#### **Arrays**

- Arrays are a data structure in JavaScript used to store multiple values.
- Arrays are ordered and can be accessed by their index.
- Arrays are resizable and can contain a mix of different data types. (numbers, strings, objects, etc.)

#### **Creating an Array**

Using an array literal is the easiest way to create a JavaScript Array.

```
const array_name = [ item1 , item2 , ...];
or
const array_name = new Array(item1, item2, ...)
```

The two examples above do exactly the same.

There is no need to use new Array().

For simplicity, readability and execution speed, use the array literal method.

#### **Accessing Array Elements**

You access an array element by their index:

```
const fruits = ["banana", "apple", "peach"]
console.log(fruits[0]);
```

#### **Changing an Array Element**

```
fruits[0] = "orange"
```

# **Array Properties and Methods**

• arr.push(...items) - adds items to the end,

• arr.shift() - extracts an item from the beginning,

• arr.unshift(...items) - adds items to the beginning.

• arr.pop() - extracts an item from the end,

console.log(arr); // [1, 'a', 'b', 3]

## **Array: length**

The length property of an array returns the length of an array (the number of array elements).

## **Methods:**

```
Add/remove items
```

```
• Array elements can be deleted using the JavaScript operator delete. Using delete leaves undefined holes in
   the array.
<u>splice</u>
It can do everything: insert, remove and replace elements.
splice(start)
splice(start, deleteCount)
splice(start, deleteCount, item1)
splice(start, deleteCount, item1, item2)
splice(start, deleteCount, item1, item2, /* ..., */ itemN)
Let's start with the deletion:
console.log(arr); //[1,3]
adding a new item:
replacing a item:
```

# array with some elements removed and/or replaced at a given index.

to<u>Splice</u>d

slice The **slice()** method of <u>Array</u> instances returns a <u>shallow copy</u> of a portion of an array into a new array object

The toSpliced() method of Array instances is the copying version of the splice() method. It returns a new

### selected from start to end (end not included) where start and end represent the index of items in that array.

The original array will not be modified. slice() slice(start)

```
slice(start, end)
Examples:
const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];
console.log(animals.slice(2)); // ["camel", "duck", "elephant"]
console.log(animals.slice(2, 4)); // ["camel", "duck"]
console.log(animals.slice(1, 5)); // ["bison", "camel", "duck", "elephant"]
console.log(animals.slice(-2)); // ["duck", "elephant"]
console.log(animals.slice()); // ["ant", "bison", "camel", "duck", "elephant"]
<u>concat</u>
The concat() method of Array instances is used to merge two or more arrays. This method does not change the
```

#### Example: let arr = [1, 2];

});

alert( arr.concat([3, 4]) ); // 1,2,3,4

```
Iterate: for Each
The forEach() method of <u>Array</u> instances executes a provided function once for each array element.
arr.forEach(function(item, index, array) {
```

// ... do something with item

existing arrays, but instead returns a new array.

arr.concat(arg1, arg2...)

Searching in array

```
indexOf/lastIndexOf and includes
<u>indexOf</u>
The indexOf() method of <u>Array</u> instances returns the first index at which a given element can be found in the
```

# arr.indexOf(searchElement, fromIndex)

array, or -1 if it is not present.

# <u>includes</u>

The includes() method of Array instances determines whether an array includes a certain value among its

arr.includes(searchElement, fromIndex)

The lastIndexOf() method of Array instances returns the last index at which a given element can be found in

```
<u>lastIndexOf</u>
```

entries, returning true or false as appropriate.

### the array, or -1 if it is not present. lastIndexOf(searchElement, fromIndex)

some/every The **some()** method of <u>Array</u> instances tests whether at least one element in the array passes the test implemented by the provided function.

The every() method of Array instances tests whether all elements in the array pass the test implemented by the

The **find()** method of <u>Array</u> instances returns the first element in the provided array that satisfies the provided

```
find and findIndex/findLastIndex
find
```

// for falsy scenario returns undefined

let user = users.find(item => <u>item.id</u> === 1);

{id: 1, name: "Akbar"},

console.log( <u>user.name</u> ); // Akbar

provided function. It returns a Boolean value.

# testing function. If no values satisfy the testing function, <u>undefined</u> is returned.

**findIndex** 

Example:

// if true is returned, item is returned and iteration is stopped

```
The findIndex() method of <u>Array</u> instances returns the index of the first element in an array that satisfies the
provided testing function. If no elements satisfy the testing function, -1 is returned.
console.log( users.findIndex(item => item.name === "Akbar") ); // 0
<u>findLastIndex</u>
let users = [
```

The find method looks for a single (first) element that makes the function return true.

The syntax is similar to find, but filter returns an array of all matching elements:

# **Transform an array** <u>map</u>

sort(fn)

for number:

for **strings**:

<u>reverse</u>

<u>filter</u>

on every element in the calling array. let result = arr.map (function(item, index, array) { // returns the new value instead of item

The sort() method of Array instances sorts the elements of an array in place and returns the reference to the

same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then

The elements order in the array will be turned towards the direction opposite to that previously stated.

The map() method of Array instances creates a new array populated with the results of calling a provided function

```
arr.sort( (a, b) => a.localeCompare(b) )
toSorted
The toSorted() method of Array instances is the copying version of the sort() method. It returns a new array
```

with the elements sorted in ascending order.

new array with the elements in reversed order.

returned without using the separator.

comparing their sequences of UTF-16 code units values.

If there may be many, we can use <u>arr.filter(fn)</u>.

```
<u>join</u>
The join() method of <u>Array</u> instances creates and returns a new string by concatenating all of the elements in this
```

reduce/reduceRight

}, [initialValue]);

**toReversed** 

let value = arr.reduce(function(accumulator, currentValue, currentIndex, array) {

array, separated by commas or a specified separator string. If the array has only one item, then that item will be

The toReversed() method of Array instances is the copying counterpart of the reverse() method. It returns a

```
provided).
```

• accumulator - is the result of the previous function call, equals initial the first time (if initial is

```
console.log(array.reduce((accumulator, currentValue) =>
accumulator.concat(currentValue)));//[0, 1, 2, 3, 4, 5]
console.log(array.reduceRight((accumulator, currentValue) =>
accumulator.concat(currentValue))); //[4, 5, 2, 3, 0, 1]
```

```
Array.isArray() - The Array.isArray() static method determines whether the passed value is an <a href="Array">Array</a>.
<u>arr.fill(value, start, end)</u> - fills the array with repeating value from index start to end.
arr.flat(depth) / arr.flatMap(fn) create a new flat array from a multidimensional array.
```

#### **Objects**

What are Objects in JavaScript?

- Objects are a fundamental data type in JavaScript.
  - They are versatile and commonly used for organizing and manipulating data.
- Objects consist of key-value pairs.
  - Keys (also known as properties) are strings or symbols.
  - Values can be of any data type: strings, numbers, booleans, functions, other objects, and more.
- Objects can represent real-world entities and their properties.
  - For example, you can model a "person" with properties like name, age, and address.
- JavaScript uses objects extensively, including built-in objects like Date, Math, and Array.
- Objects are like containers that hold related data.
  - Each key-value pair inside an object is referred to as a "property."
- The key (property name) is a string or symbol.
  - It serves as a unique identifier for the value associated with it.
- The value can be of any data type.
  - You can have strings, numbers, booleans, arrays, functions, or even nested objects as values.
- Objects can be nested within other objects.
  - This enables you to represent complex data structures.

#### Example:

```
let person = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "Exampleville",
    postalCode: "12345"
  },
  hobbies: ["Reading", "Hiking", "Cooking"]
};
```

Objects are used extensively in JavaScript for various purposes:

- 1. **Data Structures**: Objects are excellent for organizing and storing structured data, such as user profiles, product details, and configuration settings.
- 2. **Object-Oriented Programming**: JavaScript uses objects to implement object-oriented programming concepts. You can define object constructors and create instances of objects.
- 3. **Associative Arrays**: Objects are often used as associative arrays, where you can access values using keys (property names).
- 4. **JSON (JavaScript Object Notation)**: JSON is a data interchange format that closely resembles JavaScript objects. Objects in JavaScript are easily converted to JSON and vice versa, making them a common choice for data exchange between servers and clients.
- 5. **DOM (Document Object Model)**: In web development, the DOM is represented as a hierarchical tree of objects, where HTML elements are represented as objects with properties and methods.

#### 1. Object Literal Syntax

- The most common and straightforward way to create an object.
- Uses curly braces {} to define the object and its properties.

#### Example:

```
age: 30,
address: "123 Main St"
```

#### • Constructor functions are used to create objects with the new keyword.

2. Constructor Functions

- They act as templates for creating objects of a specific type.
- Example:

```
this.name = name;
this.age = age;
```

# Creates a new object with the specified prototype object.

3. Object.create() Method

- Useful for creating objects with a specific prototype.
- Example:

#### let person = Object.create(null); person.name = "Fagan";

```
person.age = 30;
4. Class Syntax (ES6 and later)
 • Introduced in ECMAScript 6 (ES6) to simplify object creation.
```

## • Provides a more structured way to define object types.

<u>this.name</u> = name;

- Example:

```
this.age = age;
let john = new Person("Fagan", 30);
Choosing the Right Method
• The choice of object creation method depends on your specific needs and coding style preferences.
```

# Object literal syntax is simple and quick for creating one-off objects.

- Constructor functions and classes are suitable for creating multiple instances of the same type of object with shared methods.
- Object.create() is useful for working with prototypes and inheritance.
- Factory functions provide flexibility in creating and initializing objects and are often used in design patterns. • ECMAScript 6 (ES6) introduced the class syntax, which is widely adopted and simplifies object-oriented programming.
- **Property Access Best Practices** When accessing object properties in JavaScript, consider the following best practices:

#### • Use bracket notation when:

Dealing with dynamic property names.

• The property name contains special characters or spaces. • Ensure that the property you're trying to access exists in the object to avoid errors (e.g., using if statements or

• Use dot notation for simple and straightforward property access.

- optional chaining in modern JavaScript).

console.log(person.age?.toString());

Example with optional chaining (ES11 and later):

```
Modifying Object Properties
There are two primary ways to modify object properties:
```

# bracket notation.

Example:

1. Direct Assignment

```
age: 26,
person.age = 24;
```

• You can change the value of an existing property by directly assigning a new value using either dot notation or

```
2. Adding New Properties
You can also add new properties to an object by assigning a value to a property that doesn't exist yet. JavaScript will
create the property for you.
Example:
```

## **Property Deletion** To remove a property from an object, you can use the delete operator. Be cautious when using delete, as it can

**Defining Object Methods** 

objects can contain not only properties but also methods. Methods are functions that are associated with an object

and can be used to perform actions or computations related to that object. Here's how to define and use object

have unexpected consequences, especially when dealing with inherited properties or object prototypes.

### age: 30, sayHello: function() {

age: 30,

methods:

• In JavaScript, objects are linked to a prototype. A prototype is another object from which the current object inherits

• When you access a property or method on an object, JavaScript first checks if the object has that property or

method. If not, it looks up the prototype chain until it finds the property or until it reaches the end of the chain.

• You can access an object's prototype using the \_\_proto\_\_ property, but it's not recommended for most use

The recommended way to work with prototypes is to use the Object.create() method, which allows you to

```
console.log(`${greeting}, my name is ${ this.name }.`);
Inside the method, you can access object properties using this.
Object Iteration with:
for...in Loop
```

console.log(`Hello, my name is \${ this.name }.`);

ES6 introduced a more concise method definition syntax using arrow functions.

**Object Prototypes** 

Object.keys(), Object.values(), and Object.entries() Methods(ES5+)

# • The prototype chain allows objects to inherit properties and methods from other objects, forming a hierarchy. 2. \_\_proto\_\_ Property (Not Recommended)

let student = {

grade: "A",

properties and methods.

cases. It's considered a legacy feature.

1. Prototype Basics

forEach() Method (ES5+)

Example: let person = { name: "Test",

```
explicitly specify the prototype of an object.
Object.create() creates a new object with the specified prototype.
Example:
```

properties and methods.

3. Object.create() Method

```
let student = Object.create(person);
```

student.\_\_proto\_\_ = person; // Set 'person' as the prototype of 'student'

console.log(student.name); // Output: "Test" (inherited from 'person')

4. Constructor Functions and Prototypes

Properties and methods added to the prototype are shared among all instances created with the constructor.

• Constructor functions and classes (in ES6) are commonly used in JavaScript for creating objects with shared

• Objects created using constructor functions have a prototype property (prototype) that references an object.

console.log( student.name ); // Output: "Test" (inherited from 'person')

```
Person.prototype.sayHello = function() {
 console.log(`Hello, my name is ${this.name}.`);
let veli = new Person("Veli");
```

let eli = new Person("Eli");

```
function Person(name) {
<u>this.name</u> = name;
```

```
veli.sayHello(); // Output: "Hello, my name is Veli."
eli.sayHello(); // Output: "Hello, my name is Eli."
```