

Programação em Dispositivos Móveis

Teste Global de Época Normal, Inverno de 2021/2022

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 do 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 correta. Não responda arbitrariamente: cada resposta incorreta desconta 1/3 da cotação da pergunta ao total obtido na prova.

1. Uma das utilidades do ficheiro de manifesto de uma aplicação Android é:
 - ☐ definir o idioma em que a aplicação é apresentada
 - ☐ definir se a aplicação utiliza Java ou Kotlin
 - ☐ indicar as permissões necessárias à aplicação
 - ☐ todas as outras opções
2. O uso de *resources* em Android tem a consequência de:
 - ☐ reduzir o tamanho do APK gerado no procedimento de *build*
 - ☐ facilitar a adequação da aplicação à configuração do dispositivo onde está a ser executada
 - ☐ identificar os dispositivos alvo da aplicação
 - ☐ nenhuma das outras opções
3. A inclusão de instâncias de tipos definidos pela aplicação como *extras* de *intents* é possível se:
 - ☐ esses tipos cumprirem o contrato *Parcelable*
 - ☐ esses tipos forem anotados com *@Parcelize*, independentemente da sua definição
 - ☐ a sua definição apenas incluir tipos primitivos
 - ☐ nenhuma das outras opções
4. Para a correta resolução de um *intent* implícito é imprescindível:
 - ☐ a definição de, pelo menos, um *intent-filter* no manifesto da aplicação de destino
 - ☐ a criação do *intent* implícito com o nome completo da classe do componente de destino
 - ☐ que a aplicação de destino esteja ativa com o componente no estado *STARTED* ou *RESUMED*
 - ☐ todas as outras opções

5. Dada a aplicação Android composta pelas *activities* apresentadas de seguida:

```
const val TAG: String = "TAG"
abstract class BaseActivity(private val name: String) : AppCompatActivity() {
    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(s: Bundle?) {
        super.onCreate(s)
        setContentView(R.layout.activity_layout)
        findViewById<TextView>(R.id.name).text = "Activity A"
        findViewById<Button>(R.id.navigateToOther).setOnClickListener {
            startActivity(Intent(this, ActivityB::class.java))
        }
    }
}
class ActivityB : BaseActivity("B") {
    override fun onCreate(s: Bundle?) {
        super.onCreate(s)
        setContentView(R.layout.activity_layout)
        findViewById<TextView>(R.id.name).text = "Activity B"
        findViewById<Button>(R.id.navigateToOther).setOnClickListener {
            startActivity(Intent(this, ActivityA::class.java))
        }
    }
}
```

- 5.1. Para a sequência de acontecimentos: "ActivityA é lançada pela primeira vez" → "utilizador prime botão navigateToOther" → "utilizador prime botão back", o número de vezes que as mensagens aparecem em log é:
- ☐ "A onStart" = 1; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 0
 - ☐ "A onStart" = 1; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 1
 - ☐ "A onStart" = 2; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 1
 - ☐ "A onStart" = 2; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 0
- 5.2. Para a sequência de acontecimentos: "ActivityA é lançada pela primeira vez" → "utilizador prime botão navigateToOther" → "utilizador selecciona outra *user task*", o número de vezes que as mensagens aparecem em log é:
- ☐ "A onStart" = 1; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 0
 - ☐ "A onStart" = 1; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 1
 - ☐ "A onStart" = 2; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 1
 - ☐ "A onStart" = 2; "A onStop" = 1; "B onStart" = 1; "B onDestroy" = 0
- 5.3. Para a sequência de acontecimentos: "ActivityA é lançada pela primeira vez" → "utilizador prime botão navigateToOther" → "utilizador prime botão navigateToOther", e admitindo o comportamento por omissão na activação de *activities*, o número de vezes que as mensagens aparecem em log é:
- ☐ "A onCreate" = 1; "A onStart" = 1; "A onStop" = 1; "A onDestroy" = 0
 - ☐ "A onCreate" = 1; "A onStart" = 1; "A onStop" = 1; "A onDestroy" = 1
 - ☐ "A onCreate" = 2; "A onStart" = 2; "A onStop" = 1; "A onDestroy" = 1
 - ☐ "A onCreate" = 2; "A onStart" = 2; "A onStop" = 1; "A onDestroy" = 0
- 5.4. Por observação do código e sabendo que não há erros de compilação, conclui-se que
- ☐ a aplicação faz uso do suporte para *view binding*
 - ☐ ambas as *activities* partilham o mesmo *layout*
 - ☐ existem dois ficheiros de *layout* com o nome `activity_layout.xml`
 - ☐ todas as outras opções

6. Considerando a API *Worker Manager* e dada a seguinte definição de *SomeListenableWorker*:

```
fun syncFetchAndSave() {  
    // Synchronously fetches data from a remote API and stores it in a local DB  
    // Throws an exception if the operation failed  
}  
  
class SomeListenableWorker(app: Context, params: WorkerParameters)  
: ListenableWorker(app, params) {  
    override fun startWork(): ListenableFuture<Result> {  
        return CallbackToFutureAdapter.getFuture { completer ->  
            try { syncFetchAndSave(); completer.set(Result.success()) }  
            catch (e: Exception) { completer.setException(e) }  
        }  
    }  
}
```

Após análise da implementação conclui-se que...

- ☐ a implementação está correcta
- ☐ para que esteja correcta, o método `startWork()` tem que retornar sempre `Result.success()`
- ☐ para que esteja correcta, a chamada a `syncFetchAndSave()` tem que ser executada numa *thread* alternativa
- ☐ nenhuma das opções anteriores

7. Considerando a biblioteca *Retrofit* usada nas aulas e as seguintes definições:

```
data class SomeDTO(val data: String)  
  
interface SomeRetrofitService {  
    @GET("/")  
    fun getData(): Call<SomeDTO>  
}
```

7.1. A chamada a `getData()` a partir da *main thread*:

- ☐ lança excepção caso ocorra um erro no acesso à rede
- ☐ só é permitida se nenhuma *activity* da aplicação estiver visível
- ☐ não é permitida porque é bloqueante
- ☐ é permitida porque não é bloqueante

7.2. Pretende-se que o tipo *SomeDTO* represente o *payload* JSON das respostas HTTP produzidas pela API remota representada pela interface *SomeRetrofitService*. Para isso é necessário que:

- ☐ na iniciação da *framework* seja indicada uma biblioteca de codificação JSON
- ☐ a classe *SomeDTO* seja anotada com `@Parcelize`
- ☐ a classe *SomeDTO* apenas contenha propriedades do tipo *String*
- ☐ todas as anteriores

8. No modelo de programação disponibilizado pela biblioteca *Room* existem os seguintes elementos: *Database*, *Data Access Objects (DAO)* e *Entities*.

- ☐ DAOs são as classes definidas pelo programador e que representam os dados a armazenar na base de dados
- ☐ DAOs são as classes definidas pelo programador e que contêm a implementação do código relativo aos acessos à base de dados
- ☐ DAOs são as interfaces definidas pelo programador que caracterizam as operações de acesso a dados e cuja implementação é gerada em tempo de *build*
- ☐ nenhuma das anteriores

9. Dada a definição de TheActivity apresentada de seguida e o seu *view model* ...

```
class TheViewModel: ViewModel() {
    private val _stuff = MutableLiveData("")
    val stuff: LiveData<String> = _stuff
    fun fetchStuff() {
        // Fetches stuff (a string) from a remote API and publishes it
        // to the LiveData instance exposed through the stuff property
    }
    fun fetchMoreStuff(callback: (String) -> Unit) {
        // Fetches stuff (a string) from a remote API and publishes it
        // by calling the received callback
    }
}

class TheActivity : AppCompatActivity() {
    private val theViewModel: TheViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_the)
        val textView = findViewById<TextView>(R.id.theTextView)
        findViewById<Button>(R.id.fetchStuff).setOnClickListener { theViewModel.fetchStuff() }
        findViewById<Button>(R.id.fetchMoreStuff).setOnClickListener {
            theViewModel.fetchMoreStuff { textView.text = it }
        }
        theViewModel.stuff.observe(this) { textView.text = it }
    }
}
```

9.1. Para a sequência de acontecimentos: “TheActivity é lançada pela primeira vez” → “utilizador prime o botão fetchMoreStuff” → “ecrã do dispositivo é rodado”, e admitindo que os dados são obtidos com sucesso a partir da API remota, podemos afirmar que no final da sequência:

- ☐ os dados PODEM ou NÃO ser apresentados em textView, dependendo do tempo que demora a serem obtidos
- ☐ os dados PODEM ou NÃO ser apresentados em textView, independentemente do tempo que demora a serem obtidos
- ☐ os dados NUNCA são apresentados em textView, independentemente do tempo que demora a serem obtidos
- ☐ os dados são SEMPRE apresentados em textView, independentemente do tempo que demora a serem obtidos

9.2. Para a sequência de acontecimentos: “TheActivity é lançada pela primeira vez” → “utilizador prime o botão fetchStuff” → “ecrã do dispositivo é rodado”, e admitindo que os dados são obtidos com sucesso a partir da API remota, podemos afirmar que no final da sequência:

- ☐ os dados PODEM ou NÃO ser apresentados em textView, dependendo do tempo que demora a serem obtidos
- ☐ os dados PODEM ou NÃO ser apresentados em textView, independentemente do tempo que demora a serem obtidos
- ☐ os dados NUNCA são apresentados em textView, independentemente do tempo que demora a serem obtidos
- ☐ os dados são SEMPRE apresentados em textView, independentemente do tempo que demora a serem obtidos

- 9.3. Para a seguinte sequência de acontecimentos: “TheActivity é lançada pela primeira vez” → “utilizador prime o botão `fetchStuff`” → “é apresentado o resultado de `fetchStuff()`” → “utilizador prime o botão `fetchMoreStuff`” → “é apresentado o resultado de `fetchMoreStuff()`” → “ocorre uma reconfiguração”, podemos afirmar que:
- ☐ continua a ser apresentado o resultado de `fetchMoreStuff()`
 - ☐ passa a ser apresentado o valor inicial de `textView`
 - ☐ passa a ser apresentado o resultado de `fetchStuff()`
 - ☐ nenhuma das anteriores
- 9.4. De acordo com o modelo de programação Android e relativamente à relação entre o número de instâncias de `TheActivity` e do seu *view model* (`TheViewModel`), podemos afirmar que durante a execução da aplicação:
- ☐ existirão tantas instâncias de `TheActivity` como de `TheViewModel`
 - ☐ existirão tantas ou mais instâncias de `TheActivity` do que de `TheViewModel`
 - ☐ existirão tantas ou mais instâncias de `TheViewModel` do que de `TheActivity`
 - ☐ não é possível determinar a relação entre o número de instâncias de ambas

Duração: 40 minutos
ISEL, 30 de Janeiro de 2021