**Лабораторная работа №24**

**Тема работы:** Разработка программ с применением классов и наследования в JavaScript.

**Цель работы:** закрепить навыки по работе с классами и их наследованием.

**Теоретические сведения**

**Наследование классов**

Допустим, у нас есть два 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.`);

}

}

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

…И Rabbit:

class Rabbit {

constructor(name) {

this.name = name;

}

hide() {

alert(`${this.name} прячется!`);

}

}

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

Сейчас они полностью независимы.

Но мы хотим, чтобы Rabbit расширял Animal. Другими словами, кролики должны происходить от животных, т.е. иметь доступ к методам Animal и расширять функциональность Animal своими методами.

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}.

Ниже Rabbit наследует от 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.`);

}

}

// Наследуем от Animal указывая "extends Animal"

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!

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать метод run как и все животные.

Ключевое слово extends работает, используя прототипы. Оно устанавливает Rabbit.prototype.[[Prototype]] в Animal.prototype. Так что если метод не найден в Rabbit.prototype, JavaScript берёт его из Animal.prototype.

Как мы помним из главы [Встроенные прототипы](https://learn.javascript.ru/native-prototypes), в JavaScript используется наследование на прототипах для встроенных объектов. Например Date.prototype.[[Prototype]] это Object.prototype, поэтому у дат есть универсальные методы объекта.

**После extends разрешены любые выражения**

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

Пример вызова функции, которая генерирует родительский класс:

function f(phrase) {

return class {

sayHi() { alert(phrase) }

}

}

class User extends f("Привет") {}

new User().sayHi(); // Привет

Здесь class User наследует от результата вызова f("Привет").

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

**[Переопределение методов](https://learn.javascript.ru/class-inheritance" \l "pereopredelenie-metodov)**

Давайте пойдём дальше и переопределим метод. Сейчас Rabbit наследует от Animal метод stop, который устанавливает this.speed = 0.

Если мы определим свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

class Rabbit extends Animal {

stop() {

// ...будет использован для rabbit.stop()

}

}

…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(...) вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

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!

Теперь у класса Rabbit есть метод stop, который вызывает родительский super.stop() в процессе выполнения.

**У стрелочных функций нет super**

Как упоминалось в главе [Повторяем стрелочные функции](https://learn.javascript.ru/arrow-functions), стрелочные функции не имеют super.

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

class Rabbit extends Animal {

stop() {

setTimeout(() => super.stop(), 1000); // вызывает родительский stop после 1 секунды

}

}

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:

// Unexpected super

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

**[Переопределение конструктора](https://learn.javascript.ru/class-inheritance" \l "pereopredelenie-konstruktora)**

С конструкторами немного сложнее.

До сих пор у Rabbit не было своего конструктора.

Согласно [спецификации](https://tc39.github.io/ecma262/#sec-runtime-semantics-classdefinitionevaluation), если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор:

class Rabbit extends Animal {

// генерируется для классов-потомков, у которых нет своего конструктора

constructor(...args) {

super(...args);

}

}

Как мы видим, он просто вызывает конструктор родительского класса. Так будет происходить, пока мы не создадим собственный конструктор.

Давайте добавим конструктор для Rabbit. Он будет устанавливать earLength в дополнение к name:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

this.speed = 0;

this.name = name;

this.earLength = earLength;

}

// ...

}

// Не работает!

let rabbit = new Rabbit("Белый кролик", 10); // Error: this is not defined.

Упс! При создании кролика – ошибка! Что не так?

Если коротко, то в классах-потомках конструктор обязан вызывать super(...), и (!) делать это перед использованием this.

…Но почему? Что происходит? Это требование кажется довольно странным.

Конечно, всему есть объяснение. Давайте углубимся в детали, чтобы вы действительно поняли, что происходит.

В JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]:"derived".

Разница в следующем:

* Когда выполняется обычный конструктор, он создаёт 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.

Поэтому, если мы создаём собственный конструктор, мы должны вызвать super, в противном случае объект для this не будет создан, и мы получим ошибку.

Чтобы конструктор Rabbit работал, он должен вызвать super() до того, как использовать this, чтобы не было ошибки:

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

**Порядок выполнения работы**

Задача 1

Создайте базовый класс, который будет описывать некоторую геометрическую фигуру, на его основе создайте три дочерних класса описывающих примоугольник, ромб и трапецию, классы должны позволять вычислиять значения диагоналей, площади, периметри, высоты (Там, где она есть), углов у основания.