Topic	LAZY LOADING	
Class Description	**The student learns to display items in a list using the FlatList component. The student also builds a search bar in the search screen to display the list of transactions queried by the user.**	
Class	**C74**	
Class time	**40 mins**	
Goal	• Display transactions using the Flatlist. • Build a search bar to query transactions collection to search for a field value. • Display the query results in a Flatlist.	

Resources Required	• Teacher Resources
	  ○ Laptop with internet connectivity
	  ○ Earphones with mic
	  ○ Notebook and pen
	  ○ Android/iOS Smartphone with Expo App installed
	
	• Student Resources
	  ○ Laptop with internet connectivity
	  ○ Earphones with mic
	  ○ Notebook and pen
	  ○ Android/iOS Smartphone with Expo App installed	

Class structure	**Warm-Up** **Teacher-led Activity'** **Student-led Activity** **Wrap-Up**	**5 mins** **15 min** **15 min** **5 min**
WARM-UP SESSION - 5 mins		
Teacher starts slideshow  **from slides 1 to 13** Refer to speaker notes and follow the instructions on each slide.		
Activity details		**Solution/Guidelines**

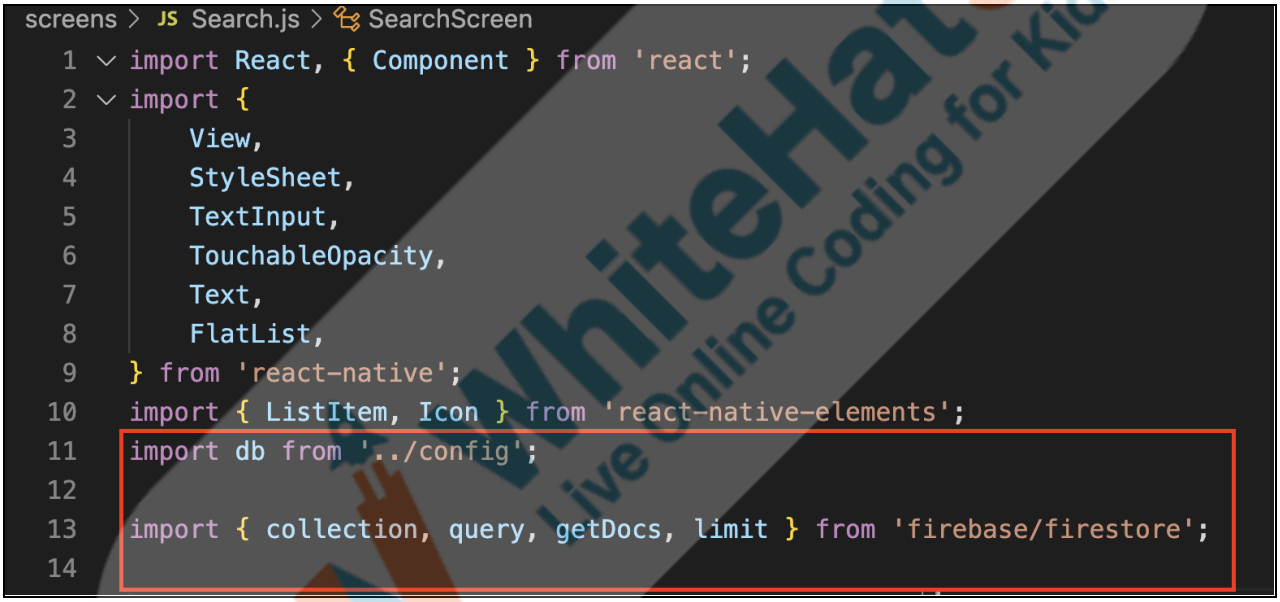
Hi, how have you been? Are you excited to learn something new? ***Run the presentation from slide 1 to slide 4.*** **The following are the warm-up session deliverables:** ● Reconnect with previous class topics. ● Warm-Up quiz session.	**ESR**: Varied Response. Click on the slide show tab and present the slides.
QnA Session	
Question	**Answer**
Which prop of the Barcode Scanner component calls a function to handle data received after scanning? A. onCodeScanned B. onBarScanned C. BarCodeScanned D. onBarCodeScanned	**D**
Which predefined function of the Permission component can help us request various permissions? A. .askPermission() B. .Async() C. .askAsync() D. .Permission()	**C**
Continue the warm-up session	
Activity details	**Solution/Guidelines**

Run the presentation from slide 5 to slide 13 to set the problem statement. **The following are the warm-up session deliverables:** • Talk about the usage of FlatList to show the list of all transactions.	
Teacher ends slideshow 	
TEACHER-LED ACTIVITY - 15 mins	
Teacher Initiates Screen Share	
CHALLENGE • **Display all transactions in a FlatList.**	
Teacher Action	**Student Action**
The teacher clones the boilerplate code from Teacher Activity 1 and installs all the dependencies for the project. ***Steps to clone the project:-*** ***git clone <projectURL>*** ***cd <projectFolder>*** ***npm install*** ***Note:*** *UI for the search bar and the* ***getTransaction()*** *function is given in the link mentioned above.*	
As usual, before proceeding, let's understand the given boilerplate code. *__Note:__ Get the student to focus on how firebase queries are done over the past classes.	*The student listens and understands the code.*

As seen in the visual, let's build our search screen first. Before we learn to use the **FlatList** component, let us display all the transaction documents from the transaction collections in our search screen. How do you think we can do that?	**ESR:** We will create a state array that will hold all the transactions. When the screen mounts, we will query all the transactions and store them in the **state** array. In the **render()** function for the component, we will map over the state array and display each item.
Awesome! Let's try to do this. *The teacher explains the code for the **getTransaction()** function.* To store all the transactions, we have an empty array called as **allTransactions** in the state.	

```
export default class SearchScreen extends Component {
  constructor(props) {
    super(props);
    this.state = {
      allTransactions: [],
    };
  }
```

First, we have imported the required database query function and database as **db** to the screen. Then we are writing a **getTransactions()** function in which we are making a query on the transactions' collection to get all the transactions and store them in the **allTransactions** array. Then we call the **getTransactions()** function inside the **componentDidMount()** function, as we want to get the data as soon as the component is loaded.	*The student listens and asks questions if any.*



```
getTransactions = async () => {
    let dbQuery = query(collection(db, 'transactions'));

    let querySnapShot = await getDocs(dbQuery);

    querySnapShot.forEach((doc) => {
        this.setState({
            allTransactions: [...this.state.allTransactions, doc.data()],
            lastVisibleTransaction: doc,
        });
    });
};
```

Now we have all the data, but we don't have any means to show this data in a list manner. Can you tell me which would be more efficient to show the data?	**ESR:** Varied
In the visuals, do you recall the term - **FlatList**?	**ESR**: Yes!
React Native has a component called as **FlatList** which will help us to showcase this data in a List manner.	
*The teacher shows documentation for **FlatList** to the student. Teacher Activity 2.*	*The student and teacher go through the FlatList documentation together.*
*The teacher also explains more about **FlatList** via documentation.*	

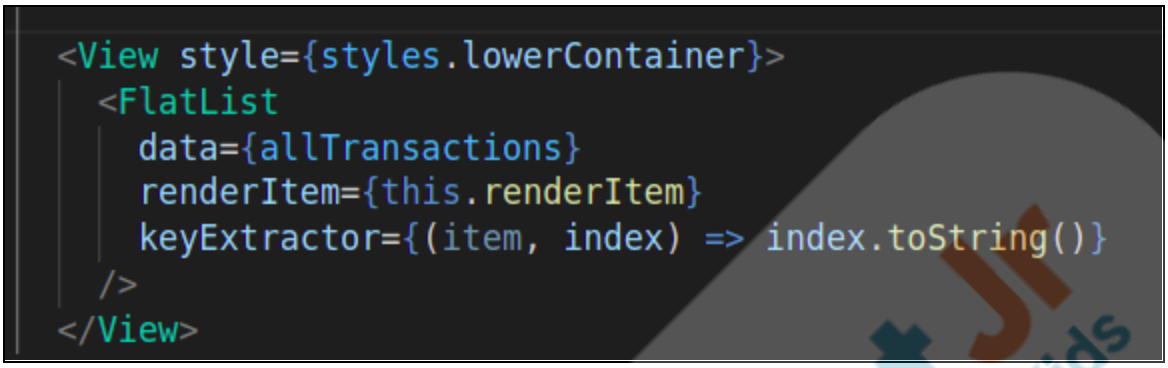
FlatList

A performant interface for rendering basic, flat lists, supporting the most handy features:

- Fully cross-platform.
- Optional horizontal mode.
- Configurable viewability callbacks.
- Header support.
- Footer support.
- Separator support.
- Pull to Refresh.
- Scroll loading.
- ScrollToIndex support.
- Multiple column support.

If you need section support, use `<SectionList>`.

Let's use FlatList to list down all our transactions. *The teacher can choose to show the visual slide 11-12 to connect this bit to the visual.* FlatList has 3 key props: • **data**: This contains all the data in the array which needs to be rendered. • **renderItem**: This takes each item from the data array and renders it as described using JSX. This should return a JSX component. • **keyExtractor**: It gives a unique key prop to each item in the list. The unique key prop should be a string.	*The student asks questions around the FlatList if any.*

*The teacher explains the usage of **FlatList** for rendering the transactions using the 3 props described above.*	*The student listens and asks questions for clarification.*
	
Before executing and testing our code, let's understand the **renderItem()** function we used. In this function, we are first converting the timestamp date to a normal human-readable string when the book is being issued or returned by the student. To convert this date we use the **split()** function which splits the string into an array and from that array we are slicing the index using the **splice()** function, and then we join all indexes using the **join** method. ***Note**: For more information, refer to the following documents:* *Teacher Activity 3 for **split()**,* *Teacher Activity 4 for **splice()**,* *and Teacher Activity 5 for **join()**.* Coming back to our application, first, we are checking the type of transaction. If the transaction type is an **issue** we are changing it to issues else changing it to **return**. In the **return()** function using the component **ListItem** from react-native-elements, we are creating a UI for a transaction list. To get the first letter of the string we use the **CharAt(0)** and to convert it into the upper case we use the **toUpperCase() method.**	

The **charAt()** method returns the character at the specified index in a string. The index of the first character is **0**, the second character is **1**, and so on.	
Complete code for the **renderItem()** function.	

```
renderItem = ({ item, i }) => {
  var date = item.date
    .toDate()
    .toString()
    .split(" ")
    .splice(0, 4)
    .join(" ");

  var transactionType =
    item.transaction_type === "issue" ? "issued" : "returned";
  return (
    <View style={{ borderWidth: 1 }}>
      <ListItem key={i} bottomDivider>
        <Icon type={"antdesign"} name={"book"} size={40} />
        <ListItem.Content>
          <ListItem.Title style={styles.title}>
            {`${item.book_name} ( ${item.book_id} )`}
          </ListItem.Title>
          <ListItem.Subtitle style={styles.subtitle}>
            {`This book ${transactionType} by ${item.student_name}`}
          </ListItem.Subtitle>
          <View style={styles.lowerLeftContaiiner}>
            <View style={styles.transactionContainer}>
              <Text
                style={[
                  styles.transactionText,
                  {
                    color:
                      item.transaction_type === "issue"
                        ? "#78D304"
                        : "#0364F4"
                  }
                ]}
              >
                {item.transaction_type.charAt(0).toUpperCase() +
                  item.transaction_type.slice(1)}
              </Text>
              <Icon
                type={"ionicon"}
                name={
                  item.transaction_type === "issue"
                    ? "checkmark-circle-outline"
                    : "arrow-redo-circle-outline"
                }
                color={
                  item.transaction_type === "issue" ? "#78D304" : "#0364F4"
                }
              />
            </View>
            <Text style={styles.date}>{date}</Text>
          </View>
        </ListItem.Content>
      </ListItem>
    </View>
  );
};
```

The teacher runs the code and tests. You can see that this **FlatList** is getting rendered.	*The student and teacher test the code.*
	
Awesome, we are getting the expected output. We also have the UI for the search bar. But as of now, it does nothing. So let's write a function that will help us to search for the desired transaction in the list.	
Generally, how does the search bar work in other apps? *Note: When the user enters the text in the search bar, we*	**ESR:** In other apps, we enter the text that we want to search in the search bar and hit the

*are storing it in the **searchText** variable in the state.* Great! Let's create a function called **handleSearch()** which takes the text parameter. Using this text, it will search through all the transactions. In this function first, we'll get the text entered by the user, split it to get the first character and convert it to upper case using the **toUpperCase()** function. If the entered text starts with **"B"** we'll make a query on **allTransactions** to get the book ID that matches the entered text. If the entered text starts with **"S"** we'll make a query in the **allTransactions** to get all the Student IDs that match the entered text. *The teacher writes the code for the **handleSearch()** function.*	search button and below we get the desired results.

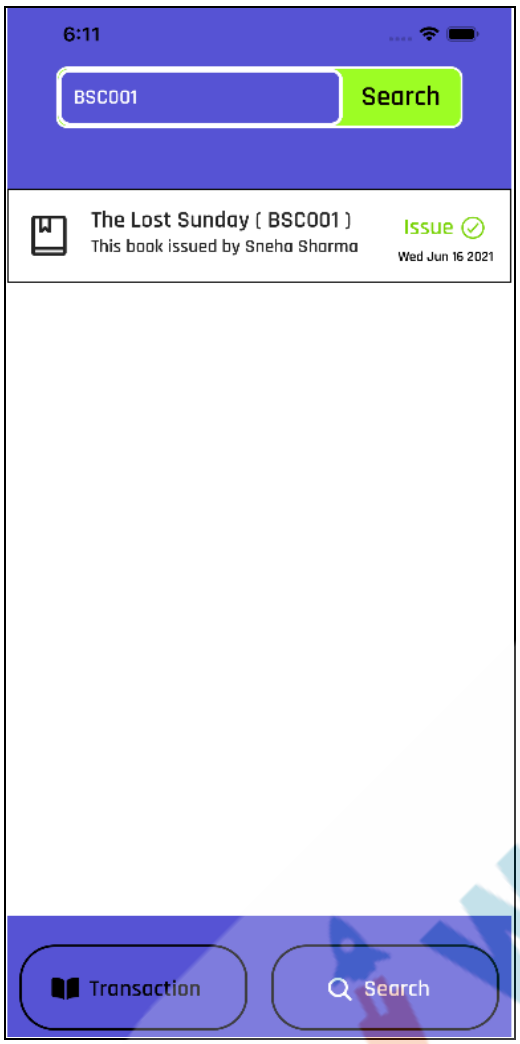
```
handleSearch = async (text) => {
    var enteredText = text.toUpperCase().split('');
    text = text.toUpperCase();
    this.setState({
        allTransactions: [],
    });
    if (!text) {
        this.getTransactions();
    }

    if (enteredText[0] === 'B') {
        let dbQuery = query(
            collection(db, 'transactions'),
            where('book_id', '==', text)
        );

        let querySnapShot = await getDocs(dbQuery);

        querySnapShot.forEach((doc) => {
            this.setState({
                allTransactions: [...this.state.allTransactions, doc.data()],
            });
        });
    } else if (enteredText[0] === 'S') {
        let dbQuery = query(
            collection(db, 'transactions'),
            where('student_id', '==', text)
        );

        let querySnapShot = await getDocs(dbQuery);

        querySnapShot.forEach((doc) => {
            this.setState({
                allTransactions: [...this.state.allTransactions, doc.data()],
            });
        });
    }
};
```

We will call this function when the search button is pressed.

```
return (
  <View style={styles.container}>
    <View style={styles.upperContainer}>
      <View style={styles.textinputContainer}>
        <TextInput
          style={styles.textinput}
          onChangeText={text => this.setState({ searchText: text })}
          placeholder={"Type here"}
          placeholderTextColor={"#FFFFFF"}
        />
        <TouchableOpacity
          style={styles.scanbutton}
          onPress={() => this.handleSearch(searchText)}
        >
          <Text style={styles.scanbuttonText}>Search</Text>
        </TouchableOpacity>
      </View>
    </View>
```

Now let's run and test the output.

The teacher runs the code to check the output.



Even in **FlatList**, we are waiting for all the transaction documents to load in **ComponentDidMount()** before rendering these items. We don't have to do that. Can you tell me how can we fix this? Yes!! Loading a few transactions will be quicker than getting all the transactions from the collection.	**ESR:** We can load a few transactions which are visible on our screen.

Awesome. Can you take this as a challenge and do it now?	*The student takes up the challenge.*
Teacher Stops Screen Share	
Now it's your turn. Please share your screen with me.	
STUDENT-LED ACTIVITY - 15 mins	
• **Ask Student to press ESC key to come back to panel** • **Guide Student to start Screen Share** • **Teacher gets into Fullscreen**	
ACTIVITY • **Build a search bar to query for book ID or student ID in transactions collection.** • **Display the results of the query in a FlatList.**	
Teacher starts slideshow  **: Slide 14 and slide 15.**	
Teacher Action	**Student Action**
The teacher guides the student to clone the boilerplate code from Student Activity 1.	*The student clones the code from Student Activity 1 and sets up the project.*
To solve the problem that we discussed earlier, we can get only the first 10 transactions in **componentDidMount**. We'll use the **limit()** function to limit the transactions that we get to **10**. This will be quicker than getting all the transactions from the collection. Let's store the last transaction doc, which we get inside	*The student writes the code to create a state called as **lastVisibleTransaction**. And also uses the **limit()** function to get only 10 transactions.*

another state called **lastVisibleTransaction**.

The teacher guides the student to write the code.

```
export default class SearchScreen extends Component {
  constructor(props) {
    super(props);
    this.state = {
      allTransactions: [],
      lastVisibleTransaction: null,
      searchText: ""
    };
  }
```

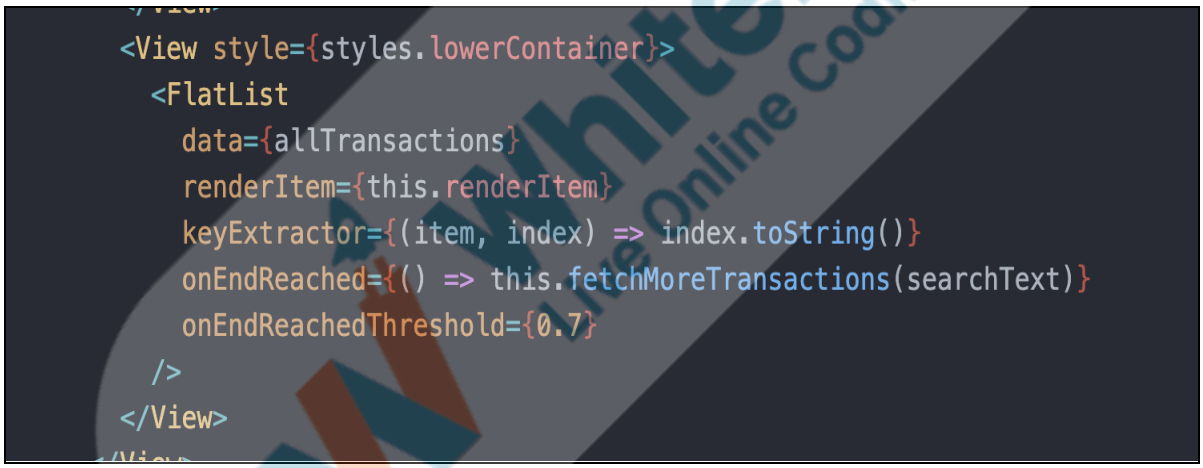
```
  getTransactions = async () => {
      let dbQuery = query(collection(db, 'transactions'));

      let querySnapShot = await getDocs(dbQuery);

      querySnapShot.forEach((doc) => {
          this.setState({
              allTransactions: [...this.state.allTransactions, doc.data()],
              lastVisibleTransaction: doc,
          });
      });
  };
```

FlatList has two more important props. **onEndReached** and **onEndThreshold:**

Using this we'll perform lazy loading as you saw in the visual, **Lazy Loading** is the technique of rendering only-needed or critical user-interface items first, then quietly unrolling/loading the non-critical items later. Hence, we can say Lazy Loading is a technique that can be used

The student listens and asks questions for clarification.

in FlatList. • **onEndReached** can call a function to get more transaction documents after the last transaction document was fetched. • **onEndThreshold** defines when we want to call the function inside the **onEndReached** prop. If **onEndThreshold** is **1**, the function will be called when the user has completely scrolled through the list. If **onEndThreshold** is **0.5**, the function will be called when the user is mid-way during scrolling the items.	
Let's set the **onEndReachedThreshold** as **0.7** and write a function to fetch more transaction documents after the last transaction document we fetched.	


`<View style={styles.lowerContainer}>`
`<FlatList`
`data={allTransactions}`
`renderItem={this.renderItem}`
`keyExtractor={(item, index) => index.toString()}`
`onEndReached={() => this.fetchMoreTransactions(searchText)}`
`onEndReachedThreshold={0.7}`
`/>`
`</View>`	

We'll be doing a similar thing as we did earlier in the **handleSearch()** function. Here the only change would be to add the limit of **10** and getting the transactions after the last visible transaction, which we have stored earlier in our state variable called **lastVisibleTransaction**. We'll use the **startAfter()** function to start getting the transactions from the last visible transaction on the screen.	*The student writes the code*

The teacher guides the students to write the code for ***fetchMoreTransactions()*** *function.*	*for* ***fetchMoreTransactions()*** *function.*

```
fetchMoreTransactions = async (text) => {
    var enteredText = text.toUpperCase().split('');
    text = text.toUpperCase();

    if (enteredText[0] === 'B') {
        let dbQuery = query(
            collection(db, 'transactions'),
            where('book_id', '==', text),
            startAfter(this.state.lastVisibleTransaction),
            limit(10)
        );

        let querySnapShot = await getDocs(dbQuery);

        querySnapShot.forEach((doc) => {
            this.setState({
                allTransactions: [...this.state.allTransactions, doc.data()],
                lastVisibleTransaction: doc,
            });
        });
    } else if (enteredText[0] === 'S') {
        let dbQuery = query(
            collection(db, 'transactions'),
            where('student_id', '==', text),
            startAfter(this.state.lastVisibleTransaction),
            limit(10)
        );

        let querySnapShot = await getDocs(dbQuery);

        querySnapShot.forEach((doc) => {
            this.setState({
                allTransactions: [...this.state.allTransactions, doc.data()],
                lastVisibleTransaction: doc,
            });
        });
    }
};
```

The teacher guides the student to run and tests the code.	*The student runs the code and observes the output.*

You can see how lazy loading works. Every time we scroll down, we see new items getting added to the list.

Teacher Guides Student to Stop Screen Share

WRAP-UP SESSION - 5 Mins

Teacher starts slideshow  from slide 16 to slide 25	
Activity details	**Solution/Guidelines**
Run the presentation from slide 16 to slide 25. **Following are the wrap-up session deliverables:** • **Explain the facts and trivias** • **Next class challenge** • **Project for the day** • **Additional Activity**	Guide the student to develop the project and share it with us.
Quiz time - Click on in-class quiz	
Question	**Answer**
Which component of React-Native did we use to implement lazy loading? A. ScrollView B. Flatview C. FlatList D. renderItem	**C**
What does limit() do? A. Retrieves the data from the db, sorted in ascending order. B. Retrieves the data from the db, sorted in descending order. C. Limits the “where” query to be only used once. D. Retrieves a particular number of rows from the db.	**D**
Which of the following is true for renderItem? A. It is a prop in <Flatlist/>. B. This should return a JSX component. C. This takes each item from the data array and renders it as described using JSX. D. All of the above.	**D**

End the quiz panel	
Let's wrap up today's class. What did we learn today?	**ESR:** We learned how to build lazy loading in our app using FlatList.
Awesome! We have the app almost ready now. In the next class, we are going to work on authentication for our app so that only teachers who are authorized can use this app. Hope to see you in the next class then. Meanwhile, you can work on styling your app to make it look better.	
Does the word 'Challenge' ring any bells? Well, I am talking about the upcoming challenge. In the next class, we will continue to build a complete library management system - e-library - which will allow teachers to issue or students to return books from a library by scanning QR codes.	
You get a “hats off” for the amazing performance today!	*Make sure you have given at least 2 Hats Off during the class for:* 

Project Overview

E-RIDE STAGE 7

Goal of the Project:

In class 74, you learned to display items in a list using the FlatList component. You built a search bar in the search screen to display the list of transactions queried by the user. You will practice the same concept in today's project.

* This is a continuation of Project-68, 69, 70, 71, 72 & 73 to make sure you have completed and submitted that before attempting this one.

Story:

With the addition of all the eligibility checks, the app is running super smoothly. You have released a beta version of your app for tests and feedback to his known riders. They are finding your app easy to use and very useful.

However, the users of the app have requested to view their history of riding rented bikes. You too already have that idea in your mind. It is time to implement that functionality in your App.

I am very excited to see your project solution and I know you both will do really well.

Bye Bye!

Teacher ends slideshow 	
Teacher Clicks ✖ End Class	
ADDITIONAL ACTIVITIES	
Additional Activities *Encourage the student to write reflection notes in their reflection journal using Markdown.* Use these as guiding questions: ● What happened today? ○ Describe what happened ○ Code I wrote ● How did I feel after the class? ● What have I learned about programming and developing games? ● What aspects of the class helped me? ● What did I find difficult?	*The student uses the Markdown editor to write her/his reflection as a reflection journal.*

Links:

Activity	Activity Name	Links
Teacher Activity 1	Boilerplate code	https://github.com/React-Native-Frontier/PRO-C74-E-Library-TA-boilerplate
Teacher Activity 2	Flatlist Documentation	https://facebook.github.io/react-native/docs/flatlist
Teacher Activity 3	Split function	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split
Teacher Activity 4	Splice function	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice
Teacher Activity 5	Join function	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join
Teacher Activity 6	Final Solution	https://github.com/React-Native-Frontier/PRO-C74-E-Library
Student Activity 1	Boilerplate code	https://github.com/React-Native-Frontier/PRO-C74-E-Library-SA-boilerplate
Student Activity 2	Flatlist Documentation	https://facebook.github.io/react-native/docs/flatlist
Visual Aid Link	VA Link	https://curriculum.whitehatjr.com/Visual+Project+Asset/PRO_VD/BJFC-PRO-V3-C74-withcues.html
In-Class Quiz	In-Class quiz doc	https://s3-whjr-curriculum-uploads.whjr.online/291d8198-4151-49e9-bb4f

		-ad873676b78b.pdf