JavaScript - классы

Класс: базовый синтаксис;
Наследование классов;
Статические свойства и методы;
Приватные и защищенные методы и свойства;
Расширение встроенных классов;
Проверка класса: "instanceof";
Примеси;

Класс - что это

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы). [википедия]

Класс в программировании — это шаблон для создания объектов, который позволяет инкапсулировать данные и методы, относящиеся к этим данным, в одну сущность. В объектно-ориентированном программировании (ООП) классы используются для определения типов объектов и их поведения. Они содержат поля (свойства) и методы (функции), которые управляют этими свойствами.

В JavaScript классы были введены в ECMAScript 2015 (ES6) и являются синтаксическим сахаром для прототипного наследования, которое было основой объектной модели JavaScript с самого начала. То есть классы в JavaScript предоставляют более удобный и понятный способ работы с объектами и наследованием.

Класс: базовый синтаксис

- Синтаксис «class»
- Что такое класс в javascript-е
- Не просто синтаксический сахар
- Class Expression
- Геттеры/сеттеры, другие сокращения

Синтаксис «class»

В коде вызов new MyClass() нужен для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

```
class MyClass {
  // методы класса
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }

}

// Использование:
let user = new User("Иван");
user.sayHi();
```

constructor в JavaScript классе

constructor в JavaScript классе — это специальный метод, который используется для инициализации объектов, созданных с использованием этого класса. Он вызывается автоматически при создании нового экземпляра класса и позволяет задавать начальные значения свойств объекта.

Вот основные моменты, которые стоит знать о конструкторе в JavaScript классе:

- Определение конструктора: Конструктор определяется внутри класса с использованием ключевого слова constructor.
- Инициализация свойств: Конструктор часто используется для инициализации свойств объекта с начальными значениями, переданными в качестве аргументов при создании объекта.
- Автоматический вызов: Конструктор вызывается автоматически при создании нового объекта с использованием ключевого слова new.
- Использование super(): В подклассах (классах, которые наследуются от других классов) конструктор родительского класса может быть вызван с помощью функции super(). Это необходимо для правильной инициализации свойств, унаследованных от родительского класса.

Что такое класс в javascript-е

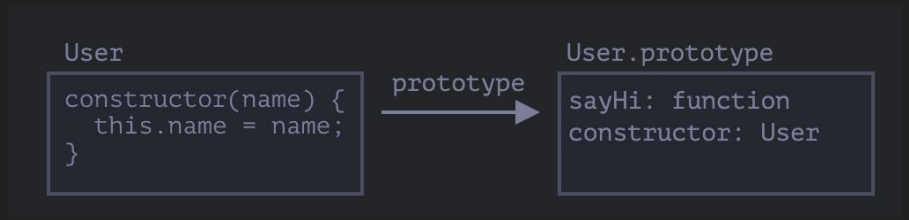
В JavaScript класс – это разновидность функции.

Вот что на самом деле делает конструкция class User {...}:

1. Создает функцию с именем User, которая становится результатом объявления класса. Код функции берется из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

При вызове метода объекта new User он будет взят из прототипа. Таким образом, объекты new User имеют доступ к методам класса.

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// доказательство: User – это функция
alert(typeof User); // function
```




```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// класс – это функция
alert(typeof User); // function

// ...или, если точнее, это метод constructor
alert(User === User.prototype.constructor); // true

// Методы находятся в User.prototype, например:
alert(User.prototype.sayHi); // sayHi() { alert(this.name); }

// в прототипе ровно 2 метода
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

Не просто синтаксический сахар

Важные отличия:

- Функция, созданная с помощью class, помечена специальным внутренним свойством [[IsClassConstructor]]: true. Поэтому это не совсем то же самое, что создавать ее вручную.
- В отличие от обычных функций, конструктор класса не может быть вызван без new
- Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype".
- Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

```
// перепишем класс User на чистых функциях

// 1. Создаём функцию constructor
function User(name) {
  this.name = name;
}
// каждый прототип функции имеет свойство constructor по умолчанию,
// поэтому нам нет необходимости его создавать

// 2. Добавляем метод в прототип
User.prototype.sayHi = function() {
  alert(this.name);
};

// Использование:
let user = new User("Иван");
user.sayHi();
```

Class Expression

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Аналогично Named Function Expression, Class Expression может иметь имя.

Если у Class Expression есть имя, то оно видно только внутри класса

Мы даже можем динамически создавать классы «по запросу»

```
let User = class {
  sayHi() {
    alert("Привет");
  }
};
```

```
// "Named Class Expression"
// (в спецификации нет такого термина, но происходящее похоже на Named Function Expression)
let User = class MyClass {
  sayHi() {
    alert(MyClass); // имя MyClass видно только внутри класса
  }
};

new User().sayHi(); // работает, выводит определение MyClass

alert(MyClass); // ошибка, имя MyClass не видно за пределами класса
```

```
function makeClass(phrase) {
  // объявляем класс и возвращаем его
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}

// Создаём новый класс
let User = makeClass("Привет");

new User().sayHi(); // Привет
```

Геттеры/сеттеры, другие сокращения

Как и в литеральных объектах, в классах можно объявлять вычисляемые свойства, геттеры/сеттеры и т.д.

```
class User {

  constructor(name) {
    // вызывает сеттер
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("Имя слишком короткое.");
      return;
    }
    this._name = value;
  }

}

let user = new User("Иван");
alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.
```

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

```
class User {

  ['say' + 'Hi']() {
    alert("Привет");
  }

}

new User().sayHi();
```

Наследование классов

Наследование классов – это способ расширения одного класса другим классом.

Таким образом, мы можем добавить новый функционал к уже существующему.

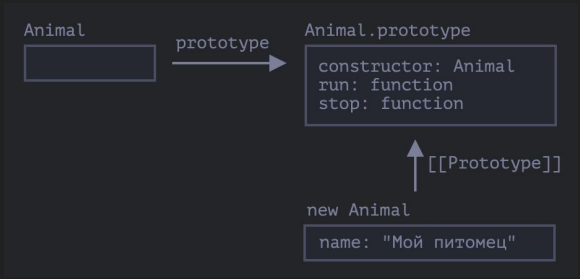
- Ключевое слово «extends»
- Переопределение методов

Ключевое слово «extends»

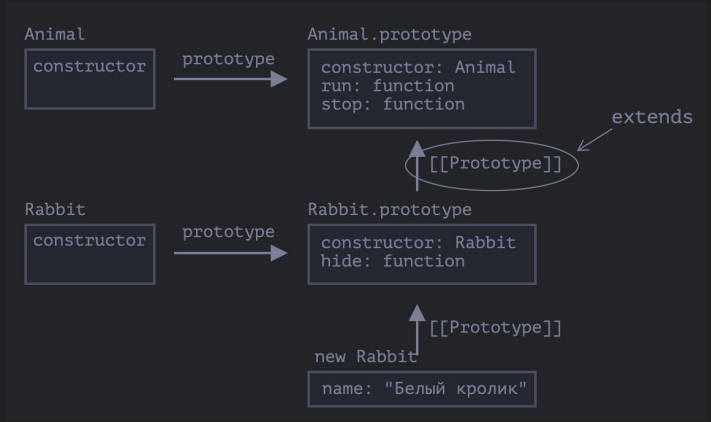
Синтаксис для расширения другого класса следующий:

class Child extends Parent.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит неподвижно.`);
  }
}

let animal = new Animal("Мой питомец");
```




```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.hide(); // Белый кролик прячется!
```



После extends разрешены любые выражения

Синтаксис создания класса допускает указывать после extends не только класс, но и любое выражение.

Это может быть полезно для продвинутых приемов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase); }
  };
}

class User extends f("Привет") {}

new User().sayHi(); // Привет
```

Переопределение методов

По умолчанию все методы, не указанные в классе Rabbit, берутся непосредственно «как есть» из класса Animal.

Но если мы укажем в Rabbit собственный метод, например stop(), то он будет использован вместо него.

Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев.

- super.method(...) вызывает родительский метод.
- super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).

```
class Rabbit extends Animal {
  stop() {
    // ...теперь это будет использоваться для rabbit.stop()
    // вместо stop() из класса Animal
  }
}
```

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
  }

}

class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }

  stop() {
    super.stop(); // вызываем родительский метод stop
    this.hide(); // и затем hide
  }
}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!
```

У стрелочных функций нет super

При обращении к super стрелочной функции он берется из внешней функции:

В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается. Если бы мы указали здесь «обычную» функцию, была бы ошибка.

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000);
    // вызывает родительский stop после 1 секунды
  }
}
```

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Переопределение конструктора

Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор.

Конструкторы в наследуемых классах должны обязательно вызывать super(...), и (!) делать это перед использованием this.

Разница в следующем:

- Когда выполняется обычный конструктор, он создает пустой объект и присваивает его this .
- Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

```
class Rabbit extends Animal {
  // генерируется для классов-потомков, у которых нет своего конструктора
  constructor(...args) {
    super(...args);
  }
}
```

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

// теперь работает
let rabbit = new Rabbit("Белый кролик", 10);
alert(rabbit.name); // Белый кролик
alert(rabbit.earLength); // 10
```

Статические свойства и методы

Мы также можем присвоить метод самому классу. Такие методы называются статическими.

В объявление класса они добавляются с помощью ключевого слова static.

Обычно статические методы используются для реализации функций, которые будут принадлежать классу в целом, но не какому-либо его конкретному объекту.

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true
```

```
class User { }

User.staticMethod = function() {
  alert(this === User);
};
```

Статические методы недоступны для отдельных объектов

Статические методы могут вызываться для классов, но не для отдельных объектов.

```
// ...
article.createTodays(); /// Error: article.createTodays is not a function
```

Статические свойства

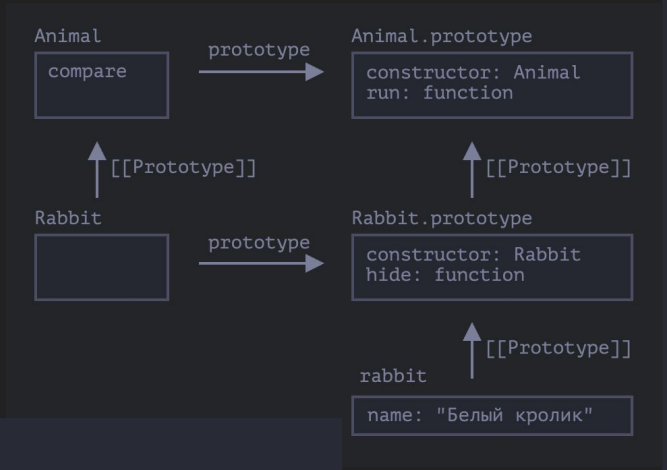
Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

```
Article.publisher = "Илья Кантор";
```

```
class Article {
  static publisher = "Илья Кантор";
}

alert( Article.publisher ); // Илья Кантор
```

Наследование статических свойств и методов

Статические свойства и методы наследуются.




```
class Animal {}
class Rabbit extends Animal {}

// для статики
alert(Rabbit.__proto__ === Animal); // true

// для обычных методов
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
```

```
class Animal {

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }

}

// Наследует от Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbits = [
  new Rabbit("Белый кролик", 10),
  new Rabbit("Чёрный кролик", 5)
];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.
```

Приватные и защищенные методы и свойства

В JavaScript можно создавать приватные и защищенные методы и свойства в классах, чтобы ограничить доступ к ним из внешнего кода и повысить инкапсуляцию данных. Это позволяет скрывать внутреннюю реализацию объекта и предоставлять только необходимые интерфейсы для взаимодействия с ним.

Приватные свойства и методы:

- Используются только внутри класса.
- Защищают внутреннее состояние объекта.
- Обеспечивают строгую инкапсуляцию.

Защищенные свойства и методы:

- Используются внутри класса и его подклассов.
- Поддерживают наследование и расширяемость.
- Реализуются с помощью соглашений (символ _).

Приватные свойства и методы

Приватные свойства и методы доступны только внутри класса и не могут быть использованы за его пределами. Для создания приватных свойств и методов в JavaScript используется символ # перед именем свойства или метода.

Предназначение:

- Приватные свойства и методы предназначены для использования только внутри класса, в котором они определены.
- Они обеспечивают строгую инкапсуляцию, скрывая внутренние детали реализации от внешнего кода.
- Используются для хранения внутреннего состояния объекта и выполнения вспомогательных операций, которые не должны быть доступны или изменяемы извне.

```
class Person {
  // Приватные свойства
  #name;
  #age;

  constructor(name, age) {
    this.#name = name;
    this.#age = age;
  }

  // Приватный метод
  #getAge() {
    return this.#age;
  }

  // Публичный метод, который может использовать приватные свойства и методы
  describe() {
    return `${this.#name} is ${this.#getAge()} years old.`;
  }
}

const person1 = new Person('Alice', 30);
console.log(person1.describe()); // Выведет: Alice is 30 years old.
console.log(person1.#name); // Ошибка: приватное свойство
console.log(person1.#getAge()); // Ошибка: приватный метод
```

Защищенные свойства и методы

JavaScript не имеет встроенной поддержки защищенных (protected) свойств и методов, как в некоторых других языках программирования. Однако, можно использовать соглашения по именованию для обозначения защищенных свойств и методов. Обычно такие свойства и методы начинаются с подчеркивания _.

Предназначение:

- Защищенные свойства и методы предназначены для использования внутри класса и его подклассов.
- Они позволяют подклассам использовать и изменять состояние родительского класса, обеспечивая гибкость и расширяемость кода.
- Защищенные свойства и методы реализуются с помощью соглашений (обычно с использованием символа _), так как JavaScript не имеет встроенной поддержки защищенных членов.

```
class Person {
  // Защищённые свойства (по соглашению)
  _name;
  _age;

  constructor(name, age) {
    this._name = name;
    this._age = age;
  }

  // Защищённый метод (по соглашению)
  _getAge() {
    return this._age;
  }

  describe() {
    return `${this._name} is ${this._getAge()} years old.`;
  }
}

class Employee extends Person {
  constructor(name, age, jobTitle) {
    super(name, age);
    this.jobTitle = jobTitle;
  }

  describe() {
    return `${this._name} is ${this._getAge()} years old and works as a ${this.jobTitle}.`;
  }
}

const employee1 = new Employee('Bob', 25, 'Developer');
console.log(employee1.describe()); // Выведет: Bob is 25 years old and works as a Developer.
console.log(employee1._name); // Доступно, но по соглашению следует избегать
console.log(employee1._getAge()); // Доступно, но по соглашению следует избегать
```

Расширение встроенных классов

От встроенных классов, таких как Array, Map и других, тоже можно наследовать.

встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

Поэтому при вызове метода arr.filter() он внутри создаёт массив результатов, именно используя arr.constructor, а не обычный массив. Это замечательно, поскольку можно продолжать использовать методы PowerArray далее на результатах.

При помощи специального статического геттера Symbol.species можно вернуть конструктор, который JavaScript будет использовать в filter, map и других методах для создания новых объектов.

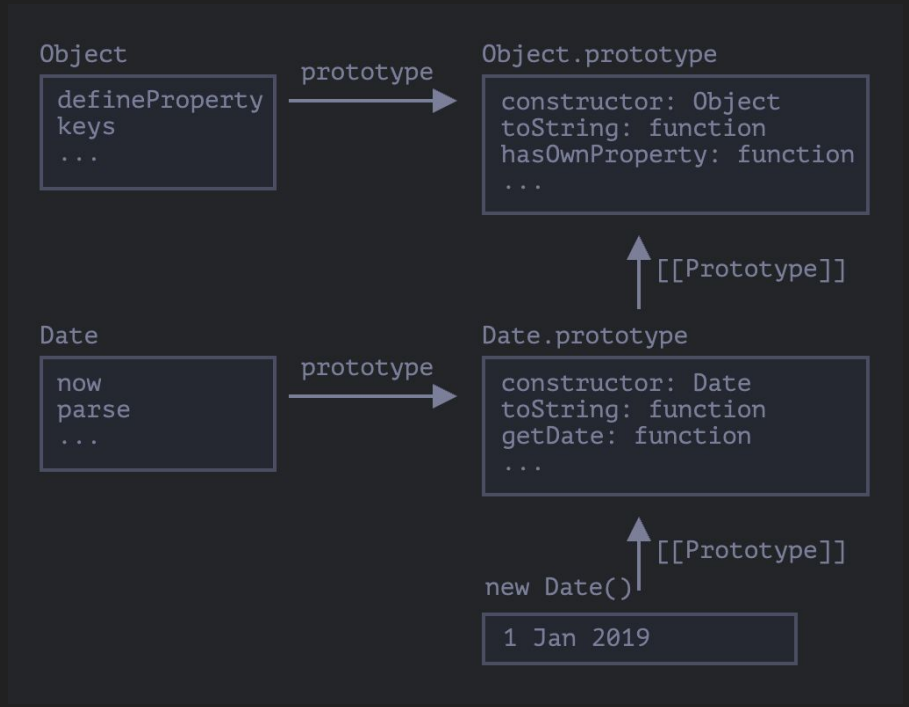
```
// добавим один метод (можно более одного)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // встроенные методы массива будут использовать этот метод как конструктор
  static get [Symbol.species]() {
    return Array;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter создаст новый массив, используя arr.constructor[Symbol.species] как конструктор
let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function
```

Отсутствие статического наследования встроенных классов

Обычно, когда один класс наследует другой, то наследуются и статические методы.

Но встроенные классы – исключение. Они не наследуют статические методы друг друга.



Проверка класса: "instanceof"

Оператор instanceof позволяет проверить, принадлежит ли объект указанному классу, с учетом наследования.

Такая проверка может потребоваться во многих случаях. Здесь мы используем её для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

obj instanceof Class

Оператор вернет true, если obj принадлежит классу Class или наследующему от него.

	работает для	возвращает
`typeof`	примитивов	строка
`{}.toString`	примитивов, встроенных объектов, объектов с `Symbol.toStringTag`	строка
`instanceof`	объектов	true/false

Алгоритм работы obj instanceof Class

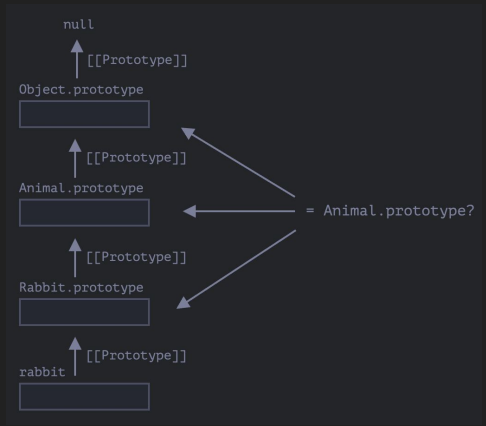
- Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false, и это конец. Это как раз и есть возможность ручной настройки instanceof.
- Большая часть классов не имеет метода Symbol.hasInstance. В этом случае используется стандартная логика: проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj.

```
// проверка instanceof будет полагать,
// что всё со свойством canEat - животное Animal
class Animal {
  static [Symbol.hasInstance](obj) {
    if (obj.canEat) return true;
  }
}

let obj = { canEat: true };
alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)
```

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// если какой-то из ответов true – возвратить true
// если дошли до конца цепочки – false
```

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // true

// rabbit.__proto__ === Animal.prototype (нет совпадения)
// rabbit.__proto__.__proto__ === Animal.prototype (совпадение!)
```

есть метод objA.isPrototypeOf(objB), который возвращает true, если объект objA есть где-то в прототипной цепочке объекта objB. Так что obj instanceof Class можно перефразировать как Class.prototype.isPrototypeOf(obj).



Особенности

Забавно, но сам конструктор Class не участвует в процессе проверки! Важна только цепочка прототипов Class.prototype.

Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта.

```
function Rabbit() {}
let rabbit = new Rabbit();

// заменяем прототип
Rabbit.prototype = {};

// ...больше не rabbit!
alert( rabbit instanceof Rabbit ); // false
```

Object.prototype.toString возвращает тип

Мы уже знаем, что обычные объекты преобразуются к строке как [object Object]:

Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Мы можем использовать его как расширенную версию typeof и как альтернативу instanceof.

Согласно спецификации встроенный метод toString может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

- Для числа это будет [object Number]
- Для булева типа это будет [object Boolean]
- Для null: [object Null]
- Для undefined: [object Undefined]
- Для массивов: [object Array]
- …и т.д. (поведение настраивается).

```
let obj = {};

alert(obj); // [object Object]
alert(obj.toString()); // то же самое
```

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag.

Такое свойство есть у большей части объектов, специфичных для определенных окружений.

В итоге мы получили «typeof на стероидах», который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

```
// toStringTag для браузерного объекта и класса
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

Примеси - в общем

Примесь – общий термин в объектно-ориентированном программировании: класс, который содержит в себе методы для других классов.

Некоторые другие языки допускают множественное наследование. JavaScript не поддерживает множественное наследование, но с помощью примесей мы можем реализовать нечто похожее, скопировав методы в прототип.

Мы можем использовать примеси для расширения функциональности классов, например, для обработки событий, как мы сделали это выше.

С примесями могут возникнуть конфликты, если они перезаписывают существующие методы класса. Стоит помнить об этом и быть внимательнее при выборе имён для методов примеси, чтобы их избежать.

Примеси в JavaScript

Примеси (mixins) в JavaScript — это способ добавления функциональности к классам без использования наследования. Примеси позволяют комбинировать свойства и методы из нескольких источников в одном классе, что делает код более модульным и повторно используемым.

Примеси особенно полезны, когда нужно добавить одинаковую функциональность к нескольким классам, не создавая сложной иерархии наследования.

Преимущества примесей

- Повторное использование кода: Примеси позволяют использовать одну и ту же функциональность в нескольких классах без дублирования кода.
- Модульность: Примеси способствуют созданию небольших, изолированных модулей кода, которые легко комбинировать.
- Гибкость: Примеси могут добавлять свойства и методы к любому классу, не меняя его исходного кода или структуры наследования.

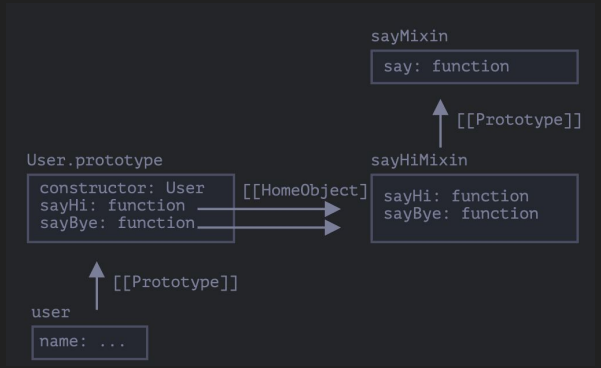
Ограничения и недостатки

- Потенциальные конфликты: Если две примеси добавляют методы или свойства с одинаковыми именами, это может привести к конфликтам.
- Запутанность: Использование большого количества примесей может сделать код сложным для понимания и отладки.
- Невозможность использования с приватными полями: Примеси не могут напрямую взаимодействовать с приватными полями и методами (начинающимися с #), что может ограничить их полезность в некоторых сценариях.

Простой пример

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть легко добавлены в прототип любого класса.

Примеси могут наследовать друг друга.




```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin,

  sayHi() {
    // вызываем метод родителя
    super.say(`Привет, ${this.name}`); // (*)
  },
  sayBye() {
    super.say(`Пока, ${this.name}`); // (*)
  }
};

class User {
  constructor(name) {
    this.name = name;
  }
}

// копируем методы
Object.assign(User.prototype, sayHiMixin);

// теперь User может сказать Привет
new User("Вася").sayHi(); // Привет, Вася!
```

Более сложный пример

Примеси в JavaScript обычно реализуются как функции, которые принимают класс и возвращают новый класс с добавленной функциональностью.

Примеси могут включать и более сложную функциональность, взаимодействовать с приватными свойствами и методами и использовать статические методы.

```
const timestampMixin = (Base) => class extends Base {
  constructor(...args) {
    super(...args);
    this.created = new Date();
  }

  getTimestamp() {
    return this.created;
  }
};

class Document {
  constructor(title) {
    this.title = title;
  }

  describe() {
    return `This document is titled "${this.title}".`;
  }
}

class TimestampedDocument extends timestampMixin(Document) {}

const doc = new TimestampedDocument('My Document');
console.log(doc.describe()); // This document is titled "My Document".
console.log(doc.getTimestamp()); // Время создания документа
```

Ресурсы

Класс: базовый синтаксис - ТЫК

Наследование классов - ТЫК

Статические свойства и методы - ТЫК

Приватные и защищенные методы и свойства - ТЫК

Расширение встроенных классов - ТЫК

Проверка класса: "instanceof" - ТЫК

Примеси - ТЫК

Приложение TODO реализованное с помощью классов - ТЫК