



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

JavaScript Basics for Web Pages

Make web pages interactive with JavaScript

INSIDE THIS BOOK

- Where JavaScript runs
- Variables, strings and operators
- Making decisions and repeating
- Functions, arrays and objects
- Changing the page: the DOM and events
- ...and 3 more chapters, practice questions and a glossary

Beginner

Students who know some HTML and CSS

engineergurukul.com

+91 97116 99511

JavaScript Basics for Web Pages

A friendly first course in JavaScript for students who know some HTML and CSS. Learn variables, loops, functions, arrays and objects, then change a page, react to clicks and build three projects: a form that checks itself, a to-do list that remembers and a quiz.

WHO IT IS FOR	Students who know some HTML and CSS
LEVEL	Beginner
SUBJECT AREA	Programming and web
WHAT IS INSIDE	8 chapters, practice questions with answers, and a glossary

CONTENTS

1	Where JavaScript runs	3
2	Variables, strings and operators	4
3	Making decisions and repeating	7
4	Functions, arrays and objects	9
5	Changing the page: the DOM and events	12
6	Project: a form that checks itself	15
7	Projects: a to-do list and a quiz	17
8	Finding and fixing bugs	22
	Practice questions	24
	Answers	25
	Glossary	26

Free ebook: you may read, print and share it for learning; it may not be sold or changed. © 2026 Gurukul Technicals Institute. First edition, October 2026. Contact details are on the back cover.

Where JavaScript runs

A web page is built from three languages. **HTML** gives the page its structure: headings, paragraphs, buttons. **CSS** decides how it looks: colours, fonts, spacing. **JavaScript** makes it do things: show a message when you click, check a form before it is sent, or count your score in a quiz.

JavaScript is a **programming language**: a language for giving step-by-step instructions to a computer. Every modern web browser, such as Chrome, Edge, Firefox or Safari, has a JavaScript **engine** built in. The engine reads your code and runs it on the visitor's own computer or phone. So you need nothing special to start: a browser and a plain text editor are enough. Many students use Visual Studio Code, but even Notepad works.

DID YOU KNOW

JavaScript has nothing to do with Java. It was first written in 1995 for the Netscape browser, and the similar name was chosen mainly for publicity. Today JavaScript can also run outside the browser, on servers, using a program called Node.js.

The script tag

You add JavaScript to a page with the `<script>` tag. Make a folder called js-practice, create a file called first.html in it, and type this:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My first script</title>
</head>
<body>
  <h1>Hello from HTML</h1>
  <script>
    console.log("Hello from JavaScript");
    document.querySelector("h1").textContent = "Hello from JavaScript!";
  </script>
</body>
</html>
```

Save it and double-click the file to open it in your browser. The heading says "Hello from JavaScript!" because the script changed it.

Notice that the script sits just before `</body>`. The browser reads a page from top to bottom. If the script came first, it would look for the heading before the heading existed.

For anything longer than a few lines, keep your JavaScript in its own file, such as `app.js`, in the same folder, and link it with `<script src="app.js"></script>`. You may also place that line in the `<head>` if you add the word `defer`, which tells the browser to run the script only after the whole page has loaded.

The console

The **console** is a hidden panel in the browser where JavaScript writes messages and errors. Open it now:

1. Press F12, or Ctrl+Shift+I. A panel called the developer tools opens.
2. Click the **Console** tab.
3. You should see "Hello from JavaScript", printed by `console.log()`.

You can also type JavaScript straight into the console and press Enter. Try `2 + 3` or `alert("Namaste")`.

TIP

Keep the console open whenever you write JavaScript. If something does not work, the reason is almost always printed there in red.

CHAPTER 2

Variables, strings and operators

A **variable** is a name that stores a value, like a labelled box. Modern JavaScript makes variables with two words:

- `let` makes a variable whose value can change later.
- `const` makes a variable that must keep the same value. Use it whenever you can.

JS

```
const price = 45;
let notebooks = 3;
let total = notebooks * price;
console.log(total);           // 135

notebooks = notebooks + 2; // a let variable can change
total = notebooks * price;
console.log(total);           // 225
```

Everything after `//` on a line is a **comment**, a note that JavaScript ignores. Each statement usually ends with a semicolon (;). If you try `price = 50;`, the console shows `TypeError: Assignment to constant variable.`, because `price` was made with `const`. Old code also uses the word `var`; you can read it, but use `let` and `const` in new code.

Variable names may contain letters, digits, `_` and `$`, but cannot start with a digit or contain spaces. JavaScript programmers join words using **camelCase**: the first word small, each later word starting with a capital, such as `totalMarks` or `studentName`. Names are case sensitive, so `total` and `Total` are different.

Types of values

Type	Examples	<code>typeof</code> gives
Number	42, 3.5, -7	"number"
String (text)	"Pune", 'hello'	"string"
Boolean	true, false	"boolean"
Undefined	a variable with no value yet	"undefined"
Null	<code>null</code> , meaning "empty on purpose"	"object"
Object and array	<code>{ city: "Agra" }</code> , <code>[1, 2, 3]</code>	"object"

`typeof` is an operator that tells you the type of a value. The "object" answer for `null` is a famous old mistake in the language that can no longer be changed.

Operators

An **operator** is a symbol that works on values. Arithmetic uses `+`, `-`, `*`, `/`, `%` (remainder) and `**` (power). So `17 % 5` is 2 and `2 ** 3` is 8. `count++` adds 1 to count, and `total += 50` adds 50 to total.

To compare values, use these. Each gives `true` or `false`:

Operator	Meaning
<code>===</code>	Equal, and the same type
<code>!==</code>	Not equal
<code>></code> <code><</code>	Greater than, less than
<code>>=</code> <code><=</code>	Greater or equal, less or equal

To join conditions, use `&&` (and: both must be true), `||` (or: at least one true) and `!` (not).

REMEMBER

Always compare with three equals signs. The older `==` changes types before comparing, so `5 == "5"` is true and even `0 == ""` is true. With `===`, `5 === "5"` is false, which is what you mean.

A trap with plus

The `+` sign adds numbers but **joins** strings. When one side is a string, it joins. So `"5" + 2` gives `"52"`, while `"5" - 2` gives 3. Text typed into a form box always arrives as a string, so turn it into a number first with `Number()`: `Number("5") + 2` is 7. If the text is not a number, `Number("abc")` gives `NaN`, which means "Not a Number".

Strings

A **string** is a piece of text. You can write it in single quotes, double quotes or backticks (the key usually found above Tab). Pick one style and stay with it.

Useful string tools

Each character has a position called an **index**, counting from 0. A **method** is a function that belongs to a value; you call it with a dot.

JS

```
const city = "Chandigarh";
console.log(city.length);           // 10
console.log(city[0]);               // C
console.log(city.toUpperCase());    // CHANDIGARH
console.log(city.slice(0, 5));      // Chand
console.log(city.includes("garh")); // true
console.log(" Rahul ".trim());      // Rahul
console.log("red,green,blue".split(",")); // [ 'red', 'green', 'blue' ]
```

Method	What it does
<code>.length</code>	Number of characters (a property, so no brackets)
<code>.toUpperCase()</code> , <code>.toLowerCase()</code>	Change to capitals or small letters
<code>.trim()</code>	Remove spaces at both ends
<code>.slice(start, end)</code>	Take a part, stopping before end
<code>.includes(text)</code>	True if the text is inside
<code>.replaceAll(old, new)</code>	Swap every copy of one piece of text
<code>.split(separator)</code>	Break into an array of pieces

Strings cannot be changed in place. Each method gives back a new string and leaves the old one as it was.

Template literals

Joining strings with `+` gets messy quickly. A **template literal** is a string written between backticks. Inside it, anything in `${ }` is worked out and placed into the text. It can also run over several lines.

JS

```
const customer = "Ishaan";
const items = 3;
const price = 120;
const message = `Dear ${customer}, you bought ${items} items.
Total to pay: ₹${(items * price).toFixed(2)}`;
console.log(message);
```

This prints `Dear Ishaan, you bought 3 items.` and, on the next line, `Total to pay: ₹360.00`. The `.toFixed(2)` method shows a number with exactly two decimal places, which suits rupees and paise.

EXAMPLE

For Indian-style commas, use `toLocaleString("en-IN")`. So `(1250000).toLocaleString("en-IN")` gives `"12,50,000"`, which is how we write twelve lakh fifty thousand.

TRY THIS

Make two variables, `firstName` and `city`. Use one template literal to print a sentence such as `"Meera lives in Bhopal, and her city's name has 6 letters."`

CHAPTER 3

Making decisions and repeating

if, else if and else

An `if` statement runs code only when a **condition** is true. The condition goes in round brackets, and the code to run goes in curly brackets `{ }`, called a **block**.

JS

```
const marks = 82;
if (marks >= 90) {
  console.log("Grade A");
} else if (marks >= 75) {
  console.log("Grade B");
} else if (marks >= 60) {
  console.log("Grade C");
} else {
  console.log("Keep practising");
}
```

JavaScript checks from the top and runs only the first block whose condition is true. Here it prints `"Grade B"`. (These grade bands are just an example.)

For a quick choice between two values, use the **ternary operator**: condition, then `?`, the value if true, `:`, the value if false.

JS

```
const age = 16;
const label = age >= 18 ? "Adult" : "Minor";
console.log(label); // Minor
```

for loops

A **loop** repeats code. The **for** loop has three parts in its brackets, separated by semicolons: where to start, when to keep going, and what to change each time.

JS

```
for (let i = 1; i <= 10; i++) {
  console.log(`9 x ${i} = ${9 * i}`);
}
```

This prints the table of 9. It starts with `i` at 1, runs while `i <= 10`, and adds 1 to `i` after each round.

while loops

A **while** loop repeats as long as its condition stays true. It suits jobs where you do not know in advance how many rounds you need.

JS

```
let saved = 0;
let weeks = 0;
while (saved < 2000) {
  saved += 150;
  weeks++;
}
console.log(`Meera needs ${weeks} weeks and will have ₹${saved}.`);
```

If Meera saves ! 150 every week, this prints that she needs 14 weeks and will have ! 2100.

for...of

To visit every item in an array (a list of values), **for...of** is the simplest loop:

JS

```
const colours = ["saffron", "white", "green"];
for (const colour of colours) {
  console.log(colour);
}
```

Use **break** to leave a loop early and **continue** to skip to the next round.

TIP

If a loop's condition never becomes false, the page freezes. This is an **infinite loop**. Close the tab, fix the code so the condition can change, and open the page again.

CHAPTER 4

Functions, arrays and objects

A **function** is a named block of code that does one job. You write it once and use it as often as you like.

JS

```
function areaOfRectangle(length, width) {  
    return length * width;  
}  
  
console.log(areaOfRectangle(12, 10)); // 120  
console.log(areaOfRectangle(5, 4));   // 20
```

`length` and `width` are **parameters**: names that receive the values you pass in. The values themselves, such as 12 and 10, are **arguments**. `return` sends an answer back to the line that called the function, and leaves the function at once.

A parameter can have a **default value**, used when no argument is given:

JS

```
function billTotal(price, qty, discount = 0) {  
    const total = price * qty;  
    return total - (total * discount) / 100;  
}  
  
console.log(billTotal(250, 4));           // 1000  
console.log(billTotal(250, 4, 10));       // 900
```

Arrow functions

An **arrow function** is a shorter way to write a function, using `=>`. If the body is a single expression, its value is returned without writing `return`.

JS

```
const square = (n) => n * n;
const add = (a, b) => a + b;
const greet = (person) => {
  const text = `Namaste, ${person}!`;
  return text;
};

console.log(square(7));           // 49
console.log(add(15, 27));         // 42
console.log(greet("Zoya"));       // Namaste, Zoya!
```

When the body has curly brackets and more than one line, you do need `return`.

Functions as values

In JavaScript a function is a value, just like a number. You can hand it to another function, which will call it later. A function passed in this way is called a **callback**.

JS

```
setTimeout(() => {
  console.log("Two seconds have passed");
}, 2000);
```

`setTimeout` waits 2000 milliseconds (2 seconds) and then calls the arrow function. You will use callbacks all the time with arrays and with clicks.

REMEMBER

Variables made with `let` or `const` inside a block or function exist only inside it. This is called **scope**. It stops one part of a program from accidentally changing another part's variables.

Arrays

An **array** is an ordered list of values inside square brackets. Like strings, arrays count positions from 0.

JS

```
const marks = [72, 88, 65, 91];
marks.push(79);           // add to the end
console.log(marks.length); // 5
console.log(marks[0]);     // 72
console.log(marks.includes(91)); // true
console.log(marks.indexOf(65)); // 2
console.log(marks.join(" | ")); // 72 | 88 | 65 | 91 | 79
```

Other useful methods: `pop()` removes the last item, `shift()` removes the first, `unshift()` adds to the front, and `splice(index, 1)` removes one item at a position.

Array methods that take a function

These methods call your callback once for every item. They are among the most useful tools in JavaScript.

JS

```
const prices = [120, 450, 80, 999, 300];

prices.forEach((p, i) => console.log(i, p));           // each item and its
index
const doubled = prices.map((p) => p * 2);              // [240, 900, 160,
1998, 600]
const cheap = prices.filter((p) => p < 300);           // [120, 80]
const firstBig = prices.find((p) => p > 400);          // 450
const total = prices.reduce((sum, p) => sum + p, 0);    // 1949
console.log(doubled, cheap, firstBig, total);
```

Method	Gives back
<code>forEach</code>	Nothing; just runs the callback for each item
<code>map</code>	A new array with every item changed
<code>filter</code>	A new array with only the items that pass a test
<code>find</code>	The first item that passes, or <code>undefined</code>
<code>reduce</code>	One value built up from all items, such as a total

DID YOU KNOW

`sort()` with no callback sorts items as text, so `[10, 9, 100].sort()` gives `[10, 100, 9]`. To sort numbers, pass a compare function: `prices.sort((a, b) => a - b)`. This sorts the array itself, from smallest to largest.

Objects

An **object** groups related values under names. Each name and value pair is called a **property**.

JS

```
const student = {
  name: "Kavya",
  rollNo: 17,
  city: "Kochi",
};
console.log(student.name);           // Kavya
console.log(student["city"]);        // Kochi
student.section = "B";               // add a new property
student.rollNo = 18;                 // change one
console.log(student);
```

Read a property with a dot, or with square brackets and a string. Real data is often an array of objects:

```
JS

const students = [
  { name: "Kavya", marks: 88 },
  { name: "Rahul", marks: 64 },
  { name: "Sneha", marks: 93 },
];
const toppers = students.filter((s) => s.marks >= 85).map((s) => s.name);
console.log(toppers); // [ 'Kavya', 'Sneha' ]
```

JSON

JSON (JavaScript Object Notation) is a way to write objects and arrays as plain text, so they can be saved or sent. `JSON.stringify(students)` turns the array into a string, and `JSON.parse(text)` turns such a string back into real objects. You will need both for saving data in the to-do list project.

CHAPTER 5

Changing the page: the DOM and events

When a browser loads a page, it turns the HTML into a family tree of objects called the **DOM** (Document Object Model). Each tag becomes an object, sometimes called a **node**, that JavaScript can find and change. `document` is the object at the top of the tree.

Finding elements

Method	What it finds
<code>document.getElementById("title")</code>	The element with <code>id="title"</code>
<code>document.querySelector(".note")</code>	The first match for a CSS selector
<code>document.querySelectorAll(".note")</code>	Every match, as a list you can loop through

`querySelector` uses the same selectors as CSS: `"#title"` for an id, `".note"` for a class, `"p"` for a tag. If nothing matches, it gives back `null`.

Changing text, styles and classes

Try this page. Save it as `dom.html` and open it.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DOM practice</title>
  <style>
    .dark { background: #222; color: #eee; }
    .highlight { border-left: 4px solid orange; padding-left: 8px; }
  </style>
</head>
<body>
  <h1 id="title">Welcome</h1>
  <p class="note">First note</p>
  <p class="note">Second note</p>
  <button id="plus">Add one</button> <span id="counter">0</span>
  <button id="themeBtn">Dark mode</button>
  <p><input id="nameBox" placeholder="Your name" aria-label="Your name"></p>
  <p id="greeting"></p>
  <script>
    const title = document.getElementById("title");
    title.textContent = "Welcome to JavaScript";
    title.style.color = "darkblue";

    document.querySelectorAll(".note").forEach((note) => {
      note.classList.add("highlight");
    });

    const extra = document.createElement("p");
    extra.textContent = "This paragraph was made by JavaScript.";
    document.body.append(extra);

    let count = 0;
    const counter = document.querySelector("#counter");
    document.querySelector("#plus").addEventListener("click", () => {
      count++;
      counter.textContent = count;
    });

    document.querySelector("#themeBtn").addEventListener("click", () => {
      document.body.classList.toggle("dark");
    });

    const nameBox = document.querySelector("#nameBox");
    const greeting = document.querySelector("#greeting");
    nameBox.addEventListener("input", () => {
      greeting.textContent = `Hello, ${nameBox.value}!`;
    });
  </script>
</body>
</html>

```

Here is what the script does:

- `textContent` sets the text inside an element.
- `style` changes one CSS property. CSS names with a dash become camelCase, so `backgroundColor` is written `style.backgroundColor`.
- `classList.add()`, `.remove()` and `.toggle()` switch CSS classes on and off. This is usually better than `style`, because the looks stay in your CSS.
- `createElement()` makes a new element, and `append()` puts it on the page.
- For form boxes, `.value` reads or sets what is typed in them. In the HTML, `placeholder` shows grey hint text in an empty box, and `aria-label` gives the box a name that screen readers speak aloud.

Events

An **event** is something that happens on the page: a click, a key press, typing in a box.

`addEventListener(eventName, callback)` asks the browser to call your function whenever that event happens on that element.

Event	Happens when
<code>click</code>	The element is clicked or tapped
<code>input</code>	The text in a box changes
<code>change</code>	A drop-down or checkbox choice changes
<code>submit</code>	A form is sent
<code>keydown</code>	A key is pressed

Your callback receives an **event object** with details. `event.target` is the element that was clicked, `event.key` is the key that was pressed, and `event.preventDefault()` stops the browser's normal action, such as reloading the page when a form is sent.

SAFETY

There is also an `innerHTML` property, which reads text as HTML tags. Never put anything a user typed into `innerHTML`. Someone could type code, and your page would run it. Use `textContent` for text, as every example in this book does.

Project: a form that checks itself

Forms collect names, phone numbers and emails. If someone leaves a box empty or types a 9-digit number, it is kinder to say so at once than after the form is sent. Checking input like this is called **validation**.

Save this as form.html:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Admission enquiry</title>
  <style>
    .error { color: #b00020; font-size: 0.9em; }
  </style>
</head>
<body>
  <h1>Admission enquiry</h1>
  <form id="enquiry" novalidate>
    <p><label for="fullName">Full name</label><br>
      <input id="fullName"> <span class="error" id="fullNameError"></span></p>
    <p><label for="mobile">Mobile number</label><br>
      <input id="mobile" inputmode="numeric"> <span class="error"
id="mobileError"></span></p>
    <p><label for="email">Email</label><br>
      <input id="email" type="email"> <span class="error" id="emailError"></
span></p>
    <button type="submit">Send</button>
  </form>
  <p id="result"></p>
  <script src="form.js"></script>
</body>
</html>
```

And save this as form.js in the same folder:

```

const form = document.querySelector("#enquiry");
const result = document.querySelector("#result");

function check(id, message) {
  document.querySelector(`#${id}Error`).textContent = message;
  return message === "";
}

form.addEventListener("submit", (event) => {
  event.preventDefault();
  const fullName = document.querySelector("#fullName").value.trim();
  const mobile = document.querySelector("#mobile").value.trim();
  const email = document.querySelector("#email").value.trim();

  const nameOk = check("fullName", fullName.length >= 3 ? "" : "Please enter your full name.");
  const mobileOk = check("mobile", /^\d{10}$/.test(mobile) ? "" : "Mobile number must be exactly 10 digits.");
  const emailOk = check("email", email.includes("@") && email.includes(".") ? "" : "Please enter a valid email.");

  if (nameOk && mobileOk && emailOk) {
    result.textContent = `Thank you, ${fullName}. Your details look correct.`;
    form.reset();
  } else {
    result.textContent = "";
  }
});

```

How it works

1. `novalidate` on the form switches off the browser's own pop-up messages, so our messages show instead. `inputmode="numeric"` asks phones to show a number keypad for the mobile box.
2. When the form is sent, `event.preventDefault()` stops the page from reloading.
3. `.trim()` removes extra spaces, so a box with only spaces counts as empty.
4. `check()` writes a message into the error span next to a box. An empty message clears old errors. It returns `true` when there was no error.
5. The mobile test uses a **regular expression**, a pattern for text, written between slashes. `^\d{10}$` means: from the start (^), exactly ten digits (`\d{10}`), then the end (\$). So "981001234" fails because of the space.
6. The email test is simple: it only checks for an @ and a dot.
7. Only when all three are fine does it thank the user and clear the form with `form.reset()`.

EXAMPLE

Type "Al" as the name and "98765" as the mobile, and leave the email empty. A red message appears beside each box. Correct them to "Aliya Khan" and "9876543210", add an email, and the thank-you line appears.

REMEMBER

Checks in the browser help the user, but anyone can switch JavaScript off. A real website must check the same data again on its server before saving it.

CHAPTER 7

Projects: a to-do list and a quiz

A to-do list that remembers

A to-do list loses its tasks when you close the tab, unless you save them. **localStorage** is a small storage area that the browser keeps for each website. It holds text under names called **keys**, and the text stays after the browser is closed and opened again.

Code	What it does
<code>localStorage.setItem("tasks", text)</code>	Saves text under the key "tasks"
<code>localStorage.getItem("tasks")</code>	Reads it back, or gives <code>null</code> if nothing is saved
<code>localStorage.removeItem("tasks")</code>	Deletes that key

localStorage stores only strings, so we turn our array into JSON text before saving and back again after reading.

Save as todo.html:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My to-do list</title>
  <style>
    .done { text-decoration: line-through; color: gray; }
    li span { cursor: pointer; margin-right: 8px; }
  </style>
</head>
<body>
  <h1>My to-do list</h1>
  <form id="addForm">
    <input id="taskInput" placeholder="New task" aria-label="New task">
    <button>Add</button>
  </form>
  <ul id="list"></ul>
  <p id="count"></p>
  <script src="todo.js"></script>
</body>
</html>
```

Save as todo.js:

```

const form = document.querySelector("#addForm");
const input = document.querySelector("#taskInput");
const list = document.querySelector("#list");
const count = document.querySelector("#count");

let tasks = JSON.parse(localStorage.getItem("tasks")) || [];

function save() {
  localStorage.setItem("tasks", JSON.stringify(tasks));
}

function render() {
  list.textContent = "";
  tasks.forEach((task, index) => {
    const li = document.createElement("li");
    const text = document.createElement("span");
    text.textContent = task.text;
    if (task.done) text.classList.add("done");
    text.addEventListener("click", () => {
      task.done = !task.done;
      save();
      render();
    });
    const del = document.createElement("button");
    del.textContent = "Delete";
    del.addEventListener("click", () => {
      tasks.splice(index, 1);
      save();
      render();
    });
    li.append(text, del);
    list.append(li);
  });
  const left = tasks.filter((t) => !t.done).length;
  count.textContent = `${left} of ${tasks.length} tasks left`;
}

form.addEventListener("submit", (event) => {
  event.preventDefault();
  const text = input.value.trim();
  if (text === "") return;
  tasks.push({ text: text, done: false });
  input.value = "";
  save();
  render();
});

render();

```

How the to-do list works

1. Each task is an object such as `{ text: "Buy milk", done: false }`, and `tasks` is an array of them.

2. On loading, `getItem` reads the saved text. The first time, it gives `null`, `JSON.parse(null)` also gives `null`, and `|| []` then starts us with an empty array.
3. `render()` clears the list and builds it again from the array. Clicking a task's text flips `done`; the Delete button removes it with `splice()`.
4. After every change, `save()` writes the whole array back to `localStorage`.
5. The button inside the form sends the form, so pressing Enter in the box also adds a task.

TRY THIS

Add a "Clear completed" button that keeps only unfinished tasks: `tasks = tasks.filter((t) => !t.done)`; then save and render.

NOTE

`localStorage` belongs to one browser on one device, and it is wiped if the user clears site data. Never keep passwords or private details in it. If saving does not seem to work when you double-click the file, open the page through a small local web server instead, such as the "live server" add-ons that many code editors offer.

A quiz page

This quiz shows one question at a time with clickable answers, says whether each answer is right, and gives a score at the end. The questions live in an array of objects, so you can add more without touching the rest of the code.

Save as `quiz.html`:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Quick quiz</title>
  <style>
    #options button { display: block; margin: 6px 0; min-width: 240px; }
  </style>
</head>
<body>
  <h1>Quick JavaScript quiz</h1>
  <p id="question"></p>
  <div id="options"></div>
  <p id="feedback"></p>
  <button id="next" hidden>Next question</button>
  <script src="quiz.js"></script>
</body>
</html>
```

Save as `quiz.js`:

```

const questions = [
  { text: "Which tag adds JavaScript to a page?", options: ["<style>", "<script>", "<link>"], answer: 1 },
  { text: "Which word makes a variable that cannot be reassigned?", options: ["let", "var", "const"], answer: 2 },
  { text: "What does 5 === '5' give?", options: ["true", "false", "an error"], answer: 1 },
  { text: "Which method adds an item to the end of an array?", options: ["push()", "pop()", "shift()"], answer: 0 },
];

let current = 0;
let score = 0;
const questionEl = document.querySelector("#question");
const optionsEl = document.querySelector("#options");
const feedbackEl = document.querySelector("#feedback");
const nextBtn = document.querySelector("#next");

function showQuestion() {
  const q = questions[current];
  questionEl.textContent = `Q${current + 1}. ${q.text}`;
  feedbackEl.textContent = "";
  optionsEl.textContent = "";
  nextBtn.hidden = true;
  q.options.forEach((option, i) => {
    const btn = document.createElement("button");
    btn.textContent = option;
    btn.addEventListener("click", () => choose(i));
    optionsEl.append(btn);
  });
}

function choose(i) {
  const q = questions[current];
  if (i === q.answer) {
    score++;
    feedbackEl.textContent = "Correct!";
  } else {
    feedbackEl.textContent = `Not quite. The answer is ${q.options[q.answer]}.`;
  }
  optionsEl.querySelectorAll("button").forEach((b) => (b.disabled = true));
  nextBtn.hidden = false;
}

nextBtn.addEventListener("click", () => {
  current++;
  if (current < questions.length) {
    showQuestion();
  } else {
    questionEl.textContent = `Quiz over! You scored ${score} out of ${questions.length}.`;
    optionsEl.textContent = "";
    feedbackEl.textContent = "";
  }
});

```

```
    nextBtn.hidden = true;
  }
});

showQuestion();
```

How the quiz works

1. Each question object has its `text`, an array of `options`, and the index of the right option in `answer`. In the first question, `answer: 1` points to "`<script>`", because counting starts at 0.
2. `showQuestion()` clears the old buttons and makes one button per option. Each button's callback remembers its own `i`.
3. `choose(i)` compares the choice with the answer, updates the score, and disables all the buttons so nobody can answer twice.
4. The Next button starts with the `hidden` attribute, so it is invisible until an answer is chosen; setting `nextBtn.hidden` to `false` or `true` shows or hides it. Each click moves `current` forward. After the last question, the score is shown.
5. Because the options are set with `textContent`, the text `<script>` appears on the button as it is, instead of being read as a tag.

TRY THIS

Add three questions of your own. Then show "Excellent!" when the final score is full marks.

CHAPTER 8

Finding and fixing bugs

A **bug** is a mistake that makes a program go wrong. Finding and fixing bugs is called **debugging**. Every programmer spends a lot of time on it, so it is a skill worth learning well.

Read the error message

When JavaScript hits an error, it stops and prints a red message in the console. It names the type of error, gives a short reason, and shows the file and line number on the right. Click that link and the browser shows you the exact line.

Error	What it usually means
SyntaxError	A typing mistake: a missing bracket, quote or comma
ReferenceError: x is not defined	A misspelt name, or a variable used outside its scope
TypeError: Cannot set properties of null	<code>querySelector</code> found nothing, often a wrong id or a script running before the HTML

Error	What it usually means
TypeError: x is not a function	Calling something that is not a function, often a misspelt method name

Look at this bug:

JS

```
const totalBox = document.querySelector("#totl");
totalBox.textContent = "₹500";
```

The console says `Uncaught TypeError: Cannot set properties of null (setting 'textContent')`. "null" tells you `querySelector` found nothing, because the id is spelt `totl` instead of `total`.

Print what is happening

When there is no error but the answer is wrong, add `console.log()` lines to see the values as the code runs:

JS

```
const qty = document.querySelector("#qty").value;
console.log("qty is", qty, typeof qty);
```

If this prints `qty is 2 string`, you have found the reason why `qty + 1` gave "21": the value is text. `console.table(students)` shows an array of objects as a neat table.

Pause the code with a breakpoint

1. Open the developer tools and click the **Sources** tab (in Firefox it is called **Debugger**).
2. Open your .js file and click a line number. A marker appears. This is a **breakpoint**.
3. Do whatever runs that code, such as clicking the button. The page pauses on that line.
4. Hover over any variable to see its value, then use the step buttons to run one line at a time.

Writing the word `debugger;` in your code does the same thing whenever the developer tools are open.

Good habits

- Save the file and refresh the page after every change. Ctrl+Shift+R reloads the page without using copies the browser saved earlier.
- Write a few lines, test them, then write a few more.
- Check spelling and capitals: `getElementById` is not `getElementByID`.
- Use `===`, and turn text into numbers with `Number()`.
- Name things clearly, so that the bug is easier to see.

DID YOU KNOW

Programmers often explain their code, line by line, to a rubber duck or an empty chair. Saying each step out loud is a surprisingly good way to notice the step that is wrong.

CHECK YOURSELF

Practice questions

1. What are the jobs of HTML, CSS and JavaScript on a web page?
2. Why is the `<script>` tag usually placed just before `</body>`?
3. How do you open the console in Chrome or Edge on Windows?
4. What is the difference between `let` and `const`?
5. What do `typeof 42`, `typeof "42"` and `typeof true` give?
6. What do `"7" + 3` and `"7" - 3` give, and why are they different?
7. Why should you use `===` instead of `==`?
8. Write a template literal that prints "Total: !" followed by the value of a variable `amount`.
9. What does `"Bengaluru".slice(0, 4)` give?
10. Write an `if` and `else` that logs "Pass" when `marks` is 40 or more and "Try again" otherwise.
11. Write a `for` loop that logs the even numbers from 2 to 20.
12. Rewrite `function cube(n) { return n * n * n; }` as an arrow function.
13. What does a function return if it has a default parameter `discount = 0` and you call it without that argument?
14. Given `const nums = [5, 12, 8, 20]`, what do `nums.filter((n) => n > 10)` and `nums.map((n) => n / 2)` give?
15. How would you add up all the numbers in `nums` with `reduce`?
16. Make an object for a book with a title, an author and a price, and log its title.
17. What is the DOM?
18. What is the difference between `querySelector` and `querySelectorAll`?
19. Write code that makes a button with id `hello` show an alert saying "Hi!" when clicked.
20. Why is `textContent` safer than `innerHTML` for text a user types?
21. In the form project, what does `event.preventDefault()` do?
22. Why must data be turned into JSON before saving it in `localStorage`?
23. The console shows "Cannot set properties of null". What are two likely causes?
24. What is a breakpoint, and how do you set one?

Answers

1. HTML gives structure (headings, paragraphs, buttons), CSS controls looks (colours, fonts, layout), and JavaScript adds behaviour (reacting to clicks, typing and more).
2. The browser reads the page from top to bottom. Placed at the end, the script runs after the elements it needs have been created.
3. Press F12 or Ctrl+Shift+I, then click the Console tab.
4. A `let` variable can be given a new value later; a `const` variable cannot be reassigned.
5. "number", "string" and "boolean".
6. "7" + 3 gives "73" because + joins when one side is a string. "7" - 3 gives 4 because - only works on numbers, so the string is turned into a number.
7. `===` checks both value and type without converting anything, so `5 === "5"` is false. `==` converts types first and gives surprising results such as `0 == ""` being true.
8. ``Total: ${amount}``, for example inside `console.log()`.
9. "Beng", the characters at index 0, 1, 2 and 3.
10.

```
if (marks >= 40)
  { console.log("Pass"); } else
  { console.log("Try again"); }
```
11.

```
for (let i = 2; i <= 20; i += 2)
  { console.log(i); }
```
12. `const cube = (n) => n * n * n;`
13. The parameter takes its default value, 0, so the function works as if no discount was given.
14. `[12, 20]` and `[2.5, 6, 4, 10]`.
15. `nums.reduce((sum, n) => sum + n, 0)`, which gives 45.
16. `const book = { title: "Panchatantra Tales", author: "Vishnu Sharma", price: 199 }; then console.log(book.title);`
17. The Document Object Model: the browser's tree of objects made from the page's HTML, which JavaScript can read and change.
18. `querySelector` gives the first matching element (or null); `querySelectorAll` gives every match as a list.
19. `document.querySelector("#hello").addEventListener(() => { alert("Hi!"); });`
20. `textContent` shows text exactly as typed. `innerHTML` reads it as HTML, so typed code could run on the page.
21. It stops the browser's normal action for a sent form, which is to reload the page, so our checks can run and show messages.
22. `localStorage` can only store strings. `JSON.stringify` turns an array or object into a string; `JSON.parse` turns it back.
23. The selector's id or class is misspelt, or the script runs before the element exists (for example, it is in the head without `defer`).
24. A place where the browser pauses your code so you can look at variables. Open the Sources (or Debugger) tab, open the file and click a line number, or write `debugger` in the code.

Glossary

Argument

the actual value passed to a function when it is called.

Array

an ordered list of values inside square brackets, counted from 0.

Arrow function

a short way to write a function using `=>`.

Boolean

a value that is either true or false.

Breakpoint

a marked line where the browser pauses the code for checking.

Bug

a mistake in a program that makes it behave wrongly.

Callback

a function handed to another function, to be called later.

camelCase

joining words with each later word starting with a capital, as in `totalMarks`.

Console

the browser panel where JavaScript prints messages and errors.

const

a keyword that makes a variable which cannot be reassigned.

DOM

the Document Object Model, the browser's tree of objects built from a page's HTML.

Event

something that happens on a page, such as a click or a key press.

Function

a named block of code that does one job and can be reused.

Index

the position number of an item in a string or array, starting from 0.

JSON

a text format for writing objects and arrays so they can be saved or sent.

let

a keyword that makes a variable whose value can change.

localStorage

a small browser storage area that keeps text for a website after the tab closes.

Method

a function that belongs to a value and is called with a dot.

NaN

short for Not a Number, the result of a failed number conversion.

Object

a group of named values called properties, inside curly brackets.

Parameter

a name in a function's definition that receives an argument.

Property

one name and value pair inside an object.

String

a piece of text inside quotes or backticks.

Template literal

a string in backticks that can include values with

`${ }.`

Validation

checking that information typed into a form is complete and sensible.



Gurukul Technicals Institute

Computer education institute in Delhi, with authorised member centres, robotics classes and science projects, working models and DIY kits.

COURSES

Computer & Office Skills · Accounting & Business Software · Programming & Web Development · Design & Digital Skills · Hardware & Networking · Robotics & Coding · Other Programmes

PROJECTS AND KITS

Ready-made school projects · DIY and STEM kits · Robotics and coding · Components and spares · Engineering projects

CONTACT

Head office: A-10, Chief Commissioner Colony, opposite Rana Pratap Bagh, Delhi 110007

Phone: 011 2721 1512 · **Mobile and WhatsApp:** +91 97116 99511

Email: engineergurukul@yahoo.co.in

Website: engineergurukul.com · **Instagram:** [@science_projects_club](https://www.instagram.com/science_projects_club)

More free ebooks for computer courses, robotics and science projects: engineergurukul.com/resources/ebooks/

Free to read, print and share. Not for sale.