How to display arrays

Alright, now the React fun really begins! This is where we experience the power of iterating objects, allowing us to create multiple instances of JSX, one set for each thing in a collection. We'll start with the films in LandingPage.

Showing multiple films in LandingPage

1. Open LandingPage in the browser and the IDE. Hey, we're showing only the first film in state.films!
2. Locate the JSX where that film card is being shown. It is probably a <section> whose style is styles.wrapper.
3. Create a JSX expression that iterates state.films. Here's a suggested starter:

```
{state.films.map(film => (
  <p>{film.title}</p>
))}
```

4. Run and test. You'll see all the films names in your browser.
5. Add a key. film.id would be a good one.
6. Replace the <p> with all the JSX for one film card.
7. Run and test. Adjust until you see every film in its own card. Just for fun adjust the size of your browser window. The cards are responsive if you wrote the styles for it to wrap.

Adding a day picker to LandingPage

We want to add a list of days so that the user can tap on one as if to say "I want to see a movie on this day."

8. You should have already imported the helpers/Date.js file into App.js. If not, do so now. It adds a static method to JavaScript's Date class called getArrayofDays() and adds some methods to Date's prototype.
9. Edit LandingPage.js. Use Date.getArrayOfDays(7) to create an array of the next seven days.
10. Using an expression (hint: curly braces in the JSX) map over that array and add a for each day just below the "Showings for …" line.
11. Run and test. Make sure you can see the days.
12. We don't want to see the long, ugly date, we just want to see the abbreviated day of the week. So call theDate.toShortDayOfWeekString() on each one. Here's some code that will work but see if you can solve It yourself without peeking.

```
<div style={styles.pickDateWrapper}>
  {Date.getArrayOfDays(7).map(date => (
    <span
      style={styles.days}
      key={date.getTime()}>
      {date.toShortDayOfWeekString()}
    </span>
  ))}
</div>
```

13. Run and test. Make sure you're just seeing the abbreviated day. It doesn't have to do anything yet. We'll handle that later.
14. Bonus! Apply these styles to make it look good.

```
pickDateWrapper: {
```

```
  backgroundColor: 'rgba(0,0,0,0.1)',
  padding: '1em',
},
days: {
  color: 'rgba(0,0,0,0.75)',
  fontSize: '1.2em',
  padding: '1em',
  cursor: 'pointer',
}
```

Creating tables and seats in PickSeats

15. Open PickSeats in both the browser and the IDE.
16. We're hardcoding a number of things currently. Let's actually read it from the store. Do this:

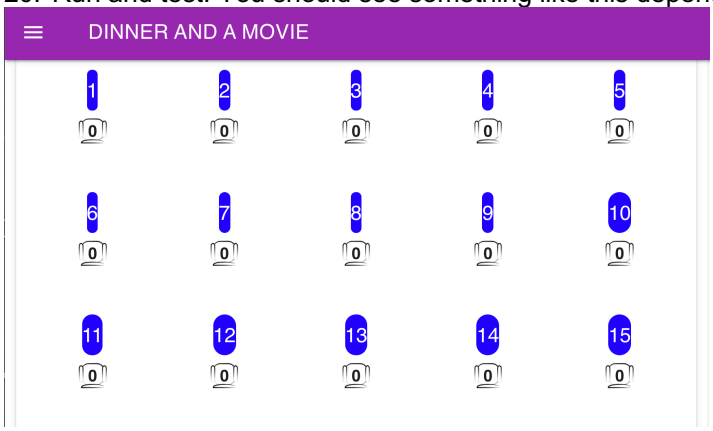
```
const state = store.getState()
// If state.showings doesn't exist, we can't draw anything ... yet.
// But in App.js, we're dispatching fetchShowings() and rerendering
// when a store.dispatch() happens so this component will in turn
// be rerendered once showings are populated.
if (state.showings && state.showings.length) {
  currentShowing = state.showings.find(showing => showing.id === +showingId);
  currentFilm = state.films.find(film => film.id === currentShowing.film_id);
  currentTheater = state.theaters.find(theater =>
                            theater.id === currentShowing.theater_id) || {};
}
const tables = currentTheater && currentTheater.tables;
```

If you were to console.log(tables), you'd see that the component actually draws twice, once with empty data … but then that useEffect fires, the one we wrote with a store.dispatch() in it. The next time it re-renders, the tables are populated.

17. Find where it says LIST OF TABLES WILL GO HERE. After that line, put this:

```
{tables && tables.map(table => (<>{table.table_number}</>))}
```

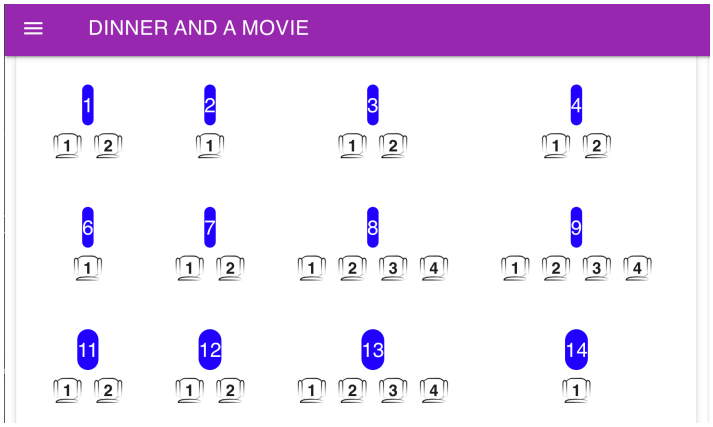
18. Run and test. You should see a bunch of numbers in the browser.
19. Let's draw some tables! Instead of rendering just the table_number, render the entire JSX for the table which you'll find after the " Here is one table: "
20. Run and test. You should see something like this depending on your styling:



Iterating the seats

We've got a bunch of tables so let's show each of their seats.

21. You have a that defines one seat. It has a style of styles.seatWrapper. Locate that seat.
22. Iterate that seat , one for each table.seats.
23. Run and test. You should now be seeing 1 to 4 seats behind each table. You'll see something like this:



Bonus! Size the tables

The tables are too skinny. Let's make them as wide as they need to be with some clever styling.

24. Find the table itself. It'll look like this:

```
<div style={{ ...styles.tableItself }}>
```

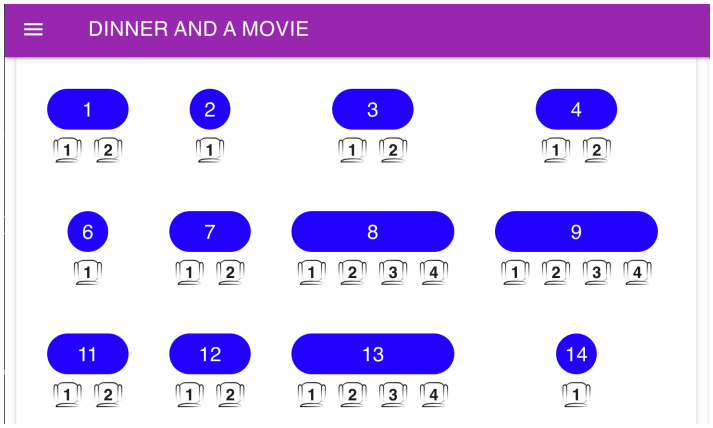
25. Change it to look like this:

```
<div style={{ ...styles.tableItself, ...getTableWidth(table.seats) }}>
```

26. Add this function:

```
function getTableWidth(seats) {
  return {width: seats.length * 40 + "px"}
}
```

27. Run and test. You should see the tables as wide as the seats need to fit:



28. Take a few minutes to parse what went on there. We made use of JavaScript's destructuring.