Object Oriented Javascript

Agenda

- Functions
- Classes
- Objects
- new Keyword
- this keyword
- Bind, call & apply methods
- Inheritance
- Events
- Timer functions
- Instance of
- Built in functions

Function

- Reusable piece of code. Used heavily in JS.
- Functions can be defined using the function keyword.
- Functions can **return** a value. This is not mandatory.
- No type needs to be specified for arguments or return values.
- Variables defined inside a function are within that function’s scope only.
- **Calling functions**: just like we’ve been using console.log so far.
- Functions are *first-class* citizens in Javascript.

Function

- Type of functions :
  - **Named** functions : as used earlier in example.
  - **Anonymous** functions : without any name, for one-off use.
    function () { // body here }

    Are generally executed immediately, i.e.

    (function () { // body here })();

Function

- Usages :
  - Function as dataType // var a = function(){};
  - Function as variable // var a = alert;
  - Function as arguments

Classes

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactic sugar over JavaScript's existing prototype-based inheritance. The class syntax *does not* introduce a new object-oriented inheritance model to JavaScript.
- Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.
- The body of a class is executed in strict mode

Classes

- One way to define a class is using a **class declaration**. To declare a class, you use the **class** keyword with the name of the class ("Rectangle" here).
- An important difference between **function declarations** and **class declarations** is that function declarations are hoisted and class declarations are not.

```
class Rectangle {

    constructor(height, width) {

          this.height = height; this.width = width;

     }

}
```

Classes

- A **class expression** is another way to define a class. Class expressions can be named or unnamed.

```
 let Rectangle = class {

  constructor(height, width) {

     this.height = height;

      this.width = width;

   }

}; // Unnamed
```

What is Object ?

An **object** is a collection of properties, and a property is an association between a name (or *key*) and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects.

Creating new objects

1. **Using object initializers**

```
var obj = {id: ‘abc’, name:‘xyz’}
```

1. **Using a constructor function**

```
function Car(make, model, year) {
        this.make = make;
        this.model = model;
        this.year = year;
}

var mycar = new Car("Eagle", "Talon TSi", 1993);
```

1. **Using the Object.create method**

```
var mycar1 = Object.create(mycar);
```

Object prototype

All objects contain a hidden link property. This link points to the prototype member of the constructor of the object.

When you access an object property by the dot or subscript notation:

- If the property is found in the object itself, it is returned.
- Otherwise, the `prototype` link is examined. If the `prototype` has the property, then it is returned.
- Otherwise, it's `prototype` is examined, and so on, until we reach `Object.prototype`. If the property is still not found,`undefined` is returned.

What is “this” !

Few thumb rules to help you understand what this means, in any given situation :

- By default, this always points to the global window object.
- console.log(this === window);
- When you use the ‘dot’ notation to call a function, then this points to the immediate parent object.
- Using the new keyword forces this to point to the newly created object, whenever you call a constructor.

Using this for object references

JavaScript has a special keyword, this, that can be used within a method to refer to the current object

```
function setName(name) {
            this.name = name;
   }
 function getName() {
              return this.name;
  }
setName();
getName();
```

By default(if we don’t specify any object) every property is the part of global object(window)

this ...

- Various uses for *this* :
    - Functions have their own prototype methods.
    - Function.bind, Function.call, Function.apply

Function.bind

- It is used to **fix** the value of this to a specific object (called the context object).

-
  ```
  var person = {
    name: 'Abhishek',
    print: function () {
      console.log('Name is: ', this.name); // this.name means person.name in this context
    }
  };
  ```
- `var p = person.print;`
- `p(); // won't print the name`

- `p = p.bind(person);`
- `p(); // will print 'Abhishek'`

- ``var x = {name: 'Anil', print: p}; // put this `p` function inside a new object``
- `x.print(); // Using dot notation. The print still uses the old context object.`

Function.call

- It is an alternate means to **invoke** the function.
- `Function.call` takes a `context` parameter as the first argument: anything passed here is accessible using `this` inside the function body.
- Any arguments that follow the `context` end up being the arguments to the function itself.

Function.call

```
var sayHello = function (greeting) {
  greeting = greeting || 'Hello';
  console.log(greeting, this.name);
};

var abhi = {name: 'Abhishek'};
sayHello.call(abhi); // Hello Abhishek

var anil = {name: 'Anil'};
sayHello.call(anil, 'Hiiiii'); // Hiiiii Anil
```

Function.apply

- Similar to `Function.call`.
- The only difference is that the arguments are taken as an array.

Inheritance

JavaScript objects are dynamic "bags" of properties (referred to as **own properties**). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Inheritance Example

```
function Animal(){
  this.eats = true;
}

function Rabbit(){
 this.canRunFast = true;
}

Rabbit.prototype = new Animal();
var rabbit = new Rabbit();
console.log(rabbit.eats);
console.log(rabbit.canRunFast);
```

Prototype chain

```
function Animal(){
  this.eats = true;
}

function Rabbit(){
 this.canRunFast = true;
}
Rabbit.prototype = new Animal();

function WhiteRabbit(){
  this.color = “white”;
}
WhiteRabbit.prototype = new Rabbit();

var rabbit = new WhiteRabbit();
console.log(rabbit.eats);
console.log(rabbit.canRunFast);
console.log(rabbit.color);
```

Events

- JavaScript is entirely event-driven. It is designed to add interactivity to pages — something gets executed as a response to an user action.
- The most common types of events that you deal with include **interaction** events — mouse actions, keyboard entry, etc.
- Some events are fired by the browser itself — page load, document ready, etc.
- Finally, we have time-based events — timeouts & intervals.

Events

- When doing pure DOM-manipulation, you just need to be aware of 2 methods:
  - addEventListener() and removeEventListener.
- Example :
  - target.addEventListener(type, listener, useCapture);
  - target.removeEventListener(type, listener, useCapture);
  - target — may be a single node in a document, the document itself, a window, or an XMLHttpRequest.
  - type — a string representation of the event’s type.
  - listener — a function that will be called when this event occurs.
  - useCapture — optional boolean. Decides whether the capture or bubbling phase is used. Default value is false.

Timers

- **setInterval** :
    - is used to call a function repeatedly after a certain interval of time..

    -
      ```
      function iAmAlive() {
       console.log('I am alive');
      }
      setInterval(iAmAlive, 5000); // call iAmAlive every 5 seconds
      ```
    - This function also returns a *handle*. At any time, **clearInterval** function can be called with this handle in order to cancel execution.

Timers

- **setTimeout** :
  - is used to call a function after a certain amount of time has passed.
  - function iHaveBeenCalled() {
     console.log('I have been called');
    }
    setTimeout(iHaveBeenCalled, 2000); // call iHaveBeenCalled after 2000 ms
  - This function returns a *handle*. At any time, **clearTimeout** function can be called with this handle in order to cancel execution.

Instance of

- The **instanceof operator** tests whether the prototype property of a constructor appears anywhere in the prototype chain of an object.
- The instanceof operator tests the presence of **constructor.prototype** in object's prototype chain.
- The value of an **instanceof** test can change based on changes to the prototype property of constructors, and it can also be changed by changing an object prototype using Object.setPrototypeOf.

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
var auto = new Car('Honda', 'Accord', 1998);

console.log(auto instanceof Car);  // expected output: true
console.log(auto instanceof Object);  // expected output: true
```

Built in functions

Object.keys

- The **Object.keys()** method returns an array of a given object's own property **names**

```
const object = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.keys(object));  // expected output: Array ["a", "b", "c"]
```

Built in functions

Object.seal

- The **Object.seal()** method seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable.
- Values of present properties can still be changed as long as they are writable.
- The prototype chain remains untouched. However, the `__proto__` property is sealed as well.

```
const object = { property: 42 };
Object.seal(object);


object.property = 33;
console.log(object.property);  // expected output: 33


delete object.property;  // cannot delete when sealed
console.log(object.property);  // expected output: 33
```

Built in functions

Object.freeze

- The **Object.freeze()** method **freezes** an object. A frozen object can no longer be changed.
- It prevents new properties from being added to it, existing properties from being removed or changed, prevents changing the enumerability, configurability, or writability of existing properties.
- In addition, freezing an object also prevents its prototype from being changed. freeze() returns the same object that was passed in.

```
const object = { property: 42 };

const frozenObject = Object.freeze(object1);

frozenObject.property = 33;  // Throws an error in strict mode

console.log(frozenObject.property1);  // expected output: 42
```

ASSIGNMENT:

Q1. Create a hierarchy of person, employee and developers.

Q2. Given an array, say [1,2,3,4,5]. Print each element of an array after 3 secs.

Q3. Explain difference between Bind and Call (example).

Q4. Explain 3 properties of argument object.

Q4. Create a function which returns number of invocations and number of instances of a function.

Q5. JS code to create a watch.

Thank You