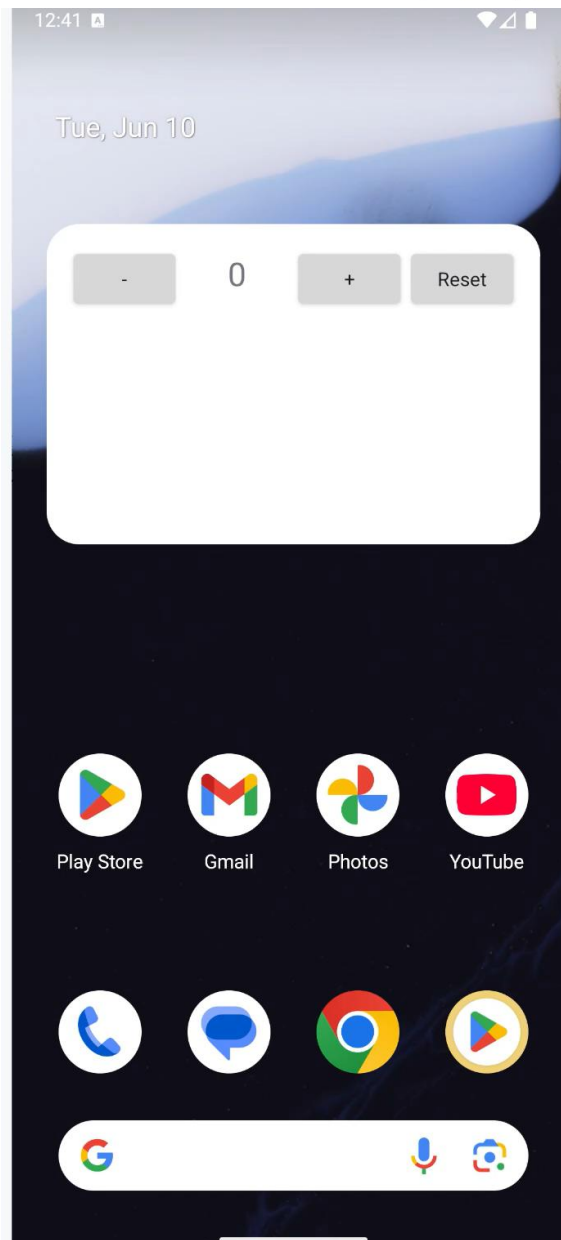




# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

## WYKŁAD 14

- Aplikacja Hybrydowa
- SharedPreferences
- Widżety



- **MainActivity** + **CounterApp** (Jetpack Compose) - Główny ekran aplikacji, zbudowany przy użyciu Jetpack Compose

- **MainActivity + CounterApp** (Jetpack Compose) - Główny ekran aplikacji, zbudowany przy użyciu Jetpack Compose
- Stan zarządzany przez **SharedPreferences** - Wartość licznika jest przechowywana w SharedPreferences (klucz "count"). Aplikacja subskrybuje zmiany za pomocą OnSharedPreferencesChangeListener, dzięki czemu UI automatycznie się aktualizuje.

- **MainActivity + CounterApp** (Jetpack Compose) - Główny ekran aplikacji, zbudowany przy użyciu Jetpack Compose
- Stan zarządzany przez **SharedPreferences** - Wartość licznika jest przechowywana w SharedPreferences (klucz "count"). Aplikacja subskrybuje zmiany za pomocą OnSharedPreferencesChangeListener, dzięki czemu UI automatycznie się aktualizuje.
- **CounterWidget (AppWidgetProvider)** - Widżet na ekranie głównym, który wyświetla ten sam licznik co aplikacja i pozwala na jego modyfikację. UI zdefiniowany w „tradycyjnym” XML (bo widżety nie obsługują Compose).

- **MainActivity + CounterApp** (Jetpack Compose) - Główny ekran aplikacji, zbudowany przy użyciu Jetpack Compose
- Stan zarządzany przez **SharedPreferences** - Wartość licznika jest przechowywana w SharedPreferences (klucz "count"). Aplikacja subskrybuje zmiany za pomocą OnSharedPreferencesChangeListener, dzięki czemu UI automatycznie się aktualizuje.
- **CounterWidget (AppWidgetProvider)** - Widżet na ekranie głównym, który wyświetla ten sam licznik co aplikacja i pozwala na jego modyfikację. UI zdefiniowany w „tradycyjnym” XML (bo widżety nie obsługują Compose).
- **CounterReceiver (BroadcastReceiver)** - Odbiera i przetwarza akcje (INCREMENT, DECREMENT, RESET) wysyłane z widżetu. Po zmianie danych wywołuje CounterWidget().onUpdate(), aby odświeżyć UI widżetu.

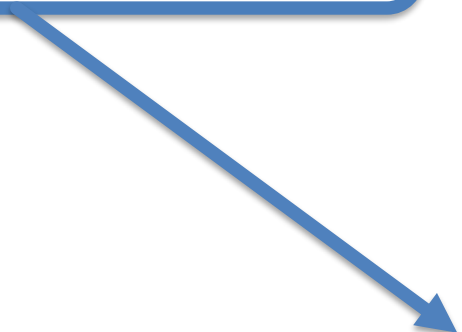
- **MainActivity + CounterApp** (Jetpack Compose) - Główny ekran aplikacji, zbudowany przy użyciu Jetpack Compose
- Stan zarządzany przez **SharedPreferences** - Wartość licznika jest przechowywana w SharedPreferences (klucz "count"). Aplikacja subskrybuje zmiany za pomocą OnSharedPreferencesChangeListener, dzięki czemu UI automatycznie się aktualizuje.
- **CounterWidget (AppWidgetProvider)** - Widżet na ekranie głównym, który wyświetla ten sam licznik co aplikacja i pozwala na jego modyfikację. UI zdefiniowany w „tradycyjnym” XML (bo widżety nie obsługują Compose).
- **CounterReceiver (BroadcastReceiver)** - Odbiera i przetwarza akcje (INCREMENT, DECREMENT, RESET) wysyłane z widżetu. Po zmianie danych wywołuje CounterWidget().onUpdate(), aby odświeżyć UI widżetu.
- **MainActivity (Compose)** –UI, reagujący na zmiany w SharedPreferences.
- **CounterWidget (AppWidgetProvider)** – tradycyjny widżet, który komunikuje się przez Broadcast.
- **CounterReceiver** – pośrednik między widżetem a danymi.
- **SharedPreferences** – źródło prawdy, zapewniające spójność stanu.

- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych



- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych

getSharedPreferences("nazwa", tryb)  
**Otwiera/tworzy** plik preferences na  
**wybranym kontekście.**



```
val sharedPref = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)

fun incrementCounter() {
    val currentCount = sharedPref.getInt("count", 0) // Domyślna wartość: 0
    sharedPref.edit()
        .putInt("count", currentCount + 1)
        .commit()
}
```

# SharedPreferences

- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych

`getSharedPreferences("nazwa", tryb)`  
**Otwiera/tworzy** plik preferences na  
**wybranym kontekście**.

Tryby dostępu do pliku:

**MODE\_APPEND** – dopisuje **bez nadpisywania**

**MODE\_PRIVATE** – dostęp do pliku tylko z **poziomu**  
aplikacji

**MODE\_WORLD\_READABLE** – zezwala **innym**  
aplikacjom na odczyt

**MODE\_WORLD\_WRITABLE** – zezwala **innym**  
aplikacjom na zapis

```
val sharedPref = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)

fun incrementCounter() {
    val currentCount = sharedPref.getInt("count", 0) // Domyślna wartość: 0
    sharedPref.edit()
        .putInt("count", currentCount + 1)
        .commit()
}
```

# SharedPreferences

- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych

`getSharedPreferences("nazwa", tryb)`  
**Otwiera/tworzy** plik preferences na wybranym kontekście.

Tryby dostępu do pliku:

**MODE\_APPEND** – dopisuje **bez nadpisywania**

**MODE\_PRIVATE** – dostęp do pliku tylko z **poziomu aplikacji**

**MODE\_WORLD\_READABLE** – zezwala innym aplikacjom na odczyt

**MODE\_WORLD\_WRITABLE** – zezwala innym aplikacjom na zapis

```
val sharedPref = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)

fun incrementCounter() {
    val currentCount = sharedPref.getInt("count", 0) // Domyślna wartość: 0
    sharedPref.edit()
        .putInt("count", currentCount + 1)
        .commit()
}
```

Odczytuje wartość pod kluczem.

# SharedPreferences

- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych

`getSharedPreferences("nazwa", tryb)`  
**Otwiera/tworzy** plik preferences na wybranym kontekście.

Tryby dostępu do pliku:

**MODE\_APPEND** – dopisuje **bez nadpisywania**

**MODE\_PRIVATE** – dostęp do pliku tylko z **poziomu aplikacji**

**MODE\_WORLD\_READABLE** – zezwala innym aplikacjom na odczyt

**MODE\_WORLD\_WRITABLE** – zezwala innym aplikacjom na zapis

```
val sharedPref = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)

fun incrementCounter() {
    val currentCount = sharedPref.getInt("count", 0) // Domyślna wartość: 0
    sharedPref.edit()
        .putInt("count", currentCount + 1)
        .commit()
}
```

Odczytuje wartość pod kluczem.

Rozpoczyna edycję

Zapisuje wartość pod kluczem.

# SharedPreferences

- Przechowuje dane w plikach **XML** i obsługuje tylko **typy proste**
- Oferuje tylko operacje **synchroniczne**
- **Nie posiada** żadnych **wbudowanych** mechanizmów
- **Nie oferuje** wbudowanej obsługi **reagowania** na zmiany danych

`getSharedPreferences("nazwa", tryb)`  
**Otwiera/tworzy** plik preferences na wybranym kontekście.

Tryby dostępu do pliku:

**MODE\_APPEND** – dopisuje **bez nadpisywania**

**MODE\_PRIVATE** – dostęp do pliku tylko z **poziomu aplikacji**

**MODE\_WORLD\_READABLE** – zezwala innym aplikacjom na odczyt

**MODE\_WORLD\_WRITABLE** – zezwala innym aplikacjom na zapis

```
val sharedPref = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)
```

```
fun incrementCounter() {  
    val currentCount = sharedPref.getInt("count", 0) // Domyślna wartość: 0  
    sharedPref.edit()  
        .putInt("count", currentCount + 1)  
        .commit()  
}
```

Zapisuje dane synchronicznie (zwraca true/false).

Odczytuje wartość pod kluczem.

Rozpoczyna edycję

Zapisuje wartość pod kluczem.

```
@Composable
fun CounterApp() {
    val context = LocalContext.current
    val prefs = remember { context.getSharedPreferences("CounterPrefs", Context.MODE_PRIVATE) }
    var count by remember { mutableIntStateOf(prefs.getInt("count", 0)) }

    DisposableEffect(Unit) {
        val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->
            if (key == "count") count = prefs.getInt(key, 0)
        }
        prefs.registerOnSharedPreferenceChangeListener(listener)
        onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }
    }

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text("Licznik: $count", fontSize = 24.sp, modifier = Modifier.padding(16.dp))
        Button(onClick = { prefs.edit { putInt("count", count + 1) } }) {
            Text("Zwiększ")
        }
        Button(onClick = { prefs.edit { putInt("count", count - 1) } }) {
            Text("Zmniejsz")
        }
        Button(onClick = { prefs.edit { putInt("count", 0) } }) {
            Text("Resetuj")
        }
    }
}
```

"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

Wykonanie kodu **przy wejściu komponentu do kompozycji** (analogia do onCreate() w Activity). Automatyczne sprzątanie **przy opuszczaniu kompozycji** (analogia do onDestroy())

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```




"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

Wykonanie kodu **przy wejściu komponentu do kompozycji** (analogia do onCreate() w Activity). Automatyczne sprzątanie **przy opuszczaniu kompozycji** (analogia do onDestroy())

- Wejście komponentu: Gdy funkcja **@Composable** po raz pierwszy pojawia się na ekranie.
- Ponowne komponowanie (**recomposition**): **Nie uruchamia się ponownie**, chyba że **zmieniają się klucze** przekazane do DisposableEffect (tutaj Unit oznacza, że działa tylko raz).
- Wyjście komponentu: Gdy komponent jest trwale usuwany z UI (np. przy nawigacji wstecz).

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.



```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

# DisposableEffect

"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

Nasłuchuje **zmian** w SharedPreferences. Gdy zmieni się wartość pod kluczem "count", aktualizuje zmienną count.

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

# DisposableEffect

"**efekt uboczny**„ (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

Nasłuchuje **zmian** w SharedPreferences. Gdy zmieni się wartość pod kluczem "count", aktualizuje zmienną count.

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

Rejestracja/wyrejestrowanie listener'a będzie wywoływany przy każdej zmianie w prefs.

# DisposableEffect

"**efekt uboczny**," (side effect) w Compose, który wykonuje kod przy wejściu komponentu do kompozycji i sprząta przy wyjściu.

Nasłuchuje **zmian** w SharedPreferences. Gdy zmieni się wartość pod kluczem "count", aktualizuje zmienną count.

```
DisposableEffect(Unit) {  
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->  
        if (key == "count") count = prefs.getInt(key, 0)  
    }  
    prefs.registerOnSharedPreferenceChangeListener(listener)  
    onDispose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }  
}
```

Przy **wyjściu** z komponentu automatycznie wywołana jest metoda onDispose, usuwając listenera. Zapobiega **wyciekowi pamięci**

Rejestracja/wyrejestrowanie listener'a będzie wywoływany przy każdej zmianie w prefs.

```
class CounterWidget : AppWidgetProvider() {  
    override fun onUpdate(context: Context, appWidgetManager: AppWidgetManager, appWidgetIds: IntArray) {  
        for (appWidgetId in appWidgetIds) {  
            updateAppWidget(context, appWidgetManager, appWidgetId)  
        }  
    }  
}  
  
private fun updateAppWidget(context: Context, appWidgetManager: AppWidgetManager, appWidgetId: Int) {  
    val prefs = context.getSharedPreferences("CounterPrefs", Context.MODE_PRIVATE)  
    val views = RemoteViews(context.packageName, R.layout.widget_counter)  
  
    views.setTextViewText(R.id.appwidget_text, prefs.getInt("count", 0).toString())  
  
    // Konfiguracja przycisków  
    fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
        context,  
        0,  
        Intent(context, CounterReceiver::class.java).setAction(action),  
        PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
    )  
  
    views.setOnClickPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
    views.setOnClickPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
    views.setOnClickPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))  
  
    appWidgetManager.updateAppWidget(appWidgetId, views)  
}
```

# Layout – widget\_counter.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#FFFFFF"
    android:orientation="horizontal"
    android:padding="16dp">

    <Button
        android:id="@+id/button_decrement"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="-"/>

    <TextView
        android:id="@+id/appwidget_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="center"
        android:text="0"
        android:textSize="24sp"/>

    <Button
        android:id="@+id/button_increment"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="+"/>

    <Button
        android:id="@+id/button_reset"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Reset"/>

</LinearLayout>
```



```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="250dp"
    android:minHeight="100dp"
    android:updatePeriodMillis="0"
    android:initialLayout="@layout/widget_counter"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen"/>
```



**AppWidgetProvider** to uproszczona klasa bazowa do tworzenia widżetów.

**Automatycznie** obsługuje broadcasty systemowe związane z widżetami (np. aktualizacje).

```
override fun onUpdate(  
    context: Context,  
    appWidgetManager: AppWidgetManager,  
    appWidgetIds: IntArray) {  
    for (appWidgetId in appWidgetIds) {  
        updateAppWidget(context, appWidgetManager, appWidgetId)  
    }  
}
```

**Kiedy się wywołuje?**


- Przy **pierwszym dodaniu** widżetu na ekran.
- Po upływie **interwału** aktualizacji (zdefiniowanego w widget\_info.xml).
- Po **ręcznym odświeżeniu** przez użytkownika.

**Co robi?**

- Dla **każdej instancji** widżetu (**może być wiele!**) wywołuje updateAppWidget().

```
private fun updateAppWidget(  
    context: Context, appWidgetManager: AppWidgetManager,  
    appWidgetId: Int) {  
    val prefs = context.getSharedPreferences("CounterPrefs", Context.MODE_PRIVATE)  
    val views = RemoteViews(context.packageName, R.layout.widget_counter)  
  
    views.setTextViewText(R.id.appwidget_text, prefs.getInt("count", 0).toString())  
  
    // Konfiguracja przycisków  
    fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
        context,  
        0,  
        Intent(context, CounterReceiver::class.java).setAction(action),  
        PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
    )  
  
    views.setOnClickPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
    views.setOnClickPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
    views.setOnClickPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))  
  
    appWidgetManager.updateAppWidget(appWidgetId, views)  
}
```

```
private fun updateAppWidget(  
    context: Context, appWidgetManager: AppWidgetManager,  
    appWidgetId: Int) {  
    val prefs = context.getSharedPreferences("CounterPrefs", Context.MODE_PRIVATE)  
    val views = RemoteViews(context.packageName, R.layout.widget_counter)
```



Klasa pozwalająca na **manipulowanie** widżetem aplikacji **z poziomu kodu**. Widżety działają w **procesie systemowym** (nie w aplikacji!!!), więc nie można bezpośrednio modyfikować ich UI. RemoteViews to "most" między aplikacją a systemowym procesem launchera.

```
private fun updateAppWidget(  
    context: Context, appWidgetManager: AppWidgetManager,  
    appWidgetId: Int) {  
    val prefs = context.getSharedPreferences("CounterPrefs", Context.MODE_PRIVATE)  
    val views = RemoteViews(context.packageName, R.layout.widget_counter)
```

Klasa pozwalająca na **manipulowanie** widżetem aplikacji **z poziomu kodu**. Widżety działają w **procesie systemowym** (nie w aplikacji!!!), więc nie można bezpośrednio modyfikować ich UI. RemoteViews to "most" między aplikacją a systemowym procesem launchera.

R.layout.widget\_counter:  
Wskazuje **plik XML z layoutem widżetu**  
( res/layout/widget\_counter.xml).

Nazwa pakietu aplikacji.  
System musi „wiedzieć”,  
do której aplikacji należy  
widżet (i gdzie szukać  
zasobów).

<Button

```
android:id="@+id/button_decrement"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:text="-"/>
```

<TextView

```
android:id="@+id/appwidget_text"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:gravity="center"
android:text="0"
android:textSize="24sp"/>
```

```
views.setTextViewText(
    R.id.appwidget_text,
    prefs.getInt("count", 0).toString())
```

Identyfikator pola TextView,  
które będzie aktualizowane

# PendingIntent

**PendingIntent** to **specjalny rodzaj intencji** w Androidzie, który pozwala **innym aplikacjom** lub systemowi **wykonać akcję** w „imieniu aplikacji”. Jest potrzebny dla widżetów, powiadomień i alarmów. Działa nawet gdy aplikacja jest zamknięta. Zapewnia autoryzację – tylko aplikacja może go użyć.

// Konfiguracja przycisków

```
fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
    context,  
    0, ←  
    Intent(context, CounterReceiver::class.java).setAction(action),  
    PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
)
```

Kod żądania (używany  
do identyfikacji intentów)

```
views.setOnClickListenerPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))
```

# PendingIntent

**PendingIntent** to **specjalny rodzaj intencji** w Androidzie, który pozwala **innym aplikacjom** lub systemowi **wykonać akcję** w „imieniu aplikacji”. Jest potrzebny dla widżetów, powiadomień i alarmów. Działa nawet gdy aplikacja jest zamknięta. Zapewnia autoryzację – tylko aplikacja może go użyć.

// Konfiguracja przycisków

```
fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
    context,  
    0, ←  
    Intent(context, CounterReceiver::class.java).setAction(action),  
    PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
)
```

Kod żądania (używany  
do identyfikacji intentów)

Aktualizuje istniejący PendingIntent

Wymagany ze względów bezpieczeństwa

```
views.setOnClickListenerPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))
```

# PendingIntent

**PendingIntent** to **specjalny rodzaj intencji** w Androidzie, który pozwala **innym aplikacjom** lub systemowi **wykonać akcję** w „imieniu aplikacji”. Jest potrzebny dla widżetów, powiadomień i alarmów. Działa nawet gdy aplikacja jest zamknięta. Zapewnia autoryzację – tylko aplikacja może go użyć.

// *Konfiguracja przycisków*

```
fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
    context,  
    0,  
    Intent(context, CounterReceiver::class.java).setAction(action),  
    PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
)
```

Kod żądania (używany  
do identyfikacji intentów)

Aktualizuje istniejący PendingIntent

Wymagany ze względów bezpieczeństwa

```
views.setOnClickListenerPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))
```

Każdy przycisk w widżecie dostaje swój PendingIntent  
Akcje są rozróżniane przez różne stringi  
("INCREMENT", "DECREMENT", "RESET")



# PendingIntent

**PendingIntent** to **specjalny rodzaj intencji** w Androidzie, który pozwala **innym aplikacjom** lub systemowi **wykonać akcję** w „imieniu aplikacji”. Jest potrzebny dla widżetów, powiadomień i alarmów. Działa nawet gdy aplikacja jest zamknięta. Zapewnia autoryzację – tylko aplikacja może go użyć.

// Konfiguracja przycisków

```
fun getPendingIntent(action: String) = PendingIntent.getBroadcast(  
    context,  
    0,  
    Intent(context, CounterReceiver::class.java).setAction(action),  
    PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE  
)
```

Kod żądania (używany do identyfikacji intentów)

Aktualizuje istniejący PendingIntent

Wymagany ze względów bezpieczeństwa

```
views.setOnClickListenerPendingIntent(R.id.button_increment, getPendingIntent("com.example.INCREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_decrement, getPendingIntent("com.example.DECREMENT"))  
views.setOnClickListenerPendingIntent(R.id.button_reset, getPendingIntent("com.example.RESET"))
```

Każdy przycisk w widżecie dostaje swój PendingIntent  
Akcje są rozróżniane przez różne stringi  
("INCREMENT", "DECREMENT", "RESET")

Jak to działa w praktyce?  
Użytkownik klika przycisk w widżecie  
System wykonuje PendingIntent  
Wysyłany jest broadcast do CounterReceiver  
CounterReceiver odbiera akcję i modyfikuje licznik

CounterReceiver to **odbiornik broadcastów**, który reaguje na akcje związane z **modyfikacją licznika** w widżecie.

```
class CounterReceiver : BroadcastReceiver() {  
    override fun onReceive(context: Context, intent: Intent) {  
        val prefs = context.getSharedPreferences("CounterPrefs",  
            Context.MODE_PRIVATE  
        )  
        when (intent.action) {  
            "com.example.INCREMENT" ->  
                prefs.edit().putInt("count", prefs.getInt("count", 0) + 1)  
                    .apply()  
            "com.example.DECREMENT" ->  
                prefs.edit().putInt("count", prefs.getInt("count", 0) - 1)  
                    .apply()  
            "com.example.RESET" -> prefs.edit().putInt("count", 0).apply()  
        }  
  
        // Aktualizacja wszystkich widżetów  
        val appWidgetManager = AppWidgetManager.getInstance(context)  
        val appWidgetIds = appWidgetManager  
            .getAppWidgetIds(ComponentName(context, CounterWidget::class.java))  
        CounterWidget().onUpdate(context, appWidgetManager, appWidgetIds)  
    }  
}
```

```
<application
  <activity
    android:theme="@style/Theme.MyApplication">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <receiver
      android:name=".CounterWidget"
      android:exported="true">
        <intent-filter>
          <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
        </intent-filter>
        <meta-data
          android:name="android.appwidget.provider"
          android:resource="@xml/widget_counter_info"/>
        </receiver>

        <receiver
          android:name=".CounterReceiver"
          android:exported="false">
            <intent-filter>
              <action android:name="com.example.INCREMENT"/>
              <action android:name="com.example.DECREMENT"/>
              <action android:name="com.example.RESET"/>
            </intent-filter>
          </receiver>
        </application>
```