State Management & UI Controls

Asst. Prof. Monlica Wattana, Ph.D
Department of Computer Science,
Khon Kaen University

SC 362 007 Mobile Device Programming
CP 410 804 Programming for Mobile Application

Outline

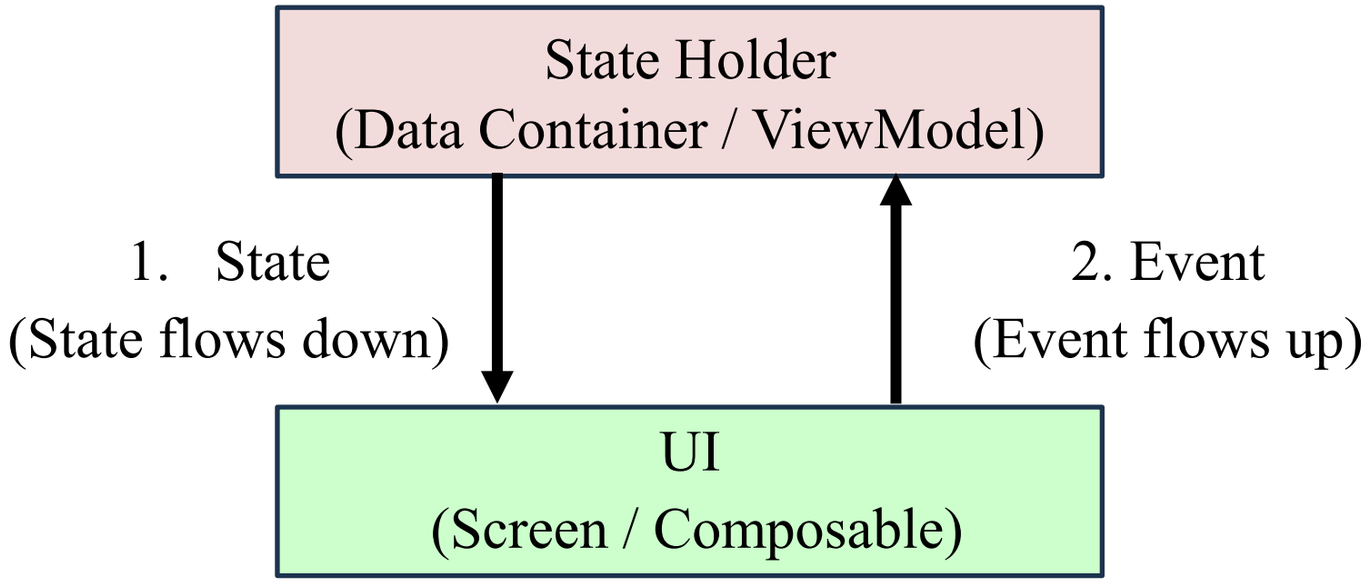
- State in Android Jetpack Compose
- UI Controls
  - Toast
  - Text
  - Button
  - TextField
  - Radio Button
  - Drop Down
  - Pickers
  - Scroll

State in Android Jetpack Compose

- The most of applications dynamically update their user interfaces. In Jetpack Compose, this is achieved through state management.
- State is any value that can change over time. It decides what should be displayed in the UI.
- MutableState objects holds a value and whenever the value is changed, it updates the UI (corresponding composables).

State in Android Jetpack Compose

- Unidirectional Data Flow (State & Events)



State in Android Jetpack Compose

- Create a MutableState object: 2 methods (using by and =)

```
var variableName by rememberSaveable { mutableStateOf(initialValue ) }
```

  - mutableStateOf(): a function accepts a value and creates an observable state holder. When the value changes, Compose automatically triggers a UI update (Recomposition).
  - rememberSaveable : a composable function helps us to store a single object in memory. A value computed by remember is stored during the initial composition, and returned whenever we update the UI.
  - by : it converts the MutableState object into a regular Kotlin variable.

State in Android Jetpack Compose

State Example: Use = instead of by Delegate. But, use counter.value to update or get the counter value.

```
@Composable
fun MyScreen(modifier: Modifier = Modifier) {
val counter = rememberSaveable { mutableStateOf(0) }

    Button( onClick = {
            counter.value++
        },
        modifier =modifier.padding(20.dp)
        )
     {
        Text(text = "Count: ${counter.value}")
    }
}
```

State in Android Jetpack Compose

State Example: Use by

** import androidx.compose.runtime.getValue

import androidx.compose.runtime.setValue

OR import androidx.compose.runtime.*

```
@Composable
fun MyScreen (modifier: Modifier = Modifier) {

    var counter by rememberSaveable {mutableStateOf(0)}
     Button(onClick = {
            counter++
        },
            modifier =modifier.padding(20.dp)
        ) {
            Text(text = "Counter: $counter")
          }
}
```

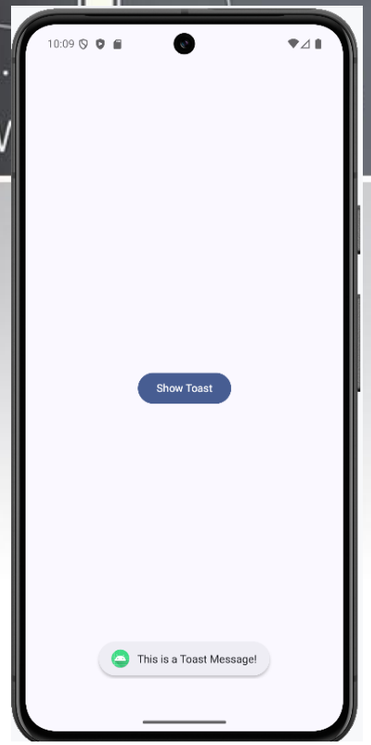
State in Android Jetpack Compose

- State Persistence: remember vs rememberSaveable
  - remember stores objects in the Composition memory.Issue: If you rotate the screen (Configuration Change), the Activity is recreated, and the memory is cleared. The state is lost!
  - remember: Use for temporary UI state (e.g., animations).
  - Use rememberSaveable. It saves the state into a Bundle. This allows the data to survive screen rotation and process death.
  - rememberSaveable: Use for user input (e.g., TextField, Checkbox).


ACCOUNT
SEARCH
SEARCH
NETWORK
ACCOUNT
APPLICATION
COMMUNICATION
WEBSITE
SEARCH
CONTENT
MONITORING
CONTENT
RESOURCE
WEBSITE

UI Controls

Toast



Toast is used to display a piece of text for a short span of time.

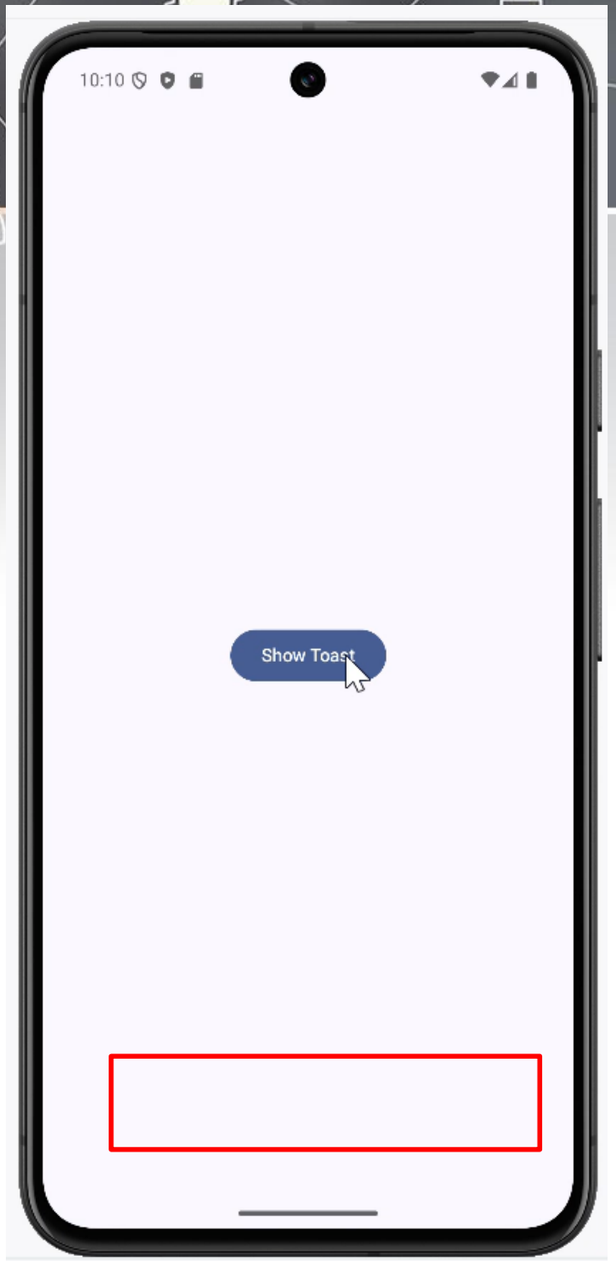
The Toast.makeText() method requires three parameters:

1. Context: The context of the application/activity.In Compose,
   use: val context = LocalContext.current
2. Message: The text string to display.
3. Duration: How long to show the message.Toast.LENGTH_SHORT or Toast.LENGTH_LONG

Important: You must call .show() inside an event handler (like onClick) or a SideEffect, **never directly inside a Composable function**.

Toast

```
@Composable
fun MyToast(modifier: Modifier = Modifier) {
    val contextForToast = LocalContext.current
    Column(
        modifier = modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Button(
            onClick = {
                Toast.makeText(
                    contextForToast,
                    text = "This is a Toast Message!",
                    duration = Toast.LENGTH_LONG
                ).show()
            }
        ) {
            Text(text = "Show Toast")
        }
    }
}
```



Text

- The Text Composable is the simplest and most used widget or component used in every application.
- Common Attributes of the Text Composable
  - text: The content string to display on the screen.
  - color: Sets the text color. You can use standard colors (e.g., Color.Red) or define your own.
  - fontSize: Sets the size of the text. (unit: .sp)
  - fontWeight: Sets the thickness of the font (e.g., FontWeight.Bold, FontWeight.Light).
  - fontStyle: Sets the style of the font, such as FontStyle.Italic.
  - textAlign: Aligns the text within its container (e.g., TextAlign.Center, TextAlign.End).

Text

- Common Attributes of the Text Composable (Cont.)
  - maxLines: Limits the number of visible lines. If text exceeds this, it gets truncated.
  - style: Allows advanced styling (e.g., Shadow, TextDecoration) or applying a Theme (e.g., MaterialTheme.typography.bodyLarge).
  - modifier: Used for layout adjustments like:
    - Background Color: Modifier.background(Color.Yellow)
    - Padding: Modifier.padding(16.dp)
    - Size: Modifier.width(...), Modifier.fillMaxWidth()

Text

Text Example:

```
@Composable
fun MyText(modifier: Modifier = Modifier) {
    Text(
        modifier = modifier
            .background(color = Color.Yellow) ,
        text = "My Text",
        fontSize = 20.sp,
        color= Color( red = 11, green = 11, blue = 200)

    )
}
```

My Text

Text

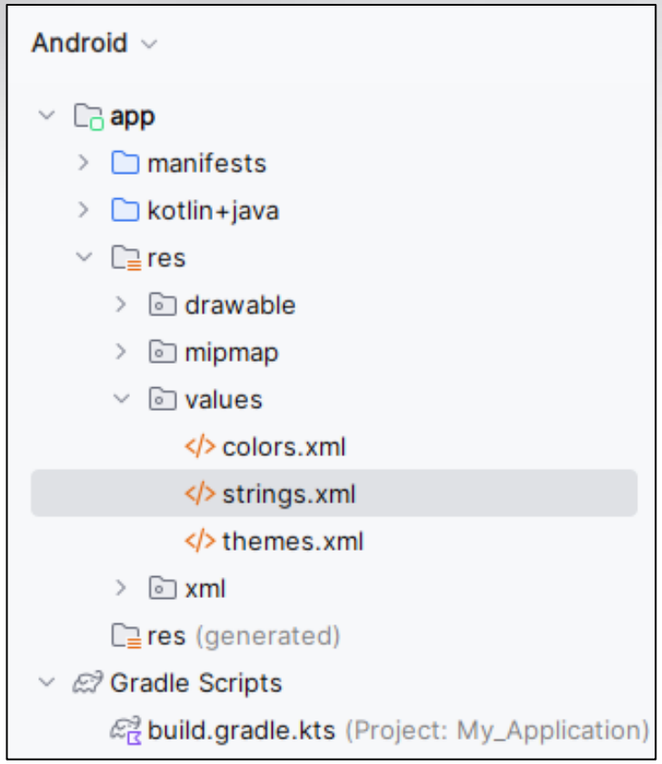
- Displaying Text from String Resource:

1. Define the String (in strings.xml)
   - Path: res > values > strings.xml
   - Add your text inside a <string> tag with a unique name.

```xml
<resources>
    <string name="app_name">My Application</string>
    <string name="my_text">This is my text.</string>
</resources>
```

2. Use in Composable (in Kotlin file)
   - Use the stringResource(...) function to load the text.

```kotlin
@Composable
fun MyText(modifier: Modifier = Modifier) {
    Text(
        modifier = modifier
            .background(color = Color.Yellow) ,
        text = stringResource(id= R.string.my_text),
        fontSize = 20.sp,
        color= Color( red = 11, green = 11, blue = 200)
    )
}
```



This is my text.

Adding Space with Padding

- Modifier: Padding is applied using the Modifier.padding(...) function.
- Universal: It is applicable to almost any composable (Text, Button, Row, etc.).
- Flexibility: You can apply padding to:
  - All sides equally. Modifier.padding(16.dp)

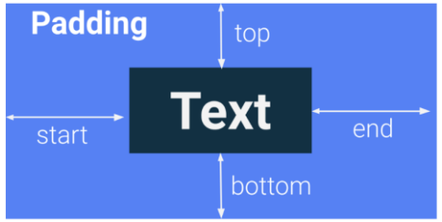
My Text

  - Horizontal or Vertical axes.
    Modifier.padding(horizontal = 16.dp, vertical = 36.dp)

My Text

  - Specific sides (Start, Top, End, Bottom).
    Modifier.padding(start = 8.dp, end = 32.dp, top = 24.dp, bottom = 56.dp)

My Text



Align and arrange the text

My Text

```
@Composable
fun MyText(modifier: Modifier = Modifier) {
    Text(
        modifier = modifier
            .background(color = Color.Yellow)
            .padding( all = 16.dp),
        text = "My Text",
        fontSize = 20.sp,
        color= Color( red = 11, green = 11, blue = 200)
    )
}
```

Spacer

- A Composable used to insert empty space (vertical or horizontal) between other elements in a layout.
- Sizing: Since Spacer is empty, you must define its size using modifiers:
  - Modifier.width(...) for horizontal space.
  - Modifier.height(...) for vertical space.
  - Modifier.size(...) for both.

Spacer

```
@Composable
fun MySpacer(modifier: Modifier = Modifier) {
    Row(
        modifier = modifier
            .border(width = 3.dp, color = Color.Green),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(text = "Text 1")

        Spacer(modifier = Modifier.width( width = 100.dp))

        Text(text = "Text 2")
    }
}
```

Text 1 Text 2

Spacer

- Modifier.height() sets the empty space vertically.

```
Spacer(modifier = Modifier.height(height = 100.dp))
```

- Modifier.size() adds the space both vertically and horizontally.

```
Spacer(modifier = Modifier.size(size = 100.dp))
```

Spacer

Modifier.weight(float)

- Distributes the remaining space among composables.
- Only works inside Row or Column.
- How it works:
  - weight(1f) on a single item: It fills all remaining space.
  - weight(1f) on multiple items: They share the space equally.
  - Useful Trick: Use Spacer(Modifier.weight(1f)) to push elements to opposite sides of the screen.

Spacer

```
@Composable
fun MySpacer2(modifier: Modifier = Modifier) {
    Row(
        modifier = modifier
            .border(width = 3.dp, color = Color.Green),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(text = "Text 1")

        Spacer(modifier = Modifier.weight( weight = 1f))

        Text(text = "Text 2")
    }
}
```

Spacer

```
@Composable
fun MySpacer(modifier: Modifier = Modifier) {
    Row(
        modifier = modifier
            .height( height = 40.dp)
            .border(width = 3.dp, color = Color.Green),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(text = "Text 1")
        // Spacer (20%)
        Spacer(modifier = Modifier.weight( weight = 0.2f)
            .fillMaxHeight()
            .background(Color.Cyan))
        // Text 2 (80%)
        Text(
            text = "Text 2",
            modifier = Modifier.weight( weight = 0.8f)
                .background(Color.Yellow)
        )
    }
}
```

Button

- A button control provides an area on the UI where a user can touch to initiate an action.

Common Buttons:

- Button() Button
- ElevatedButton(ElevatedButton
- FilledTonalButton() FilledTonalButton
- OutlinedButton() OutlinedButton
- TextButton() TextButton

Icon Buttons:

- IconButton()
- FilledIconButton()
- FilledTonalIconButton()
- OutlinedIconButton()

Icon Toggle Buttons:

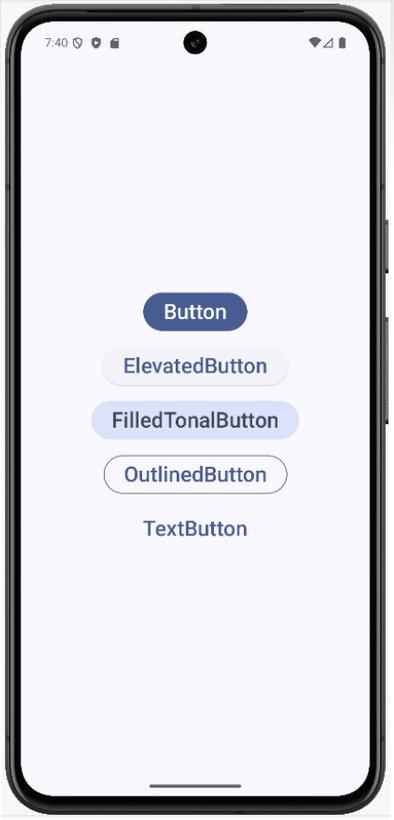
- IconToggleButton()
- FilledIconToggleButton()
- FilledTonalIconToggleButton()
- OutlinedIconToggleButton()

Button

Common Buttons

```
@Composable
fun MyButton(modifier: Modifier = Modifier) {
    val contextForToast = LocalContext.current
    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.spacedBy(
            space = 16.dp,
            alignment = Alignment.CenterVertically),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Button(
            onClick = {
                Toast.makeText(contextForToast, text = "Button",
                    duration = Toast.LENGTH_SHORT).show()
            }
        ) {
            Text(text = "Button",
                fontSize = 25.sp)
        }
```

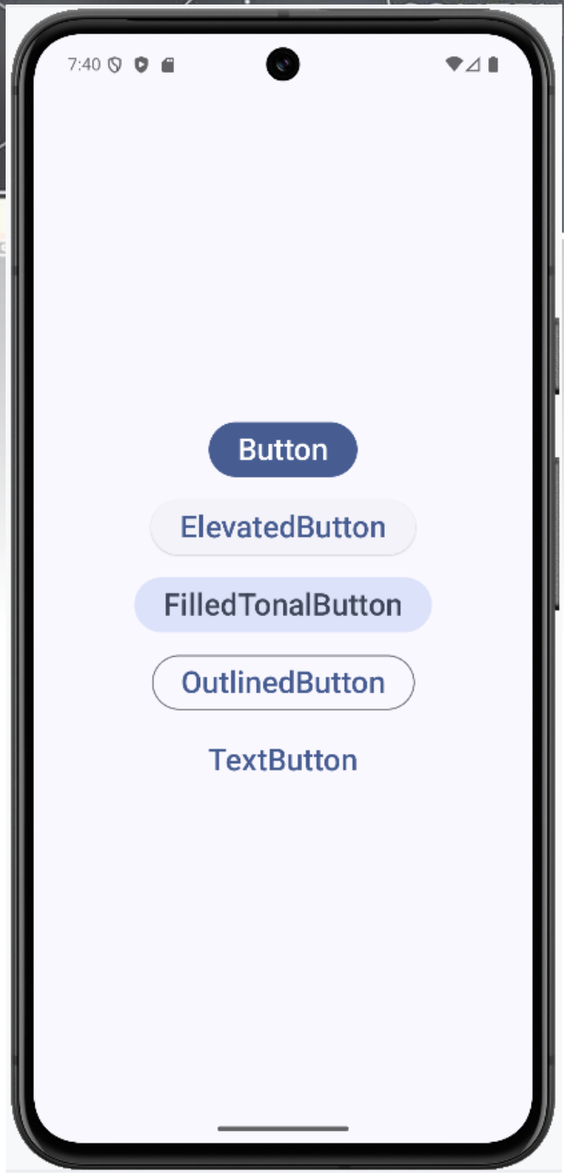
```
    ElevatedButton(
        onClick = {
            Toast.makeText(contextForToast, text = "ElevatedButton",
                duration = Toast.LENGTH_SHORT).show()
        }
    ) {
        Text(text = "ElevatedButton",
            fontSize = 25.sp)
    }
    FilledTonalButton(
        onClick = {
            Toast.makeText(contextForToast, text = "FilledTonalButton",
                duration = Toast.LENGTH_SHORT).show()
        }
    ) {
        Text(text = "FilledTonalButton",
            fontSize = 25.sp)
    }
    OutlinedButton(
        onClick = {
            Toast.makeText(contextForToast, text = "OutlinedButton",
                duration = Toast.LENGTH_SHORT).show()
        }
    ) {
        Text(text = "OutlinedButton",
            fontSize = 25.sp)
    }
```

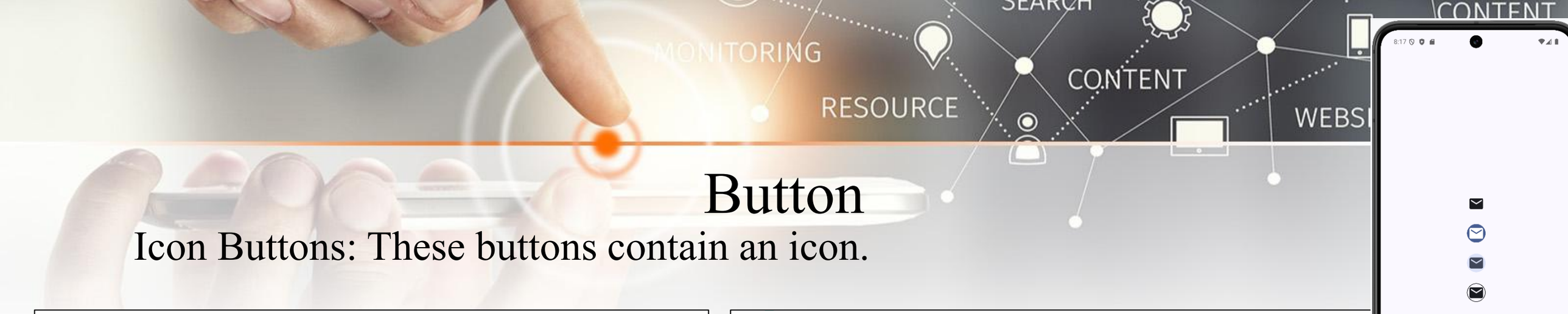


Button

Common Buttons: Ex. TextButton

```
        TextButton(
            onClick = {
                Toast.makeText(contextForToast, text = "TextButton",
                    duration = Toast.LENGTH_SHORT).show()
            }
        ) {
            Text(text = "TextButton",
                fontSize = 25.sp)
        }
    }
}
```





Button

Icon Buttons: These buttons contain an icon.

```
@Composable
fun MyIconButton(modifier: Modifier = Modifier) {
    val contextForToast = LocalContext.current
    Column(modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.spacedBy(
            space = 16.dp,
            alignment = Alignment.CenterVertically),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        IconButton(
            onClick = {
                Toast.makeText(contextForToast, text = "Icon Button",
                    duration = Toast.LENGTH_SHORT).show()
            }
        ) {
            Icon(imageVector = Icons.Default.Email,
                contentDescription = "Email",
                modifier = Modifier.size( size = 35.dp))
        }
```

```
    FilledIconButton(
        onClick = {
            Toast.makeText(contextForToast, text = "Filled Icon Button",
                duration = Toast.LENGTH_SHORT).show()
        }
    ) {
        Icon(imageVector = Icons.Default.Email,
            contentDescription = "Email",
             modifier = Modifier.size( size = 35.dp))
    }
    FilledTonalIconButton(
        onClick = {
            Toast.makeText(contextForToast, text = "Filled Tonal Icon Button",
                duration = Toast.LENGTH_SHORT).show()
        }
    ) {
        Icon(imageVector = Icons.Default.Email,
            contentDescription = "Email",
            modifier = Modifier.size( size = 35.dp))
    }
```

Button

Icon Buttons: These buttons contain an icon. (Cont)

```
        OutlinedIconButton(
            onClick = {
                Toast.makeText(contextForToast, text = "Outlined Icon Button",
                    duration = Toast.LENGTH_SHORT).show()
            }
        ) {
            Icon(imageVector = Icons.Default.Email,
                contentDescription = "Email",
                modifier = Modifier.size( size = 35.dp))
        }
    }
}
```

Button

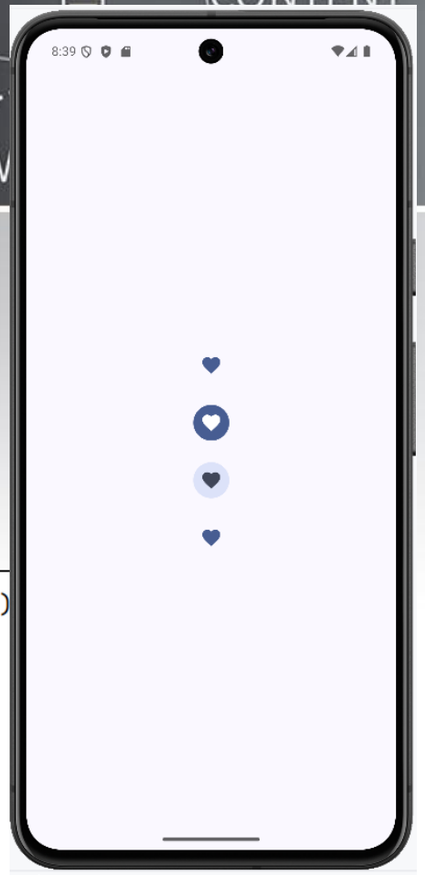
Icon Toggle Buttons: They have two states – on and off.

```
@Composable
fun MyIconToggleButton(modifier: Modifier = Modifier) {
    val contextForToast = LocalContext.current
    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.spacedBy(
            space = 16.dp,
            alignment = Alignment.CenterVertically),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        var checked1 by rememberSaveable {mutableStateOf( value = true)}
        IconToggleButton(
            checked = checked1,
            onCheckedChange = { _checked1 ->
                checked1 = _checked1
                Toast.makeText(contextForToast, text = "Icon Toggle Button $checked1",
                    duration = Toast.LENGTH_SHORT)
                    .show()
            }
        ) {
            Icon(
                imageVector = if (checked1) Icons.Filled.Favorite
                              else Icons.Outlined.FavoriteBorder,
                contentDescription = "Favorite Item"
            )
        }
```

```
    var checked2 by rememberSaveable { mutableStateOf( value = true) }
    FilledIconToggleButton(
        checked = checked2,
        onCheckedChange = { _checked2 ->
            checked2 = _checked2
            Toast.makeText(
                contextForToast,
                text = "Filled Icon Toggle Button $checked2",
                duration = Toast.LENGTH_SHORT
            ).show()
        }
    ) {
        Icon(
            imageVector = Icons.Default.Favorite,
            contentDescription = "Favorite Item"
        )
    }
```

Button

Icon Toggle Buttons: They have two states – on and off.



```
var checked3 by rememberSaveable {mutableStateOf( value = true)}
FilledTonalIconToggleButton(
    checked = checked3,
    onCheckedChange = { _checked3 ->
        checked3 = _checked3
        Toast.makeText(
            contextForToast,
            text = "Filled Tonal Icon Toggle Button $checked3",
            duration = Toast.LENGTH_SHORT
        ).show()
    }
) {
    Icon(
        imageVector = Icons.Default.Favorite,
        contentDescription = "Favorite Item"
    )
}
```

```
        var checked4 by rememberSaveable {mutableStateOf( value = true)}
        OutlinedIconToggleButton(
            checked = checked4,
            onCheckedChange = { _checked4 ->
                checked4 = _checked4
                Toast.makeText(
                    contextForToast,
                    text = "Outlined Icon Toggle Button $checked4",
                    duration = Toast.LENGTH_SHORT
                ).show()
            }
        ) {
            Icon(
                imageVector = Icons.Default.Favorite,
                contentDescription = "Favorite Item"
            )
        }
    }
}
```

TextField

- It is used to get the data from the user.
- For example, a signup page that contains a name, phone number, password, etc.
- Jetpack Compose provides three TextField APIs:
  - TextField

  - OutlinedTextField

  - BasicTextField

TextField

Attributes of the TextField

- value – This is the text to be shown in the field.
- onValueChange – It is a lambda that gets called every time the user updates the data in the field. It provides the new (updated) text.
- label – This is an optional text that floats to the top when the field is in focus or has content. placeholder – An optional text to be displayed when the text field is in focus and the input text is empty.
- leadingIcon – Placed at the start of the text field container.
- trailingIcon – Placed at the end of the text field container.
- visualTransformation = PasswordVisualTransformation() for password textfield

TextField

- Keyboard Options

1. Capitalization:
   - KeyboardCapitalization.None – Do not auto-capitalize text.
   - KeyboardCapitalization.Characters – Capitalize all characters.
   - KeyboardCapitalization.Words – Capitalize the first character of every word.
   - KeyboardCapitalization.Sentences – Capitalize the first character of each sentence.
2. Auto Correct:

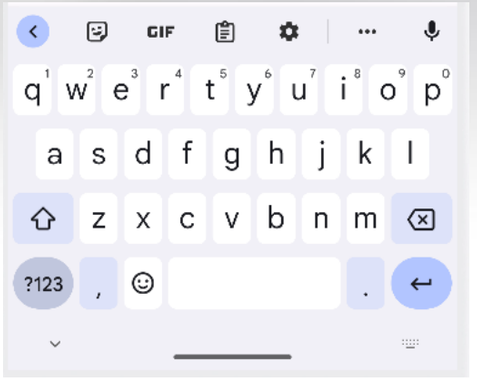
   A boolean value that informs the keyboard whether to enable auto-correct.

TextField

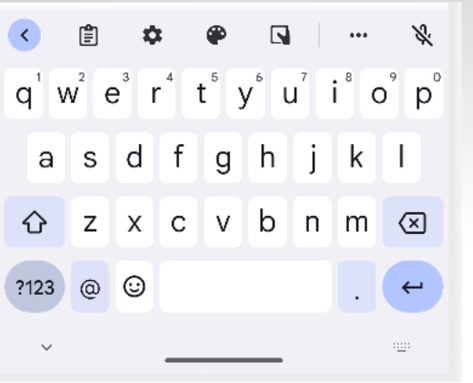
➢ Keyboard Options

3. Keyboard Type:
   - KeyboardType.Text – Regular keyboard.
   - KeyboardType.Number – For digits.
   - KeyboardType.Phone – For mobile numbers.
   - KeyboardType.Email – For email addresses.
   - KeyboardType.Password – For passwords.
   - KeyboardType.NumberPassword – For number passwords.

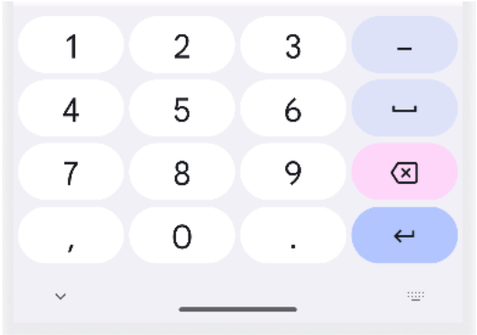
Text



Email

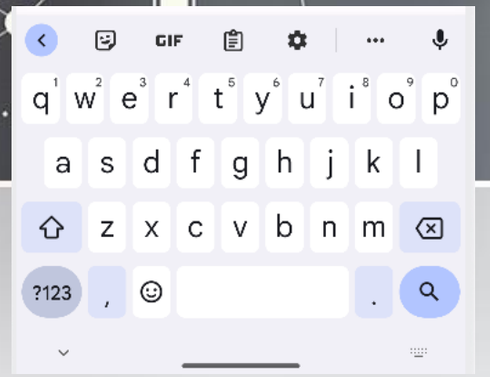


Number



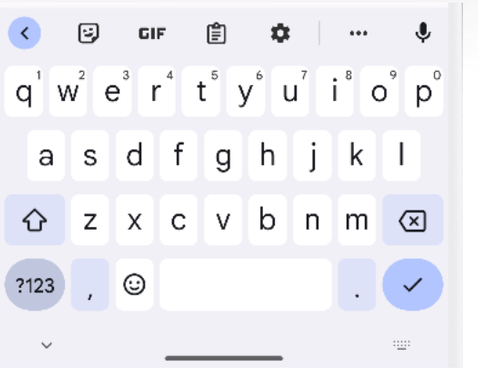
TextField

Keyboard Options



Search

4. IME Action:
   - ImeAction.Go – Represents that the user would like to go to the target of the text in the input i.e. visiting a URL.
   - ImeAction.Search – To execute a search.
   - ImeAction.Send – To send a text, for example, SMS.
   - ImeAction.Previous – To go back to the previous text field in a form.
   - ImeAction.Next – To move to the new line or next text field in a form.
   - ImeAction.Done – It represents the user done providing the input. For example, the last text field in a form.



Done

TextField

Simple TextField Example:

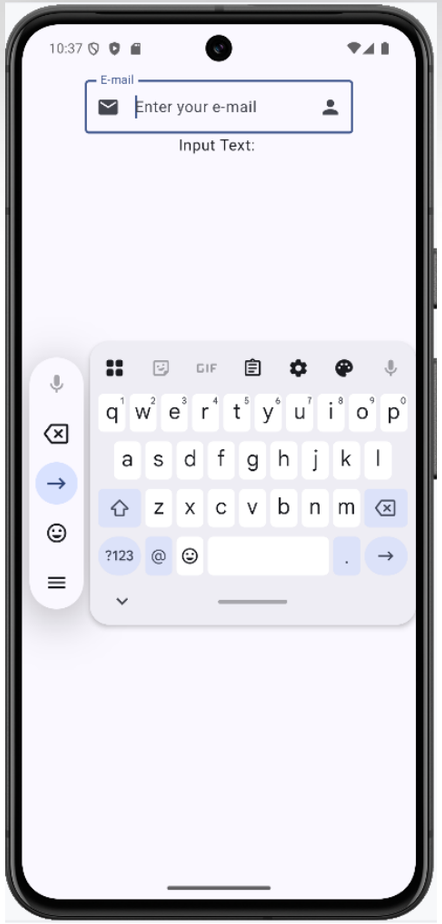
The value and onValueChange parameters are mandatory for a simple text field.

```
@Composable
fun MyTextField(modifier: Modifier = Modifier) {
    var value by rememberSaveable  { mutableStateOf( value = "") }
    Column(modifier = modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
    ){
        TextField(
            value = value,
            onValueChange = { newText ->
                value = newText
            },
            label = { Text(text = "Name") }
        )
        Text(text = "Input Text: $value")
    }
}
```

TextField

TextField Example 2:

```
@Composable
fun MyTextEmail(modifier: Modifier = Modifier) {
    var value by rememberSaveable {
        mutableStateOf( value = "")
    }
    Column(modifier = modifier.fillMaxSize(),
           horizontalAlignment = Alignment.CenterHorizontally
    ) {
        OutlinedTextField(
            value = value,
            onValueChange = { newText ->
                value = newText
            },
            leadingIcon = {
                Icon(imageVector = Icons.Default.Email, contentDescription = null)
            },
            trailingIcon = {
                Icon(imageVector = Icons.Default.Person, contentDescription = null)
            },
            label = { Text(text = "E-mail") },
            placeholder = { Text(text = "Enter your e-mail") },
            keyboardOptions = KeyboardOptions(
                keyboardType = KeyboardType.Email,
                imeAction = ImeAction.Go
            )
        )
        Text(text = "Input Text: $value")
    }
}
```

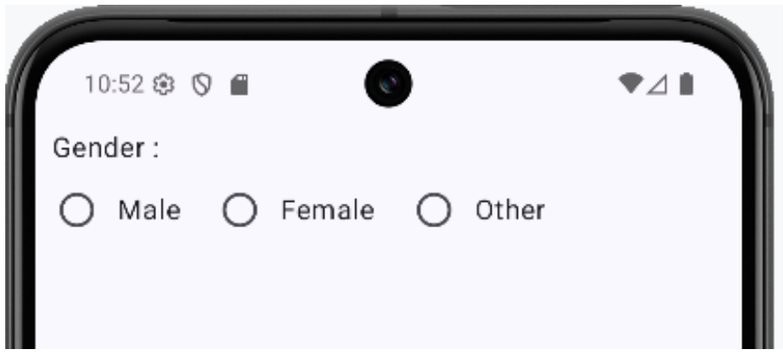


Radio Button

- Radio buttons allow users to select a single option from a list.
- They function as a group where only one option can be selected at a time (e.g., gender selection).
- In Jetpack Compose, we ensure this behavior by controlling them with a shared state variable.

RadioButton uses Row layout with Text().

```
Row(verticalAlignment = Alignment.CenterVertically) {
        RadioButton(
                selected = true,
                onClick = { }
        )
        Text(text = "Label")
}
```



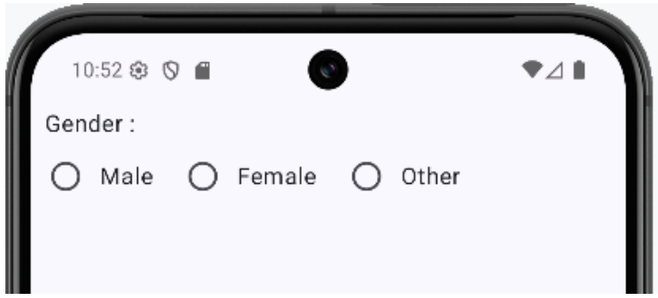
Radio Button

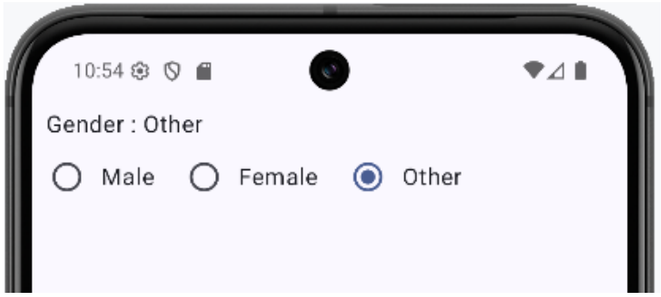
- Attributes of the Radio Button
  - selected – Whether this radio button is selected or not.
  - onClick – It is invoked when the RadioButton is clicked. If null, this RadioButton will not handle input events.
  - modifier – The modifier for changing the layout.
  - enabled – Whether the radio button is enabled. If it is false, the button will not be selectable and appears disabled.
  - interactionSource – The MutableInteractionSource for observing the interactions.
  - RadioButtonDefaults.colors – The colors to be applied to the button.

Radio Button

- Create RadioGroup

```
@Composable
fun GenderSelectionScreen(modifier: Modifier = Modifier) {
     val kinds = listOf("Male", "Female", "Other")
    val (selected, setSelected) = rememberSaveable { mutableStateOf( value = "") }
    Column(
        modifier = modifier
            .fillMaxWidth()
            .wrapContentHeight()
    ) {
        Text(
            text = "Gender : $selected",
            modifier = Modifier.padding( start = 10.dp)
        )
        Row(
            verticalAlignment = Alignment.CenterVertically
        ) {
            kinds.forEach { kind ->
                Row(
                    verticalAlignment = Alignment.CenterVertically,
                    modifier = Modifier
                        .clickable { setSelected(kind) }
                       .padding(end = 10.dp)
                ) {
                    RadioButton(
                        selected = (selected == kind),
                        onClick = { setSelected(kind) }
                    )
                    Text(
                        text = kind,
                    )
                }
            }
        }
    }
}
```



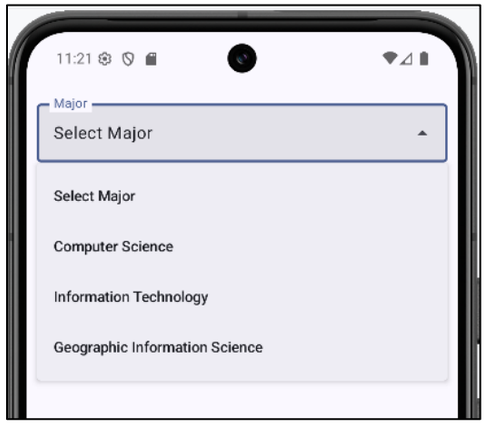


Dropdown

Use Exposed Dropdown Menu

It displays the currently selected item. We can implement it with the following APIs:

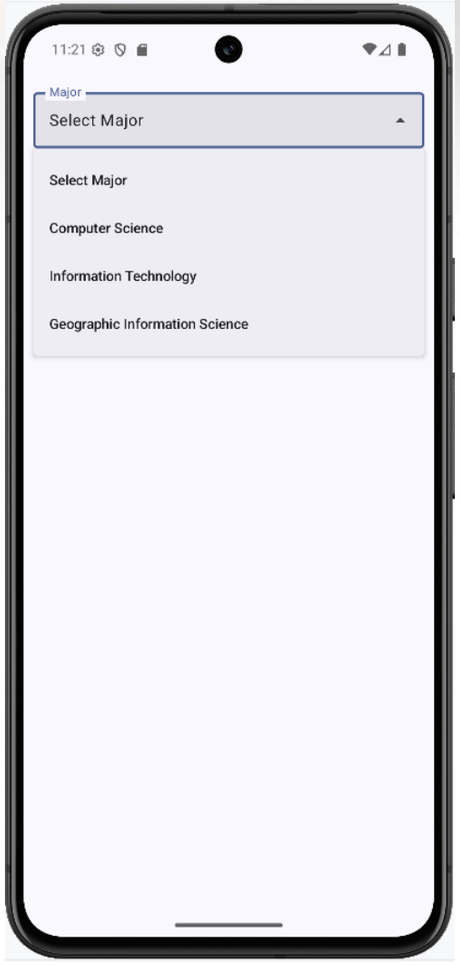
- ExposedDropdownMenuBox()
- TextField()
- ExposedDropdownMenu()
- DropdownMenuItem()



The ExposedDropdownMenuBox() contains TextField() and ExposedDropdownMenu(). In the menu, we add items using the DropdownMenuItem().

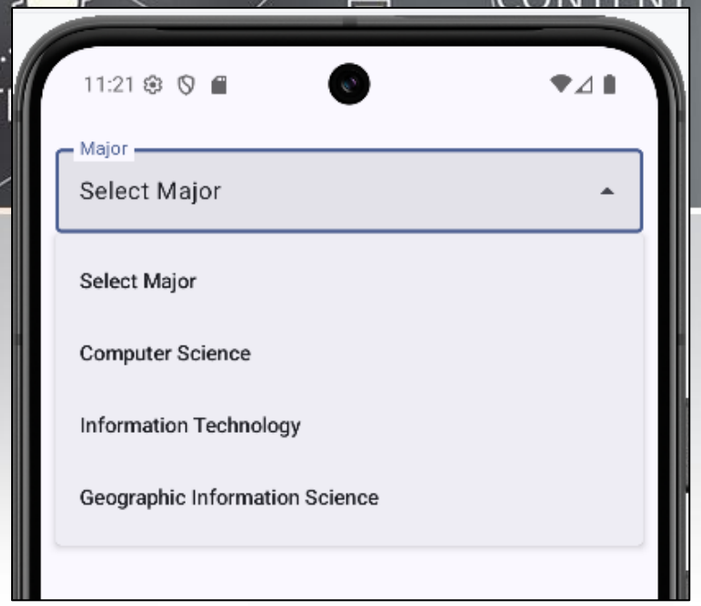
Dropdown


11:05
Major
Select Major


11:21
Major
Select Major
Select Major
Computer Science
Information Technology
Geographic Information Science


11:22
Major
Information Technology
Item is Information Technology

Dropdown




```
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun MyDropdown(modifier: Modifier = Modifier) {
    val majorList = listOf(
        "Select Major",
        "Computer Science",
        "Information Technology",
        "Geographic Information Science"
    )
    var expanded by rememberSaveable { mutableStateOf( value = false) }
    var selectedMajor by rememberSaveable { mutableStateOf( value = majorList[0]) }
    val context = LocalContext.current

    Column(
        modifier = modifier.padding( all = 10.dp)
    ) {
        ExposedDropdownMenuBox(
            expanded = expanded,
            onExpandedChange = { expanded = !expanded }
        ) {
```

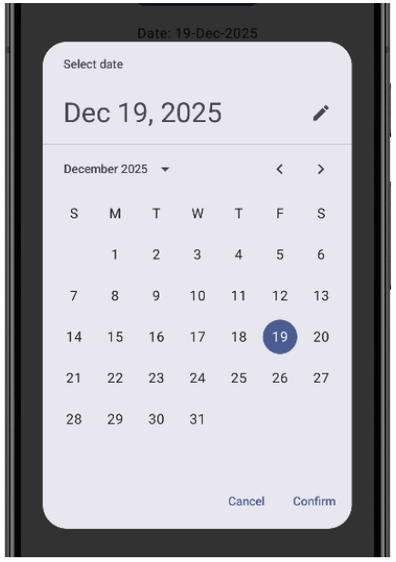
```
        OutlinedTextField(
            modifier = Modifier
                .menuAnchor( type = MenuAnchorType.PrimaryNotEditable,  enabled = true)
                .fillMaxWidth(),
            readOnly = true,
            value = selectedMajor,
            onValueChange = {},
            label = { Text( text = "Major") },
            trailingIcon = { ExposedDropdownMenuDefaults.TrailingIcon(expanded = expanded) },
            colors = ExposedDropdownMenuDefaults.textFieldColors(),
        )
```

Dropdown

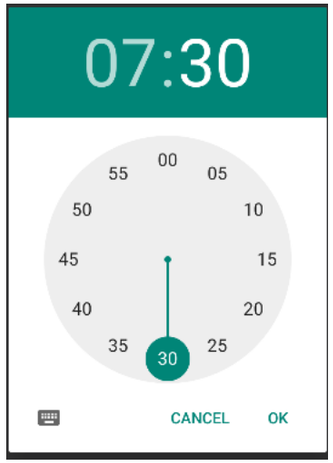
```
            ExposedDropdownMenu(
                expanded = expanded,
                onDismissRequest = { expanded = false },
            ) {
                majorList.forEach { selectionOption ->
                    DropdownMenuItem(
                        text = { Text( text = selectionOption) },
                        onClick = {
                            selectedMajor = selectionOption
                            expanded = false
                            Toast.makeText(context,  text = "Item is $selectionOption",
                                duration = Toast.LENGTH_SHORT).show()
                        },
                        contentPadding = ExposedDropdownMenuDefaults.ItemContentPadding,
                    )
                }
            }
        }
    }
}
```

Pickers

- Android provides controls for the user to pick a time or a date as ready-to-use dialogs.
- Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year).
- Using these pickers helps ensure that users can pick a time or date that is valid, formatted correctly, and adjusted to the user's locale
- **Date Picker** and **Time Picker**

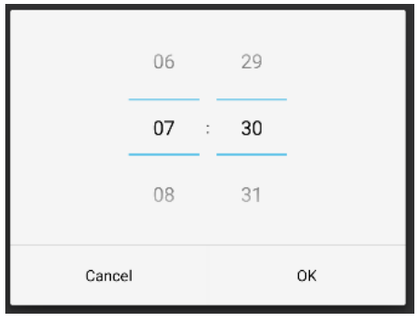


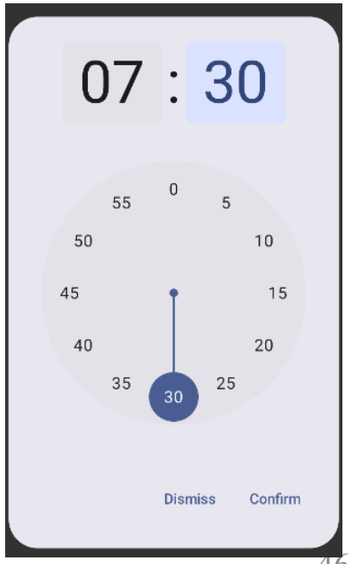
themeResId = 0



themeResId = 1

themeResId = 3





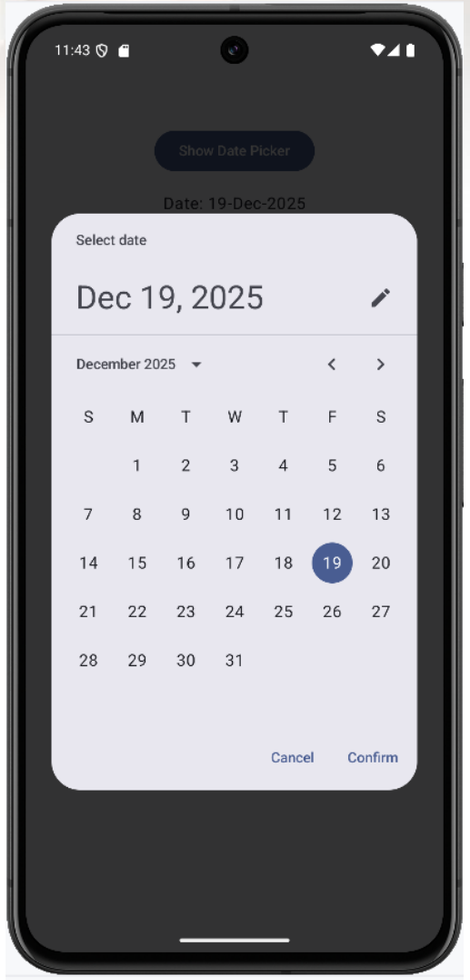
Date Picker

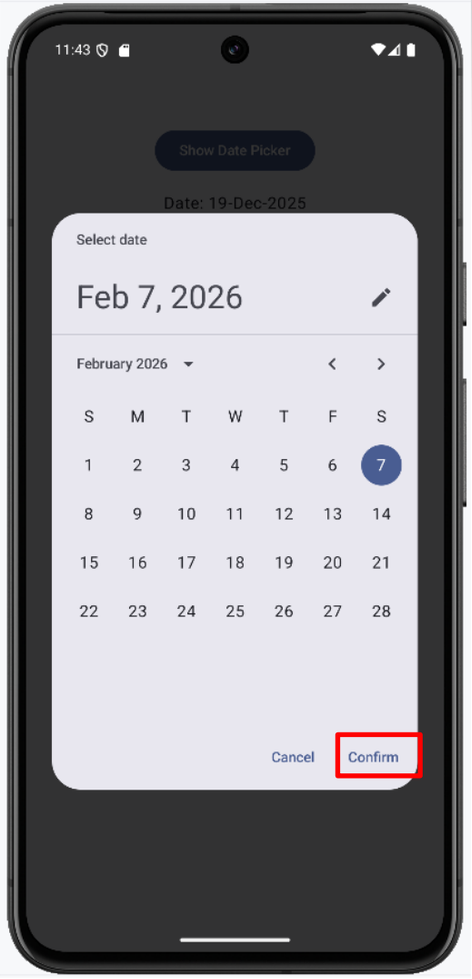
Attributes of the Date Picker

- state – To manage the state of the date picker. rememberDatePickerState()
- modifier – The Modifier to be applied to this date picker.
- dateFormatter – The date format to be used while displaying the date.
- title – The title to be displayed in the date picker.
- headline – The headline to be displayed in the date picker.
- showModeToggle – This is to toggle between the picker and input mode.
- colors – Colors of the date picker at different states.

Date Picker

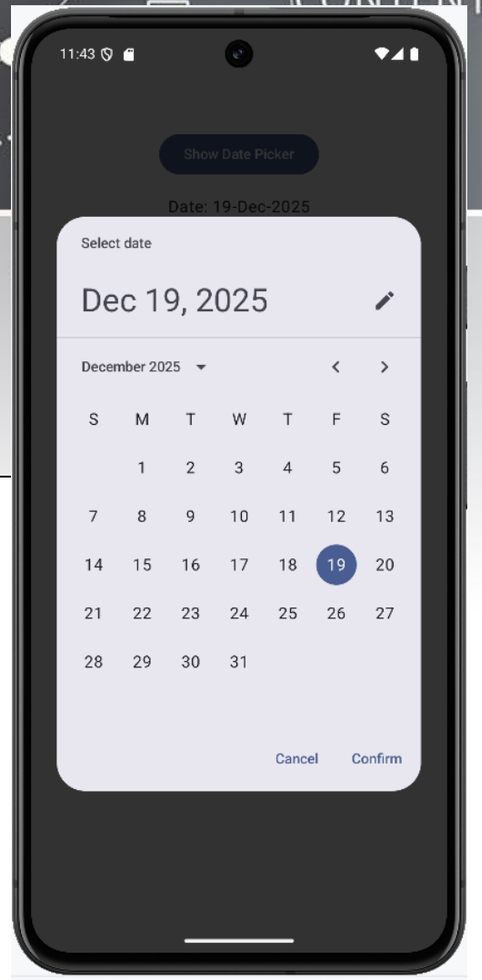

11:42
Show Date Picker
Date: 19-Dec-2025


11:43
Show Date Picker
Date: 19-Dec-2025
Select date
Dec 19, 2025
December 2025
S M T W T F S
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
Cancel
Confirm


11:43
Show Date Picker
Date: 19-Dec-2025
Select date
Feb 7, 2026
February 2026
S M T W T F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
Cancel
Confirm

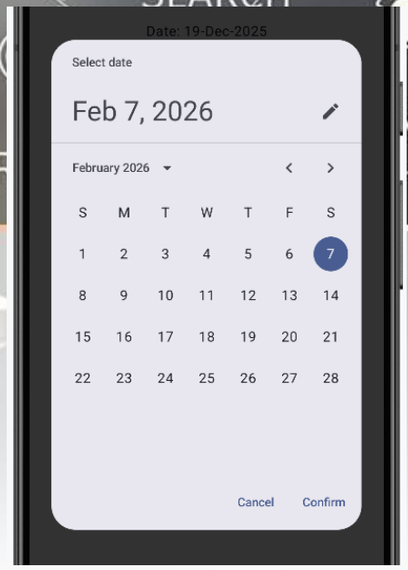

11:43
Show Date Picker
Date: 07-Feb-2026

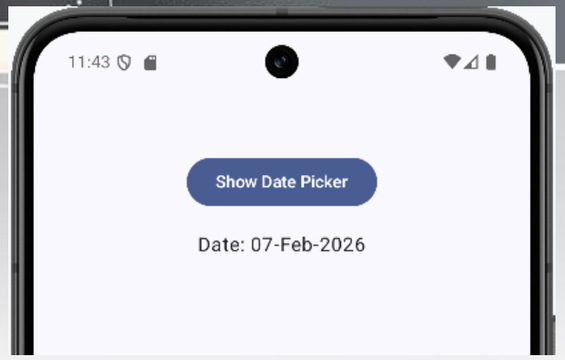
Date Picker




```
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun MyDatePicker(modifier: Modifier = Modifier) {
    // Initialize the start date
    val calendar = Calendar.getInstance()
    val datePickerState = rememberDatePickerState(
        initialSelectedDateMillis = calendar.timeInMillis
    )

    var showDatePicker by rememberSaveable { mutableStateOf( value = false) }
    var selectedDate by rememberSaveable { mutableLongStateOf( value = calendar.timeInMillis) }

    val formatter = rememberSaveable { SimpleDateFormat( pattern = "dd-MMM-yyyy", Locale.getDefault()) }
```

Date Picker






```
if (showDatePicker) {
    DatePickerDialog(
        onDismissRequest = { showDatePicker = false },
        confirmButton = {
            TextButton(onClick = {
                showDatePicker = false
                selectedDate = datePickerState.selectedDateMillis ?: selectedDate
            }) {
                Text(text = "Confirm")
            }
        },
        dismissButton = {
            TextButton(onClick = { showDatePicker = false }) {
                Text(text = "Cancel")
            }
        }
    ) {
        DatePicker(state = datePickerState)
    }
}
```

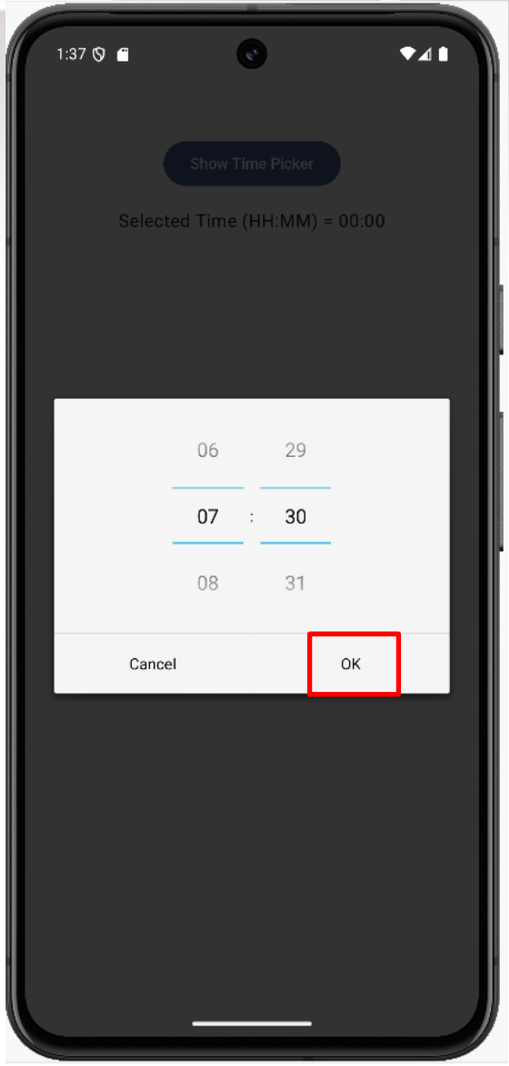
```
    Column(
        modifier = modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Spacer(modifier = Modifier.height( height = 50.dp))

        Button(onClick = { showDatePicker = true }) {
            Text(text = "Show Date Picker")
        }
        Text(
            modifier = Modifier.padding(top = 16.dp),
            text = "Date: ${formatter.format(Date(selectedDate))}"
        )
    }
}
```

Time Picker

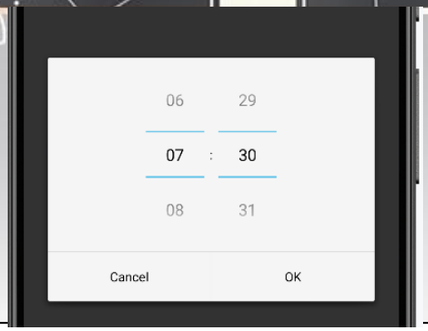
Attributes of the Time Picker :
import android.app.TimePickerDialog

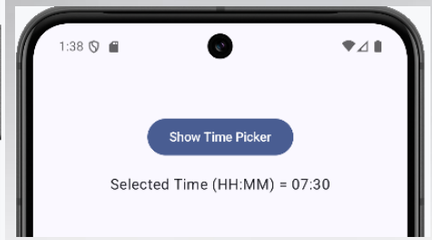
- context - Context: the parent context
- themeResId - int: the resource ID of the theme to apply to this dialog
- listener - TimePickerDialog.OnTimeSetListener: the listener to call when the time is set
- hourOfDay -int: the initial hour
- minute - int: the initial minute
- is24HourView - boolean: Whether this is a 24 hour view, or AM/PM.

Time Picker with Dialog



Time Picker with Dialog





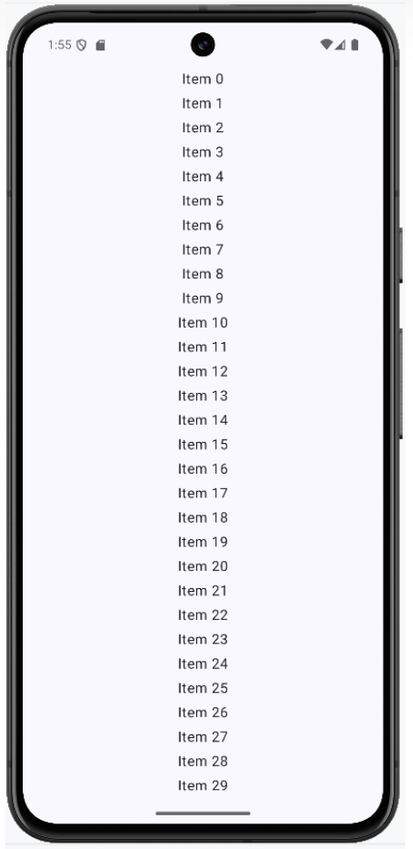

```
@Composable
fun MyTimePicker(modifier: Modifier = Modifier) {
    // Fetching local context
    val mContext = LocalContext.current

    // Declaring and initializing a calendar
    val mCalendar = Calendar.getInstance()
    val mHour = mCalendar[Calendar.HOUR_OF_DAY]
    val mMinute = mCalendar[Calendar.MINUTE]

    // State for selected time variables
    var selectedHour by rememberSaveable { mutableStateOf( value = "00") }
    var selectedMinute by rememberSaveable { mutableStateOf( value = "00") }

    // Creating a TimePicker dialog
    val mTimePickerDialog = TimePickerDialog(
        mContext, themeResId = 3, // 3 is a theme ID (Holo Light usually)
        listener = { _, hour: Int, minute: Int ->
            selectedHour  = if (hour < 10) "0$hour" else "$hour"
            selectedMinute = if (minute < 10) "0$minute" else "$minute"
        }, hourOfDay = mHour, mMinute, is24HourView = true
    )
```

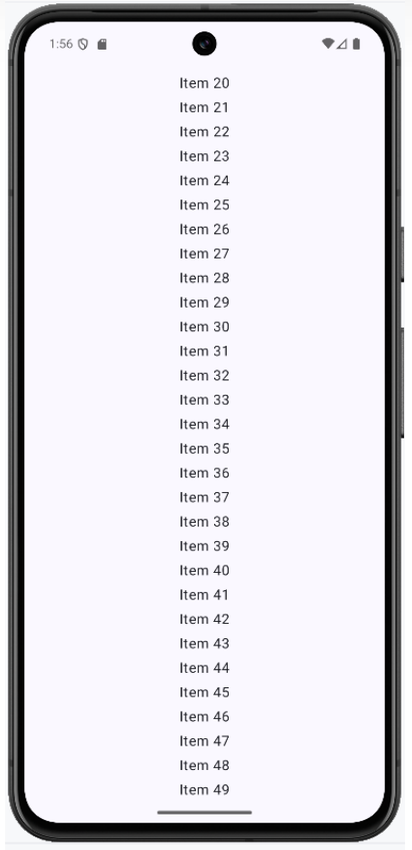
```
    Column(
        modifier = modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Spacer(modifier = Modifier.height( height = 50.dp))

        // On button click, TimePicker is displayed
        Button(
            onClick = { mTimePickerDialog.show() }
        ) {
            Text(text = "Show Time Picker", color = Color.White)
        }

        // Display selected time
        Text(
            modifier = Modifier.padding(top = 16.dp),
            text = "Selected Time (HH:MM) = $selectedHour:$selectedMinute"
        )
    }
}
```

Scroll

- Scroll modifiers: The verticalScroll and horizontalScroll modifiers provide the simplest way to allow the user to scroll an element when the bounds of its contents are larger than its maximum size constraints

```
@Composable
fun ScrollBoxes(modifier: Modifier = Modifier) {
    Column(
        modifier = modifier
            .fillMaxSize()
            .verticalScroll( state = rememberScrollState()),
        horizontalAlignment = Alignment.CenterHorizontally,
    ) {
        repeat( times = 50) {
            Text(
                text = "Item $it",
                modifier = Modifier.padding( all = 2.dp),
            )
        }
    }
}
```





References

- https://www.codeproject.com/Articles/807524/Android-UI-Layouts-and-Controls
- https://semicolonspace.com/jetpack-compose-tutorial/
- https://medium.com/@expertappdevs/text-in-jetpack-compose-android-311cd429bd1e
- https://stackoverflow.com/questions/73248554/how-to-get-value-from-radio-group-in-jetpack-compose
- https://developer.android.com/jetpack/compose/touch-input/pointer-input/scroll