**Q1**

**Polymorphism​**

Polymorphism allows computer code to become contextual. In other words, a computer instruction can take on different meanings depending on situation in which the instruction is used. This is no different than how we speak. A person can use the same word, ‘there’, ‘they’re’, or ‘their’ to mean different things even though all three words are said the same way.

Kotlin supports two forms of polymorphism because it is both strongly and statically typed. The first form of polymorphism happens when the code is compiled. The other form happens at runtime. Understanding both forms of polymorphism is critical when writing code in Kotlin.

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks.

Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance. Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.

一個訊息（message or event or stimulus）的意義是由接收者（接收到這個訊息的物件）來解釋，而不是由訊息發出者（sender）來解釋。所以，在runtime時只要接受者換成不同的物件或是instance，系統的行為就會改變。具有這樣的特性就稱之為polymorphism。

**Q2**

**7 important lifecycle methods of an Activity**



|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Killable** | **Next** |
| **onCreat()** | Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one.  Always followed by onStart(). | No | **onStart()** |
| **onRestart()** | Called after your activity has been stopped, prior to it being started again.  Always followed by onStart() | No | **onStart()** |
| **onStart()** | Called when the activity is becoming visible to the user.  Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden. | No | **onResume()or**  **onStop()** |
| **onResume** | Called when the activity will start interacting with the user. At this point your activity is at the top of its activity stack, with user input going to it.  Always followed by onPause(). | No | **onPause()** |
| **onPause()** | Called when the activity loses foreground state, is no longer focusable or before transition to stopped/hidden or destroyed state. The activity is still visible to user, so it's recommended to keep it visually active and continue updating the UI. Implementations of this method must be very quick because the next activity will not be resumed until this method returns.  Followed by either onResume() if the activity returns back to the front, or onStop() if it becomes invisible to the user. | **P**re-[Build.VERSION\_CODES.HONEYCOMB](https://developer.android.com/reference/android/os/Build.VERSION_CODES" \l "HONEYCOMB) | **onResume()or onStop()** |
| **onStop()** | Called when the activity is no longer visible to the user. This may happen either because a new activity is being started on top, an existing one is being brought in front of this one, or this one is being destroyed. This is typically used to stop animations and refreshing the UI, etc.  Followed by either onRestart() if this activity is coming back to interact with the user, or onDestroy() if this activity is going away. | Yes | **onRestart()or onDestroy()** |
| **onDestroy()** | The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called [Activity#finish](https://developer.android.com/reference/android/app/Activity" \l "finish()) on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the [Activity#isFinishing](https://developer.android.com/reference/android/app/Activity" \l "isFinishing())method. | Yes | nothing |

**Q3**

**Android Jetpack**

Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about.

1.介紹

Jetpack 是一套庫、工具和指南，可幫助開發者更輕鬆地編寫優質應用。這些組件可幫助您遵循最佳做法、讓您擺脫編寫樣板代碼的工作並簡化複雜任務，以便您將精力集中放在所需的代碼上。

換言之，Google利用Jetpack將一些優秀的Android組件庫進行了標準化。

2.作用

Android Jetpack 完美兼容 Kotlin 語言，利用 Android KTX 可大幅節省代碼量。

Jetpack 包含與平臺 API 解除捆綁的 androidx.\* 軟件包庫。這意味著，它可以提供向後兼容性，且比 Android 平臺的更新頻率更高，以此確保您始終可以獲取最新且最好的 Jetpack 組件版本。

加速開發：組件可以單獨採用（不過這些組件是為協同工作而構建的），同時利用 Kotlin 語言功能幫助您提高工作效率。

消除樣板代碼：Android Jetpack 可管理繁瑣的 Activity（如後臺任務、導航和生命週期管理）。以便您可以專注於如何讓自己的應用出類拔萃。

構建高質量的強大應用：Jetpack 組件圍繞現代化設計實踐構建而成，具有向後兼容性，可以減少崩潰和內存洩漏。

3.Jetpack包含那些組件

Android Jetpack 組件覆蓋以下 4 個方面：基礎（Foundation）、架構（Architecture）、行為（Behavior） 、界面（UI）

**Q4**

**Coroutines**

1.Introduction

A coroutine is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines were added to Kotlin in version 1.3 and are based on established concepts from other languages.

On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive. Over 50% of professional developers who use coroutines have reported seeing increased productivity. This topic describes how you can use Kotlin coroutines to address these problems, enabling you to write cleaner and more concise app code.

2.Features

Coroutines is our recommended solution for asynchronous programming on Android. Noteworthy features include the following:

**\*Lightweight:** You can run many coroutines on a single thread due to support for suspension, which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations.

**\*Fewer memory leaks:** Use structured concurrency to run operations within a scope.

**\*Built-in cancellation support:** Cancellation is propagated automatically through the running coroutine hierarchy.

**\*Jetpack integration:** Many Jetpack libraries include extensions that provide full coroutines support. Some libraries also provide their own coroutine scope that you can use for structured concurrency.