***Use State***

* *useState() is a hook that allows you to have state variables in functional components.*
  + *It returns two items* 
    - *State Object(getter)*
    - *Updater function (setter)*

|  |
| --- |
| function Button() {  const [counter, setCounter] = useState(1)  return <button onClick={() => setCounter(counter+counter)}>{counter}</button>;  }  ReactDOM.render(  <Button />,  document.getElementById('mountNode'),  ); |

* *3 ways to rendering multiple items*
  + *render multiple items using array*
    - *this is good option when renderings are coming from same component in dynamic way*
    - *not ideal most of the time*

ReactDOM.render(

[<Button />, <Display/>]

document.getElementById('mountNode'),

);

* + *Render using tree structure*

|  |  |
| --- | --- |
| ReactDOM.render(  <React.Freagment>  <Button />  <Display />  </React.Freagment>,  document.getElementById('mountNode'),  ); | ReactDOM.render(  < >  <Button />  <Display />  </ >,  document.getElementById('mountNode'),  ); |

* + *Render using function which include all functions*

|  |
| --- |
| function App(){  return(  <div>  <Button />  <Display />  </div>  )  }  ReactDOM.render(  <App />,  document.getElementById('mountNode'),  ); |

* *To make state variables available in other components, declare the variable in its parent component. Then it’s available in all child components* 
  + *Parent component can flow both data and components into its child component*

|  |
| --- |
| *function* Button(*props*){  return (  <button onclick={props.onclickFunction}>  +1  </button>  );  }  *function* Display(*props*){  return (  <div>{props.message}</div>  );  }  *function* App(){  *const* [counter, setCounter] = useState(43);  *const* incrementCounter = () *=>* setCounter(counter+1);  return(  <div>  <*Button* onclickFunction={incrementCounter}/> //it flows component to button(child) component  <*Display* message={counter}/> // it flows data to button(child) component  </div>  );  }  ReactDOM.render(  <*App* />,  document.getElementById('mountNode')  ) |

* *String is immutable in JavaScript*
* *“const” 🡪 is immutable and you cannot change the reference either*
  + *If the “const” refer to an Array or object, then you cannot change the reference, but you can change object and Array values*

|  |  |  |
| --- | --- | --- |
| ***Var*** | ***Let*** | ***Const*** |
| *var and let are both used for function declaration in JavaScript* | |  |
| *var is function scoped* | *let is block scoped.* | * *const assigned to a variable is meant to be a constant one* * *References assigned with const cannot be changed* * *The const will guarantee that the variable is pointing to the same array or object, but the content of the array or object can still be mutated* |

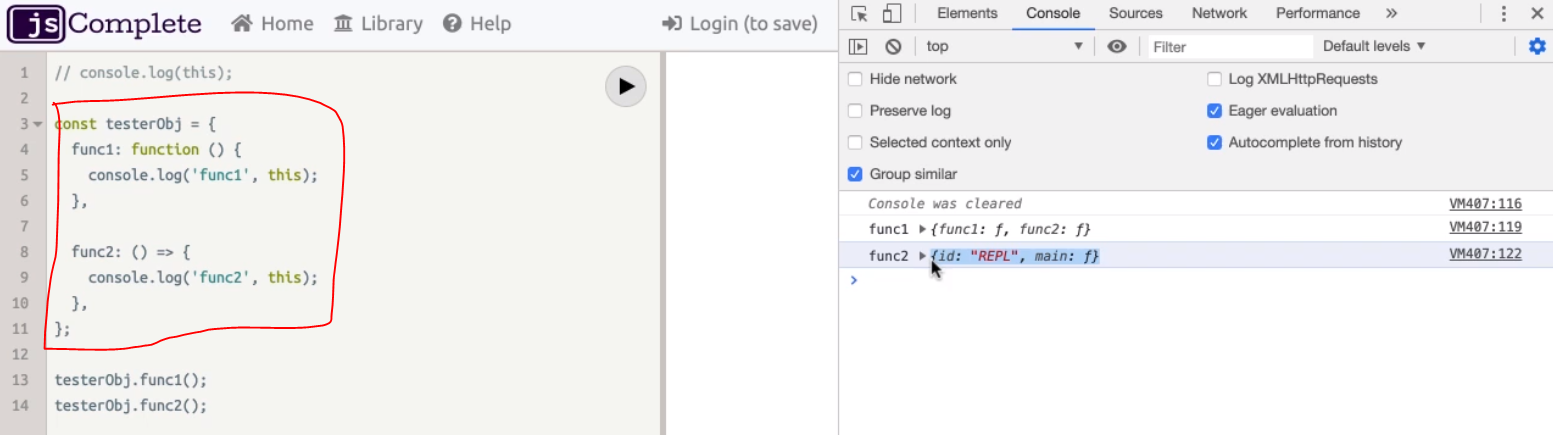
***Arrow Functions***

* *It is a way to define a function without typing the keyword function, but rather by using an arrow symbol*

***const* y = () *=>* { } ;**

* *An arrow function does not care who calls it, while a regular function cares very much about that.*
* *Regular functions give access to their “calling” environment while arrow functions give to their “defining” environment*
* *The value of the “this” keyword inside a regular function depends on how the function was called (the object that made the call)*
* *The value of the “this” keyword inside an arrow function depends on WHERE the function was DEFINED (the SCOPE that defined the function)*

*Ex:*



* [*the top-level*](https://app.pluralsight.com/player?course=react-js-getting-started&author=samer-buna&name=react-js-getting-started-m2&clip=2&mode=live&start=86.3751818181818)[*this keyword is associated with a special*](https://app.pluralsight.com/player?course=react-js-getting-started&author=samer-buna&name=react-js-getting-started-m2&clip=2&mode=live&start=88.90128571428572)[*object, which has the id of REPL*](https://app.pluralsight.com/player?course=react-js-getting-started&author=samer-buna&name=react-js-getting-started-m2&clip=2&mode=live&start=92.20866666666664)
* *tester object that defines two similar functions. In both functions*
* *When function 1 is called, it's this keyword will be associated with its caller, which, in this case, is the tester object itself. Therefore, you see the printed value for this keyword in function 1 representing the tester object itself.*
* *when function 2 is called, it's this keyword will be associated with the same this keyword that was available in the function's scope when it was defined. It was this playground's REPL object that we've seen on the top level.*
* *This is a big benefit when working with listeners and event handlers,*
* *One other cool thing about arrow functions is that if a function only has a single line that returns something, you can make it even more concise by removing the curly braces and the return keyword altogether.*

*const* square = (*a*) *=>* {

return a\*a;

};

OR

*const* square = (*a*) *=>* return a\*a;

* *You can also remove the parentheses around the argument if the function receives a single argument,*

*const* square = *a* *=>* return a\*a;

* *This syntax is usually popular for functions that get passed to array methods, like map, reduce, and filter, and functional programming in general.*

[1,2,3,4].map(*a* *=>* a\*a)

***Object Literals***

* *object literal is a comma-separated list of name-value pairs wrapped in curly braces*
* *Literal initiation is very common in JavaScript.*
* *We use it for objects, arrays, strings, numbers, and even things like regular expressions.*
* *a simple example where this object defined two regular properties.*

|  |
| --- |
| *const* obj = {  p1: 10,  p2: 20,  f1() {},  f2: () *=>* {},  }; |

}

* *If you need to define a property that holds a function, you can use this short syntax with object literals*
* *if you need an arrow function, you can still use this regular property syntax.*
* *Modern object literals also support dynamic properties using this syntax.*
* *It looks like an array literal, but don't confuse it with that.*

|  |  |
| --- | --- |
| *const* mystery = 'answer';  *const* obj = {  p1: 10,  p2: 20,  f1() {},  f2: () *=>* {},  [mystery]: 42,  }  *console*.log(obj.mystery);  *console*.log(obj.answer); | *Output: undefined*  *Output: 42* |

* *assume we have a variable named mystery defined before this X object,*
* *here's a JavaScript interview question.*

*What is the value of obj.mystery?*

* + *It's undefined because this mystery property was defined with a dynamic property syntax.*
  + *the object will have a property namyupped answer with the value of 42.*
* *Another widely popular feature about object literals is available to you when you need to define an object with property names to map values that exist in the current scope with the exact same name.*

|  |
| --- |
| *const* mystery = 'answer';  *const* InverseOfPI = 1/ Math.PI  *const* obj = {  p1: 10,  p2: 20,  f1() {},  f2: () *=>* {},  [mystery]: 42,  //InverseOfPI: InverseOfPI;  InverseOfPI;  }  *console*.log(obj.mystery); |

* *Here's an example. If we have a variable named InverseOfPI and we would like obj here to have a property named InverseOfPI holding the same value as the variable InverseOfPI, instead of typing that name twice, you can use the shorter syntax by removing the second part.*

***Destructuring and Rest/Spread***

* *Destructuring mean get the particular value from object literals*
* *Destructuring works for both arrays and objects.*
* *Here's an example for objects using the built-in Math object in JavaScript.*

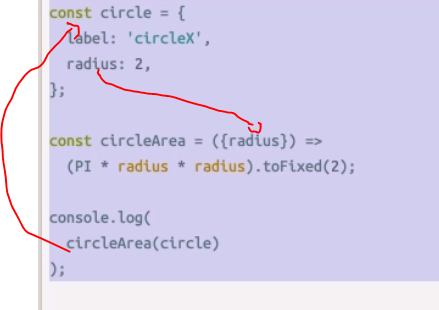
|  |
| --- |
| *const* PI = Math.PI  *const* E = Math.E  *const* SQRT2 = Math.SQRT2  (or)  *const* {PI, E, SQRT2} = Math; |

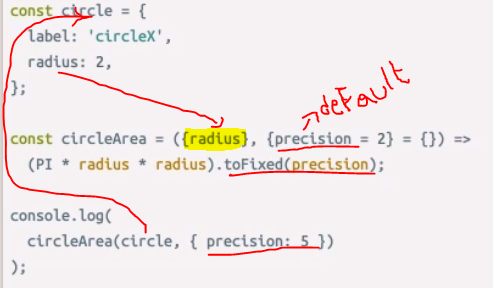
* *here's a line to destructure Component, Fragment, and useState out of the React API.*

*const* {Component, Fragment, useState} = require('react');

useState();

* *After this line, I can use the useState()directly like above.*
* *Destructuring also works inside function arguments. If the argument passed to a function is an object, instead of using the name of the object every time you want to access its properties, you can use the destructuring syntax within the function parentheses to destructure just the properties that you're interested in and make them local to that function.*





* *Second argument is default value{precision = 2}*
  + *here we have a circleArea function which expects an object as its argument,*
  + *and it expects that object to have a radius property.*
  + *destructuring the radius property out of that object and using it locally in the function.*
  + *If we call this circleArea function with an object like circle 🡪  it will use its radius property inside of its calculation.*
* *Destructured arguments can also be defined with defaults like regular arguments.* 
  + *Ex: I'd like to use a default value of 2 for a precision property here.*
* *destructuring feature offers a good alternative to using named arguments in functions, which is much better than relying on positional arguments.*

*Array Destructuring*

* *Destructuring, whether you do it in function arguments or directly with variables, also works for arrays.*
* *If you have an array of values and you want to extract these values into local variables, you can use the item's positions to destructure their values into local variables just like this.*

*const* [first, second,, forth] = [10, 20, 30, 40];

* *Note double commas here to skip destructuring the third item in the array.*

*So this array will contain [10, 20, 40],The destructured variable fourth here will hold the value of 40.*

* *Array destructuring is useful when combined with the rest operator*

*const* [first, ...restOfItems] = [10, 20, 30, 40];

*console*.log(firsrt);

*console*.log(restOfItems)

#output

10

[20, 30, 40] 🡪 *restOfItems*

* *By using these three dots, we are asking JavaScript to destructure only one item out of this array and then create a new array under the name restOfItems to hold the rest of the items after removing the first one.*
* *This is powerful for splitting the array, it very useful when working with objects to filter out certain properties from an object.*



* *Say that we have this data object that has a few temp properties,*
* *we'd like to create a new object that has the same data, except for temp1 and temp2.*
* *We can destructure temp1 and temp2 and then use the rest operator to get the remaining properties into a new object called person.*

*const* {temp1, temp2, ...person} = date;

*const* newArray=[...restofItems];

*const* newObject = {

...person

}

* *Just like the three dots of rest, you can use the three dots to spread one array or object into a new array or object.*
* *This is useful for copying arrays and objects.*
* *New array here will be a copy of the restOfItems array that we destructured above.*
* *Similarly, you can also spread the key-value pairs of an object into a new object,*
* *Note that these copies are shallow copies. Any nested objects or arrays will be shared between these copies. Don't forget that.*

*Template Strings*

* *You can define strings in JavaScript using either single quotes or double quotes.*
* *JavaScript has a third way to define strings, and that's using the backtick character*

|  |
| --- |
| *const html =`<div>${Math.random()}</div>`;* |

* *Strings defined with the backtick character(`) are called template strings because they can be used as a template with dynamic values. They support what we call interpolation.*
* *You can inject any dynamic expression in JavaScript within these dollars sign curly braces holders.*
* *you can also have multiple lines in the string. Something that was not possible with the regular, quoted strings.*

|  |
| --- |
| *const html =`*  *<div>*  *${Math.random()}*  *</div>*  *`;* |

* *Backticks look very similar to single quotes, so make sure to train your eyes to spot template strings when they are used in examples.*

*Classes*

|  |
| --- |
| *class* Person{  *constructor*(*name*){  this.name = name;  }  greet(){  *console*.log('Hello ${this.name}!');  }  }  *class* Student extends *Person*{  *constructor*(*name*, *level*){  super(name);  this.level = level;  }  greet(){  *console*.log(`Hello ${this.name} from ${this.level}`);  }  }  *const* o1 = new Person("Sandeep");  *const* o2 = new Student("Aarvi", "1st Grade");  *const* o3 = new Student("Kalyani", "MTech");  o3.greet = () *=>* *console*.log(`I am Special`);  o1.greet();  o2.greet();  o3.greet();  output  Hello Sandeep!  Hello Aarvi from 1st Grade  Hello Kalyani from MTech |

* *JavaScript offers many programming paradigms, and object-oriented programming is one of them. Everything in JavaScript is an object, including functions.*
* *Modern JavaScript also added support for the class syntax. A class is a template, or blueprint, for you to define shared structure and behavior between similar objects.*
* *You can define new classes, make them extend other classes, and instantiate objects out of them using the new keyword.*
* *You can customize the construction of every object and define shared functions between these objects.*

***Promises and Async, Await***

* *When you need to work with asynchronous operations, you usually must deal with promise objects.*
* *A promise is an object that might deliver data at a later point in the program.*
* *An example of an async function that returns a promise is the web fetch API that's natively available in some browsers.*

|  |
| --- |
| *const* fetchData = () *=>* {  fetch('https://apt.github.com').then(*resp* *=>* {  resp.json().then(*data* *=>* {  *console*.log(data)  });  });  };  fetchData(); |

* *Here we're fetching information from the top-level GitHub API. Since fetch returns a promise then call on the result of fetch and supply a callback function in here.*
* *This callback function will receive the data from the API. The fetch API has a raw response.*
* *If you need to parse the data as JSON, you need to call the JSON method on the response object, and that Json method is also an asynchronous one, so it returns a promise as well.*
* *To get the data, we need another. then call on the result of the Json method, and in the callback of that, we can access the data.*
* *As you can see, this syntax might get complicated with more nesting of asynchronous operations or when we need to combine this with any looping logic.*
* *You can simplify the nesting here by making each promise callback return the promise object*

|  |
| --- |
| *const* fetchData = async () *=>* {  *const* resp = await fetch('https://apt.github.com');  *const* data = await resp.json();  *console*.log(data);  };  fetchData(); |

* *You just await on the asynchronous call that returns a promise, and that will give you back the response object. Then you can await on the Json method to access the JSON data, just like this.*
* *And to make these await calls, you need to label the function as async, and this will work the same.*
* *The async await syntax is just a way for us to consume promises without having to nest. then calls.*
* *It's a bit simpler to read, but keep in mind that once you await on anything, in a function like fetch Data here, this function itself becomes asynchronous, and it will return a promise object.*

*Difference between class component and functional component*

|  |  |
| --- | --- |
| *Functional component* | *Class Component* |
| *function* Hello(*props*){  return <div> Hello!! {prop.name} </div>    }  *A functional component is just a plain JavaScript function which accepts props as an argument and returns a React element.* | *class* Hello extends React.*Component*{  render(){  return <h1> Hello, {this.props.name} </h1>  }  }  *class component requires you to extend from React.Component and create a render function which returns a React element* |
| *functional components are stateless, you cannot use setState() in your component* | *If you need a state in your component you will either need to create a class component or you lift the state up to the parent component and pass it down the functional component via props.* |
| *you cannot use in functional components are lifecycle hooks.*  *The reason is the same like for state, all lifecycle hooks are coming from the React.Component which you extend from in class components.* | *if you need lifecycle hooks you should probably use a class component.* |

*Events*

* The events generated by the browser DOM we will call DOM events. This includes things like button clicks, scrolling and change events.
* When we create our own components, we can give them their own events. We will call these component events.

DOM Events

* DOM Events are the events generated by the browser.
* Things like button click events, change events for text inputs and form submission events.
* All browsers provide an event-based programming model. They all provide a similar set of events that behave in a similar way. So that the programmer does not have to understand and account for the specific quirks for each browser,
* React provides a normalized event abstraction called Synthetic Events.
* DOM Events are the events that receive a react Synthetic Events object.

function Events(props){

const clickHandler = (synthEvent) => {

console.log(synthEvent)

};

return (<button onclick={clickHandler}> Make an event</button>)

}

*Prevent Default*

* *When an event occurs, it may be desirable to handle the event and prevent the browser from doing what it would normally do.*
* *The SyntheticEvent object has a prevent default method like preventing forms from submitting.*
* *The default behavior of a form is to submit the page and reload.*
* *In a single page application this is usually undesirable, so we handle the form submission, cancel the browsers default behavior using the prevent default method and submit the data as an Ajax request that does not result in the page reloading.*

function Nocheckbox(){

return <input type="checkbox" onclick={(e) => {e.preventDefault();}}/>;

}

ReactDOM.render(<Nocheckbox />,

document.getElementById('root'));

* *example as shown here is the check box input. When clicked the browsers default behavior is the check the check box. This code calls preventDefault which will stop the check box from being checked when it is clicked.*
* *another example.*

class Reloader extends React.Component{

constructor(props){

super(props);

this.state = {content:""};

this.onChar = this.onChar.bind(this);

this.onGoTime = this.onGoTime.bind(this);

}

onchar(event){

this.setState({content: event.target.value});

}

onGoTime(event){

if(this.state.content !== "reload"){

event.preventDefault();

}

}

render(){

return(

<form onSubmit ={this.onGoTime}>

<input type="text" value={this.state.content} onChange={this.onChar}/>

<input type="submit" value = "Go Time" />

</form>);

}

}

* *This one is a Java script class style react component called reloader.*
* *Let's start by looking at the render method and we can see that the reloaded component brings a form. The form has an event handler for the on submit DOM event. It also has an event handler for the on changer events of its text input which is bound to the content property of the components state. And finally there's a submit button for submitting the form. The onChar event handler that gets called each time the user types a character into the text box, updates the component state with the current content of the input. The event handler that's registered for the form submission checks what the state of the content currently is and if it's not equal to reload then it cancels thedefault behavior for this event. Now the default behavior for form submission is that the page will reload. In effect this means that we have a text box that we can type into. When we submit the form, under normal circumstances the default behavior should be prevented and nothing should happen, but if the content of the text box is reload, then the default behavior won't be granted and the browser will reload. Let's try that. I'll submit the form and you can see that nothing happens. The page is not reloaded because the defaultbehavior is prevented. This time I set the value of the value of the input to string reload. When I click the button, the page is reloaded.*