Jsx—

Returns single element

It can be div/section/article or fragment

Use camelCase for html attribute

Use className instead of class

Close every element

Const person=()=>{return(<p>hello</p>)}

Equivalent to

Const person=()=> <p>hello</p>

Inline css is more powerful than, css imported in a file.

All values of javascript object must be enclosed in “” or ‘’

In jsx,with opening javascript,we have to access variables and statements that return something and we cant pass javascript statements.

Eg

Let title=5

<p>{title}</p> , <p>{8+9}</p>,<p>{title.toUpperCase()}</p> is valid

<p>{let x =6}</p> is invalid

<Book title=”abc” author=”def”>----throws the object{title:”abc”,author:”def”}

<Book imglink=”abc.jpg”>---throws the object {imglink:”abc.jpg”}

const booklist = [

  { id=1,title: "abc", author: "def" },

  { id=2,title: "ghi", author: "jkl" },

];

Here if we add new elements or delete existing elements, index of elements may change but id of elements doesn’t change

 {booklist.**map**((book) => {

        {

          counter++;

        }

        return <**Book** *key*={counter} {...book}></**Book**>;

      })

Here {…book} spreads out the book object and sends it to component.

Now in component,we can access it directly with props.

const **Book** = (props) => {

  console.**log**(props);

  return (

    <div *className*="book">

      <img

*className*="img\_class"

*src*="https://storage.googleapis.com/zingchart-blog/zing-content/2016/06/react-1.png"

*alt*="cant display"

*width*="100 px"

*height*="100px"

      />

      <h1 *style*={{ color: "#ffff00", fontSize: "3.5 rem" }}>{props.title}</h1>

      <h2>{props.author}</h2>

    </div>

  );

};

i.e.,{…book} sends {id,title,author} to the Book component. Which can be stored in a variable props and accessed later.

<**Book** *key*={book.id} {...book}></**Book**>;

Is equivalent to

<**Book** *key*={book.id} id=1 title=”abc” author=”def” ></**Book**>;

For book={id:1,title:”abc”,author:”def”}

const **Book** = (props) => {

  console.**log**(props);

  const **clickHandler** = () => {

**alert**("hello ");

  };

  return (

    <div *className*="book">

      <img

*className*="img\_class"

*src*="https://storage.googleapis.com/zingchart-blog/zing-content/2016/06/react-1.png"

*alt*="cant display"

*width*="100 px"

*height*="100px"

      />

      <h1

*style*={{ color: "#ffff00", fontSize: "3.5 rem" }}

*onClick*={() => {

          console.**log**(props.title);//inline handling of events

        }}

      >

        {props.title}

      </h1>

      <h2>{props.author}</h2>

      <button *type*="button" *onClick*={**clickHandler**}>//Handling events by calling a function

        button

      </button>

    </div>

  );

};

Events can be handled in two ways:with functions or inline

For button, we have defined a function and called it for onClick event.

But for h1 tag of title, we have defined a inline function which can access props.

To pass arguments to a event handler, we have to set a arrow function and call the event handler in the arrow function

Handling events is done in 3 ways

1—inline functions

2--- write functions and call them on event

3--- write inline functions which call the function with arguments .

const **Book** = (props) => {

  console.**log**(props);

  const **clickHandler** = (auth) => {**alert**(auth);};

  return (

    <div *className*="book">

      <h1

*style*={{ color: "#ffff00", fontSize: "3.5 rem" }}      >

        {props.title}

      </h1>

      <h2>{props.author}</h2>

      {*/\* <button type="button" onClick={clickHandler}>*

*button*

*</button> \*/*}

      <button

*type*="button"

*onClick*={() => {//inline function which calls event handling function

**clickHandler**(props.author);

        }}

      >

        example

      </button>

    </div>

  );

};

Here, for the button, we have to define a arrow function,which calls the clickHandler() with arguments.

But if we call clickHandler() directly with arguments, it is called on page load itself.

      <button

*type*="button"

*onMouseOver*={(e) => {

*// console.log(e);*

**clickHandler**(props.author, e);

        }}

      >

        example

      </button>

A event object is thrown to every event handler implicitly. This object has some useful information

const **Book** = (props) => {

  console.**log**(props);

  const **clickHandler** = (e) => {

    console.**log**(e);

  };

  return (

    <div *className*="book">

      <h1

*style*={{ color: "#ffff00", fontSize: "3.5 rem" }}

*onClick*={() => {

          console.**log**(props.title);

        }}

      >

        {props.title}

      </h1>

      <h2>{props.author}</h2>

      <button *type*="button" *onClick*={**clickHandler**}>

        button

      </button>

    </div>

  );

};

Here clickHandler implicitly receives a event object even if we don’t pass the object to it.

Named export :

Books.js:

Export const num=[1,2,3]

If we have to use the array num, we have to use the statement

Import {num} from “./Books.js”—**Here the name of variable used in {} has to be same as variable exported.**

**Default export:**

But if we use the following statement in Books.js,

Const num=[1,2,3]

Export default num

When we have to use num , the following statement is used

Import num from “./Books.js”

Here variable num can be anything.

There can be multiple named exports in a file but there can be only one default export

-- useState is a function which returns an array [state value s, function which controls state value s].

Here state value can be a integer,Boolean,string,object …or any javascript value.

When we invoke useState, we have to pass a default value.

console.**log**(**useState**("hello"));

gives output as:

Array [ "hello", dispatchAction() ]

​This function takes new value that needs to be assigned to statevalue.

Setting the state value with the function renders the page.

const **TextChanger** = () => {

  var Text=”chirudeep”

function **clickHandler**()

{

  Text="karthik"

}

  return (

    <**React.Fragment**>

<h2>{**Text**}</h2>

<button *className*="btn" *onClick*={clickHandler}>

        change the name

</button>

    </**React.Fragment**>

  );

};

Here changing Text won’t change the value in h2 field.

Because, once rendered, change the value of variable Text but it doesn’t render the component with new value of Text.

So we have to use states. Where changing the value of state not only changes the value of state,but also renders the component again.

Thus we will have new value updated in h2.

const **UseStateBasics** = () => {

  let [Text, **setText**] = **useState**("chirudeep");

  var **clickHandler** = () => {

**setText**("karthik");

  };

  return (

    <**React.Fragment**>

      <h2>{Text}</h2>

      <button *className*="btn" *onClick*={**clickHandler**}>

        change the name

      </button>

    </**React.Fragment**>

  );

};

Here setText sets the state value to new value passed to it.

This renders the component again changing the value in h2 tag.

const **UseStateBasics** = () => {

  let [Text, **abc**] = **useState**("a");

  var **clickHandler** = () => {

    if (Text === "a") {

**abc**("b");

    } else {

**abc**("a");

    }

  };

  return (

    <**React.Fragment**>

      <h2>{Text}</h2>

      <button *className*="btn" *onClick*={**clickHandler**}>

        change the name

      </button>

    </**React.Fragment**>

  );

};

This toggles the text of h2 between a and b .

Hooks generally starts with word use.

Eg: useState,useEffect…etc

Hooks must be used in function or a component body.

If we use hook in a component, the component name must start with upper case.

let [Text, **abc**] = **useState**("a");

const **UseStateBasics** = () => {

  var **clickHandler** = () => {

    if (Text === "a") {

**abc**("b");

    } else {

**abc**("a");

    }

  };

  return (

    <**React.Fragment**>

      <h2>{Text}</h2>

      <button *className*="btn" *onClick*={**clickHandler**}>

        change the name

      </button>

    </**React.Fragment**>

  );

}

This will give the following error:

Error: Invalid hook call. Hooks can only be called inside of the body of a function component

Hooks can’t be called conditionally.

But function of a hook can be called conditionally.

In above examples, setText can be invoked in if statement.

But, useState can’t be invoked in a if statement.

If we use

<button *onClick*={**setPeople**([])} *className*="btn"> clear items</button>

This will set people to empty array on page load itself instead of when we click button.

To prevent this, we have to call setPeople([]) in a inline function like this:

<button *onClick*={() => **setPeople**([])} *className*="btn">clear items</button>

Working with arrays in useState:

 const data = [

  { id: 1, name: 'john' },

  { id: 2, name: 'peter' },

  { id: 3, name: 'susan' },

  { id: 4, name: 'anna' },

];

const **UseStateArray** = () => {

  var [people, **setPeople**] = **React**.**useState**(data);

  function **remover**(id) //this function is invoked when remove button is clicked

{

    let newPeople = [];

    newPeople = people.**filter**((person) => person.id !== id);

//This statement removes the element with id passed on invoking the function

**setPeople**(newPeople);

  }

  return (

    <>

      {people.**map**((person) => {

        return (

          <div *key*={person.id} *className*="item">

            <h1>{person.name}</h1>

            <button *onClick*={() => **remover**(person.id)}>remove</button>

          </div>

        );

      })}

      <button *onClick*={() => **setPeople**([])} *className*="btn">

        clear items

      </button>

    </>

  );

};

const **UseStateObject** = () => {

  const [person, **setPerson**] = **React**.**useState**({

    name: "chirudeep",

    age: 24,

    message: "random message",

  });

  console.**log**(person);

  function **clickHandler**() {

**setPerson**({name: "karthik" });

  }

*// setPerson({ name: "chirud", age: 24, message: "new message" });*

  return (

    <>

      <h1>{person.name}</h1>

      <button *onClick*={**clickHandler**} *className*="btn">

        click this for change

      </button>

    </>

  );

};

Here setPerson sets the person object to { name: "karthik" } .

To set all fields, we have to send entire object like this:

setPerson({name:”karthik”,age:24,message:”random message”}).

Here even if we have to change a single field, we have to destructure entire object.

But a short cut to do is using spread out operator :

**setPerson**({ ...person, name: "karthik" });

This statement is equivalent to :

setPerson({name:”karthik”,age:24,message:”random message”}).

setPerson({copy old values using spread operator, value you want to change})

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setValue**(value + 1);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

      <button

*onClick*={() => {

**setValue**(value - 1);

        }}

*class*="btn"

      >

        decrement

      </button>

      <button

*onClick*={() => {

**setValue**(0);

        }}

*class*="btn"

      >

        Reset

      </button>

    </>

  );

};

The following code takes 2sec before setting the new value.

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setTimeout**(() => {

**setValue**(value + 1);

    }, 2000);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

    </>

  );

};

Here setTimeout takes to arguments

1—inline function which needs to be executed after time duration t.

2—time duration t.

If setTimeout is invoked, the function passed as parameter is executed after t milliseconds.

But here, even if we click the increment button multiple times, it will increment only once.

This is because setValue is asynchronous.

i.e, the setValue function takes the old value before increment and increments it instead of incrementing the updated value.

setValue takes two types of arguments:

1—new value to be set

2—inline function which returns the new value to be set.

This function by default takes previous state value as parameter.

And this has to return the updated value.

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setTimeout**(() => {

**setValue**((prevValue) => {

        return prevValue + 1;

      });

    }, 2000);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

    </>

  );

};

Here, inline function in the setValue gets previous Value of state as input by default and returns the new value to be set.

useEffect—

By default, useEffect runs after every re-render

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button *className*="btn" *onClick*={() => **setValue**(value + 1)}>

        increment

      </button>

    </>

  );

};

Here every time we click increment button, useEffectBasics is rendered.

And every time useEffectBasics is rendered, inline function defined in useEffect is run.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    document.title = value;

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here every time button is clicked, it increments the value and component is rendered.

Here useEffect is called every time component is rendered and tab’s title is changed.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    if (value > 1) {

      document.title = value;

    }

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here title of the page changes only after value becomes 2.

useEffect can have two arguments.

First is the function to be executed on each render, second is the array of dependencies.

If the list is empty, function runs only on initial render but not subsequent renders.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    document.title = value;

  }, []);

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here, title of page is set to 0 when we load the pages. Then it is not changed.

If we add value variable to list of dependencies, then the page title changes every time value variable changes

There can be multiple useEffect functions defined .

const **UseEffectCleanup** = () => {

  var [size, **setSize**] = **useState**(window.innerWidth);

function **eventHandler**() {

**setSize**(window.innerWidth);

  }

**useEffect**(() => {

    window.**addEventListener**("resize", **eventHandler**

    });

  });

  return (

    <>

      <h2>useEffect cleanup</h2>

      <h2>{size}</h2>

    </>

  );

};

Here we have added a event listener on window when the component renders first time.

This listens to changes in window size and sets the value of size variable(state) on change of the window size.

This triggers re render. Now useEffect is called again and a event listener on window is added again.

i.e, for every render, a new event listener is created on window.

This increases number of event listeners exponentially.

The syntax of useEffect is:

**useEffect**(() => {-----------function which runs on every render

    effect

    return () => {------------cleanup function-ran 1st

      cleanup

    }

  }, [input])

Here useEffect function has two arguments.

A function and a list of dependencies.

Function can have a return statement which can return a function which can perform cleanup operations .

This cleanup function runs before other operations of useEffect is ran.

const **UseEffectCleanup** = () => {

  var [size, **setSize**] = **useState**(window.innerWidth);

  function **eventHandler**() {

**setSize**(window.innerWidth);

  }

  const **cleanupfn** = () => {

    console.**log**("cleanup");

    window.**removeEventListener**("resize", **eventHandler**);

  };

**useEffect**(() => {

    console.**log**("useeffect");

    window.**addEventListener**("resize", **eventHandler**);

    return **cleanupfn**;

  });

Here the cleanup function is ran before other operations are done.

This removes the event listener previously defined thus preventing the exponential increase in number of event listeners.

Note: First render wont call cleanup function(i.e cleanup function won’t be called on page loading). This is because there Is nothing to clean on initial load.

Fetching with useEffect:

Fetchdata with useEffect:

const url = "https://api.github.com/users";

const **UseEffectFetchData** = () => {

  var [users, **setUsers**] = **useState**([]);

  function **DataGetter**() {

**fetch**(url)

      .**then**((response) => response.**json**())

      .**then**((data) => **setUsers**(data));

  }

**useEffect**(() => {

**DataGetter**();

  });

  return (

    <>

      <ul *className*="users">

        {users.**map**((user) => {

          return (

            <h1>{user.login}</h1>

          );

        })}

      </ul>

    </>

  );

};

Here useEffect calls DataGetter function which gets data with fetch operation.

This is converted to json and users is set to this data using setUsers.

But this has a problem.

We are calling DataGetter function in useEffect.

This DataGetter gets the data and changes the state of users variable.

This triggers rendering of component.

On rendering, we useEffect is invoked which again calls DataGetter.

DataGetter again changes the state which triggers render.

This causes a infinite loop of data fetching and rendering component.

useEffect🡪DataGetter()🡪fetchdata🡪setState of user variable 🡪Render component🡪useEffect.

To avoid this infinite loop, we place a empty list of dependencies, which makes the component to render only when the page is loaded instead of every change to state.

i.e,

const **UseEffectFetchData** = () => {

  var [users, **setUsers**] = **useState**([]);

  function **DataGetter**() {

**fetch**(url)

      .**then**((response) => response.**json**())

      .**then**((data) => **setUsers**(data));

  }

**useEffect**(() => {

**DataGetter**();

  },[]);

  return (

    <>

      <ul *className*="users">

        {users.**map**((user) => {

          return (

            <h1>{user.login}</h1>

          );

        })}

      </ul>

    </>

  );

};

But if we add users variable to the list of dependencies, the problem of infinite loop remains .

Because, now every change to users variable will trigger a render.

Read the following articles for understanding short circuiting:

<https://codeburst.io/javascript-short-circuit-conditionals-bbc13ac3e9eb>

<https://codeburst.io/javascript-short-circuit-evaluation-3709ffda6384>

const **ControlledInputs** = () => {

  var [name, **setName**] = **useState**("");

  var [email, **setEmail**] = **useState**("");

  return (

    <>

      <form *className*="form">

        <div *className*="form-control">

          <label *htmlFor*="">Name:</label>

          <input

*type*="text"

*id*="name"

*value*={name}

*onChange*={(e) => {

**setName**(e.target.value);

            }}

          />

        </div>

        <div *className*="form-control">

          <label *htmlFor*="">Email:</label>

          <input

*type*="text"

*id*="email"

*value*={email}

*onChange*={(e) => {

**setEmail**(e.target.value);

            }}

          />

        </div>

        <button

*type*="submit"

*onClick*={(e) => {

            e.**preventDefault**();

            console.**log**(e);

            console.**log**("onclick event");

          }}

        >

          add user

        </button>

      </form>

    </>

  );

};

Here for every change in input field,value of name(state variable ) Is assigned the value of input field at that point of time.

This changes the state and hence renders the component with new value of name.

Onchange, onclick events when triggered , generate a event object.

Now this event object has lot of properties.

Let e be the event object.

On console logging e.target of onchange event of input tag, we get:

<input type="text" id="name" value="adf">

This value is obtained by e.target.value.

To be short,

For every character we type/remove in the input field, the state variable name is changed using setName.

This setName gets the present value of the field by eventobject.target.value.

This event object is generated on every change of input field.

Note: whenever we click submit button, the page reloads..to prevent this, one can use e.preventDefault() function

const **ControlledInputs** = () => {

  var [name, **setName**] = **useState**("");

  var [email, **setEmail**] = **useState**("");

  var [users, **setUsers**] = **useState**([]);

  return (

    <>

      <form *className*="form">

        <div *className*="form-control">

          <label *htmlFor*="">Name:</label>

          <input

*type*="text"

*id*="name"

*value*={name}

*onChange*={(e) => {

**setName**(e.target.value);

              console.**log**("name render");

            }}

          />

        </div>

        <div *className*="form-control">

          <label *htmlFor*="">Email:</label>

          <input

*type*="text"

*id*="email"

*value*={email}

*onChange*={(e) => {

**setEmail**(e.target.value);

              console.**log**("email render");

            }}

          />

        </div>

        <button

*type*="submit"

*onClick*={(e) => {

            e.**preventDefault**();

            if (name && email) {

**setUsers**((users) => {

                return [...users, { name, email }];

              });

**setName**("");

**setEmail**("");

            }

          }}

        >

          add user

        </button>

      </form>

      {users.**map**((user) => {

        return (

          <div *className*="item">

            <h4>{user.name}</h4>

            <h4>{user.email}</h4>

            <button

*onClick*={() => {

                var filteredPeople = users.**filter**(

                  (item) => item.name !== user.name

                );

**setUsers**(filteredPeople);

              }}

*className*="btn"

            >

              remove

            </button>

          </div>

        );

      })}

    </>

  );

};

Onclicking add people button, we create a object and add it to users array.

Then we display each of the user enclosed in html tag.

const **ControlledInputs** = () => {

  var [person, **setPerson**] = **useState**({ name: "", email: "", age: 0 });

  const [people, **setPeople**] = **useState**([]);

  const **handleSubmit** = (e) => {

    e.**preventDefault**();

    if (person.name && person.email && person.age) {

      const person\_with\_id = { ...person, id: new **Date**().**getTime**().**toString**() };

      console.**log**(person);

**setPeople**((people) => {

        return [...people, person\_with\_id];

      });

**setPerson**({ name: "", email: "", age: 0 });

    } else {

      console.**log**("empty values");

    }

  };

  return (

    <>

      <article>

        <form *className*="form" *onSubmit*={**handleSubmit**}>

          <div *className*="form-control">

            <label *htmlFor*="firstName">Name : </label>

            <input

*type*="text"

*id*="name"

*name*="name"

*value*={person.name}

*onChange*={(e) => **setPerson**({ ...person, name: e.target.value })}

            />

          </div>

          <div *className*="form-control">

            <label *htmlFor*="email">Email : </label>

            <input

*type*="email"

*id*="email"

*name*="email"

*value*={person.email}

*onChange*={(e) => **setPerson**({ ...person, email: e.target.value })}

            />

          </div>

          <div *className*="form-control">

            <label *htmlFor*="age">Age : </label>

            <input

*type*="number"

*id*="age"

*name*="age"

*value*={person.age}

*onChange*={(e) => **setPerson**({ ...person, age: e.target.value })}

            />

          </div>

          <button *type*="submit">add person</button>

        </form>

        {people.**map**((person) => {

          console.**log**(person);

          const { name, email, age } = person;

          return (

            <div *className*="item">

              <h4>{name}</h4>

              <p>{email}</p>

              <p>{age}</p>

            </div>

          );

        })}

      </article>

    </>

  );

};

Here instead of creating 3 different state variables ,we are creating a single state object.

We update the corresponding property based on the input filed.

Forms with multiple inputs:

  const **handleSubmit** = (e) => {

    e.**preventDefault**();

**setPeople**([...people, person]);

  };

  function **handleChange**(e) {

    const name = e.target.name;

    const value = e.target.value;

    console.**log**("in change function", { ...person, [name]: value });

**setPerson**({ ...person, [name]: value });

  }

  return (

    <>

      <article>

        <form *className*="form" *onSubmit*={**handleSubmit**}>

          <div *className*="form-control">

            <label *htmlFor*="firstName">Name : </label>

            <input

*type*="text"

*id*="firstName"

*name*="firstName"

*value*={person.firstName}

*onChange*={(e) => **handleChange**(e)}

            />

          </div>

          <div *className*="form-control">

            <label *htmlFor*="email">Email : </label>

            <input

*type*="email"

*id*="email"

*name*="email"

*value*={person.email}

*onChange*={(e) => **handleChange**(e)}

            />

          </div>

          <div *className*="form-control">

            <label *htmlFor*="email">Email : </label>

            <input

*type*="number"

*id*="age"

*name*="age"

*value*={person.age}

*onChange*={(e) => **handleChange**(e)}

            />

          </div>

          <button *type*="submit" *onClick*={**handleSubmit**}>

            add person

          </button>

        </form>

        {people.**map**((person) => {

          const { firstName, email, age } = person;

          return (

            <div *className*="item">

              <h4>{firstName}</h4>

              <p>{email}</p>

              <p>{age}</p>

            </div>

          );

        })}

      </article>

    </>

  );

};

Here name property of input tag plays a key role.

For every input tag,we submit a corresponding property in object

For example, if we have 3 input tags with names firstName, email and age,

Then we will create a person object with same names as properties. i.e, {firstName,email,age}.

On handleChange function is called for every change in input field, which retrieves the name and value of input field at that time by using e.target.

Then using object destructuring, corresponding property of object is changed.

Since this is done using setPerson, the component renders with new value of input fields.

Then on clicking Addperson button, handleSubmit is called which destructures the people array and returns the array with new person object added

    const name = e.target.name;

    const value = e.target.value;

**setPerson**({ ...person, [name]: value });

for input field Name with value “as” will have the above statement as:

setPerson({…person,firstName:”as”})

for input field Age with value 90 will have the above statement as:

setPerson({…person,age:90})

useRef works like useState but it doesn’t trigger a render on change in state value.