USING LIT

LESSON 1

```
import {LitElement, html} from 'lit';

export class MyElement extends LitElement {
Static properties defines variable names and is located INSIDE the MyElement class
static properties = {
  version: {},
 };

Constructor sets the variable values and is located INSIDE the MyElement class
 constructor() {
  super();
  this.version = 'STARTING';
 }

Render sets the html for what is displayed and is located INSIDE the MyElement class
${ } is a placeholder
 render() {
  return html`
  <p>Welcome to the Lit tutorial!</p>
  <p>This is the ${this.version} code.</p>
  `;
 }
}
Define registers the new element and is located OUTSIDE the MyElement class
customElements.define('my-element', MyElement);
```

```
<!DOCTYPE html>
<html>
 <head>
You need to reference the javascript file
  <script type="module" src="my-element.js"></script>

  <style>
   body {
    font-family: 'Open Sans', sans-serif;
    font-size: 1.5em;
    padding-left: 0.5em;
   }
  </style>

 </head>
 <body>
Then just use the element tag
  <my-element></my-element>
 </body>
</html>
```

LESSON 2 – DEFINING AN ELEMENT

```
import {LitElement, html} from 'lit';


This defines the element
export class MyElement extends LitElement {
This defines the html that will be rendered for the element
render() {
  return html`
    <p>Hello world! From my-element.</p>
  `; }
}
```

Notice that the render method is INSIDE the MyElement class

Notice that the define method is OUTSIDE the MyElement class and at the end of the module.

```
customElements.define('my-element', MyElement);
```

```
<!DOCTYPE html>
<html>
 <head>
You need to reference the javascript file
  <script type="module" src="my-element.js"></script>

  <style>
   body {
    font-family: 'Open Sans', sans-serif;
    font-size: 1.5em;
    padding-left: 0.5em;
   }
  </style>

 </head>
 <body>
Then just use the element tag
  <my-element></my-element>
 </body>
</html>
```

LESSON 3 – ADDING PROPERTIES

Components can have STATIC properties and REACTIVE properties

A static property is a variable. It is given a name then allotted a value

Here the property is given a name

```
static properties = {
   message: {},
 };
```

Here the property is assigned a value using constructor

```
 constructor() {
   super();
   this.message = 'Hello again.';
 }
```

These are placed INSIDE the MyElement Class

```
import {LitElement, html} from 'lit';


export class MyElement extends LitElement {
 static properties = {
  message: {},
 };
 constructor() {
  super();
  this.message = 'Hello again.';
 }
 render() {
  return html`
   <p>${this.message}</p>
  `;
 }
}
customElements.define('my-element', MyElement);
```

Once again ${ } is a placeholder

Using constructor on its own will be sufficient to define a literal, but to create a dynamic property you must use the @properties as above.

```
<!DOCTYPE html>
<html>
 <head>
  <script type="module" src="my-element.js"></script>
  <style>
   body {
    font-family: 'Open Sans', sans-serif;
    font-size: 1.5em;
    padding-left: 0.5em;
   }
  </style>
 </head>
 <body>
  <my-element></my-element>
 </body>
</html>
```

LESSON 4 – Event Listeners

The only new items here are that we are adding an event listener, and an event handler

```
import {LitElement, html} from 'lit';
export class NameTag extends LitElement {
 static properties = {
  name: {},
 };

 constructor() {
  super();
  this.name = 'Your name here';
 }
```

Here the event listener is placed inside the input tag

```
 render() {
  return html`
   <p>Hello, ${this.name}</p>
   <input @input=${this.changeName} placeholder="Enter your name">
  `;
 }
```

Here the event handler takes the input from the event target, which in this case is the input box

```
 changeName(event) {
  const input = event.target;
  this.name = input.value;
 }
}
customElements.define('name-tag', NameTag);
```

An event listener can also be attached to a button

```
<button @click=${this.handleClick}>Click me!</button>
```

Here I use an event for the input box to change a variable value, then use a click event to assign that variable to a placeholder.

```
import {LitElement, html} from 'lit';

export class NameTag extends LitElement {
 static properties = {
  name: {},
identity: {},
 };

 constructor() {
  super();
  this.name = 'Your name here';
  this.identity = "";
 }

 render() {
  // TODO: Add declarative event listener to input.
  return html`
   <p>Hello, ${this.identity}</p>
   <input id="Go" @input=${this.changeName} placeholder="Enter your name">
    <button @click=${this.handleClick}>Click me!</button>
  `;
 }
changeName(event) {

 const input = event.target;
 this.name = input.value;
}
```

```
handleClick(event) {


  this.identity = this.name ;
}
}
customElements.define('name-tag', NameTag);
```

```
import {LitElement, html} from 'lit';

export class NameTag extends LitElement {
 static properties = {
  name: {},
 };

 constructor() {
  super();
  this.name = 'Your name here';
 }

 render() {
  // TODO: Add declarative event listener to input.
  return html`
   <p>Hello, ${this.name}</p>
   <input id="Me" placeholder="Enter your name">
    <button @click=${this.handleClick}>Click me!</button>
  `;
 }
```

This is another way of getting the value from an input box to use elsewhere

```
get input() {
 return this.renderRoot?.querySelector('#Me') ?? null;
}

handleClick() {
this.name = this.input.value;}
}
customElements.define('name-tag', NameTag);
```

LESSON 5 – Using an expression to set an attribute value

An expression can add an event listener, or can change an attributes value.

```
import {LitElement, html} from 'lit';

export class MoreExpressions extends LitElement {
 static properties = {
  checked: {},
 };

 constructor() {
  super();
  this.checked = false;
 }
```

A ? makes an attribute on or off based on a Boolean value that is set by another object

```
 render() {
  return html`
   <div>
    <!-- TODO: Add expression to input. -->
    <input type="text" ?disabled=${!this.checked} value="Hello there.">
   </div>
   <label><input type="checkbox"  @change=${this.setChecked}> Enable editing</label>
  `;
 }

 setChecked(event) {
  this.checked = event.target.checked;
 }
}
customElements.define('more-expressions', MoreExpressions);
```

LESSON 6 Building a List

```
import {LitElement, html} from 'lit';

This sets the state to internal reactive, so its state is private
export class ToDoList extends LitElement {
 static properties = {
  _listItems: {state: true},
 };

This shows how to preset text items for a list
 constructor() {
  super();
  this._listItems = [
   {text: 'Start Lit tutorial', completed: true},
   {text: 'Make to-do list', completed: false},
  ];
 }

This maps an item to html according to the format  `<li>${item.text}</li>`

 render() {
  return html`
   <h2>To Do</h2>
   <ul>
    ${this._listItems.map((item) => html
    `<li>${item.text}</li>`
    )}
   </ul>
   <input id="newitem" aria-label="New item">
   <button @click=${this.addToDo}>Add</button>
```

```
    `;
  }
```

...this._listitems is the current list

Text: this.input.value is what is added

Completed:false means you can continue adding

This.input.value =’’ clears the input box

Note that the input box is made available by the get input() method

And is then read by this.input.value

And is then cleared by this.input.value = “”

We can use this method to get the value of any element simply by using its ID, and changing the get input() to for example get input2()

```
 addToDo() {
this._listItems = [...this._listItems, {text: this.input.value, completed: false}];
  this.input.value = '';}
```

This extracts the input value from the input box by ID. Note the # before the ID.

```
get input() {
  return this.renderRoot?.querySelector('#newitem') ?? null;
}
```

LESSON 7 Adding Styles

```
import {LitElement, html, css} from 'lit';


export class ToDoList extends LitElement {
 static properties = {
  _listItems: {state: true},
 };
```

This sets the css where class .completed is true

```
 static styles = css`
 .completed {
  text-decoration-line: line-through;
  color: #777;
 }


 constructor() {
  super();
  this._listItems = [
   {text: 'Make to-do list', completed: false},
   {text: 'Add some styles', completed: false},
  ];
 }
```

Here we have added a class and a click event listener WITHIN the <li> tag

class=${item.completed ? 'completed' : ''} determines whether class .completed is true. Default is set to false.

@click=${() => this.toggleCompleted(item)}> ${item.text}

On click,

These 2 events are called **${() => this.toggleCompleted(item)}> ${item.text}**

The first event is the toggleCompleted event

The second event is a placeholder that inserts the item text again.

```
 render() {
```

```
    return html`
      <h2>To Do</h2>
      <ul>
        ${this._listItems.map((item) => html
`<li class=${item.completed ? 'completed' : ''} @click=${() => this.toggleCompleted(item)}>
${item.text} </li>`
        )}
      </ul>
      <input id="newitem" aria-label="New item">
      <button @click=${this.addToDo}>Add</button>
    `;
  }
```

This event handler reverses the item.completed status, and updates it.

```
  toggleCompleted(item) {
    item.completed = !item.completed;
    this.requestUpdate();
  }

  get input() {
    return this.renderRoot?.querySelector('#newitem') ?? null;
  }

  addToDo() {
    this._listItems = [...this._listItems,
      {text: this.input.value, completed: false}];
    this.input.value = '';
  }
}
customElements.define('todo-list', ToDoList);
```

Lesson 8

```
import {LitElement, html, css} from 'lit';

export class ToDoList extends LitElement {
 static properties = {
  _listItems: {state: true},
  hideCompleted: {},
 };
 static styles = css`
  .completed {
   text-decoration-line: line-through;
   color: #777;
  }
 `;

 constructor() {
  super();
  this._listItems = [
   {text: 'Make to-do list', completed: true},
   {text: 'Complete Lit tutorial', completed: false},
  ];
  this.hideCompleted = false;
 }

 render() {
  const items = this.hideCompleted
   ? this._listItems.filter((item) => !item.completed)
   : this._listItems;
  const todos = html`
   <ul>
```

```
    ${items.map(
      (item) => html`
        <li
          class=${item.completed ? 'completed' : ''}
          @click=${() => this.toggleCompleted(item)}>
         ${item.text}
        </li>`
    )}
  </ul>
 `;
 const caughtUpMessage = html`
  <p>
  You're all caught up!
  </p>
 `;
 const todosOrMessage = items.length > 0 ? todos : caughtUpMessage;

 return html`
  <h2>To Do</h2>
  ${todosOrMessage}
  <input id="newitem" aria-label="New item">
  <button @click=${this.addToDo}>Add</button>
  <br>
  <label>
   <input type="checkbox"
    @change=${this.setHideCompleted}
    ?checked=${this.hideCompleted}>
   Hide completed
  </label>
 `;
}
```

```
 toggleCompleted(item) {
  item.completed = !item.completed;
  this.requestUpdate();
 }

 setHideCompleted(e) {
  this.hideCompleted = e.target.checked;
 }

 get input() {
  return this.renderRoot?.querySelector('#newitem') ?? null;
 }

 addToDo() {
  this._listItems = [
   ...this._listItems,
   {text: this.input.value, completed: false},
  ];
  this.input.value = '';
 }
}
customElements.define('todo-list', ToDoList);
```