**Створення об’єктів на основі прототипів**

Особливості:

* прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об’єктів немає власної властивості, то цю властивість шукаємо у об’єкті прототипі (об’єкті, адреса якого міститься у властивості \_\_proto\_\_) );
* операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об’єктом;
* при зміні значення властивості (яку читали з прототипу) у об’єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

|  |  |  |
| --- | --- | --- |
| З використанням властивості \_\_proto\_\_  (**НЕ РЕКОМЕНДУЄТЬСЯ**!!!) | З використанням Object.create   * Object.create ( прототип ) – створює новий об’єкт з вказаним прототипом (посилання міститься у \_\_proto\_\_); * Object.create(прототип , descriptors) – створює новий об’єкт з вказаним прототипом і додає нові властивості, описані за допомогою дискриптора | Явне присвоєння властивості prototype функції-конструктора об’єкта-прототипу |
| ========== Варіант 1 ===================  //--- Створюємо об’єкт  var obj = {  prop\_1: 11,  meth\_1: function () {  document.write("meth\_1");  }  };  //-------- Додаємо предка у \_\_proto\_\_  obj.\_\_proto\_\_ =  {  prop\_2: 22,  meth\_2: function () {  document.write("meth\_2");  }  };  ========== Варіант 2 ===================  //--- Створюємо об’єкт  var obj = {  prop\_1: 11,  meth\_1: function () {  document.write("meth\_1");  }  };  //-------- Об"єкт предок (прототип)  var obj\_parent = {  prop\_2: 22,  meth\_2: function () {  document.write("meth\_2");  }  };  //---- Додаємо об"єкт предок (прототип)  **Object.setPrototypeOf**(obj, obj\_parent)  Або напряму записуємо у властивість \_\_proto\_\_    obj.**\_ \_proto\_ \_** = obj\_parent;  obj.prop\_2=90 | ================Варіант 1 =====================  //-------- Створюємо предка  var obj\_parent = {  prop\_2: 22,  meth\_2: function () {  document.write("meth\_2");  }  };  //-------- Створюємо об"єкт на основі прототипу  var obj = Object.create(obj\_parent);  //-------- Додаємо нові властивості  obj.prop\_1 = 11;  obj.meth\_1 = function () {  document.write("meth\_1");    };  ================Варіант 2 =====================  //-------- Створюємо предка  obj\_parent = {  prop\_2: 22,  meth\_2: function () {  document.write("meth\_2");  }  };  //-------- Створюємо об"єкт на основі прототипу,  //-- за допомогою дискрипторів додаємо властивості  var obj = Object.create(obj\_parent, {  prop\_1: {  value: 11,  enumerable:true  },  meth\_1: {  value: function () {  document.write("meth\_1");  },  enumerable:true  }  }); | //-------- Створюємо предка  obj\_parent = {  prop\_2: 22,  meth\_2: function () {  document.write("meth\_2");  }  };  //------ Опис функції конструктора  function Obj1() {  this.prop\_1 = 11;  this.meth\_1= function () {  document.write("meth\_1");  }  }  //------ Додавання прототипу  //------ (буде присвоєно у \_\_proto\_\_об’єктів )  Obj1.prototype = obj\_parent;  //----- Cтворюємо об"єкт  var obj = new Obj1(); |

Метод для аналізу властивості \_\_proto\_\_

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Object.getPrototypeOf(obj) | Повертає obj.\_\_proto\_\_ (крім IE8-) |  |
| Object.setPrototypeOf(obj, proto) | Встановлює obj.\_\_proto\_\_ = proto (крім IE10-). |  |

Визначення того, чи є прототипом деякого об’єкта

* obj1.isPrototypeOf(obj2) дозволяє визначити, чи є obj1 у ланцюзі прототипів obj2;
* Оператор **instanceof перевіряє, чи об’єкт належить до даного класу**. Іншими словами, object instanceof constructor перевіряє, чи присутній об’єкт constructor.prototype у ланцюгу прототипів object.

**Алгоритм проверки obj instanceof Constructor:**

1. Получить obj.\_\_proto\_\_
2. Сравнить obj.\_\_proto\_\_ с Constructor.prototype
3. Если не совпадает, тогда заменить obj на obj.\_\_proto\_\_ и повторить проверку на шаге 2 до тех пор, пока либо не найдется совпадение (результат true), либо цепочка прототипов не закончится (результат false).

|  |  |
| --- | --- |
|  |  |
| предок . isPrototypeOf( нащадок ) | var obj\_parent = {  . . . . . .  };  var obj\_child={  . . . . . .  }  obj\_child.\_\_proto\_\_ = obj\_parent;  document.write(obj\_parent.isPrototypeOf(obj\_child)); //true |
| предок . isPrototypeOf( нащадок ) | //---- КЛАС ПРЕДОК -----  function Class\_parent() {  . . . . .  };  //////---- КЛАС НАЩАДОК -----  function Class\_child() {  . . . . .  }  //--- Створюємо прототип нащадка на основі прототипу предка  Class\_child.prototype = Object.create(Class\_parent.prototype);  . . . . .  var obj = new Class\_child();  document.write( Class\_parent.prototype.isPrototypeOf(obj)); //true |
|  |  |

Методи для аналізу властивостей об’єкта

|  |  |  |
| --- | --- | --- |
|  |  |  |
| obj.hasOwnProperty(prop) | **повертає true, якщо властивість prop належить безпосередньо самому об’єкту obj (а якомусь його нащадку), інакше false.** |  |
| Object.defineProperty(obj, prop, descriptor) | Аргументы:  **obj**  Об’єкт, для якого створюється властивість.  **prop**  ім’я властивості, яка оголошується або модифікується.  **descriptor**  об’єкт-дискриптор, який містить властивості об’єкта  В нём могут быть следующие поля:   * value – значение свойства, по умолчанию undefined * writable – значение свойства можно менять, если true. По умолчанию false. * configurable – если true, то свойство можно удалять, а также менять его в дальнейшем при помощи новых вызовов defineProperty. По умолчанию false. * enumerable – если true, то свойство просматривается в цикле for..in и методе Object.keys(). По умолчанию false. * get – функция, которая возвращает значение свойства. По умолчанию undefined. * set – функция, которая записывает значение свойства. По умолчанию undefined. | "use strict";  var user = {};  Object.defineProperty(user, "name", {  value: "Вася",  writable: false, // запретить присвоение "user.name="  configurable: false // запретить удаление "delete user.name"  });  // Теперь попытаемся изменить это свойство.  // в strict mode присвоение "user.name=" вызовет ошибку  user.name = "Петя"; |
| Object.defineProperties(obj, descriptors) | Дозволяє додати декілька властивостей за один раз | var user = {}  Object.defineProperties(user, {  firstName: {  value: "Петя"  },  surname: {  value: "Иванов"  },  fullName: {  get: function() {  return this.firstName + ' ' + this.surname;  }  }  });  alert( user.fullName ); // Петя Иванов |
| Object.keys(obj) | Object.keys возвращает массив только enumerable-свойств | var obj = {  a: 1,  b: 2,  internal: 3  };  Object.defineProperty(obj, "internal", {  enumerable: false  });  alert( Object.keys(obj) ); // a,b |
| Object.getOwnPropertyNames(obj) | Возвращает массив всех свойств | var obj = {  a: 1,  b: 2,  internal: 3  };  Object.defineProperty(obj, "internal", {  enumerable: false  });  alert( Object.getOwnPropertyNames(obj) ); // a, internal, b |
| Object.getOwnPropertyDescriptor(obj, prop) | Возвращает дескриптор для свойства obj[prop]. Полученный дескриптор можно изменить и использовать defineProperty для сохранения изменений. | var obj = {  test: 5  };  var descriptor = Object.getOwnPropertyDescriptor(obj, 'test');  // заменим value на геттер, для этого...  delete descriptor.value; // ..нужно убрать value/writable  delete descriptor.writable;  descriptor.get = function() { // и поставить get  alert( "Preved :)" );  };  // поставим новое свойство вместо старого  // если не удалить - defineProperty объединит старый дескриптор с новым  delete obj.test;  Object.defineProperty(obj, 'test', descriptor);  obj.test; // Preved :) |

**УСПАДКУВАННЯ У ФУНКЦІОНАЛЬНОМУ СТИЛІ**

|  |  |
| --- | --- |
|  |  |
| **function ім’я функції-конструктора нащадка {**  **//-------- Додавання властивостей з предка -----**  **Конструктор предка . apply(this, параметри для конструктора предка )**  **//-------- Опис власних властивостей -----**  **this.властивість1 = значення1;**  **. . .**  **}**  **//---- Створення прототипу на основі прототипу пердка -----**  **Конструктор нащадка = Object.create( Конструктор предка . prototype );**  **//--- Встановлення властивості constructor**  **Конструктор нащадка .prototype.constructor = ім’я функції конструктора ;**  **//--- Додавання власних методів**  **функція-конструктор нащадка.prototype. метод 1 = function(форм.пар.)**  **{**  **. . .**  **}** | //---- КЛАС ПРЕДОК -----  function Class\_parent() {  //-- Властивості у конструкторі  this.prop\_2 = 22;  };  //--- Методи у прототипі  Class\_parent.prototype.meth\_2= function () {  document.write("meth\_2");  }  //---- КЛАС НАЩАДОК -----  function Class\_child() {  //--- Додаємо властивосі предка у об"єкт нащадка (якщо треба)  //this={}  Class\_parent.call(this); //або ж Obj\_parent.apply(this);  //this={prop\_2 : 22}  //--- Додаємо власні властивості дочірнього класу ---  this.prop\_1 = 11;  //this={prop\_2 : 22, prop\_1 : 11}  }  //--- Створюємо прототип на основі прототипу предка (якщо треба)  Class\_child.prototype = Object.create(Class\_parent.prototype);  //--- Встановлюємо конструктор  Class\_child.prototype.constructor = Class\_child;  //--- Додаємо власні методи дочірнього класу (якщо треба) ----  Class\_child.prototype.meth\_1 = function () {  document.write("meth\_1");  }  var obj = new Class\_child();  document.write(obj instanceof Class\_parent); //true  document.write(obj instanceof Class\_child); //true  document.write(Class\_child.prototype.isPrototypeOf(obj)); //true |
|  | Приклад 2.  // Shape — суперкласс  function Shape() {  this.x = 0;  this.y = 0;  }  // метод суперкласса  Shape.prototype.move = function(x, y) {  this.x += x;  this.y += y;  console.info('Фигура переместилась.');  };  // Rectangle — подкласс  function Rectangle() {  Shape.call(this); // вызываем конструктор суперкласса  }  // подкласс расширяет суперкласс  Rectangle.prototype = Object.create(Shape.prototype);  Rectangle.prototype.constructor = Rectangle; |

Використання сучасного стилю

|  |  |
| --- | --- |
|  |  |
| //---- КЛАС НАЩАДОК -----  class клас\_нащадок extends клас\_предок {  constructor() {  //---- виклик конструктора предка -----  super();  //---- додавання власних властивостей -----  this. Властивіть1 = значення1 ;  . . . . .  }  //---- додавання власних методів -----  meth\_1 () {  . . . . .  }  . . . . .  }  -----------------------------------------------------------  //---- КЛАС НАЩАДОК -----  class клас\_нащадок extends клас\_предок {  constructor(параметри) {  //---- виклик конструктора предка -----  super(...arguments); //Передаємо параметри у конструктор  //---- додавання власних властивостей -----  this. Властивіть1 = значення1 ;  . . . . .  }  //---- додавання власних методів -----  meth\_1 () {  . . . . .  }  . . . . .  } | //---- КЛАС ПРЕДОК -----  class Class\_parent {  constructor(){  //-- Властивості у конструкторі  this.prop\_2 = 22;  }  //-- Методи ---  meth\_2 () {  document.write("meth\_2");  }  };  //---- КЛАС НАЩАДОК -----  class Class\_child extends Class\_parent {  constructor() {  //---- виклик конструктора предка -----  super();  //---- додавання власних властивостей -----  this.prop\_1 = 11;  }  //---- додавання власних методів -----  meth\_1 () {  document.write("meth\_1");  }  }  var obj = new Class\_child(); |

Конструктор constructor родителя наследуется автоматически. То есть, если в потомке не указан свой constructor, то используется родительский. В примере выше Rabbit, таким образом, использует constructor от Animal.

Если же у потомка свой constructor, то, чтобы в нём вызвать конструктор родителя – используется синтаксис super() с аргументами для родителя.

Виклик перевизначених методів з класу предка

|  |  |
| --- | --- |
| Загальна форма. Виклик здійснюється через використання ключового слова  super |  |
| //---- КЛАС НАЩАДОК -----  class Class\_child extends Class\_parent {  . . . .  Метод нащадка () {  super. Метод предка (форм.парам.); // виклик метода предка  document.write("-- child --");  }  . . . . . .  } | //---- КЛАС ПРЕДОК -----  class Class\_parent {  . . . . . .  method () {  document.write(" -- parent -- ");  }  . . . . . .  };  //---- КЛАС НАЩАДОК -----  class Class\_child extends Class\_parent {  . . . .  method() { // перевизначення (опис метода з таким же іменем)  super.method(); // виклик метода предка  document.write(" -- child --");  }  . . . . . .  }  var obj = new Class\_child();  obj.method(); //-- parent -- -- child -- |

Деякі методи Object для маніпуляцій з об’єктами

|  |  |  |
| --- | --- | --- |
| Метод | Призначення | Приклад |
| Object.assign(target, ...sources ); | Копирует значения из одного или нескольких исходных объектов в целевой объект |  |
| Object.preventExtensions(obj) | Запрещает добавление свойств в объект |  |
| Object.seal(obj) | Запрещает добавление и удаление свойств, все текущие свойства делает configurable: false. |  |
| [**Object.freeze(obj)**](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/freeze) | Запрещает добавление, удаление и изменение свойств, все текущие свойства делает configurable: false, writable: false. |  |
| Object.isExtensible(obj) | Возвращает false, если добавление свойств объекта было запрещено вызовом метода Object.preventExtensions |  |
| Object.isSealed(obj) | Возвращает true, если добавление и удаление свойств объекта запрещено, и все текущие свойства являются configurable: false. |  |
| Object.isFrozen(obj) | Возвращает true, если добавление, удаление и изменение свойств объекта запрещено, и все текущие свойства являются configurable: false, writable: false |  |

Следующие связанные функции предотвращают изменение атрибутов объекта.

|  |  |  |  |
| --- | --- | --- | --- |
| **Функция** | **Объект становится нерасширяемым.** | **configurable получает значение false для каждого свойства** | **writable получает значение false для каждого свойства** |
| [Object.preventExtensions](https://msdn.microsoft.com/ru-ru/library/ff806191(v=vs.94).aspx) | Да | Нет | Нет |
| [Object.seal](https://msdn.microsoft.com/ru-ru/library/ff806192(v=vs.94).aspx) | Да | Да | Нет |
| **Object.freeze** | Да | Да | Да |

Следующие функции возвращают **true**, если все условия, отмеченные в следующей таблице, имеют значение true.

|  |  |  |  |
| --- | --- | --- | --- |
| **Функция** | **Расширяем ли объект?** | **configurable имеет значение false для всех свойств?** | **writable имеет значение false для всех свойств данных?** |
| [Object.isExtensible](https://msdn.microsoft.com/ru-ru/library/ff806188(v=vs.94).aspx) | Да | Нет | Нет |
| [Object.isSealed](https://msdn.microsoft.com/ru-ru/library/ff806189(v=vs.94).aspx) | Нет | Да | Да |
| [Object.isFrozen](https://msdn.microsoft.com/ru-ru/library/ff806185(v=vs.94).aspx) | Нет | Да | Да |

==================================================== ПРАКТИКА ============================

Приклад. Створити клас Client

|  |  |
| --- | --- |
| Властивості | ID  ПІБ  Кількість грошей на рахунку |
| Методи | Додавання грошей  Зняття грошей  ToString |

На основі цього класу створити клас GoldenClient

|  |  |
| --- | --- |
| Властивості | ID  ПІБ  Кількість грошей на рахунку  Ліміт кредитних коштів  Відсоток за використання кредитних коштів |
| Методи | Додавання грошей  Зняття грошей  Визначення пені за використання кредитних коштів  ToString |

Створити клас Bank, у якому зберігається масив клієнтів. Виконати такі операції

* Вивести усіх простих клієнтів;
* Вивести усіх клієнтів GoldenClient;
* Знати сумарну кількість грошей на рахунку;
* Приклад. Створити клас ПРАЦІВНИК.

Приклад 2. Створити клас Employee

|  |  |
| --- | --- |
| Властивості | ID  ПІБ  Заробітна плата |
| Методи | Визначення ЗП за рік  ToString |

На основі цього класу створити клас МЕНЕДЖЕР

|  |  |
| --- | --- |
| Властивості | ID  ПІБ  Заробітна плата  Номер відділу, у якому керує |
| Методи | Визначення ЗП за рік  ToString |

class Employee {

constructor(id="-",name="unknown",salary=3200)

{

Object.defineProperty(this, "ID", { value: id });

this.Name = name;

Object.defineProperty(this, "Salary", {

get: function () {

return salary;

},

set: function (value) {

if (value >= 0) {

salary = value;

}

else

throw new Error("errr");

}

})

}

getSeleryPerYear() {

return this.Salary \* 12;

}

toString() {

return `${this.Name} - ${this.getSeleryPerYear()}`;

}

}

//---------------------------

class Manager extends Employee {

constructor(id = "-", name = "unknown", salary = 3200, department = "-") {

super(id, name, salary); // Виклик конструктора предка

// super(...arguments); //аналогічний виклик

Object.defineProperty(this, "Department", { value: department });

}

toString() {

return super.toString() + "-" + this.Department;

}

}

//----------------

//===================================

var manager = new Manager("1221", "Ivan", 50000, "HR");

document.write(manager);

Задача 0. Створити клас Range (діапазон)

|  |  |
| --- | --- |
| Властивості | minValue  maxValue |
| Методи | inRange(value) – метод визначення того, чи є вказане значення у заданому діапазоні  ToString |

На основі класу Range створити клас PensionerChecker

|  |  |
| --- | --- |
| Властивості | minValue – мінімальний вік пенсіонера  maxValue – максимальний вік |
| Методи | isPensioner(age) – метод визначення того, чи є пенсіонером  render(containerID) – метод виведення розмітки    ToString |

Задача 1. Створити клас Dice, який представляє гральний кубик (одне поле faceCount - кліькістьграней) і дозволяє генерувати випадкове число від 1 до кількості граней. Потім на основі цього класу створити клас , який дозволяє вказати не тільки кількість граней, а й кількість спроб - attemptCount (при цьому як випадкове число повертається середнє значення).

Задача 2. Створити клас Rectangle, який представляє прямокутник.

|  |  |
| --- | --- |
| Властивості | A  B |
| Методи | Знаходження площі  Знаходження периметру  ToString |

На основі цього класу створити клас Paralelepiped

|  |  |
| --- | --- |
| Властивості | A  B  H |
| Методи | Знаходження площі поверхні  Знаходження об’єму  ToString |

Задача 3. Створити клас TPair, який представляє пару чисел і містить методи для їх збільшення/зменшення на 1. Реалізувати класи нащадки TTime (“години.хвилини”) та TMoney (“гривні.копійки”). Згенерувати поступово випадковим чином  пар (час, гроші), де час – тривалість виконання роботи, а гроші – вартість однієї хвилини роботи працівників. Обчислити витрати на виконання кожної із робіт.