ПМ3 Разработка модулей ПО.

РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.

Тема 1. Введение в ООП.

Лекция 4. Прототипы и прототипное наследование.

Цель занятия:

Познакомиться с прототипами в JavaScript и понять механизм наследования через цепочку прототипов.

Учебные вопросы:

1. Свойство prototype у функции-конструктора.
2. Как объекты получают доступ к методам через цепочку прототипов.
3. Добавление методов в прототип.
4. Переопределение методов в потомках.
5. Отличие собственных свойств объекта от унаследованных.
6. Примеры прототипного наследования.

1. Свойство prototype у функции-конструктора.

В JavaScript каждая функция (кроме стрелочных) имеет специальное свойство **prototype**.

Это объект, который используется как прототип для всех экземпляров, созданных через **new**.

В **prototype** обычно кладут методы, чтобы они не дублировались у каждого объекта.

 Важно: свойство **prototype** есть только у функций (которые можно вызвать с **new**), у обычных объектов его нет.

Пример. Проверка prototype.

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

// У функции есть свойство prototype
console.log(User.prototype); // { constructor: ƒ User }

// Экземпляры через new используют этот объект как прототип
const u1 = new User("Аня");
const u2 = new User("Петя");

console.log(Object.getPrototypeOf(u1) === User.prototype); // true
console.log(Object.getPrototypeOf(u2) === User.prototype); // true
```

Метод **Object.getPrototypeOf(obj)** — это стандартный способ получить прототип (т.е. скрытое внутреннее свойство **[[Prototype]]**) объекта.

◆ Что делает:

- Принимает объект.
- Возвращает его прототип (обычно это объект, например **User.prototype** или **Object.prototype**).
- Если прототипа нет → возвращает **null**.

Пример. Добавление метода в **prototype:**

```
function User(name) {
  this.name = name;
}
// Добавляем метод в прототип
User.prototype.sayHello = function() {
  console.log(`Привет, я ${this.name}`);
};

const u1 = new User("Вася");
const u2 = new User("Катя");

u1.sayHello(); // Привет, я Вася
u2.sayHello(); // Привет, я Катя

// Оба объекта используют один и тот же метод
console.log(u1.sayHello === u2.sayHello); // true

// { sayHello: [Function (anonymous)] }
console.log(Object.getPrototypeOf(u1))
```

Итог:

- **prototype** — это объект, который "делится" между всеми экземплярами, созданными конструктором.
- В **prototype** обычно кладут общие методы (экономия памяти, единое поведение).
- Каждый экземпляр объекта связывается с этим прототипом через внутреннее свойство [[Prototype]].

2. Как объекты получают доступ к методам через цепочку прототипов

Когда у объекта запрашивают свойство (например, метод), движок JavaScript ищет его сначала в самом объекте.

Если свойства нет → поиск продолжается в прототипе объекта (**[[Prototype]]**).

Если и там нет → поднимается по цепочке прототипов (**prototype chain**) до самого верха — **Object.prototype**.

Если свойство не найдено даже там → результат **undefined**.

Этот механизм называется **прототипное наследование**.

◇ Пример. Поиск свойства в объекте.

```
function User(name) {
  this.name = name;
}
User.prototype.sayHello = function () {
  console.log(`Привет, я ${this.name}`);
};

const u1 = new User("Вася");

// JS ищет метод:
// 1. В u1 → нет
// 2. В User.prototype → нашёл
u1.sayHello(); // "Привет, я Вася"
```

◇ Пример. Цепочка глубже.

```
const arr = [1, 2, 3];

// Проверим методы
console.log(arr.toString()); // "1,2,3"

// Где ищется метод?
// 1. В arr → нет
// 2. В Array.prototype → найден toString()
```

◆ Пример. Поднятие до Object.prototype.

```
const obj = { a: 10 };

// У объекта нет метода toString → JS ищет дальше
console.log(obj.toString()); // "[object Object]"

// Здесь сработал Object.prototype.toString
```

◆ Пример. Если свойства нет нигде.

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

const u2 = new User("Петя");

console.log(u2.walk); // undefined
// walk нет ни в u2, ни в User.prototype, ни в Object.prototype
```

Итог:

- Доступ к свойствам и методам работает через поиск по цепочке прототипов.
- Если свойство не найдено, результат будет **undefined**.
- Цепочка всегда заканчивается на **Object.prototype**, у которого [[Prototype]] = null.

3. Добавление методов в прототип.

Методы можно определять прямо внутри конструктора, но тогда при каждом вызове **new** они будут дублироваться в памяти.

Более оптимально — добавлять методы в **Function.prototype** конструктора.

Все экземпляры конструктора будут разделять один метод, что экономит память.

◆ Пример. Метод внутри конструктора (неэффективно).

```
function User(name) {
  this.name = name;
  this.sayHello = function () {   // каждый раз создаётся новая функция
    console.log(`Привет, я ${this.name}`);
  };
}

const u1 = new User("Вася");
const u2 = new User("Петя");

console.log(u1.sayHello === u2.sayHello); // false (разные функции)
```

◆ Пример. Метод в prototype (оптимально).

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

User.prototype.sayHello = function () {
  console.log(`Привет, я ${this.name}`);
};

const u1 = new User("Вася");
const u2 = new User("Петя");

u1.sayHello(); // "Привет, я Вася"
u2.sayHello(); // "Привет, я Петя"

console.log(u1.sayHello === u2.sayHello); // true (одна и та же функция)
```

◆ Пример. Добавление нескольких методов.

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

User.prototype.sayHello = function () {
  console.log(`Привет, я ${this.name}`);
};

User.prototype.rename = function (newName) {
  this.name = newName;
};

const u = new User("Аня");
u.sayHello(); // "Привет, я Аня"
u.rename("Оля");
u.sayHello(); // "Привет, я Оля"
```

Пример. Переопределение метода.

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

User.prototype.sayHello = function () {
  console.log(`Привет, я ${this.name}`);
};

const u = new User("Вася");

// Переопределим метод только у одного объекта
u.sayHello = function () {
  console.log(`Хай, я ${this.name}`);
};

u.sayHello(); // "Хай, я Вася"
(new User("Петя")).sayHello(); // "Привет, я Петя"
```

Итог:

Методы в **prototype** — общие для всех объектов.

Методы внутри конструктора — уникальные для каждого экземпляра (и занимают лишнюю память).

Лучше выносить общие методы в **prototype**, а уникальные данные — в сам конструктор.

4. Переопределение методов в потомках.

В JavaScript наследование можно реализовать через цепочку прототипов.

Если в потомке определить метод с таким же именем, он переопределяет метод родителя.

При вызове метода движок идёт по цепочке прототипов и берёт первое найденное совпадение.

Пример. Родитель + потомок (override).

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.say = function () {
  console.log(`${this.name} издаёт звук`);
};

function Dog(name) {
  this.name = name;
}
```

```
// Унаследуем прототип Animal
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Переопределим метод say
Dog.prototype.say = function () {
  console.log(`${this.name} гавкает`);
};

const a = new Animal("Зверь");
a.say(); // "Зверь издаёт звук"

const d = new Dog("Шарик");
d.say(); // "Шарик гавкает"
```

Пример. Вызов метода родителя через call.

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.say = function () {
  console.log(`${this.name} издаёт звук`);
};

function Cat(name) {
  this.name = name;
}
Cat.prototype = Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;
```

```
// Расширяем метод, а не полностью заменяем
Cat.prototype.say = function () {
  Animal.prototype.say.call(this); // вызов метода родителя
  console.log(`${this.name} мяукает`);
};

const c = new Cat("Мурзик");
c.say();
// "Мурзик издаёт звук"
// "Мурзик мяукает"
```

Пример. Приоритет свойств.

```
function Animal() {}
Animal.prototype.type = "животное";

function Bird() {}
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Bird;

const b = new Bird();
console.log(b.type); // "животное"

b.type = "птица"; // переопределение свойства у экземпляра
console.log(b.type); // "птица"
delete b.type;       // убрали локальное свойство
console.log(b.type); // снова "животное" (из прототипа)
```

📝 Итог:

- Потомки могут заменять (override) методы и свойства родителя.
- Если метод не найден в объекте — поиск идёт вверх по цепочке прототипов.
- Чтобы использовать метод родителя внутри переопределённого — применяют Parent.prototype.method.call(this, …)

5. Отличие собственных свойств объекта от унаследованных.

В JavaScript свойства объекта могут быть:

- Собственные — определены прямо внутри объекта.
- Унаследованные — приходят из прототипа.

Иногда не нужно, чтобы объект "подхватил" чужие методы/свойства из прототипа.

Проверка свойств.

- Оператор in — проверяет наличие свойства (и собственного, и унаследованного).
- Метод hasOwnProperty — проверяет только собственные свойства.

Пример:

```
function Person(name) {
  this.name = name; // собственное свойство
}
Person.prototype.sayHello = function() { // унаследованное свойство
  console.log("Привет!");
};

const user = new Person("Аня");

// Проверка
console.log("name" in user);               // true (есть в объекте)
console.log("sayHello" in user);           // true (унаследовано из прототипа)

console.log(user.hasOwnProperty("name"));      // true (собственное)
console.log(user.hasOwnProperty("sayHello")); // false (унаследованное)
```

Итог:

- obj.hasOwnProperty(key) → проверка только собственных свойств.
- "key" in obj → проверка и собственных, и унаследованных свойств.

6. Примеры прототипного наследования

Прототипное наследование — это механизм, при котором один объект может использовать свойства и методы другого объекта через цепочку прототипов.

1. Наследование через prototype.

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayHello = function() {
  console.log(`Я животное, меня зовут ${this.name}`);
};

function Dog(name, breed) {
  Animal.call(this, name); // вызываем конструктор родителя
  this.breed = breed;
}
```

```
// Наследуем методы Animal
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Добавляем метод Dog
Dog.prototype.bark = function() {
  console.log("Гав!");
};

const dog = new Dog("Бобик", "Овчарка");

dog.sayHello(); // Я животное, меня зовут Бобик (унаследовано)
dog.bark();     // Гав! (собственный метод)
```

2. Наследование через Object.create.

```
const animal = {
  eat() {
    console.log("Я ем");
  }
};

const dog = Object.create(animal);
dog.bark = function() {
  console.log("Гав!");
};

dog.eat();  // Я ем (унаследовано из animal)
dog.bark(); // Гав! (собственный метод)
```

3. Современный синтаксис, через классы (изучим позже).

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`Привет, я ${this.name}`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Гав!");
  }
}

const dog = new Dog("Шарик");
dog.sayHello(); // Привет, я Шарик
dog.bark();     // Гав!
```

Итог:

- Наследование можно реализовать через prototype, через Object.create, или с помощью class / extends.
- Все эти способы используют один и тот же механизм прототипов внутри.

📝 Итоги лекции:

У каждой функции-конструктора есть свойство prototype, в котором можно хранить методы и свойства, общие для всех экземпляров.

При обращении к свойству или методу у объекта сначала ищется собственное свойство, а затем поиск продолжается по цепочке прототипов (prototype chain).

Методы, добавленные в prototype, не копируются в каждый объект, а разделяются всеми экземплярами → это экономит память.

Можно переопределять методы: если в объекте задать метод с тем же именем, что и в прототипе, будет использован именно объектный метод.

Отличить собственное свойство от унаследованного помогает метод hasOwnProperty().

Прототипное наследование позволяет строить иерархии объектов и делиться общим функционалом.

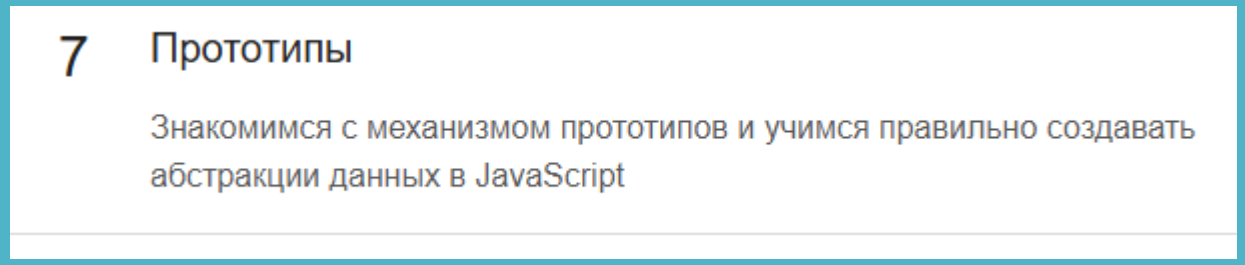
⚡ Современный JavaScript предлагает синтаксис **классов** (class), который решает те же задачи, но выглядит привычнее и проще для чтения. Под капотом классы используют всё те же прототипы.

Контрольные вопросы:

- Что такое прототип объекта?
- Как работает доступ к свойствам по цепочке прототипов?
- Чем отличается метод, определённый в конструкторе, от метода в прототипе?
- Как проверить, что свойство — собственное, а не унаследованное?
- Как классы связаны с прототипами?

Домашнее задание:

1. https://ru.hexlet.io/courses/js-introduction-to-oop



2. Повторить материал лекции.

Материалы лекций:

https://github.com/ShViktor72/Education2025

Обратная связь:

colledge20education23@gmail.com