Front-end

JavaScript

JS. Object

- Объекты

JS. Object

Объекты, как мы знаем, содержат свойства. У каждого из свойств объекта, кроме значения, есть ещё три флага конфигурации, которые могут принимать значения true или false. Эти флаги называются дескрипторами:

- **writable** — доступно ли свойство для записи (значение свойства можно менять, если true);
- **enumerable** — является ли свойство видимым при перечислениях (например, в цикле for..in, если true, свойство просматривается в цикле for..in и методе Object.keys());
- **configurable** — доступно ли свойство для переконфигурирования (контролирует, может ли свойство быть удалено из объекта и могут ли быть изменены его атрибуты (кроме контроля изменения атрибута writable), препятствует возможности использовать операторdeletedля удаления существующего свойства)

Когда мы создаём свойство объекта «обычным способом», эти три флага устанавливаются в значение true.

Для изменения значений дескрипторов применяется статический метод Object.defineProperty(), а для чтения значений — Object.getOwnPropertyDescriptors().

Другими словами, дескрипторы — это пары ключ-значение, которые описывают поведение свойства объекта при выполнении операций над ним (например, чтения или записи).

JS. Object

Создадим объект и добавим в него свойство ОС для ноутбука. Сделаем это с помощью дескрипторов и статического метода Object.defineProperty().

Передаём в метод:

- объект, которому добавляем свойство;
- название свойства строкой;
- объект со значениями дескрипторов и ключом value, содержащим значение свойства.

JS. Object

```
const laptop = {}


Object.defineProperty(laptop, 'os', {
  value: 'MacOS',
  writable: false,
  enumerable: true,
  configurable: true
})
```

Свойство os будет недоступно для перезаписи, но будет видно при перечислении и доступно для переконфигурирования.

JS. Object

Как пишется

Object.defineProperty(объект, имяСвойства, дескрипторы)

Функция принимает следующие параметры:

- объект — объект, свойство которого изменяем или добавляем;
- имяСвойства — свойство, для которого нужно применить дескриптор;
- дескриптор — дескриптор, описывающий поведение свойства.

Если свойство уже существует, Object.defineProperty() обновит флаги.

Если свойство не существует, метод создаёт новое свойство с указанным значением и флагами. Если какой-либо флаг не указан явно, ему присваивается значение false.

JS. Object

Для определения сразу нескольких свойств можно применять метод Object.defineProperties:

```
Object.defineProperties(flower,
  name: { writable: false, configurable: false },
  color: { enumerable: false }
);
```

Так как свойству color мы установили флаг enumerable: false, то теперь оно не будет доступно для перебора:

```
for (let key in flower) {
  console.log(key);
}
```

JS. Object

Получить полное описание свойства можно с помощью метода Object.getOwnPropertyDescriptor:

```
const flower = {
  name: 'rose',
  color: 'red'
}
const descriptor = Object.getOwnPropertyDescriptor(flower, 'name');
console.log(descriptor);
```

JS. Object

Этот метод позволяет получить полное описание одного свойства. Для этого в Object.getOwnPropertyDescriptor мы должны в качестве первого аргумента передать объект, содержащий это свойство, а посредством второго – его само.

Получить описание сразу всех свойств объекта можно с помощью статического метода **Object.getOwnPropertyDescriptors**

JS. Object

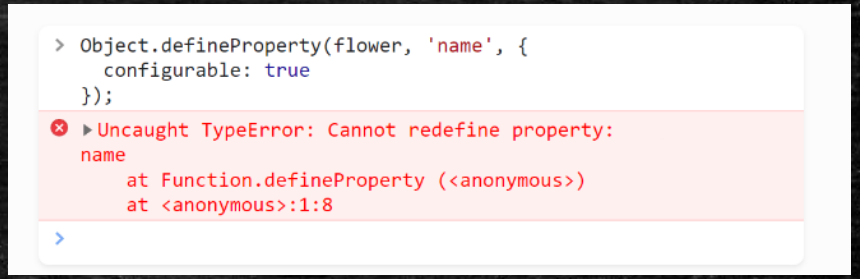
Если свойству установить флаг configurable: false, то оно становится не конфигурируемым. Такое свойство нельзя будет удалить и его флагам нельзя будет установить новые значения. Таким свойствам мы сделали name.

```
Object.defineProperties(flower,

 name: { writable: false, configurable: false },

 color: { enumerable: false }

);
```



При попытке установить этому свойству новые значения флагам мы получим ошибку

JS. Object

Методы Object.defineProperty и Object.defineProperties можно использовать для добавления новых свойств объекту:

```
const city = {};
Object.defineProperties(city, {
  name: {value: 'New York', enumerable: true, writable: true},
  area: {value: 1223.59, enumerable: true, writable: true},
});
console.log(Object.getOwnPropertyDescriptors(city));
```

JS. Object

Когда мы добавляем новые свойства с помощью этих методов, все флаги имеют по умолчанию значение false. Так как в этом примере мы пропустили флаг configurable для всех свойств, то он у всех них имеет значение **false**, то все эти свойства являются не конфигурируемыми.

JS. Object

Геттеры, сеттеры

Дескриптор доступа — это дескриптор, который определяет работу свойства через функции чтения и записи свойства (геттера и сеттера).

get (геттер) — функция, используемая для получения значения свойства, возвращает значение или undefined. (это метод, который выполняется, когда мы хотим получить значение динамического свойства. Он не имеет параметров и возвращает значение, которое и будет значением этого свойства.)

set (сеттер) — функция, используемая для установки значения свойства. Принимает единственным аргументом новое значение, присваиваемое свойству. Далее это значение мы можем присвоить некоторому свойству или выполнить какие-либо другие действия.

JS. Object

Геттеры, сеттеры

```
const animal = { _hiddenName : 'Кот' }
Object.defineProperty(animal, 'name', {
   get: function() { return this._hiddenName }
})

const animal2 = {
  name: 'И здесь тоже кот',
}

console.log(animal.name) // => Кот
console.log(animal2.name) // => И здесь тоже кот
```

JS. Object

Геттеры, сеттеры

Оба объекта имеют одинаковое поведение. Стоит только сказать, что за свойством в первом случае стоит функция, которая вызывается автоматически. Достаточно написать animal.name.

Если нам понадобится изменить значение свойства name, мы выполним animal.name = 'Серый кот', ничего не произойдёт. Дело в том, что с ключом name не связана функция-сеттер, поэтому значение этому свойству установить невозможно.

JS. Object

Геттеры, сеттеры

Добавим сеттер:

```
const animal = { _hiddenName : 'Кот' }
Object.defineProperty(animal, 'name', {
   get: function() { return this._hiddenName },
   set: function(value){ this._hiddenName = value }
})

animal.name = 'Собака'
console.log(animal.name)
```

JS. Object

Геттеры, сеттеры

По сути, мы можем регулировать возможность читать и получать значение свойства, как и в дескрипторе данных, только более тонко. Такой подход используется часто, поэтому для объявления геттеров и сеттеров придумали синтаксис без вызова Object.defineProperty()

```
const animal = {

 get name() {

  return this._name

 },

 set name(value) {

  this._name = value

 }

}

console.log(animal.name) // undefined

animal.name = 'Кот'

console.log(animal.name) // Кот
```

JS. Object

Периодически разработчику нужно защищать объекты от вмешательства извне. По ошибке легко изменить свойство объекта.

Для защиты объектов от подобных изменений и управления их иммутабельностью предлагается использовать дескрипторы, такие как writable и configurable, сеттеры, а также методы Object.preventExtensions(), Object.seal(), и Object.freeze() для ограничения доступа к объекту целиком.

JS. Object

Не влияют на вложенный объект. Действуют только на свой уровень

Object.preventExtensions(); - запретить расширение, нельзя добавлять новые поля

Object.isExtensible(); - проверяет на расширение

Object.freeze(); - нельзя ничего делать

Object.isFrozen();

Object.seal(); - нельзя удалять и добавлять, можно только менять

Object.isSealed();

JS. Object

Метод Object.preventExtensions() позволяет сделать объект нерасширяемым (предотвращает добавление новых собственных (неунаследованных) свойств). При этом допускается как удаление, так и изменение свойств объекта, на котором был вызван метод.

Действие этого метода необратимо, так как нерасширяемый объект нельзя вновь сделать расширяемым объектом.

Чтобы проверить является ли объект расширяемым вы можете воспользоваться методом Object.isExtensible()

JS. Object

Метод freeze() позволяет сделать объект нерасширяемым (предотвращает добавление новых собственных (неунаследованных) свойств), устанавливает все его собственные свойства как ненастраиваемые (предотвращает их удаление и изменение дескриптора), а также устанавливает все его собственные свойства недоступными для записи (изменение свойства объекта с помощью оператора присваивания не допускается).

JS. Object

Заморозка.

Метод Object.freeze(..)создает замороженный объект, что означает, что он принимает существующий объект и по сути применяет к нему Object.seal(..), но также помечает все свойства «доступа к данным» как writable:false, так, что их значения не могут быть изменены.

Этот подход дает наивысший уровень иммутабельности, который вы можете получить для самого объекта, поскольку он предотвращает любые изменения в объекте или его непосредственных свойствах.

Вы можете «глубоко заморозить» объект, применив Object.freeze(..)к объекту и рекурсивно перебрать все объекты, на которые он ссылается (которые еще не были затронуты) применив к ним Object.freeze(..). Однако, будьте осторожны, поскольку это может затронуть другие (общие) объекты, которые вы не планировали менять.

JS. Object

Заморозка.

Object.isFrozen()

Объект является неизменяемым?

Синтаксис

Object.isFrozen(obj)

Возвращаемое значение: true, если объект о является зафиксированным и неизменяемым, и false – если нет.

JS. Object

Метод Object.seal() запечатывает объект . Запечатывание объекта предотвращает расширение и делает существующие свойства недоступными для настройки. Запечатанный объект имеет фиксированный набор свойств: нельзя добавить новые свойства, нельзя удалить существующие свойства, нельзя изменить их перечисляемость и конфигурируемость, нельзя переназначить его прототип. Значения существующих свойств можно изменить, если они доступны для записи. seal() возвращает тот же объект, который был передан.

Синтаксис

Object.seal(obj)

JS. Object

Object.isSealed()

Метод Object.isSealed() определяет, запечатан ли объект.

Синтаксис

Object.isSealed(obj)

Параметры

obj - объект,который необходимо проверить

JS. Object

Сравнение подобных методов

Метод	Объект становится нерасширяемым	Атрибут configurable для каждого собственного (неунаследованного) свойства имеет значение false	Атрибут writable для каждого собственного (неунаследованного) свойства имеет значение false
preventExtensions()	Да	Нет	Нет
seal()	Да	Да	Нет
freeze()	Да	Да	Да

JS. Object

Конструктор, оператор "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество похожих, однотипных объектов, таких как пользователи, элементы меню и так далее.

Это можно сделать при помощи функции-конструктора и оператора "new".

Функции-конструкторы технически являются обычными функциями. Но есть два соглашения:

- Имя функции-конструктора должно начинаться с большой буквы.
- Функция-конструктор должна выполняться только с помощью оператора "new".

JS. Object

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
alert(user.name); // Jack
alert(user.isAdmin); // false
```

JS. Object

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
alert(user.name); // Jack
alert(user.isAdmin); // false
```

Когда функция вызывается как new User(...), происходит следующее:

- Создаётся новый пустой объект, и он присваивается this.
- Выполняется тело функции. Обычно оно модифицирует this, добавляя туда новые свойства.
- Возвращается значение this.

JS. Object

new User(...) делает что-то вроде:

```
function User(name) {
  // this = {};  (неявно)
  // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;
  // return this;  (неявно)
}
```

Таким образом, let user = new User("Jack") возвращает тот же результат, что и:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

JS. Object

Теперь, если нам будет необходимо создать других пользователей, мы можем просто вызвать new User("Ann"), new User("Alice") и так далее. Данная конструкция гораздо удобнее и читабельнее, чем многократное создание литерала объекта.

Это и является основной целью конструкторов – реализовать код для многократного создания однотипных объектов.

JS. Object

Создание методов в конструкторе

Использование конструкторов для создания объектов даёт большую гибкость. Функции-конструкторы могут иметь параметры, определяющие, как создавать объект и что в него записывать.

Конечно, мы можем добавить к this не только свойства, но и методы.

Например, new User(name) ниже создаёт объект с заданным name и методом sayHi:

```
function User(name) {
 this.name = name;

 this.sayHi = function() {
  alert( "Меня зовут: " + this.name );
 };
}
let john = new User("John");
john.sayHi(); // => Меня зовут: John
```

JS. Object

Функции-конструкторы или просто конструкторы, являются обычными функциями, но существует общепринятое соглашение именовать их с заглавной буквы.

Функции-конструкторы следует вызывать только с помощью new. Такой вызов подразумевает создание пустого this в начале и возврат заполненного в конце.

Мы можем использовать конструкторы для создания множества похожих объектов.