

---

## OBJECT-ORIENTED JAVASCRIPT (OOJS)

---

- ◆ Definition:

OOJS (Object-Oriented JavaScript) is a programming style where code is organized around objects (data + behavior).

Objects represent real-world things.

Example: A Car object has brand, color (properties) and start() (method).

---

### ① Ways to create "classes" in JS

---

- A) Pre-ES6: Constructor functions + prototypes
  - B) ES6 and later: class keyword (syntactic sugar)
- 

### ② Implementing Class (Prototype method)

---

```
function Car(brand, color) {  
    this.brand = brand;  
    this.color = color;  
}  
  
// Prototype methods (shared by all objects)  
Car.prototype.start = function() {  
    console.log(`${this.brand} started!`);  
};  
  
Car.prototype.showDetails = function() {  
    console.log(`Brand: ${this.brand}, Color: ${this.color}`);  
};  
  
// Creating objects  
const car1 = new Car("Tesla", "Red");  
car1.start();
```

---

### ③ Static methods and properties (pre-ES6)

---

- Static members belong to the class, not to its instances.
- Define them directly on the constructor function.

```
Car.category = "Vehicle";
```

```
Car.compare = function(a, b) {
  return a.brand === b.brand ? "Same brand!" : "Different brands.";
};

console.log(Car.category);
console.log(Car.compare(car1, car2));
```

---

#### 4 Property declaration

---

- Instance properties are declared inside the constructor using `this`.
- Methods are declared on the prototype so they are shared.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

```
Person.prototype.greet = function() {
  console.log(`Hi, I'm ${this.name}, ${this.age} years old.`);
};
```

---

#### 5 ES6 equivalent (for understanding)

---

```
class Car {
  constructor(brand, color) {
    this.brand = brand;
    this.color = color;
  }

  start() { ... }

  static category = "Vehicle";
  static compare(a, b) { ... }
}
```

---

---

## =====

## ES6 CONCEPTS – IMPORT/EXPORT, ASYNC/AWAIT, CLASSES

---

## =====

◆ ① Import and Export (Modules)

---

Purpose: Split JS into multiple files for organization and reusability.

A) Named export

---

File: math.js

```
export const PI = 3.14;  
export function add(a, b) { return a + b; }
```

Import:

```
import { PI, add } from "./math.js";
```

B) Default export

---

File: greet.js

```
export default function greet(name) {  
  console.log(`Hello, ${name}!`);  
}
```

Import:

```
import greet from "./greet.js";
```

You can mix:

```
import greet, { PI } from "./math.js";
```

---

◆ ② Async / Await

---

- `async` marks a function as asynchronous (it returns a Promise).
- `await` pauses execution until the Promise is resolved.

Example:

```
async function fetchData() {  
  try {  
    const res = await fetch("https://api.example.com");  
    const data = await res.json();  
    console.log(data);  
  } catch (err) {  
    console.error(err);  
  }
}
```

---

◆ ③ Classes

---

- Classes are blueprints for creating objects.

Syntax:

```
class Car {  
    constructor(brand, color) {  
        this.brand = brand;  
        this.color = color;  
    }  
  
    start() {  
        console.log(`${this.brand} started!`);  
    }  
  
    static info() {  
        console.log("Static method on class.");  
    }  
}
```

Usage:

```
const car1 = new Car("Tesla", "Red");  
car1.start();  
Car.info(); // static method
```

---

## =====

## STRING, MATH & DATE METHODS – JAVASCRIPT

## =====

### ① STRING METHODS

---

length → number of characters  
toUpperCase(), toLowerCase()  
trim(), trimStart(), trimEnd()  
includes(substring)  
indexOf(), lastIndexOf()  
slice(start, end)  
replace(), replaceAll()  
split(delimiter)  
startsWith(), endsWith()

Use cases:

- Input validation
- Search

- Text formatting
  - Parsing data
- 

## ② MATH OBJECT

---

```
Math.round(x)  
Math.floor(x)  
Math.ceil(x)  
Math.random()  
Math.max(a, b, c)  
Math.min(a, b, c)  
Math.pow(a, b) or a ** b  
Math.sqrt(x)  
Math.abs(x)
```

Math is static → no new keyword

---

## ③ DATE & TIME

---

```
new Date() → current date/time  
new Date("YYYY-MM-DD")
```

Get values:

```
getFullYear(), getMonth(), getDate()  
getDay(), getHours(), getMinutes(), getSeconds()
```

Set values:

```
setFullYear(), setMonth(), setDate()
```

Timestamp:

```
Date.now()
```

Difference:

```
date2 - date1 → milliseconds
```

Formatting:

```
toDateString()  
toTimeString()  
toLocaleDateString()  
toLocaleTimeString()
```

---

# jQuery Notes — Basics

## 1. What is jQuery?

- A JavaScript library for simpler DOM tasks.
- Uses `$()` as main function.
- Makes JavaScript easier.

## 2. Difference from JavaScript:

- Vanilla JS is native language.
- jQuery is library built on top of JS.
- jQuery simplifies tasks like selectors, DOM changes, events, animations, AJAX.

## 3. Selectors:

- `$("#id")` — selects element with id
- `$(".class")` — selects elements with class
- `$("tag")` — selects elements by tag
- `$("tag, .class")` — multiple selections

## 4. HTML Methods:

- `.html()` — get/set HTML content
- `.text()` — get/set text content
- `.val()` — get/set form input value

## 5. CSS Methods:

- `.css()` — get/set CSS property
- `.addClass()` — add class
- `.removeClass()` — remove class

## 6. Usage:

- Wrap code inside `$(document).ready()`  
so document loads before scripts.
- 

# jQuery Events Notes

## 1. Basic Events:

- Events are actions like click, keyup, mouseover.
- Use `.on()` to listen for events.
- Syntax: `$(selector).on("eventName", handler);`

## 2. Common Event Examples:

- click: runs when an element is clicked.
- mouseover: runs when mouse enters element.
- keyup: runs when key is released in input.
- submit: runs when form is submitted.

## 3. Programmatic Event Firing:

- Use `.trigger()` to fire an event from code.
- Syntax: `$(selector).trigger("eventName");`
- Runs handlers as if user triggered the event.

## 4. Custom Events:

- Define your own event names (e.g. "customSaved").
- Attach with `.on("myEvent", handler);`
- Trigger with `.trigger("myEvent", [data]);`
- Data passed becomes parameters in handler.

## 5. Why use custom events?

- Decouple logic from UI triggers.
  - Reuse code across the app.
  - Pass custom data between components.
- 

# jQuery Validator Basics and Custom Validation Notes

## 1. What is jQuery Validator?

- A jQuery plugin that simplifies form validation.
- It validates forms on client-side before submission.
- Includes many built-in rules (required, email, number, etc.).

## 2. Setup and Basic Usage

- Include jQuery and the plugin script in HTML.
- Call `$('#form').validate({ rules: {...}, messages: {...} });` to activate.
- rules: defines validation conditions.
- messages: custom text shown on validation failure.

## 3. Built-in Validation Rules

- required: field must be filled.
- email: must look like email.
- number: numeric only.

- minlength/maxlength: length constraints.
- equalTo: reproduce another field (e.g., for password confirmation).

#### 4. Custom Validation

- Use `$.validator.addMethod(ruleName, function(value, element) { }, message);`
- The function runs and returns true/false.
- Attach this rule in `rules{}` like built-in rules.

#### 5. Custom Messages

- Use `messages: { fieldName: { ruleName: "message" } }`
- You can override messages for both built-in and custom rules.

#### 6. Always use server-side validation in addition to client-side.

---

## jQuery Utility Functions — Notes

### 1. `$.each(collection, callback)`:

- Loops through arrays or objects.
- `callback(indexOrKey, value)`
- Useful to iterate without normal for loops.

### 2. `$.map(arrayOrObject, callback)`:

- Returns a new array based on mapping logic.
- `callback(value, indexOrKey)` returns new value or null/array.

### 3. `$.grep(array, callback, [invert])`:

- Filters array elements that pass a test function.
- If `invert=true`, includes elements where callback returns false.

### 4. `$.merge(firstArray, secondArray)`:

- Combines two arrays into first array.
- Modifies first array; to avoid this, merge into `[]` first.

### 5. `$.extend([deep], target, object1, objectN)`:

- Merges properties of objects into target.
- `deep=true` merges nested objects recursively.

Other useful utilities: `$.inArray()`, `$.makeArray()`, `$.isNumeric()`, `$.noop()`, `$.parseHTML()`, etc.

---

## Regular Expressions (RegEx) in jQuery

### 1. What is RegEx?

- RegEx is a pattern matching tool used in JavaScript to test, search, replace, or validate string patterns. RegEx patterns are native JavaScript and can be used inside jQuery callbacks.

### 2. Creating Regular Expressions

- Two ways:  
`/pattern flags`  
`new RegExp("pattern", "flags")`
- Common flags:  
`g` (global), `i` (case insensitive), `m` (multi-line).

### 3. Using RegEx with jQuery

- RegEx is used with string methods (`.match`, `.replace`, `.test`):  
`regex.test(string)` // returns true/false  
`string.match(regex)` // returns match array
- Inside jQuery handlers, use `.val()` to get input and test it.

### 4. Common Patterns

- Email: `/^[\w\.-%+-]+@[a-zA-Z\.-]+\.[a-zA-Z]{2,}\$/i`
- Alphabet only: `/^[\w\.-%+-]+@[a-zA-Z\.-]+\.[a-zA-Z]{2,}\$/i`
- 10-digit phone: `^\\d{10}\\$`
- These patterns help validate inputs before further action.

### 5. jQuery Selector + RegEx

- For complex selection, you can use JavaScript inside `.filter()` to apply RegEx on attributes or text.