**Методи копіювання об’єкта start:**

**Методи не глибокого копіювання об’єктів:**

Копіює усі поля, але вложені об’єкти в копії і оригінаі матимуть посилання на одне місце в області пам’яті, що при зміні данних вложеного об’єкта-поля в копії приведе до його зміни і в оригіналі.

1. **for( let** key **in** obj**){}** – цикл для перебору об’єкта по ключам

Являється не найбезпечнішим способом перебору ключів об’єкта, так як може зайти в властивості по ланцюгу proto і для забезбечення безпечної перевірки ми задаємо перевірку obj**.hasOwnProperty(key)**, що перевіряє, чи ключі, що ми перебераємо належать безпосередньо поточному об’єкту і не підтягуються із proto

1. **Object.assign(**current-object || {}, new-obj, new-obj) – неглибоке копіювання полів інших об’єктів в поточний чи пустий

Перший параметр вказує об’єкт який буде модифікуватись

**Методи глибокого копіювання об’єктів:**

1. **JSON.parse(JSON.stringify(**obj**)) –** виконує глибоке копіювання, завдяки чому внутрішні об’єкти отримують посилання на окрему область в пам’яті і їх зміни в копії не впливають на той же об’єкт в оригіналі. **Повертає глибоку копію об’єкта.**
2. **const** newObj = {…obj} – через присвоєння і диструктуризацію.

**Методи непрямого копіювання об’єктів:**

1. **Object.create (**obj, descriptor**) –** **Повертає пустий об’єкт прототипом якого буде переданий в параметри create об’єкт**, що дозволить нам використовувати його поля і методи без прямого копіювання. Працює, як наслідування класів.

**Методи перевірки об’єкта:**

1. **Instanceof** – перевіряє чи об’єкт належить конкретному класу через ланцюжок prototype прототипів obj.

**Методи перебору і трансформування об’єкта:**

Методи не змінюють поточного об’єкта, а на його основі створюють масив

1. **Object.entries(**obj**)** – **Повертає масивне представлення об’єкта в вигляді масива масивів ключ-значення**

**[key, value].** // [ [key, velue], [key, velue] ]

1. **Object.fromEntries(**map.entries()**)** – переводить масив масивів ключ-значення в об’єкт
2. **Object.keys()** – **Повертає масив ключів об’єкта**
3. **Object.values()** - **Повертає масив значень об’єкта**
4. **Object**.**getOwnPropertyDescriptor(**obj, property**)** – поверне об’єкт дискріптор даного поля. Якщо задати лише один параметр (об’єкт) – то ми отримаємо об’єкт ключами якого будуть назви ключів поточного об’єкта, а їх значеннями будуть об’єкт-дискриптор кожного поля

**Методи зміни контексту об’єкту:**

1. **.call(**object-changed-this, param1, param2,…**)** – можемо позичати методи і виставляти контекст this, параметри

передаються в індивідуальному порядку, в залежності від к-сть аргументів від методу, з яким ми взаємодіємо.

1. **.apply(**object-changed-this, [param1, param2, …]**)** – можемо позичати методи і виставляти контекст this, приймає довільну к-сть параметрів у вигляді масива для методів, які мають змінне число аргументів. \* Якщо в варіативний метод (з плаваючою кількістю аргументів) ми хочемо передати масив аргументів, кожен елемент якого в нас має бути окремим аргументом на пряму – так не вийде. Методи із плаваючою к-стю аргументів не приймає масив елементів, як окремі аргументи, тому, якщо нам потрібно передати такий масив, кожен елемент якого має бути аргументом методу – використовуємо apply

Методи call i apply – використовують, щоб присвоювати this callback методам

function getUserInput (firstName, lastName, callback, callbackObj) {  
**// Использование метода apply ниже, выставит this для callbackObj**  
callback.apply (callbackObj, [firstName, lastName]);  
}

Методи call i apply – використовуються для більш гнучкого позичання методів

а. позичаємо методи в масивів / строк - **Array.prototype.method.call(**object-like-array, param**)**

1. **.bind(**object-changed-this, param1, param2,…**)** **– Створює нову функцію з тим самим тілом і діями, що і поточна, але там, де в поточної this – в нової функції буде пов’язано із першим елементом bind**.

Іншими словами дозволяє нам легко виставити який саме об’єкт буде прив’язаний до this, в момент виклику функції чи методу. This метода для якого ми викликаємо даний метод матиме значення object, який ми передаємо, як параметр.

Ще ми можемо позичати методи викликаючи їх із інших об’єктів в контексті переданого в параметри об’єкта. Але є ризик переписати вже існуючий метод в об’єктів, який позичає метод з таким же іменем. Тому найкраще позичати методи через Apply or Call

// cars.showData = user.showData.bind(cars);

На додачу ми можемо карірувати функцію з допомогою bind.

//var greetAYoungster = greet.bind (null, "", 16);  
//greetAYoungster ("Alex"); **// "Hey, Alex."**  
//greetAYoungster ("Emma Waterloo"); **// "Hey, Emma Waterloo."**

Першому параметру bind ми передали null, так як НЕ викликаємо метод в контексті іншого об'єкта, і вписуємо 2/3 параметрів даного методу в bind, завдяк чому всі подальші виклики даного методу каріруваного через p будут здійснюватісь вже із вказаним 2-ма параметрами по замовчуванню.