Lecture 7: Navigator

7.1 Navigation Fundamentals with React Navigation (Stack Navigator)

Purpose. This subsection establishes the conceptual and practical foundations of navigation in React Native using React Navigation, with an emphasis on the Stack Navigator model. It formalizes the navigation state, route representation, and action semantics necessary for rigorous reasoning and reproducible demonstrations.

Explanation. Navigation in React Native is governed by a navigation state that is immutable and deterministic. A stack navigator represents this state as a last-in, first-out (LIFO) structure. Each route contains a `name`, optional `params`, and a position within the stack. Actions such as `navigate`, `push`, and `goBack` transform the state and yield predictable user interface transitions. Rendering is always a pure function of the navigation state.

The recommended implementation is the `native-stack` navigator (`@react-navigation/native-stack`), which provides better gesture fidelity and performance compared to the JS stack implementation. A minimal example is shown below:

```
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

const Stack = createNativeStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Detail" component={DetailScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Parameters may be passed between routes using `navigate`:

```
navigation.navigate('Detail', { unitId: item.id });

// In DetailScreen
const { unitId } = route.params;
```

Best practice is to pass stable identifiers and resolve large payloads inside the destination screen. In TypeScript, a route map provides compile-time validation of parameters, ensuring safe navigation during demonstrations.

Dynamic headers can be specified as functions of the route:

```
<Stack.Screen
```

```
  name="Detail"
  component={DetailScreen}
  options={({ route }) => ({ title: route.params.unitId })}
/>
```

Navigation actions such as `replace` and `reset` allow re-rooting flows and managing state transitions. To demonstrate lifecycles, instructors may use hooks such as `useFocusEffect` to illustrate resource management when screens are mounted or blurred.

7.2 Data Modeling for Educational Content

Purpose. This subsection articulates principled methods for representing educational content in React Native applications. It advances a schema-oriented perspective that links conceptual entities (lessons, objectives, strategies, materials) to practical concerns such as immutability, serialization, and internationalization.

Canonical Dataset. For instructional purposes, we employ a dataset of lessons structured as plain JavaScript objects:

```
const lessons = [
  {
    title: 'Educational Psychology for Teachers',
    subtitle: 'Basic theories and classroom applications',
    objectives: [
      'Explain key learning theories (Behaviorism, Constructivism)',
      'Relate theories to the design of learning activities',
    ],
    strategies: ['Case-based discussion', 'Reflective journaling activities'],
    materials: [
      'Academic articles in Thai/English',
      'Theory summary infographics',
    ],
    imageUrl: 'https://static.vecteezy.com/system/resources/thumbnails/006/487/917/small_2x/man-avatar-icon-free-ve
  },
  {
    title: 'Educational Measurement and Evaluation',
    subtitle: 'Design valid and reliable assessment tools',
    objectives: [
      'Identify types of assessment (formative/summative)',
      'Design rubrics aligned with learning objectives',
    ],
    strategies: [
      'Workshop on test/rubric design',
      'Observe mock classes and provide feedback',
    ],
    materials: ['Sample rubrics/forms', 'Score recording software'],
    imageUrl: 'https://static.vecteezy.com/system/resources/thumbnails/006/487/917/small_2x/man-avatar-icon-free-ve
  },
  {
    title: 'Positive Classroom Management',
    subtitle: 'Create a safe and engaging learning environment',
    objectives: [
      'Analyze student behavior and environmental factors',
      'Apply techniques for handling classroom situations',
    ],
    strategies: ['Role-play activities', 'Co-create classroom agreements'],
    materials: [
      'Classroom rules posters',
      'Individual behavior tracking forms',
    ],
    imageUrl: 'https://static.vecteezy.com/system/resources/thumbnails/006/487/917/small_2x/man-avatar-icon-free-ve
  },
];
```

Why This Structure. This representation is intentionally *denormalized* to suit small-scale educational applications and classroom demonstrations:

- **Coherence.** Each lesson is a self-contained unit bundling identity (title, subtitle), pedagogy (objectives, strategies), resources (materials), and a media reference (`imageUrl`). This mirrors how educators think about a "lesson plan."
- **Readability.** Nested arrays for objectives, strategies, and materials allow direct mapping to bullet lists in a detail screen, reducing cognitive translation between schema and UI.
- **Serialization.** Plain JSON types (strings, arrays) remain safe for React Navigation route parameters and local storage, avoiding non-serializable pitfalls.
- **Pedagogical Transparency.** Students can immediately observe how each conceptual field (e.g., objectives) becomes a visible UI section, reinforcing the link between data modeling and interface design.
- **Scalability Path.** For larger or mutable datasets, this structure can evolve toward normalization (e.g., factoring out reusable media assets), but for teaching labs, embedding fields minimizes indirection.

Demonstration in Use. In practice, this dataset allows us to drive both a `FlatList` (showing titles and subtitles) and detail screens (expanding into objectives, strategies, and materials) without additional transformation. Thus, the structure balances *simplicity for learning* with *fidelity to production principles*.

7.3 List Rendering with `FlatList`

Purpose. This subsection introduces the virtualization model of `FlatList`, which renders only a sliding window of data to maintain performance and determinism.

Explanation. Correct list rendering requires that each item has a unique and stable key. While small demonstrations often use the list index, production-ready applications should rely on stable identifiers such as `id`. The `FlatList` API expects `data`, `keyExtractor`, and `renderItem` as core props.

Minimal Example. A simple list rendering titles only might look as follows:

```
<FlatList
  data={DATA}
  keyExtractor={(item) => item.id}
  renderItem={({ item }) => <Text>{item.title}</Text>}
/>
```

Demonstration Example. The following example shows a more complete instructional pattern where each row combines an image, text, and a navigation action. It illustrates correct styling, layout, and use of icons for clarity.

```
<FlatList
  data={lessons}
  keyExtractor={(item, idx) => item.title + ':' + idx}
  renderItem={({ item }) => (
    <TouchableOpacity
      onPress={() => navigation.navigate('Detail', { unit: item })}
      style={{
        flexDirection: 'row',
        alignItems: 'center',
        backgroundColor: '#faf5ff', // light purple background
        borderRadius: 12,
        padding: 16,
        marginVertical: 8,
        marginHorizontal: 16,
      }}
    >
      {/* Left icon */}
      <Image
        source={{ uri: item.imageUrl }}
        style={{ width: 40, height: 40, marginRight: 12 }}
      />

      {/* Title + Subtitle */}
      <View style={{ flex: 1 }}>
        <Text style={{ fontSize: 16, fontWeight: '600', color: '#111' }}>
          {item.title}
        </Text>
        <Text style={{ fontSize: 14, color: '#6b7280', marginTop: 2 }}>
          {item.subtitle}
        </Text>
      </View>

      {/* Right arrow */}
      <Ionicons name="chevron-forward" size={20} color="#4b5563" />
    </TouchableOpacity>
  )}
/>
```

Explanation of Components.

- **KeyExtractor.** Uses the item's `title` concatenated with its index. While suitable for demonstrations, in practice it is better to use a stable `id` to prevent churn during reordering.
- **TouchableOpacity.** Provides an interactive row. When pressed, it triggers navigation to a `Detail` screen, passing the selected lesson as a parameter.
- **Image.** Renders a left-aligned thumbnail fetched from a remote URI, constrained to fixed dimensions to avoid layout instability.
- **Title and Subtitle.** Placed inside a `View` with flexible width, maintaining readable typography and spacing.
- **Ionicons Arrow.** Adds a visual affordance that indicates navigability, aligned to the right for clarity.

Pedagogical Value. This richer example demonstrates:

1. How layout primitives (`flexDirection`, `alignItems`) compose list rows.
2. How data models (`lesson.title`, `lesson.subtitle`, `lesson.imageUrl`) bind directly to UI.
3. How navigation and parameter passing connect list items to detail screens.
4. How visual rhythm (consistent padding, margins, rounded corners) improves readability in classroom contexts.

7.4 Assets and Iconography

Purpose. This subsection systematizes asset handling in React Native, focusing on correctness, resolution, and theming.

Explanation. Local assets are imported using `require`, enabling Metro bundler to manage scaling automatically:

```
<Image source={require('./assets/logo.png')}
       style={{ width: 44, height: 44 }} />
```

Remote assets must be declared with explicit dimensions:

```
<Image source={{ uri: 'https://example.com/hero.png' }}
       style={{ width: 92, height: 92 }} />
```

Density variants (`logo.png`, `logo@2x.png`, `logo@3x.png`) ensure fidelity across devices. For icons, use a coherent library (e.g., Ionicons) and bind them to semantic roles:

```
<Ionicons name="chevron-forward"
          size={20}
          accessibilityLabel="Go to details" />
```

7.5 Styling and Layout in React Native

Purpose. This subsection explains layout construction in React Native, linking Flexbox/Yoga semantics with spacing, typography, and accessibility.

Explanation. React Native uses Flexbox rules to compute layout. Developers compose layouts with small, predictable containers:

```
<View style={{ flexDirection: 'row', justifyContent: 'space-between' }}>
  <Text>Left</Text>
  <Text>Right</Text>
</View>
```

Styles should be centralized using `StyleSheet.create`:

```
const styles = StyleSheet.create({
  card: { padding: 16, borderRadius: 12, backgroundColor: 'white' }
});
```

Responsive design relies on fluid dimensions and `SafeAreaView`:

```
<SafeAreaView style={{ flex: 1, padding: 16 }}>
  <Text>Responsive Layout</Text>
</SafeAreaView>
```

Color and elevation must consider platform differences. Accessibility requires adequate contrast, hit slop for touch targets, and screen reader validation.

7.6 Showcase: Building the Lesson App

Purpose. This showcase illustrates a full incremental workflow for constructing the "EduLesson" app using React Native and Expo. Each step is grounded in theoretical principles: immutability, declarative UI, navigation state formalism, virtualization, and pedagogical clarity.

Step 0 – Project Initialization.

```
npx create-expo-app edulesson-rn --template
  # select "blank"
cd edulesson-rn
npx expo install react-dom react-native-web @expo/metro-runtime
npm i @react-navigation/native
npm install @react-navigation/stack
npx expo install react-native-screens react-native-safe-area-context

npm run android
```

Theory. Initialization establishes the baseline environment. Expo provides a managed workflow that abstracts platform-specific setup, while React Navigation introduces the formal navigation state model. Installing `react-native-screens` and `safe-area-context` enforces predictable UI layout across devices, consistent with the principle of reproducible demonstrations.

Step 1 – Minimal App (Hello World).

```
import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator screenOptions={{ headerShadowVisible: false }}>
        <Stack.Screen name="Home" component={() => <div>Home Screen</div>} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Theory. This step illustrates declarative UI: the navigation state is mapped purely into UI components (*state* → *view*). Even the trivial "Home Screen" showcases determinism: given identical inputs, the same output UI is always produced. The `NavigationContainer` introduces the concept of a global navigation state machine.

Step 2 – Add a Component.

```
import { Text, View } from 'react-native';

function LessonListScreen() {
  return (
    <View style={{ flex: 1, justifyContent:'center', alignItems: 'center' }}>
      <Text style={{ fontSize: 22, fontWeight: '700' }}>Hello, EduLesson</Text>
    </View>
  );
}
```

Theory. Adding a component demonstrates composition: complex UIs are built from smaller reusable parts. This follows the principle of *local reasoning*: a screen is a pure function of its props and internal state, improving pedagogical clarity and testability.

Step 3 – Add Items and Display in a List.

```
const lessons = [
  { title: 'Educational Psychology for Teachers',
    subtitle: 'Basic theories and classroom applications' },
  { title: 'Educational Measurement and Evaluation',
    subtitle: 'Design valid and reliable assessment tools' },
  { title: 'Positive Classroom Management',
    subtitle: 'Create a safe and engaging learning environment' },
];

<FlatList
  data={lessons}
  keyExtractor={(item, idx) => item.title + idx}
  renderItem={({ item }) => (
    <TouchableOpacity style={{ backgroundColor:'#fff', padding:12 }}>
      <Text>{item.title}</Text>
      <Text>{item.subtitle}</Text>
    </TouchableOpacity>
  )}
/>
```

Theory. This introduces virtualization: `FlatList` renders only visible items, recycling off-screen cells. Keys ensure referential integrity—without stable identifiers, cell reuse breaks determinism. The list illustrates the cognitive principle of *overview first*: presenting many items compactly before diving into detail.

Step 3b – Connect to a Detail Screen.

```
function LessonDetailScreen({ route }) {
  const { unit } = route.params;
  return (
    <View style={{ flex:1, padding:16 }}>
      <Text style={{ fontSize:20, fontWeight:'700' }}>{unit.title}</Text>
      <Text style={{ fontSize:14 }}>{unit.subtitle}</Text>
    </View>
  );
}

onPress={() => navigation.navigate('Detail', { unit: item })}
```

Theory. Navigation formalizes the *list-to-detail* pattern, a canonical UI flow. Passing `params` demonstrates serialization requirements: only JSON-compatible data ensures predictable rehydration. This step exemplifies *progressive disclosure*: details are revealed only upon explicit navigation.

Step 4 – Enrich with Objectives, Strategies, and Materials.

```
const lessons = [
  {
    title: 'Educational Psychology for Teachers',
    subtitle: 'Basic theories and classroom applications',
    objectives: ['Explain key learning theories', 'Relate theories to practice'],
    strategies: ['Case-based discussion'],
    materials: ['Theory summary infographics'],
  }
];

function BulletList({ items }) {
  return (
    <View>
      {items.map((t, i) => (
        <View key={i} style={{ flexDirection: 'row' }}>
          <Text>{'•'}</Text>
          <Text>{t}</Text>
        </View>
      ))}
    </View>
  );
}
```

Theory. Enriching the schema illustrates *data modeling*: educational content is decomposed into objectives, strategies, and materials. Each list enforces parallelism, aligning with cognitive load theory—consistent structures reduce extraneous processing. `BulletList` embodies reusability and semantic consistency.

Step 5 – Add Images.

```
<Image source={{ uri: unit.imageUrl }}
       style={{ width: 50, height: 50, borderRadius: 25 }} />
```

Theory. Adding images highlights asset resolution and density independence. By constraining dimensions, we achieve deterministic layout. The *dual coding theory* suggests that combining verbal and visual elements improves retention, aligning with pedagogical best practices.

Step 6 – Refine List Item Design.

```
<FlatList
  data={lessons}
  renderItem={({ item }) => (
    <TouchableOpacity style={{ flexDirection: 'row' }}>
      <Image source={{ uri: item.imageUrl }} style={{ width:40, height:40 }} />
      <View style={{ flex:1 }}>
        <Text>{item.title}</Text>
        <Text>{item.subtitle}</Text>
      </View>
      <Ionicons name="chevron-forward" size={20} />
    </TouchableOpacity>
  )}
/>
```

Theory. Refining design illustrates *affordance*: the rightward chevron signals navigability. Layout choices (`flexDirection: 'row'`) embody Gestalt principles of alignment and proximity, supporting scanability. Rounded corners and padding reinforce visual rhythm, improving user trust and readability.

Pedagogical Outcome. This incremental construction embodies both *software engineering principles* (immutability, determinism, modularity) and *instructional design principles* (progressive disclosure, cognitive load reduction, dual coding). By the final step, students can connect abstract theories of data modeling, navigation, and virtualization to a tangible educational app.