How to handle events

Let's start off with a simple event and then build up to some more complex ones.

Responding to a simple click

1. Make sure that PickSeats is your startup component and run the site. Show the browser's console.
2. Click/tap on the seat. Did anything happen? It shouldn't have … we haven't added a click event yet.
3. Add a little JavaScript to the top of your component.

```
let table = {id: 0, table_number: 0, x: 1, y: 1, seats: []};
let seat = { id: 0, seat_number: 0, price: 10.75 };
let currentShowing = {id:0, film_id:0, theater_id:0, showing_time:new Date() };
let currentFilm = { title: "A Cool Movie" };
let currentTheater = { id: 0, name: "Theater #1" };
```

   (Note: A few of these are in preparation for future labs so don't worry if the IDE says they're not used yet).

4. And add a little more inside of your component.

```
function reserveSeat(seat) {
  console.log(seat)
  store.dispatch(actions.addSeatToCart(seat, currentShowing));
}
```

   (Hint: You will want to import store and actions like we've done in previous labs).

5. Find the that shows the seat itself. (Hint: it's the one with a style of seatItself). Add a click event to it. Call reserveSeat when tapped.
6. Test it out. Look at the console.log. What was console.log()ged? _____
7. That's cool but clearly we wanted the seat to be logged and not the event object. How do we get the seat object to be passed into reserve(seat)?
8. Change the event to use an arrow function to pass the seat into reserveSeat. You'll know you have it right when you see the seat (with the id, seat_number, and price) in the console.

Practicing with the event object

9. Make Account your startup component. Take a look at it in the browser and in the IDE.

Account has a <form> that can be submitted. You'll be tempted to wire up the onClick event on the submit button. That isn't the best practice. Instead you should wire up the onSubmit event on the <form> itself.

10. Add an onSubmit event that calls a function named register and of course add a register function that will simply console.log('hello world').
11. Run and test. Can you see 'hello world' in the console when you submit the form? Don't blink!
12. What is happening? HTML is doing what it was programmed to do: When a form is submitted it posts back to the server. We want to prevent that.
13. You can prevent a postback in the javascript by calling e.preventDefault() where "e" is the event object.
14. So write register like this:

```
function register(e) {
```

```
  e.preventDefault();
  console.log("hello world");
}
```

15. Try it again. You should see hello world stay because we're not posting back.
16. What you just did with Account … do the same thing with the Login component's form.

Bonus! Making these actions do something.

If you are finished early, add these steps in so that our form submits will do some useful work through our Redux store.

(Hint: for both of these, you'll need to do some imports.)

17. Have Account's register function also add a new user:

```
function register(e) {
  e.preventDefault();
  const user = {
    email: e.target['email'].value,
    password: e.target['password'].value,
    name: { given: e.target['given'].value, family: e.target['family'].value },
    phone: e.target['phone'].value,
    credit_card: {
      number: e.target['number'].value,
      expiration: e.target['expiration'].value
    },
  };
  store.dispatch(actions.register(user));
}
```

18. And have Login's login function do this:

```
function login(e) {
  e.preventDefault();
  store.dispatch(actions.login({
    email: e.target['email'].value,
    password: e.target['password'].value }));
}
```

19. To test them, just click/tap the submit buttons and check the console to make sure that actions got dispatched