# Java package

## Built-in packages:

**import java.util.Scanner;** -> used to get user input

|  |
| --- |
| Scanner myObj = new Scanner(System.in);  System.out.println("Enter username");  String userName = myObj.nextLine();  System.out.println("Username is: " + userName); |

import java.util.\*; -> to import whole packages

|  |  |
| --- | --- |
| Method | Description |
| nextBoolean() | Reads a boolean value from the user |
| nextByte() | Reads a byte value from the user |
| nextDouble() | Reads a double value from the user |
| nextFloat() | Reads a float value from the user |
| nextInt() | Reads a int value from the user |
| nextLine() | Reads a String value from the user |
| nextLong() | Reads a long value from the user |
| nextShort() | Reads a short value from the user |

## Get user input

|  |
| --- |
| import java.util.Scanner; // Import the Scanner class  class MyClass {  public static void main(String[] args) {  int x, y, sum;  Scanner myObj = new Scanner(System.in); // Create a Scanner object  System.out.println("Type a number:");  x = myObj.nextInt(); // Read user input  System.out.println("Sum is: " + x); // Print x  }  } |

## Create package:

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

* To create a package, use the package keyword:

|  |
| --- |
| package mypack;  class MyPackageClass {  public static void main(String[] args) {  System.out.println("This is my package!");  }  } |

* Save the file as **MyPackageClass.java**, and compile it:

C:\Users\>javac MyPackageClass.java

* Then compile the package:

C:\Users\>javac -d . MyPackageClass.java

* When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\ >java mypack.MyPackageClass

# Class

## Encapsulation

* Use private keyword
* Use getter and setter
  + Default int = 0;
  + Default String = null;
  + Default boolean = false;

## Inheritance

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

Did you notice the protected modifier in Vehicle?

|  |
| --- |
| class Vehicle {  protected String brand = "Ford"; // Vehicle attribute  public void honk() { // Vehicle method  System.out.println("Tuut, tuut!");  }  }  class Car extends Vehicle { } |

We set the **brand** attribute in **Vehicle** to a protected access modifier. If it was set to private, the Car class would not be able to access it.

**Why And When To Use "Inheritance"?**

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

## Polymorphism

allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

|  |
| --- |
| class Animal {  public void animalSound() {  System.out.println("The animal makes a sound");  }  }  class Pig extends Animal {  public void animalSound() {  System.out.println("The pig says: wee wee");  }  }  class Dog extends Animal {  public void animalSound() {  System.out.println("The dog says: bow wow");  }  } |

## Abstraction

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

|  |
| --- |
| abstract class Animal {  public abstract void animalSound();  public void sleep() {  System.out.println("Zzz");  }  } |

## Interface

Another way to achieve abstraction in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

|  |
| --- |
| // interface  interface Animal {  public void animalSound(); // interface method (does not have a body)  public void run(); // interface method (does not have a body)  } |

|  |
| --- |
| **Notes on Interfaces:**  * Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass) * Interface methods do not have a body - the body is provided by the "implement" class * On implementation of an interface, you must override all of its methods * Interface methods are by default abstract and public * Interface attributes are by default public, static and final * An interface cannot contain a constructor (as it cannot be used to create objects)  **Why And When To Use Interfaces?** 1) To achieve security - hide certain details and only show the important details of an object (interface).  2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below). |

**Differences between "extending" and "implementing"**

The major difference is that when a class extends the class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class  MyClass extends OtherClass implements Runnable.

## Inner classes

To nest multiple class

|  |
| --- |
| class FirstApp {      int x = 5;      class SecondApp {          int y = 10;      }  }  FirstApp firstApp = new FirstApp();  FirstApp.SecondApp secondApp = firstApp.new SecondApp();  System.out.println(firstApp.x);  System.out.println(secondApp.y); |

## ArrayList vs. LinkedList

The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.

The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

**How the ArrayList works**

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

**How the LinkedList works**

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

**When To Use**

Use an ArrayList for storing and accessing data, and LinkedList to manipulate data.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addFirst() | Adds an item to the beginning of the list. |
| addLast() | Add an item to the end of the list |
| removeFirst() | Remove an item from the beginning of the list. |
| removeLast() | Remove an item from the end of the list |
| getFirst() | Get the item at the beginning of the list |
| getLast() | Get the item at the end of the list |

|  |
| --- |
| ArrayList<String> cars = new ArrayList<String>();  cars.add("Volvo"); |
| LinkedList<String> cars = new LinkedList<String>();  cars.add("Volvo"); |

## HashMap

store items in "**key**/**value**" pairs, and you can access them by an index of another type

|  |
| --- |
| // Create a HashMap object called capitalCities  HashMap<String, String> capitalCities = new HashMap<String, String>();  // Add keys and values (Country, City)  capitalCities.put("England", "London");  // Print keys and values  for (String i : capitalCities.keySet()) {  System.out.println("key: " + i + " value: " + capitalCities.get(i));  } |

Methods:

add, get, remove, size

## Iterator

Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop -> use Iterator

## Exceptions

try – catch:

|  |
| --- |
| try {  System.out.println("try block");  } catch (Exception err) {  System.out.println(err);  System.out.println("catch block");  } finally {  System.out.println("The 'try catch' is finished.");  } |

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java:

ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException

## Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

**Create thread:**

It can be created by extending the Thread class and overriding its run() method:

|  |
| --- |
| public class Main extends Thread {  public void run() {  System.out.println("This code is running in a thread");  }  } |
| public class Main implements Runnable {  public void run() {  System.out.println("This code is running in a thread");  }  } |

**Running thread:**

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

|  |
| --- |
| public class Main extends Thread {  public static void main(String[] args) {  Main thread = new Main();  thread.start();  System.out.println("This code is outside of the thread");  }  public void run() {  System.out.println("This code is running in a thread");  }  } |

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

|  |
| --- |
| public class Main implements Runnable {  public static void main(String[] args) {  Main obj = new Main();  Thread thread = new Thread(obj);  thread.start();  System.out.println("This code is outside of the thread");  }  public void run() {  System.out.println("This code is running in a thread");  }  } |

## Lambda Expressions

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

**Syntax:**

parameter -> expression

more than 1 parameter

(parameter1, parameter2) -> expression

(parameter1, parameter2) -> { code block }

Example:

|  |
| --- |
| ArrayList<Integer> numbers = new ArrayList<Integer>();  numbers.add(5);  numbers.add(9);  numbers.add(8);  numbers.add(1);  numbers.forEach( (n) -> { System.out.println(n); } ); |

# File handling

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

The File class from the java.io package, allows us to work with files. To use the File class, create an object of the class, and specify the filename or directory name:

Example:

|  |
| --- |
| import java.io.File; // Import the File class  File myObj = new File("filename.txt"); // Specify the filename |

|  |  |  |
| --- | --- | --- |
| Method | Type | Description |
| canRead() | Boolean | Tests whether the file is readable or not |
| canWrite() | Boolean | Tests whether the file is writable or not |
| createNewFile() | Boolean | Creates an empty file |
| delete() | Boolean | Deletes a file |
| exists() | Boolean | Tests whether the file exists |
| getName() | String | Returns the name of the file |
| getAbsolutePath() | String | Returns the absolute pathname of the file |
| length() | Long | Returns the size of the file in bytes |
| list() | String[] | Returns an array of the files in the directory |
| mkdir() | Boolean | Creates a directory |

## Create a File

To create a file in Java, you can use the createNewFile() method.

This method returns a boolean value:

* true if the file was successfully created
* false if the file already exists.

Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

|  |
| --- |
| import java.io.File; // Import the File class  import java.io.IOException; // Import the IOException class to handle errors  public class CreateFile {  public static void main(String[] args) {  try {  File myObj = new File("filename.txt");  if (myObj.createNewFile()) {  System.out.println("File created: " + myObj.getName());  } else {  System.out.println("File already exists.");  }  } catch (IOException e) {  System.out.println("An error occurred.");  e.printStackTrace();  }  }  }  ex: File myObj = new File("C:\\Users\\MyName\\filename.txt"); |

## Write a file

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

|  |
| --- |
| import java.io.FileWriter; // Import the FileWriter class  import java.io.IOException; // Import the IOException class to handle errors  public class WriteToFile {  public static void main(String[] args) {  try {  FileWriter myWriter = new FileWriter("filename.txt");  myWriter.write("Files in Java might be tricky, but it is fun enough!");  myWriter.close();  System.out.println("Successfully wrote to the file.");  } catch (IOException e) {  System.out.println("An error occurred.");  e.printStackTrace();  }  }  } |

## Read a file

we use the Scanner class to read the contents of the text file we created

|  |
| --- |
| import java.io.File; // Import the File class  import java.io.FileNotFoundException; // Import this class to handle errors  import java.util.Scanner; // Import the Scanner class to read text files  public class ReadFile {  public static void main(String[] args) {  try {  File myObj = new File("filename.txt");  Scanner myReader = new Scanner(myObj);  while (myReader.hasNextLine()) {  String data = myReader.nextLine();  System.out.println(data);  }  myReader.close();  } catch (FileNotFoundException e) {  System.out.println("An error occurred.");  e.printStackTrace();  }  }  } |

## Get File Information

|  |
| --- |
| import java.io.File; // Import the File class  public class GetFileInfo {   public static void main(String[] args) {  File myObj = new File("filename.txt");  if (myObj.exists()) {  System.out.println("File name: " + myObj.getName());  System.out.println("Absolute path: " + myObj.getAbsolutePath());  System.out.println("Writeable: " + myObj.canWrite());  System.out.println("Readable " + myObj.canRead());  System.out.println("File size in bytes " + myObj.length());  } else {  System.out.println("The file does not exist.");  }  }  } |

## Delete a file

|  |
| --- |
| import java.io.File; // Import the File class  public class DeleteFile {  public static void main(String[] args) {  File myObj = new File("filename.txt");  if (myObj.delete()) {  System.out.println("Deleted the file: " + myObj.getName());  } else {  System.out.println("Failed to delete the file.");  }  }  } |

## Delete a Folder

|  |
| --- |
| import java.io.File;  public class DeleteFolder {  public static void main(String[] args) {  File myObj = new File("C:\\Users\\MyName\\Test");  if (myObj.delete()) {  System.out.println("Deleted the folder: " + myObj.getName());  } else {  System.out.println("Failed to delete the folder.");  }  }  } |

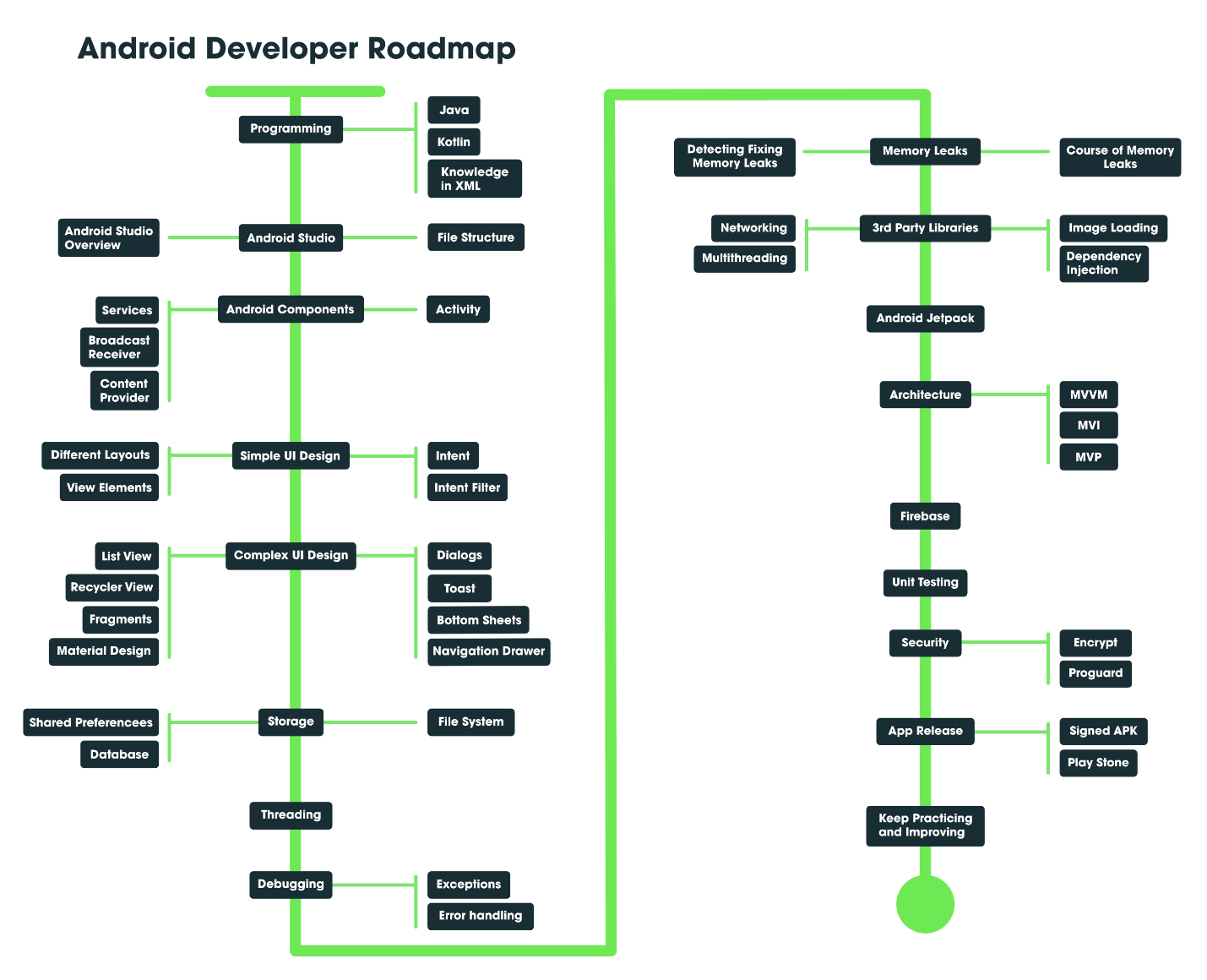
Gradle is an open-source build automation tool that helps software engineers to test, build, and release high-performance software product

Maven is an open-source build tool, used primarily for Java projects.

# Android

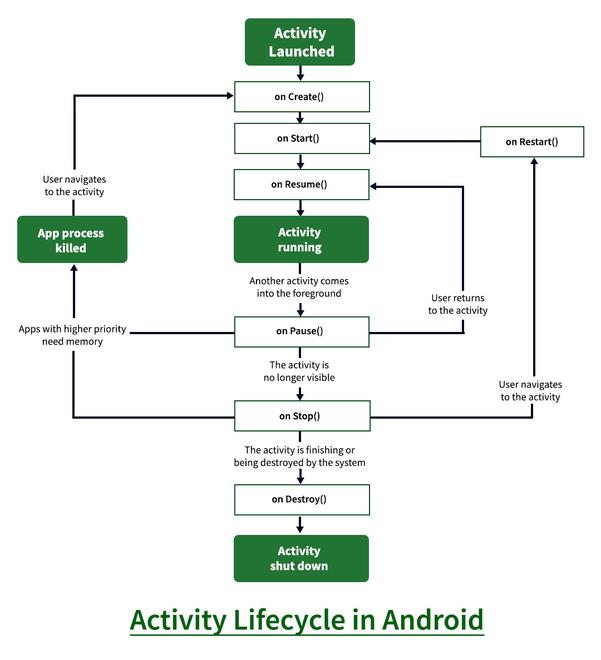
## Roadmap

|  |  |
| --- | --- |
| 1. File structure:   * AndroidManifest.xml file * Java file * Drawable file * Layout file * mipmap file * colors.xml file * strings.xml file * styles.xml file * build.gradle(Module: app) file | 2. Android component:   * Activity: * Activity life cycle * Handle Activity State Changes * Understand Tasks and Back Stack * Processes and Application Lifecycle * Services: * Types of Android Services * The Life Cycle of Android Services * Content Provider: * Content URI * Operations in Content Provider * Working of the Content Provider * Creating a Content Provider * Broadcast Receiver: * Implicit Broadcast Exceptions |
| 3. Simple UI Design   * Explore different layouts:   + Frame   + Linear   + Relative   + Constraint * View Elements:   + TextView   + EditText   + Buttons   + ImageView * Intent:   + Implicit   + Explicit   + Intent Filter | 4. Complex UI Design   * ListView * RecycleVIew * Fragments * Dialogs * Toast * Bottom Sheets * Navigation Drawer * Tabs * Material Design * Some inserting Animations |
| 5. Storage   * Shared Preferences * File System * Database   + RoomDB | 6. Build   * Gradle * Debug/ Release Configuration |
| 7. Threading   * Threads * Looper | 8. Debugging  One of the most important skills of a developer is debugging skills. So the developer must learn these things:   * Exceptions * Error Handling * Logging * Memory Profiling |
| 9. Memory Leaks   * Cause of memory leaks * Detecting and fixing memory leaks * Context | 10. Third-Party Libraries   * Image Loading Libraries * Glide * Picasso * Fresco * COIL * Dependency Injection * Dragger * Networking * Retrofit * Multithreading * Coroutines * Rxjava |
| 11. Android Jetback   * AppCompat library * Architecture components, * Animation and transitions * Android Ktx * Navigation * Paging * Slices * WorkManager | 12. Android Architecture  The three famous architecture in the Android world are:   * MVVM (Model–View–ViewModel) * MVI (Model-View-Intent) * MVP (Model View Presenter) |
| 13.  Firebase   * FCM (Firebase Cloud Messaging) * Analytics * Remote Config * App Indexing | 14. Unit Testing   * Local Unit Testing * Instrumentation Testing |
| 15. Security   * Encrypt / Decrypt * Proguard | 16. App Release   * Signed APK * Play Store |



## Lifecycle

[Activity Lifecycle in Android with Demo App - GeeksforGeeks](https://www.geeksforgeeks.org/activity-lifecycle-in-android-with-demo-app/)



1. **OnCreate:** This is called when activity is first created.

If an activity is in the foreground of the screen i.e at the top of the stack, then it is said to be active or running. This is usually the activity that the user is currently interacting with.

If an activity has lost focus and a non-full-sized or transparent activity has focused on top of your activity. In such a case either another activity has a higher position in multi-window mode or the activity itself is not focusable in the current window mode. Such activity is completely alive.

If an activity is completely hidden by another activity, it is stopped or hidden. It still retains all the information, and as its window is hidden thus it will often be killed by the system when memory is needed elsewhere.

The system can destroy the activity from memory by either asking it to finish or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

1. **OnStart:** This is called when the activity becomes visible to the user.

It is invoked when the activity is visible to the user. It is followed by onResume() if the activity is invoked from the background. It is also invoked after onCreate() when the activity is first started.

1. **OnRestart:** This is called when activity is stopped, and restarted again.

It is invoked after the activity has been stopped and prior to its starting stage and thus is always followed by onStart() when any activity is revived from background to on-screen.

1. **OnResume:** This is called when the activity starts to interact with the user.

It is invoked when the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, with a user interacting with it. Always followed by onPause() when the activity goes into the background or is closed by the user.

1. **OnPause:** This is called when activity is not visible to the user.

It is invoked when an activity is going into the background but has not yet been killed. It is a counterpart to onResume(). When an activity is launched in front of another activity, this callback will be invoked on the top activity (currently on screen). The activity, under the active activity, will not be created until the active activity’s onPause() returns, so it is recommended that heavy processing should not be done in this part.

1. **OnStop:** This is called when activity is no longer visible.

It is invoked when the activity is not visible to the user. It is followed by **onRestart()** when the activity is revoked from the background, followed by onDestroy() when the activity is closed or finished, and nothing when the activity remains on the background only. Note that this method may never be called, in low memory situations where the system does not have enough memory to keep the activity’s process running after its onPause() method is called.

1. **OnDestroy:** This is called when activity is to be closed or destroyed.

The final call received before the activity is destroyed. This can happen either because the activity is finishing (when finish() is invoked) or because the system is temporarily destroying this instance of the activity to save space. To distinguish between these scenarios, check it with **isFinishing()**method.

* If an activity is in the foreground of the screen i.e at the top of the stack, then it is said to be active or running. This is usually the activity that the user is currently interacting with.
* If an activity has lost focus and a non-full-sized or transparent activity has focused on top of your activity. In such a case either another activity has a higher position in multi-window mode or the activity itself is not focusable in the current window mode. Such activity is completely alive.
* If an activity is completely hidden by another activity, it is stopped or hidden. It still retains all the information, and as its window is hidden thus it will often be killed by the system when memory is needed elsewhere.
* The system can destroy the activity from memory by either asking it to finish or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

## Platform libraries

The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.

* Media library provides support to play and record an audio and video formats.
* Surface manager responsible for managing access to the display subsystem.
* SGL and OpenGL both cross-language, cross-platform application program interface (API) are used for 2D and 3D computer graphics.
* SQLite provides database support and FreeType provides font support.
* Web-Kit This open source web browser engine provides all the functionality to display web content and to simplify page loading.
* SSL (Secure Sockets Layer) is security technology to establish an encrypted link between a web server and a web browser.

# Data Binding

Data Binding is a library that helps us to bind the data and the UI

For example:

Instead of using the findViewById

|  |
| --- |
| TextView textView = findViewById(R.id.sample\_text); textView.setText(viewModel.getUserName()); |

We can use

|  |
| --- |
| <TextView     android:text="@{viewmodel.userName}" />  # in .xml file |

* When the userName in viewModel changed, the TextView also changed automatically

How to create Data Binding:

* Init the databinding in the gradle file (module)

|  |
| --- |
| buildFeatures {  dataBinding = true  } |

* In .xml file, we have to wrap all the tags in <layout> tag

|  |
| --- |
| <layout xmlns:android="http://schemas.android.com/apk/res/android"         xmlns:app="http://schemas.android.com/apk/res-auto">     <data>         <variable             name="viewmodel"             type="com.myapp.data.ViewModel" />     </data>     <ConstraintLayout... /> <!-- UI layout's root element --> </layout> |

* In Activity:

|  |
| --- |
| @Override protected void onCreate(Bundle savedInstanceState) {    super.onCreate(savedInstanceState);    ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_main);    User user = new User("Test", "User");    binding.setUser(user); } |

# Text View can show only data with String type, so if you want to show another data type you have to parse data into String.

#ImageView in databinding, we should use

app:imageResource="@{item.itemImage}"

so when we binding data it can show, if we use android:src="" it won’t show

# Live Data

LiveData is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

## The advantages of using LiveData

* Ensures your UI matches your data state
* No memory leaks
* No crashes due to stopped activities
* No more manual lifecycle handling
* Always up to date data
* Proper configuration changes
* Sharing resources

## Work with LiveData objects

**1.** Create an instance of LiveData to hold a certain type of data. This is usually done within your ViewModel class.

**2.** Create an [Observer](https://developer.android.com/reference/androidx/lifecycle/Observer) object that defines the [onChanged()](https://developer.android.com/reference/androidx/lifecycle/Observer#onChanged(T)) method, which controls what happens when the LiveData object's held data changes. You usually create an Observer object in a UI controller, such as an activity or fragment.

**3.** Attach the Observer object to the LiveData object using the [observe()](https://developer.android.com/reference/androidx/lifecycle/LiveData#observe(androidx.lifecycle.LifecycleOwner,%0Aandroidx.lifecycle.Observer%3CT%3E)) method. The observe() method takes a [LifecycleOwner](https://developer.android.com/reference/androidx/lifecycle/LifecycleOwner) object. This subscribes the Observer object to the LiveData object so that it is notified of changes. You usually attach the Observer object in a UI controller, such as an activity or fragment.

## Create LiveData Object:

LiveData is a wrapper that can be used with any data, including objects that implement [Collections](https://developer.android.com/reference/java/util/Collections), such as [List](https://developer.android.com/reference/java/util/List). A LiveData object is usually stored within a ViewModel object and is accessed via a getter method, as demonstrated in the following example:

**#initially, the data in a LiveData object is not set**

|  |
| --- |
| public class NameViewModel extends ViewModel {  // Create a LiveData with a String private MutableLiveData<String> currentName;      public MutableLiveData<String> getCurrentName() {         if (currentName == null) {             currentName = new MutableLiveData<String>();         }         return currentName;     }  // Rest of the ViewModel... } |

## Observe LiveData objects

|  |
| --- |
| public class NameActivity extends AppCompatActivity {      private NameViewModel model;      @Override     protected void onCreate(Bundle savedInstanceState) {         super.onCreate(savedInstanceState);          // Other code to setup the activity...          // Get the ViewModel.         model = new ViewModelProvider(this).get(NameViewModel.class);          // Create the observer which updates the UI.         final Observer<String> nameObserver = new Observer<String>(){             @Override             public void onChanged(@Nullable final String newName) {                 // Update the UI, in this case, a TextView.                 nameTextView.setText(newName);             }         };          // Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.         model.getCurrentName().observe(this, nameObserver);     } } |

* You can also bind a button with a function with a condition that function have no input parameter

android:onClick="@{viewModel::onButtonClick}"

* if function has input parameter we can use instead

android:onClick="@{(view) -> viewModel.onButtonClick(`YourParameter`)}"

* but you should use binding in activity like below (because the .xml file we shouldn’t init more variable -> make the code complex and the .xml file bigger)

binding.button.setOnClickListener(v -> {  
 myViewModel.changeUser(new User("quoc", 22));  
});