ANDROID LAB - 9
Sri Harshini Jilloju - N01649103

1.What are the key differences between explicit and implicit intents?

Explicit and implicit intents serve different purposes in Android development. Explicit intents specify the exact component (such as an activity or service) that should handle the intent by providing the target component's class name. They are commonly used for internal communication within the same application, such as navigating between activities. In contrast, implicit intents do not specify a target component; instead, they declare an action to be performed, allowing the system to find the most suitable application to handle the request. These are useful when interacting with other apps, such as opening a web page, sharing data, or sending emails. One key difference is that explicit intents are safer as they limit communication to known components, whereas implicit intents rely on the system's ability to resolve a matching handler. However, implicit intents require proper security measures to prevent malicious apps from intercepting sensitive data. The table below highlights their differences:

Feature	Explicit Intent	Implicit Intent
Target Component	Specified using class name	Not specified; system resolves suitable app
Usage	Internal app navigation	Interacts with other apps
Security	More secure (direct communication)	Requires security measures (intent filtering)
Example	Intent(this, SecondActivity::class.java)	Intent(Intent.ACTION_VIEW, uri)

2.How would you handle a scenario where an application is unavailable to handle an implicit intent?

When an implicit intent is sent, the Android system attempts to find an application that can handle it. If no suitable application is available, the intent call can fail, leading to a crash or unexpected behavior. To handle this scenario gracefully, developers should always check if an intent has a valid handler before starting an activity. This can be done using packageManager.resolveActivity(intent, PackageManager.MATCH_DEFAULT_ONLY), which ensures that at least one application is available to process the request. If no app is found, the app can

notify the user with a Toast message or provide an alternative approach, such as redirecting them to install a required application from the Play Store. Additionally, using a try-catch block when calling startActivity(intent) can prevent runtime crashes. Another approach is to offer manual input options when an automatic resolution is not possible. For instance, if no email app is installed, the app can provide a text box for users to copy the email address manually. By implementing these measures, developers can ensure a smooth user experience and avoid application crashes when handling implicit intents.

3.Describe the process of securing an intent to prevent unauthorized data access.

To secure an intent and prevent unauthorized access, developers should apply proper security measures such as intent filters, permission checks, and data validation. One common approach is using explicit intents whenever possible, as they limit communication to known components within the app. When implicit intents are necessary, intent filters should be restricted to limit access to only trusted apps. Additionally, for sensitive data, custom permissions can be defined in the AndroidManifest.xml file to restrict which apps can receive or send certain intents. Another important technique is using FLAG_SECURE when dealing with confidential information, preventing screenshots or unauthorized viewing of sensitive data. Intent extras should also be validated to prevent intent injection attacks, where malicious apps try to insert harmful data into an intent. Developers should avoid exposing broadcast receivers unnecessarily by setting their android:exported attribute to false unless required. Using signature-level permissions ensures that only apps signed with the same certificate can communicate with the app securely. These security measures help protect user data and prevent unauthorized access to sensitive information when using intents.

4.When should you prefer Parcelable over Serializable, and why?

In Android, Parcelable and Serializable are used to transfer objects between activities or services, but Parcelable is preferred due to its better performance. Parcelable is optimized for Android by using manual serialization of object data, making it faster and more efficient compared to Java's built-in Serializable interface, which relies on reflection and is significantly slower. Parcelable is ideal for high-performance applications, especially when transferring large datasets or passing objects frequently between activities. Serializable, while easier to implement, is less efficient and results in higher memory overhead, making it unsuitable for performance-critical applications. A major advantage of Parcelable is its customizable control over serialization, allowing developers to

define precisely how data is written and read. However, Serializable is still useful for cases where compatibility with non-Android Java code is required. The table below summarizes their differences:

Feature	Parcelable	Serializable
Performance	Fast - optimized for Android	Slow - uses reflection
Implementation	Manual (writeToParcel() and CREATOR)	Automatic (implements Serializable)
Use Case	Android-specific object passing	General Java object persistence
Memory Usage	Low (efficient memory usage)	High (overhead due to reflection)

5.How can you ensure large data transfers do not compromise application performance or security?

When transferring large amounts of data between components, it is essential to ensure that it does not negatively impact performance or security. Instead of passing large objects directly via intents or bundles, developers should consider using a lightweight approach, such as storing data in a database, shared preferences, or external storage, and passing only references (e.g., file paths, database IDs). Parcelable should be used instead of Serializable for object transfer, as it is more efficient. For security, sensitive data should be encrypted before transmission to prevent unauthorized access. Another approach is using ContentProviders, which allow controlled access to data using permissions. When working with background processes, WorkManager or IntentService should be used to handle data transfer asynchronously, preventing UI lag. Additionally, developers should limit the amount of data held in memory by using streams instead of loading entire files into RAM. These best practices ensure that large data transfers remain efficient, secure, and scalable without affecting app performance.

6.How can implicit intents be used to interact with other apps in a secure way?

Implicit intents enable interaction between applications by specifying an action rather than a specific component. To ensure secure inter-app communication, developers should use intent filters with proper permission handling, ensuring that only trusted applications can process sensitive intents. When sending

sensitive data, custom permissions should be enforced to restrict access. For example, using FLAG_GRANT_READ_URI_PERMISSION allows temporary access to files instead of exposing them completely. Intent validation should also be performed to prevent intent spoofing attacks, where malicious apps attempt to intercept or manipulate data. Additionally, using PendingIntent ensures that only authorized applications can trigger certain actions within another app. Developers should avoid using broadcast intents for sensitive operations, as they can be intercepted by any app on the device. By following these practices, implicit intents can be safely used to interact with other apps without exposing vulnerabilities.

7.What are the best practices for ensuring data integrity during inter-app communication?

Ensuring data integrity when communicating between apps is crucial to prevent data manipulation and security risks. One of the best practices is using signed and encrypted data when transferring information between apps to prevent unauthorized tampering. Custom permissions should be implemented to control which apps can send and receive data. When using Intent or Bundle for data transfer, validating input data is essential to avoid injection attacks. ContentProviders with proper permission settings (READ/WRITE_PERMISSION) can be used to securely expose app data. Additionally, for APIs or web-based communication, HTTPS and token-based authentication (such as OAuth) should be used to ensure data integrity and prevent eavesdropping. Developers should also use PendingIntents to ensure only the intended receiver can process a specific intent. By combining these techniques, developers can establish a secure and reliable communication channel between applications while ensuring data remains protected and unaltered.