"
   This hint is provided when doing object to string conversion.
2. "number"
   This hint is provided when doing object to number conversion.
3. "default"
   This hint is provided when operator is unsure what type to expect, for example the binary `plus` operator can work both on string and number, so if a binary plus operator encounters an object, the "default" hint is provided.

   > In addition, the `==` operator also uses the "default" hint.

JavaScript conversion

1. If `obj[Symbol.toPrimitive](hint)` method exists with the symbol key `Symbol.toPrimitive` (system symbol), then it is called
2. Otherwise, if the method doesn't exist and the hint is `"string"` `obj.toString()` or `obj.valueOf()` is called whichever exists
3. Otherwise, if the method doesn't exist and the hint is `"number"` `obj.valueOf()` or `obj.toString()` is called whichever exists

Symbol.toPrimitive

If the object have this key to function property then this method is used and rest of the conversion method are ignored.

```
let obj = {
name: "John"
};

obj[Symbol.toPrimitive] = function(hint) {
// Here the code to convert this object to a primitive
    // It MUST return a primitive value!
    // hint can be either "string", "number", "default"
}
```

toString/valueOf

If there is no `Symbol.toPrimitive` then JavaScript will call `toString` and `valueOf`.

- For `"string"` hint `toString` method is called, if it doesnt' exist or it returns an object instead of primitive, then `valueOf` is called
- For other hints `valueOf` method is called, and again if it doesn't exist or it returns an object instead of primitive, then `toString` is called

>

> By default, `toString` returns a String "[object Object]"
> By default, `valueOf` returns the object itself

```
let user = {name: "John"};


"hello".concat(user) // hello[object Object]
user.valueOf() === user // True
```

If you only implement `toString()` then it is sort of like a catch all case to handle all primitive conversion. You can't just implement `valueof()` to handle all primitive conversion since `toString()` return a primitive String by default already.

Revision #6
Created 2022-12-19 21:12:52 UTC by Tamarine
Updated 2022-12-24 19:19:41 UTC by Tamarine