**Ресурси:**

1. <https://docs.google.com/presentation/d/198EvGeoV5TEg7ThMmiMPFvpRThbSGQ-Pm4uwCer_pD8/edit#slide=id.g4f0ce183c7_2_65>
2. <https://docs.google.com/presentation/d/1MY0RBMHO9uoeeoMtG40NiyZFKCHQuF8BXuh25ygXJIs/edit#slide=id.g33cb0b2a8f_0_49>

**ООП** - це не конструкція мови, а не технологія. Це спосіб організації коду в єдину логічно осмислену структуру з метою оптимізації коду і забезпечення ефективної взаємодії всіх компонентів ООП-системи.

**Наслідування в JS**

**Ресурси:**

1. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Inheritance_and_the_prototype_chain>
2. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf>
3. <https://learn.javascript.ru/function-prototype>
4. <https://habr.com/ru/post/131714/>
5. <https://learn.javascript.ru/class-inheritance> - класове наслідування
6. <https://learn.javascript.ru/native-prototypes> - prototypes
7. <https://habr.com/ru/company/nix/blog/302518/>
8. <https://www.freecodecamp.org/news/prototype-in-js-busted-5547ec68872/> - \_\_proto\_\_ (English)
9. <https://betterprogramming.pub/prototypes-in-javascript-5bba2990e04b> (English - чудово пояснено)

**Поліфіл** - це термін, який означає емуляцію методу, який існує в специфікації JavaScript, але ще не підтримується поточним движком JavaScript.

**Функції контсруктори:**

1. Функції-конструктори мають назву з великої літери
2. Звичайна функція стає функцією контсруктором, коли викликається з new
3. Тоді в даному методі утворюється пустий об’єкт до якого прив’язується this

***Завдяки методу шаблону*** – ми створюємо нові об’єкти з набором властивостей і методів, самі об’єкти не однакові, але створені за спільним шаблоном

**Принцип роботи: const str = new String();**

1. Створюємо новий об’єкт
2. На цей об’єкт встановлюється посилання на this
3. Повертається даний об’єкт

***Екземпляр класу*** – це об’єкти створені від цього класу

**Статичні методи класу** - це такі методи, які можуть бути викликані без створення екземпляра класу.

function Box(title) { this.title = title; }  
  
Box.getBoxClassName = function () { return “Box”; }  
Box.getBoxClassName();

**Способи наслідування**

**Наслідування через Object.create(**parent**):**

Якщо властивості prototype будь-якого конструктора добавити метод, то цей метод буде доступним для усіх екземплярів цього класу. Задати прототипне наслідування можна і іншим способом.

1. Створити об’єк поля і методи якого ми хочемо використовувати в похідних від нього об’єктах
2. Ствоюємо новий об’єкт через **Objec.create(**parent-object**).** Таким чином наш об’єкт-батько буде прототипом для нового об’єкта нащадка і всі його методи будуть доступні нащадку, при бажанні другим параметром можна задати дискріптор для нового об’єкта і вписати йому властивості.
3. **Child.prototype = Object.create(Parent.prototype***);* - Прототип батька присвоюємо прототипу наслідника
4. Для прототипа наслідника, який був заміненим прототипом батька потрібно повернути його конструктор - **Child.prototype.constructor = Child;**

**Функціональне наслідування / прототипне**

В конструкторі, який хоче унаслідувати методи викликамо конструктор бадьківського об’єкта завдяки контекту виклику call. Якщо параметрів багато, то можемо використовувати **apply** для передачі параметрів констурктору батька через **arguments**

1. ParentConstructor.**call**(this, arg1, arg2, …);
2. Всі персональні поля для цього конструктора ми задаємо після функціонального наслідування
3. Якщо методи батьківського констуктора реалізовані напряму в prototype даного об’єкта, то функціональним методом наслідування вони не перенесуться в дочірній конструктор і доведеться переопреділити дочірній prototype

Child.**prototype** = Object.**create**(Parent.**prototype**);

Child.**prototype**.constructor = Child;

1. Всі свої методи ми об’являємо лише після цих 2-х строк вище

**Prototype** – об’єкт, який є у всіх об'єктів, і містить властивості та методи своїх батьків (і властивість батьків батьків - вгору по ланцюжку, до самого верху). Даний об’єкт доступний усім екземплярам класу чи функції конструктора породженими від них і JS при виклику методу в цих екземплярів спочатку шукає їх в даних екзепляраї, а якщо їх там немає, шукає в прототипі і вверх по ланцюгу усіх батьків. Об’являючи метод в прототипі функції конструктора js не дублює її для кожного екземпляру, але дає їм доступ до даного методу через доступ до прототипу батьківського об’єкта.

При створенні об'єкта через new, в його прототип \_\_proto\_\_ записується посилання з prototype функції-конструктора.

В JavaScript об'єкти мають спеціальне приховане властивість [[Prototype]]. Властивість **\_\_proto\_\_** - історично обумовлений геттер / сетер для [[Prototype]]. Коли ми хочемо прочитати властивість з object, а воно відсутнє, JavaScript автоматично бере його з прототипу окб’єкта батька, якщо даний метод там є. У програмуванні такий механізм називається «прототипна спадкуванням».

Посилання не можуть йти по колу. JavaScript видасть помилку, якщо ми спробуємо призначити **\_\_proto\_\_** по колу.

Значення **\_\_proto\_\_** може бути об'єктом або null. Інші типи ігноруються.

**\_\_proto\_\_** - посилання на prototype батьківського об’єкта

Фактично **\_\_proto\_\_**of **Function** та **prototype** of - **Function** це два вказівники, що відносяться до одного і того ж об’єкта.

**Прототипне спадкування** - це коли нащадок (екземпляр, об'єкт) має доступ до властивостей і методів свого батька (конструктора, класу).

Будь-яка функція містить властивість prototype (який є об'єктом). За замовчуванням prototype містить тільки одну властивість: constructor - яке зберігає посилання на функцію:

Кожен об'єкт має внутрішню посилання на інший об'єкт, званий його prototype. У об'єкта-прототипу також є свій власний прототип і так далі до тих пір, поки ланцюжок не завершиться об'єктом, у якого властивість prototype одно null

Ми можемо використовувати властивість constructor існуючого об'єкта для створення нового.

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

Щоб добавити в prototype нову властивість не потірібно його переписувати ціляком, так як зникне властивість constructor, якщо її не задати під час перепису вручну. Безпечні способи задати prototype властивість (Rabbit.prototype.jumps = true) або (

Rabbit.prototype = {

jumps: true,

constructor: Rabbit

};

)

* Властивість F.prototype (не плутати з [[Prototype]]) встановлює [[Prototype]] для нових об'єктів при виклику new F ().
* Значення F.prototype має бути або об'єктом, або null. Інші значення не будуть працювати.
* Властивість "prototype" є особливим, тільки коли воно призначене функції-конструктору, яка викликається оператором new.

**Методи**

1. **Object**.**getPrototypeOf(**сhild**)** – метод для отримання пототипа об’єкта. Заглядаємо безпосередньо в властивість об’єкта \_proto\_, який є у будь-якого об’єкта
2. **child**.**hasOwnProperty**('name') – Повертає true/false в залежності від того рідна властивість об’єкта чи унаслідувана

**Наслідування через Класи:**

1. <https://learn.javascript.ru/private-protected-properties-methods>

Функції, створені через class, не можна викликати без оператора new

Для того, щоб наслідувати клас від іншого, ми повинні використовувати ключове слово **"extends"**

**Super** – ключове слово, яке являється посиланням на батьківську функцію. **This** всередині методу **super** в конструкторі посилатиметься на екземпляр класу-наслідника.

**Super** являється посиланням на батьківський прототип, якщо ми запускаємо super.method() в середині класу-наслідника, при цьому this всередині цього методу буде посиланням на екземпляр класу-наслідника. Завдяки **super** в екземплярі класу наслідника поле батьківського об’єкта можна викликати в контекті наслідника (**super(property)** аналог **Parent.call(this, property)**) і так само із методами, що спрощує читаємість коду. **Super.method() = Parent.prototype.method.call(this)**

**Наслідування через класи:**

class *Journal* extends *Book* {

  constructor(name, publisher) {

*super*(name); this.publisher = publisher;

  }

  getName() { return 'The name is ' + **super**.getName(); }

}

**Наслідування через прототип:**

function *Journal* (name, publisher) {

*Book*.call(this, name); this.publisher = publisher;

  }  
  
*Journal.*prototype = Object.create(*Book*.prototype);  
*Journal.*prototype.constructor = *Journal*;

*Journal.*prototype.getName = function () {   
  return 'The name is ' + *Book*.prototype.getName.call(this); }

}

**Статичні методи наслідуються також**

**Инкапсуляція JS**

1. <https://learn.javascript.ru/private-protected-properties-methods>
2. <https://medium.com/nuances-of-programming/%D0%B8%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F-%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F-%D0%B2-javascript-%D0%B1%D0%B5%D0%B7-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2-%D0%B2-2019-%D0%B3%D0%BE%D0%B4%D1%83-%D1%81-%D0%BF%D1%80%D0%B8%D0%B2%D0%B0%D1%82%D0%BD%D1%8B%D0%BC%D0%B8-%D0%BF%D0%BE%D0%BB%D1%8F%D0%BC%D0%B8-5431df2a3bf7>

**Инкапсуляция** - це коли:

а) дані та методи, пов'язані з логічним контекстом (загалом смыслом), об'єднуються в єдину сутність;

б) доступний цим даним розмежуванням на рівнях. Деталі реалізації скриваються в класі (найвищий рівень доступу)

Зазвичай доступ до даного зазвичай містить три:

1) приватні (приватні)

2) захищені

3) публічні (загальнодоступні)

**Способи задання приватності:**

При використанні prototype без this не обійтись. Тому для створення приватної змінної до неї додають ‘\_’ – **this**.**\_title**

В ES6 для задання приватності полям використовують **`#`,** завдяки чому властивості доступні лише в об’єкті

Використання замикання – Створюємо конструктор який містить поля і повертає методи взаємодії із ними чрез getter / setter вписаними в поверненому конструктором об’єкті.

Використання методу return Object.freeze({getter/setter}); - не зможемо setter-ом переоприділити значення поля

Використання Symbol() – в змінну присвоюємо значення Symbol() в конструкторі. В return, що повертає об’єкт – який містить в собі значення змінної під ключем параметра Symbol() – [symbol]:name і getter/setter, що вносять зміни до поля з даним ключем. Завдяки чому ми безпосередньо до поля не маємо доступу

В модульній системі приватність полів можна досягти за рахунок зберігання полів безпосередньо в модулі, а методи взаємодії із ними виносити в export. Таким чином взаємодіяти із даними можна лише через дані методи

**Поліморфізм JS**

Перевизначення вже снуючого методу для екемпляру класу

function **Metal**(title) {

  this.**\_title** = title || ''; // приватная переменная

}

Metal.prototype.**getName** = function () {

  return this.\_title;

}

**var** gold = new Metal('gold');

gold.**getName**();

*gold.****getName*** *= function () {*

*var originResult = Metal.prototype.****getName****.call(this); // вызвали оригинальный* ***getName***

*return 'This is a ' + originResult; // переопределили возвращаемый результат*

*};*

В новому синтаксисі приватність полів і методів позначається #

**GET / SET**

<https://developer.mozilla.org/uk/docs/Web/JavaScript/Reference/Functions/set>

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/get>

Спеціальні поля, що поводяться, як методи

При использовании get свойство будет определено в прототипе объекта, в то время, как при использовании [Object.defineProperty ()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) свойство будет определено в экземпляре, к которому применяется.