

## Успадкування

Успадкування – базовий принцип об'єктно-орієнтованого програмування, згідно з яким створення нових класів може здійснюватись на основі вже існуючих.

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .


```
// --- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

obj.prop_1 = 7
obj.meth_1()
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .



```
// --- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

obj.prop_1 = 7
obj.meth_1()
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
// --- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

obj.prop_1 = 7
obj.meth_1()
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
//--- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

//----- Додаємо предка у __proto__
obj.__proto__ = {
  prop_2: 22,
  meth_2: function () {
    document.write('meth_2')
  },
}

obj.prop_1 = 7
obj.meth_1()

obj.prop_2
obj.meth_2()
```

**ТАК РОБИТИ НЕ РЕКОМЕНДУЄТЬСЯ!!!**

```
> dir(obj)

▼ Object 1
  ▶ meth_1: f ()
  ▶ prop_1: 11
  ▼ __proto__:
    ▶ meth_2: f ()
      prop_2: 22
    ▶ __proto__: Object
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
//--- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

//----- Додаємо предка у __proto__
obj.__proto__ = {
  prop_2: 22,
  meth_2: function () {
    document.write('meth_2')
  },
}

obj.prop_1 = 7
obj.meth_1()

obj.prop_2
obj.meth_2()
```

```
> dir(obj)

▼ Object 1
  ▶ meth_1: f ()
  prop_1: 11
  ▼ __proto__:
    ▶ meth_2: f ()
    prop_2: 22
    ▶ __proto__: Object
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
//--- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

//----- Додаємо предка у __proto__
obj.__proto__ = {
  prop_2: 22,
  meth_2: function () {
    document.write('meth_2')
  },
}

obj.prop_1 = 7
obj.meth_1()

obj.prop_2
obj.meth_2()
```

```
> dir(obj)

▼ Object 1
  ▶ meth_1: f ()
  prop_1: 11
  ▼ __proto__:
    ▶ meth_2: f ()
    prop_2: 22
    ▶ __proto__: Object
```

## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
//--- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}
//----- Додаємо предка у __proto__
obj.__proto__ = {
  prop_2: 22,
  meth_2: function () {
    document.write('meth_2')
  },
}

obj.prop_3 = 33
```

```
> dir(obj)
▼ Object ⓘ
  ▶ meth_1: f ()
  prop_1: 7
  prop_3: 33
  ▼ [[Prototype]]: Object
    ▶ meth_2: f ()
    prop_2: 22
```



## Створення об'єктів на основі прототипів

### Особливості:

- прототип використовується тільки як резервне сховище властивостей для **ЧИТАННЯ** (якщо у об'єктів немає власної властивості, то цю властивість шукаємо у об'єкті прототипі (об'єкті, адреса якого міститься у властивості `__proto__`));
- операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об'єктом;
- при зміні значення властивості (яку читали з прототипу) у об'єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію) .

```
// --- Створюємо об'єкт
var obj = {
  prop_1: 11,
  meth_1: function () {
    document.write('meth_1')
  },
}

// ----- Додаємо предка у __proto__
obj.__proto__ = {
  prop_2: 22,
  meth_2: function () {
    document.write('meth_2')
  },
}

obj.prop_2 = 44
```

```
> dir(obj)
▼ Object i
  ▶ meth_1: f ()
  prop_1: 7
  prop_2: 44
  ▼ [[Prototype]]: Object
    ▶ meth_2: f ()
    prop_2: 22
```

# Створення об'єктів на основі прототипів

З використанням властивості `__proto__`  
(НЕ РЕКОМЕНДУЄТЬСЯ!!!)  
Безпосередньо записуємо у властивість `__proto__`

```
obj.__proto__ = obj_parent;  
або  
Object.setPrototypeOf(obj, obj_parent)
```

```
===== Варіант 1 =====  
//--- Створюємо об'єкт  
let obj = {  
  prop_1: 11,  
  meth_1: function () {  
    document.write("meth_1")  
  }  
}  
  
//----- Додаємо предка у __proto__  
obj.__proto__ =  
  {  
    prop_2: 22,  
    meth_2: function () {  
      document.write("meth_2");  
    }  
  };  
  
===== Варіант 2 =====  
//--- Створюємо об'єкт  
let obj = {  
  prop_1: 11,  
  meth_1: function () {  
    document.write("meth_1");  
  }  
};  
  
//----- Об'єкт предок (прототип)  
let obj_parent = {  
  prop_2: 22,  
  meth_2: function () {  
    document.write("meth_2");  
  }  
};  
  
//----- Додаємо об'єкт предок (прототип)  
  
Object.setPrototypeOf(obj, obj_parent)
```

З використанням `Object.create`

- **Object.create** ( `прототип` ) – створює новий об'єкт з вказаним прототипом (посилання міститься у `__proto__`);
- **Object.create**(`прототип`, `descriptors`) – створює новий об'єкт з вказаним прототипом і додає нові властивості, описані за допомогою дескриптора

```
===== Варіант 1 =====  
//----- Створюємо предка  
let obj_parent = {  
  prop_2: 22,  
  meth_2: function () {  
    document.write("meth_2");  
  }  
};  
  
//----- Створюємо об'єкт на основі прототипу  
let obj = Object.create(obj_parent);  
  
//----- Додаємо нові властивості  
obj.prop_1 = 11;  
obj.meth_1 = function () {  
  document.write("meth_1");  
};  
  
===== Варіант 2 =====  
//----- Створюємо предка  
obj_parent = {  
  prop_2: 22,  
  meth_2: function () {  
    document.write("meth_2");  
  }  
};  
  
//----- Створюємо об'єкт на основі прототипу,  
//-- за допомогою дескрипторів додаємо властивості  
let obj = Object.create(obj_parent, {  
  prop_1: {  
    value: 11,  
    enumerable: true  
  },  
  meth_1: {  
    value: function () {  
      document.write("meth_1");  
    },  
    enumerable: true  
  }  
});
```

```
▼ Object ⓘ  
  ▼ [[Prototype]]: Object  
    ► meth_2: f ()  
    ► prop_2: 22  
  ► [[Prototype]]: Object
```

## Методи для аналізу властивості/зміни властивості `__proto__`

<i><b>Object.setPrototypeOf( obj, proto )</b></i>	Встановлює <code>obj.__proto__ = proto</code>	<pre> let obj_parent = {   . . . . . }; let obj_child={   . . . . . } <b>Object.setPrototypeOf</b>(obj_child, obj_parent) //аналог obj_child.<b>__proto__</b> = obj_parent; </pre>
<i><b>Object.getPrototypeOf( obj )</b></i>	Повертає <code>obj.__proto__</code>	<pre> let obj_parent = {   . . . . . }; let obj_child={   . . . . . } <b>Object.setPrototypeOf</b>(obj_child, obj_parent)  <b>Object.getPrototypeOf</b>(obj_child)  // obj_parent </pre>
<i><b>obj1.isPrototypeOf( obj2 )</b></i>	дозволяє визначити, чи є obj1 у ланцюзі прототипів obj2	<pre> let obj_parent = {   . . . . . }; let obj_child={   . . . . . } obj_child.<b>__proto__</b> = obj_parent; document.write(obj_parent.<b>isPrototypeOf</b>(obj_child)); //true </pre>

# Деякі методи для аналізу та маніпуляцій з властивостями об'єкта

<div>obj.hasOwnProperty(prop)</div> <div>повертає true, якщо властивість prop належить безпосередньо самому об'єкту obj (а якомусь його нащадку), інакше false</div>	<pre>let obj_parent = {   prop_2: 22,   . . . . . }; let obj_child={   prop_1: 11,   . . . . . } obj_child. <i>hasOwnProperty</i> ('prop_1') //true obj_child. <i>hasOwnProperty</i> ('prop_2') //false</pre>
<div>Object.<i>getOwnPropertyNames</i>(obj)</div> <div>Дозволяє отримати масив імен всіх властивостей (навіть тих, що не є ітерованими)</div>	<pre>let obj = {   a: 1,   b: 2,   internal: 3 };  Object.defineProperty(obj, "internal", {   enumerable: false });  alert( Object.<i>getOwnPropertyNames</i>(obj) ); // a, internal, b</pre>
<div>Object.<i>getOwnPropertyDescriptor</i>(obj, prop)</div> <div>Дозвляє отримати об'єкт параметрів власності і за потреби змінити його</div>	<pre>let obj = {   test: 5 }; let descriptor = Object.<i>getOwnPropertyDescriptor</i>(obj, 'test'); //отримуємо об'єкт параметрів  // змінюємо параметри власності delete descriptor.value; delete descriptor.writable; descriptor.get = function() {   alert( "Test message" ); };  // заміняємо властивість (видаляємо попередню і додаємо нову)  delete obj.test; //видаляємо попередню властивість  Object.defineProperty(obj, 'test', descriptor); //додаємо змінену obj.test; // Test message</pre>

Загальна форма	<p style="text-align: right;"><u><code>Object.defineProperty(obj, prop, descriptor)</code></u></p> <p>Аргументи:</p> <p><b>obj</b> - об'єкт, для якого створюється властивість.</p> <p><b>Prop</b> - ім'я властивості, яка оголошується або модифікується.</p> <p><b>Descriptor</b> -об'єкт-дискриптор, який містить властивості об'єкта</p> <p>Він може містити такі поля:</p> <ul style="list-style-type: none"> <li><code>value</code> – значення властивості, за замовчуванням <code>undefined</code></li> <li><code>writable</code> – значення властивості можна змінювати якщо <b>true</b>, За замовчуванням <b>false</b></li> <li><code>configurable</code> – якщо <code>true</code>, то властивість можна видаляти а також змінювати з іншими викликами <code>defineProperty</code>.</li> <li><code>enumerable</code> – якщо <code>true</code>, то властивість можна переглядати у циклі <code>for..in</code> і методі <code>Object.keys()</code></li> <li><code>get</code> – функція-геттер</li> <li><code>set</code> – функція-сеттер</li> </ul> <p style="text-align: right;"><a href="https://uk.javascript.info/property-descriptors">https://uk.javascript.info/property-descriptors</a></p>
Приклад	<pre>let user = {};  Object.defineProperty(   user,   "name",   {     value: "Ivan",     writable: false,    // заборонити зміну"user.name="     configurable: false // заборонити видалення"delete user.name"   } )</pre> <div style="border: 1px solid black; padding: 5px; text-align: right;"> <a href="https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty">https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty</a> </div>

Загальна форма	<div>Object. <u>defineProperties</u> (<u>obj</u>, <u>props</u>)</div> <p>Аргументи:</p> <p><b>Obj</b> - об'єкт, для якого створюється властивість.</p> <p><b>props</b> – об'єкт, який містить опис властивостей, які необхідно описати</p> <p>Object. <b>defineProperties</b> (<b>obj</b>, {     <u>'назва_властивості_1'</u> : <b>дискриптор_властивості_1</b>,     <u>'назва_властивості_2'</u> : <b>дискриптор_властивості_2</b>,     . . . . . })</p> <p><b>дискриптор_властивості</b> -об'єкт-дискриптор, який містить властивості об'єкта</p> <p>Він може містити такі поля:</p> <ul style="list-style-type: none"><li>• <b>value</b> – значення властивості, за замовчуванням <b>undefined</b></li><li>• <b>writable</b> – значення властивості можна змінювати якщо <b>true</b>, за замовчуванням <b>false</b></li><li>• <b>configurable</b> – якщо <b>true</b>, то властивість можна видаляти а також змінювати з іншими викликами <b>defineProperty</b>.</li><li>• <b>enumerable</b> – якщо <b>true</b>, то властивість можна переглядати у циклі <b>for...in</b> і методі <b>Object.keys()</b></li><li>• <b>get</b> – функція-геттер</li><li>• <b>set</b> – функція-сеттер</li></ul>
Приклад	<pre>let user = {}  Object.defineProperties(user, {   firstName: {     value: "Ivan"   },   surname: {     value: "Sirko"   },   fullName: {     get: function() {       return this.firstName + ' ' + this.surname;     }   } })  alert( user.fullName ); // Ivan Sirko</pre> <div><a href="https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties">https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties</a></div>

## Деякі методи Object для маніпуляцій з об'єктами

Метод	Призначення
<code>Object.assign(target, ...sources )</code>	Копіює значення з списку ресурсних об'єктів <b><i>sources</i></b> у цільовий об'єкт <b><i>target</i></b>
<code>Object.preventExtensions(obj)</code>	Забороляє додавання властивостей в об'єкт
<code>Object.isExtensible(obj)</code>	Повертає <code>false</code> , якщо додавання властивостей було заборонено з використанням методу <code>Object.preventExtensions</code>
<code>Object.seal(obj)</code>	Забороляє додавання і видалення властивостей. Для всіх властивостей <code>configurable: false</code> .
<code>Object.isSealed(obj)</code>	Повертає <code>true</code> , якщо додавання і видалення властивостей об'єкта заборонено
<code>Object.freeze(obj)</code>	Забороляє додавання, видалення і зміну властивостей <code>configurable: false, writable: false</code> .
<code>Object.isFrozen(obj)</code>	Повертає <code>true</code> , якщо додавання, видалення і зміна властивостей заборонено

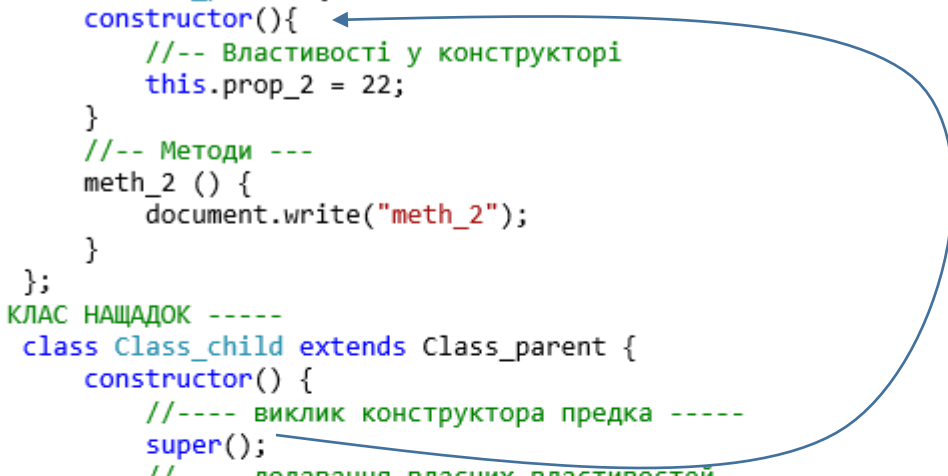
# УСПАДКУВАННЯ У ФУНКЦІОНАЛЬНОМУ СТИЛІ

Загальна форма	<pre>function ім'я_функції-конструктора_нащадка {   //----- Додавання властивостей з предка -----   Конструктор_предка . apply(this, параметри_для_конструктора_предка )   //----- Опис власних властивостей -----   this.властивість1 = значення1;   . . . } //---- Створення прототипу на основі прототипу предка ---- Конструктор_нащадка = Object.create( Конструктор_предка . prototype ); //--- Встановлення властивості constructor Конструктор_нащадка .prototype.constructor = ім'я_функції_конструктора ; //--- Додавання власних методів функція-конструктор_нащадка.prototype.метод_1 = function(форм.пар.) {   . . . }</pre>
Приклад	<pre>//---- КЛАС ПРЕДОК ----- function Class_parent() {   //-- Властивості у конструкторі   this.prop_2 = 22; }; //--- Методи у прототипі Class_parent.prototype.meth_2= function () {   document.write("meth_2"); }  //---- КЛАС НАЩАДОК ----- function Class_child() {   //--- Додаємо властивості предка у об'єкт нащадка (якщо треба)   Class_parent.call(this); //або ж   Obj_parent.apply(this);   //this={prop_2 : 22}    //--- Додаємо власні властивості дочірнього класу ---   this.prop_1 = 11;           //this={prop_2 : 22, prop_1 : 11}  } //--- Створюємо прототип на основі прототипу предка (якщо треба) Class_child.prototype = Object.create(Class_parent.prototype); //--- Встановлюємо конструктор Class_child.prototype.constructor = Class_child; //--- Додаємо власні методи дочірнього класу (якщо треба) ---- Class_child.prototype.meth_1 = function () {   document.write("meth_1"); }  let obj = new Class_child(); document.write(obj instanceof Class_parent); //true document.write(obj instanceof Class_child);  //true document.write(Class_child.prototype.isPrototypeOf(obj)); //true</pre>

```
> dir(obj)
```

```
▼ Obj_child ⓘ
  prop_1: 11
  prop_2: 22
  ▼ __proto__: Obj_parent
    ► constructor: f Obj_child()
    ► meth_1: f ()
    ▼ __proto__:
      ► meth_2: f ()
      ► constructor: f Obj_parent()
      ► __proto__: Object
```



Загальна форма	Приклад
<pre> class клас_нащадок extends клас_предок {   constructor(параметри) {     //----- виклик конструктора предка -----     super(...arguments); //Передаємо параметри у конструктор     //----- додавання власних властивостей -----     this. Властивість1 = значення1 ;     . . . . .   }   //----- додавання власних методів -----   meth_1 () {     . . . . .   }   . . . . . } </pre> <ul style="list-style-type: none"> <li>нащадок успадковує властивості і методи предка</li> <li>конструктор також успадковується (якщо у нащадка немає конструктора, то використовується конструктор предка)</li> </ul>	<pre> //----- КЛАС ПРЕДОК ----- class Class_parent {   constructor(){     //-- Властивості у конструкторі     this.prop_2 = 22;   }   //-- Методи ---   meth_2 () {     document.write("meth_2");   } };  //----- КЛАС НАЩАДОК ----- class Class_child extends Class_parent {   constructor() {     //----- виклик конструктора предка -----     super();     //----- додавання власних властивостей -----     this.prop_1 = 11;   }    //----- додавання власних методів -----   meth_1 () {     document.write("meth_1");   } }  let obj = new Class_child(); &gt; dir(obj) ▼ Class_child ⓘ   prop_1: 11   prop_2: 22   __proto__: Class_parent     ▶ constructor: class Class_child     ▶ meth_1: f meth_1()     __proto__:       ▶ constructor: class Class_parent       ▶ meth_2: f meth_2()       ▶ __proto__: Object </pre> 

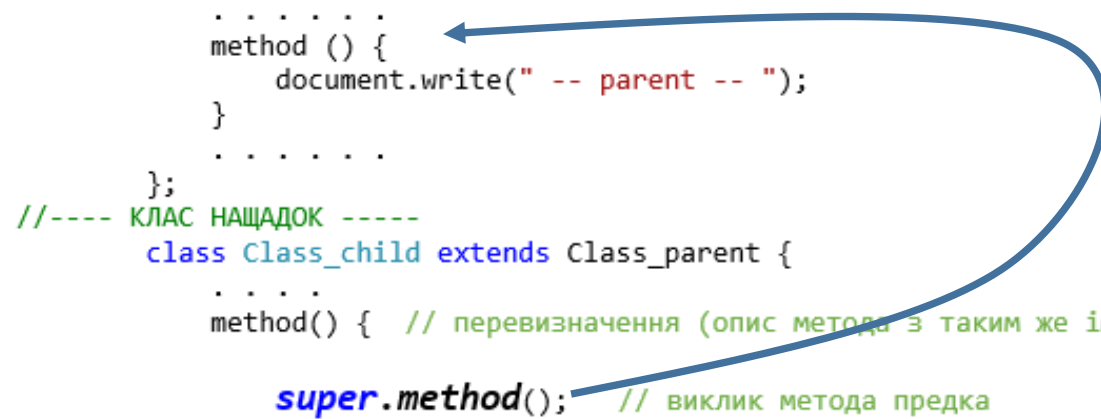
Задача 0. Створити клас Range (діапазон)

Властивості	minValue maxValue
Методи	inRange(value) – метод визначення того, чи є вказане значення у заданому діапазоні ToString

На основі класу Range створити клас PensionerChecker

Властивості	minValue – мінімальний вік пенсіонера maxValue – максимальний вік
Методи	isPensioner(age) – метод визначення того, чи є пенсіонером render(containerID) – метод виведення розмітки <div><input type="text"/> <input type="button" value="Чи пенсіонер"/></div> ToString

Перевизначення методів – опис у нащадків методів з таким же іменем як і у предка

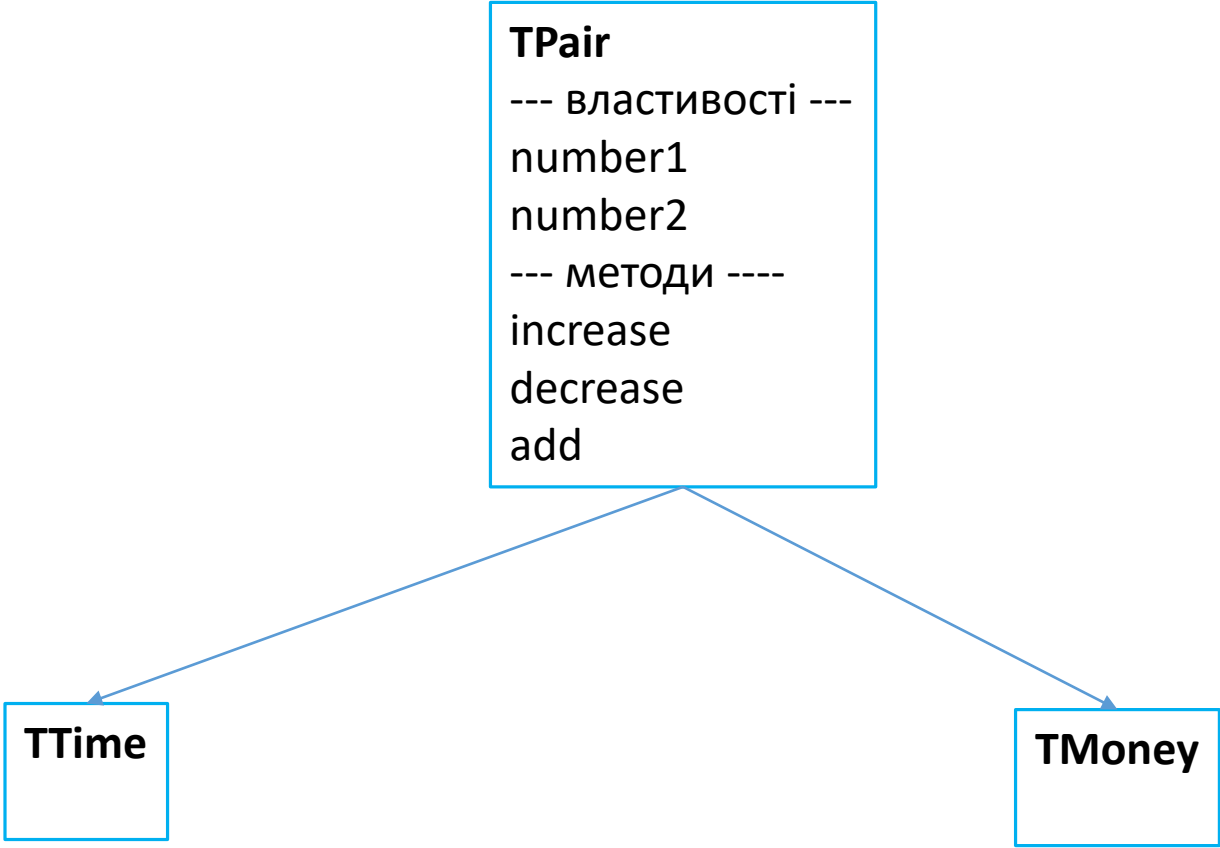
Загальна форма.	
<pre>//----- КЛАС НАЩАДОК ----- class Class_child extends Class_parent {     . . . . .     Метод нащадка () {         super. Метод предка (форм.парам.); // виклик метода предка     }     . . . . . }</pre> <p>виклик перевизначеного метода здійснюється через використання ключового слова <b>super</b></p>	<pre>//----- КЛАС ПРЕДОК ----- class Class_parent {     . . . . .     method () {         document.write(" -- parent -- ");     }     . . . . . }; //----- КЛАС НАЩАДОК ----- class Class_child extends Class_parent {     . . . . .     method() { // перевизначення (опис метода з таким же іменем)         super.method(); // виклик метода предка         document.write(" -- child --");     }     . . . . . }  let obj = new Class_child(); obj.method(); //-- parent -- -- child --</pre> 

Задача 1. Створити клас `Dice`, який представляє гральний кубик (одне поле `faceCount` - кількість граней) і дозволяє генерувати випадкове число від 1 до кількості граней. Потім на основі цього класу створити клас `Roller`, який дозволяє вказати не тільки кількість граней, а й кількість спроб - `attemptCount` (при цьому як випадкове число повертається середнє значення).

Задача . Реалізувати класи TTime (“години.хвилини”) та TMoney (“гривні.копійки”), які містять методи збільшення/зменшення величин на 1 та додавання двох величин. Згенерувати поступово випадковим чином пар (час, гроші), де час – тривалість виконання роботи, а гроші – вартість однієї хвилини роботи працівників. Обчислити витрати на виконання кожної із робіт.

Задача . Реалізувати класи TTime (“години.хвилини”) та TMoney (“гривні.копійки”), які містять методи збільшення/зменшення величин на 1 та додавання двох величин. Згенерувати поступово випадковим чином пар (час, гроші), де час – тривалість виконання роботи, а гроші – вартість однієї хвилини роботи працівників. Обчислити витрати на виконання кожної із робіт.

Як бачимо, у обох класах є два поля, для кожного з яких можна задати мінімальне та мксимальне значення.. Тому, щоб не дублювати код можна описати спільного предка - клас TPair, який представляє пару чисел і містить методи для їх збільшення/зменшення на 1 та додавання двох величин.



## Задача. Розробити каси

### WorkersList

--- властивості ---

- масив з віком працівників

--- методи ---

- знаходження найстаршого
- знаходження наймолодшого
- знаходження середнього віку
- виведення списку у формі нумерованого списку при цьому ті, що більші за середнє значення відмічаються червоним кольором
- введення з використанням списку input
- знаходження працівників пенсійного віку

### PriceList

--- властивості ---

- масив з цінами товарів

--- методи ---

- знаходження найбільшої ціни
- знаходження найменшої ціни
- знаходження середньої ціни
- виведення цін у формі нумерованого списку при цьому ті, що більші за середнє значення відмічаються червоним кольором
- введення з використанням елементів div
- знаходження загальної вартості товарів

Бачимо, що ці класи мають спількі властивості і методи

## Задача. Розробити каси

Спільні властивості і методи винесемо у спільний клас предок **DataList**

### **DataList**

--- властивості ---

- масив з числами

--- методи ---

- знаходження найбільшого числа
- знаходження найменшого числа
- знаходження середнього числа
- виведення чесел у формі нумерованого списку  
при цьому ті, що більші за середнє значення  
відмічаються червоним кольором
- введення даних з використанням елементів(селектор, відповідні властивості  
елемента вводу задаються)

### **WorkersList**

--- методи ---

- знаходження найстаршого
- знаходження наймолодшого
- знаходження середнього віку
- введення з використанням списку input
- знаходження працівників пенсійного віку

### **PriceList**

--- методи ---

- знаходження найбільшої ціни
- знаходження найменшої ціни
- знаходження середньої ціни
- введення з використанням елементів div
- знаходежння загальної вартості товарів