WeakMap:

- Keeps metadata about a specific object (form) without exposing it publicly.
- Automatically deletes the data when the form is removed from memory (no memory leaks).
- Great for tracking state like validation or timestamps without modifying the object itself.
- Avoids polluting form with custom properties like form.validated = true.

Type Guards:

- These check that an element is **exactly the type** you're expecting.
- Prevents errors like calling .value on something that's not an <input>.
- Helps make your code more robust, especially in dynamic or JS-heavy apps.

Proxy with Reflect:

- Intercepts reads and writes to any property of the inputs object.
- You can add logging, validation, transformations, or even block operations.
- Great for:
    - Debugging: "Who accessed the phone field?"
    - Tracking changes: "How many times was zip edited?"
    - Reactivity systems (like Vue or MobX use Proxies under the hood)

Part One: Contact Page

This script:

- Validates a form with phone, email, and ZIP fields.
- Uses type guards for safety.
- Uses a Proxy to log access to input values.
- Uses WeakMap to track whether the form has passed validation.
- Sends the form data to a server if it's valid.
- Displays error or success messages.

1. WeakMap to store validation status privately:

```
const formMeta = new WeakMap();
```

WeakMap is like a Map, but keys are objects (e.g. form) and are garbage-collected when not in use.

- This is used to track if the form was validated (without attaching data directly to the form).

2. Type guards for form and input elements:

```
function isForm(el) {
  return el instanceof HTMLFormElement;
}
function isInput(el) {
  return el instanceof HTMLInputElement;
}
```

- Ensures you're working with the correct kind of DOM element.
- Helpful for safety and debugging.

3. Utility to safely get trimmed input values:

```
function getInputValue(id) {
  const el = document.getElementById(id);
  if (!isInput(el)) throw new Error(`Invalid input: ${id}`);
  return el.value.trim();
}
```

- Gets a form field by its id, ensures it's an input, and returns its trimmed value.
-
- If the field doesn't exist or is the wrong type, it throws a clear error.

4. Proxy wrapper to log property access:

```
function createFormProxy(obj) {
  return new Proxy(obj, {
    get(target, prop) {
      const value = Reflect.get(target, prop);
      console.log(`Accessed ${String(prop)}:`, value);
      return value;
    },
    set(target, prop, value) {
      console.log(`Set ${String(prop)} = ${value}`);
      return Reflect.set(target, prop, value);
    }
  });
}
```

- Wraps the inputs object in a Proxy.
- Every time you read (inputs.phone) or write to it, it logs to the console.
- Reflect makes this behavior clean and future-safe.

5. DOM ready logic (runs after the page is fully loaded):

```
document.addEventListener('DOMContentLoaded', () => {
```

- Everything inside here waits for the full DOM to be loaded before running.

6. Grab the form and result message area:

```
const form = document.getElementById('myForm');
 const responseDiv = document.getElementById('response');
```

- Assumes you have a <form id="myForm"> and a in your HTML.

7. Confirm the form and response div are found:

```
 if (!isForm(form) || !responseDiv) {
   console.error('Missing form or response div');
   return;
 }
```

- Ensures you don't proceed unless the necessary elements exist.

8. Set up the submit event handler:

```
form.addEventListener('submit', async (e) => {
   e.preventDefault();
```

- Prevents the form from actually submitting via browser default.
- Instead, we handle everything manually in JS.

9. Store "not validated" in the WeakMap for this form:

```
    formMeta.set(form, { validated: false });
```

- Later, we'll overwrite it with { validated: true } if everything passes.

10. Wrap the user input data in a Proxy:

```
  const inputs = createFormProxy({
     phone: getInputValue('phone'),
     email: getInputValue('email'),
     zip: getInputValue('zip')
   });
```

- This is the main data object we'll validate and submit.
- It's wrapped in a Proxy, so accessing properties will log to the console.

11. Run validation on inputs:

```
    const errors = [];

    if (!/^\d{10}$/.test(inputs.phone)) errors.push('Phone must be 10
digits.');
    if (!/^[\w.-]+@[\w.-]+\.[a-zA-Z]{2,}$/.test(inputs.email))
errors.push('Invalid email format.');
    if (!/^\d{5}$/.test(inputs.zip)) errors.push('Zip must be 5
digits.');
```

Uses regular expressions to validate:

- Phone = 10 digits only
- Email = standard email format
- Zip = exactly 5 digits

12. Store validation result in WeakMap:

```
    formMeta.set(form, { validated: errors.length === 0 });
```

You now know whether this form passed validation, and the info is stored safely per form object.

13. If there are errors, display them and stop:

```
    if (errors.length > 0) {
      responseDiv.innerHTML = `<div class="alert alert-danger">$
{errors.join('<br>')}</div>`;
      return;
    }
```

Outputs the errors inside a styled as an alert (you can style with Bootstrap or your own CSS).

14. Submit the data if it's valid:

```
    try {
      const res = await fetch('/submit-contact', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(inputs)
      });
```

- Sends the wrapped inputs object as JSON to your server endpoint.

15. Display success or failure message:

```
      const result = await res.json();
      responseDiv.innerHTML = `<div class="alert alert-success">$
{result.message || 'Success!'}</div>`;
    } catch (err) {
      console.error(err);
      responseDiv.innerHTML = `<div class="alert alert-
danger">Submission failed. Try again.</div>`;
    }
```

- Shows success if the request worked.
- Shows a red error box if the request failed.

Part Two: Filter / Frameworks

`Set` – tracks currently visible rows

`Symbol` – tags rows invisibly when they're shown so the tag is hidden from HTML/CSS

`Iterator` – custom iteration over table rows

All `filter.js` code with comments:

```
// === DOM References ===
const input = document.getElementById('search');
const rows = document.querySelectorAll('#infoTable tbody tr')
// === Advanced JS Features ===
// 1. Symbol: hidden unique property to tag visible rows
const VISIBLE = Symbol('visible');
// 2. Set: tracks rows that match the filter
const matchedSet = new Set();
// 3. Generator Function: creates a custom iterator over rows
function* rowIterator() {
  for (const row of rows) yield row;
}
// === Type Guard ===
// Ensures you're working with the correct input element
function isInput(el) {
  return el instanceof HTMLInputElement;
}
// === Main Filtering Logic ===
if (isInput(input)) {
  input.addEventListener('input', (e) => {
    const value = e.target.value.toLowerCase(); // normalize input
    matchedSet.clear(); // reset for each keystroke
    // Custom iterator usage
```

```
    for (const row of rowIterator()) {
      const text = row.textContent?.toLowerCase() || '';
      const match = text.includes(value);
      // Show or hide rows based on match
      row.style.display = match ? '' : 'none';
      if (match) {
        matchedSet.add(row);        // Add to tracked set
        row[VISIBLE] = true;        // Tag as visible using Symbol
      } else {
        delete row[VISIBLE];        // Remove tag if not matched
      }
    }
    // Log how many rows matched
    console.log(`🔍 Filter matched ${matchedSet.size} row(s).`);
  });
} else {
  console.error('Search input not found or not an input element.');
}
```

Part Three: Enhance quiz questions

`Set` – avoids duplicate questions

`Symbol` – adds hidden unique IDs to questions

`Generator` – auto-generates IDs

`Iterator` – allows looping through each question object's content

`yield` is a keyword used inside a generator function (function*) to pause and resume the function's execution, and return a value one at a time. (Think of it like a faucet that drips one value at a time, instead of pouring out everything at once.)

Full `quizquestions.js` code with comments:

```
// === Data Structures ===
// Prevents duplicate questions
const questionSet = new Set();
// Generator for unique question IDs (q-1, q-2, etc.)
function* idGen() {
  let id = 1;
  while (true) yield `q-${id++}`;
}
const generateId = idGen();
// === Main Quiz Data ===
let quizData = [];
// === Type Guard ===
function isDiv(el) {
  return el instanceof HTMLDivElement;
}
// === Enhancer: Adds ID, prevents duplicates, enables iteration ===
function enhanceQuestions(data) {
  return data
    .filter(q => !questionSet.has(q.question)) // skip duplicates
    .map(q => {
      questionSet.add(q.question);
      const enhanced = {
        ...q,
        id: generateId.next().value,
        [Symbol.iterator]: function* () {
          yield this.question;
```

```
          yield* this.choices;
          yield this.answer;
        }
      };
      return enhanced;
    });
}
// === Load and Render Quiz ===
fetch('/api/quiz')
  .then(res => res.json())
  .then(data => {
    quizData = enhanceQuestions(data);
    renderQuiz(quizData);
  })
  .catch(err => {
    console.error('Error loading quiz:', err);
  });
// === Render Quiz on Page ===
function renderQuiz(data) {
  const container = document.getElementById('quiz');
  if (!isDiv(container)) {
    console.error('Quiz container not found');
    return;
  }
  container.innerHTML = ''; // Clear previous
  data.forEach((q, index) => {
```

```
  const questionDiv = document.createElement('div');
  questionDiv.classList.add('mb-4');
  const question = document.createElement('h5');
  question.textContent = `${index + 1}. ${q.question}`;
  questionDiv.appendChild(question);
  q.choices.forEach((choice) => {
    const wrapper = document.createElement('div');
    wrapper.classList.add('form-check');
    const input = document.createElement('input');
    input.type = 'radio';
    input.name = `question-${index}`;
    input.value = choice;
    input.classList.add('form-check-input');
    input.id = `q${index}-${choice}`;
    const label = document.createElement('label');
    label.classList.add('form-check-label');
    label.setAttribute('for', input.id);
    label.textContent = choice;
    wrapper.appendChild(input);
    wrapper.appendChild(label);
    questionDiv.appendChild(wrapper);
  });

  container.appendChild(questionDiv);
});
const submitBtn = document.createElement('button');
```

```
  submitBtn.textContent = 'Submit';
  submitBtn.classList.add('btn', 'btn-success', 'mt-3');
  submitBtn.onclick = handleSubmit;
  container.appendChild(submitBtn);
}
// === Handle Quiz Submission ===
function handleSubmit() {
  let correctCount = 0;
  quizData.forEach((q, index) => {
    const selected = document.querySelector(`input[name="question-${index}"]:checked`)?.value;
    if (selected === q.answer) {
      correctCount++;
    }
  });
  alert(`You got ${correctCount} out of ${quizData.length} correct!`);
}
```

Part Four: Enhancing the Admin Page

`Symbol` — assigns internal unique IDs to each question

`Map` — tracks how many times each question has been edited

`WeakMap` — stores metadata per question

`Proxy` — wraps each question to intercept edits

`Reflect` — used inside Proxy for safe `get`/`set`

Full admin.js code with comments:

```
// === Unique hidden symbol for internal question ID ===
const QUESTION_ID = Symbol('questionId');
```

```
// === Maps and Metadata ===

const editTracker = new Map(); // Tracks number of edits per question

const questionMeta = new WeakMap(); // Stores last edited by (e.g.,
'Admin')

// === Wraps question object with Proxy ===

function wrapQuestion(q) {

  const wrapped = {

    ...q,

    [QUESTION_ID]: crypto.randomUUID?.() ||
Math.random().toString(36).slice(2)

  };

  return new Proxy(wrapped, {

    get(target, prop) {

      return Reflect.get(target, prop);

    },

    set(target, prop, value) {

      console.log(`✏️ Admin editing ${String(prop)}: ${target[prop]} →
${value}`);

      editTracker.set(target, (editTracker.get(target) || 0) + 1);

      questionMeta.set(target, { editedBy: 'Admin', timestamp:
Date.now() });

      return Reflect.set(target, prop, value);

    }

  });

}

// === Type guards ===

function isInput(el) {
```

```
  return el instanceof HTMLInputElement;
}
function isForm(el) {
  return el instanceof HTMLFormElement;
}
function isDiv(el) {
  return el instanceof HTMLDivElement;
}
// === Utility for safe element lookup ===
function getElement(id) {
  const el = document.getElementById(id);
  if (!el) throw new Error(`Element with ID '${id}' not found.`);
  return el;
}
// === Main Data Array ===
let quizData = [];
// === Load quiz data and wrap ===
fetch('/api/quiz')
  .then(res => res.json())
  .then(data => {
    quizData = data.map(wrapQuestion);
    renderQuizAdmin();
  })
  .catch(err => {
    console.error('Failed to load quiz:', err);
  });
```

```
// === Render editable quiz questions ===
function renderQuizAdmin() {
  const listDiv = getElement('quiz-admin-list');
  listDiv.innerHTML = '';
  quizData.forEach((q, index) => {
    const div = document.createElement('div');
    div.className = 'border p-3 mb-3';
    // Editable Question
    const questionInput = document.createElement('input');
    questionInput.type = 'text';
    questionInput.className = 'form-control mb-2';
    questionInput.value = q.question;
    questionInput.oninput = () => {
      quizData[index].question = questionInput.value;
    };
    div.appendChild(questionInput);
    // Editable Choices
    const choicesInput = document.createElement('input');
    choicesInput.type = 'text';
    choicesInput.className = 'form-control mb-2';
    choicesInput.value = q.choices.join(', ');
    choicesInput.oninput = () => {
quizData[index].choices = choicesInput.value.split(',').map(c =>
c.trim());
    };
```

```
  div.appendChild(choicesInput);
  // Editable Answer
  const answerInput = document.createElement('input');
  answerInput.type = 'text';
  answerInput.className = 'form-control mb-2';
  answerInput.value = q.answer;
  answerInput.oninput = () => {
    quizData[index].answer = answerInput.value.trim();
  };
  div.appendChild(answerInput);
  // Delete Button
  const delBtn = document.createElement('button');
  delBtn.className = 'btn btn-danger btn-sm';
  delBtn.textContent = 'Delete';
  delBtn.onclick = () => {
    quizData.splice(index, 1);
    renderQuizAdmin();
  };
  div.appendChild(delBtn);
  listDiv.appendChild(div);
});
// Save Button
const saveBtn = document.createElement('button');
saveBtn.textContent = 'Save All Changes';
saveBtn.className = 'btn btn-primary mt-3';
saveBtn.onclick = () => {
```

```
    fetch('/api/quiz', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(quizData)
    })
      .then(res => res.json())
      .then(data => {
        alert(data.message || 'Saved!');
      })
      .catch(err => {
        console.error('Save failed:', err);
        alert('Failed to save changes.');
      });
  };
  listDiv.appendChild(saveBtn);
}
// === Handle adding a new question ===
const addForm = document.getElementById('add-question-form');
if (isForm(addForm)) {
  addForm.addEventListener('submit', function (e) {
    e.preventDefault();
    const questionEl = document.getElementById('new-question');
    const choicesEl = document.getElementById('new-choices');
    const answerEl = document.getElementById('new-answer');
    const question = questionEl?.value.trim() ?? '';
    const choicesInput = choicesEl?.value.trim() ?? '';
```

```
    const answer = answerEl?.value.trim() ?? '';

    const choices = choicesInput.split(',').map(c =>
c.trim()).filter(Boolean);

    if (!question || choices.length < 2 || !answer) {

      alert('Please enter a question, at least 2 choices, and an
answer.');

      return;

    }

    const newQ = wrapQuestion({ question, choices, answer });

    quizData.push(newQ);

    renderQuizAdmin();

    if (questionEl) questionEl.value = '';

    if (choicesEl) choicesEl.value = '';

    if (answerEl) answerEl.value = '';

  });

} else {

  console.error('Add-question form not found.');

}
```