Розробка мобільних додатків

Лекція 6 - Взаємодія користувача з додатком

Взаємодія користувача з додатком у **React Native**

Взаємодія користувача (User Interaction) – це всі дії, які користувач виконує у додатку, і те, як додаток на них реагує.

Гарна взаємодія забезпечує:

- **Швидкий та плавний відгук** (без затримок у натисканнях, свайпах тощо).
- **Інтуїтивність** (елементи працюють так, як очікує користувач).
- **Доступність** (адаптація для всіх користувачів, включаючи людей з особливими потребами).

Основні способи взаємодії

✔ **Натискання** → Користувач натискає кнопку, посилання або будь-який інший інтерактивний елемент (`onPress`, `onLongPress`).
✔ **Жести** → Свайпи, масштабування, перетягування.
✔ **Введення тексту** → Заповнення форм, пошук, введення паролів (`TextInput`).
✔ **Прокрутка** → Перегляд списків та контенту (`ScrollView`, `FlatList`).
✔ **Системні події** → Зміна орієнтації екрана, натискання апаратних кнопок, відкриття клавіатури.

Чому важливо забезпечити коректну взаємодію?

- **Гарний користувацький досвід (UX)** → Додаток працює **плавно та передбачувано**.
- **Безпека** → Запобігає **помилковим діям** (наприклад, випадковому подвійному кліку на "Оплатити").
- **Інклюзивність** → Доступність для людей із вадами зору або обмеженими можливостями.

Як працюють події у **JavaScript?**

У стандартному JavaScript події обробляються через **Event Listeners**:

```
document.getElementById('btn').addEventListener('click', function (event) {
  console.log('Кнопку натиснуто!', event);
});
```

`event` – це об'єкт, який містить інформацію про подію (тип, координати кліку, елемент-ціль тощо).

Події у **React (Web)**

У React події обробляються через **синтетичні події (Synthetic Events)**, які нормалізують їхню роботу в різних браузерах.

```
const App = () => {
  const handleClick = (event) => {
    console.log('Синтетична подія:', event);
  };
  return <button onClick={handleClick}>Натисни мене</button>;
};
export default App;
```

Тут `onClick` – це атрибут у JSX, а `event` – синтетична подія (`SyntheticEvent`).

Що означає "синтетична" в контексті подій у React?

"Синтетична" означає **створена штучно**. У контексті React синтетична подія (`SyntheticEvent`) — це **обгортка навколо нативної події браузера чи мобільної платформи**.

Події у **React Native**

У React Native також використовуються **синтетичні події**, але вони базуються на мобільних подіях (натискання, свайпи тощо).

```
const App = () => {
    Show usages  new *
    const handlePress = (event) : void  => {
        console.log('SyntheticEvent:', event); // Виводимо всю подію в консоль

        console.log('Timestamp:', event.timeStamp); // Час події
        console.log('Target:', event.target); // Ціль події
        console.log('Native Event:', event.nativeEvent); // Оригінальна подія платформи
    };

    return (
        <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Button title="Натисни мене" onPress={handlePress} />
        </View>
    );
};
```

```
(NOBRIDGE) LOG Timestamp: 1740664819618
(NOBRIDGE) LOG Native Event: {"changedTouches": [{"identifier": 0, "locationX": 10.17
6846504211426, "locationY": 6.880681991577148, "pageX": 151.99502563476562, "pageY": 37
5.9715881347656, "target": 4, "targetSurface": 31, "timestamp": 2432611}], "identifier"
: 0, "locationX": 10.176846504211426, "locationY": 6.880681991577148, "pageX": 151.9950
2563476562, "pageY": 375.9715881347656, "target": 4, "targetSurface": 31, "timestamp":
2432611, "touches": []}
```

Event Handler

Обробник подій – це **функція, яка виконується у відповідь на певну подію** (наприклад, натискання кнопки, введення тексту, прокрутку списку тощо).

Він прив’язується до елемента через спеціальний атрибут (наприклад, `onPress`, `onChangeText`, `onScroll`) і запускається автоматично, коли подія відбувається.

Синтаксис обробника подій у **React Native**

1. Передача обробника як функції (РЕКОМЕНДОВАНО)

```
<Button title="Натисни" onPress={handlePress} />

const handlePress = () => {

  console.log('Кнопку натиснуто!');

};
```

Функцію передаємо напряму, що забезпечує кращу продуктивність.

2. Анонімна (inline) функція

```
<Button title="Натисни" onPress={() => console.log('Натиснуто!')} />
```

Мінуси: створює нову функцію при кожному рендері.

3. Передача параметрів у обробник

```
<Button title="Привіт" onPress={() => handlePress('Hello!')} />

const handlePress = (message) => {

  console.log(message);

};
```

Використовуємо **стрілкову функцію**, щоб передати параметри.

4. Обробка текстового вводу (`onChangeText`)

```
<TextInput onChangeText={(text) => console.log('Введено:', text)} />
```

В React Native немає `event.target.value`, значення передається напряму.

Події натискання **(Press Events)** у **React Native**

Press Events — це події, що відстежують взаємодію користувача з елементами інтерфейсу за допомогою дотику або натискання. Вони застосовуються для обробки взаємодії з кнопками, текстовими елементами, зображеннями тощо.

1️⃣ Обробка стандартного натискання (`onPress`)

```
import React from 'react';
import { Button, Alert } from 'react-native';
const App = () => {
  return (
    <Button title="Натисніть" onPress={() => Alert.alert('Кнопку натиснуто!')} />
  );
};
export default App;
```

Подія `onPress` спрацьовує після короткого натискання на кнопку.

Обробка тривалого натискання (`onLongPress`)

```
<TouchableOpacity
  onPress={() => console.log('Коротке натискання')}
  onLongPress={() => console.log('Довге натискання')}
>
  <Text>Тест</Text>
</TouchableOpacity>
```

`onLongPress` викликається, якщо користувач утримує елемент довше стандартного часу, що дозволяє реалізувати контекстні дії або додаткові функції.

Якщо утримати елемент довго, **`onLongPress` спрацює першим**, а `onPress` **не викличеться**.

Кастомізація довгого натискання **(onLongPress)** у **React Native**

Встановлення власної тривалості (`delayLongPress`)

```
<TouchableOpacity
  onLongPress={() => console.log('Довге натискання!')}
  delayLongPress={800} // Час у мілісекундах (800ms = 0.8 секунди)
>
  <Text>Текст</Text>
</TouchableOpacity>
```

За замовчуванням `delayLongPress = 500` мс.

Використання `onPressIn` та `onPressOut` для відстеження дотику

```
<TouchableOpacity
  onPressIn={() => console.log('Дотик до елемента')}
  onPressOut={() => console.log('Відпускання елемента')}
>
  <Text>Торкніться та відпустіть</Text>
</TouchableOpacity>
```

`onPressIn` спрацьовує в момент контакту пальця з екраном, а `onPressOut` — коли користувач відпускає елемент.

Компонент	Події натискання	Опис
`Button`	`onPress`	Базова кнопка, що реагує на натискання
`TouchableOpacity`	`onPress`, `onLongPress`, `onPressIn`, `onPressOut`	Контейнер, що змінює прозорість при натисканні
`TouchableHighlight`	`onPress`, `onLongPress`, `onPressIn`, `onPressOut`	Контейнер, що змінює колір фону при натисканні
`TouchableWithoutFeedback`	`onPress`	Використовується для обробки натискання без візуальних змін
`Pressable`	`onPress`, `onLongPress`, `onPressIn`, `onPressOut`	Гнучкий контейнер для обробки жестів із можливістю керування стилями в різних станах
`Text`	`onPress`	Дозволяє робити текст натискним
`Image`	`onPress` (через `TouchableOpacity` або `Pressable`)	Дозволяє робити зображення інтерактивними

Події введення тексту **(Text Input Events)** у **React Native**

Text Input Events – це події, що реагують на введення тексту в поле вводу (`TextInput`). Вони використовуються для відстеження змін тексту, обробки фокусу, підтвердження введення та інших взаємодій користувача.

onChangeText – Обробка змін у тексті

📌 **Призначення:** Викликається при кожному введеному або видаленому символі.

```
const [text : string , setText] = useState( initialState: '');

return (
    <View>
        <TextInput
            placeholder="Введіть текст"
            value={text}
            onChangeText={(newText : string ) : void  => setText(newText)}
            style={{ borderBottomWidth: 1, padding: 10 }}
        />
        <Text>Ви ввели: {text}</Text>
    </View>
);
```

onEndEditing – Завершення редагування

📌 **Призначення:** Викликається, коли користувач завершує введення та виходить з поля.

📌 **Корисно для валідації введених даних після редагування.**

```
<View>
    <TextInput
        placeholder="Введіть текст"
        value={text}
        onChangeText={(newText : string ) : void  => setText(newText)}
        onEndEditing={() : any  => console.log('Редагування завершено')}
        style={{ borderBottomWidth: 1, padding: 10 }}
    />
    <Text>Ви ввели: {text}</Text>
</View>
```

onSubmitEditing – Підтвердження введення

📌 **Призначення:** Спрацьовує після натискання "Enter" (або "Done" на мобільній клавіатурі).

📌 **Корисно для форм, пошуку та переходу до наступного поля.**

```
<View>
    <TextInput
        placeholder="Введіть текст"
        value={text}
        onChangeText={(newText : string ) : void  => setText(newText)}
        returnKeyType="done"
        onSubmitEditing={() : any  => console.log('Введення підтверджено')}
        style={{ borderBottomWidth: 1, padding: 10 }}
    />
    <Text>Ви ввели: {text}</Text>
</View>
```

onFocus – Отримання фокусу

📌 **Призначення:** Викликається, коли поле отримує фокус (користувач починає вводити текст).

📌 **Корисно для зміни стилю або відображення підказок.**

```
<View>
    <TextInput
        placeholder="Введіть текст"
        value={text}
        onChangeText={(newText : string ) : void  => setText(newText)}
        onFocus={() : any  => console.log('Фокус отримано')}
        style={{ borderBottomWidth: 1, padding: 10 }}
    />
    <Text>Ви ввели: {text}</Text>
</View>
```

onBlur – Втрата фокусу

📌 **Призначення:** Викликається, коли користувач торкається поза межами поля.

📌 **Корисно для приховування клавіатури або автоматичного збереження даних.**

```
<TextInput
    placeholder="Введіть текст"
    value={text}
    onChangeText={(newText : string ) : void  => setText(newText)}
    onBlur={() : any  => console.log('Фокус втрачено')}
    style={{ borderBottomWidth: 1, padding: 10 }}
/>
<Text>Ви ввели: {text}</Text>
```

onKeyPress – Відстеження натискання клавіш

📌 **Призначення:** Викликається при кожному натисканні клавіші, включаючи `Backspace`.

📌 **Корисно для обробки натискання "Backspace" або створення кастомних шорткатів.**

```
<TextInput
    placeholder="Введіть текст"
    value={text}
    onChangeText={(newText : string ) : void  => setText(newText)}
    onKeyPress={({ nativeEvent : TextInputKeyPressEventData  }) : any  => console.log('Натиснута клавіша:', nativeEvent.key)}
    style={{ borderBottomWidth: 1, padding: 10 }}
/>
<Text>Ви ввели: {text}</Text>
```

onSelectionChange – Зміна виділення тексту

📌 **Призначення:** Викликається, коли змінюється позиція курсора або виділення.

📌 **Корисно для роботи з автозаповненням або підказками.**

```
<TextInput
    placeholder="Введіть текст"
    value={text}
    onChangeText={(newText : string ) : void  => setText(newText)}
    onSelectionChange={({ nativeEvent : TextInputSelectionChangeEventD... }) : any  => console.log('Позиція курсора:', nativeEvent.selection)}
    style={{ borderBottomWidth: 1, padding: 10 }}
/>
<Text>Ви ввели: {text}</Text>
```

```
(NOBRIDGE) LOG  Позиція курсора: {"end": 1, "start": 1}
(NOBRIDGE) LOG  Позиція курсора: {"end": 2, "start": 2}
(NOBRIDGE) LOG  Позиція курсора: {"end": 3, "start": 3}
(NOBRIDGE) LOG  Позиція курсора: {"end": 4, "start": 4}
(NOBRIDGE) LOG  Позиція курсора: {"end": 5, "start": 5}
(NOBRIDGE) LOG  Позиція курсора: {"end": 6, "start": 6}
(NOBRIDGE) LOG  Позиція курсора: {"end": 7, "start": 7}
```

onContentSizeChange – Динамічна зміна розміру

📌 **Призначення:** Викликається, коли змінюється висота тексту (`multiline`).

```
<TextInput
    placeholder="Введіть текст"
    value={text}
    multiline
    onChangeText={(newText) => setText(newText)}
    onContentSizeChange={(event) => console.log('Новий розмір:', event.nativeEvent.contentSize)}
    style={{ borderBottomWidth: 1, padding: 10 }}
/>
<Text>Ви ввели: {text}</Text>
```

```
Новий розмір: {"height": 40.3636360168457, "width": 392.7272644042969}
Новий розмір: {"height": 40, "width": 392.7272644042969}
Новий розмір: {"height": 56.727272033691406, "width": 392.7272644042969}
Новий розмір: {"height": 73.45454406738281, "width": 392.7272644042969}
Новий розмір: {"height": 90.18181610107422, "width": 392.7272644042969}
```

Події клавіатури (Keyboard Events)

Події клавіатури (`Keyboard Events`) дозволяють відстежувати відкриття, закриття та зміни статусу клавіатури у React Native. Це корисно для **автоматичного зміщення інтерфейсу**, приховування клавіатури або зміни поведінки додатка залежно від її стану.

Відстеження відкриття та закриття клавіатури

📌 **Призначення:** Визначає, коли клавіатура відкривається або закривається.

```
useEffect( effect: () => {
    const showSubscription = Keyboard.addListener( eventType: 'keyboardDidShow', listener: () => {
        console.log('Клавіатура відкрита');
    });

    const hideSubscription = Keyboard.addListener( eventType: 'keyboardDidHide', listener: () => {
        console.log('Клавіатура закрита');
    });

    return () => {
        showSubscription.remove();
        hideSubscription.remove();
    };
}, deps: []);
```

Події прокрутки (Scroll Events)

Події прокрутки (`Scroll Events`) у React Native використовуються для відстеження руху користувача при перегортанні контенту у **`ScrollView`**, **`FlatList`**, **`SectionList`** та інших прокручуваних компонентах. Це корисно для **динамічного завантаження даних, приховування елементів під час прокрутки, визначення кінця списку тощо**.

Відстеження позиції прокрутки (onScroll)

📌 **Призначення:** Викликається під час кожного руху прокручуваного контейнера.

📌 **Корисно для відстеження позиції прокрутки та створення ефектів (наприклад, приховування заголовка).**

```
<ScrollView
    onScroll={(event) => console.log('Поточна позиція Y:', event.nativeEvent.contentOffset.y)}
    scrollEventThrottle={16} // Частота оновлення події (в мс)
>
    {[...Array( arrayLength: 50)].map((_, i) => (
        <Text key={i} style={{ fontSize: 20, padding: 10 }}>Елемент {i + 1}</Text>
    ))}
</ScrollView>
```

```
(NOBRIDGE) LOG  Поточна позиція Y: 112
(NOBRIDGE) LOG  Поточна позиція Y: 111.2727279663086
(NOBRIDGE) LOG  Поточна позиція Y: 110.18181610107422
(NOBRIDGE) LOG  Поточна позиція Y: 108.36363983154297
(NOBRIDGE) LOG  Поточна позиція Y: 106.54545593261719
(NOBRIDGE) LOG  Поточна позиція Y: 105.45454406738281
```

Визначення, коли користувач починає/закінчує прокручування **(onScrollBeginDrag, onScrollEndDrag)**

📌 **Призначення:** Використовується для визначення початку та кінця взаємодії користувача з прокруткою.

📌 **Корисно для анімацій або зупинки оновлення контенту під час прокрутки.**

```
<ScrollView
    onScroll={(event) => console.log('Поточна позиція Y:', event.nativeEvent.contentOffset.y)}
    scrollEventThrottle={16} // Частота оновлення події (в мс)
    onScrollBeginDrag={() => console.log('Користувач розпочав прокрутку')}
    onScrollEndDrag={() => console.log('Користувач закінчив прокрутку')}
>
    {[...Array( arrayLength: 50)].map((_, i) => (
        <Text key={i} style={{ fontSize: 20, padding: 10 }}>Елемент {i + 1}</Text>
    ))}
</ScrollView>
```

Визначення інерційної прокрутки **(onMomentumScrollBegin, onMomentumScrollEnd)**

📌 **Призначення:** Використовується для визначення моменту, коли список продовжує рухатися після того, як користувач відпустив його.

📌 **Корисно для додавання ефектів або довантаження контенту після завершення інерційного руху.**

```
<ScrollView
    onScroll={(event) => console.log('Поточна позиція Y:', event.nativeEvent.contentOffset.y)}
    scrollEventThrottle={16} // Частота оновлення події (в мс)
    onMomentumScrollBegin={() => console.log('Інерційна прокрутка почалася')}
    onMomentumScrollEnd={() => console.log('Інерційна прокрутка завершена')}
>
    {[...Array( arrayLength: 50)].map((_, i) => (
        <Text key={i} style={{ fontSize: 20, padding: 10 }}>Елемент {i + 1}</Text>
    ))}
</ScrollView>
```

Автоматична прокрутка (scrollTo)

📌 **Призначення:** Використовується для програмної прокрутки списку.

```
<Button title="Прокрутити вниз" onPress={() => scrollViewRef.current.scrollTo({ y: 500, animated: true })} />
<ScrollView ref={scrollViewRef} style={{ marginTop: 10 }}>
    {[...Array( arrayLength: 30)].map((_, i) => (
        <Text key={i} style={{ fontSize: 20, padding: 10 }}>Елемент {i + 1}</Text>
    ))}
</ScrollView>
```

```
<Button title="Прокрутити в кінець" onPress={() => scrollViewRef.current.scrollToEnd({ animated: true })} />
<ScrollView ref={scrollViewRef} style={{ marginTop: 10 }}>
    {[...Array( arrayLength: 30)].map((_, i) => (
        <Text key={i} style={{ fontSize: 20, padding: 10 }}>Елемент {i + 1}</Text>
    ))}
</ScrollView>
```

Довантаження контенту при досягненні кінця списку **(FlatList + onEndReached)**

Призначення: Використовується для автоматичного підвантаження даних, коли користувач доходить до кінця списку.

📌 **Корисно для реалізації "нескінченного скролу" у списках.**

```
<FlatList
    data={data}
    renderItem={({ item }) => <Text style={{ padding: 20 }}>{item}</Text>}
    keyExtractor={(item, index) => index.toString()}
    onEndReached={loadMore}
    onEndReachedThreshold={0.5} // Завантажувати контент, коли користувач прокрутив 50% списку
/>
```

Події апаратних кнопок (Hardware Events)

Події апаратних кнопок (`Hardware Events`) дозволяють відстежувати взаємодію користувача з фізичними кнопками пристрою, такими як **"Назад" (Back Button) на Android**

📌 **iOS не дозволяє обробляти натискання кнопки "Назад", оскільки там немає апаратної кнопки для цієї дії.**

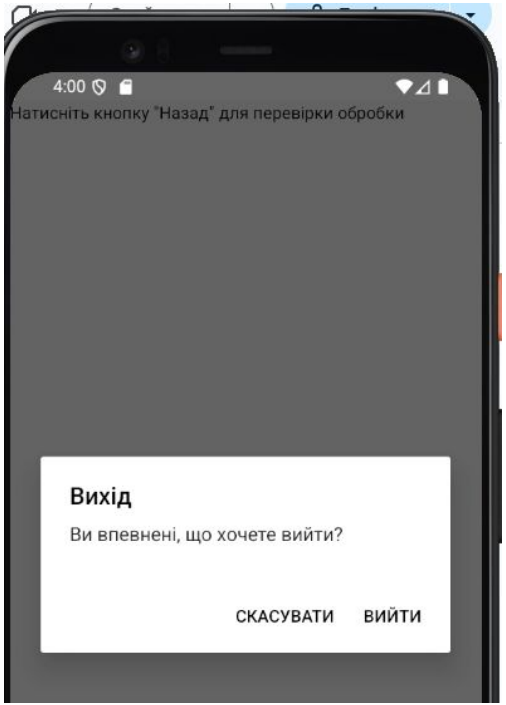
Обробка кнопки "Назад" **(BackHandler)** на **Android**

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

Show usages  new *
const App = () => {
    useEffect( effect: () => {
        Show usages  new *
        const backAction = () => {
            Alert.alert( title: "Вихід",  message: "Ви впевнені, що хочете вийти?",  buttons: [
                { text: "Скасувати", style: "cancel" },
                { text: "Вийти", onPress: () => BackHandler.exitApp() }
            ]);
            return true; // Запобігає закриттю додатка
        };

        const backHandler = BackHandler.addEventListener( eventName: 'hardwareBackPress', backAction);

        return () => backHandler.remove(); // Видаляємо обробник при виході з екрану
    },  deps: []);
```



Закриття додатка вручну (BackHandler.exitApp())

📌 **Призначення:** Використовується для закриття додатка при натисканні "Назад".

📌 **Корисно для кнопки "Вийти" у меню.**

```
import { BackHandler, Button } from 'react-native';

Show usages  new *
const App = () => {
    return <Button title="Закрити додаток" onPress={() => BackHandler.exitApp()} />;
};

Show usages  new *
export default App;
```

Події стану застосунку (App State Events)

Події стану застосунку (`App State Events`) дозволяють відстежувати, коли додаток **активний, у фоні або неактивний**. Це корисно для **паузи відео, збереження даних, обробки виходу з додатка та оптимізації ресурсів**.

- `active`- Програма працює на передньому плані
- `background`- Програма працює у фоновому режимі. Користувач:
  - в іншій програмі
  - на головному екрані
  - [Android] на іншому `Activity`(навіть якщо він був запущений вашою програмою)
- [iOS] `inactive`– це стан, який виникає під час переходу між активним і фоновим режимами, а також під час періодів бездіяльності, таких як перехід у режим багатозадачності, відкриття Центру сповіщень або у разі вхідного дзвінка.

Відстеження змін стану застосунку **(AppState.change)**

📌 **Призначення:** Викликається щоразу, коли додаток переходить у фон, активний режим або стає неактивним.

📌 **Корисно для виявлення, коли користувач залишає або повертається в застосунок.**

```
useEffect( effect: () => {
    const subscription = AppState.addEventListener( type: 'change', listener: (nextAppState) => {
        console.log('Стан додатка змінився:', nextAppState);
        setAppState(nextAppState);
    });

    return () => {
        subscription.remove(); // Видаляємо слухача при виході з компонента
    };
}, deps: []);

return (
    <View>
        <Text>Поточний стан: {appState}</Text>
    </View>
);
```

```
LOG  Стан додатка змінився: background
LOG  Стан додатка змінився: active
```

Призупинення дій, коли застосунок переходить у фон

📌 **Призначення:** Використовується для паузи відео, зупинки анімацій або збереження прогресу.

```
useEffect( effect: () => {
    const subscription = AppState.addEventListener( type: 'change', listener: (nextAppState) => {
        if (nextAppState === 'background') {
            console.log('Додаток перейшов у фон, зупиняємо музику...');
            // Зупинити відтворення аудіо або відео
        }
    });

    return () => subscription.remove();
}, deps: []);
```

Події жестів **(Gestures)**

Події жестів у **React Native** дозволяють обробляти взаємодію користувача з екраном за допомогою **дотиків, свайпів, перетягувань, масштабування та довгих натискань**. Вони використовуються для створення **інтерактивних елементів керування, жестової навігації та анімацій**.

Відстеження дотику **(onTouchStart, onTouchMove, onTouchEnd)**

📌 **Призначення:** Використовується для відстеження **дотику, переміщення та відпускання**

```
<View
    onTouchStart={() => console.log('onTouchStart')}
    onTouchMove={() => console.log('onTouchMove')}
    onTouchEnd={() => console.log('onTouchEnd')}
    style={{ flex: 1, justifyContent: 'center', alignItems: 'center', backgroundColor: 'lightblue' }}
>
    <Text>Move</Text>
</View>
```

PanResponder

📌 **Що таке `PanResponder`?**

`PanResponder` – це **вбудований API React Native**, який дозволяє **відстежувати рухи пальця по екрану** (свайпи, перетягування тощо).

🔹 Використовується для:

✔ **Свайпів (onSwipe)**

✔ **Перетягування елементів (drag & drop)**

✔ **Кастомних жестів**

Метод	Опис	Приклад
`onStartShouldSetPanResponder`	Чи слід обробляти дотик?	Використовується для вмикання жестів
`onMoveShouldSetPanResponder`	Чи слід реагувати на рух?	Використовується для свайпів
`onPanResponderMove`	Викликається при русі пальця	Використовується для **drag & drop**
`onPanResponderRelease`	Викликається при відпусканні пальця	Використовується для свайпів
`onPanResponderTerminate`	Викликається при скасуванні жесту	Використовується, якщо жест перервано

Визначення свайпів (onSwipe)

📌 **Призначення:** Визначає, чи був свайп вліво чи вправо.

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

Show usages  new *
const App = () => {
    const panResponder = useRef(
        PanResponder.create({
            onStartShouldSetPanResponder: () => true,
            onPanResponderEnd: (evt, gestureState) => {
                if (gestureState.dx > 50) {
                    console.log('Свайп вправо');
                } else if (gestureState.dx < -50) {
                    console.log('Свайп вліво');
                }
            },
        })
    ).current;

    return (
        <View {...panResponder.panHandlers} style={{ flex: 1, justifyContent: 'center', alignItems: 'center', backgroundColor: 'lightgray' }}>
            <Text>Свайпніть ліворуч або праворуч</Text>
        </View>
    );
};
```

Перетягування елемента **(Drag & Drop)**

📌 **Призначення:** Дозволяє користувачу **перетягувати елемент по екрану**.

```
const App = () => {
    const position = useRef(new Animated.ValueXY()).current;

    const panResponder = useRef(
        PanResponder.create({
            onStartShouldSetPanResponder: () => true,
            onPanResponderMove: Animated.event( argMapping: [null, { dx: position.x, dy: position.y }], config: { useNativeDriver: false }),
            onPanResponderRelease: () => {
                Animated.spring(position, config: { toValue: { x: 0, y: 0 }, useNativeDriver: false }).start();
            },
        })
    ).current;

    return (
        <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Animated.View {...panResponder.panHandlers} style={[position.getLayout(), { width: 100, height: 100, backgroundColor: 'red' }]} />
        </View>
    );
};

Show usages  new *
export default App;
```

Використання **react-native-gesture-handler** для складних жестів

Що таке `react-native-gesture-handler`?

`react-native-gesture-handler` — це бібліотека для **ефективної обробки жестів** у React Native, яка замінює стандартні події (`onTouchStart`, `onPanResponder`) та забезпечує **гнучку, плавну та продуктивну роботу з жестами**.

📌 **Основні переваги:**
✔ **Краща продуктивність** – жести обробляються у **рідному коді** (C++, Java, Objective-C), що робить їх плавними.
✔ **Менше затримок** – на відміну від `PanResponder`, події не проходять через JavaScript-обробку.
✔ **Гнучкість** – можна легко комбінувати жести (`Pinch`, `Fling`, `Pan`, `Tap`).
✔ **Підтримка вбудованих жестів** – використовується у `react-navigation` для створення навігації через свайпи.

react-native-gesture-handler

2.24.0 • Public • Published 7 days ago

Readme | Code Beta | 3 Dependencies | 1 727 Dependents | 145 Versions

Declarative API exposing platform native touch and gesture system to React Native.

React Native Gesture Handler provides native-driven gesture management APIs for building best possible touch-based experiences in React Native.

With this library gestures are no longer controlled by the JS responder system, but instead are recognized and tracked in the UI thread. It makes touch interactions and gesture tracking not only smooth, but also dependable and deterministic.

Installation

Install

```
> npm i react-native-gesture-handler
```

Repository

github.com/software-mansion/react-n...

Homepage

github.com/software-mansion/react-n...

Weekly Downloads

1 279 475

Version	License
2.24.0	MIT

Unpacked Size	Total Files
4.24 MB	1084

Як встановити **react-native-gesture-handler?**

Встановлення через npm або yarn:

```
npm install react-native-gesture-handler
```

Додавання `GestureHandlerRootView` у `App.js`:

```
import { GestureHandlerRootView } from 'react-native-gesture-handler';

const App = () => {
  return (
    <GestureHandlerRootView style={{ flex: 1 }}>
      {/* Ваш код */}
    </GestureHandlerRootView>
  );
};
```

📌 **`GestureHandlerRootView`** – це **кореневий контейнер** (`wrapper`), необхідний для роботи `react-native-gesture-handler`. Він забезпечує **правильне оброблення жестів** і **унеможливлює конфлікти з вбудованими жестами React Native**.

Додавання **GestureHandlerRootView** у **React Navigation**

Якщо ви використовуєте `react-navigation`, обгортайте `NavigationContainer` у `GestureHandlerRootView`:

```
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { GestureHandlerRootView } from 'react-native-gesture-handler';
import { Text } from 'react-native'

const Stack = createStackNavigator();

Show usages  new *
const App = () => {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <NavigationContainer>
                <Stack.Navigator id='stack'>
                    <Stack.Screen name="Home" component={() => <Text>Головний екран</Text>} />
                </Stack.Navigator>
            </NavigationContainer>
        </GestureHandlerRootView>
    );
};
```

Опції **GestureHandlerRootView** у **React Native**

`style` (Стилі контейнера)

Визначає стилі для кореневого `View`.

📌 **Приклад: Додаємо фон і розтягуємо на весь екран**

```
<GestureHandlerRootView style={{ flex: 1, backgroundColor: 'lightgray' }}>

  {/* Інший контент */}

</GestureHandlerRootView>
```

- `flex: 1` – обов’язковий, щоб контейнер займав весь екран.
- **Можна додавати кольори, падінги та інші стилі**.

`enabled` (Увімкнути або вимкнути обробку жестів)

- **Тип:** `boolean` (за замовчуванням `true`)
- Якщо `false`, усі жести всередині `GestureHandlerRootView` будуть заблоковані.

📌 **Приклад: Тимчасове відключення жестів**

```
<GestureHandlerRootView enabled={false}>

  {/* Жести тут не працюватимуть */}

</GestureHandlerRootView>
```

- **Може бути корисно для блокування жестів у певних ситуаціях (наприклад, під час завантаження екрану).**

GestureDetector

📌 **`GestureDetector`** – це компонент із бібліотеки `react-native-gesture-handler`, який використовується для **обробки жестів** у React Native.

- Він обгортає ваші компоненти та дозволяє застосовувати **Tap, Pan, Swipe, Pinch, Rotate** та інші жести.
- Підтримує **кілька жестів одночасно** та дозволяє керувати їхнім **пріоритетом і конфліктами**.

Як працює `GestureDetector`?

📌 **Схема обробки подій:**

1️⃣ Користувач торкається екрана →
2️⃣ `GestureDetector` передає подію у `GestureHandler` →
3️⃣ Перевіряється, який жест виконується (Tap, Pan, Fling тощо) →
4️⃣ Виконується відповідний обробник (`onUpdate`, `onEnd`)

TapGestureHandler – коротке натискання

📌 **Призначення:** Викликається при **одному короткому натисканні**.

```
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';

const tapGesture = Gesture.Tap()
    .onEnd( callback: () => console.log('Елемент натиснуто'));

Show usages  new *
const App = () => {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={tapGesture}>
                <View style={{ padding: 20, backgroundColor: 'lightblue' }}>
                    <Text>Натисніть мене</Text>
                </View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

Подвійне натискання **(Double Tap)** у **react-native-gesture-handler**

📌 **Призначення:** Використовується для обробки подвійного натискання, наприклад:

✔ **Лайк фото** (як у Instagram)

✔ **Збільшення зображення**

✔ **Швидке відкриття контенту**

```
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';

const doubleTapGesture = Gesture.Tap()
    .numberOfTaps( count: 2) // Подвійне натискання
    .onEnd( callback: () => console.log('Подвійне натискання виконано!'));

Show usages  new *
const App = () => {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={doubleTapGesture}>
                <View style={{ padding: 20, backgroundColor: 'lightblue', alignItems: 'center' }}>
                    <Text>Двічі натисніть на мене</Text>
                </View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

Опції **Gesture.Tap()** у **react-native-gesture-handler**

Обробник жесту `Gesture.Tap()` у `react-native-gesture-handler` має різні **опції для налаштування** поведінки натискання.

Опція	Опис
`.numberOfTaps(n)`	Кількість натискань перед спрацюванням (1 – одне натискання, 2 – подвійне тощо)
`.maxDuration(ms)`	Максимальний час між натисканням і відпусканням, щоб вважати його дійсним (мс)
`.maxDelay(ms)`	Максимальний інтервал між послідовними натисканнями (наприклад, для подвійного натискання)
`.minPointers(n)`	Мінімальна кількість пальців, що мають торкатися екрана для спрацювання жесту
`.maxDistance(px)`	Максимальна відстань (у пікселях), на яку можна зрушити палець під час натискання
`.onBegin(callback)`	Викликається, коли натискання починається
`.onEnd(callback)`	Викликається, коли натискання завершується

Довге натискання **(LongPressGestureHandler)** у **react-native-gesture-handler**

📌 **Призначення:** `LongPressGestureHandler` використовується для **обробки довгого натискання** на елемент.
✔ Викликається, якщо користувач **утримує палець** на елементі.
✔ Використовується для **контекстного меню, видалення, додаткових дій**.

Опція	Опис
`.minDuration(ms)`	Мінімальна тривалість натискання для спрацювання жесту (мс)
`.maxDistance(px)`	Максимальне зміщення пальця, щоб жест не скасувався (пікселі)
`.onBegin(callback)`	Викликається при початку утримання
`.onEnd(callback)`	Викликається при завершенні утримання
`.onFinalize(callback)`	Викликається у будь-якому разі (навіть якщо жест не спрацював)

Базовий приклад – Логування довгого натискання

📌 **Працює за замовчуванням після **500 мс утримання** (можна змінити через `.minDuration(ms)`).

```
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';

const longPressGesture = Gesture.LongPress()
    .onEnd( callback: () => console.log('Довге натискання виконано!'));

Show usages  new *
const App = () => {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={longPressGesture}>
                <View style={{ padding: 20, backgroundColor: 'lightblue', alignItems: 'center' }}>
                    <Text>Утримуйте для активації</Text>
                </View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

PanGestureHandler – Перетягування (Drag & Drop, Свайпи)

📌 **Призначення:** Переміщення елемента по екрану.

Опція	Опис	Приклад
`.minPointers(2)`	Мінімум 2 пальці для активації	Використовується для multi-touch
`.maxPointers(1)`	Дозволяє жест лише для 1 пальця	Запобігає випадковим multi-touch
`.minDistance(50)`	Мінімальна відстань для свайпу	Уникає випадкових свайпів
`.onBegin()`	Викликається при початку жесту	Використовується для зміни стилю
`.onUpdate()`	Викликається при русі пальця	Відстежує зміну координат X/Y
`.onEnd()`	Викликається після завершення жесту	Виконує логіку після руху
`.onFinalize()`	Викликається у будь-якому разі після жесту	Завершальний стан жесту

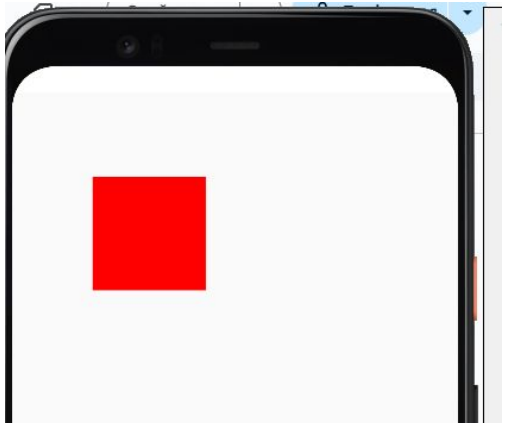
```tsx
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';
import Animated, {useSharedValue, useAnimatedStyle, withSpring} from 'react-native-reanimated';

Show usages  new *
const App = () => {
    const translateX = useSharedValue( initialValue: 0);
    const translateY = useSharedValue( initialValue: 0);

    const panGesture = Gesture.Pan()
        .onUpdate( callback: (event) => {
            translateX.value = event.translationX;
            translateY.value = event.translationY;
        })
        .onEnd( callback: () => {
            translateX.value = withSpring( toValue: 0);
            translateY.value = withSpring( toValue: 0);
        });

    const animatedStyle = useAnimatedStyle( updater: () => ({
        transform: [{ translateX: translateX.value }, { translateY: translateY.value }],
    }));

    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={panGesture}>
                <Animated.View style={[animatedStyle, { width: 100, height: 100, backgroundColor: 'red' }]} />
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```



PinchGestureHandler – Масштабування (Pinch to Zoom)

📌 **Призначення:** `PinchGestureHandler` використовується для **масштабування (збільшення або зменшення об'єктів)**

✔ Використовується для **зображень, карт, тексту або будь-яких інтерактивних UI-елементів**.

Опція	Опис
`.minPointers(n)`	Мінімальна кількість пальців для активації жесту (за замовчуванням `2`)
`.maxPointers(n)`	Максимальна кількість пальців для масштабування
`.onBegin(callback)`	Викликається при **початку** жесту (коли пальці торкаються екрана)
`.onUpdate(callback)`	Викликається при **зміні масштабу** (під час руху пальців)
`.onEnd(callback)`	Викликається після **завершення жесту** (коли пальці відпущені)
`.onFinalize(callback)`	Викликається **завжди** після завершення жесту (навіть якщо не спрацював)

```tsx
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';
import Animated, {useSharedValue, useAnimatedStyle, withSpring} from 'react-native-reanimated';

Show usages  new *
const App = () => {
    const scale = useSharedValue( initialValue: 1);

    const pinchGesture = Gesture.Pinch()
        .onUpdate( callback: (event) => {
            scale.value = event.scale;
        }).onEnd( callback: () => {
            scale.value = withSpring( toValue: 1); // Повернення до нормального розміру
        });

    const animatedStyle = useAnimatedStyle( updater: () => ({
        transform: [{ scale: scale.value }],
    }));

    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={pinchGesture}>
                <Animated.View style={[animatedStyle, { backgroundColor: 'lightgreen', padding: 50 }]}>
                    <Text>Pinch</Text>
                </Animated.View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

RotationGestureHandler – Обертання двома пальцями (Rotate)

📌 **Призначення:** `RotationGestureHandler` використовується для **обертання об'єктів двома пальцями**.

✔ Підходить для **зображень, інтерактивних UI-елементів, 3D-моделей тощо**.

✔ Використовує `useSharedValue` та `useAnimatedStyle` з **Reanimated 2**.

Опція	Опис
`.minPointers(n)`	Мінімальна кількість пальців для активації жесту (`2` за замовчуванням)
`.maxPointers(n)`	Максимальна кількість пальців, які можуть використовуватися
`.onBegin(callback)`	Викликається на початку обертання
`.onUpdate(callback)`	Викликається при зміні кута повороту
`.onEnd(callback)`	Викликається після завершення жесту
`.onFinalize(callback)`	Викликається у будь-якому разі після жесту (навіть якщо він не завершився)

```jsx
import { GestureHandlerRootView, GestureDetector, Gesture } from 'react-native-gesture-handler';
import Animated, {useSharedValue, useAnimatedStyle, withSpring} from 'react-native-reanimated';

Show usages  new *
const App = () => {
    const rotation = useSharedValue( initialValue: 0);

    const rotationGesture = Gesture.Rotation()
        .onUpdate( callback: (event) => {
            rotation.value = event.rotation;
        }).onEnd( callback: () => {
            rotation.value = withSpring( toValue: 0); // Повертаємо у вихідне положення
        });

    const animatedStyle = useAnimatedStyle( updater: () => ({
        transform: [{ rotate: `${rotation.value}rad` }],
    }));

    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={rotationGesture}>
                <Animated.View style={[animatedStyle, { backgroundColor: 'lightblue', padding: 50, alignItems: 'center' }]}>
                    <Text>Rotate</Text>
                </Animated.View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

FlingGestureHandler – Швидкий свайп (Fling)

📌 **Призначення:** `FlingGestureHandler` використовується для **обробки швидких свайпів у певному напрямку**.

✔ Відрізняється від `PanGestureHandler`, оскільки реагує **лише на швидке проведення пальцем, без перетягування**.

✔ Використовується для **жестової навігації, закриття модалок, видалення елементів тощо**.

Опція	Опис
`.direction(Directions.UP | DOWN | LEFT | RIGHT)`	Напрямок свайпу (обов'язковий параметр)
`.numberOfPointers(n)`	Мінімальна кількість пальців для виконання жесту
`.onBegin(callback)`	Викликається при **початку жесту**
`.onEnd(callback)`	Викликається при **завершенні жесту**
`.onFinalize(callback)`	Викликається у будь-якому разі після завершення жесту (навіть якщо не спрацював)

```jsx
import { GestureHandlerRootView, GestureDetector, Gesture, Directions } from 'react-native-gesture-handler';

const flingGesture = Gesture.Fling()
    .direction(Directions.RIGHT)
    .onEnd( callback: () => console.log('Меню відкрито'));

Show usages  new *
const App = () => {
    return (
        <GestureHandlerRootView style={{ flex: 1 }}>
            <GestureDetector gesture={flingGesture}>
                <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center', backgroundColor: 'lightblue' }}>
                    <Text>Швидко свайпніть вправо</Text>
                </View>
            </GestureDetector>
        </GestureHandlerRootView>
    );
};
```

Gesture.Exclusive() – Пріоритетний жест

🔹 Що таке `Gesture.Exclusive()`?

📌 `Gesture.Exclusive()` дозволяє **запобігати конфліктам жестів**.

- Якщо користувач виконує **декілька жестів одночасно**, **тільки один з них буде виконаний**.
- Решта жестів **ігноруються**.

Типове використання:

- Пріоритет **перетягування (`PanGestureHandler`) над прокруткою (`ScrollView`)**.
- Вибір між **свайпом (`FlingGestureHandler`) і натисканням (`TapGestureHandler`)**.
- Блокування масштабування (`PinchGestureHandler`), якщо працює перетягування (`PanGestureHandler`).

Як працює **Gesture.Exclusive()?**

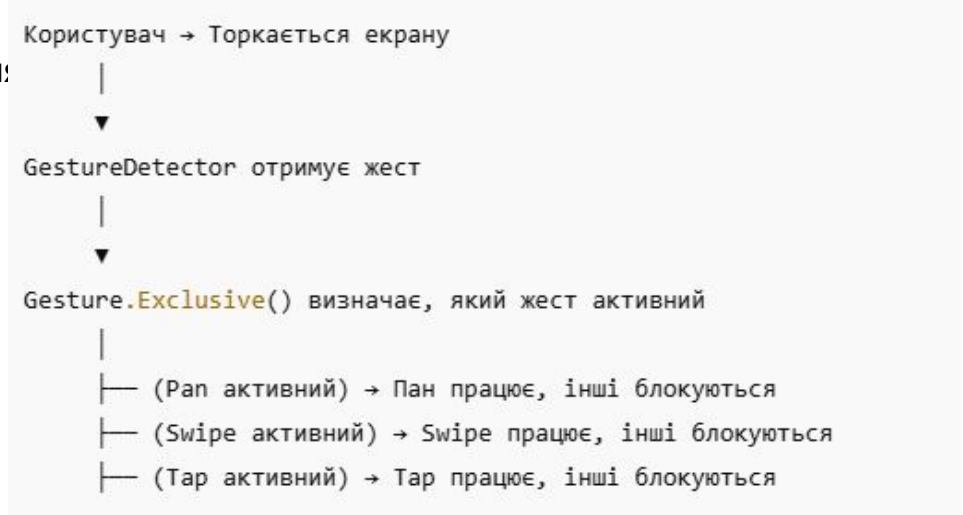
📌 **Механізм роботи:**

1️⃣ Користувач виконує дію (наприклад, перетягування та свайп одночасно).

2️⃣ `Gesture.Exclusive()` перевіряє, **який жест почався першим**.

3️⃣ Виконується тільки **один із жестів**.

4️⃣ Інші жести **блокуються**.



Основні пропси **Gesture.Exclusive()**

Пропс	Опис	Приклад
`Gesture.Exclusive(gesture1, gesture2, ...)`	Виконує тільки **один жест**, інші блокуються	`Gesture.Exclusive(panGesture, pinchGesture)`
`.onBegin()`	Викликається при старті жесту	`.onBegin(() => console.log('Жест розпочато'))`
`.onEnd()`	Викликається, коли жест завершено	`.onEnd(() => console.log('Жест завершено'))`
`.onFinalize()`	Викликається **завжди** після завершення жесту (навіть якщо не виконався)	`.onFinalize(() => console.log('Жест остаточно завершено'))`

```
// ⚡ Жест свайпу вправо (Fling)
const swipeGesture = Gesture.Fling()
    .direction(Directions.RIGHT)
    .onEnd( callback: () => console.log('Свайп вправо!'));

// ⚡ Жест натискання (Tap)
const tapGesture = Gesture.Tap()
    .onEnd( callback: () => console.log('Тап!'));

// ❗ Виконується тільки один жест – або свайп, або тап
const exclusiveGesture = Gesture.Exclusive(swipeGesture, tapGesture);

return (
    <GestureHandlerRootView style={{ flex: 1 }}>
        <GestureDetector gesture={exclusiveGesture}>
            <View style={styles.container}>
                <Text style={styles.text}>Спробуйте свайпнути вправо або натиснути</Text>
            </View>
        </GestureDetector>
    </GestureHandlerRootView>
);
```

Gesture.Simultaneous() – Одночасне виконання жестів

🔹 Що таке `Gesture.Simultaneous()`?

📌 `Gesture.Simultaneous()` дозволяє **виконувати кілька жестів одночасно**.

- Використовується, коли два або більше жестів можуть працювати **паралельно**.
- Наприклад, ви можете **масштабувати (`PinchGestureHandler`) і обертати (`RotationGestureHandler`) об'єкт одночасно**.

Типове використання:

- **Перетягування (`PanGestureHandler`) + Свайп (`FlingGestureHandler`)**
- **Масштабування (`PinchGestureHandler`) + Обертання (`RotationGestureHandler`)**

Як працює **Gesture.Simultaneous()?**

📌 **Механізм роботи:**

1️⃣ Користувач виконує **два жести одночасно** (наприклад, `Pinch` + `Rotate`).

2️⃣ `Gesture.Simultaneous()` **не блокує** жоден з них.

3️⃣ Обидва жести виконуються **незалежно один від одного**.

```
Користувач → Виконує одночасно два жести
      |
      ▼
GestureDetector отримує подію
      |
      ▼
Gesture.Simultaneous() дозволяє виконати обидва жести
      |
      ├── (Масштабування активне) → Об'єкт збільшується
      ├── (Обертання активне) → Об'єкт обертається
```

Gesture.Race() – Виконання першого жесту що спрацював

📌 **Що таке `Gesture.Race()`?**

- `Gesture.Race()` виконує **тільки один жест – той, який спрацював першим**.
- Якщо два або більше жестів активуються одночасно, **виграє той, який завершиться першим**.
- Інші жести **ігноруються після спрацювання одного з них**.

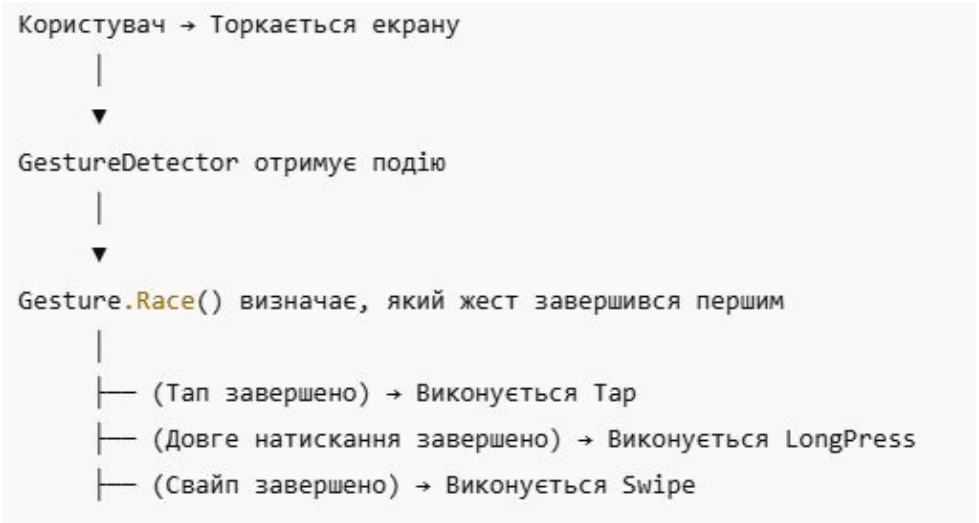
Типове використання:

✔ **Свайп (`FlingGestureHandler`) vs Перетягування (`PanGestureHandler`)**
✔ **Тап (`TapGestureHandler`) vs Довге натискання (`LongPressGestureHandler`)**
✔ **Свайп (`FlingGestureHandler`) vs Прокрутка (`ScrollView`)**

Як працює `Gesture.Race()`?

📌 **Механізм роботи:**

1. Користувач виконує **два або більше жести одночасно**.
2. `Gesture.Race()` чекає, який жест завершиться **першим**.
3. Виконується тільки **цей жест**.
4. Решта жестів **ігноруються**.



Метод	Що робить?	Приклад
`Gesture.Simultaneous()`	Обидва жести можуть виконуватися разом	`Gesture.Simultaneous(pinchGesture, rotateGesture)`
`Gesture.Exclusive()`	Виконується тільки **один жест**, інші блокуються	`Gesture.Exclusive(swipeGesture, panGesture)`
`Gesture.Race()`	Виконується перший жест, інші ігноруються	`Gesture.Race(tapGesture, longPressGesture)`