PassQueen
a.k.a dumpsqueen.com

Google Developers Certification - Associate Android Developer (Kotlin and Java Exam)

Google Associate-Android-Developer

Version Demo

Total Demo Questions: 17

Total Premium Questions: 172

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Functionality	46
Topic 2, User Interface	42
Topic 3, Data Management	48
Topic 4, Debugging	17
Topic 5, Testing	19
Total	172

QUESTION NO: 1

Select four different types of app components. (Choose four.)

A. Application

B. Layouts

C. Activities

D. Services

E. AlarmManager

F. WorkManager

G. Broadcast receivers

H. Content providers

ANSWER: C D G H

Explanation:

Android defines four fundamental types of application components: activities, services, broadcast receivers, and content providers. **Activities** provide a focused screen with which a user can interact, typically serving as an entry point for a user-facing task. **Services** are entry points for keeping an operation running in the background, such as playing audio or performing work initiated by another component. **Broadcast receivers** enable an app to respond when the Android system or another app broadcasts an event, including events such as device boot completion or a change in connectivity. **Content providers** manage access to a structured set of app data and can expose that data securely to other apps through a standard interface. Each of these component types can be declared in the app manifest when appropriate, allowing Android to recognize and launch the component in response to intents, system events, or data requests. Android's official component overview identifies these four categories as the building blocks of an Android app: App fundamentals. For the platform definition and role of content providers specifically, see Content providers overview.

QUESTION NO: 2

What statements about Android string resources are correct? (Choose two.)

A. A string resource can use positional format arguments such as %1$s.

B. Android can select a locale-qualified version of a string resource at runtime.

C. Each string resource must have a unique ID for every supported language.

D. getString() cannot accept format arguments.

E. String resources can be stored only in the res/raw directory.

ANSWER: A B

Explanation:

A string resource can contain format arguments such as `%1$s` and `%2$d`. The application supplies values for those arguments through `getString()`, for example `getString(R.string.welcome_message, userName)`. Positional arguments help translators reorder values when required by another language.

String resources can have locale-qualified alternatives, such as resources in `res/values-es/`. Android selects the most appropriate resource for the device configuration, allowing the same resource ID to resolve to localized text. See the Android documentation on string resources and localizing your app.

QUESTION NO: 3

The easiest way of adding menu items (to specify the options menu for an activity) is inflating an XML file into the Menu via MenuInflater. With menu_main.xml we can do it in this way:

A. override fun onCreateOptionsMenu(menu: Menu): Boolean { menuInflater.inflate(R.menu.menu_main, menu) return true }

B. override fun onOptionsItemSelected(item: MenuItem): Boolean { menuInflater.inflate(R.menu.menu_main, menu) return super.onOptionsItemSelected(item) }

C. override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState) setContentView(R.menu.menu_main) }

ANSWER: A

Explanation:

`override fun onCreateOptionsMenu(menu: Menu): Boolean { menuInflater.inflate(R.menu.menu_main, menu) return true }` is the correct implementation because `onCreateOptionsMenu()` is the Activity lifecycle callback Android invokes when it needs the activity's options menu. The supplied `Menu` object is the destination into which the menu resource must be inflated. Calling `menuInflater.inflate(R.menu.menu_main, menu)` parses the XML resource located at `res/menu/menu_main.xml` and adds its declared menu items to that destination menu.

Returning `true` informs the framework that the activity has provided a menu and that it should be displayed. This is the standard Kotlin pattern for a traditional Views-based Activity options menu. The XML menu resource can define item IDs, titles, icons, grouping, ordering, and display behavior, while selection handling is performed separately when a user chooses an item. Android's menu guide documents inflating an XML menu in `onCreateOptionsMenu()`, and the API reference describes `MenuInflater` as the class used to instantiate a menu hierarchy from an XML resource. See the Android menus guide and the MenuInflater API reference.

QUESTION NO: 4

What statements about a Room DAO method annotated with `@Transaction` are correct? (Choose two.)

A. It causes the database operations performed by the method to be executed atomically.

B. It can be used for a DAO method that returns a POJO containing @Relation fields.

C. It causes Room to execute the method on the main thread.

D. It can be applied only to methods annotated with @Insert.

E. It permanently locks all Room tables after the method returns.

ANSWER: A B

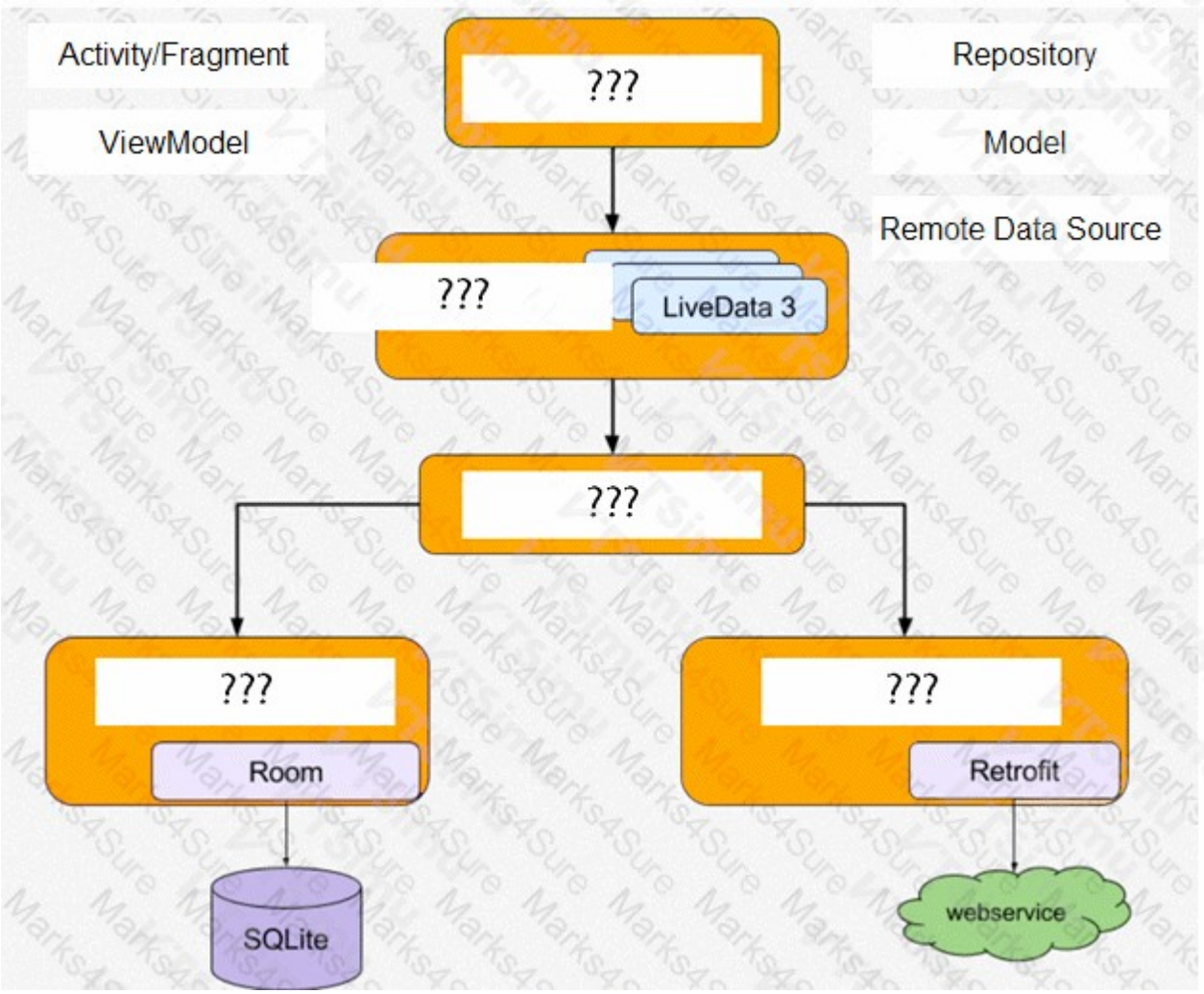
Explanation:

A method annotated with `@Transaction` runs its database operations atomically. If the method completes successfully, its changes are committed together; if it fails, the transaction can be rolled back. This is useful when an operation requires several related inserts, updates, deletes, or queries to be handled consistently.

Room also uses `@Transaction` for DAO methods that return relation-based POJOs, such as objects containing `@Relation` fields. The transaction ensures Room reads the related tables consistently while it assembles the returned object graph. See the @Transaction API reference and the Room guide on relationships.

QUESTION NO: 5 - (SIMULATION)

With recommended app architecture. Fill the following diagram, which shows how all the modules usually should interact with one another after designing the app (drag modules to correct places).



ANSWER: See the explanation for the answer

Explanation:

Place the modules in this order:

Top box: `Activity/Fragment`
Second box (the one containing LiveData): `ViewModel`
Middle box below ViewModel, which branches to both lower boxes: `Repository`
Lower-left box above Room and SQLite: `Model`
Lower-right box above Retrofit and the webservice: `Remote Data Source`

This follows the recommended flow: Activity/Fragment observes data exposed by the ViewModel through LiveData; the ViewModel requests data from the Repository; and the Repository coordinates local model/database access (Room/SQLite) and remote access (Retrofit/web service).

QUESTION NO: 6

What is demonstrated by the code below?

// RawDao.kt @Dao

interface RawDao { @RawQuery fun getUserViaQuery(query: SupportSQLiteQuery?): User? }

// Usage of RawDao

...

val query =

SimpleSQLiteQuery("SELECT * FROM User WHERE id = ? LIMIT 1", arrayOf(sortBy)) val user = rawDao.getUserViaQuery(query) ...

A. A method in a Dao annotated class as a raw query method where you can pass the query as a SupportSQLiteQuery.

B. A method in a Dao annotated class as a query method.

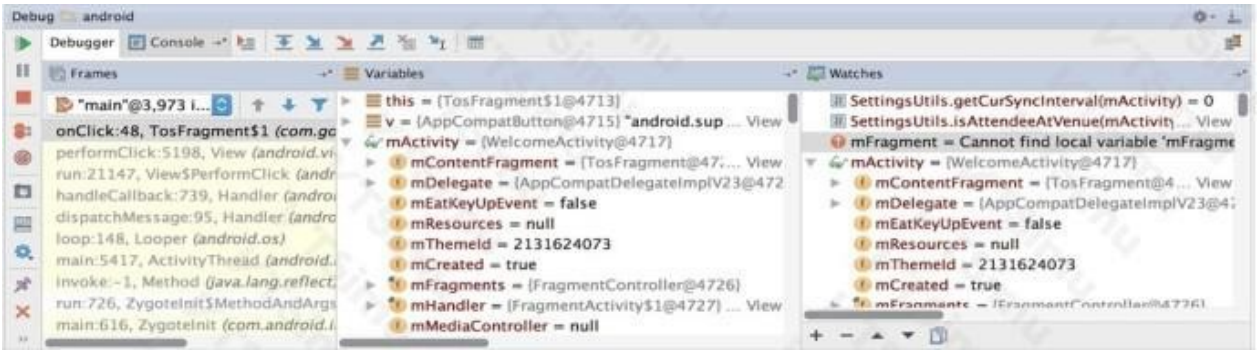
C. A method in a RoomDatabase class as a query method.

ANSWER: A

Explanation:

The code demonstrates a DAO method annotated with `@RawQuery` that accepts a `SupportSQLiteQuery`. This Room feature is intended for cases where an SQL statement must be constructed at runtime instead of being supplied as a fixed string in a `@Query` annotation. The caller creates a `SimpleSQLiteQuery` containing the SQL statement and its bound arguments, then passes that query object to `getUserViaQuery`. Room executes the supplied query and maps the resulting row to the declared `User?` return type; the nullable return permits no matching row to be returned as `null`. Parameter binding through the query object's argument array is important because it supplies values separately from the SQL text. A raw query method must be declared in a type annotated with `@Dao`, as shown by `RawDao`. Android's Room documentation describes `@RawQuery` as the annotation for DAO methods that receive a `SupportSQLiteQuery`, and documents `SimpleSQLiteQuery` as a convenient implementation for creating such queries. See the Room RawQuery reference and the SimpleSQLiteQuery reference.

QUESTION NO: 7



What is illustrated in the picture?

A. Logcat window with filter settings

B. Debugging native code using LLDB

C. The Variables and Watches panes in the Debugger window

D. The Breakpoints window lists all the current breakpoints and includes behavior settings for each

E. Adding a watchpoint to a variable in memory

ANSWER: C

Explanation:

The Variables and Watches panes in the Debugger window is correct. In Android Studio, when an app is paused at a breakpoint, the Debug tool window provides contextual information about the current execution state. The Variables pane displays values that are available in the selected stack frame, such as local variables, object fields, and relevant properties. This lets a developer inspect an app's state at the exact point where execution stopped.

The Watches pane complements variable inspection by allowing expressions or variables of particular interest to remain visible while stepping through code. A watch is useful when a developer wants to monitor a value repeatedly without searching through the current variable tree each time execution pauses. Together, these panes are core parts of Android Studio's interactive debugger workflow: pause at a breakpoint, select a stack frame, inspect state, evaluate or watch values, and continue or step through execution. Google's Android Studio documentation describes inspecting variables and adding watches during a debugging session in the Debug your app guide. JetBrains also documents the debugger's variable and watch views in its Examine a suspended program reference.

QUESTION NO: 8

Once your test has obtained a UiObject object, you can call the methods in the UiObject class to perform user interactions on the UI component represented by that object. You can specify such actions as: (Choose four.)

A. click() : Clicks the center of the visible bounds of the UI element.

B. touch() : Touch the center of the visible bounds of the UI element.

C. dragTo() : Drags this object to arbitrary coordinates.

D. moveTo() : Move this object to arbitrary coordinates.

E. setText() : Sets the text in an editable field, after clearing the field's content. Conversely, the clearTextField() method clears the existing text in an editable field.

F. swipeUp() : Performs the swipe up action on the UiObject. Similarly, the swipeDown(), swipeLeft(), and swipeRight() methods perform corresponding actions.

ANSWER: A C E F

Explanation:

The supported interaction methods are `click()`, `dragTo()`, `setText()`, and `swipeUp()`. A `UiObject` represents a matched UI element and exposes methods that allow a UI Automator test to act directly on that element. `click()` performs a click at the center of the object's visible bounds. `dragTo()` initiates a drag gesture from the object toward specified screen coordinates, allowing tests to model drag-and-drop or other drag interactions. For editable UI controls, `setText()` replaces the field's current content with the supplied text; `clearTextField()` is also available when a test needs to clear editable content. Finally, `swipeUp()` performs an upward swipe on the object, with companion directional methods available for downward, leftward, and rightward swipes. These APIs let automated tests reproduce common user gestures after locating a target view through UI Automator selectors. See the UiObject API reference and Android's UI Automator testing guide.

QUESTION NO: 9

If no any folder like res/anim-, res/drawable-, res/layout-, res/raw-, res/xml- exist in the project. Which folders are required in the project anyway? (Choose two.)

A. res/anim/

B. res/drawable/

C. res/layout/

D. res/raw/

E. res/xml/

ANSWER: B C

Explanation:

`res/drawable/` and `res/layout/` are the required default resource directories in the situation described. Android resource directories can include configuration qualifiers, such as language, screen size, orientation, or density, allowing the application to supply alternative versions of a resource. However, a resource that is supplied in a qualified directory should also have a default version, stored in the corresponding unqualified directory. The default resource provides the fallback when a device configuration does not match one of the qualified resource directories.

For drawable resources, the unqualified `res/drawable/` directory is the default location for drawable files and XML drawables. For user-interface layouts, `res/layout/` is the default location for layout XML files. Maintaining these default versions lets Android resolve the resource consistently across device configurations and avoids missing-resource failures when no alternate configuration is selected. Android's resource documentation describes default resources as necessary fallbacks when configuration-specific alternatives are provided. See the Providing resources overview and the localization resource guidance.

QUESTION NO: 10

An Activity uses the Navigation component and has a NavController named `navController`. Which call navigates to the destination represented by `R.id.settingsFragment`?

A. navController.navigate(R.id.settingsFragment)

B. navController.open(R.id.settingsFragment)

C. navController.replace(R.id.settingsFragment)

D. navController.addFragment(R.id.settingsFragment)

ANSWER: A

Explanation:

`navController.navigate(R.id.settingsFragment)` instructs the Navigation component to navigate to the destination whose ID is declared in the navigation graph. The NavController handles the associated Fragment transaction and updates the navigation back stack according to the graph configuration.

The destination ID must belong to the navigation graph currently managed by that NavController. Arguments can be passed through overloads of `navigate()`, or through generated directions when using Safe Args. See the Navigate to a destination guide and the NavController.navigate() reference.

QUESTION NO: 11 - (DRAG DROP)

DRAG DROP

Move the major components of the Android platform to correct places in diagram.

Select and Place:

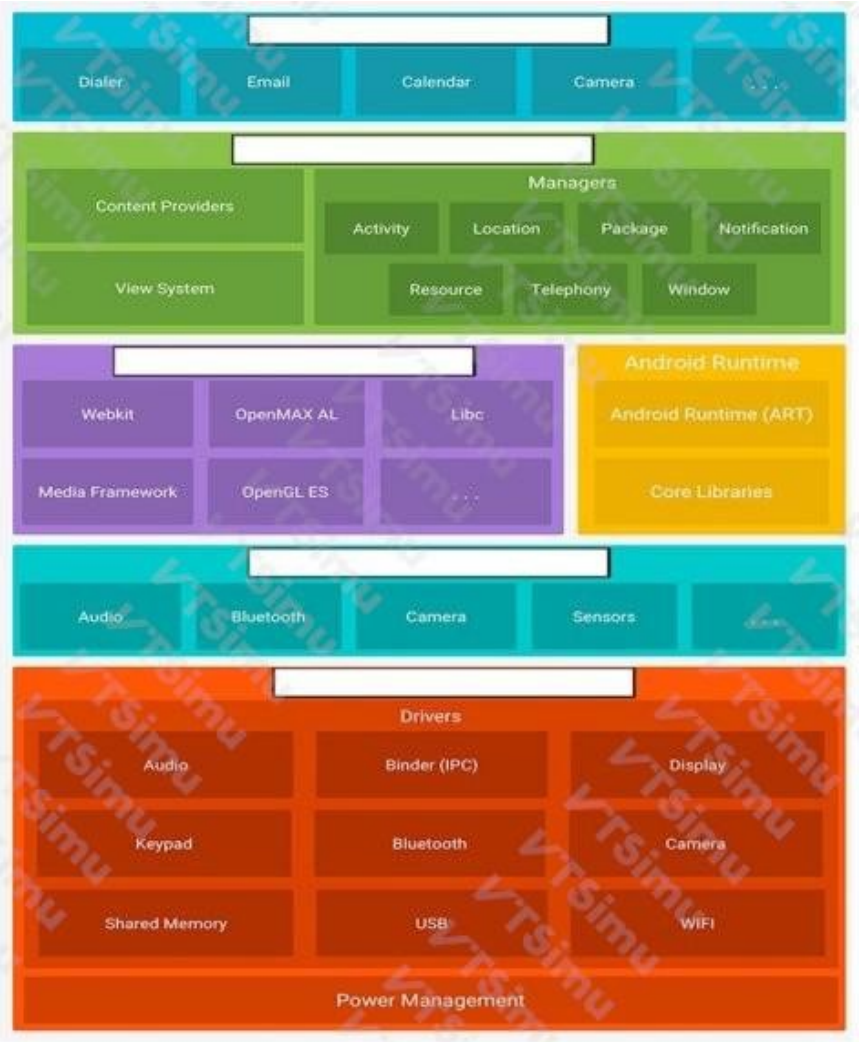
System Apps

Java API Framework

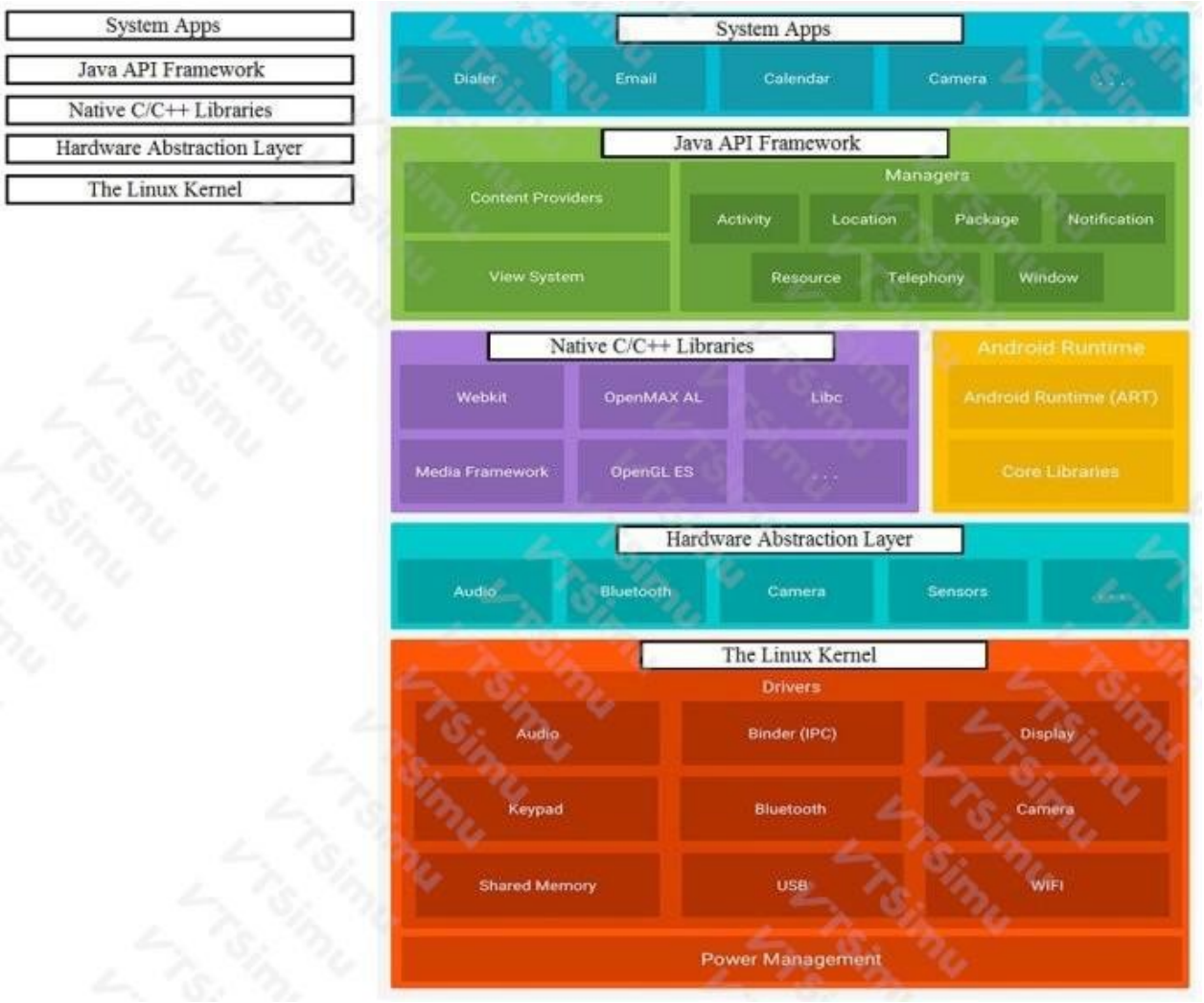
Native C/C++ Libraries

Hardware Abstraction Layer

The Linux Kernel



ANSWER:



Explanation:

The completed diagram correctly represents Android as a layered platform, with each major component placed according to the responsibility it provides to the layer above it. System apps belong at the top because they are the user-facing applications, such as dialing, email, calendar, and camera functionality. These apps use the Java API Framework, which supplies the high-level system services and managers used by application code. The framework exposes capabilities such as activities, notifications, package management, resources, telephony, location, and window management in a consistent application-development interface.

Below the framework, native C/C++ libraries provide lower-level capabilities used by the framework and runtime environment. The libraries shown include graphics, media, web rendering, and standard native-library functionality. They are correctly shown alongside Android Runtime because both support execution of Android application code and core platform behavior. The Hardware Abstraction Layer is correctly positioned beneath them because it defines standard interfaces between higher-level Android services and device-specific hardware implementations, including audio, Bluetooth, camera, and sensors.

The Linux Kernel correctly forms the base of the stack. It provides the essential low-level operating-system functions, including hardware drivers, process and memory support, interprocess communication through Binder, networking support, and power management. This layered arrangement allows Android software to access device hardware through stable abstractions instead of requiring every application or framework service to handle hardware-specific details directly. The diagram therefore follows the Android platform architecture described in the official Android platform architecture documentation.

QUESTION NO: 12

For example, our preferences.xml file was added by addPreferencesFromResource (R.xml.preferences). Our preferences.xml file contains such item:

< ListPreference android:id= " @+id/order_by " android:key= " @string/pref_sort_key "

android:title= " @string/pref_sort_title " android:summary= " @string/pref_sort_summary " android:dialogTitle= " @string/pref_sort_dialog_title " android:entries= " @array/sort_oder " android:entryValues= " @array/sort_oder_value " android:defaultValue= " @string/pref_default_sort_value " app:iconSpaceReserved= " false " / >

In our Fragment, we can dynamically get current notification preference value in this way:

A. String sortBy = PreferenceManager.getDefaultSharedPreferences(getContext ()).getString(getContext().getString(R.string.pref_sort_key), getContext().getResources().getBoolean(R.bool.pref_default_sort_value));

B. String sortBy = PreferenceManager.getSharedPreferences(getContext()).getString(getContext().getString(R.string.pref_default_sort_value), getContext().getString(R.string.pref_sort_key));

C. boolean sortBy = PreferenceManager.getSharedPreferences(getContext()).getBoolean (getContext().getResources().getBoolean(R.bool.pref_default_sort_value), getContext().getString(R.string.pref_sort_key));

D. String sortBy = PreferenceManager.getDefaultSharedPreferences(getContext ()).getString(getContext().getString(R.string.pref_sort_key), getContext().getString(R.string.pref_default_sort_value))

ANSWER: D

Explanation:

`String sortBy = PreferenceManager.getDefaultSharedPreferences(getContext ()).getString(getContext().getString(R.string.pref_sort_key), getContext().getString(R.string.pref_default_sort_value))` is correct because a `ListPreference` persists its selected value as a string in the default shared-preferences file when it is created through the standard preference framework. The value used to retrieve it must be the preference key, which is supplied by `android:key="@string/pref_sort_key"`. Calling `getContext().getString(R.string.pref_sort_key)` resolves that string resource to precisely the key under which the selected entry value is stored.

The second argument to `SharedPreferences.getString()` is the fallback value returned when no value has yet been persisted. Because the XML declares `android:defaultValue="@string/pref_default_sort_value"`, resolving that resource with `getString()` provides a matching string default. The result is therefore correctly assigned to `String sortBy`. This pattern lets the fragment read the currently saved list-preference value at runtime, while still providing a safe default before a user makes a selection. Android documents that `getString(key, defValue)` returns the string associated with a key or the supplied default if none exists. See SharedPreferences.getString() and ListPreference.

QUESTION NO: 13

As an example. In an Activity we have our TimerViewModel object (extended ViewModel), named mTimerViewModel. mTimerViewModel.getTimer() method returns a LiveData value. What can be a correct way to set an observer to change UI in case if data was changed?

A. mTimerViewModel.getTimer().getValue().toString().observe(new Observer() {
@Override
public void onChanged(Long aLong) {
callAnyChangeUIMethodHere(aLong);
}
});

B. mTimerViewModel.getTimer().observe(this, new Observer() {
@Override
public void onChanged(Long aLong) {
callAnyChangeUIMethodHere(aLong);
}
});

C. mTimerViewModel.observe(new Observer() {
@Override
public void onChanged(Long aLong) {
callAnyChangeUIMethodHere(aLong);

}
});

ANSWER: B

Explanation:

`mTimerViewModel.getTimer().observe(this, new Observer<Long>() { ... });` is the correct lifecycle-aware observation pattern for an Activity. `getTimer()` returns the `LiveData` instance, and calling `observe()` on that instance registers an `Observer` whose `onChanged()` callback receives the latest timer value. UI-update work can safely be performed in that callback, such as rendering the new value in a `TextView`.

The Activity supplies itself as the `LifecycleOwner` through `this`. This is important because LiveData automatically delivers updates only while that owner is in an active lifecycle state, and it removes the observer when the Activity is destroyed. Consequently, the UI remains synchronized with timer data without requiring manual lifecycle cleanup. In Java, the generic type must be the boxed type `Long`, rather than primitive `long`, because Java generics accept reference types. Android's LiveData documentation describes this lifecycle-aware `observe(owner, observer)` API and its automatic observer management. See the LiveData overview and the LiveData.observe reference.

QUESTION NO: 14

We have a custom view that extends android.widget.ProgressBar. Our progress bar is not touchable, focusable, etc.: it just shows progress. Style for our custom progress bar extends

â€œWidget.AppCompat.ProgressBar.Horizontalâ€□. An item, named â€œprogressDrawableâ€□, in our style, is a xml file . What we usually can see as a main single element in this xml file:

A. A State List (element)

B. A element with elements using android:id="@android:id/background" and android:id="@android:id/progress".

C. An element with android:id="@+id/progress" identifier

ANSWER: B

Explanation:

A `<layer-list>` element containing `<item>` elements identified by `android:id="@android:id/background"` and `android:id="@android:id/progress"` is the standard structure for a horizontal `ProgressBar`'s `progressDrawable`. A progress drawable must represent at least the static track/background and the portion that is filled as progress changes. A layer-list drawable combines those visual components into one drawable resource, allowing the framework to locate and scale the progress layer independently as the view's progress value changes. The framework-recognized IDs are Android framework resource IDs, so they use the `@android:id/...` namespace rather than newly declared application IDs. A drawable can also include a layer identified as `@android:id/secondaryProgress` when secondary progress is needed, but it is not required for a basic determinate progress indicator. Each item commonly contains a drawable such as a shape, bitmap, or clip drawable; the progress layer is often clipped horizontally to reflect the current value. Android documents layer-list resources as drawable XML whose root element is `<layer-list>`, and documents `ProgressBar` as using a drawable for its progress visualization. See Android's layer-list drawable documentation and the ProgressBar API reference.

QUESTION NO: 15

The easiest way of adding menu items (to specify the options menu for an activity) is inflating an XML file into the Menu via MenuInflater. With menu_main.xml we can do it in this way:

A. @Override public boolean onCreateOptionsMenu(Menu menu) { getMenuInflater().inflate(R.menu.menu_main, menu);
return true;
}

B. @Override public boolean onOptionsItemSelected(MenuItem item) { getMenuInflater().inflate(R.menu.menu_main, menu); return super.onOptionsItemSelected(item); }

C. @Override protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.menu.menu_main); }

ANSWER: A

Explanation:

`@Override public boolean onCreateOptionsMenu(Menu menu) { getMenuInflater().inflate(R.menu.menu_main, menu); return true; }` is the correct implementation because `onCreateOptionsMenu(Menu)` is the activity lifecycle callback intended for creating the app bar's options menu. Android calls this method when it needs the activity to populate its menu. The supplied `Menu` instance is the destination into which menu resources must be inflated.

`getMenuInflater()` obtains the activity's `MenuInflater`, and `inflate(R.menu.menu_main, menu)` reads the XML resource located at `res/menu/menu_main.xml` and adds its declared `<item>` elements to that supplied menu. Returning `true` confirms that the activity has provided a menu and that it should be displayed. This keeps menu structure in an XML resource, separating presentation and configuration from activity code while allowing Android to create the appropriate menu UI for the current device and app-bar configuration.

Android's menu guide documents `onCreateOptionsMenu()` as the place to inflate a menu resource with `MenuInflater`. See Android menus overview and Activity.onCreateOptionsMenu(Menu) reference.

QUESTION NO: 16

The Log class allows you to create log messages that appear in logcat. Generally, you could use the following log methods: (Choose five.)

A. Log.e(String, String) (error)

B. Log.a(String, String) (all outputs)

C. Log.w(String, String) (warning)

D. Log.i(String, String) (information)

E. Log.q(String, String) (questions)

F. Log.d(String, String) (debug)

G. Log.v(String, String) (verbose)

ANSWER: A C D F G

Explanation:

`Log.e(String, String) (error)`, `Log.w(String, String) (warning)`, `Log.i(String, String) (information)`, `Log.d(String, String) (debug)`, and `Log.v(String, String) (verbose)` are the standard Android `Log` methods for writing messages to logcat. Each accepts a tag and a message, allowing developers to categorize and filter output while diagnosing application behavior. The methods correspond to Android's established log priorities: error is used for failures, warning for potentially harmful or unexpected conditions, information for general runtime events, debug for diagnostic development output, and verbose for the most detailed diagnostic messages. These priorities support practical filtering in Logcat so that developers can focus on messages relevant to the current troubleshooting task. Android's `Log` API documentation lists these methods and their overloads, including the two-`String`-argument forms stated in the question. See the Android Log API reference and Android's Logcat documentation for usage and filtering guidance.

QUESTION NO: 17

To run your local unit tests, follow these steps:

1. Be sure your project is synchronized with Gradle by clicking Sync Project  in the toolbar.

2. Run your test in one of the following ways (select possible): (Choose three.)

A. To run a single test, open the Project window, and then right-click a test and click Run .

B. To test all methods in a class, right-click the class in the test file and click Run .

C. To run all tests in a directory, right-click on the directory and select Run tests .

ANSWER: A B C

Explanation:

Local unit tests execute on the development machine's JVM, so Android Studio can run them at several levels of scope directly from the Project tool window. To execute one specific test, open the Project window, right-click the desired test, and choose *Run*. This is useful when validating a focused behavior while developing or troubleshooting. To execute every test method declared in one test class, right-click that class in the test source file and choose *Run*; Android Studio creates an appropriate test run configuration for the selected class. For a broader scope, right-clicking a directory that contains test sources and selecting *Run tests* runs the tests under that directory. These actions rely on Gradle recognizing the test sources and dependencies, which is why syncing the project before running tests is important. Android's testing guidance identifies the local test source set as `src/test/java` and describes running tests from Android Studio, while Android Studio documentation explains that tests can be run from a class, method, package, or directory context. See Build local unit tests and Test in Android Studio.