

LU07: Debugging - Fehler finden und verstehen

Debugging bedeutet, den Browser (oder den Editor) zu **befragen** – mit den richtigen Werkzeugen, in der richtigen Reihenfolge.



Ein Fehler, den Sie nicht reproduzieren können, ist schwer zu beheben. Beschreiben Sie den Fehler zuerst genau – nicht „es funktioniert nicht,,“, sondern „wenn ich auf den Button klicke, passiert X statt Y“.

Warum Debugging wichtig ist

Viele Anfänger raten, was falsch sein könnte. Debugging ist das Gegenteil: Sie **prüfen Fakten**.

Eine gute Debugging-Gewohnheit folgt immer dieser Reihenfolge:

| Schritt | Frage |
|---------------------|---|
| 1. Read | Lesen Sie die Fehlermeldung genau. Was steht da wirklich? |
| 2. Reproduce | Können Sie den Fehler wiederholt auslösen? Unter welchen Bedingungen? |
| 3. Reduce | Welches ist der kleinste mögliche Code, der das Problem zeigt? |
| 4. Fix | Ändern Sie eine Sache – dann testen Sie sofort. |

Oft passiert beim Debugging gar nichts: keine Fehlermeldung, kein Hinweis. Das macht es schwierig. Deshalb brauchen Sie die richtigen Werkzeuge.

Die Debugging-Reihenfolge

Nicht jedes Werkzeug hilft bei jedem Fehler. Gehen Sie diese Reihenfolge durch:

| Schritt | Werkzeug | Hilft bei |
|---------|-----------------------|--|
| 1 | Formatter / Linter | Syntaxfehler (fehlende Klammern, Tippfehler, falsch geschachteltes HTML) |
| 2 | Console / Breakpoints | Laufzeitfehler (was passiert, während der Code läuft) |
| 3 | Elements-Tab | Rendering-Fehler (was macht der Browser aus Ihrem CSS/HTML) |

Werkzeug 1: Formatter & Linter

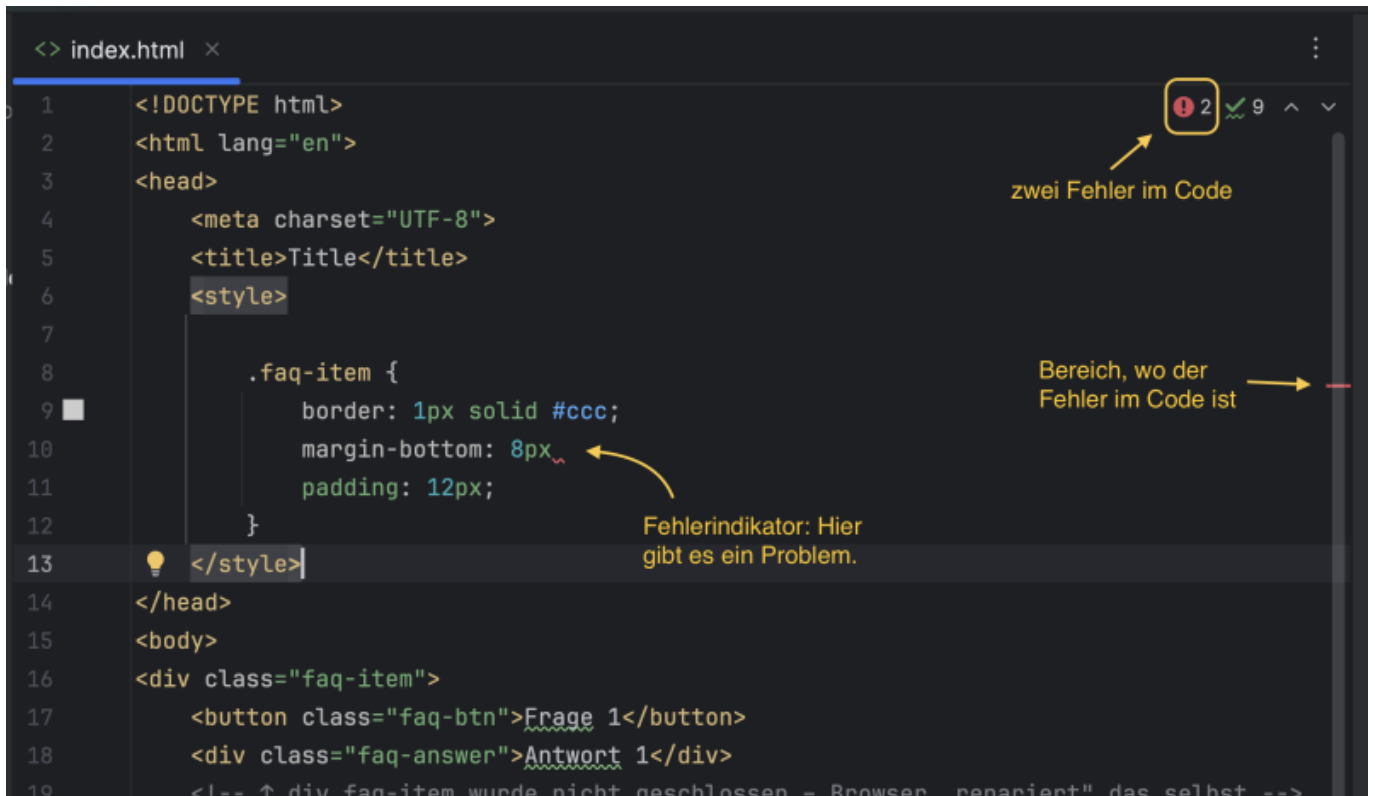
Bevor Sie die DevTools öffnen: Schauen Sie in den **Code-Editor**. Viele Fehler können gefunden werden, bevor der Code überhaupt im Browser läuft.

WebStorm

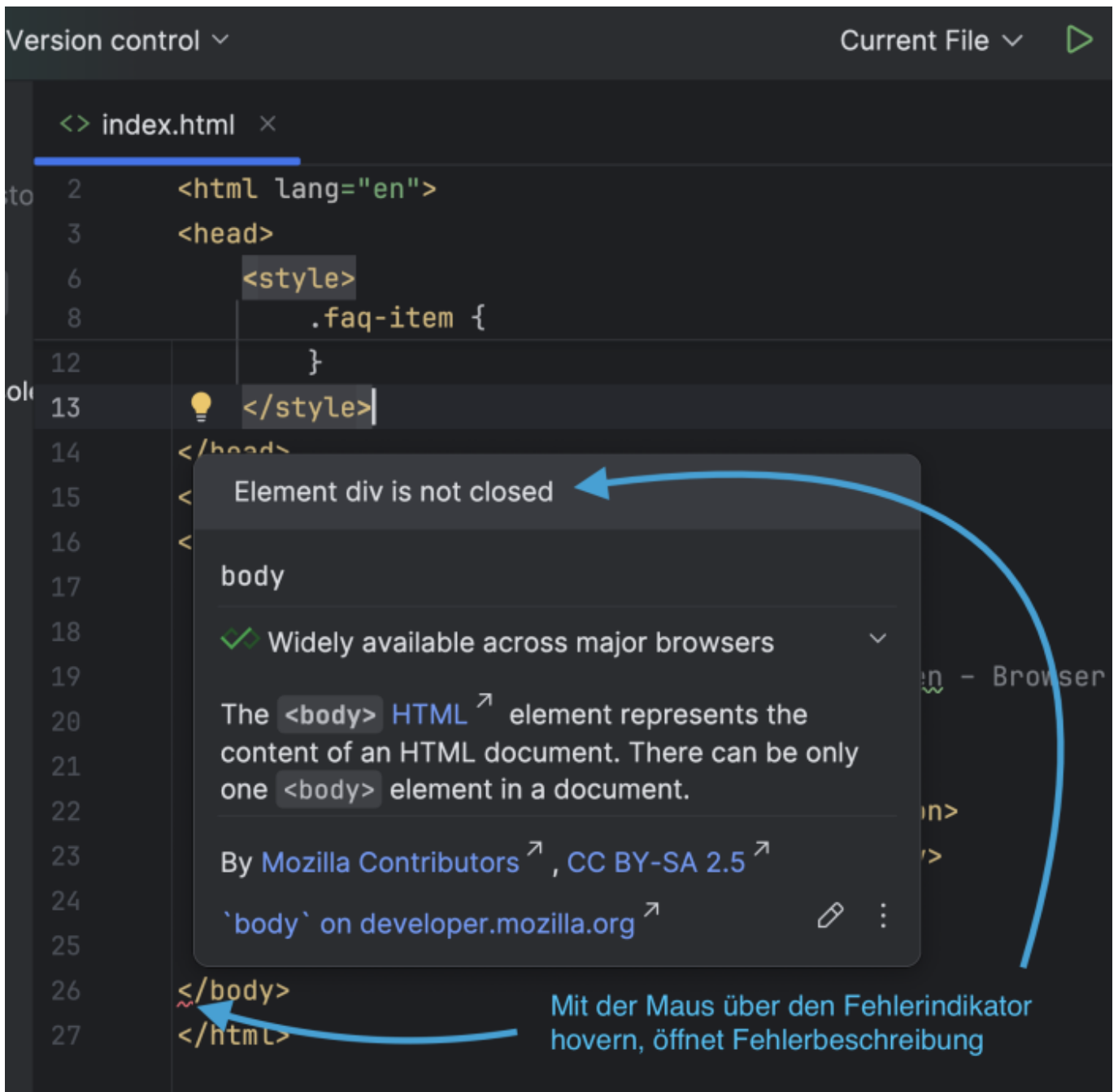


WebStorm prüft den Code automatisch im Hintergrund – ohne zusätzliche Extensions. Farbige Unterwellungen bedeuten:

- **Rot** → Fehler (der Code wird nicht korrekt ausgeführt)
- **Gelb** → Warnung (sollte behoben werden)
- **Grau** → Hinweis (Stil-Empfehlung)



WebStorm Code mit Fehleranzeige.



Fahren Sie mit der Maus über die Markierung - Sie sehen eine Erklärung des Problems.

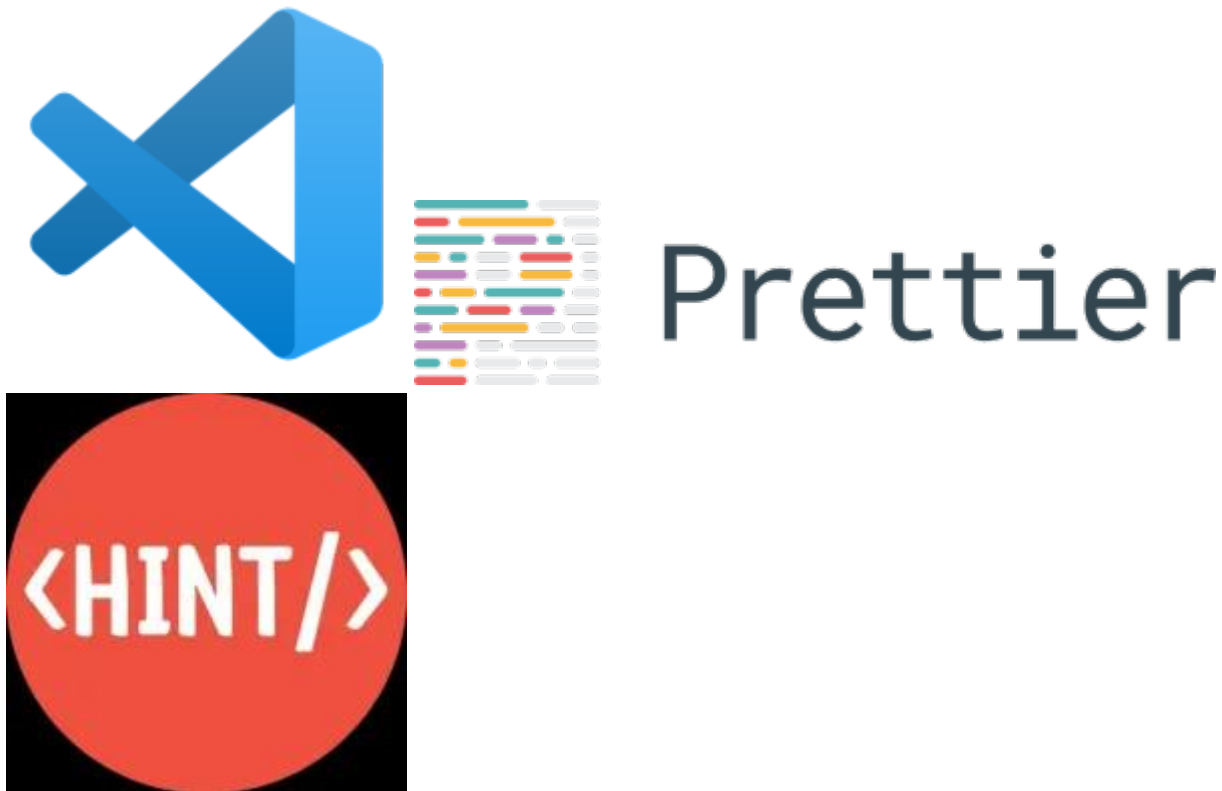
WebStorm erkennt unter anderem:

- Fehlende Dateieindungen: `<link href=„style“>` → *Cannot resolve file 'style'*
- Falsche Anführungszeichen: `<link href=„style.css„>` → *Cannot resolve file '„style.css“'*
- Tippfehler in CSS-Eigenschaften: `heihgt` → unbekannte Eigenschaft

Formatieren

Mit **Ctrl+Alt+L** (Windows) / **Cmd+Alt+L** (Mac) formatieren Sie die ganze Datei (alles wird schön ausgerichtet).

VS Code: Prettier + HTMLHint



Für VS Code brauchen Sie zwei separate Extensions, um dasselbe Niveau wie WebStorm zu erreichen:

Prettier formatiert den Code automatisch beim Speichern (Einrückung, Strukturierung):

1. Extensions-Tab öffnen (Ctrl+Shift+X / Cmd+Shift+X)
2. Suchen: **Prettier - Code formatter**, installieren
3. Settings öffnen (Ctrl+, / Cmd+,), suchen nach `format on save`, Checkbox aktivieren

Ab jetzt: **Ctrl+S** / **Cmd+S** → Code wird automatisch aufgeräumt und eingerückt.



Prettier erkennt **keine** fehlenden Dateiendungen und **keine** falschen Anführungszeichen. Dafür brauchen Sie HTMLHint.

HTMLHint prüft die HTML-Struktur und meldet typische Fehler wie falsche Anführungszeichen:

1. Suchen: **HTMLHint**, installieren
2. Keine weitere Konfiguration nötig - funktioniert sofort
3. Website mit Dokumentation aller Regeln: htmlhint.com



Hinweis für Fortgeschrittene: Für CSS gibt es [Stylelint](#) - ein Linter, der ähnlich wie HTMLHint funktioniert, aber für CSS-Dateien. Tippfehler in CSS-Eigenschaften wie `heihgt` werden damit sofort markiert. Im Moment reicht der Blick in den Editor - Stylelint ist ein nächster Schritt.

Typische Fehler, die diese Tools erkennen

Fehlende Dateieindung (WebStorm & HTMLHint, nicht Prettier):

```
<!-- ☐ Dateieindung fehlt – Browser findet die Datei nicht,
kein Console-Fehler -->
<link href="style" rel="stylesheet">
<script src="script"></script>

<!-- ☐ Korrekt -->
<link href="style.css" rel="stylesheet">
<script src="script.js"></script>
```

Falsche Anführungszeichen (WebStorm & HTMLHint, nicht Prettier):

```
<!-- ☐ Typografische Anführungszeichen aus Word oder Moodle
– Browser lädt die Datei nicht -->
<link href=„style.css“ rel=„stylesheet“>

<!-- ☐ Gerade Anführungszeichen -->
<link href="style.css" rel="stylesheet">
```



Vorsicht beim Kopieren aus Moodle oder Word:

Textverarbeitungsprogramme ersetzen gerade Anführungszeichen („ „) durch typografische („ “). Im Browser-Code führt das zu stillen Fehlern.

Fügen Sie kopierten Code immer als **Plain Text** ein: **Cmd+Shift+V** (Mac) / **Ctrl+Shift+V** (Windows).

Tipfehler in CSS-Eigenschaften (WebStorm & Stylelint, nicht Prettier):

```
/* ☐ 'heihgt' statt 'height' – Browser ignoriert diese Zeile
einfach, kein Console-Fehler */
.panel.open {
  heihgt: auto;
  opacity: 1;
}

/* ☐ Korrekt */
.panel.open {
```

```
height: auto;  
opacity: 1;  
}
```

Dieser Fehler ist besonders tückisch: das Accordion-Panel öffnet sich visuell nicht, obwohl das JavaScript korrekt funktioniert. Die Console zeigt keinen Fehler.

Werkzeug 2: Die Browser-Console

Die Console ist der direkte Draht zum laufenden JavaScript-Code. Öffnen Sie sie mit **F12** → **Console**.



Öffnen Sie die Console, **bevor** Sie anfangen zu raten. Die Fehlermeldung sagt Ihnen meistens genau, was falsch ist - und in welcher Zeile.

Fehlermeldungen lesen

Zeigt wo der Fehler auftaucht:
- File: script.js
- Zeile: 5

Beschreibt den Fehler ("e" wurde nicht gefunden) und Error-Typ (ReferenceError)

Fehlermeldungen sehen einschüchternd aus, bedeuten aber meist eines von wenigen Dingen:

| Fehlermeldung | Bedeutung | Häufige Ursache |
|---|---|---|
| TypeError: Cannot read properties of null | Sie greifen auf ein Element zu, das nicht existiert | Falscher Selector oder Script im <head> statt vor </body> |
| ReferenceError: x is not defined | Variable x existiert an dieser Stelle nicht | Tippfehler im Variablennamen, e im Callback vergessen |
| ReferenceError: Cannot access 'x' before initialization | Variable wird vor ihrer Deklaration benutzt | let / const zu weit unten im Code |
| addEventListener is not a function | Das Objekt ist kein DOM-Element | querySelector hat null zurückgegeben |

Die Fehlermeldung enthält immer eine **Zeilennummer** – klicken Sie darauf, um direkt zur betroffenen Stelle zu springen.

console.log() richtig einsetzen

console.log() schreibt Werte in die Console. So sehen Sie, was Ihr Code gerade „denkt“.

```
// ☐ Wenig hilfreich – was ist das genau?  
console.log(buttons);  
  
// ☐ Mit Label – sofort lesbar  
console.log('buttons:', buttons);  
console.log('Anzahl gefundene Buttons:', buttons.length);  
  
// ☐ Fehler bewusst kennzeichnen (erscheint rot in der  
Console)  
console.error('Kein Element gefunden für Selector:',  
selector);
```

Read → Reproduce → Reduce: Ein konkretes Beispiel

Hier ein typischer Fehler aus dem Accordion-Projekt:

```
// ☐ Fehler: 'e' wurde im Callback nicht deklariert  
buttons.forEach((button) => {  
  button.addEventListener('click', () => { // ← 'e' fehlt  
    als Parameter  
    const panelElement = e.target.nextElementSibling;  
  });  
});
```

```
// ...
});
});
```

Zeigt wo der Fehler auftaucht:
- File: script.js
- Zeile: 5

Beschreibt den Fehler ("e" wurde nicht gefunden) und Error-Typ (ReferenceError)

Was passiert:

- **Read:** Console zeigt Uncaught ReferenceError: e is not defined
- **Reproduce:** Jedes Mal beim Klick auf einen Button
- **Reduce:** Das Problem liegt in Zeile mit `e.target` - `e` ist nirgends definiert
- **Fix:** Den Parameter `e` in der Klammer (`e`) ergänzen

```
// ☐ Korrekt: 'e' als Parameter deklariert
buttons.forEach((button) => {
  button.addEventListener('click', (e) => { // ← 'e' als
    Parameter
    const panelElement = e.target.nextElementSibling;
    // ...
  });
});
```

Häufige JS-Bugs aus dem Accordion-Projekt

Bug: `querySelector` statt `querySelectorAll`

```
// ☐ Wählt nur den ersten Button – forEach funktioniert,
aber nur 1 Element reagiert
```

```
const buttons = document.querySelector('.accordion-btn');  
  
// ☐ Wählt alle Buttons  
const buttons = document.querySelectorAll('.accordion-btn');
```

Bug: Fehlender Punkt vor dem Klassennamen

```
// ☐ Sucht nach einem HTML-Tag <accordion-btn> – findet  
nichts, gibt leere NodeList zurück  
const buttons = document.querySelectorAll('accordion-btn');  
  
// ☐ Punkt vor Klassennamen nicht vergessen  
const buttons = document.querySelectorAll('.accordion-btn');
```

Dieser Fehler erzeugt **keinen roten Fehler** in der Console. Prüfen Sie mit:

```
console.log('Gefundene Buttons:', buttons.length); // → 0  
zeigt das Problem
```

Bug: Variablenname-Konflikt in forEach

```
// ☐ Aussen heisst die Variable 'button' (singular) – innen  
im forEach auch 'button'  
// → der äussere 'button' wird überschrieben, setAttribute  
betrifft immer den letzten  
button.forEach((button) => {  
  button.addEventListener('click', (e) => {  
    // ...  
    if (!panelIsOpen) {  
      panelElement.classList.add('open');  
      button.setAttribute('aria-expanded', 'true'); // ←  
welches 'button' ist das jetzt?  
    }  
  });  
});  
  
// ☐ Klare, unterschiedliche Namen  
buttons.forEach((button) => {  
  button.addEventListener('click', (e) => {  
    // ...  
    if (!panelIsOpen) {
```

```
    panelElement.classList.add('open');
    button.setAttribute('aria-expanded', 'true'); // ←
    eindeutig: der geklickte Button
  }
});
});
```

Bug: Doppeltes classList.add - Panel lässt sich nicht schliessen

```
// □ classList.add('open') steht doppelt – einmal
ausserhalb, einmal im if-Block
// → Panel wird immer geöffnet, nie geschlossen
buttons.forEach((button) => {
  button.addEventListener('click', (e) => {
    const panelElement = e.target.nextElementSibling;
    const panelIsOpen =
panelElement.classList.contains('open');

    buttons.forEach((andererButton) => {
andererButton.nextElementSibling.classList.remove('open');
andererButton.setAttribute('aria-expanded', 'false');
    });

    panelElement.classList.add('open'); // ← diese Zeile
wurde vergessen zu löschen

    if (!panelIsOpen) {
      panelElement.classList.add('open');
      button.setAttribute('aria-expanded', 'true');
    }
  });
});

// □ Nur innerhalb des if-Blocks
buttons.forEach((button) => {
  button.addEventListener('click', (e) => {
    const panelElement = e.target.nextElementSibling;
    const panelIsOpen =
panelElement.classList.contains('open');

    buttons.forEach((andererButton) => {
andererButton.nextElementSibling.classList.remove('open');
andererButton.setAttribute('aria-expanded', 'false');
    });

    if (!panelIsOpen) {
      panelElement.classList.add('open');
      button.setAttribute('aria-expanded', 'true');
    }
  });
});
```

```
    }  
  });  
});
```

Bug: Syntaxfehler in forEach - fehlende öffnende Klammer

```
// ❌ Syntaxfehler: '(' fehlt vor 'andererButton'  
// → Console: Uncaught SyntaxError: Unexpected token '=>  
buttons.forEach(andererButton) => {  
  andererButton.nextElementSibling.classList.remove('open');  
});  
  
// ✅ Korrekt  
buttons.forEach((andererButton) => {  
  andererButton.nextElementSibling.classList.remove('open');  
});
```

Script-Position: Ein stiller, häufiger Fehler

```
<!-- ❌ Script im <head> – HTML ist noch nicht geladen wenn JS läuft -->  
<!-- → document.querySelectorAll('.accordion-btn') gibt null zurück -->  
<!DOCTYPE html>  
<html>  
<head>  
  <script src="script.js"></script> <!-- zu früh! -->  
</head>  
<body>  
  <button class="accordion-btn">Frage 1</button>  
</body>  
</html>  
  
<!-- ✅ Script am Ende von <body> – HTML ist vollständig geladen -->  
<!DOCTYPE html>  
<html>  
<head>  
  <link href="style.css" rel="stylesheet">  
</head>  
<body>  
  <button class="accordion-btn">Frage 1</button>
```

```
<script src="script.js"></script> <!-- ← hier -->
</body>
</html>
```

Wenn `document.querySelector(...)` null zurückgibt, obwohl das Element im HTML vorhanden ist, ist dies die häufigste Ursache.

Werkzeug 3: Der Elements-Tab

Der Elements-Tab zeigt, was der Browser **tatsächlich** aus Ihrem Code gemacht hat – nicht was Sie geschrieben haben, sondern was nach dem Parsen und Rendern übrig bleibt.

CSS live ausprobieren

Workflow:

1. Elements-Tab öffnen (F12 → Elements)
2. Rechts: Styles-Panel
3. Eigenschaft anklicken und Wert ändern
4. **Sofortiges Feedback** – ohne die Datei zu speichern

Wenn der Look stimmt: dann erst in die CSS-Datei übertragen.



Deaktivierte CSS-Regeln (durchgestrichen) bedeuten: eine andere Regel mit höherer Spezifität gewinnt. Der **Computed-Tab** zeigt den tatsächlich angewendeten Wert.

Der Computed-Tab

Wenn `height: auto` im CSS steht, aber die Transition nicht funktioniert:
→ Computed-Tab öffnen → dort steht der **berechnete Pixelwert**.

Das erklärt, warum `transition: height auto` nicht direkt funktioniert – der Browser kennt den Zielwert nicht. (Lösung: `interpolate-size: allow-keywords;`)

Breakpoints setzen

Manchmal reicht `console.log` nicht aus. Breakpoints stoppen den Code an einer bestimmten Stelle und ermöglichen es, den Zustand genau zu untersuchen.

So geht's:

1. DevTools öffnen → Tab **Sources**
2. Die JS-Datei links in der Dateiliste anklicken
3. Zeilennummer links anklicken → blauer Punkt = Breakpoint gesetzt
4. Seite neu laden oder Aktion ausführen → Code stoppt an dieser Stelle
5. Rechts sehen Sie alle aktuellen Variablenwerte

Navigieren:

- **Step over** (F10): Nächste Zeile ausführen
- **Step into** (F11): In eine Funktion hineinsteigen
- **Resume** (F8): Weiter bis zum nächsten Breakpoint



Für den Anfang reicht: Breakpoint setzen, einmal F10 drücken, Variablenwert rechts ablesen.
Den Rest lernen Sie automatisch, wenn Sie ihn brauchen.

Das Debugging-Protokoll

Dieses Format hilft, strukturiert vorzugehen – und aus Fehlern zu lernen. Füllen Sie es aus, wenn Sie einen Bug gefunden haben:

| Schritt | Frage | Beispiel |
|----------------------|------------------------------|---|
| 1. Beobachten | Was passiert genau? | „Wenn ich auf Frage 2 klicke, passiert gar nichts,“ |
| 2. Hypothese | Was könnte die Ursache sein? | „querySelector findet den Button vielleicht nicht“ |
| 3. Testen | Wie prüfe ich das? | <code>console.log('Buttons:', buttons.length)</code> |
| 4. Fazit | Was habe ich gelernt? | „buttons.length war 0 – Punkt vor Klassennamen fehlte,“ |

Übungsaufgabe: Bugs im Accordion finden

Der folgende Code enthält **4 absichtliche Bugs** – alle stammen aus echten Fehlern aus dem Unterricht. Öffnen Sie DevTools und finden Sie die Fehler, ohne den Code anzustarren. Benutzen Sie Console, Elements-Tab oder Breakpoints.

HTML-Datei:

```
<!DOCTYPE html>  
<html>
```

```
<head>
  <script src="script.js"></script>
  <link href="style" rel="stylesheet">
</head>
<body>
  <div class="accordion">
    <button class="accordion-btn">Frage 1</button>
    <div class="panel">Antwort 1</div>
    <button class="accordion-btn">Frage 2</button>
    <div class="panel">Antwort 2</div>
  </div>
</body>
</html>
```

JavaScript-Datei:

```
const buttons = document.querySelectorAll('accordion-btn');

buttons.forEach((button) => {
  button.addEventListener('click', () => {
    const panelElement = e.target.nextElementSibling;
    panelElement.classList.toggle('open');
  });
});
```

Tipps:

- Bug 1: Erzeugt keinen Fehler - aber die Styles werden nicht geladen
- Bug 2: Erzeugt keinen Fehler - aber `buttons.length` verrät ihn
- Bug 3: `TypeError: Cannot read properties of null` - Zusammenhang mit der Script-Position
- Bug 4: `ReferenceError: e is not defined`



Schreiben Sie für jeden gefundenen Bug das **Debugging-Protokoll** aus: Was haben Sie beobachtet? Was war Ihre Hypothese? Wie haben Sie es geprüft?

Zusammenfassung

| Werkzeug | Wann einsetzen |
|----------------------|---|
| Formatter / Prettier | Zuerst immer - Syntaxfehler, Tippfehler in CSS-Eigenschaften, falsche Anführungszeichen |

| Werkzeug | Wann einsetzen |
|--------------|---|
| Console | Wenn etwas passiert oder nicht passiert - Fehlermeldung lesen, <code>console.log()</code> einsetzen |
| Elements-Tab | Wenn das Layout falsch aussieht - CSS live anpassen, Computed-Tab prüfen |
| Breakpoints | Wenn <code>console.log</code> nicht ausreicht - Code anhalten, Variablenwerte prüfen |

From:
<https://wiki.bzz.ch/> - BZZ - Modulwiki

Permanent link:
https://wiki.bzz.ch/de/modul/m291/learningunits/lu07/theorie/lu07a_debugging?rev=1774190564

Last update: 2026/03/22 15:42

