**Welcome to Day 5 of our Android 14 Masterclass**, where we delve into the transformative impact of **Jetpack Compose** on **UI development**, offering an in-depth comparison with XML, **detailed exploration of composables, text handling, and layout management**. By unpacking the core concepts of Jetpack Compose, including composables, text fields, buttons, and context utilization, it equips you with the skills to create responsive, modern, and dynamic Android applications.

## 1. XML vs. Jetpack Compose

**Jetpack Compose** is a modern, fully declarative UI toolkit for building native Android applications. It simplifies and accelerates UI development on Android by using Kotlin programming language features.

- **Declarative UI:** In Jetpack Compose, the UI is defined declaratively. You describe the UI in terms of what it should look like and what it should do, and the framework takes care of rendering it and managing UI changes over time.
- **Kotlin-Based:** Being based on Kotlin, Jetpack Compose allows for powerful, concise, and expressive syntax, making the code more readable and easier to write.

**XML (eXtensible Markup Language)**, on the other hand, has traditionally been used to design UI layouts in Android. XML is a markup language where the UI components and layouts are defined in a structured, hierarchical manner.

- **Separation of Concerns:** XML allows for a clear separation between the UI design and the application logic. The UI is defined in XML files, while the behavior is implemented in Kotlin or Java code.
- **Static Layouts:** XML layouts are more static and require more boilerplate code to manage UI changes and user interactions.

## Understanding UI in the Context of Android Development

**UI**, or **User Interface**, in the context of Android development, refers to the visual elements and interactions that users encounter when using an application. It encompasses all the things a user can interact with – such as screens, pages, buttons, icons, text fields, and other visual elements – to use the application effectively.

- **Visual Components:** UI includes various visual components that make up the application's appearance. These could be buttons, text fields, images, sliders, and other elements that users see and interact with.
- **Layout and Structure:** UI involves organizing the visual components in a way that is user-friendly and intuitive. It includes the arrangement of elements, their size, color, and position, contributing to the overall user experience.
- **Interactivity:** UI also includes the interactive aspects of an application, such as responding to user inputs, navigating between different screens, and providing feedback to user actions. It ensures that the application is responsive and user-friendly.

## 2. Understanding and Creating Composables in Jetpack Compose

**Composables** are at the heart of UI development with Jetpack Compose. They are **functions** that define a part of the UI and can be reused and composed together to build a complete UI.

**What are Composables?**

Composables are Kotlin functions annotated with `@Composable`. They can define anything from a single UI element, like a button or a text field, to a complete screen or a part of it.

Composables can be composed together, meaning you can use one composable inside another to build hierarchical and organized UIs.

## Creating Your Own Composable

Creating your own composable allows for reusable and modular code. Here's a simple guide on how to do it:

- **Define a Function:** Start by defining a Kotlin function. The function should be annotated with `@Composable`.
- **Add UI Elements:** Inside the function, you can add predefined composable functions like `Text()`, `Button()`, etc., or other custom composables.
- **Parameters for Customization:** Your composable function can take parameters, allowing it to be customized when called.
- **Compose Together:** You can use your custom composable inside other composables, allowing for code reuse and modular design.

## Example

Here's a simple example of creating a custom composable:

```kotlin
@Composable
fun CustomText(text: String) {
    Text(text = text, style = TextStyle(fontSize = 20.sp, color = Color.Red))
}

// Using the CustomText composable
@Composable
fun MyApp() {
    Column {
        CustomText(text = "Hello, World!")
        CustomText(text = "Welcome to Jetpack Compose!")
    }
}

```

In this example, `CustomText` is a custom composable that displays a text string with specific styling. It is then used inside another composable, `MyApp`, demonstrating the composability and reuse of custom composables.

## Jetpack Compose: Columns

Columns let you place items (like text or buttons) vertically, one below the other.

**Example:**

```
@Composable
fun MyColumn() {
    Column {
        Text("First item")
        Text("Second item")
        Text("Third item")
    }
}

```

In this code, `MyColumn` is a box that holds texts. The texts are shown one below the other.

## Modifier Property for Column

Modifiers are like magical tools that help you customize the appearance and behavior of elements in your app, such as columns, rows, and buttons.

**Example**:

Imagine you have a column (a vertical box), and you want it to be as big as the screen and have a blue background. A modifier can do that!

- **.fillMaxSize**

`.fillMaxSize` is a specific modifier that makes an element (like a column or row) take up as much space as it can.

If you use `.fillMaxSize` on a column, the column will stretch to fill the entire screen.

- **Vertical Arrangement**

Vertical arrangement is about deciding how the items inside a column are spaced and aligned vertically (up and down).

**Example**:

You can choose whether the items are close together, spaced out, or centered in the column.

- **Horizontal Alignment**

Horizontal alignment is about deciding how the items inside a column are aligned horizontally (left and right).

**Example**:

You can choose whether the items are aligned to the left, right, or centered in the column.

**Putting It All Together: Code Example**

```
@Composable
fun CustomColumn() {
    Column(
        modifier = Modifier.fillMaxSize(),
// Makes the column take up the whole screen
        verticalArrangement = Arrangement.Center, // Centers items vertically
        horizontalAlignment = Alignment.CenterHorizontally
// Centers items horizontally
     ) {
        Text("Item 1")
        Text("Item 2")
        Text("Item 3")
     }
}

```

**In Simple Terms**

- **Modifier:** A tool to style and arrange elements.
- **.fillMaxSize:** A command that says, “Be as big as you can!”
- **Vertical Arrangement:** Decides the up-and-down position of items.
- **Horizontal Alignment:** Decides the left-and-right position of items.

Imagine arranging pictures (items) on a wall (column). You decide how high or low each

Imagine arranging pictures (items) on a wall (column). You decide how high or low each picture is (vertical arrangement), and whether they are in the center or towards the sides of the wall (horizontal alignment). Modifiers are like the tools and rules you use to arrange the pictures perfectly.

### Jetpack Compose: Rows

Rows let you place items horizontally, side by side.

**Example:**

```
@Composable
fun MyRow() {
    Row {
        Text("First item")
        Text("Second item")
        Text("Third item")
    }
}

```

Here, `MyRow` is a box that holds texts. The texts are shown side by side, in a line.

## 3. Understanding Text Composables in Jetpack Compose

Text composables are parts of the user interface where users can enter or display text. Let’s break down the different types of text composables:

### Jetpack Compose: TextField

`TextField` is like a text box where users can type things. It comes with a nice design and some features like a label.

**Example:**

```kotlin
@Composable
fun MyTextField() {
    TextField(
        value = "",
        onValueChange = {},
        label = { Text("Enter your name") }
    )
}

```

- This creates a text box with a label “Enter your name.”
- Users can type in it, but the text won’t be remembered or used for now.

## Jetpack Compose: BasicTextField

`BasicTextField` is a simpler text box without any extra design. It’s just a place to type text.

```kotlin
@Composable
fun MyBasicTextField() {
    BasicTextField(
        value = "",
        onValueChange = {}
    )
}

```

- This creates a simple text box without any label or extra design.
- Users can type in it, but the text won’t be remembered or used for now.

## Jetpack Compose: OutlinedTextField

`OutlinedTextField` is a text box with an outline around it. It also can have a label.

```kotlin
@Composable
fun MyOutlinedTextField() {
    OutlinedTextField(
        value = "",
        onValueChange = {},
        label = { Text("Enter your email") }
    )
}

```

- This creates a text box with an outline and a label “Enter your email.”
- Users can type in it, but the text won’t be remembered or used for now.

## What is `onValueChange`?

**`onValueChange`** is like a helper that listens when you type or delete something in a text box. Every time you press a key, **`onValueChange`** notices and can do something with the new text.

**Imagine you are typing in a text box**:

- You type the letter “a”.
- **`onValueChange`** sees the “a” and says, “Okay, the text is now ‘a’.”

If you type another letter, like “b”, **`onValueChange`** updates the text:

- **`onValueChange`** sees “ab” and says, “Okay, the text is now ‘ab’.”

## Simple Example

Let’s say we have a text box, and we want to show the typed text somewhere else. We could use **`onValueChange`** to keep the text updated.

```kotlin
@Composable
fun SimpleExample() {
    var typedText by remember { mutableStateOf("Type something...") }

    TextField(
        value = typedText,
        onValueChange = { newText -&gt; typedText = newText }
    )

    Text("You typed: $typedText")
}

```

**Putting It All Together**

- `var typedText by remember { mutableStateOf("Type something...") }`
  - This line creates a variable `typedText`.
  - It starts with the value `"Type something..."`.
  - The value can change, and the app will remember the changes.
- `onValueChange = { newText -> typedText = newText }`: This part means, "Whenever new text is typed, update `typedText` with the new text."
  - `onValueChange` listens and updates the text.
  - `{ newText -> typedText = newText }`: This is the task given to `onValueChange`. It means, "Take the new text (`newText`) and make `typedText` the same as `newText`."
- `Text("You typed: $typedText")`: This part shows the text that was typed.

## 4. What is the `Preview` Composable?

The `Preview` composable is like a magical mirror. It lets you see what your app's screen or a part of it (like a button or a text box) will look like without running the whole app on a phone or emulator.

## How Does It Work?

When you create a part of your app's screen (a composable), you can put `@Preview` above it. This tells Android Studio to show you a preview of that part right inside the editor.

## Example

Imagine you made a button for your app:

```
@Composable
fun MyButton() {
    Button(onClick = {}) {
        Text("Click me")
    }
}

```

Now, you want to see what it looks like. You can use `@Preview` like this:

```
@Preview
@Composable
fun PreviewMyButton() {
    MyButton()
}

```

## What Happens Next?

- In Android Studio, you'll see a picture of your button.
- You can look at it, make sure it's nice, and make changes if you want, all without running the whole app.

## 5. What is the `Button` Composable?

A **`Button`** composable is like a clickable sign in your app. It's something users can tap or click on to do an action, like submitting a form or going to another screen.

### How to Create a Button?

Creating a button is like making a sign. You decide what the sign says and what happens when it's clicked.

### Example

Let's make a simple button that says "Click me":

```
@Composable
fun MyButton() {
    Button(onClick = {
        println("Button was clicked!")
    }) {
        Text("Click me")
    }
}

```

### Breaking Down the Example

- **`Button(onClick = { println("Button was clicked!") })`**: This part makes the button and decides what happens when it's clicked. In this case, it will show a message "Button was clicked!".
- **`Text("Click me")`**: This part puts text on the button. The button will say "Click me".

## 6. What is `onClick`?

**`onClick`** is like a magic word that makes something happen when you click a button or some other part of the app. It's like a command that says, "Do this when the button is clicked!"

### How Does `onClick` Work?

When you create a button, you can give it an `onClick` command. This command tells the button what to do when it's clicked.

**Example**

Imagine you have a button, and you want it to show a message when it's clicked:

```
Button(onClick = {
    println("Hello, you clicked me!")
}) {
    Text("Click me")
}

```

**Breaking Down the Example**

- `Button(onClick = { println("Hello, you clicked me!") })`: This part creates a button and gives it a command. The command is `{ println("Hello, you clicked me!") }`.
- `Text("Click me")`: This part puts words on the button, so it says "Click me".

# 7. What is Context?

In Android, `Context` is like a bridge that connects different parts of an app to the system. It's like a messenger that helps parts of your app talk to the operating system and access resources, services, and other stuff.

**How is Context Used?**

- **Accessing Resources:** `Context` helps your app find and use resources like images, colors, and strings.
- **Launching Activities:** `Context` can help start new screens (activities) in your app.
- **Showing Toast Messages:** `Context` is used to show brief messages (toasts) on the screen.

## Setting Up Context in a Composable

In Jetpack Compose, you can get **`Context`** directly within a composable function using the **`LocalContext`** provider.

**Example**

Here's how you might use **`Context`** to show a toast message:

```
@Composable
fun ShowToastButton() {
    val context = LocalContext.current

    Button(onClick = {
        Toast.makeText(context, "Button clicked!", Toast.LENGTH_SHORT).show()
    }) {
        Text("Click me")
    }
}

```

**Breaking Down the Example**

- **`val context = LocalContext.current`**: This line gets the current **`Context`** and keeps it ready for use.
- **`Toast.makeText(context, "Button clicked!", Toast.LENGTH_SHORT).show()`**: This line creates and shows a toast message. It uses **`Context`** to know where to show the message.

## 8. What is the `Toast` Class?

A **`Toast`** in Android is like a quick message that pops up on the screen. It's a way to give users a short message or feedback, like "Message sent" or "Error occurred."

## How to Create and Show a Toast?

The `Toast` class has special commands (methods) to create and show these messages. Let's look at them:

## 1. `Toast.makeText`

- This command creates the toast. It prepares the message you want to show.
- Example: `Toast.makeText(context, "Hello!", Toast.LENGTH_SHORT)`

## 2. `.length_long` and `.length_short`

- These options decide how long the toast message will stay on the screen.
- `.length_long` : The message stays longer.
- `.length_short` : The message stays for a shorter time.

## 3. `.show()`

- This command tells the toast to appear on the screen.
- Example: `toast.show()`

**Full Example**

Here's how you might use these commands together:

```
val toast = Toast.makeText(context, "Button clicked!", Toast.LENGTH_SHORT)
toast.show()

```

**Breaking Down the Example**

- `val toast = Toast.makeText(context, "Button clicked!", Toast.LENGTH_SHORT)` : This line creates the toast. It says the message will be "Button clicked!" and it will stay for a short time.
- `toast.show()` : This line makes the toast appear on the screen.

## 9. What is the `Box` Composable?

A **Box** composable in Jetpack Compose is like a container or a box where you can put other things, like text, buttons, or images. It allows you to stack these things on top of each other, creating layers.

### Jetpack Compose: How to Use the `Box` Composable?

You use a **Box** when you want to put one thing over another, like putting text over an image.

### Example

Let's say you want to create a button with an icon and text on it:

```
@Composable
fun IconButton() {
    Box(
        contentAlignment = Alignment.Center,
        modifier = Modifier.size(100.dp)
    ) {
        Icon(imageVector = Icons.Default.Star, contentDescription = null)
        Text("Star")
    }
}

```

### Breaking Down the Example

- **`Box(...) { ... }`**: This part creates the box.
- **`contentAlignment = Alignment.Center`**: This part makes sure everything inside the box is centered.
- **`Icon(...)`** and **`Text("Star")`**: These parts put an icon and text inside the box.

They will appear on top of each other, both centered.

## 10. What is the `Icon` Composable?

An **`Icon`** composable in Jetpack Compose is like a small picture or symbol that you can use in your app. Icons are used to represent actions, content, or to give users a visual cue.

### Jetpack Compose: How to Use the `Icon` Composable?

You use an **`Icon`** to add these small pictures or symbols to buttons, menus, or other parts of your app.

### Example

Let’s say you want to create a star icon:

```
@Composable
fun StarIcon() {
    Icon(
        imageVector = Icons.Default.Star,
        contentDescription = "Star Icon"
    )
}

```

### Breaking Down the Example

- **`Icon(...)`**: This part creates the icon.
- **`imageVector = Icons.Default.Star`**: This part chooses the picture or symbol for the icon. In this case, it’s a star.
- **`contentDescription = "Star Icon"`**: This part gives the icon a description. It’s like a label that tells what the icon represents.

## 11. What is a Dropdown Menu?

A dropdown menu is like a secret list that appears when you click on something, like a button or an arrow. It shows you more options to choose from.

### How to Create a Dropdown Menu?

1. **Button or Arrow:** First, you create something to click on, like a button or an arrow.
2. **Menu:** Then, you create the secret list (menu) that will appear when the button or arrow is clicked.

### Example: Dropdown Menu with an Arrow

Let's create a dropdown menu that appears when you click on an arrow.

```
@Composable
fun DropdownMenuExample() {
    var expanded by remember { mutableStateOf(false) }

    Box {
        IconButton(onClick = { expanded = !expanded }) {
            Icon(imageVector = Icons.Default.ArrowDropDown,
            contentDescription = null)
        }

        DropdownMenu(expanded = expanded,
            onDismissRequest = { expanded = false }) {
            DropdownMenuItem(onClick = { /* Do something when clicked */ }) {
                Text("Option 1")
            }
            DropdownMenuItem(onClick = { /* Do something when clicked */ }) {
                Text("Option 2")
            }
        }
    }
}
```

### Breaking Down the Example

- `IconButton(onClick = { expanded = !expanded }) { ... }`: This part creates an arrow that you can click. When clicked, it will show or hide the menu.
- `Icon(imageVector = Icons.Default.ArrowDropDown, contentDescription = null)`: This part chooses an arrow icon.
- `DropdownMenu(expanded = expanded, onDismissRequest = { expanded = false }) { ... }`: This part creates the secret list (menu). It will appear or disappear when the arrow is clicked.
- `DropdownMenuItem(onClick = { /* Do something when clicked */ }) { ... }`: These parts are the options in the menu. You can click on them to do something.

## 12. What are Parent Containers?

Parent containers in app development are like big boxes that hold and organize smaller boxes (components) inside them. These big boxes help to arrange, align, and manage the smaller parts of your app, like buttons, text, images, and other containers.

### Why Use Parent Containers?

- **Organization:** They help keep the app tidy by holding related items together.
- **Layout:** They help in arranging the parts of your app nicely on the screen.
- **Control:** They allow you to manage and adjust the properties of multiple items at once.

### Examples of Parent Containers

- **Column:** A vertical container that arranges items from top to bottom.
- **Row:** A horizontal container that arranges items from left to right.
- **Box:** A container that can layer items on top of each other.

### Simple Example: Using a Column as a Parent Container

Imagine you want to create a screen with a picture at the top and a button below it.

```
@Composable
fun SimpleScreen() {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier.fillMaxSize()
    ) {
        Image(painter = painterResource(id = R.drawable.ic_launcher_foreground),
        contentDescription = null)
        Button(onClick = { /* Do something */ }) {
            Text("Click me")
        }
    }
}

```

### Breaking Down the Example

- `Column(...) { ... }`: This is the big box (parent container). It will hold and arrange the items inside it vertically.
- `Image(...)`: This is a smaller box (component) inside the Column. It's like a picture inside the big box.
- `Button(...) { ... }`: This is another smaller box inside the Column. It's like a button inside the big box.

## 13. Space vs. Padding

- **Space:** Space refers to the empty area between different elements in your app, like buttons or texts. It helps avoid clutter and makes the app look neat.
- **Padding:** Padding is the space inside the borders of an element. It creates some room between the content of an element (like text inside a button) and its edges.

## Padding Modifier

A padding modifier is a tool that adds padding to an element. It helps in creating space around the content inside an element.

**Example**:

```
Text("Hello", modifier = Modifier.padding(16.dp))

```

This will add space around the text “Hello,” pushing it away from the edges of its container.

## Spacer Elements

A **`Spacer`** is a composable that just adds empty space. It’s like an invisible box that you can put between other elements to create distance between them.

**Example**:

```
Spacer(modifier = Modifier.height(16.dp))

```

This will add vertical space of 16 density-independent pixels.

## Jetpack Compose: Spacer Composable

The **`Spacer`** composable is used to create space in your layout. It’s like an invisible helper that makes sure elements in your app are not too close together.

**Example**:

```
Column {
    Text("Item 1")
    Spacer(modifier = Modifier.height(20.dp))
    Text("Item 2")
}

```

This will create space between “Item 1” and “Item 2.”

## In Simple Terms

- **Padding Modifier:** Imagine putting a picture in a frame, and the padding is like the border of the frame that surrounds the picture.
- **Spacer Elements:** Think of spacers as invisible cushions that you put between items to keep them apart.
- **Spacer Composable:** It’s like an invisible shelf that you add between items to give them more room.

Using padding and spacers helps make the app look organized and easy to read, like arranging furniture in a room to make it look tidy and comfortable.

## Conclusion: The Power of Jetpack Compose and UI Customization – Day 5 Android 14 Masterclass

As we wrap up Day 5, it’s evident that Jetpack Compose is set to redefine Android app development. With its Kotlin-based, declarative approach, this modern toolkit not only streamlines the UI design process but also enhances the overall developer experience. From creating intuitive layouts with Rows and Columns to implementing interactive Text composables and understanding the crucial role of Context, this article has equipped you with the knowledge to excel in UI customization.

If you want to skyrocket your Android career, check out our **The Complete Android 14 & Kotlin Development Masterclass.** Learn Android 14 App Development From Beginner to Advanced Developer.

**Master Jetpack Compose** to harness the cutting-edge of Android development, **gain proficiency in XML** — an essential skill for numerous development roles, and **acquire**