FREE COMPANION DOWNLOAD

TYPE · PREDICT · RUN · MODIFY

Practice Test Pack

Thirty mixed questions covering the whole book

From **HTML, CSS & JavaScript the TPRM Way** by John I. Jerkovic

PRACTICAL TECH MADE SIMPLE

BEFORE YOU START

Instructions

> Work on paper, away from a computer, and give yourself about ninety minutes. Where a question asks you to predict output, write it exactly, punctuation and spacing included. Where it asks for markup or a stylesheet, write it out in full rather than describing it. Answer every question before you check anything, then open a browser and run what you can. A wrong prediction is the useful part of this: find the first place where your reading of the code parted company with the browser, and note which chapter that belongs to. The questions run roughly in book order within each part.

QUESTIONS

Part 1 — Predict the output (10 questions)

1 CH 2 · **PREDICT**

Write the heading outline this document produces, in order and with its levels. Then name the four landmark regions, and say which one holds the primary content.

```
<body>
  <header>
    <h1>Trail Notes</h1>
    <nav aria-label="Primary">
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="gear.html">Gear</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <article>
      <h2>Ridge Walk</h2>
      <section>
        <h3>Conditions</h3>
        <p>Dry and clear.</p>
      </section>
    </article>
  </main>
  <footer>Written after each walk.</footer>
</body>
```

2 CH 3 · **PREDICT**

This markup is in site/pages/`gear.html`. The site folder holds `index.html` and two folders, pages and images. Name the file the browser asks for in each of the three cases, and say which one asks for nothing at all.

```
<a href="../index.html">Home</a>
<img src="../images/boots.jpg" alt="Worn walking boots">
<a href="#care">Care instructions</a>
```

3 CH 5 · PREDICT

The page contains one element: Read me . Which color does it take? Then give the winner after the first rule is deleted, and again after the second is deleted as well.

```
#lead { color: navy; }
.note.important { color: olive; }
.important { color: brown; }
p { color: black; }
```

4 CH 6 · PREDICT

Give the content width, then the total rendered width excluding margin, then the total horizontal space the element occupies including margin. Then give all three again with `box-sizing` changed to `border-box`.

```
.card {
  width: 400px;
  padding: 24px;
  border: 2px solid #333;
  margin: 16px;
  box-sizing: content-box;
}
```

5 CH 7 · PREDICT

The container is exactly `800px` wide and holds six cards. How many columns does the browser create, and how wide is each one? Show the arithmetic. Then say what changes at `620px`.

```
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  gap: 20px;
}
```

6 CH 8 · PREDICT

Assume the browser's base font size is the usual `16px`. Give the rendered width of the wrapper at viewport widths of `400px`, `900px`, and `1600px`, and say what the margin declaration is doing at each.

```
.wrapper {
  width: min(100% - 3rem, 45rem);
  margin-inline: auto;
}
```

7 CH 9 · PREDICT

Write the three Console lines exactly. Two of the three operators behave differently from what the values look like they should do. Explain each line.

```
const a = 3;
const b = "3";

console.log(a + b);
console.log(a * b);
console.log(a === Number(b));
```

8 CH 10 · PREDICT

Write every Console line in order. Then say which value stops the loop, and what the last line would read if the break were removed.

```
const names = ["Ana", "Ben", "Cara", "Dan"];
let found = "";

for (const n of names) {
  console.log(n);

  if (n.length === 4) {
    found = n;
    break;
  }
}

console.log("found", found);
```

9 CH 12 · PREDICT

Write the three Console lines. Then say why the second number is what it is, and which of the two operations changed an existing array.

```
const cart = [4, 8, 15];

const doubled = cart.map(function (n) {
  return n * 2;
});

cart.push(16);

console.log(cart.length);
console.log(doubled.length);
console.log(doubled[2]);
```

10 CH 13 · PREDICT

Write the three Console lines exactly, the third one character for character. Then explain why pushing onto the copy left the original alone here, when assigning one object to another does not.

```
const original = { name: "Ana", scores: [1, 2] };
const text = JSON.stringify(original);
const copy = JSON.parse(text);

copy.scores.push(3);

console.log(original.scores.length);
console.log(copy.scores.length);
console.log(text);
```

QUESTIONS

Part 2 — Find and fix the defect (8 questions)

11 CH 2 · **DEBUG**

Name four things wrong with this markup, then rewrite it. The visible text does not change.

```
<div class="page">
  <h3>Trail Notes</h3>
  <div class="nav">
    <div><a href="index.html">Home</a></div>
    <div><a href="gear.html">Gear</a></div>
  </div>
  <h1>Ridge Walk</h1>
  <p>Dry and clear all the way up.
</div>
```

12 CH 4 · **DEBUG**

This form is unusable with a keyboard and unusable with a screen reader. Name five faults, then rewrite it.

```
<form>
  <div>Name</div>
  <input id="name" placeholder="Name">

  <div>Email</div>
  <input id="email" placeholder="you@example.org">

  <a href="#" onclick="send()">Send</a>
</form>
```

13 CH 5 · **DEBUG**

The second rule never wins. Explain why in terms of the cascade, then rewrite both rules so that an ordinary class override works.

```
#page #main .content ul li a.link { color: #0b5; }
.link--muted { color: #666; }
```

14 CH 8 · **DEBUG**

This gallery scrolls sideways on a phone and the breakpoint does not save it. Name four problems and rewrite the stylesheet.

```
.gallery {
  width: 960px;
  display: flex;
}

.gallery img {
  width: 320px;
}

@media (max-width: 480px) {
  .gallery { display: block; }
}
```

15 CH 9 · **DEBUG**

This program throws. Name the message, name three faults in three lines, and write the corrected version so that the Console reads Total: $59.97.

```
const priceText = "19.99";
const quantity = 3;
const total = priceText * quantity;

if (total = 0) {
  console.log("Nothing to pay");
}

console.log("Total: $" + total.toFixed(2));
```

16 CH 11 · **DEBUG**

Nothing happens when the button is clicked, and one message appears in the Console. Name the message, name three faults, and rewrite the script.

```
const output = document.querySelector("#count-output");
const button = document.querySelector("#count-buton");

let count = 0;

button.addEventListener("click", function () {
  count += 1;
  output.innerHTML = "Count: " + count;
});
```

17 CH 12 · **DEBUG**

Deleting a task appears to work, and the task comes back the next time anything is rendered. Explain what is wrong with this design, then rewrite the function.

```
function removeTask(idToRemove) {
  const item = document.querySelector("#task-" + idToRemove);
  item.remove();
}
```

18 CH 13 · **DEBUG**

This is the whole of a dashboard's data loading. Name five reliability or safety problems, then rewrite it.

```
async function showDashboard() {
  const data = await fetch("data.json").then(function (r) {
    return r.json();
  });

  document.querySelector("#count").innerHTML = data.records.length;
}
```

QUESTIONS

Part 3 — Short answer (6 questions)

19 CH 1 · **EXPLAIN**

What does the defer attribute on a script element do, and why does every DOM example in this book depend on it? Describe the symptom you would see without it.

20 CH 3 · **EXPLAIN**

When is `alt=""` the right choice for an image, and when is it wrong? Describe how you decide, and say what `alt` text a photograph should carry when it is the entire content of a link.

21 CH 5 · **EXPLAIN**

Two rules set the same property on the same element. Describe, in order, what the browser considers when deciding which one wins, and say where in developer tools you can watch it happen.

22 CH 7 · **EXPLAIN**

Flexbox, Grid, and normal flow all lay out content. Say what each is best at, give one layout that each solves cleanly, and name the trap that catches beginners in both Flexbox and Grid.

23 CH 4 · **EXPLAIN**

A page validates a registration form with required, type="email", and a min attribute, and the developer concludes that the data reaching the server is safe. Explain why that conclusion is wrong, and say what browser validation is genuinely good for.

24 CH 13 · **EXPLAIN**

Why does a page that calls `fetch()` have to be served over http, when every other example in this book runs from a file you double-clicked? Describe what the Console and the Network panel show when this is the problem.

QUESTIONS

Part 4 — Write the code (6 questions)

25 CH 1 · **BUILD**

From memory, write the complete HTML document that every project in this book starts from. It has to declare its type and language, declare its text encoding, behave on a phone, carry a title, load a stylesheet named `styles.css`, and load a script named `app.js` in a way that lets the script find the elements it needs. Then name the two files that sit beside it.

26 CH 3 · **BUILD**

Write a table showing two months of rainfall in millimeters. It has to name itself, mark the column headings and the row headings as headings rather than as data, and state which cells each heading labels. Then say what a screen reader can announce because of the markup you wrote.

27 CH 8 · **BUILD**

Write the stylesheet for a page that has a centered reading column and, inside it, a gallery of cards that adds and drops columns on its own. Use predictable box sizing, a small named spacing scale, a comfortable measure, and no media query at all. Say why no media query is needed.

28 CH 10 · **BUILD**

Write `summarize(scores)`. It returns an object holding the count, the total, the average, the highest score, and the lowest score. Decide what it does with an empty array and say why you chose that. Then write the three calls that test a normal case, a single-value case, and the empty case.

29 CH 12 · **BUILD**

Write `renderEntries(entries, list)`. It empties the container and rebuilds it from the array. An empty array produces one item reading No entries yet. Every other entry shows its title and page count. A title containing angle brackets must appear as characters. Say why the function empties and rebuilds rather than adding only what changed.

30 CH 13 · **BUILD**

Write `saveLog(entries)` and `loadLog()`. Together they store an array of entry objects in the browser and restore it on the next load. The loader always returns an array, whatever it finds. List the four situations the loader has to survive, and name one kind of data that must never be stored this way.

MARK YOUR OWN PAPER

Answer Key

> Mark your predictions before you run anything. Count a predicted output correct only when the text matches exactly, spaces and punctuation included, since that precision is the skill being tested. For the box-model and grid questions, check the arithmetic rather than the description, and then confirm it in the box diagram developer tools draws for the element. Where an answer offers a choice, such as what a function returns for an empty array, either is correct as long as you can say which rule you chose and why. If a question defeats you, the chapter number beside it says where to go back to, and the Extra Practice Questions for that chapter are the next place to work.

ANSWERS

Part 1 — Predict the output (10 questions)

1 CH 2 · PREDICT

The outline is:

```
h1  Trail Notes
h2  Ridge Walk
h3  Conditions
```

It descends one level at a time, so the section sits inside the article, which sits inside the main content.

The four landmarks are header, nav, main, and footer. The main element holds the primary unique content, and there is normally one per document. The nav is inside the header here, which is common and perfectly correct; the aria-label names it, which matters as soon as a page carries more than one navigation region.

Everything in this answer can be read off the markup with the stylesheet switched off, which is the point of writing it this way.

2 CH 3 · PREDICT

1. ../`index.html` goes up out of pages and asks for site/`index.html`.
2. ../images/boots.jpg goes up and back down, asking for site/images/boots.jpg.
3. #care asks for nothing at all. A destination beginning with a number sign is a fragment: it stays on this page and scrolls to the element whose id is care.

Every relative reference is resolved from the folder holding the current document. That is why moving one file can break several links at once, and why a renamed page produces a file-not-found screen whose address bar shows you exactly what the browser built.

3 CH 5 · PREDICT

As written the paragraph is navy. An id outweighs any number of classes and element names, so the first rule wins outright and source order never comes into it.

Delete the first rule and the paragraph is olive. Now .note.important carries two classes and beats .important, which carries one.

Delete the second rule as well and the paragraph is brown. Only .important and p still match, and one class beats one element name.

Delete .important too and it falls back to the element rule, black.

Read the matching rules in order of specificity first. Source order settles a tie, and nothing else.

4 CH 6 · PREDICT

As written, under content-box:

```
Content width: 400px, because the declared width applies to the content only.
Total rendered width, excluding margin: 400 + 24 + 24 + 2 + 2, which is 452px.
Total horizontal space: 452 + 16 + 16, which is 484px.
```

With `box-sizing: border-box`:

```
Content width: 400 - 24 - 24 - 2 - 2, which is 348px.
Total rendered width, excluding margin: 400px.
Total horizontal space: 400 + 16 + 16, which is 432px.
```

Margin is never counted inside a declared width under either model. That is the single fact that makes `border-box` worth applying everywhere: the number you write is the number the box occupies, before margins.

5 CH 7 · PREDICT

Three columns, each 253.33px wide.

Work from the minimum. Four columns would need 4 x 200 plus the three gaps, 3 x 20, which is `860px`. That does not fit in 800. Three columns need 3 x 200 plus 2 x 20, which is `640px`, and that does fit. So three tracks are created, and the 1fr share divides what is left: (800 - 40) / 3, which is 253.33px each.

At `620px`, three columns would need `640px`, which no longer fits. Two columns need 2 x 200 plus 20, which is `420px`, so two tracks are created at (620 - 20) / 2, which is `300px` each.

No media query is involved. The declaration states what one card needs, and the browser decides how many fit.

6 CH 8 · PREDICT

`45rem` is 45 x 16, which is `720px`. The `min()` function takes whichever value is smaller at that moment.

```
At 400px: 100% - 3rem is 352px, which is smaller than 720, so the wrapper is 352px.
At 900px: 100% - 3rem is 852px, which is larger than 720, so 720 caps it and the wrapper is
    ↳ 720px.
At 1600px: 100% - 3rem is 1552px, so 720 caps it again and the wrapper is 720px.
```

The cap therefore takes effect at `768px` of viewport, which is where 100% - `3rem` first reaches 720.

The margin declaration centers the wrapper by splitting the leftover space evenly. Below `768px` there is no leftover space, so it has nothing to do. Above it, the leftover grows and the margin keeps the reading column in the middle of the screen. The subtraction of `3rem` is what keeps a gutter down each side on a narrow phone.

7 CH 9 · PREDICT

The output is:

```
33
9
true
```

The first line joins text. One operand is the string `"3"`, and with a string operand the + operator concatenates rather than adds.

The second line multiplies and gives 9. The * operator has no text meaning at all, so JavaScript converts the string to a number first and does the arithmetic. That inconsistency between + and * is exactly why explicit conversion is the habit worth forming.

The third line is `true`. `Number(b)` produces the number 3 before the comparison, so === compares two numbers of the same type. Written as a === b it would be `false`, since a string is never strictly equal to a number.

8 CH 10 · PREDICT

The output is:

```
Ana
Ben
Cara
found Cara
```

The loop prints each name before testing it. Ana and Ben are three characters long, so the test fails and the loop continues. Cara is four characters long, so found is set and break leaves the loop immediately. Dan is never printed, because break stops the loop rather than skipping one pass.

Remove the break and the loop runs to the end. Dan would be printed, and since Dan is three characters long the assignment would not run again, so the last line would still read found Cara. Change Dan to Dana and the same removal would change the answer, which is what makes break worth reading carefully rather than assuming.

9 CH 12 · PREDICT

The output is:

```
4
3
30
```

The map call ran while cart still held three values, and it returned a new array of three doubled values. The push that followed changed cart, adding a fourth value, and it did not touch doubled at all. So cart has length 4 and doubled has length 3.

The third line is 30, because `doubled[2]` came from `cart[2]`, which was 15.

The operation that changed an existing array is `push()`. Methods such as `push()` and `pop()` mutate; methods such as `map()` and `filter()` build something new and leave the source alone. Knowing which style you are using is what prevents accidental state changes, and it is why a filtered view can be derived safely from a source array.

10 CH 13 · PREDICT

The output is:

```
2
3
{"name":"Ana","scores":[1,2]}
```

`JSON.stringify()` produced text. Text holds no references to anything, so `JSON.parse()` had to build entirely new objects and arrays from it. The copy shares nothing with the original, nested arrays included, which is why the push reached only one of them.

Contrast that with `const copy = original`. That creates a second name for the same object, and a change through either name is visible through both.

The third line is the stored text as it stood at the moment of stringify. It never changed afterwards, because it is a string. Notice that JSON always quotes property names with double quotation marks, and that the spacing you wrote in the source is not preserved.

ANSWERS

Part 2 — Find and fix the defect (8 questions)

11 CH 2 · DEBUG

1. The heading levels are inverted. The page title is marked h3 and the article title h1, so the outline reads as a subheading containing a top-level heading. Heading level follows document structure, and not the size you happen to want.
2. There are no landmarks. Every region is a div, so nothing tells a browser or a screen reader where the banner, the navigation, the main content, or the footer begins.
3. The navigation is a set of divs rather than a nav containing a list. As written, nothing says how many links there are or where the group ends.
4. The paragraph is never closed.

The rewrite changes no text and adds no styling. What it adds is meaning, and the page now survives with its CSS switched off.

```
<header>
  <h1>Trail Notes</h1>
  <nav aria-label="Primary">
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="gear.html">Gear</a></li>
    </ul>
  </nav>
</header>
<main>
  <article>
    <h2>Ridge Walk</h2>
    <p>Dry and clear all the way up.</p>
  </article>
</main>
```

12 CH 4 · DEBUG

1. The field captions are divs. A div is not a label, so nothing associates the word Name with the control under it. Clicking the caption does nothing, and assistive technology announces an unnamed field.
2. The placeholders are being used as labels. Placeholder text disappears the moment the user types, and it is frequently too faint to read.
3. Neither input has a name attribute, so neither value would be submitted.
4. The email field has no type, so the browser applies none of its own checking and offers no email keyboard on a phone.
5. The submit control is a link with behavior written into an onclick attribute. It is not a button, the `href` goes nowhere, and the behavior belongs in the script file rather than in the markup.

The rewrite needs no JavaScript at all, and the browser supplies the keyboard behavior, the focus indicator, and the validation messages.

```
<form>
  <p>
    <label for="name">Name</label>
    <input id="name" name="name" type="text" required>
  </p>
  <p>
    <label for="email">Email</label>
    <input id="email" name="email" type="email" required>
  </p>
  <button type="submit">Send</button>
</form>
```

13 CH 5 · DEBUG

The first selector carries an id, two more ids in the ancestry, three classes, and two element names. The second carries one class. Nothing an ordinary class can do will outweigh that, so the muted color never appears.

The fault is the weight of the first rule, and not the weakness of the second. Adding !important to the second would win this argument and lose the next one, because the next override would need something stronger still. That is the spiral the chapter warns about: needing !important repeatedly is a symptom of a selector strategy that has become too hard to override.

The repair is to style the component through one low-weight class and to express the variation as a second low-weight class placed after it. Both rules then sit at the same specificity, and source order settles the matter, which is exactly what you want.

```
.link {
  color: #0b5;
}

.link--muted {
  color: #666;
}
```

14 CH 8 · DEBUG

1. The gallery is nailed to `960px`. Every viewport narrower than that scrolls sideways, and the media query at the bottom does nothing about the widths between `481px` and `959px`.
2. The images are nailed to `320px`, so they overflow their own container regardless of what the gallery width does.
3. There is no `flex-wrap`, so three `320px` images in a row demand `960px` of space and shrink or overflow rather than moving to a new line.
4. The breakpoint was chosen from a device size rather than from the content. Nothing about `480px` relates to what this gallery actually needs.

The rewrite states constraints instead of measurements. The gallery is as wide as the viewport allows, with a gutter, up to a comfortable ceiling. The images never exceed their container and keep their proportions. The track count follows the space available, so no media query is required at all.

```
.gallery {
  width: min(100% - 2rem, 60rem);
  margin-inline: auto;
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(16rem, 1fr));
  gap: 1rem;
}

.gallery img {
  max-width: 100%;
  height: auto;
}
```

15 CH 9 · **DEBUG**

The Console reads, in the browser's own words, Uncaught TypeError: Assignment to constant variable.

The three faults:

1. The condition uses a single equals sign, which assigns rather than compares. Assigning to a `const` throws, which is the message you see. Had total been a `let`, the assignment would have succeeded, the condition would have been 0, which is falsy, and the program would have carried on with a total of 0 and no message at all. That is the quieter and worse version of this bug, and it is why === is the default.
2. The multiplication relies on JavaScript converting `"19.99"` to a number on its own. It happens to work, because * has no text meaning, and the same value written with + would have concatenated instead. Convert on purpose.
3. The empty-total check is testing the wrong thing and in the wrong place. If the intent is to notice a zero total, compare with === after the arithmetic.

Once corrected the program prints Total: $59.97.

```
const priceText = "19.99";
const quantity = 3;
const price = Number(priceText);

if (!Number.isFinite(price)) {
  console.error("Invalid price.");
} else {
  const total = price * quantity;

  if (total === 0) {
    console.log("Nothing to pay");
  }

  console.log(`Total: $${total.toFixed(2)}`);
}
```

16 CH 11 · **DEBUG**

The Console reads, in the browser's own words, Uncaught TypeError: Cannot read properties of `null` (reading '`addEventListener`').

1. The selector is misspelled. There is no element with the id count-buton, so `querySelector` returned `null`, and the next line asked `null` for a method. The property named in the message is what tells you which line failed.
2. Nothing guards either element. Even with the spelling fixed, a script that assumes an element exists will throw the first time the markup changes. A short check near the top costs two lines and saves a great deal of guessing.
3. The count is written with `innerHTML`. That runs the HTML parser over a string for no reason. Displaying text is a different operation from parsing markup, and `textContent` is the default here.

The rewrite also moves the display into a small render function, so that a Reset button added later changes the state and calls the same function.

```
const output = document.querySelector("#count-output");
const button = document.querySelector("#count-button");

if (!output || !button) {
  console.error("The counter markup is missing an element.");
} else {
  let count = 0;

  function render() {
    output.textContent = `Count: ${count}`;
  }

  button.addEventListener("click", function () {
    count += 1;
    render();
  });

  render();
}
```

17 CH 12 · **DEBUG**

The array is the source of truth, and the page is only a picture of it. This function edits the picture. It removes one li element and leaves the array untouched, so the two are immediately out of step, and the next call to renderTasks rebuilds the list from the array and the task reappears.

Save at that moment and the confusion becomes permanent: the stored data still holds a task the user believes they deleted.

There is a second fault as well. Nothing checks that the element was found, so a wrong id throws on `null`.

The repair is to change the data and render again. The filter call builds a new array without the matching task, the assignment points tasks at it, and one render redraws the whole list. Nothing is ever removed from the page by hand, which is exactly why the list on screen can never disagree with the array behind it.

```
function removeTask(idToRemove) {
  tasks = tasks.filter(function (task) {
    return task.id !== idToRemove;
  });

  renderTasks(tasks);
}
```

18 CH 13 · **DEBUG**

1. The HTTP status is never checked. A completed connection is not a successful response, and a not-found page arrives as a valid response whose body is HTML. Parsing it as JSON produces a confusing message about an unexpected < character, which sends you looking in the wrong place.
2. Nothing catches a failure. An offline device or a malformed body escapes as an unhandled rejection, and the interface simply stays as it was.
3. There is no loading state and no failure state. The dashboard has three states to design, not one, and an empty region is indistinguishable from a broken one.
4. The count is written with `innerHTML`. Remote data is untrusted data, never trusted markup.
5. Neither the element nor the shape of the data is verified. If nothing matches #count the assignment throws on `null`, and if the response has no records array the length lookup throws on `undefined`.

The rewrite makes all three states visible and checks the shape of what came back before reading into it.

```
async function showDashboard() {
  const output = document.querySelector("#count");

  if (!output) {
    console.error("Missing #count element.");
    return;
  }

  output.textContent = "Loading...";

  try {
    const response = await fetch("data.json");

    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }

    const data = await response.json();

    if (!data || !Array.isArray(data.records)) {
      throw new Error("Unexpected data shape");
    }

    output.textContent = String(data.records.length);
  } catch (error) {
    output.textContent = "Could not load the data.";
    console.error("Dashboard failed:", error.message);
  }
}
```

ANSWERS

Part 3 — Short answer (6 questions)

19 CH 1 · **EXPLAIN**

The defer attribute tells the browser to carry on parsing the HTML and to run the script only once the document has been built.

Without it, a script element in the head executes the moment the parser reaches it, which is while the body is still being read. None of the elements the script looks for exist yet, so every call to document.`querySelector` returns `null`, and the first line that uses one of those results throws.

The symptom is distinctive: the script clearly ran, and every element is `null`. The Console shows a message of the form Cannot read properties of `null`, naming whatever property the failing line asked for. Nothing is wrong with the selector, and no amount of correcting the spelling will help.

That is why the pattern in this book is one line, used everywhere: a script element in the head, carrying `src` and defer.

20 CH 3 · EXPLAIN

An empty `alt` is right when the image contributes no information: a decorative divider, a background flourish, an icon that repeats the text beside it. The empty value tells assistive technology to skip the image rather than announce a file name, which is what happens when the attribute is left off altogether.

It is wrong whenever the image carries information the page would lose without it. A chart, a photograph the text refers to, a diagram, or an image of text all need alternative text that communicates what the image contributes.

The decision is contextual, not a property of the picture. The same photograph may be decorative in one place and informative in another, so ask what the reader would lose if the image did not load.

When the image is the whole content of a link, its `alt` text describes where the link goes, and not what the picture looks like. A photograph of a team linking to the team page carries `alt`="Meet the team", because that is what the reader needs in order to decide whether to follow it.

21 CH 5 · EXPLAIN

The browser considers origin and importance first, then specificity, then source order.

Origin and importance decide between the browser's own defaults, the user's settings, and your stylesheet, and this is where ! important changes the ranking. For ordinary authored CSS it rarely matters.

Specificity is the weight of the selector. An id outweighs any number of classes, and a class outweighs any number of element names. Two classes beat one class.

Source order settles a tie, and only a tie. When two selectors carry the same weight, the later rule wins.

Inheritance is not part of that contest. It supplies a value only when no declaration wins on the element itself, by taking the parent's value, which is why any declaration on the element, however weak, beats an inherited one.

You can watch the whole decision in the Styles panel. Select the element and read the rules from the top: the winning declarations are plain and the losing ones are crossed out, in the order the browser considered them.

22 CH 7 · EXPLAIN

Normal flow is the baseline, and it is already useful. Block elements stack, inline content runs along the line, and text wraps. A page of headings and paragraphs needs nothing else, and reaching for a layout system merely because it exists costs you code and gains you nothing.

Flexbox arranges items along one axis and aligns them across the other. It solves a navigation bar cleanly: a row of links with a consistent gap, vertically centered, with one link pushed to the far end by an auto margin.

Grid defines tracks in two dimensions. It solves a card gallery cleanly: repeat(auto-fit, `minmax(220px, 1fr)`) fits as many columns as the space allows and drops one as the container narrows, with no media query at all.

The trap in both is the same. They lay out the direct children of the container, and nothing deeper. A grid applied to a wrapper does not arrange the elements inside each card; those need a layout context of their own. That is why a page often uses Grid for its major tracks and Flexbox inside each card.

23 CH 4 · EXPLAIN

Browser validation runs in the user's browser, and the user controls their browser completely. The attributes can be edited in developer tools, the request can be sent without the page at all, and a script can be disabled outright. Anything the browser checks can be bypassed by anyone who wants to, which means it can never be a security boundary.

The server has to validate every rule that matters, all over again, because the server is the only place the user does not control.

What browser validation is genuinely good for is the user experience. It reports a problem instantly, beside the field, without a round trip. It moves focus to the control that needs attention. It offers the right keyboard on a phone, because type=`"email"` and type=`"number"` tell the device what kind of value is wanted. It costs one attribute and it catches the great majority of honest mistakes.

Use it everywhere it fits. Just do not confuse a convenience with a guarantee.

24 CH 13 · **EXPLAIN**

A page opened straight from disk has an address beginning with file:///, and such a page has no origin. The rules that govern which documents may read which resources are written in terms of origins, so a request from a page that has none is refused before it is made.

The Console reports it as a cross-origin failure, and the wording differs between browsers: some say the request was blocked by the cross-origin policy and that only certain schemes are supported, others say a cross-origin request was blocked because the request was not made over http. The distinctive part is the Network panel, which usually shows no request at all, because nothing was ever sent.

The fix is to serve the folder over http and open the served address rather than the file. Nothing about the code changes.

This is the one exception in the whole book. Everything else, including `localStorage`, works from a file you opened by double-clicking it.

ANSWERS

Part 4 — Write the code (6 questions)

25 CH 1 · **BUILD**

Two lines are worth stating explicitly. The charset declaration says the file is UTF-8 text, so accented letters and symbols appear exactly as typed. The viewport line tells a phone to use its real width instead of pretending to be a desktop screen, without which every responsive rule you write later is ignored.

The defer attribute on the script element is what makes the DOM chapters work. Without it the script runs while the body is still being parsed, and every selector returns `null`.

Both paths are relative to this file, so the two files that sit beside it are `styles.css` and `app.js`, in the same folder, spelled exactly as the page spells them. A misspelled `href` produces no Console message at all; the evidence is a failed request in the Network panel.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Project</title>
  <link rel="stylesheet" href="styles.css">
  <script src="app.js" defer></script>
</head>
<body>
  <h1>Project</h1>
</body>
</html>
```

The caption names the table and is read before the data, so someone who cannot see the layout knows what they are about to hear. It belongs first, inside the table element.

The head row uses th with scope="col", which says each heading labels the cells below it. The first cell of each body row uses th with scope="row", which says it labels the cells beside it.

Because of that markup, a screen reader can announce the value 84 as April, Rainfall, 84 rather than as a bare number. That association is the whole reason to write a data table properly, and it is what a table built from td cells alone cannot provide.

Keep in mind that a table is for genuinely tabular data. Using one to position content on a page gives assistive technology a set of relationships that do not exist.

```
<table>
  <caption>Rainfall in millimeters</caption>
  <thead>
    <tr>
      <th scope="col">Month</th>
      <th scope="col">Rainfall</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">April</th>
      <td>84</td>
    </tr>
    <tr>
      <th scope="row">May</th>
      <td>61</td>
    </tr>
  </tbody>
</table>
```

No media query is needed because nothing here requires a conditional decision. Every rule adapts continuously.

The `min()` call gives the page a fluid width with a gutter and a ceiling, so it fills a phone and stops growing on a wide monitor. The auto-fit and minmax pair states what one card needs and lets the browser work out how many fit, dropping to a single column below the width of two cards. The image rule keeps pictures inside their container whatever size they were saved at.

A media query earns its place when the content genuinely calls for a different arrangement, such as moving a sidebar from below the article to beside it. Save it for that. Fluid sizing solves a great many responsive problems on its own, and it solves them at every width rather than at two.

The spacing scale is defined once on :root, so widening the medium step moves every card, every gap, and every section together.

```
:root {
  --space-s: 0.5rem;
  --space-m: 1rem;
  --space-l: 2rem;
}

*, *::before, *::after {
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: system-ui, sans-serif;
  line-height: 1.5;
}

.page {
  width: min(100% - var(--space-l), 65rem);
  margin-inline: auto;
  padding: var(--space-l) 0;
}

.reading {
  max-width: 65ch;
}

.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(16rem, 1fr));
  gap: var(--space-m);
}

.card {
  padding: var(--space-m);
  border: 1px solid #ccc;
}

.card img {
  max-width: 100%;
  height: auto;
}
```

28 CH 10 · **BUILD**

The empty array is the decision, and not the arithmetic. Dividing by zero would give `NaN`, and an average of `NaN` inside an otherwise sensible object is the kind of value that travels a long way before anything notices. Returning zeros with a count of 0 is one honest answer, and returning `null` for the average is another. What matters is that you chose, and that the caller can tell which.

I return zeros here, because every caller can display a zero and no caller has to test for a missing property.

The function takes its data as a parameter and returns a value, with no dependence on anything outside it. That is what lets you test it from the Console before any page exists, and it is what lets the same function later serve a button click, a form submission, and a render function without a rewrite.

The three calls produce, in order: a count of 3 with a total of 243 and an average of 81; a count of 1 where the highest and the lowest are the same value; and a count of 0 with no division performed.

```
function summarize(scores) {
  if (scores.length === 0) {
    return { count: 0, total: 0, average: 0, highest: 0, lowest: 0 };
  }

  let total = 0;
  let highest = scores[0];
  let lowest = scores[0];

  for (const score of scores) {
    total += score;

    if (score > highest) {
      highest = score;
    }

    if (score < lowest) {
      lowest = score;
    }
  }

  return {
    count: scores.length,
    total: total,
    average: total / scores.length,
    highest: highest,
    lowest: lowest
  };
}

console.log(summarize([70, 82, 91]));
console.log(summarize([64]));
console.log(summarize([]));
```

29 CH 12 · **BUILD**

Emptying and rebuilding sounds wasteful, and for a few hundred items the browser will not notice. What it buys you is the one guarantee that matters: the list on screen can never disagree with the array behind it. Add only what changed and every path through the program has to remember to update the display, and the third one you write will forget.

The call to replaceChildren with no arguments empties the container in one line. Each item is then created with `createElement` and placed with append.

The titles come from the user, so they are set with `textContent`. Any angle brackets the user typed are displayed as characters rather than parsed as markup. Building the same list as an HTML string would hand the decision about what the document contains to whoever typed the title.

The empty case returns early, which keeps the two situations visibly separate and stops the loop from running over nothing.

```
function renderEntries(entries, list) {
  list.replaceChildren();

  if (entries.length === 0) {
    const empty = document.createElement("li");
    empty.textContent = "No entries yet.";
    list.append(empty);
    return;
  }

  for (const entry of entries) {
    const item = document.createElement("li");
    item.textContent = `${entry.title} — ${entry.pages} pages`;
    list.append(item);
  }
}
```

30 CH 13 · **BUILD**

The four situations:

1. The key has never been written. getItem returns `null` and the loader returns an empty array.
2. The key holds valid JSON for an array. It parses and the array is returned.
3. The key holds text that is not JSON at all, because it was edited by hand or damaged. `JSON.parse` throws, catch reports it, and the loader returns an empty array.
4. The key holds valid JSON of the wrong shape, such as a single object left over from an earlier version of the program. Nothing throws, so the type has to be checked, and Array.isArray is what does it.

The fourth is the one people leave out, and it produces the strangest failures later, because the value looks fine right up until something tries to loop over it.

What must never be stored this way: passwords, authentication tokens, API secrets, and confidential personal data. Browser storage is convenient, not private. Everything in it stays in one browser on one device, and anyone using that device can read it.

```
function saveLog(entries) {
  localStorage.setItem("reading-log", JSON.stringify(entries));
}

function loadLog() {
  const saved = localStorage.getItem("reading-log");

  if (!saved) {
    return [];
  }

  try {
    const value = JSON.parse(saved);
    return Array.isArray(value) ? value : [];
  } catch (error) {
    console.error("Stored data is invalid:", error.message);
    return [];
  }
}
```