JAVASCRIPT PROGRAMMING

A practical guide to writing useful programs and interactive web pages

LEARNSAPPEAL | ORIGINAL COURSE

Learn the language, then make the page respond.

Practice values, functions, events, browser interactions, and web data as you build useful features.

What you need

A text editor and a current browser. Save an HTML file and a JavaScript file in the same folder, then open the page. Type the examples, run them, and change one part at a time.

Course map: JavaScript building blocks (Modules 1-5), then browser interactions and projects (Modules 6-12). Includes code examples, questions with answers, projects, and glossary.

How to use this course

Create a folder for experiments. Keep page structure in HTML and behavior in JavaScript. Use browser developer tools and check the Console when something goes wrong. Read the error, check its line, and test a small change.

By the end: use variables, functions, and collections; respond to page events; update content safely; validate forms; request data with Fetch; organize modules; and test an interactive feature.

Course map

• 1. What JavaScript does
• 2. Values and variables
• 3. Decisions and loops
• 4. Arrays and objects
• 5. Functions and scope
• 6. Connect JavaScript to a page
• 7. Events and interactions
• 8. Forms and validation
• 9. Async JavaScript and Fetch
• 10. Modules and reusable code
• 11. Debugging, testing, and safety
• 12. Capstone: course finder

JAVASCRIPT BUILDING BLOCKS

1. What JavaScript does

JavaScript runs in browsers and other environments. On a page it can react to clicks, check form input, change content, and request information. A browser runs a JavaScript file linked from HTML. Begin with a Console message to confirm the script loaded.

Example

```
console.log('Hello from JavaScript!');
// HTML: <script src="app.js"></script>
```

Check your understanding

• What can JavaScript do on a page?

• Where does console.log output appear?

Suggested answers

1. Add behavior and respond to actions.

2. In the browser developer Console.

2. Values and variables

Values include text, numbers, true or false, and other data. Use const for a name that will not be reassigned, and let when reassignment is needed. Use === for strict equality. Template literals use backtick characters to include expressions.

Example

```
const course = 'JavaScript';
let completed = 2;
const total = 12;
const percent = completed / total * 100;
console.log(course + ': ' + percent.toFixed(0) + '%');
```

Check your understanding

• When is const useful?

• What does === compare?

Suggested answers

1. When the name will not be reassigned.

2. Both value and type.

3. Decisions and loops

Use if and else to choose instructions. Comparisons produce true or false. A for loop repeats work; for...of is useful for visiting each value in an array. Every loop needs a stopping rule so it does not run forever.

Example

```
const score = 76;
if (score >= 70) {
  console.log('Keep going');
} else {
  console.log('Try another lesson');
}
for (const topic of ['HTML', 'CSS']) console.log(topic);
```

Check your understanding

• What does an if statement choose?

• When is for...of useful?

Suggested answers

1. Which branch runs based on a condition.

2. When iterating through values in a collection.

4. Arrays and objects

An array stores an ordered list. An object groups named properties. Methods such as map transform lists and filter keeps matching items. These methods return new arrays, making data changes easier to follow.

Example

```
const courses = [
  { title: 'Python', topic: 'coding' },
  { title: 'Web Design', topic: 'web' },
];
const web = courses.filter(c => c.topic === 'web');
```

Check your understanding

• What is an array useful for?

• What does filter return?

Suggested answers

1. Keeping an ordered list of values.

2. A new array with matching items.

5. Functions and scope

A function groups instructions under a name. Parameters receive inputs; return sends a result back. Arrow functions are compact. A variable scope is where it can be used. Declare variables in the smallest useful scope to avoid unexpected changes.

Example

```
function progress(done, total) {
  if (total <= 0) return 0;
  return done / total * 100;
}
const percent = progress(3, 12);
```

Check your understanding

• What does return do?

• What is a parameter?

Suggested answers

1. Sends a result to the code that called the function.

2. A named input accepted by a function.

BROWSER INTERACTIONS AND PROJECTS

6. Connect JavaScript to a page

The Document Object Model (DOM) represents the page as objects JavaScript can access. Select elements with querySelector. Update ordinary text with textContent. Keep structure in HTML and behavior in JavaScript, and run scripts after elements exist.

Example

```
// HTML: <p id="message">Welcome</p>
const message = document.querySelector('#message');
if (message) message.textContent = 'Next lesson is ready.';
```

Check your understanding

• What does the DOM represent?

• Which property sets ordinary text?

Suggested answers

1. The current page structure as objects.

2. textContent.

7. Events and interactions

An event describes something that happened, such as a click or form submission. Add a listener to respond. Keep handlers focused: read needed state, perform an action, update the page. Use semantic buttons so keyboard users can activate actions.

Example

```
const button = document.querySelector('#show-tip');
const tip = document.querySelector('#tip');
button?.addEventListener('click', () => {
  tip.textContent = 'Take a short break.';
});
```

Check your understanding

- Name two browser events.
- Why use a button instead of a clickable div?

Suggested answers

1. Click and submit; input and change are others.
2. Buttons have expected keyboard and accessibility behavior.

8. Forms and validation

Handle form submission with an event listener. Call preventDefault when the page handles the submission itself. Read values, check them, and show clear feedback. Browser checks help users but cannot replace server validation when data is sent to a server.

Example

```
const form = document.querySelector('#signup');
form?.addEventListener('submit', event => {
  event.preventDefault();
  const data = new FormData(form);
  const email = String(data.get('email') || '').trim();
  document.querySelector('#status').textContent = email ? 'Thanks!' : 'Enter your email.';
});
```

Check your understanding

- What does preventDefault stop?
- Can browser checks be the only validation?

Suggested answers

1. The browser default form submission behavior.
2. No. The server must validate data it receives.

9. Async JavaScript and Fetch

Network work takes time. A Promise represents a result that may arrive later. async and await make promise-based steps easier to read. Fetch returns a Response; check response.ok because HTTP error statuses still produce a response. Show loading, success, and error states.

Example

```
async function loadCourses() {
  const status = document.querySelector('#status');
  status.textContent = 'Loading...';
  try {
    const response = await fetch('/courses.json');
    if (!response.ok) throw new Error('Request failed');
    const data = await response.json();
    status.textContent = data.length + ' courses found.';
  } catch { status.textContent = 'Could not load courses.'; }
}
```

Check your understanding

- What does await do?
- Why check response.ok?

Suggested answers

1. Waits for a promise result needed by the next step.
2. Fetch does not reject automatically for every HTTP error status.

10. Modules and reusable code

A module keeps related code in its own file. Export values another file needs and import them where used. Modules organize a project and reduce global name collisions. In a browser, set script type to module.

Example

```
// progress.js
export function percent(done, total) {
  return total ? done / total * 100 : 0;
}
// app.js: import { percent } from './progress.js';
```

Check your understanding

• What does export allow?

• What does a module help organize?

Suggested answers

1. Another module can import the value or function.
2. Related code in separate files.

11. Debugging, testing, and safety

The Console reports errors and lets you inspect values. Test typical, boundary, and invalid inputs. Treat user data as untrusted: use textContent rather than inserting it as HTML. Never put private keys or passwords in browser code; visitors can inspect it.

Example

```
function clamp(value) {
  return Math.min(100, Math.max(0, value));
}
console.assert(clamp(50) === 50);
console.assert(clamp(-3) === 0);
```

Check your understanding

• Why is user input untrusted?

• Can a secret in browser code be private?

Suggested answers

1. Visitors can change values sent from their browser.
2. No. Code delivered to a browser can be inspected.

12. Capstone: course finder

Create a course finder that lists titles and topics, filters by topic, and shows a no-results message. Add links to course PDFs. Keep data separate from display logic, support keyboard use, test empty values, and document how to update the project.

Example

```
const courses = [
  { title: 'Python', topic: 'coding' },
  { title: 'HTML & CSS', topic: 'web' },
];
const webCourses = courses.filter(c => c.topic === 'web');
```

Check your understanding

• Why show a no-results message?

• Name one useful test.

Suggested answers

1. It confirms the search ran but found no match.

2. Empty list, missing title, or keyboard use.

Practice projects

1. Reading list

Store article titles in an array, show them on a page, filter by topic, and display a helpful empty state. Use textContent for user-entered text.

2. Lesson progress tracker

Mark lessons complete and show a percentage. Put calculations in a function and check zero, first, and all lessons.

3. Course search

Fetch course data, show loading and error states, render clear links, check response.ok, and handle an empty result.

Project checklist
Does the page explain what to do? Do buttons work with a keyboard? Are empty, loading, and error states clear? Are user values inserted safely?

Quick reference

Declare a value

```
const siteName = 'LearnsAppeal';
```

Filter a list

```
const selected = items.filter(item => item.active);
```

Handle an event

```
button.addEventListener('click', handleClick);
```

Load JSON

```
const response = await fetch('/data.json');
if (!response.ok) throw new Error('Request failed');
const data = await response.json();
```

Glossary

Term	Meaning
Array	An ordered collection of values.
DOM	The browser representation of a page as objects.
Event	An occurrence such as a click, input, or form submission.
Function	A reusable group of instructions.
Module	A JavaScript file that exports or imports code.
Object	A collection of named properties.
Parameter	A named input accepted by a function.
Promise	An object representing a result that may arrive later.
Scope	The part of a program where a name can be accessed.
API	A defined way for software to request data or actions.
Validation	Checking data against expected rules.
textContent	A property for reading or setting an element plain text.

Keep learning

Use the maintained MDN JavaScript Guide for language concepts, Fetch, and browser APIs. Check current documentation when working with new browser features.

- MDN Web Docs. *JavaScript Guide.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
- MDN Web Docs. *Fetch API.* https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- MDN Web Docs. *Using promises.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- MDN Web Docs. *Web security.* https://developer.mozilla.org/en-US/docs/Web/Security

Course completion checklist

I can use variables, conditions, arrays, objects, and functions; connect JavaScript to a page; handle events and form input; request data with async code; organize modules; debug common errors; and build an interactive feature with safe, clear feedback.