




Programming with Kotlin


Agenda

1. Operacje asynchroniczne w Androidzie
 2. Korutyny
 3. Flow
 4. Zaawansowane widoki
 5. Testowanie w Androidzie
- 
-



Operacje asynchroniczne w androidzie

MainThread

- specjalny wątek tworzony na starcie aplikacji
 - jedyny, który może modyfikować UI
 - obsługuje zdarzenia np. dotyk
 - lifecycle callbacks np. onCreate, onResume
 - blokada powoduje ANR (Application Non Responding)
- 
-

ANR (Application Non Responding)

- operacje sieciowe
- operacje bazodanowe
- operacje na systemie plików
- długotrwałe obliczenia

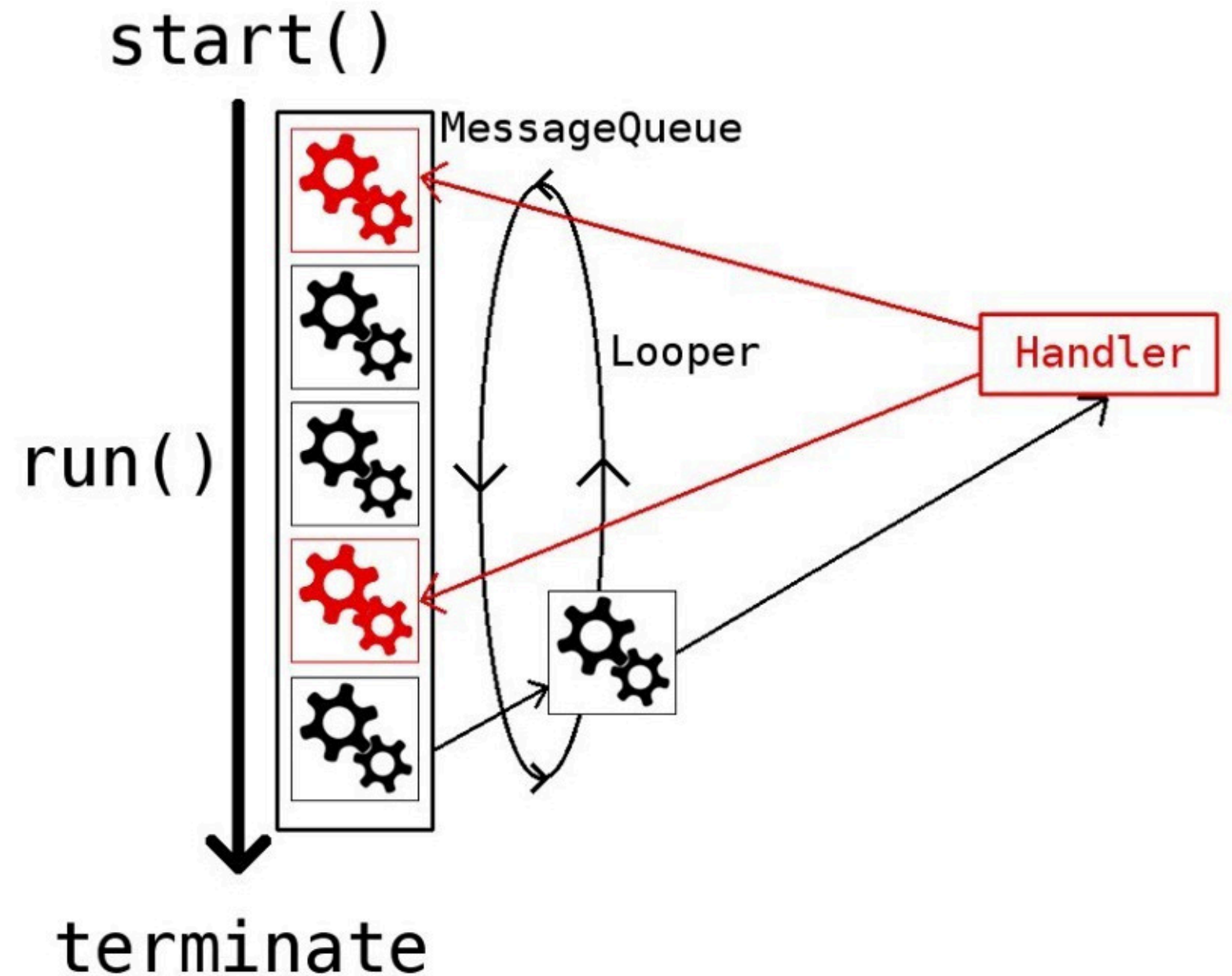
Uruchamianie kodu w MainThread

- `Activity.runOnUiThread(Runnable)`
- `View.post(Runnable)`
- Handler z main Looper

```
Handler mainHandler = new Handler(Looper.getMainLooper());
mainHandler.post(new Runnable() {
    @Override
    public void run() {
        // Code to run on MainThread
    }
});
```


Najważniejsze klasy

- Thread
- Looper
- MessageQueue
- Handler
- HandlerThread (Thread z Looper)
- Message & Runnable



Looper

```
class LooperThread extends Thread {  
    public Handler mHandler;  
  
    public void run() {  
        Looper.prepare();  
  
        mHandler = new Handler(Looper.myLooper()) {  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
            }  
        };  
  
        Looper.loop();  
    }  
}
```




Handler i Message


```
Handler mHandler = new Handler(new Handler.Callback() {
    @Override
    public boolean handleMessage(Message msg) {
        if(msg.what == MESSAGE_TYPE_1) {
            Log.d("HandlerDemo", "Message type 1 received");
            return true;
        }
        return false;
    }
});

// Send a message to the handler after a delay
Message msg = mHandler.obtainMessage(MESSAGE_TYPE_1);
mHandler.sendMessageDelayed(msg, 1000); // 1 second delay
```

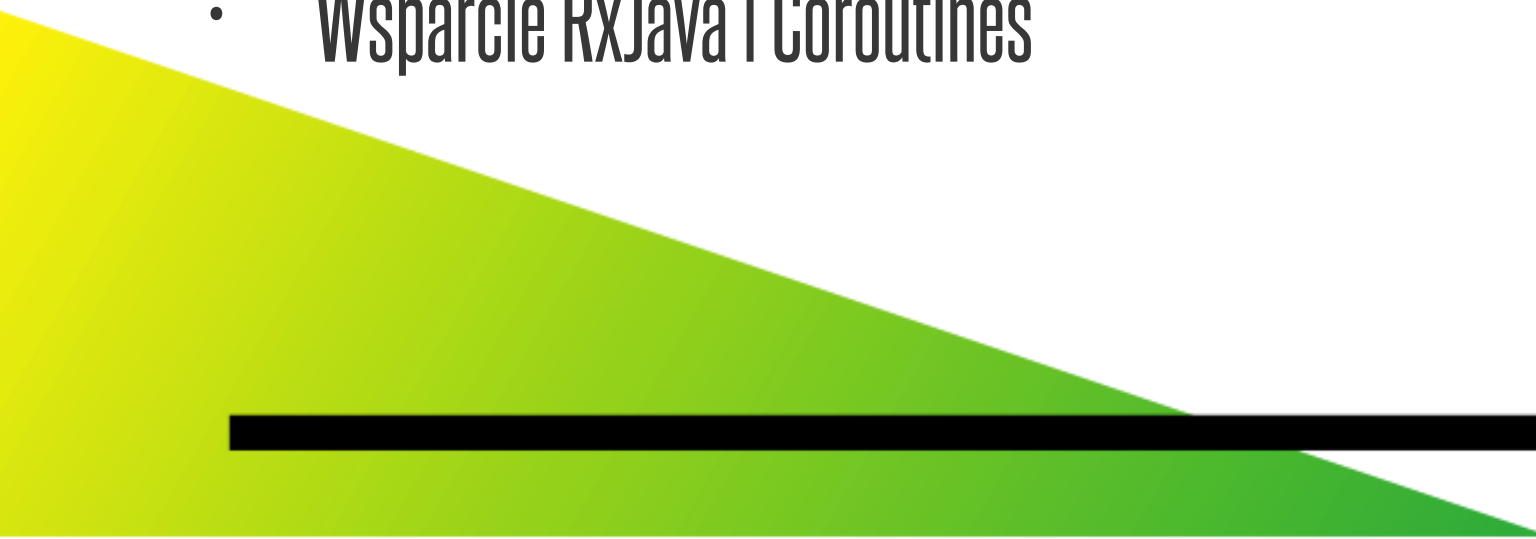
AsyncTask

- klasa do wykonywania zadań w tle i publikowania wyników w MainThread
 - `onPreExecute()`
 - `doInBackground(Params...)`
 - `onProgressUpdate(Progress...)`
 - `onPostExecute(Result)`
- 
-

@Deprecated AsyncTask

- deprecated od API 30 (Android 11)
 - wycieki pamięci
 - Executor (single, pool)
 - not aware of Lifecycle
 - raczej krótkie zadania
- 
-

Retrofit - łatwy dostęp do API

- Interface + Annotacje = Klient API
 - Converter (Gson, Moshi, Scalars)
 - Interceptor
 - Sync i Async (enqueue() vs execute())
 - Wsparcie RxJava i Coroutines
- 
-

GSON

- mapowanie POJO <-> JSON
 - convention over configuration
 - @SerializedName
 - pola transient
 - własne adaptory (serializer/deserializer)
 - GsonConverterFactory
-

Callbacks

- callback hell
 - trudna analiza kodu
 - brak jednego miejsca do obsługi błędów
 - wycieki pamięci
 - anulowanie operacji
 - brak powiązania z lifecycle
-

Zadanie 1

Korzystając z AsyncTask napisz kod pobierający listę planet z API przy użyciu synchronicznej metody `execute()`.

Wyniki wyświetl w wątku Main.



Kotlin Coroutines

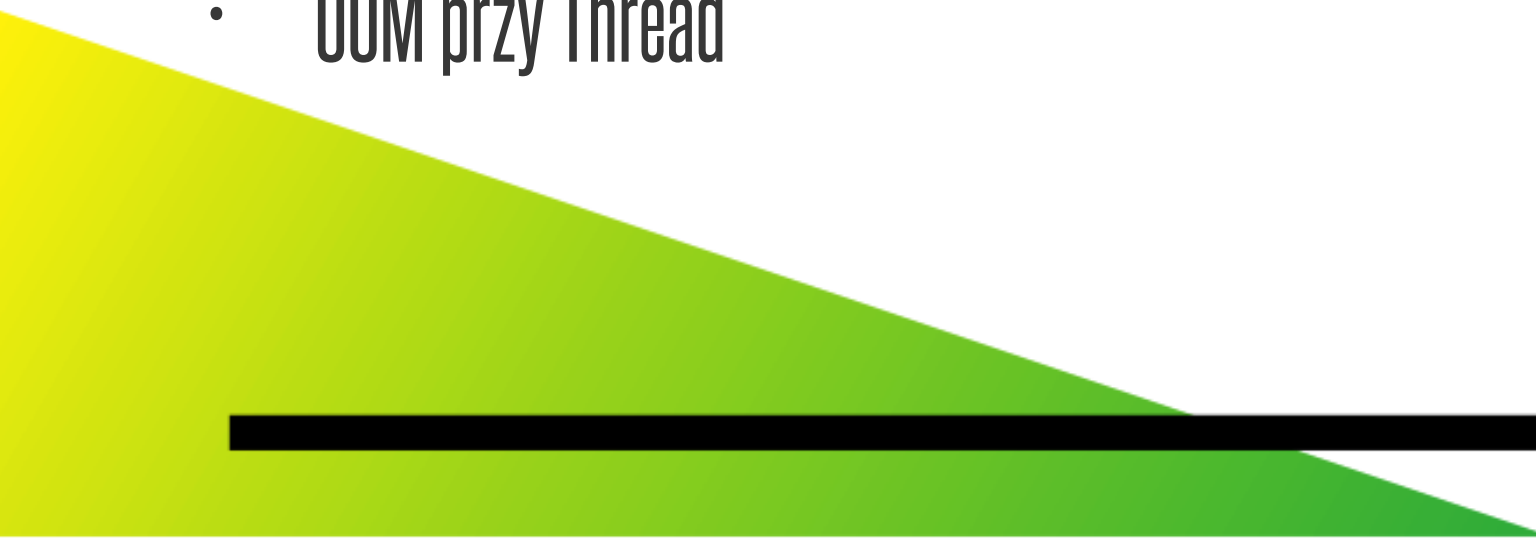
Coroutine - co to?

- Routine vs Cooperative Routine
- "lekke wątki"
- generowany kod

Suspend functions

- pozwalają wykonywać długie operacje
- mogą być wstrzymywane
- punkty wstrzymania
- użycie: wewnątrz suspend fun albo coroutine


Coroutine vs Thread

- dużo lżejsze
 - coroutine używa pod spodem wątków (często main)
 - `Thread.currentThread().name`
 - bez problemu możemy stworzyć 1M coroutines
 - OOM przy Thread
- 
-


Delay vs Thread.sleep

- suspend function
- nie blokuje wątku
- korzysta z różnych implementacji zależnie do Dispatchera np. `postDelayed()`

Przełączanie wątku (Dispatcher)

- `withContext()`
 - Dispatchers:
 - Default (Thread Pool, CPU Cores (min 2))
 - IO (Thread Pool, max 64)
 - Main (Android only)
- 
-


Mniej popularne

- `Dispatchers.Unconfined (current thread)`
 - `newSingleThreadContext`
 - `newFixedThreadPoolContext`
 - `Executor.asCoroutineDispatcher()`
- 

Coroutine builders

- `launch`
- `async`
- `runBlocking`

Launch

- pozwala na uruchomienie korutyny
 - przyjmuje `CoroutineContext`
 - zwraca `Job`
 - `Job.join()` - blokuje
 - nie pozwala zwracać wyniku
- 
-

Coroutines z Retrofit

- przerobienie interfejsu
- GlobalScope
- lifecycleScope
- viewModelScope

Main Safety

- suspend fun nie blokuje wątku, z którego jest wołana
- zwykle Main




Error handling

- try / catch

Zadanie 2

Pobierz listę planet i innych obiektów niebieskich (sekwencyjnie) i wyświetl je wszystkie na liście.

Async

- `async` odpala korutynę
 - zwraca `Deferred<T>` (podtyp `Job` z rezultatem)
 - wykonywanie zapytań równocześnie
 - `suspend fun await()`, `awaitAll()` (join, ale z wynikiem)
 - `try / catch` na `await()`
- 
-

Zadanie 3

Pobierz listę księżyców dla każdej z planet. Wykonaj te zapytania równoległe, korzystając z `async/await`.

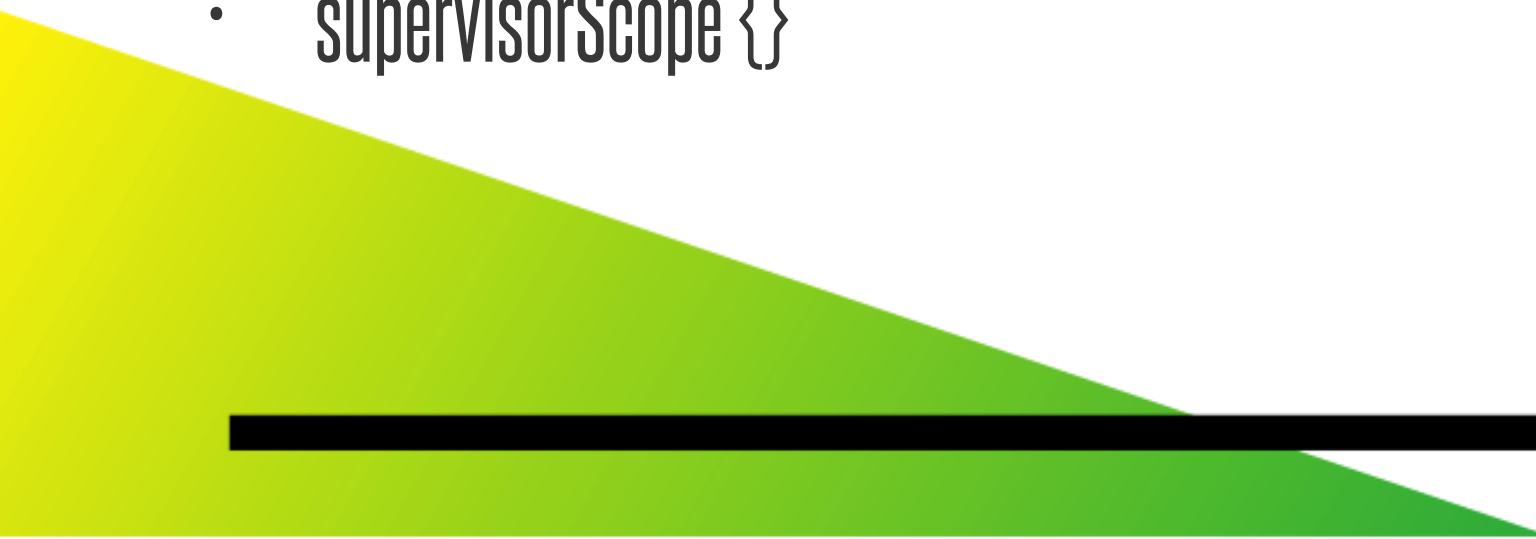
Coroutine Context

- Set:
 - Dispatcher
 - Job
 - ErrorHandler
 - Coroutine Name
 - `withContext()` pozwala zmienić Context
-

CoroutineScope vs CoroutineContext

- Scope zawiera CoroutineContext
- many coroutine contexts
- one scope


Whudowane Scope

- `viewModelScope`
 - `lifecycleScope`
 - `GlobalScope`
 - `coroutineScope {} = launch {}.join()`
 - `supervisorScope {}`
- 
-

viewModelScope

- Dispatchers.Main.Immediate + SupervisorJob
- viewModelScope.launch(context = ...)


Structured Concurrency

1. Korutyna ma swój scope i czas życia np. `viewModel`, `Activity.lifecycleScope`
 2. Korutyny w tym samym scope tworzą hierarchię (Job ma rodzica)
 3. Job rodzic zakończy się tylko jeśli zakończyły się jego dzieci.
 4. Anulowanie Joba anuluje dzieci.
 5. Exception w korutynie propagowane jest do rodzica i rodzeństwo jest anulowane (Job) lub nie (SupervisorJob).
- 
-

Anulowanie Korutyny

- `CancellationException`
- `job.invokeOnCompletion { }`

CoroutineExceptionHandler

- `try { launch { /blad/ } } catch {}` nie działa
 - `ExceptionHandler` tylko w scope albo top level
 - `CancellationException` nie trafia do `ExceptionHandler`
 - `ExceptionHandler` jest wołany po fakcie
- 

...ale

- `coroutineScope {}` rzuca exception
- przydatne do `suspend fun doSth() = coroutineScope {}`
- `supervisorScope {}` tego nie ma
- `launch` wewnątrz `supervisorScope` jest jak top level

Zadanie 4

Obsłuż błędy z pobierania planet korzystając z:

- try catch
- exception handlera

The logo for Kotlin Flow is a white rectangle with a thick black border. It is positioned in the upper middle of the image. The background consists of a white upper half and a lower half divided diagonally from the bottom-left to the top-right. The bottom-left portion is yellow, and the bottom-right portion is green.

Kotlin Flow

Co to jest Flow?

- asynchroniczny strumień danych
- implementacja Reactive Programming (Observe Pattern)
- Sequence vs Flow
- LiveData vs Flow

Flow = Cold Observable

- staje się aktywny przy obserwowaniu
- kończy się gdy przestaje być obserwowany
- osobna emisja dla każdego obserwatora



Flow Builders

- `flowOf()`
- `.asFlow()`
- `flow { } i emit()`

Terminal Operators

- bez operatora końzonego flow nie emituje
- `collect()` - nie kończy się aż strumień się zamknie
- `first()`, `firstOrNull()` - kończy po 1 emisji
- `last()` - czeka do końca
- `single()` - `IllegalArgumentException` jeśli więcej niż 1
- `toSet()`, `toList()` - czeka do końca

launchIn i onEach

- `fun Flow.launchIn(scope): Job`
- `onEach { item }`
- nie jest suspend, więc nie blokuje

Lifecycle Operators

- `onStart {}`
- `onCompletion { cause: Throwable? }`
- `pipeline, upstream, downstream`

Konwersja Flow -> LiveData

- `.asLiveData()`

Intermediate Operators

- See: flowmarbles.com
- map - może zmieniać typ
- filter, filterNotNull, filterIsInstance
- take, takeWhile - zamyka flow po N emisjach lub warunku
- drop, dropWhile
- transform - map ale można emitować ile się chce
- ~~distinctUntilChanged~~

Error Handling

- try / catch na collect
- collect zbiera błędy z operatorów
- catch {} - operator

Flow do UIState

- emituje nawet gdy apka jest w tle
- UI próbuje renderować w tle
- wiele obserwacji = duplikacja emisji
- zmiana konfiguracji = restart flow
- można poprawić startując/zatrzymując joba w onStart/onStop

repeatOnLifecycle


- `repeatOnLifecycle(Lifecycle.State.STARTED)`
- `.flowWithLifecycle(lifecycle, Lifecycle.State.STARTED)`

SharedFlow = Hot Observable

- emituje niezależnie od obserwatorów
- zostaje aktywny nawet jeśli znikną obserwatorzy
- SharedFlow może być współdzielony pomiędzy obserwatorami

SharedFlow

```
val sharedFlow = MutableSharedFlow<UiState>()  
  
flow {}.shareIn(  
    scope = viewModelScope,  
    started = Eagerly, Lazily, WhileSubscribed(5000),  
    replay = 0 (1 = StateFlow)
```



StateFlow

- zoptymalizowany SharedFlow
- idealny do publikowania UIState
- musi mieć początkową wartość
- emituje tylko nowe wartości (jak distinctUntilChanged)
- stateFlow.value - pobierz i ustaw wartość bez suspend
- stateFlow.update { old-> new }
- stateIn{



Widoki w Androidzie

Przegląd podstawowych widoków (Views) i ich właściwości

- LinearLayout
- FrameLayout (FragmentManager)
- RelativeLayout

Wprowadzenie do ConstraintLayout

- Dlaczego korzystać z ConstraintLayout
- Zrozumienie koncepcji constraintów



Zaawansowane techniki ConstraintLayout

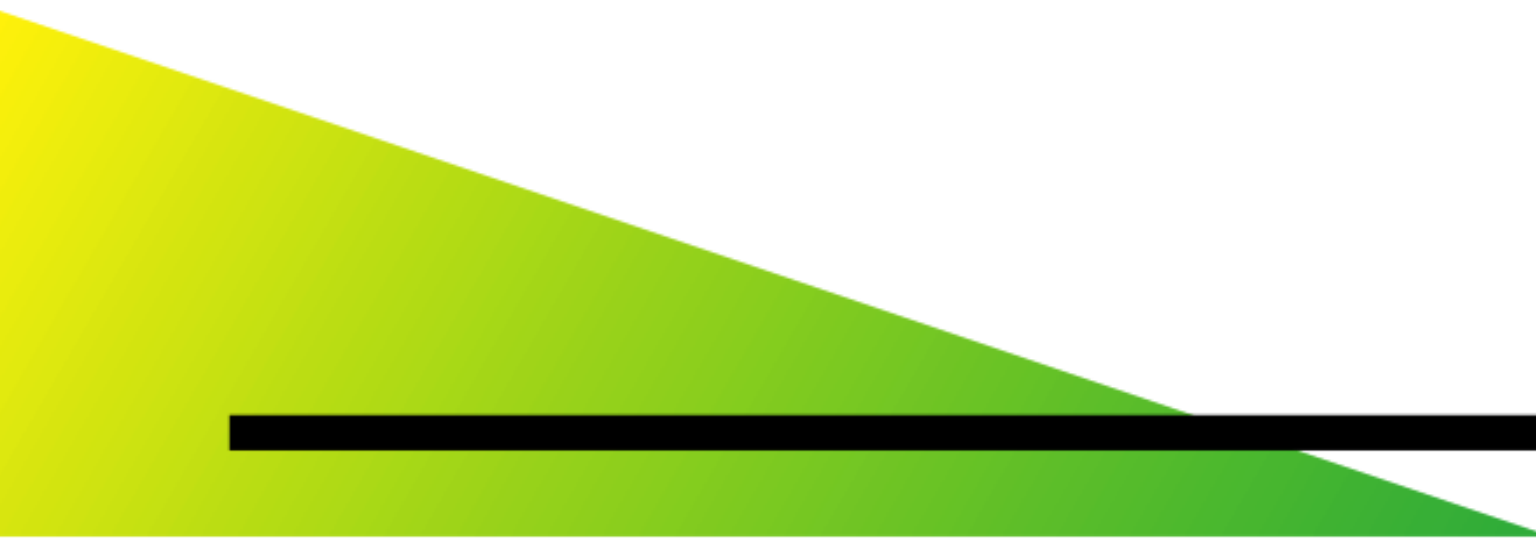
- Grupy i układy łańcuchowe (Chains)
- Wyrównanie (Guidelines) i Spacers
- Obsługa wielu punktów zaczepienia (Bi-directional Constraints)
- Przestrzenie wypełniające (Bias) i Prioritety

Material Design i widoki adaptacyjne

- Czym jest material design?
- <https://m3.material.io/>

Przygotowanie responsywnych i adaptacyjnych layoutów

- Użycie CardView, FloatingActionButton i innych komponentów Material Design



Wykorzystanie CoordinatorLayout i AppBarLayout

- Współpraca z CollapsingToolbarLayout



Tworzenie własnych widoków (Custom View)

- Personalizowanie wyglądu i zachowania widoków
- Wydajność i optymalizacja własnych widoków



Tworzenie layoutów dla różnych urządzeń

- Obsługa wielu rozdzielczości ekranu
- Wykorzystanie zasobów na różne wymiary ekranu (dimens, drawables itp.)
- Importowanie Vector Drawable z plików SVG - ograniczenia i możliwe rozwiązania
- Identyfikowanie i rozwiązywanie problemów związanych z layoutami






Wstęp do testów na Androida

Testy są jak siłownia

- każdy mówi, że warto
- rzadko kto chodzi



Dlaczego warto testować?

- zapobieganie regresji
 - jakość kodu
 - siatka bezpieczeństwa przy refactoringu
 - lepsza architektura
- 

Testy to koszt

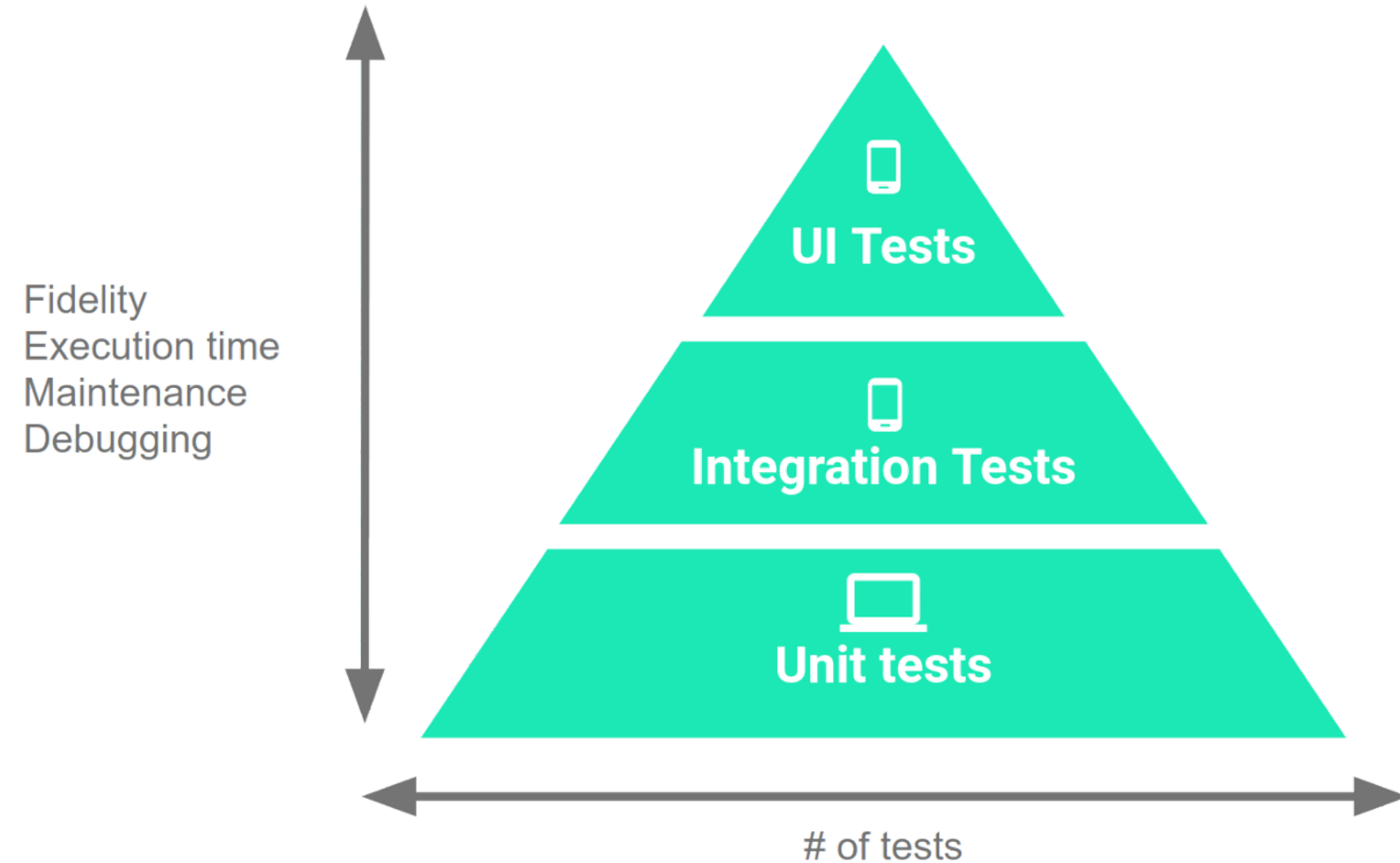
- napisanie testu
- aktualizacja testu
- poprawianie starych testów
- brak elastyczności przy źle napisanych testach

Nasze testy powinny...


- być szybkie
- pokrywać dużą część kodu
- testować wszystkie opcje
- wybierz 2 ;)

Typy testów na Androidzie

- test jednostkowy (Unit Test)
- testy integracyjne (Robolectric)
- testy UI (instrumentacyjne)



Dobry test

- Arrange - Act - Assert
 - Given - When - Then
 - jeden assert na test `assertEquals(4, 2 + 2)`
 - fluent assertions np. `assertThat(2 + 2).isEqualTo(4)`
- 
-

Testy Jednostkowe

- testujemy publiczne api
 - nie testujemy implementacji
 - doskonałe do testowania logiki
 - super szybkie
 - nie dotykamy dysku i sieci
 - najlepiej jeśli nie dotykają Androida
-

Test Driven Development

- nie piszemy kodu bez testów
- architektura powstaje "sama"
- Red-Green-Refactor

Red

- wybierz kolejny test
- napisz test, aby zbliżyć się do celu
- spraw aby się kompilował

Green

- minimalny fix (fake jeśli potrzeba)
- sprawdź czy wszystkie testy przeszły



Refactor

- usun powtórzenia
- dodaj wzorce projektowe
- cały czas green
- testy tez sie czysci

Test Doubles

- fake
- stub
- mock

Zasady Kata

- pamięć mięśniowa
- Pair Programming
- Ping Pong
- Nie piszemy kodu nie wymaganego przez testy

Kata

<http://codekata.com/kata/kata09-back-to-the-checkout/>

```
class TestPrice < Test::Unit::TestCase

  def price(goods)
    co = CheckOut.new(RULES)
    goods.split(/,/).each { |item| co.scan(item) }
    co.total
  end

  def test_totals
    assert_equal( 0, price(""))
    assert_equal( 50, price("A"))
    assert_equal( 80, price("AB"))
    assert_equal(115, price("CDBA"))

    assert_equal(100, price("AA"))
    assert_equal(130, price("AAA"))
    assert_equal(180, price("AAAA"))
    assert_equal(230, price("AAAAA"))
    assert_equal(260, price("AAAAAA"))

    assert_equal(160, price("AAB"))
    assert_equal(175, price("AABB"))
    assert_equal(190, price("AABBD"))
    assert_equal(190, price("DABABA"))
  end

  def test_incremental
    co = CheckOut.new(RULES)
    assert_equal( 0, co.total)
    co.scan("A"); assert_equal( 50, co.total)
    co.scan("B"); assert_equal( 80, co.total)
    co.scan("A"); assert_equal(130, co.total)
    co.scan("A"); assert_equal(160, co.total)
    co.scan("B"); assert_equal(175, co.total)
  end
end
```

Testowanie Korutyn

- <https://developer.android.com/kotlin/coroutines/test>
groovy
testImplementation
"org.jetbrains.kotlinx:kotlinx-coroutines-test:
\$coroutines_version"
- `runTest {} = TestScope`

Test Dispatchers

- `StandardTestDispatcher`
- `UnconfiedTestDispatcher`
- `TestCouroutineScheduler` musi być współdzielony w teście
- wstrzykiwanie dispatchera

StandardTestDispatcher

- advanceUntillIdle
- advanceTimeBy
- runCurrent

Replace Dispatchers.Main

```
// Reusable JUnit4 TestRule to override the Main dispatcher
class MainDispatcherRule(
    val testDispatcher: TestDispatcher = UnconfinedTestDispatcher(),
) : TestWatcher() {
    override fun starting(description: Description) {
        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description) {
        Dispatchers.resetMain()
    }
}
```

Testowanie Flow

- <https://developer.android.com/kotlin/flow/test>
- `StateFlow.value`
- <https://github.com/cashapp/turbine>

Testy Instrumentacyjne

- uruchamiane na urządzeniu (osobna apka)
 - startują wybrane Activity
 - testują integrację komponentów
 - mogą przechodzić przez wiele ekranów
 - są powolne, więc stosujemy wiele asercji
 - odpalanie z AS i przez skrypt
-


Espresso

- bardzo duże możliwości
- dziwna składnia Hamcrest
- wymaga konfiguracji urządzenia (np. wyłączenia animacji)


Espresso

```
@Test
fun changeText_newActivity() {
    // Type text and then press the button.
    onView(withId(R.id.editTextUserInput)).perform(typeText(String_TO_BE_TYPED),
        closeSoftKeyboard())
    onView(withId(R.id.activityChangeTextBtn)).perform(click())

    // This view is in a different Activity, no need to tell Espresso.
    onView(withId(R.id.show_text_view)).check(matches(withText(String_TO_BE_TYPED)))
}
```



Robot

- Page Object Pattern
 - separacja logiki testów od ich implementacji
 - NeoRobot
 - generowany przez template
 - reużycie kroków
- 
-

TDD w Androidzie

- pierwszy test sprawdza setup
- potem testowanie API na podstawie dokumentacji
- można testować dowolne Edge Case



Continuous Integration

- buduje aplikację po każdym commit
- odpala testy jednostkowe
- może odpalać testy UI (wymaga emulatora)

