

Nome:

Número:

Turma:

### Código de Honra

A vida académica é o preâmbulo da vida profissional. A adesão às regras de conduta é uma responsabilidade social, ou seja, é responsabilidade de todos. A participação na comunidade académica pressupõe a adesão a um código de honra que exige respeito pelo trabalho próprio e pelo trabalho dos demais (colegas e docentes). Esse código de honra proíbe liminarmente o plágio, simplesmente porque é socialmente inaceitável.

Para que possa concluir a avaliação de PDM tem que subscrever de forma explícita, e sob compromisso de honra, a autoria das respostas que entregar. A ausência de assinatura implica que a prova não será aceite.

Eu, abaixo assinado, declaro por minha honra que as respostas abaixo são de minha exclusiva autoria. Mais declaro que durante a prova apenas usei elementos de consulta autorizados.

**Assinatura:**

### Enunciado

Considere a plataforma Android estudada nas aulas da disciplina e responda às perguntas seguintes assinalando de forma inequívoca a opção mais correta. Não responda arbitrariamente: cada resposta incorreta desconta 1/3 da cotação da pergunta ao total obtido na prova.

1. O ficheiro de manifesto de uma aplicação Android
  - ☐ é incluído no APK gerado pelo procedimento de *build*
  - ☐ é usado para indicar as permissões necessárias à aplicação
  - ☐ contém a indicação das *activities* existentes na aplicação
  - ☒ Todas as anteriores
2. Considere a seguinte implementação de *SomeActivity* que apresenta numa *TextView* (com id *txtMessage*) o texto recebido no momento da sua activação.

```
class SomeActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_some)  
        txtMessage.text = intent.getStringExtra("MESSAGE")?: "Nothing" // (A)  
    }  
}
```

- 2.1. Para que o texto recebido seja correctamente apresentado, mesmo havendo reorientações do ecrã,
  - ☐ a implementação teria que fazer uso de um *view model*
  - ☒ a implementação apresentada é suficiente
  - ☐ a iniciação da *TextView* com o texto recebido teria que ser feita na descrição do *layout* da *activity*
  - ☐ Nenhuma das anteriores
- 2.2. Por observação da linha de código assinalada com // (A), conclui-se que
  - ☐ a *activity* só poderá ser activada através de *intent* implícito
  - ☐ se não há erro de execução, foi declarado um *intent filter* no ficheiro de manifesto
  - ☒ se não há erro de compilação, existe um controlo gráfico com o id *txtMessage* no *layout* usado
  - ☐ Todas as anteriores

3. Dada a ActivityA que se apresenta de seguida:

```
class ViewModelA(var value: Int) : ViewModel()
class ActivityA : AppCompatActivity() {
    private val liveData = MutableLiveData<Int>().apply { this.value = 10 }
    private val viewModel = ViewModelA(10)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        liveData.observe(this, Observer {
            Log.v("ActivityA", "liveData = $it") // (1)
        })
        Log.v("ActivityA", "liveData = ${liveData.value}") // (2)
        Log.v("ActivityA", "viewModel = ${viewModel.value}") // (3)
        liveData.value = 11
        liveData.postValue(12)
        liveData.value = 13
        viewModel.value = 13
    }
}
```

- 3.1. Para a sequência de acontecimentos: “ActivityA é lançada pela primeira vez” → “ecrã do dispositivo é rodado”, as últimas duas mensagens escritas em *log* nas linhas // (2) e // (3) contêm, respectivamente:
- ☒ liveData = 10 e viewModel = 10
  - ☐ liveData = 13 e viewModel = 13
  - ☐ liveData = 10 e viewModel = 13
  - ☐ Nenhuma das anteriores
- 3.2. Quando a ActivityA é lançada pela primeira vez, a sequência de valores contidos nas mensagens escritas em *log* na linha // (1) é:
- ☐ liveData = 11; liveData = 12; liveData = 13
  - ☒ liveData = 13; liveData = 12
  - ☐ liveData = 12; liveData = 13
  - ☐ liveData = 10; liveData = 11; liveData = 12; liveData = 13
4. A framework Room gera automaticamente implementações das interfaces anotadas com @Dao. Essas interfaces definem os contratos dos métodos de acesso a dados e estão anotados de acordo com a natureza da operação (i.e. @Query, @Insert, @Update, @Delete). A execução destes métodos ...
- ☐ é síncrona
  - ☐ é assíncrona
  - ☒ poderá ser síncrona ou assíncrona, dependendo do tipo de retorno
  - ☐ será síncrona para os métodos anotados com @Query e assíncrona para os restantes
5. Para uma instância de RecyclerView.Adapter, o número de chamadas a onCreateViewHolder é...
- ☐ igual ao número de elementos da coleção a ser apresentada
  - ☐ maior ou igual ao número de elementos da coleção a ser apresentada
  - ☐ sempre igual ao número de chamadas a onBindViewHolder
  - ☒ Nenhuma das anteriores
6. A execução de uma tarefa por via de um *Foreground Service* lançado a partir de uma *activity*:
- ☒ exige que esse serviço seja declarado no manifesto da aplicação
  - ☐ terminará quando essa *activity* deixar de estar visível
  - ☐ é equivalente à execução dessa tarefa por via de uma *AsyncTask*.
  - ☐ Todas as anteriores

7. Dada uma aplicação Android composta pelas *activities* apresentadas de seguida:

```
const val TAG: String = "TAG"
abstract class BaseActivity(private val name: String) : AppCompatActivity() {
    override fun onCreate(s: Bundle?) { super.onCreate(s); Log.v(TAG, "$name onCreate") }
    override fun onStart() { super.onStart(); Log.v(TAG, "$name onStart") }
    override fun onStop() { super.onStop(); Log.v(TAG, "$name onStop") }
    override fun onDestroy() { super.onDestroy(); Log.v(TAG, "$name onDestroy") }
}
class ActivityA : BaseActivity("A") {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_a)
        button.setOnClickListener { startActivity(Intent(this, ActivityB::class.java)) }
    }
}
class ActivityB : BaseActivity("B") {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_b)
    }
}
```

- 7.1. Para a sequência de acontecimentos: “ActivityA é lançada pela primeira vez” → “utilizador prime botão *button*”, a sequência de mensagens escritas em *log* é:
- ☒ A onCreate; A onStart; B onCreate; B onStart; A onStop
  - ☐ A onCreate; A onStart; B onCreate; B onStart
  - ☐ A onCreate; A onStart; B onCreate; B onStart; A onStop; A onDestroy
  - ☐ A onCreate; B onCreate; A onStart; B onStart;
- 7.2. Para a sequência de acontecimentos: “ActivityA é lançada pela primeira vez” → “utilizador prime botão *button*” → “utilizador selecciona outra *user task*”, a sequência de mensagens escritas em *log* é:
- ☐ A onCreate; A onStart; B onCreate; B onStart; A onStop
  - ☐ A onCreate; A onStart; B onCreate; B onStart; A onStop; B onStop; A onDestroy; B onDestroy
  - ☒ A onCreate; A onStart; B onCreate; B onStart; A onStop; B onStop
  - ☐ A onCreate; A onStart; B onCreate; B onStart; A onStop; A onDestroy
- 7.3. Para a sequência de acontecimentos: “ActivityB está visível” → “utilizador volta para a *activity* inicial (prime *back*)”, a sequência de mensagens escritas em *log* é:
- ☐ B onDestroy; A onStart
  - ☐ B onStop; A onStart
  - ☐ B onStop; B onDestroy
  - ☒ B onStop; B onDestroy; A onStart
8. De acordo com o guia de arquitectura Android Jetpack, o *ViewModel*
- ☐ determina se obtém dados de uma API remota ou da base de dados local, dirigindo os pedidos para o componente adequado
  - ☒ deve ser específico à *activity* porque fornece os dados necessários à mesma
  - ☐ deve ser geral, para ser usado em várias *activities*
  - ☐ representa os dados a armazenar na base de dados
9. Considere uma instância de `com.android.volley.RequestQueue`. A chamada ao seu método `public fun <T> add(request: Request<T>): Request<T>` no *handler* de evento de *click* de um botão ...
- ☐ lança excepção caso ocorra um erro no acesso à rede
  - ☐ só pode ser realizada se a aplicação declarar a permissão `ALLOW_REQUESTS_IN_UI_THREAD`
  - ☒ é permitida porque não é bloqueante
  - ☐ não é permitida, porque vai resultar num acesso à rede

10. Considere uma aplicação Android que inclua a ActivityA apresentada de seguida:

```
class ViewModelA(var value: Int = 0) : ViewModel()
class ActivityA : AppCompatActivity() {
    private lateinit var counter1: ViewModelA
    private var counter2: Int = 0
    override fun onCreate(state: Bundle?) {
        super.onCreate(state)
        setContentView(R.layout.activity_a)
        counter1 = ViewModelProviders.of(this).get(ViewModelA::class.java)
        counter2 = state?.getInt("counter2") ?: 0
        Log.v("ActivityA", "counter1 = ${++counter1.value}")
        Log.v("ActivityA", "counter2 = ${++counter2}")
    }
    override fun onSaveInstanceState(state: Bundle) {
        super.onSaveInstanceState(state)
        state.putInt("counter2", counter2)
    }
}
```

10.1. Para a sequência de acontecimentos: “ActivityA é lançada pela primeira vez” → “ecrã do dispositivo é rodado”, as últimas duas mensagens escritas em *log* contêm:

- ☐ counter1 = 1 e counter2 = 1
- ☐ counter1 = 1 e counter2 = 2
- ☒ counter1 = 2 e counter2 = 2
- ☐ counter1 = 2 e counter2 = 1

10.2. Para a sequência de acontecimentos: “ActivityA é lançada pela primeira vez” → “utilizador selecciona outra *user task*” → “processo hospedeiro é terminado” → “utilizador selecciona de novo a *user task* da ActivityA”, as últimas duas mensagens escritas em *log* contêm:

- ☐ counter1 = 1 e counter2 = 1
- ☒ counter1 = 1 e counter2 = 2
- ☐ counter1 = 2 e counter2 = 2
- ☐ counter1 = 2 e counter2 = 1

11. Considerando a API *WorkerManager* e dada a seguinte definição de *Someworker*:

```
class Someworker(ctx: Context, args: WorkerParameters) : Worker(ctx, args) {
    private val queue = Volley.newRequestQueue(ctx)
    override fun doWork(): Result {
        return try {
            queue.add(StringRequest(Request.Method.GET, "www.example.com",
                Response.Listener<String> { /* process response */ },
                Response.ErrorListener { throw it }
            ))
            Result.success()
        } catch (error: VolleyError) { Result.failure() }
    }
}
```

Após análise da implementação conclui-se que...

- ☐ a implementação está correcta
- ☐ para que esteja correcta, o método `doWork()` tem que retornar sempre `Result.success()`
- ☐ para que esteja correcta, a chamada a `queue.add()` tem que ser executado numa *thread* alternativa
- ☒ Nenhuma das opções anteriores