# B.4 – Testing: Unit Test

## Introduzione a Unit Test su Android

L’argomento del testing su Android è particolarmente complesso. Durante la creazione di un’app ci ritroviamo ad integrarci pesantemente con il framework, utilizzando classi al di fuori del nostro controllo, il che rende il testing difficile – a tratti impossibile – se non gestiamo correttamente il codice.

Android ci fornisce numerose librerie costruite solo allo scopo di permetterci di effettuare test.

In questa sezione del corso ci occuperemo di **Unit Test**: test automatici che permettono di valutare l’esito di una specifica funzione, che sono eseguibili sulla macchina di sviluppo in tempi molto rapidi.

## Setup Progetto

Per prima cosa, importiamo le librerie di test. Nel file build.gradle (app), sostituire gli import con la dicitura *testImplementation* con le seguenti righe:

//Unit tests dependencies  
testImplementation "junit:junit:4.13.2"  
testImplementation "com.google.truth:truth:1.1.2"  
testImplementation "androidx.test.ext:junit-ktx:1.1.3"  
testImplementation "androidx.test:core-ktx:1.4.0"  
testImplementation "org.robolectric:robolectric:4.5.1"  
testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.0"

Questo è un setup generico che coprirà gli Unit test per tutto il corso.

Il progetto richiederà ora un *gradle sync*

*Text

Description automatically generated*

## Creazione di un test

Aggiungiamo il seguente codice a MainActivity:

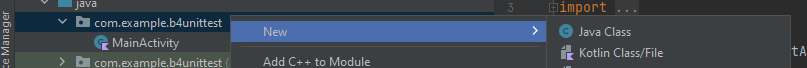
Text

Description automatically generated

Ora vogliamo testare se performSum funziona correttamente.

La nostra funzione però è contenuta in *MainActivity,* sarebbe quindi necessario instanziare tutta la classe – con annessi tutti i processi Android sottostanti - per potervi accedere. Il che equivarrebbe a testare tutto Android!   
Per evitare di instanziare *MainActivity* durante il test, andiamo ad estrarre la funzione in una classe a sé stante.

Crea una nuova *Kotlin Class* chiamata *MathHelper* nel tuo package, dove sposterai la funzione performSum rendendola pubblica:



A screenshot of a computer

Description automatically generated with medium confidence

Ora creiamo il file di test. Tasto destro sul nome della classe > Content actions (o *alt-invio*) > Create Test

Graphical user interface, application

Description automatically generated

Seleziona JUnit4 come framework di test, e segna la spunta a tutte le funzioni da testare > OK.  
Infine, assicurati che il file venga creato sotto la cartella Unit Test > OK.

A screenshot of a computer

Description automatically generated with medium confidence

Ora il nostro file di test è stato generato. Dobbiamo però fare alcune modifiche:

Negli import, sostituisci *org.junit.Assert.\** con:

* *com.google.common.truth.Truth.\**

Nella riga prima di ***class***, aggiungi:

* @RunWith(AndroidJUnit4::class)

Text

Description automatically generated

## Naming Convention

La convenzione dei nomi dei test è la seguente:

functionName\_initialStatus\_expectedResult

Modifichiamo quindi il nome del nostro test in:

Text

Description automatically generated

Lo scopo del nostro test sarà quindi di vedere se la funzione *performSum* produce il risultato corretto quando gli forniamo in input due numeri positivi.

## Assertions

Il concetto di un test è comparare un valore o un oggetto ad un risultato che ci si aspetta. Se questa *affermazione(assertion)* produrrà un esito positivo *true*, il test sarà un successo (*passed)*. Ad esempio:

*“asserisco che una macchina ha 4 ruote.”*

Se la macchina uscirà dalla fabbrica con 3 ruote, non passerà il nostro test.

Per questa procedura si usano degli strumenti di comparazione. Noi useremo la libreria *Truth* di google, con le sue assertions *assertThat*, che permette di scrivere comparazioni in maniera tale che siano facilmente comprensibili.

## Successo e Fallimento

Scriviamo il test:

Text

Description automatically generated

Sarà la linea *assertThat* a definire la condizione per cui il soggetto passerà o no il test.

Lanciamo il test con la freccia verde a sinistra del nome del test

Nella tab in basso, selezionare le spunte *show passed* e *ignored* per mostrare tutti i test eseguiti

Graphical user interface, text, application

Description automatically generated

Il nostro primo test è passato. Vediamo cosa succede se il test fallisce:

Text

Description automatically generated

Lanciando il test, potremo vedere quale è il test che non è passato, e leggere nel log la condizione che ha fallito:

A screenshot of a computer

Description automatically generated

## Strategie di test

E’ buona norma valutare tutte le possibilità a cui la nostra funzione può essere sottoposta, con una serie di casistiche:

Text

Description automatically generated

Se necessario, per ogni test è possibile inserire più condizioni, aggiungendo altre righe *assertThat*. La prima condizione non rispettata farà fallire il test.  
È importante però mantenere queste condizioni strettamente all’interno dello scopo del test, per non falsarne i risultati:

Text

Description automatically generatedEsempio corretto: stiamo valutando se lo stato iniziale del test è rispettato (numero positivo)

Text

Description automatically generatedEsempio scorretto: sebbene il test passi, non è pertinente mettere come condizione che testNumber debba essere numero primo.

## @Before, @After

Il nostro test vira intorno al fatto di testare un singolo *test subject,* in questo caso la classe *MathHelper* e le sue funzioni. Sebbene la nostra classe sia ora molto leggera, potrebbe nel tempo diventare molto complessa, anche nel costruttore. Andiamo quindi a inizializzarla una singola volta per questa serie di test:

Text

Description automatically generated

Tutto funziona, ma c’è un problema. MathHelper viene creato una sola volta, e per i nostri test questa cosa non è ideale: se il suo stato interno, ad esempio delle variabili, dovessero cambiare, i nostri test potrebbero fallire. Ad esempio il primo test potrebbe modificare una variabile che al secondo test serve sia quella di default. Scrivendo i test, ci rapportiamo ad uno specifico stato del test subject, che di solito è lo stato iniziale.

Modifichiamo MathHelper per fornigli uno stato:

Text

Description automatically generated

Ora cambiamo l’inizializzazione di MathHelper nel test

Graphical user interface

Description automatically generated with low confidence

La funzione taggata con @Before sarà lanciata prima di ogni test.

Aggiungiamo una coppia di test per la nuova funzionalità:

Text

Description automatically generated

Se avessimo mantenuto mathHelper inizializzato come val una volta sola, saremmo potuti incombere in problemi con dei test che analizzano lo stato *recentResults.*

Sebbene il framework di test cerchi automaticamente di ricreare lo stato iniziale di tutti i soggetti ogni volta che effettua un @Test, è buona norma impostare sempre le inizializzazioni che vengono ripetute dentro la funzione @Before

Una funzione taggata **@After** ha lo stesso concetto: verrà lanciata dopo ogni test.

Graphical user interface

Description automatically generated

## Esercizio: test Lancio dadi

Nell’app Lancio dadi:

* Estrai le funzioni di lancio dalla main activity in una classe specifica *DiceRoller*
* Aggiungi a *DiceRoller* la possibilità di avere uno stato che rappresenta quante facce ha il dado da lanciare. Modifica le funzioni di conseguenza, per integrare questo stato.
* Crea la classe di test con tutte le funzioni
* Instanzia il test subject correttamente, utilizzando @Before o @After se necessario
* Testa tutti i casi che ritieni validi, in particolare gli edge-cases