Front-end

JavaScript

JS. Object

- Прототипное/классовое наследование
- Классы
- ООП

JS. Classes

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

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё.

Как мы уже знаем функция-конструктор, оператор "new", с этим может помочь new function.

Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

JS. Classes

В ООП объект представляет собой блок, содержащий информацию (состояние / атрибуты) и операции (методы).

Ключевое слово this

- this - это объект, свойством которого является функция;
- this - дает функциям доступ к своему объекту и его свойствам;
- this - помогает выполнить один и тот же код для нескольких объектов;
- this - можно рассматривать как кто меня вызвал?; т.е. то, что находится слева от точки. Например, window.a();
- this - имеет динамическую область, т. е. не важно, где он был написан, важно, где он был вызван.

JS. Classes

Классы используются для создания экземпляров. Экземпляр — это объект, содержащий данные и логику класса.

Экземпляры создаются с помощью оператора new: instance = new Class().

JS. Classes

Синтаксис «class»

Базовый синтаксис выглядит так:

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

JS. Classes

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами.

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

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    console.log(this.name);
  }
}
let user = new User("Иван");
user.sayHi();
```

JS. Classes

Когда вызывается new User("Иван"):

Создаётся новый объект.

constructor запускается с заданным аргументом и сохраняет его в this.name.

Затем можно вызывать на объекте методы, такие как user.sayHi()

Методы в классе не разделяются запятой

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке.

Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются.

JS. Classes

Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что мы можем сделать всё то же самое без конструкции class

JS. Classes

Инициализация: constructor()

constructor(...) это специальный метод в теле класса, который инициализирует экземпляр. Это место, где вы можете установить начальные значения для полей или выполнить любые настройки объектов.

В следующем примере конструктор устанавливает начальное значение поля name:

```
class User {

  constructor(name) {

    this.name = name

  }

}
```

constructor класса User использует один параметр name, который используется для установки начального значения поля this.name.

Внутри конструктора значение this равно вновь созданному экземпляру.

Аргументы, используемые для создания экземпляра класса, становятся параметрами конструктора

JS. Classes

Поля

Поля класса — это переменные, содержащие определенную информацию. Поля могут быть разделены на две группы:

- Поля экземпляров класса
- Поля самого класса (статические)

Поля также имеют два уровня доступа:

- Открытые (публичные): поля доступны как внутри класса, так и в экземплярах
- Частные (приватные): поля доступны только внутри класса

JS. Classes

Открытые поля экземпляров класса

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

Выражение this.name = name создает поле экземпляра name и присваивает ему начальное значение.

Доступ к этому полю можно получить с помощью аксессора свойства:

```
const user = new User('Печорин')
user.name // Печорин
```

В данном случае name — открытое поле, поскольку оно доступно за пределами класса User.

JS. Classes

Частные поля экземпляров класса

Инкапсуляция позволяет скрывать внутренние детали реализации класса. Тот, кто использует инкапсулированный класс, опирается на публичный интерфейс, не вдаваясь в подробности реализации класса.

Хорошим способом скрыть детали является использование частных полей. Такие поля могут быть прочитаны и изменены только внутри класса, которому они принадлежат. За пределами класса частные поля недоступны.

JS. Classes

Частные поля экземпляров класса

Для того, чтобы сделать поле частным, перед его названием следует поставить символ #, например, #myPrivateField. При обращении к такому полю всегда должен использоваться указанный префикс.

JS. Classes

```
class User {
    #name
    constructor(name) {
        this.#name = name
    }
    getName() {
        return this.#name
    }
}

const user = new User('Печорин')
user.getName() // Печорин
user.#name // SyntaxError
```

JS. Classes

#name — частное поле. Доступ к нему можно получить только внутри класса User. Это позволяет сделать метод getName().

Однако, при попытке получить доступ к #name за пределами класса User будет выброшена синтаксическая ошибка

JS. Classes

Открытые статические поля

В классе можно определить поля, принадлежащие самому классу: статические поля. Такие поля используются для создания констант, хранящих нужную классу информацию.

Для создания статических полей используется ключевое слово static перед названием поля: static myStaticField.

JS. Classes

Открытые статические поля

Добавим новое поле type для определения типа пользователя: администратора или обычного. Статические поля TYPE_ADMIN и TYPE_REGULAR — константы для каждого типа пользователей:

JS. Classes

Открытые статические поля

```
class User {
  static TYPE_ADMIN = 'admin'
  static TYPE_REGULAR = 'regular'
  name
  type
  constructor(name, type) {
    this.name = name
    this.type = type
  }
}
const admin = new User('Администратор сайта', User.TYPE_ADMIN)
admin.type === User.TYPE_ADMIN // true
```

JS. Classes

Частные статические поля

Иногда статические поля также являются частью внутренней реализации класса. Для инкапсуляции таких полей можно сделать их частными.

Для этого следует перед названием поля поставить префикс #: static #myPrivateStaticFiled.

Предположим, что мы хотим ограничить количество экземпляров класса User. Для сокрытия информации о количестве экземпляров можно создать частные статические поля

JS. Classes

```
class User {
  static #MAX_INSTANCES = 2
  static #instances = 0
  name
  constructor(name) {
    User.#instances++
    if (User.#instances > User.#MAX_INSTANCES) {
      console.log('Невозможно создать экземпляр класса User')
    }
    this.name = name
  }
}
new User('Печорин')
new User('Бэла')
new User('Грушницкий') // Невозможно создать экземпляр класса User
```

JS. Classes

Статическое поле User.#MAX_INSTANCES определяет допустимое количество экземпляров, а User.#instances — количество созданных экземпляров.

Эти частные статические поля доступны только внутри класса User. Ничто из внешнего мира не может повлиять на ограничения

JS. Classes

Метод также может быть частным. Для того, чтобы сделать метод частным следует использовать префикс #.

Сделаем метод getName() частным:

```
class User {
  #name
  constructor(name) {
    this.#name = name
  }
  #getName() {
    return this.#name
  }
  nameContains(str) {
    return this.#getName().includes(str)
  }
}
const user = new User('Печорин')

user.nameContains('Печорин') // true user.nameContains('Грушницкий') // false  user.#getName // SyntaxError
```

JS. Classes

Статические методы

Статические методы — это функции, принадлежащие самому классу. Они определяют логику класса, а не его экземпляров.

Для создания статического метода используется ключевое слово static перед названием метода: static myStaticMethod().

При работе со статическими методами, следует помнить о двух простых правилах:

Статический метод имеет доступ к статическим полям

Он не имеет доступа к полям экземпляров (не используется this)

JS. Classes

Создадим статический метод для проверки того, что пользователь с указанным именем уже создан:

```
class User {
  static #takenNames = []
  static isNameTaken(name) {
    return User.#takenNames.includes(name)
  }
  name = 'Имярек'
  constructor(name) {
    this.name = name
    User.#takenNames.push(name)
  }
}
const user = new User('Печорин')
User.isNameTaken('Печорин') // true
User.isNameTaken('Грушницкий') // false
```

JS. Classes

isNameTaken() — статический метод, использующий частное статическое поле User.#takenNames для определения использованных имен.

Статические методы также могут быть частными: static #myPrivateStaticMethod(). Такие методы могут вызываться только внутри класса.

JS. Classes

Ключевое слово **extends** используется в объявлении класса или в выражениях класса для создания дочернего класса.

Синтаксис

```
class ChildClass extends ParentClass { ... }
```

JS. Classes

super()

Обычно вы используете super() для вызова конструктора родителя

```
class Fish {
 constructor(habitat, length) {
  this.habitat = habitat
  this.length = length
 }
}
class Trout extends Fish {
 constructor(habitat, length, variety) {
  super(habitat, length)
  this.variety = variety
 }
}
```

JS. Classes

Что такое прототипы?

При создании объектов, например, с помощью конструктора, каждый из них будет содержать специальное внутреннее свойство [[Prototype]], указывающее на его прототип. В JavaScript прототипы используются для организации наследования.

JS. Classes

```
// конструктор Box
function Box(width, height) {
  this.width = width;
  this.height = height;
}
```

При объявлении конструктора или класса у него автоматически появится свойство prototype. Оно содержит прототип. Прототип – это объект. В данном случае им будет являться Box.prototype. Это очень важный момент.

Этот прототип будет автоматически назначаться всем объектам, которые будут создаваться с помощью этого конструктора:

```
// создание объекта с помощью конструктора Box
const box1 = new Box(25, 30);
```

Таким образом при создании объекта, в данном случае, box1, он автоматически будет иметь ссылку на прототип, то есть на свойство Box.prototype.

JS. Classes

Проверяем

```
Object.getPrototypeOf(box1) === Box.prototype // true

box1.__proto__ === Box.prototype // true
```

JS. Classes

Получить прототип объекта в JavaScript можно с помощью статического метода Object.getPrototypeOf или специального свойства __proto__. В этом примере показаны два этих способа. Кстати, свойство __proto__ не является стандартным, но оно поддерживается всеми браузерами.

Свойство prototype имеется у каждой функции за исключением стрелочных. Это свойство как мы уже отмечали выше в качестве значения имеет объект. По умолчанию в нём находится только одно свойство constructor, которое содержит ссылку на саму эту функцию:

Box.prototype.constructor = Box // true

То есть Box.prototype.constructor – это сам конструктор Box

JS. Classes

Так что же такое прототип? Прототип в JavaScript – это просто ссылка на объект, который используется для наследования.

JS. Classes

Создаем экземпляры головного объекта

Теперь мы можем использовать эту функцию конструктор для создания отдельных экземпляров машин.

```
const tesla = new Auto();

console.log(tesla); // Object { }
```

Таким образом, мы получили экземпляр нашего головного объекта Auto - пустой объект, присвоенный нашей переменной tesla.

JS. Classes

Теперь, давайте добавим какие-то свойства в каждый из наших экземпляров головного объекта Auto. Для этого мы используем нашу функцию конструктор:

```
function Auto(brand, color, gas) {
 this.brand = brand;
 this.color = color;
 this.gas = gas;
}
```

Внутри функции Auto, значение this относится к каждому конкретному экземпляру, который мы будем создавать:

```
const bmw = new Auto('bmw', 'white', 100);
const nissan = new Auto('nissan', 'black', 100);
```

Мы создали 2 экземпляра нашего головного объекта Auto. Каждый экземпляр обладает своими собственными свойствами.

JS. Classes

Теперь давайте попробуем добавить какие-то методы к машинам. Метод drive будет расходовать бензин каждой машины (свойство gas):

```
function Auto(brand, color, gas) {
  this.brand = brand;
  this.color = color;
  this.gas = gas;
  this.drive = function () {
    if (this.gas > 0) {
      this.gas = this.gas - 10;
      return this.gas;
      console.log(`Уровень бензина = ${this.gas}`);
    } else {    console.log(`Бензин закончился!`);    }
  };
}
const bmw = new Auto('bmw', 'white', 100);
const nissan = new Auto('nissan', 'black', 100);
bmw.drive();
```

JS. Classes

Все отлично работает - но есть небольшой ньюанс!

Каждый раз, создавая новый экземпляр машины, мы также создаем новую функцию drive (для каждого экземпляра)!

Как в этом убедиться?

Давайте сравним функцию drive у разных экземпляров машин:

```
bmw.drive === nissan.drive; // false
```

Так что же в этом плохого?

В текущем примере, где у нас всего 2 экземпляра машин - проблем никаких нет. Но если количество экземпляров возрастет до 1,000 или десятков тысяч, это окажет негативное влияние на производительность.

JS. Classes

Чтобы не создавать каждый раз новый метод drive() для каждого экземпляра машины, мы можем поместить этот метод в, так называемый, прототип (prototype) нашего головного объекта.

```
function Auto(brand, color, gas) {
  this.brand = brand;
  this.color = color;
  this.gas = gas;
}
Auto.prototype.drive = function () {
  if (this.gas > 0) {
    this.gas = this.gas - 10;
    return this.gas;
    console.log(`Уровень бензина = ${this.gas}`);
  } else {
    console.log(`Бензин закончился!`);
  }
};
const bmw = new Auto('bmw', 'white', 100);
const nissan = new Auto('nissan', 'black', 100);
```

JS. Classes

Таким образом мы создадим всего один метод drive(), который будет использоваться всеми экземплярами наших машин.

В этом и заключается суть прототипного наследования. Мы получаем метод, к которому имеют доступ все экземпляры нашего головного объекта. При этом мы не создаем большого количества копий этого метода для каждого экземпляра.

Теперь давайте посмотрим на экземпляр нашего объекта в переменной nissan:

```
nissan; // Object { brand: "nissan", color: "black", gas: 100 }
```

Мы видим, что метод drive пропал из списка свойств. Это происходит потому, что теперь метод drive "живет" в прототипе объекта.

Теперь, когда мы будем запускать метод drive, используя какой-либо экземляр, Javascript будет искать этот метод сначала в свойствах экземпляра, а потом прототипе.

JS. Classes

Надо сказать, что синтаксис классов — это хорошая абстракция над прототипным наследованием. Для использования классов не нужно обращаться к прототипам.

Однако, классы являются лишь надстройкой над прототипным наследованием. Любой класс — это функция, создающая экземпляр при вызове конструктора.

Следущие два примера идентичны.

JS. Classes

Классы:

```
class User {
   constructor(name) {
      this.name = name
   }
   getName() {
      return this.name
   }
}

const user = new User('Печорин')
user.getName() // Печорин
user instanceof User // true
```

JS. Classes

Прототипы:

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

User.prototype.getName = function () {
   return this.name
}

const user = new User('Печорин')
user.getName() // Печорин
user instanceof User // true
```

JS. Classes

Ключевым принципом ООП выступает разделение задач и ответственностей по сущностям.

Сущности создаются в коде как объекты. При этом каждая из них объединяет некую информацию (свойства) и действия (методы), которые может выполнять.

JS. Classes

Принципы ООП:

- Наследование
- Инкапсуляция
- Абстракция
- Полиморфизм

JS. Classes

Наследование

Наследование – это возможность создавать классы на основе других классов.

С помощью этого принципа можно определять родительский класс (с нужными свойствами и методами), а затем дочерний класс, который будет наследовать от родителя все свойства и методы.

JS. Classes

Наследование

- Класс может наследовать только от одного родителя. Расширять несколько классов нельзя, хотя для этого есть свои хитрости.
- Вы можете безгранично увеличивать цепочку наследования, устанавливая родительский, «дедовский», «прадедовский» и так далее классы.
- Если дочерний класс наследует какие-либо свойства от родительского, то он сначала должен присвоить эти свойства через вызов функции super() и лишь затем устанавливать свои.
- При наследовании все родительские методы и свойства переходят к потомку. Здесь мы не можем выбирать, что именно наследовать (так же, как не можем выбирать достоинства или недостатки, получаемые нами от родителей при рождении. К теме выбора мы ещё вернёмся при рассмотрении композиции).
- Дочерние классы могут переопределять родительские свойства и методы.

JS. Classes

Инкапсуляция

Инкапсуляция – это ещё один принцип ООП, который означает способность объекта «решать», какую информацию он будет раскрывать для внешнего мира, а какую нет. Реализуется этот принцип через публичные и закрытые свойства и методы.

В JS все свойства объектов и методы по умолчанию являются публичными (public). «Публичный» означает возможность доступа к свойству/методу объекта извне его тела.

Инкапсуляция полезна в случаях, когда нам требуются определённые свойства или методы исключительно для внутренних процессов объекта, и мы не хотим раскрывать их вовне. Наличие закрытых свойств/методов гарантирует, что мы «случайно» не раскроем эту информацию.

JS. Classes

Абстракция

Абстракция - это способ создания простой модели, которая содержит только важные свойства с точки зрения контекста приложения, из более сложной модели. Иными словами - это способ скрыть детали реализации и показать пользователям только функциональность.

Использование только тех характеристик объекта, которые с наибольшей точностью представляют его (поверхностное описание).

JS. Classes

Полиморфизм

Полиморфизм означает «множество форм», являясь, по сути, довольно простым принципом, отражающим способность метода возвращать разные значения, согласно определённым условиям.

Одно действие - несколько реализаций