✅ 1️⃣ document.addEventListener('DOMContentLoaded', () => { ... });

Why:
This ensures **all HTML elements are fully loaded** before your JavaScript tries to access them.
If you didn't do this and put your <script> in the <head>, your code might run *before* #calc_display_input or your buttons exist in the DOM.

✅ 2️⃣ Get references for display elements

js

Copy code

```
const displayInput = document.getElementById('calc_display_input');

const displayResult = document.getElementById('calc_display_result');
```

Why:
You grab references to the **two main parts of your display**:

- displayInput → shows the current typed input **and expression** (7 + 3)
- displayResult → shows the final computed result (10)

By storing them once, you avoid repeatedly searching the DOM — more efficient and clear.

✅ 3️⃣ Set up your core state

js

Copy code

```
let currentInput = '';

let previousInput = null;

let operator = null;

let resultCalculated = false;
```

Why:
A calculator needs to remember:

- What's being typed **right now** (currentInput)
- What number was **typed before** you hit +, -, etc. (previousInput)
- Which operator is currently active (operator)
- Whether you just calculated something (resultCalculated)

These **global variables** define the **calculator's state**.
This is why they're declared **outside** any function but **inside** DOMContentLoaded → so they persist as the user clicks around.

✅ 4️⃣ Select all buttons

js

Copy code

```
const buttons = document.querySelectorAll('.calc_btns button');
```

Why:
You need every calculator button (0–9, +, DEL, = etc.) so you can attach click handlers.
Using .querySelectorAll + .forEach means you don't hardcode separate event listeners for each button — *DRY code*.

✅ 5️⃣ Loop: add click to every button

js

Copy code

```
buttons.forEach(button => {
 button.addEventListener('click', () => handleButton(button.textContent.trim()));
});
```

Why:
Each button should trigger **the same handleButton function**, passing in the button's text (like 7 or +).
textContent.trim() removes any accidental spaces or line breaks in the HTML.

This makes your button system **scalable** — add or remove buttons in HTML, no JS changes needed.

✅ 6️⃣ handleButton(value)

js

Copy code

```
function handleButton(value) {
 if (!isNaN(value) || value === '.') {
  handleNumber(value);
 } else if (value === 'DEL') {
  handleDelete();
 } else if (value === 'RESET') {
  handleReset();
  } else if (value === '=') {
    handleEqual();
  } else {
    handleOperator(value);
  }
  updateDisplay();
}
```

Why:
This is your **central router**. It decides **what type of button** the user clicked:

- !isNaN(value) → is it a number?
  isNaN() means “is Not A Number”, so !isNaN means *is* a number.
- value === '.' → allow the decimal point.
- DEL, RESET, = → special control buttons.
- Else → must be an operator (+, -, x, /).

Each case dispatches to a **dedicated function**, keeping logic organized.
Then updateDisplay() always runs at the end to refresh what the user sees.

✅ 7️⃣ handleNumber(value)

js

Copy code

```
function handleNumber(value) {
  if (resultCalculated) {
    currentInput = value === '.' ? '0.' : value;
    resultCalculated = false;
    return;
  }

  if (value === '.' && currentInput.includes('.')) return;

  if (currentInput === '0' && value !== '.') {
    currentInput = value;
  } else {
```

```
  currentInput += value;
 }
}
```

Why:
This handles **typing numbers** and the **decimal point**:

- **If a result was just calculated**, starting a new number replaces the old result.
  E.g., after 2 + 2 = 4, typing 7 should start a new calculation.
  Special case: if the first input is ., you get 0..
- **Only allow one decimal** → if there's already a ., ignore extra dots.
- If you type 0 first then 3, it should become 3, not 03.
  So if currentInput is 0 and you type something else, replace it.
- Otherwise, just **add** the new digit.

✅ 8️⃣ handleOperator(op)

js

Copy code

```
function handleOperator(op) {
 if (operator && !resultCalculated) {
  handleEqual();
 }

 previousInput = currentInput;
 operator = op;
 resultCalculated = false;
 currentInput = '';
}
```

Why:
When you click an operator:

- If there's already an active operator and you didn't hit =, it **auto-computes** first (handleEqual()).
  E.g., 2 + 3 + → when you hit the second +, it auto-computes 5.
- It stores currentInput as previousInput and sets the operator.
- Clears currentInput so you can start typing the next number.

✅ 9️⃣ handleEqual()

js

Copy code

```
function handleEqual() {
 if (!operator || previousInput === null) return;

 const prev = parseFloat(previousInput);
 const current = parseFloat(currentInput);
 let result = 0;

 switch (operator) {
  case '+':
   result = prev + current;
   break;
  case '-':
   result = prev - current;
   break;
  case 'x':
   result = prev * current;
   break;
  case '/':
   result = prev / current;
   break;
 }

 displayResult.textContent = result;
 currentInput = result.toString();
 previousInput = null;
 operator = null;
 resultCalculated = true;
 updateDisplay();
}
```

Why:
This does the **real math**:

- If there’s no operator or no previousInput, do nothing (guards against = spam).
- parseFloat converts your string inputs to real numbers.
- switch decides which math operation to perform.
- Sets displayResult → so the result appears bottom right.
- Makes currentInput the result so you can **continue calculating** if you want.
- Resets previousInput and operator → ready for next steps.
- Sets resultCalculated so the next number starts fresh.
- Calls updateDisplay() to show the new state.

✅ 🔟 handleDelete()

js

Copy code

```
function handleDelete() {
 if (currentInput.length > 1) {
   currentInput = currentInput.slice(0, -1);
 } else {
   currentInput = '';
 }
}
```

Why:
Removes the **last digit** from the current input.
If there’s only one digit left, sets it to '' → effectively a soft clear.

✅ 1️⃣ 1️⃣ handleReset()

js

Copy code

```
function handleReset() {
 currentInput = '';
 previousInput = null;
 operator = null;
 resultCalculated = false;
 displayResult.textContent = '';
}
```

Why:
Resets **everything** → clears currentInput, previousInput, operator, result flag, and wipes the displayed result.
Puts the calculator back to a fresh state.

✅ 1️⃣2️⃣ updateDisplay()

js

Copy code

```
function updateDisplay() {
 if (operator && previousInput !== null) {
  displayInput.textContent = `${previousInput} ${operator} ${currentInput}`;
 } else {
  displayInput.textContent = currentInput;
 }
}
```

Why:
Controls what the top-left display shows:

- If there's an active operator, it shows the *full expression* (7 + 3).
- If not, just the current number.
- Keeps the user informed of exactly what's happening.

✅ 1️⃣3️⃣ Call updateDisplay() on load

js

Copy code

```
updateDisplay();
```

Why:
So the calculator **starts clean**, showing '' instead of undefined or stale data.

✅ 1️⃣ 4️⃣ Theme switcher

js

Copy code

```
const themeRange = document.querySelector('.theme-range');
const body = document.body;

const currentTheme = body.getAttribute('data-theme').replace('theme', '');
themeRange.value = currentTheme;

themeRange.addEventListener('input', () => {
  const themeNum = themeRange.value;
  document.body.setAttribute('data-theme', `theme${themeNum}`);
});
```

Why:

- Grab the theme slider (.theme-range).
- Grab <body> to set data-theme.
- On load: match the slider's value to the current theme.
- When the slider changes: update the data-theme attribute → triggers your CSS theme change.

✅ ✅ ✅ Summary: Why this structure works

- **Clear state**: currentInput, previousInput, operator store all needed info.
- **Pure functions**: Each handleX function does *one thing* → easy to read & debug.
- **Central dispatcher**: handleButton routes input to the right function.
- **Single display update**: Always refreshes the UI after each step.
- **Good UX flow**: Handles edge cases like multiple operators, multiple decimals, chaining results.
- **Theme switch is isolated**: No conflict with calculator logic.