**강의1 : 구성 가능한 함수**

**Jetpack Compose는 구성 가능한 함수를 중심으로 빌드 되었다.**

**구성 가능한 함수를 만들려면 함수 이름에 @Composable 주석을 추가하기만 하면 된다.**

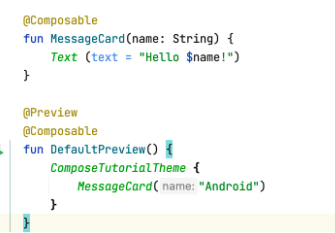
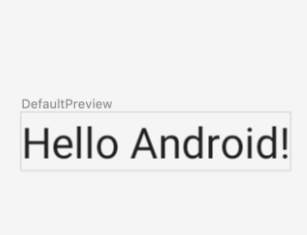
class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *setContent* **{** *Text*("Hello world!")  
 **}** }  
}

**setContent 안에 XML 대신 레이아웃을 구성할 Composable 함수가 들어갈 수 있다.**

*setContent* **{** *MessageCard*("Android")  
 **}** }  
}  
  
@Composable  
fun MessageCard(name: String) {  
 *Text*(text = "Hello $name!")  
}

**함수를 구성 가능하게 하려면 @Composable 주석을 위와 같이 추가해야 한다. 이렇게 하면 이름을 전달받는 MessageCard() 함수를 정의하고 이를 사용하여 텍스트 요소를 구성한다.**

**Android Studio를 사용하면 XML 같은 경우에는 바로 변화하는 것을 볼 수 있다. Jetpack Compose에서도 @Preview를 @Composable앞에 추가함으로써 미리보기를 보여준다.**

** **

**강의 2 : 레이아웃**

**UI요소는 계층적이며 다른 요소에 포함된 요소가 있다. Compose에서는 다른 구성 가능한 함수로부터 구성 가능한 함수를 호출하여 UI계층 구조를 빌드한다.**

**EX) 여러 텍스트 추가**

class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *setContent* **{** *MessageCard*(Message("Android", "Jetpack Compose"))  
 **}** }  
}  
  
data class Message(val author: String, val body: String)  
  
@Composable  
fun MessageCard(msg: Message) {  
 *Text*(text = msg.author)  
 *Text*(text = msg.body)  
}  
  
@Preview  
@Composable  
fun PreviewMessageCard() {  
 *MessageCard*(  
 msg = Message("Colleague", "Hey, take a look at Jetpack Compose, it's great!")  
 )  
}

**PreviewMessageCard로는 Colleague와 Hey, take a ~~ 가 겹처서 나타나고 실제로는 Android와 Jetpack Compose가 겹처서 보인다. 왜냐하면 정렬 방법에 관한 정보가 없어서 겹치게 나타난다.**

**EX) 열 사용**

**Column 함수를 사용하면 요소를 수직으로 정렬할 수 있다.**

@Composable  
fun MessageCard(msg: Message) {  
 *Column* **{** *Text*(text = msg.author)  
 *Text*(text = msg.body)  
 **}**}

**이렇게 하면 msg.author에 있는 Android가 먼저 출력되고 그 다음에 msg.body에 있는 Jetpack Compose가 순서대로 수직적으로 나타난다.**

**행을 Row를 이용하면 수평으로 정렬할 수 있고, Box를 이용하면 요소를 쌓을 수 있다.**

**EX) 이미지 요소 추가**

**Resource Manager를 사용하여 사진 라이브러리에서 이미지를 가져온다.**

@Composable  
fun MessageCard(msg: Message) {  
 *Row* **{** *Image*(  
 painter = *painterResource*(R.drawable.*img*),  
 contentDescription = "Contact profile picture",  
 )  
 *Text*(text = msg.author)  
 *Text*(text = msg.body)  
 **}**}

**메시지 레이아웃의 구조가 올바르지만 요소의 간격이 균등하지 않고 이미지가 너무 크게 나타난다.**

**EX) 레이아웃 구성**

@Composable  
fun MessageCard(msg: Message) {  
 // Padding을 추가하여 8dp 만큼에 공간을 띄워준다.  
 *Row*(modifier = Modifier.*padding*(all = 8.*dp*)) **{** *Image*(  
 painter = *painterResource*(R.drawable.*img*),  
 contentDescription = "Contact profile picture",  
 modifier = Modifier  
 // 이미지 size를 40dp로 설정한다.  
 .*size*(40.*dp*)  
 // 이미지 클립을 동그란 모양으로 설정한다.  
 .*clip*(*CircleShape*)  
 )  
  
 // 이미지와 글씨 사이의 수평적 공간을 8dp 만큼 띄워준다.  
 *Spacer*(modifier = Modifier.*width*(8.*dp*))  
  
 *Column* **{** *Text*(text = msg.author)  
 // msg.author와 msg.body 사이의 세로 길이를 4dp 만큼 띄워준다.  
 *Spacer*(modifier = Modifier.*height*(4.*dp*))  
 *Text*(text = msg.body)  
 **}  
 }**}

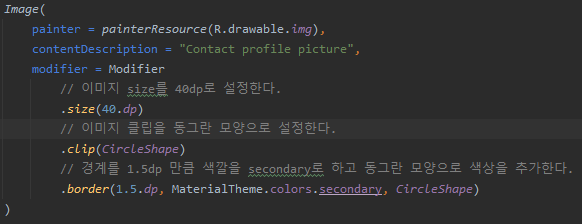
**강의 3 : Material Design**

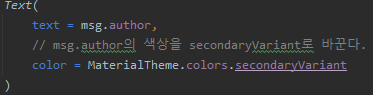
**Compose는 Material Design원칙을 지원하도록 빌드 되었다. 많은 UI 요소가 Material Design을 즉시 사용 가능하도록 구현한다. 이 강의에서는 Material Widget으로 앱의 스타일을 지정한다.**

**EX) Material Design 사용**

**프로젝트에서 생성한 Material theme(BasicCodeLab\_BBangChanTheme로 생성되었음.)로 MessageCard 함수를 감싼다. @Preview 및 setContent 함수 모두에서 이 작업을 실행한다. Material Design은 색상, 글씨체, 도형 세 가지 핵심 요소에 따라 이루어집니다.**

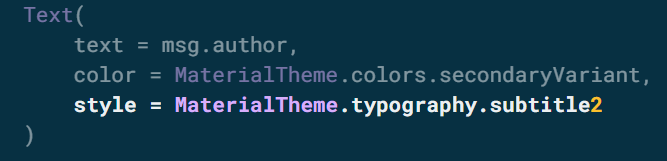
**EX) 색상 추가**

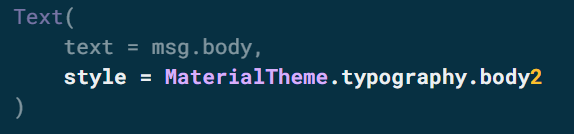
****

****

**EX) 글씨체 변경**

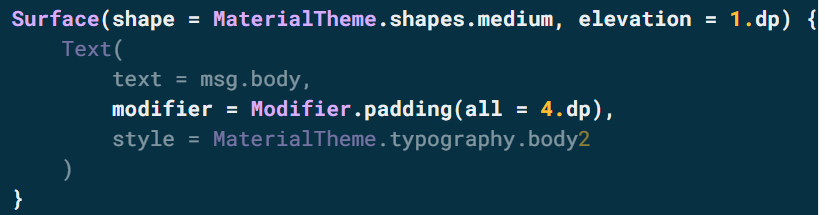
**Material 글씨체 Style은 MaterialTheme에서 사용할 수 있으며 Text Composable에 추가하기만 하면된다.**

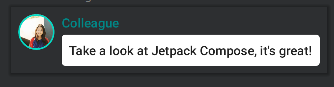
****

****

**EX) 도형**

**도형을 사용해서 최종 터치를 추가할 수 있습니다. 더 좋은 레이아웃을 위해 메시지에 패딩을 추가하기도 한다.**

****

****

**Padding을 4만큼 주어서 메시지에 도형에 4dp만큼에 공간이 생김.**

**EX) 어두운 테마 사용**

**Material Design 지원 덕분에 Jetpack Compose는 기본적으로 어두운 테마를 처리할 수 있다. Material 색상,텍스트,배경을 사용하면 어두운 배경에 맞춰 자동으로 조정이 된다.**

@Preview(name = "Light Mode")  
@Preview(  
 uiMode = Configuration.*UI\_MODE\_NIGHT\_YES*,  
 showBackground = true,  
 name = "Dark Mode"  
)

**@Preview로 Light Mode를 보고 그 다음 Dark Mode를 설정해서 Dark Mode도 확인한다.**

**강의 4 : 목록 및 애니메이션**

**목록 및 애니메이션은 앱의 모든 곳에 존재한다. 이 강의에서는 Compose를 사용하여 손쉽게 목록을 만들고 애니메이션을 이용해서 재미를 더하는 방법을 배운다.**

**EX) 메시지 목록 만들기**

import androidx.compose.foundation.lazy.items

@Composable  
fun Conversation(messages: List<Message>) {  
 *LazyColumn* **{** *items*(messages) **{** message **->** *MessageCard*(message)  
 **}  
 }**}  
  
@Preview  
@Composable  
fun PreviewConversation() {  
 *BasicCodeLab\_BBangChanTheme***{** *Conversation*(SampleData.conversationSample)  
 **}**}

**SampleData 파일에 있는 conversationSample에 배열을 Conversation의 인자로 불러온다.**

**EX) 확장 중 메시지에 애니메이션 적용**

**메시지를 확장하여 더 길게 보여주고 콘텐츠 크기와 배경 색상 모두에 애니메이션 효과를 적용하는 기능을 추가할 예정입니다. 이 로컬UI 상태를 저장하려면 메시지가 확장되었는지 여부를 추적해야 한다.이 상태 변경을 추적하려면 remember 및 mutableStateOf 함수를 사용해야 한다. Remember를 사용하여 메모리에 로컬 상태를 저장하고 mutableStateOf에 전달된 값의 변경사항을 추적할 수 있습니다. 이 상태를 사용하는 Composable 및 하위 요소는 값이 업데이트되면 자동으로 다시 그려진다. 이를 재구성이라고 한다.**

**Remember 및 mutableStateOf와 같은 Compose의 상태 API를 사용하여 상태를 변경하면 UI가 자동으로 업데이트됩니다.**

**참고 : ‘by’를 올바르게 사용하려면 다음 가져오기를 추가해야 한다. ALT+Enter를 누르면 자동으로 추가된다.**

class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *setContent* **{** *BasicCodeLab\_BBangChanTheme***{** *Conversation*(SampleData.conversationSample)  
 **}  
 }** }  
}  
  
  
@Composable  
fun MessageCard(msg: Message) {  
 *Row*(modifier = Modifier.*padding*(all = 8.*dp*)) **{** *Image*(  
 painter = *painterResource*(R.drawable.*img*),  
 contentDescription = null,  
 modifier = Modifier  
 .*size*(40.*dp*)  
 .*clip*(*CircleShape*)  
 .*border*(1.5.*dp*, MaterialTheme.colors.secondaryVariant, *CircleShape*)  
 )  
 *Spacer*(modifier = Modifier.*width*(8.*dp*))  
  
 // 우리는 메시지가 변화되었는지를 확인한다.  
 // 변수  
 var isExpanded by *remember* **{** *mutableStateOf*(false) **}** // 우리가 Column을 클릭할 때 isExpanded 변수를 전환한다.  
 *Column*(modifier = Modifier.*clickable* **{** isExpanded = !isExpanded **}**) **{** *Text*(  
 text = msg.author,  
 color = MaterialTheme.colors.secondaryVariant,  
 style = MaterialTheme.typography.subtitle2  
 )  
  
 *Spacer*(modifier = Modifier.*height*(4.*dp*))  
  
 *Surface*(  
 shape = MaterialTheme.shapes.medium,  
 elevation = 1.*dp*,  
 ) **{** *Text*(  
 text = msg.body,  
 modifier = Modifier.*padding*(all = 4.*dp*),  
 // 메시지가 확장되면 그것의 전부를 보여준다.  
 // 그렇지 않으면 우리는 첫번째 줄만 보여준다.  
 maxLines = if (isExpanded) Int.MAX\_VALUE else 1,  
 style = MaterialTheme.typography.body2  
 )  
 **}  
 }  
 }**}  
data class Message(val author: String, val body: String)  
  
@Composable  
fun Conversation(messages: List<Message>) {  
 *LazyColumn* **{** *items*(messages) **{** message **->** *MessageCard*(message)  
 **}  
 }**}

**EX) 메시지 클릭 시 Content의 배경 변경**

// surfaceColor는 점차적으로 다른 것들중 하나로 점차적으로 업데이트 될것이다.  
val surfaceColor : Color by *animateColorAsState*(  
 if (isExpanded) MaterialTheme.colors.primary else MaterialTheme.colors.surface,  
)

// surface의 컬러가 primary color로 바뀌게된다.  
color = surfaceColor,  
// 애니메이션 Content size에 padding을 1.dp 만큼 띄워준다.  
modifier = Modifier.*animateContentSize*().*padding*(1.*dp*)

**원래는 XML에 UI를 선언해야 Android Views를 사용할 수 있었다. 데이터가 바뀌면 UI도 업데이트 해야 했다. 이를 위해서는 View를 조회하고 속성을 설정해야 한다. 어플리케이션의 상태가 바뀔 때마다(데이터베이스, 네트워크 호출이 로드되거나, 사용자 상호작용이 끝나면) 새로운 정보로 UI를 업데이트해서 데이터를 동기화해야 한다. 그 과정이 복잡할 수도 있다. 모델과 UI를 동기화하는 작업에서 버그가 발생할 수 있다. 개발자가 책임지고 모든 걸 업데이트해야 한다.**

**앱과 UI가 복잡해지면서 오류가 생기기는 쉬워졌습니다.**

**Composable은 데이터를 함수 매개변수로 받아서 UI를 전달합니다.**