|  |
| --- |
|  |
|  | **OOPS IN JAVASCRIPT** |
|  |  |
|  | LINK: https://developer.mozilla.org/ms/docs/Web/JavaScript/Introduction\_to\_Object-Oriented\_JavaScript |
|  |  |
|  | Terminology |
|  | **Namespace** |
|  | A container which lets developers bundle all functionality under a unique, application-specific name. |
|  | **Class** |
|  | Defines the object's characteristics. A class is a template definition of an object's properties and methods. |
|  | **Object** |
|  | An instance of a class. |
|  | **Property** |
|  | An object characteristic, such as color. |
|  | **Method** |
|  | An object capability, such as walk. It is a subroutine or function associated with a class. |
|  | **Constructor** |
|  | A method called at the moment an object is instantiated. It usually has the same name as the class containing it. |
|  | **Inheritance** |
|  | A class can inherit characteristics from another class. |
|  | **Encapsulation** |
|  | A method of bundling the data and methods that use the data. |
|  | **Abstraction** |
|  | The conjunction of an object's complex inheritance, methods, and properties must adequately reflect a reality model. |
|  | **Polymorphism** |
|  | Poly means "many" and morphism means "forms". Different classes might define the same method or property. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **Object:**  Expect Primitive types(string,boolean,number,null,undefined,symbol), All are objects in Javascript. |
|  | Javascript Can have properties And methods. |
|  | Defination: |
|  | Object is a non-primitive data type in JavaScript. It is like any other variable, |
|  | the only difference is that an object holds multiple values in terms of properties and methods. |
|  | Properties can hold values of primitive data types and methods are functions. \*/ |
|  |  |
|  |  |
|  | //creating emptyobject: var car={}; |
|  |  |
|  | **creating object:** |
|  | We can Create an object in number of ways |
|  |  |
|  | 1. using object literal notation  2. using Constructor  3. using object.create(); |
|  | 1. |
|  | var Car={ |
|  | name:'BMW', |
|  | year:'2018', |
|  | details:function(){ |
|  | console.log(this.name + ' - ' +this.year + ' - ' + this.owner); // LITERAL NOTATION |
|  | } |
|  | }; |
|  |  |
|  | Car.name | Car['name'] // Two ways of Accessing object property |
|  | Car.owner="suresh"; // Add property to 'Cars' object |
|  | delete Car.year; // Delete Propery From Object |
|  | Car.details(); |
|  |  |
|  | // Output: BMW - 2018 |
|  | // in above 'name','year' are properties And 'details' is a methods. And 'Car' is an object. |
|  |  |
|  | **2. using Constructor** |
|  | //Constructor is a function that creates and initializes the newly created object. |
|  | var person=new Object(); // Object Creation using Constructor Function |
|  |  |
|  | // Attach properties and methods to person object |
|  |  |
|  | person.age=20; |
|  | person.name="rajesh"; |
|  | person["address"]="hyderabad"; |
|  |  |
|  | person.getDetails=function(){ |
|  | return console.log(this.name + " - " + this.address); // Attaching method |
|  | } |
|  | person.getDetails(); // Accessing method |
|  |  |
|  | **Accessing proprties And Methods** |
|  | person.age; // 20 |
|  | person.name; // rajesh |
|  | person.getDetails(); // rajesh - hyderabad |
|  |  |
|  | //hasOwnProperty(): |
|  | //if we are trying to access properety does not exist in the object it returns "undefined". |
|  | // then we can use "hasOwnProperty()" method |
|  |  |
|  | console.log(person.salary); // output: undefined |
|  | console.log(person['age']); // Another way of Accessing property of object |
|  |  |
|  | if(person.hasOwnProperty('salary')){ |
|  | console.log(person.salary); |
|  | } |
|  | else{ |
|  | console.log('no salary property'); |
|  | } |
|  |  |
|  | **Access Object keys** |
|  | for(var key in person){ // key or i |
|  | console.log(key); // output: age name address getDetails |
|  | console.log(person[key]) // output 20 rajesh hyderabad [Function] |
|  | } |
|  |  |
|  | **Creating Costructor function** |
|  | var Hotelinfo=function(name,address,price){ |
|  | this.name=name; |
|  | this.address=address; |
|  | this.price=price; |
|  | } |
|  |  |
|  | Hotelinfo.prototype.country=function(){ |
|  | console.log(this.name); |
|  | } |
|  |  |
|  | var hotel=new Hotelinfo('hayathpark','hyderabad',2000); |
|  | console.log(hotel.address); // o/p: hyderabad |
|  | console.log(hotel.country()); |
|  |  |
|  |  |
|  | Q) **WHAT IS "this" keyword** |
|  | A) 1.this is a keyword in JavaScript. When it is used in a constructor, it refers to the instance that is created with the constructor. |
|  |  |
|  | 2.If you tried to console.log(this) in a constructor, you'd notice its the same as logging the instance itself. |
|  |  |
|  | 3. this in a constructor points to the instance |
|  |  |
|  |  |
|  |  |
|  | function Human (firstName, lastName, age) { |
|  | // Other properties and methods |
|  | console.log("Human Properies:" + this) |
|  | } |
|  |  |
|  |  |
|  | var zell = new Human('Zell', 'Liew', 29) |
|  |  |
|  | /\* |
|  | Two principles with OOP in JS are: |
|  | 1.Object Creation Patter (Encapsulation) |
|  | 2.Object Reuse Pattern (Inheritance) |
|  | \*/ |
|  |  |
|  | // --------------------- xxxxxx ------------------------------------------------------- |
|  | **Q)ENCAPSULATION And ABSTRACTION** |
|  |  |
|  | //"**Encapsulation**" is a concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. |
|  | //Abstraction : Through the process of "abstraction", a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency |
|  |  |
|  |  |
|  | We create objects with methods, properties, and attributes. |
|  | When we make them bundled inside an object it’s known as Encapsulation. |
|  | Encapsulation is the actual act of hiding the irrelevant details. |
|  |  |
|  | **Abstraction** means- hiding implementation using abstract class and interfaces etc. |
|  | When these methods and attributes are abstracted from other objects, this is known as abstraction. |
|  | **Abstraction** is hiding the information or providing only necessary details to the client. |
|  |  |
|  |  |
|  | **Encapsulation** is the packing of data and functions operating on that data into a single component and restricting the access to some of the object's components. |
|  | Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. |
|  |  |
|  | **Abstraction** is a mechanism which represent the essential features without including implementation details. |
|  |  |
|  | Encapsulation:-- Information hiding. |
|  | Abstraction:-- Implementation hiding. |
|  |  |
|  |  |
|  | **Here's an example of encapsulation:** |
|  | 1. |
|  | var prop1 = "one"; //private |
|  | this.prop2= "two" //public |
|  | var tc = new test(); |
|  | var tp1 =tc.prop1; //undefined: because prop1 is private; |
|  |  |
|  | // without using this keyword function and variables are private. |
|  |  |
|  | 2. |
|  | var flights=function(){ |
|  | var arrival="inda"; |
|  | this.destination="London"; |
|  | } |
|  |  |
|  | var airpott=new flights(); |
|  | console.log(airpott.arrival); //undefined (can't Access private variable defined in funtion ->var) |
|  | console.log(airpott.destination); //London ( Access public variable defined in funtion -> this ) |
|  |  |
|  | var abc="sdsds"; |
|  | function funsss(){ |
|  | var abc1="sada"; |
|  | } |
|  | //funsss(); |
|  | console.log(abc); |
|  |  |
|  | //Example for Encapsulation |
|  |  |
|  | const Dog = (name, breed, sound) => { |
|  | const bark = () => console.log(sound); |
|  | return { |
|  | name, breed, sound |
|  | }; |
|  | } |
|  |  |
|  | const dog = Dog('Fido', 'Collie', 'Grrrr'); |
|  | //console.log(dog.bark()); |
|  |  |
|  | /\* --------------------- PROTOTYPES STRAT ------------ ------------ \*/ |
|  |  |
|  | /\* |
|  | Every object has a prototype by default. Since prototypes are themselves objects, |
|  | every prototype has a prototype too. |
|  | (There is only one exception, the default object prototype at the top of every prototype chain. ) |
|  | \*/ |
|  |  |
|  |  |
|  | //The following Employee constructor function constructs Employee object. |
|  |  |
|  | function Employee(name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | /\* There are several ways to add a function to the Employee object. |
|  | One way is as shown below. This works but the problem with this approach is that, |
|  | if you create 100 employee objects there will be 100 copies of getName() function. |
|  | We don't want to be creating copies of functions, instead we want all the objects to share the same function code. |
|  | This can be achieved using JavaScript prototype object. \*/ |
|  |  |
|  | function Employee(name) |
|  | { |
|  | this.name = name; |
|  |  |
|  | this.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  | } |
|  |  |
|  | var e1 = new Employee("Mark"); |
|  | var e2 = new Employee("Sara"); |
|  |  |
|  | console.log("e1.name = " + e1.getName() + "[br/]"); |
|  | console.log("e2.name = " + e2.getName() + "[br/]"); |
|  |  |
|  | Output : |
|  | e1.name = Mark |
|  | e2.name = Sara |
|  |  |
|  | //In this example, getName() function is added just to e1 object, and not to e2 object. So e2.getName() would throw an undefined error. |
|  | function Employee(name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | var e1 = new Employee("Mark"); |
|  |  |
|  | e1.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  |  |
|  | var e2 = new Employee("Sara"); |
|  |  |
|  | console.log("e1.name = " + e1.getName() + "[br/]"); |
|  | console.log("e2.name = " + e2.getName() + "[br/]"); |
|  |  |
|  | //In the following example getName() function is added to the Employee object using the name of the constructor function. This would also result in undefined error. |
|  |  |
|  | function Employee(name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | Employee.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  |  |
|  | var e1 = new Employee("Mark"); |
|  | var e2 = new Employee("Sara"); |
|  |  |
|  | console.log("e1.name = " + e1.getName() + "[br/]"); |
|  | console.log("e2.name = " + e2.getName() + "[br/]"); |
|  |  |
|  | /\* Using the prototype object to add functions : The following are the advantages of using the prototype object to add functions. |
|  | 1. No matter how many objects you create, functions are loaded only once into memory |
|  | 2. Allows you to override functions if required. \*/ |
|  |  |
|  | function Employee(name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | Employee.prototype.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  |  |
|  | var e1 = new Employee("Mark"); |
|  | var e2 = new Employee("Sara"); |
|  |  |
|  | console.log("e1.name = " + e1.getName() + "[br/]"); |
|  | console.log("e2.name = " + e2.getName() + "[br/]"); |
|  |  |
|  | Output : |
|  | e1.name = Mark |
|  | e2.name = Sara |
|  |  |
|  |  |
|  | /\* --------------------- **INHERITANCE STRAT** ------------ ------------ \*/ |
|  |  |
|  |  |
|  |  |
|  | /\* In this video we will discuss Inheritance in JavaScript with an example. |
|  |  |
|  | Object oriented programming languages support inheritance. Since JavaScript is also an object oriented programming language, it supports inheritance. |
|  | In Object oriented programming languages like C# and Java to implement inheritance, a class inherits from another class. |
|  | In JavaScript, we don't have the concept of classes, so inheritance in JavaScript is prototype-based. |
|  | This means to implement inheritance in JavaScript, an object inherits from another object. Let us understand this with an example. |
|  | \*/ |
|  | // Employee will be the base object (Similar to base class in c#) |
|  | var Employee = function (name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | // getName() function is added to the base object (Employee) |
|  | Employee.prototype.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  |  |
|  | // PermanentEmployee will be the derived object |
|  | var PermanentEmployee = function (annualSalary) |
|  | { |
|  | this.annualSalary = annualSalary; |
|  | } |
|  |  |
|  | // Use prototype to associate Employee as the base object for PermanentEmployee |
|  | PermanentEmployee.prototype = new Employee("Mark"); |
|  |  |
|  | var pe = new PermanentEmployee(50000); |
|  | // Derived object (permanentEmployee) can see the base object (Employee) getName() method |
|  | console.log(pe.getName()); |
|  | alert(pe instanceof Employee); // Returns true |
|  | alert(pe instanceof PermanentEmployee); // Returns true |
|  |  |
|  | /\* Notice that the derived object (PermanentEmployee) can see the base object (Employee) getName() method. |
|  | When getName() method is called, JavaScript first tries to find this method in the derived object (). |
|  | It can't find the method there so it goes to the parent object and finds it there. |
|  | If you add a new method to the parent object, it becomes available in the derived object. \*/ |
|  |  |
|  | var Employee = function (name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | Employee.prototype.getName = function () |
|  | { |
|  | return this.name; |
|  | } |
|  |  |
|  | // Adding getNameLength() method to the parent object |
|  | // which becomes available in the derived object |
|  | Employee.prototype.getNameLength = function () |
|  | { |
|  | return this.name.length + " characters"; |
|  | } |
|  |  |
|  | // PermanentEmployee will be the derived object |
|  | var PermanentEmployee = function (annualSalary) |
|  | { |
|  | this.annualSalary = annualSalary; |
|  | } |
|  |  |
|  | PermanentEmployee.prototype = new Employee("Mark"); |
|  |  |
|  | var pe = new PermanentEmployee(50000); |
|  | // Call getNameLength() method added to the parent object |
|  | console.log(pe.getNameLength()); // Output : 4 characters |
|  |  |
|  | /\* |
|  | Use hasOwnProperty() method to determine if a property is defined on the actual object or it's prototype. Here is an example. |
|  | \*/ |
|  | var Employee = function (name) |
|  | { |
|  | this.name = name; |
|  | } |
|  |  |
|  | var PermanentEmployee = function (annualSalary) |
|  | { |
|  | this.annualSalary = annualSalary; |
|  | } |
|  |  |
|  | var employee = new Employee("Mark"); |
|  |  |
|  | PermanentEmployee.prototype = employee; |
|  |  |
|  | var pe = new PermanentEmployee(50000); |
|  |  |
|  | console.log("Employee.name= " + employee.hasOwnProperty('name')); |
|  | console.log("Employee.annualSalary= " + employee.hasOwnProperty('annualSalary')); |
|  |  |
|  | console.log("PermanentEmployee.name=" + pe.hasOwnProperty('name')); |
|  | console.log("PermanentEmployee.annualSalary= " + pe.hasOwnProperty('annualSalary')); |
|  |  |
|  | Output : |
|  | Employee.name = true |
|  | Employee.annualSalary = false |
|  |  |
|  | PermanentEmployee.name= false |
|  | PermanentEmployee.annualSalary= true |
|  |  |
|  | /\* ------------------------ **POLYMORPHISM**  ------------------------ \*/ |
|  | ` |
|  | **1.what is it?** |
|  |  |
|  | Poly= many, morphism=form or behavior shifting. |
|  |  |
|  | **2.why we need it ?** |
|  |  |
|  | In programming, It is used when we want a function's (let say function X's) interface to be flexible enough to accept different types or number of parameters. |
|  | Also, based on changing parameters types or numbers, we might want the function X to behave differently (morphism). |
|  |  |
|  | **3.How it works?** |
|  |  |
|  | We write multiple implementations of X function where each implementation accepts different parameters types or number of parameters. |
|  | Based on the type or number of parameter, the compiler (at runtime) decides which implementation of X should be executed when X is called from some code. |
|  |  |
|  | **4.how can I achieve this polymorphic behavior in javascript?** |
|  |  |
|  | JS is not a typed language so it really not meant to use OOP concepts like polymorphism. |
|  | However, the newer version of JS now include classes and there is possibility that polymosphism could start making sense in JS, too. |
|  | Other answers provide some interesting workarounds. |