



PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 9

- Jetpack Compose
- Podstawy tworzenia UI
- Stan
- Kompozycja i Rekompozycja

Column oraz **Row** w **Compose** są to komponenty layoutu, które są odpowiedzialne za układanie elementów interfejsu użytkownika w kolumnie lub rzędzie.

Column, można umieścić wiele elementów interfejsu użytkownika **jeden pod drugim** w **pionowej kolumnie**.

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Podstawowe komponenty

Column oraz **Row** w **Compose** są to komponenty layoutu, które są odpowiedzialne za układanie elementów interfejsu użytkownika w kolumnie lub rzędzie.

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Column, można umieścić wiele elementów interfejsu użytkownika **jeden pod drugim** w **pionowej kolumnie**.

Podstawowy komponent do wyświetlania tekstu.

Wyświetlany tekst.

Rozmiar fontu.

Podstawowe komponenty

Column oraz **Row** w **Compose** są to komponenty layoutu, które są odpowiedzialne za układanie elementów interfejsu użytkownika w kolumnie lub rzędzie.

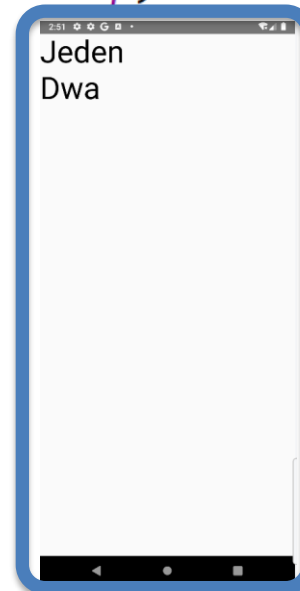
```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Column, można umieścić wiele elementów interfejsu użytkownika **jeden pod drugim** w **pionowej kolumnie**.

Podstawowy komponent do wyświetlania tekstu.

Wyświetlany tekst.

Rozmiar fontu.



Podstawowe komponenty

Column oraz **Row** w **Compose** są to komponenty layoutu, które są odpowiedzialne za układanie elementów interfejsu użytkownika w kolumnie lub rzędzie.

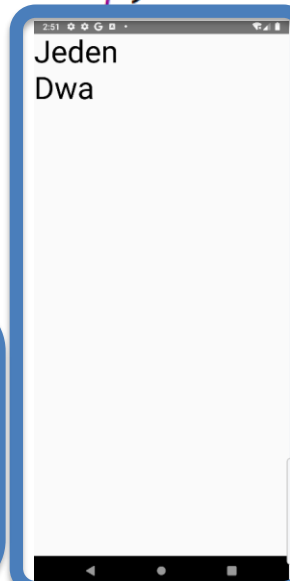
```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Column, można umieścić wiele elementów interfejsu użytkownika **jeden pod drugim** w **pionowej kolumnie**.

Podstawowy komponent do wyświetlania tekstu.

Rozmiar fontu.

Wyświetlany tekst.



dp to **jednostka miary** używana w Androidzie do określania wymiarów interfejsu użytkownika (np. wysokości, szerokości, marginesów). Jest **niezależna od fizycznej rozdzielczości ekranu**, co oznacza, że interfejs wygląda spójnie na różnych urządzeniach o **różnych gęstościach pikseli** (DPI - dots per inch). **sp** dodatkowo uwzględnia **skalowanie czcionki** ustawione w systemie.

Podstawowe komponenty

Column oraz **Row** w **Compose** są to komponenty layoutu, które są odpowiedzialne za układanie elementów interfejsu użytkownika w kolumnie lub rzędzie.

Row, można umieścić wiele elementów interfejsu użytkownika **jeden obok drugiego** w rzędzie.

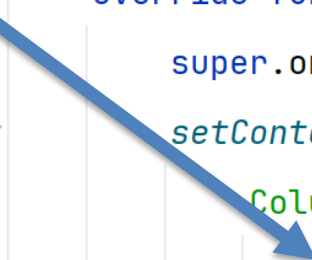
```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Row {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```



Modifier w **Compose** to obiekt, który służy do modyfikacji wyglądu lub zachowania komponentu.

Modifier działa na zasadzie łańcucha funkcji - każda kolejna funkcja modyfikuje obiekt wynikowy poprzedniej funkcji.

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxWidth()  
                    .background(Color.Cyan)  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```



Modyfikator

Modifier w **Compose** to obiekt, który służy do modyfikacji wyglądu lub zachowania komponentu.

Modifier działa na zasadzie łańcucha funkcji - każda kolejna funkcja modyfikuje obiekt wynikowy poprzedniej funkcji.

Wypełnienie całej dostępnej szerokości – nadrzędnego elementu

Zmiana właściwości tła.

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxWidth()  
                    .background(Color.Cyan)  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

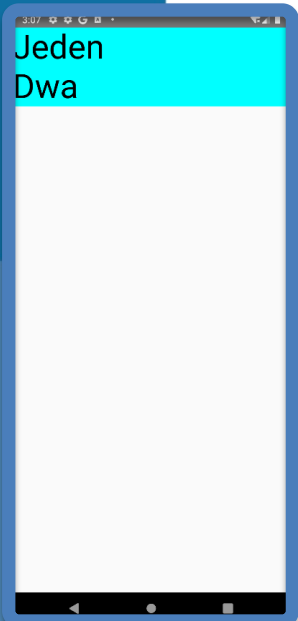

Modifier w **Compose** to obiekt, który służy do modyfikacji wyglądu lub zachowania komponentu.

Modifier działa na zasadzie łańcucha funkcji - każda kolejna funkcja modyfikuje obiekt wynikowy poprzedniej funkcji.

Wypełnienie całej dostępnej szerokości – nadrzędnego elementu

Zmiana właściwości tła.

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxWidth()  
                    .background(Color.Cyan)  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```



Alignment i Arrangement

Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxSize()  
                    .background(Color.Cyan),  
                horizontalAlignment = Alignment.CenterHorizontally  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

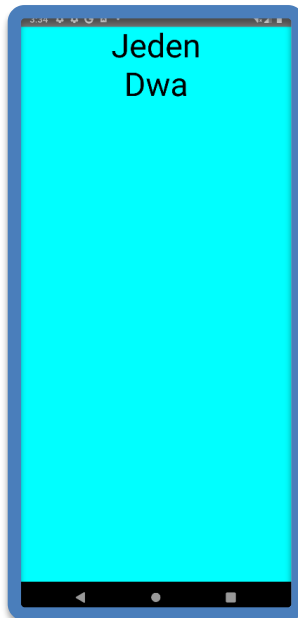
Wypełnia **całą** dostępną przestrzeń

Alignment i Arrangement

Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).

Ustawia poziome **wyrównanie elementów** w **Column**. Oznacza to, że **wszystkie dzieci** wewnątrz będą **wyśrodkowane w poziomie**.

Wypełnia **całą** dostępną przestrzeń



```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxSize()  
                    .background(Color.Cyan),  
                horizontalAlignment = Alignment.CenterHorizontally  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Alignment i Arrangement

Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxSize()  
                    .background(Color.Cyan),  
                horizontalAlignment = Alignment.CenterHorizontally,  
                verticalArrangement = Arrangement.Center  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Ustawia **pionowe wyrównanie** elementów w Column. Oznacza to, że **wszystkie dzieci** wewnątrz będą **wyśrodkowane w pionie**.

Wypełnia **całą** dostępną przestrzeń



Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).

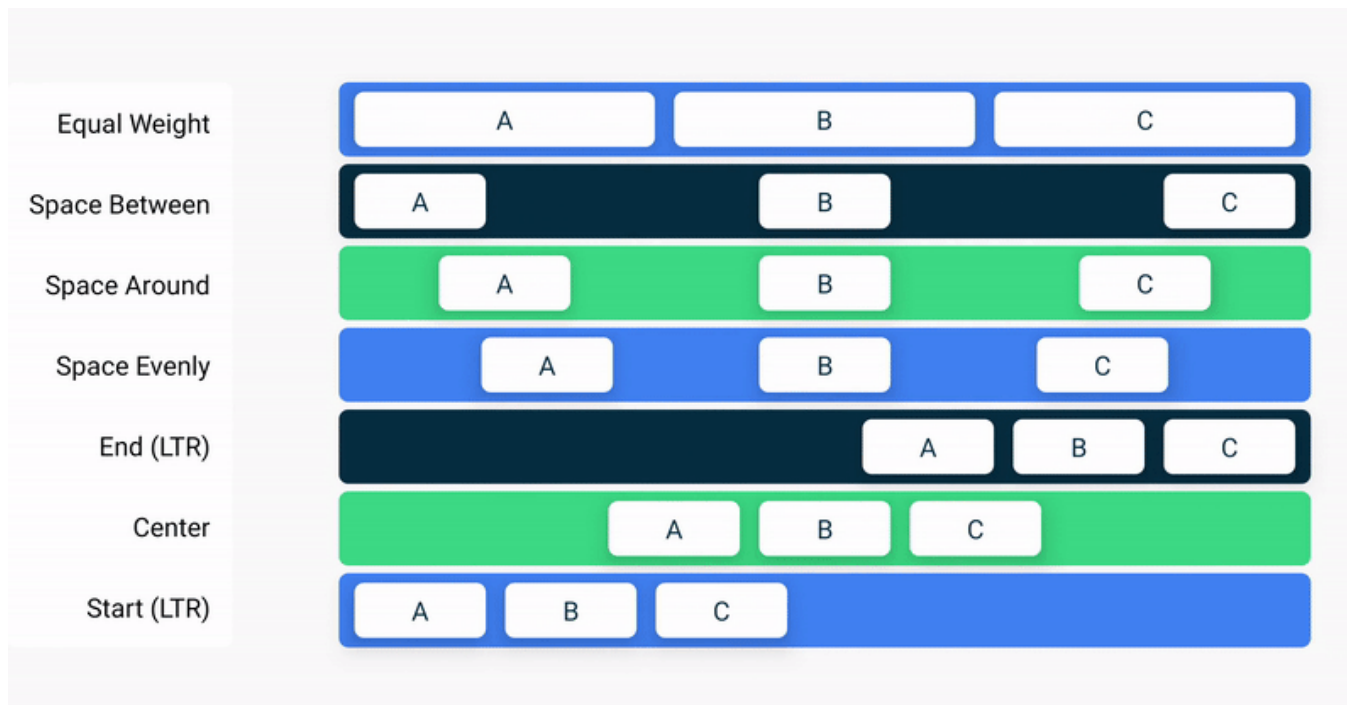
```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Row(  
                modifier = Modifier  
                    .fillMaxSize()  
                    .background(Color.Cyan),  
                horizontalArrangement = Arrangement.Center,  
                verticalAlignment = Alignment.CenterVertically  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Column(  
                modifier = Modifier  
                    .fillMaxSize()  
                    .background(Color.Cyan),  
                horizontalAlignment = Alignment.CenterHorizontally,  
                verticalArrangement = Arrangement.Center  
            ) {  
                Text(text = "Jeden", fontSize = 50.sp)  
                Text(text = "Dwa", fontSize = 50.sp)  
            }  
        }  
    }  
}
```

Alignment i Arrangement

Alignment/Arrangement dotyczy **wyrównania elementów** w ramach **jednego wiersza lub kolumny** (w zależności od tego, czy korzystamy z komponentu Row czy Column).



Stan w aplikacji to **dowolna wartość, która może ulec zmianie w czasie**. **Compose** jest deklaratywny, co oznacza, że jedynym sposobem na **aktualizację interfejsu** jest **wywołanie** tej samej funkcji kompozycyjnej z nowymi argumentami, które reprezentują **nowy stan**.

Stan w aplikacji to **dowolna wartość, która może ulec zmianie w czasie**. **Compose** jest deklaratywny, co oznacza, że jedynym sposobem na **aktualizację interfejsu** jest **wywołanie** tej samej funkcji kompozycyjnej z nowymi argumentami, które reprezentują **nowy stan**.

W **Compose** interfejs użytkownika jest budowany poprzez **kompozycję**, czyli **drzewo strukturalne komponentów (composables)** które opisują interfejs użytkownika.

Stan w aplikacji to **dowolna wartość, która może ulec zmianie w czasie**. **Compose** jest deklaratywny, co oznacza, że jedynym sposobem na **aktualizację interfejsu** jest **wywołanie** tej samej funkcji kompozycyjnej z nowymi argumentami, które reprezentują **nowy stan**.

W **Compose** interfejs użytkownika jest budowany poprzez **kompozycję**, czyli **drzewo strukturalne komponentów (composables)** które opisują interfejs użytkownika.

Gdy **stan się zmienia**, **Compose** przeprowadza **rekompozycję**, czyli ponowne uruchomienie funkcji kompozycyjnych, aby zaktualizować interfejs.

Aby **zachować stan** między rekompozycjami, możemy użyć funkcji **remember**. Funkcja przechowuje obiekt w pamięci podczas pierwszej kompozycji i zwraca go podczas kolejnych rekompozycji.

Zachowuje **stan**
name **między**
rekompozycjami

```
1  @Composable
2  fun HelloContent() {
3      Column(modifier = Modifier.padding(16.dp)) {
4          var name by remember { mutableStateOf("") }
5          if (name.isNotEmpty()) {
6              Text(
7                  text = "Hello, $name!",
8                  modifier = Modifier.padding(bottom = 8.dp),
9                  style = MaterialTheme.typography.bodyMedium
10             )
11         }
12         OutlinedTextField(
13             value = name,
14             onChange = { name = it },
15             label = { Text("Name") }
16         )
17     }
18 }
```

Aby **zachować stan** między rekompozycjami, możemy użyć funkcji **remember**. Funkcja przechowuje obiekt w pamięci podczas pierwszej kompozycji i zwraca go podczas kolejnych rekompozycji.

Zachowuje **stan**
name **między**
rekompozycjami

```
1  @Composable
2  fun HelloContent() {
3      Column(modifier = Modifier.padding(16.dp)) {
4          var name by remember { mutableStateOf("") }
5          if (name.isNotEmpty()) {
6              Text(
7                  text = "Hello, $name!",
8                  modifier = Modifier.padding(bottom = 8.dp),
9                  style = MaterialTheme.typography.bodyMedium
10             )
11         }
12         OutlinedTextField(
13             value = name,
14             onChange = { name = it },
15             label = { Text("Name") }
16         )
17     }
18 }
```

Wartość **domyślna**

Tworzy **obserwowalny**
obiekt **MutableState<T>**

Aby **zachować stan** między rekompozycjami, możemy użyć funkcji **remember**. Funkcja przechowuje obiekt w pamięci podczas pierwszej kompozycji i zwraca go podczas kolejnych rekompozycji.

Zachowuje **stan**
name **między**
rekompozycjami

Warunek utworzenia
komponentu

Wartość **domyślna**

Tworzy **obserwowalny**
obiekt **MutableState<T>**

```
1  @Composable
2  fun HelloContent() {
3      Column(modifier = Modifier.padding(16.dp)) {
4          var name by remember { mutableStateOf("") }
5          if (name.isNotEmpty()) {
6              Text(
7                  text = "Hello, $name!",
8                  modifier = Modifier.padding(bottom = 8.dp),
9                  style = MaterialTheme.typography.bodyMedium
10             )
11         }
12         OutlinedTextField(
13             value = name,
14             onChange = { name = it },
15             label = { Text("Name") }
16         )
17     }
18 }
```

Aby **zachować stan** między rekompozycjami, możemy użyć funkcji **remember**. Funkcja przechowuje obiekt w pamięci podczas pierwszej kompozycji i zwraca go podczas kolejnych rekompozycji.

```
1  @Composable
2  fun HelloContent() {
3      Column(modifier = Modifier.padding(16.dp)) {
4          var name by remember { mutableStateOf("") }
5          if (name.isNotEmpty()) {
6              Text(
7                  text = "Hello, $name!",
8                  modifier = Modifier.padding(bottom = 8.dp),
9                  style = MaterialTheme.typography.bodyMedium
10             )
11         }
12         OutlinedTextField(
13             value = name,
14             onChange = { name = it },
15             label = { Text("Name") }
16         )
17     }
18 }
```

Zachowuje **stan**
name **między**
rekompozycjami

Warunek utworzenia
komponentu

Wartość **domyślna**

Tworzy **obserwowalny**
obiekt **MutableState<T>**



Zachowanie Stanu

Aby **zachować stan** między rekompozycjami, możemy użyć funkcji **remember**. Funkcja przechowuje obiekt w pamięci podczas pierwszej kompozycji i zwraca go podczas kolejnych rekompozycji.

```
1  @Composable
2  fun HelloContent() {
3      Column(modifier = Modifier.padding(16.dp)) {
4          var name by remember { mutableStateOf("") }
5          if (name.isNotEmpty()) {
6              Text(
7                  text = "Hello, $name!",
8                  modifier = Modifier.padding(bottom = 8.dp),
9                  style = MaterialTheme.typography.bodyMedium
10             )
11         }
12         OutlinedTextField(
13             value = name,
14             onChange = { name = it },
15             label = { Text("Name") }
16         )
17     }
18 }
```

Zachowuje **stan**
name **między**
rekompozycjami

Warunek utworzenia
komponentu

Wartość **domyślna**

Tworzy **obserwowalny**
obiekt **MutableState<T>**

Wartość pola

Funkcja wywoływana
przy **zmianie wartości**

Wartość wpisana
przez użytkownika



Komponent **stateful** to taki, który przechowuje i zarządza swoim stanem wewnętrznie, zazwyczaj za pomocą funkcji **remember** lub **rememberSaveable**.

Zachowuje stan
wewnętrznie

```
1  @Composable
2  fun HelloContent() {
3      var name by rememberSaveable { mutableStateOf("") }
4      Column(modifier = Modifier.padding(16.dp)) {
5          Text(text = "Hello, $name")
6          OutlinedTextField(
7              value = name,
8              onChange = { name = it },
9              label = { Text("Name") }
10         )
11     }
12 }
```


Komponent **stateful** to taki, który przechowuje i zarządza swoim stanem wewnętrznie, zazwyczaj za pomocą funkcji **remember** lub **rememberSaveable**.

Zachowuje stan
wewnętrznie

```
1  @Composable
2  fun HelloContent() {
3
4      var name by rememberSaveable { mutableStateOf("") }
5
6      Column(modifier = Modifier.padding(16.dp)) {
7          Text(text = "Hello, $name")
8          OutlinedTextField(
9              value = name,
10             onChange = { name = it },
11             label = { Text("Name") }
12         )
13     }
14 }
```

Komponent **stateless** to taki, który **nie przechowuje stanu wewnętrznie**. Zamiast tego, stan jest **przekazywany** do komponentu **jako parametr**, a **zdarzenia** (np. zmiana wartości) są zgłaszane do **wyższych warstw aplikacji**. Aby osiągnąć stateless, stosujemy wzorzec **state hoisting**.

Wzorzec State Hoisting

State hoisting to wzorzec polegający na przeniesieniu stanu do wyższego poziomu w hierarchii komponentów, co czyni komponent **stateless**.

W praktyce oznacza to zastąpienie wewnętrznego stanu dwoma parametrami:

- **value: T** - Aktualna wartość do wyświetlenia.
- **onValueChange: (T) -> Unit**: Funkcja wywoływana, gdy wartość ma się zmienić.

```
1  @Composable
2  fun HelloScreen() {
3      var name by rememberSaveable { mutableStateOf("") }
4      HelloContent(name = name, onNameChange = { name = it })
5  }
6
7  @Composable
8  fun HelloContent(name: String, onNameChange: (String) -> Unit) {
9      Column(modifier = Modifier.padding(16.dp)) {
10         Text(text = "Hello, $name")
11         OutlinedTextField(
12             value = name,
13             onValueChange = onNameChange,
14             label = { Text("Name") }
15         )
16     }
17 }
```

Dlaczego korzystamy ze wzorca **State Hoisting**:

- **Single Source of Truth**: Stan jest przechowywany w jednym miejscu, co zapobiega niespójnościom.
- **Współdzielenie**: Stan może być współdzielony między wieloma komponentami.
- **Przechwytywanie zdarzeń**: Wyższe warstwy mogą modyfikować lub ignorować zdarzenia przed aktualizacją stanu.
- **Elastyczność**: Stan może być przechowywany np. w **ViewModel**, co ułatwia integrację z architekturą aplikacji.

HelloScreen

state

event

HelloContent

```
1  @Composable
2  fun HelloScreen() {
3      var name by rememberSaveable { mutableStateOf("") }
4      HelloContent(name = name, onNameChange = { name = it })
5  }
6
7  @Composable
8  fun HelloContent(name: String, onNameChange: (String) -> Unit) {
9      Column(modifier = Modifier.padding(16.dp)) {
10         Text(text = "Hello, $name")
11         OutlinedTextField(
12             value = name,
13             onValueChange = onNameChange,
14             label = { Text("Name") }
15         )
16     }
17 }
```

- Stan *pły*nie w dół: HelloScreen **przekazuje stan name** do HelloContent.
- Zdarzenia *płyną* w górę: HelloContent **zgłasza zmiany stanu** do HelloScreen za pomocą **onNameChange**.