

# Exercise: Object Composition

## 1. Array Extension

Extend the build-in **Array** object with additional functionality. Implement the following functionality:

- **last()** - returns the last element of the array
- **skip(n)** - returns a new array which includes all original elements, except the first **n** elements; **n** is a **Number** parameter
- **take(n)** - returns a new array containing the first **n** elements from the original array; **n** is a **Number** parameter
- **sum()** - returns a sum of all array elements
- **average()** - returns the average of all array elements

## Input / Output

Input for functions that expect it will be passed as valid parameters. Output from functions should be their **return** value.

## Constraints

Structure your code as an **IIFE**.

## Hints

If we have an **instance** of an array, since we know it's an object, adding new properties to it is pretty straightforward:

```
let myArr = [1, 2, 3];  
  
myArr.last = function () {  
    // TODO  
};
```

This however, only adds our new function to this instance. To add all functions just one time and have them work on **all arrays** is not much more complicated, we just have to attach them to Array's **prototype** instead:

```
Array.prototype.last = function () {  
    // TODO  
};
```

With such a declaration, we gain access to the context of the calling instance via **this**. We can then easily access indexes and other existing properties. Don't forget we don't want to modify the existing array, but to create a new one:

```

Array.prototype.last = () => {
  return this[this.length - 1];
};

Array.prototype.skip = n => {
  let result = [];
  for (let i = n; i < this.length; i++) {
    result.push(this[i]);
  }

  return result;
};

Array.prototype.take = n => {
  let result = [];
  for (let i = 0; i < n; i++) {
    result.push(this[i]);
  }

  return result;
};

```

Note these functions do not have any error checking - if **n** is **negative** or **outside the bounds** of the array, and exception will be thrown, so take care when using them, or add your own validation. The last two functions require a little bit of arithmetic to be performed:

```

Array.prototype.sum = () => {
  let sum = 0;
  for (let i = 0; i < this.length; i++) {
    sum += this[i];
  }

  return sum;
};

Array.prototype.average = () => {
  return this.sum() / this.length;
};

```

To test our program in the Judge, we need to wrap it in an IIFE, like it's shown on the right. There is **no return value**, since the code execution results in functionality being added to an existing object, so they take effect instantly. We are ready to submit our solution.

```

(function solve() {
  Array.prototype.last = () => {...};

  Array.prototype.skip = n => {...};

  Array.prototype.take = n => {...};

  Array.prototype.sum = () => {...};

  Array.prototype.average = () => {...};
})();

```

## 2. Construction Crew

Write a program that receives a **worker** object as a parameter and modifies its properties. Workers have the following structure:

```
{ weight: Number,  
  experience: Number,  
  levelOfHydrated: Number,  
  dizziness: Boolean }
```

Weight is expressed in **kilograms**, **experience** in **years** and **levelOfHydrated** is in **milliliters**. If you receive a worker whose **dizziness** property is set to **true** it means he needs to intake some **water** in order to be able to work correctly. The required amount is 0.1ml per **kilogram** per year of **experience**. The required amount must be **added** to the **existing amount**. Once the water is administered, change the **dizziness** property to **false**.

Workers who **do not have dizziness** should **not** be modified in any way. Return them as they were.

### Input

Your function will receive a valid **object** as **parameter**.

### Output

Return the **same object** that was passed in, **modified** as necessary.

### Examples

Input	Output
<pre>{ weight: 80,   experience: 1,   levelOfHydrated: 0,   dizziness: true }</pre>	<pre>{ weight: 80,   experience: 1,   levelOfHydrated: 8,   dizziness: false }</pre>
<pre>{ weight: 120,   experience: 20,   levelOfHydrated: 200,   dizziness: true }</pre>	<pre>{ weight: 120,   experience: 20,   levelOfHydrated: 440,   dizziness: false }</pre>
<pre>{ weight: 95,   experience: 3,   levelOfHydrated: 0,   dizziness: false }</pre>	<pre>{ weight: 95,   experience: 3,   levelOfHydrated: 0,   dizziness: false }</pre>

### 3. Car Factory

Write a program that assembles a car by **given requirements** out of **existing components**. The client will place an order in the form of an **object describing** the car. You need to **determine** which parts to use to fulfil the client's order. You have the following parts in storage:

An **engine** has **power** (given in horsepower) and **volume** (given in cubic centimeters). Both of these values are **numbers**. When selecting an engine, pick the **smallest possible** that still meets the requirements.

Small engine: { power: 90, volume: 1800 }

Normal engine: { power: 120, volume: 2400 }

Monster engine: { power: 200, volume: 3500 }

A **carriage** has a **type** and **color**. Both of these values are **strings**. You have two types of carriages in storage and can paint it **any color**.

Hatchback: { type: 'hatchback', color: <as required> }

Coupe: { type: 'coupe', color: <as required> }

The **wheels** will be represented by an **array of 4 numbers**, each number represents the **diameter** of the wheel in inches. The size can only be an **odd number**. Round **down** any requirements you receive to the nearest odd number.

#### Input

You will receive an **object** as an **argument** to your function. The format will be as follows:

```
{ model: <model name>,  
  power: <minimum power>,  
  color: <color>,  
  carriage: <carriage type>,  
  wheelsize: <size> }
```

#### Output

Return the resulting car **object** as a result of your function. See the examples for details.

#### Examples

Sample input	Output
<pre>{ model: 'VW Golf II',   power: 90,   color: 'blue',   carriage: 'hatchback',   wheelsize: 14 }</pre>	<pre>{ model: 'VW Golf II',   engine: { power: 90,             volume: 1800 },   carriage: { type: 'hatchback',              color: 'blue' },   wheels: [13, 13, 13, 13] }</pre>
<pre>{ model: 'Opel Vectra',   power: 110,</pre>	<pre>{ model: 'Opel Vectra',   engine: { power: 120,</pre>

<pre>color: 'grey', carriage: 'coupe', wheelsize: 17 }</pre>	<pre>volume: 2400 }, carriage: { type: 'coupe',            color: 'grey' }, wheels: [17, 17, 17, 17] }</pre>
--	--

## 4. Extensible Object

Create an object that can clone the functionality of another object into itself. Implement an **extend(template)** function that would copy all of the properties of template to the parent object and if the property is a function, add it to the object's prototype instead.

### Input / Output

Your code should **return** the extensible **object instance**. The **extend()** function of your object will receive a valid object as **input parameter**, and has **no** output.

### Examples

Extensible object	Resulting object
<pre>myObj: {   __proto__: {}   extend: function () {...} }</pre>	<pre>myObj: {   __proto__: {     extensionMethod: function () {...}   },   extend: function () {...},   extensionProperty: 'someString' }</pre>
Template object	
<pre>template: {   extensionMethod: function () {...},   extensionProperty: 'someString' }</pre>	

Note that **\_\_proto\_\_** is a hidden property, representing the object's **prototype** - depending on your test environment, you may not have access to it directly, but you can use other functions to do that.

### Hints

To gain access to the prototype of an instance, use the **Object.getPrototypeOf()** function. To make a function shared between all instances, it'll have to be attached to the prototype instead of the instance.

## 5. String Extension

Extend the build-in String object with additional functionality. Implement the following functions:

- **ensureStart(str)** - append **str** to the beginning of a string, **only** if it's **not** already present

- **ensureEnd(str)** - append **str** to the end of a string, **only** if it's **not** already present
- **isEmpty()** - return **true** if the string is **empty**, **false** otherwise
- **truncate(n)** - truncates the string to **n** characters by **removing words** and appends an ellipsis (three periods) to the end. If a string is less than **n** characters long, return the **same string**. If it is longer, split the string where a **space** occurs and append an ellipsis to it so that the **total length** is less than or equal to **n**. If **no space** occurs anywhere in the string, return **n - 3** characters and an ellipsis. If **n** is less than 4, return **n** number of periods.
- **format(string, ...params)** - static method to replace placeholders with parameters. A placeholder is a number surrounded by curly braces. If parameter index cannot be found for a certain placeholder, do not modify it. Note static methods are attached to the **String object** instead of its prototype. See the examples for more info.

Note strings are **immutable**, so your functions will return new strings as a result.

## Input / Output

Your main code should be structured as an IIFE **without** input or output - it should modify the existing **String prototype** instead.

Input and output of the **extension functions** should be as described above.

## Examples

Sample input	Value of str
let str = 'my string';	
str = str.ensureStart('my');	'my string' // 'my' already present
str = str.ensureStart('hello ');	'hello my string'
str = str.truncate(16);	'hello my string' // length is 15
str = str.truncate(14);	'hello my...' // length is 11
str = str.truncate(8);	'hello...'
str = str.truncate(4);	'h...'
str = str.truncate(2);	'..'
str = String.format('The {0} {1} fox', 'quick', 'brown');	'The quick brown fox'
str = String.format('jumps {0} {1}', 'dog');	'jumps dog {1}' // no parameter at 1