Props {}, e().preventDefault, {handleSubmit}, e, target, App.jsx to Input.jsx through props explained.

Great effort 👏 — you've clearly thought carefully about props, children, events, and state. Let's go point by point and clarify your explanations and your questions.

🔹 Your explanation about props & children

✅ You are correct: `children` is **just a prop** (like `type`, `value`, `onChange`) but React treats it as *special*: it automatically collects whatever you write **between opening and closing component tags**.
Example:

```
<Input>Hello</Input>
```

- → Inside `Input`, the prop `children` will be `"Hello"`.

⚠️ In your `Input.jsx`, you wrote:

```
<input>
  { children }
</input>
```

But `<input>` is a **void element** in HTML (it can't have children, like `<img>`). That's why normally you wouldn't pass `children` into `<input>`. Instead, you'd pass props like `type`, `value`, `onChange`. So in this exact case, `{ children }` won't do anything useful. A better reusable Input would look like:

```
export function Input(props) {
  return (
    <input
      className="border-4 border-sky-500/10 border-b-sky-500"
      {...props}   // spread props like type, value, onChange
    />
  );
}
```

-

So — your **general understanding of children = props that allow reusability is correct** ✅, but in this specific code, `children` isn't being used correctly (React won't complain, but it won't show anything).

🔹 Your questions

1. Why do we have to write props inside curly brackets `{}`?
Because inside JSX, everything in `{}` is interpreted as JavaScript **expressions**, not plain text.

- Without `{}`, React thinks it's literal text or HTML.
- With `{}`, React evaluates it.
  So:

```
<input value={name} />   // uses JS variable "name"
<input value="name" />   // literally sets value="name"
```

---

2. Why do we have to mention props again just after the HTML element we create for that component?
You mean like this?

```
<Input type="text" value={name} onChange={...} />
```

That's because your custom `<Input>` is just a *wrapper*. Unless you pass those props down to the actual `<input>` element inside it (`...props` or manually forwarding them), they won't be used. That's why we repeat them — it's like giving data to a machine so it knows what to render.

---

3. Why can't we write `e().preventDefault()`?
Because `e` is **not a function**, it's an **object** (the event object).

- Functions: need `()`.
- Objects: don't.

So the correct way is:

e.preventDefault(); // call preventDefault method of e

If you wrote `e()`, you'd be trying to call the event object itself like a function — error.

---

4. Why do we have to write `handleSubmit` in curly brackets?

<form onSubmit={handleSubmit}>

- `onSubmit` expects a **function reference** (not a string).
- If you wrote `onSubmit="handleSubmit"`, React would just see text.
- If you wrote `onSubmit={handleSubmit()}`, it would **run immediately** instead of waiting for form submit.

So `{handleSubmit}` gives React the actual function reference.

---

5. Does `target` property mean only keyboard input, not mouse?

No 🙂. `target` is **the element that triggered the event**, regardless of keyboard or mouse.
Examples:

- In `<input onChange={...}>`, `e.target` = the input field.
- In `<button onClick={...}>`, `e.target` = the button.
- In `<div onMouseEnter={...}>`, `e.target` = the div.

So `e.target` just points to *whatever element* caused the event.
Then `.value` only makes sense if that element has a `value` property (like `<input>`, `<textarea>`, `<select>`).

✅ **Summary fixes for you:**

- `children` is correct in theory, but not useful for `<input>`. Better use `{...props}`.
- `{}` in JSX means “evaluate JavaScript here.”
- `e.preventDefault()` works because `e` is an object, not a function.
- `onSubmit={handleSubmit}` passes the function reference, not its result.
- `target` is the element that fired the event (keyboard, mouse, or other).

Do you want me to **rewrite your Input.jsx** in the correct reusable way (so it handles `type`, `value`, `onChange` properly without `children`)?

Wait what? I just created the Input.jsx component so that I don't have to repeatedly write long in-line TailWind.css styling. Inside the Input.jsx I wrote the styling like this:

```
<input
className="border-4 border-sky-500/10 border-b-sky-500"
>
{ children }
</input>
```

So whenever I call it, I just write the component name Input, and the style appears, making the code less of a headache, but what does it have to do with the type, value, and onChange properties?

Ahhh okay ✅ now I get exactly what you were aiming for — thanks for clarifying.

👉 You built your `Input.jsx` **only as a styling shortcut**, so you don't have to rewrite long Tailwind classes every time. Perfectly valid approach 👍.

But here's the catch:

---

1. Props & styling-only components

Right now, your Input is:

```
export default function Input({ children }) {
  return (
    <input
      className="border-4 border-sky-500/10 border-b-sky-500"
    >
      {children}
    </input>
  );
}
```

Two things to notice:

- `children` won't work inside `<input>` (since inputs can't have children).
- If you **don't forward props**, your `<input>` can only ever be a static styled box. It won't accept `type`, `value`, `onChange`, etc.

That's why in your `App.jsx`, even though you wrote:

```
<Input type="text" onChange={...} value={name} />
```

those props are *received* by `Input.jsx` but are *never applied* to the actual `<input>` — so your form won't work properly.

---

2. Why `{...props}` is needed in your case

If your only goal is: **“I want my input styled, but still behave like a normal HTML input”**, then you have to forward the normal HTML attributes (`type`, `value`, `onChange`, etc) to your actual `<input>` element.

Like this:

```
export default function Input(props) {
  return (
    <input
      className="border-4 border-sky-500/10 border-b-sky-500"
      {...props}   // ← this makes your Input reusable with dynamic behavior
    />
  );
}
```

Now:

```
<Input type="text" value={name} onChange={(e) => setName(e.target.value)} />
<Input type="email" value={email} onChange={(e) => setEmail(e.target.value)} />
```

- ✅ Styling is always there (from your Tailwind classes).
- ✅ Each `<Input>` still works like a real input because `type`, `value`, and `onChange` are forwarded.
- ✅ You don’t need to rewrite the long Tailwind string everywhere.

---

3. Without `{...props}`

If you *don’t* add `{...props}`, your Input component will ignore all the extra attributes from the parent. That means:

- Your inputs will always be type="text".
- They won’t update with `value` or `onChange`.
- Basically, they’ll look nice but not work in a form.

✅ So:
You created `Input.jsx` for styling, but for it to behave like a normal input **and** keep your form functional, you need `{...props}`. Otherwise, you only get the style, not the behavior.

Would you like me to show you a **minimal broken vs fixed example** (side by side) so you can visually see how your current version fails to update state, while `{...props}` makes it work?

Thanks, but I thought props are meant to work in a one-way direction, like if a component is created with a props, then it becomes the parent, when it is passed into another component to prevent code repetition, then the receiving component becomes the child. So if App.jsx is executed, it finds the custom Input component, it goes inside the Input.jsx file, carrying the HTML attributes from App.jsx with it, finds the actual <input> element, necessary Tailwind styling alongwith the props with spread operator, so here props represent each HTML attribute of Input component back at App.jsx. So it creates the actual HTML <input> then adds:
<input type="text" onChange={(e) => setName(e.target.value)} value=name}>

Final result at index.html:
<input type="text" value="name"/>

Can you please explain this in simple words using this analogy?
So whenever React wants to write anything on the notebook (Real DOM), it first writes it on a ripped page from the notebook or anywhere (Virtual DOM), places it next to the notebook to compare, then says 'I'll compare the notebook (Real DOM) with the ripped page (Virtual DOM), prioritizing the notebook, if the notebook doesn't have whatever the ripped page has, then I'll write things from ripped page to notebook, and throw away the ripped page.' Thus, whenever React has to write, even a letter, into the Real DOM or notebook, React has to write it now a ripped page (Virtual DOM), put it next to the notebook to compare, and then write it onto the notebook.

Mistake: App.jsx is the parent and Input.jsx is the child, because App.jsx sends the blueprint, and Input.jsx is where the actual HTML element is built.

Yes 👌 you've got the right mental picture — props are **one-way** (parent → child), and React always uses the Virtual DOM like a "scratch copy" before touching the Real DOM. Let me connect your analogy with your Input component example step by step in very **simple words**.

---

1. Parent (App.jsx) gives instructions

In App.jsx, you wrote:

```
<Input
  type="text"
  value={name}
  onChange={(e) => setName(e.target.value)}
/>
```

Here, `<Input>` is **not an actual `<input>` yet**.
It's like saying:

> "Hey React, please use my custom Input blueprint, and here are some instructions (props): type is text, value is name, onChange is this function."

So yes — **props are the delivery package** going from parent (App.jsx) → child (Input.jsx).

---

2. Child (Input.jsx) receives the package

Inside Input.jsx:

```
export default function Input(props) {
  return (
    <input
      className="border-4 border-sky-500/10 border-b-sky-500"
      {...props}
    />
  );
}
```

React comes here carrying that "package" (`props`) from App.jsx.
It sees:

- Tailwind styling you added
- Spread operator `{...props}` → this unpacks the parent's instructions (`type`, `value`, `onChange`) onto the actual `<input>`.

So React now builds the final instruction sheet:

```
<input
  class="border-4 border-sky-500/10 border-b-sky-500"
  type="text"
  value="name"
  onChange="(e)=>setName(e.target.value)"
/>
```

---

3. Virtual DOM = ripped page

React doesn't write directly into the Real DOM (notebook). Instead:

- React writes this `<input>` into its **Virtual DOM** (the ripped page).
- Then React looks at the Real DOM (the notebook).
- It compares: "Do I already have an input with these properties in the notebook?"
    - If yes → do nothing.
    - If no (or something changed) → update the notebook (Real DOM).

So, when `name` changes (e.g., you type "Ali"), React:

1. Writes `<input value="Ali">` on the ripped page (Virtual DOM).
2. Compares it with the notebook's `<input value="name">`.
3. Notices the difference.
4. Updates the Real DOM so now it has `<input value="Ali">`.
5. Throws away the ripped page and waits for the next change.

---

4. Why one-way props matter here

Because **the parent always decides what instructions to give**.

- App.jsx says: “I want Input with type=text, value=name, and this onChange handler.”
- Input.jsx just delivers those instructions faithfully to the real `<input>`.
- Input.jsx never pushes values back up by itself — it only uses what the parent gives.

That’s the “one-way street” of props.

✅ **So in your analogy**:

- **App.jsx** = teacher who writes instructions (props) on a sticky note.
- **Input.jsx** = student who copies the sticky note into the ripped page (Virtual DOM) along with extra decoration (Tailwind style).
- **React** = examiner who checks ripped page vs notebook (Virtual DOM vs Real DOM).
- If different → React updates the notebook.
- If same → notebook stays as it is.

Would you like me to make a **tiny ASCII sketch** (Parent → Child → Virtual DOM → Real DOM flow) so you can literally see the “package moving” through each step?

Thanks, but I am still unclear about 'e' or event object and target property. I mean, .value and .name is simple, it just means to send only the value or (name or key) property of that HTML element. Can you explain 'e' and target using the notebook analogy? Like how 'e' can perhaps be a blank space, where the student left in his notebook because it needed someone else's input or written by someone else, so he left space, and the target can be a big oval, telling that someone where to write?

This is just my thought. If you can find a better analogy to explain 'e' and target, then feel free to do so.

Great question 👌 — I love how you’re tying this into your notebook analogy. Let’s build on that idea so `e` (event object) and `target` (the thing that triggered the event) make sense.

---

1. What is `e` (event object)?

When something happens (click, type, submit), React hands you a **report sheet** about that event.
That sheet is `e`.

- It contains **everything React knows** about the event:
    - what element caused it (`e.target`)
    - when it happened
    - what keys were pressed
    - mouse position (if a click)
    - plus helper functions like `e.preventDefault()`.

So `e` is like:
📄 **"The official event report."**

---

2. What is `target`?

Inside that event report (`e`), there's a field called `target`.

- `target` = the **exact element in the DOM that caused the event**.
- Example:
    - If you typed in the name input → `e.target` is that `<input type="text">`.
    - If you clicked the submit button → `e.target` is the `<button type="submit">`.

So `target` is like:
🎯 **"Who the report is about."**

---

3. Notebook analogy

Imagine the **Real DOM = the notebook** again.

- Every time a user writes something into the notebook (e.g., types into an input), the teacher (React) doesn't just let it happen silently. Instead, the teacher hands you a **report slip** (`e`).

That slip says:

- “This change happened.”
- “It happened in this spot on the notebook (`e.target`).”
- “Here’s what was written (`e.target.value`).”
- “Here’s the rule you can apply (e.preventDefault, etc.).”

---

4. Example with your analogy

Suppose you have a line in the notebook for “Name: ____”.
The student (user) writes **Ali** there.

- React says: “Okay, this was an event. Here’s the slip (e).”
- On that slip:
  - `e.target` → the blank “Name: ____” line in the notebook.
  - `e.target.value` → the actual text “Ali” the student wrote.
  - `e.preventDefault()` → an option that says, “Don’t do the notebook’s default reaction (like refreshing).”

---

5. Why it matters

So, in simple terms:

- `e` = the **whole slip** about what happened.
- `e.target` = the **exact line or box in the notebook** that was changed.
- `e.target.value` = the **new writing** inside that line or box.

✅ Analogy rewritten:

- **Event (`e`)** = the slip of paper with the details of what just happened in the notebook.
- **Target (`e.target`)** = the oval box (the exact blank space in the notebook) where the event occurred.

- **Value (`e.target.value`)** = the new text inside that oval.

Would you like me to also add **how this differs if it's a mouse event vs a keyboard event** using the same analogy (like, writing vs pointing)?

> Thanks, it helped me a lot. And the fact that the teacher gives the student a 'report slip' mentioning everything the student did makes me laugh and that poor student is nervous. This kind of thing helps to make learning easier.

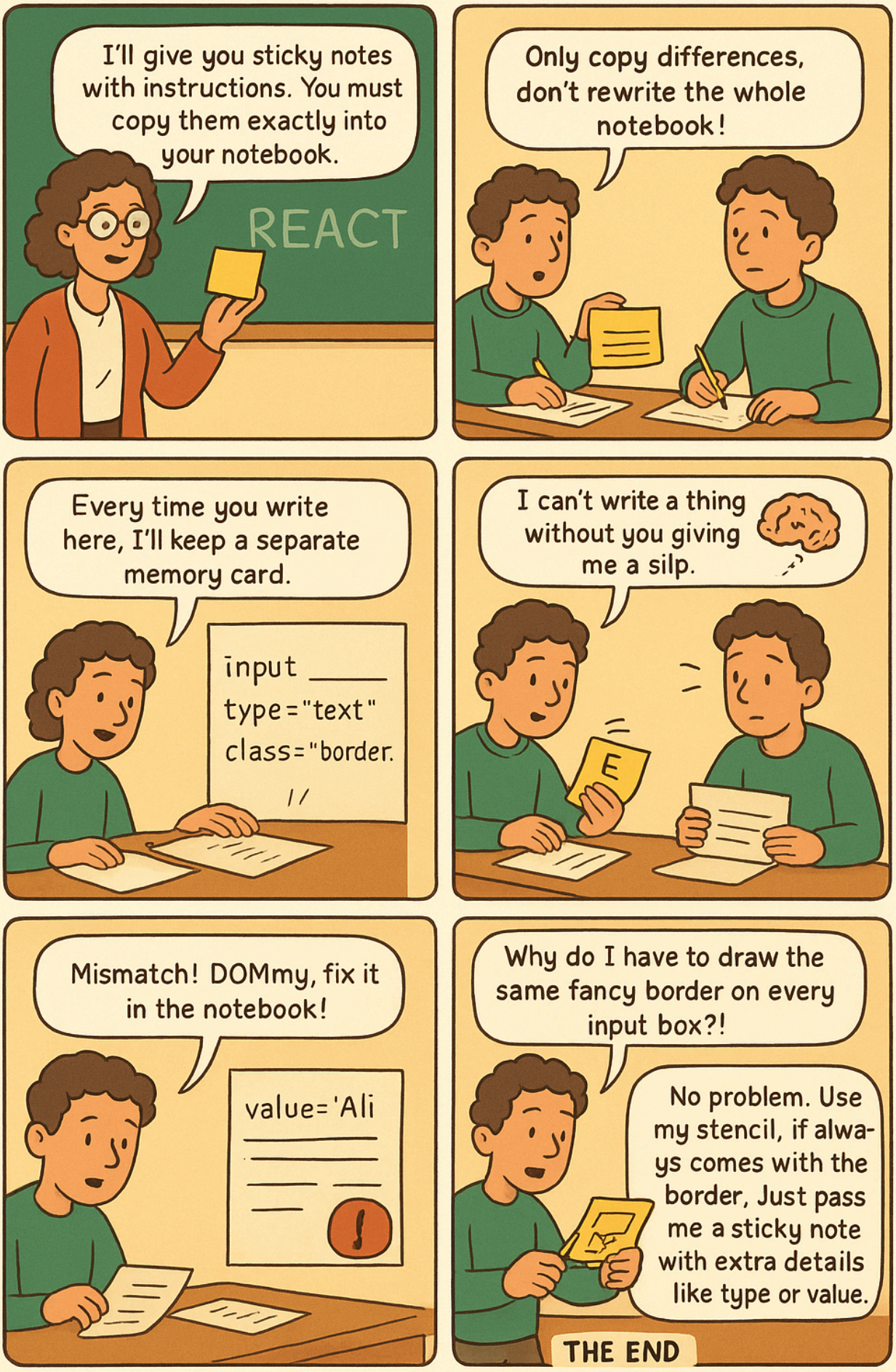
Haha 😄 I'm glad it clicked for you!
Yeah, poor student — every time they write even a single letter, the teacher runs over:

> "Here's your slip! You typed the letter **A** at 12:03:15 PM in the **Name** box, position #2. Don't forget, I'll be reporting this to React!"

No wonder the student is nervous 😂.

But that's exactly what makes React powerful — nothing slips by without being documented in that event report.

Would you like me to turn this whole **teacher–student–notebook analogy** into a little comic-style story (with props, state, Virtual DOM, events) so you'll always remember how it works?


I'll give you sticky notes with instructions. You must copy them exactly into your notebook.
REACT
Only copy differences, don't rewrite the whole notebook!
Every time you write here, I'll keep a separate memory card.
input ____
type="text"
class="border.
//
I can't write a thing without you giving me a silp.
E
Mismatch! DOMmy, fix it in the notebook!
value='Ali
Why do I have to draw the same fancy border on every input box?!
No problem. Use my stencil, if alwa-ys comes with the border, Just pass me a sticky note with extra details like type or value.
THE END

That reminds me. Can you explain to me what do onChange() and onSubmit() do? If 'e' is a entire report slip containing everything and target is a HTML element where something new was detected, then is onChange() like a teacher and onSubmit() a clerk collecting all form data from App.jsx to Back-end?

I like how you're connecting this 👌 — let's extend the "school notebook" analogy to **onChange** and **onSubmit** so it sticks clearly.

📒 Notebook Analogy

- **The Student's Notebook** → Real DOM
- **The Teacher's Ripped Page** → Virtual DOM
- **Report Slip (e)** → All details of what happened
- **Target** → The exact notebook line (HTML element) where something changed

---

🎓 onChange

Think of `onChange` as the **teacher standing over the student's shoulder**.
Every time the student writes something new (like typing a letter), the teacher immediately:

1. Notices the change.
2. Fills a **report slip (e)**.
3. Hands it to React, saying: *"Hey, the student just wrote this at this exact place!"*

So:

- `onChange` = instant notification every time the student makes a little change.
- Used for **inputs, checkboxes, selects, etc.** to update React state right away.

🗂️ onSubmit

Now imagine the **clerk** sitting at the front of the class.
The student finishes their assignment (the whole form) and hands it in.
The clerk doesn't care about every letter typed — just the **final submission** of the whole notebook page.
The clerk collects all pages (form data), attaches a report slip (`e`), and passes it to React saying:

"The form is done. Here's everything the student filled. What should I do with it — send it to the backend, or show an error?"

So:

- `onSubmit` = triggers only when the **entire form** is submitted (like pressing Enter or clicking a submit button).
- Usually used for **validation** and **sending data** to the backend.

👉 In short:

- **onChange** = teacher checking *every little change* in real time.
- **onSubmit** = clerk collecting the *entire form* at the end.

Do you want me to also show you how `onChange` and `onSubmit` actually **work together in one small form example**, using the same analogy?