Unit -1

Introduction to Android

Android is a mobile operating system developed by Google, based on a modified version of the Linux kernel and other open source software and designed primarily for touch screen mobile devices such as smart phones and tablets. In addition, Google has further developed Android TV for televisions, Android Auto for cars, and Wear OS for wrist watches, each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.

The Android Operating System is a Linux-based OS developed by the Open Handset Alliance (OHA). The Android OS was originally created by Android, Inc., which was bought by Google in 2005. Google teamed up with other companies to form the Open Handset Alliance (OHA), which has become responsible for the continued development of the Android OS.

The android is a powerful operating system and it supports large number of applications in Smart phones. These applications are more comfortable and advanced for the users. The hardware that supports android software is based on ARM architecture platform. The android is an open source operating system means that it's free and any one can use it.

Android is a mobile operating system (OS) developed by Google. It's the most widely used Smartphone operating system in the world, powering billions of devices.

Here's a brief overview of Android:

History: Android was first released in 2008, and it was initially developed by Android Inc., a company founded by Andy Rubin. Google acquired Android Inc. in 2005 and continued to develop the OS.

Key Features:

1. **Open-source**: Android is an open-source operating system, which means that its source code is freely available for modification and distribution.
2. **Customizable**: Android offers a high degree of customization, allowing users to personalize their home screens, lock screens, and notification shades.
3. **App ecosystem**: Android has a vast app ecosystem, with millions of apps available on the Google Play Store.
4. **Hardware compatibility**: Android can run on a wide range of devices, from budget-friendly smart phones to high-end flagships, and even tablets, smart watches, and TVs.
5. **Google services integration**: Android comes with integrated Google services, such as Google Search, Google Maps, Google Drive, and more.

Software developers who want to create applications for the Android OS can download the Android Software Development Kit (SDK) for a specific version. The SDK includes a debugger, libraries, an emulator, some documentation, sample code and tutorials. For faster development, interested parties

can use graphical integrated development environments (IDEs) such as Eclipse to write applications inJava.

Android Emulator:

An Android emulator is a software application that allows you to run Android operating system and applications on a non-Android device, such as a Windows or macOS computer. It creates a virtual environment that mimics the behavior of an Android device, allowing you to test, debug, and run Android apps on your computer.

Android emulators are commonly used by:

1. Developers: To test and debug their Android apps on different devices and Android versions without needing a physical device.
2. Gamers: To play Android games on their computers using a keyboard, mouse, or game controller.
3. Users: To run Android apps on their computers, such as messaging apps, social media apps, or productivity apps.

Some popular Android emulators include:

1. Android Studio Emulator (official emulator from Google)
2. BlueStacks
3. NoxPlayer
4. MEmu
5. KoPlayer

Android emulators can offer various features, such as:

- Support for different Android versions
- Customizable device settings (e.g., screen resolution, RAM)
- Support for keyboard and mouse input
- Ability to install apps from the Google Play Store
- Support for hardware acceleration (e.g., graphics processing)

However, Android emulators can also have some limitations and drawbacks, such as:

- Performance issues (e.g., slow frame rates, lag)
- Compatibility issues with certain apps or games
- Limited support for hardware features (e.g., camera, GPS)

Overall, Android emulators can be a useful tool for developers, gamers, and users who want to run Android apps on their computers.

Android versions:

Android has gone through many versions over the years, each with its own unique features and improvements. Some notable versions include:

1. Android 1.0 (2008)
2. Android 2.0 (Eclair, 2009)
3. Android 3.0 (Honeycomb, 2011)
4. Android 4.0 (Ice Cream Sandwich, 2011)
5. Android 5.0 (Lollipop, 2014)
6. Android 6.0 (Marshmallow, 2015)
7. Android 7.0 (Nougat, 2016)
8. Android 8.0 (Oreo, 2017)
9. Android 9.0 (Pie, 2018)
10. Android 10 (2019)
11. Android 11 (2020)
12. Android 12 (2021)
13. Android 13 (2022)

Android devices:

Android powers a wide range of devices, including:

1. Smart phones
2. Tablets
3. Smart watches
4. TVs
5. Set-top boxes
6. Automotive systems
7. Wearable devices

Overall, Android is a versatile and widely-used operating system that offers a range of features, customization options, and device choices.

Android 4.1! Features

Android 4.1, also known as Jelly Bean, is a version of the Android operating system released in June 2012. Here are some key features and improvements introduced in Android 4.1:

Key Features:

1. **Project Butter**: Android 4.1 introduced Project Butter, a set of performance improvements aimed at making the OS feel smoother and more responsive.
2. **Google Now**: Android 4.1 introduced Google Now, a virtual assistant that provides users with relevant information, such as weather, traffic, and sports scores.
3**. Improved Notifications**: Android 4.1 introduced expandable notifications, allowing users to view more information about a notification without having to open the associated app.
4. **Widgets**: Android 4.1 introduced a new widget system, allowing users to resize widgets and move them around on the home screen.
5. **Accessibility**: Android 4.1 introduced several accessibility features, including improved text-to-speech functionality and support for external Braille displays.

Other Improvements:

1. **Improved performance**: Android 4.1 included several performance improvements, including better memory management and improved rendering.
2. **Enhanced security**: Android 4.1 included several security enhancements, including improved malware detection and improved encryption.
3. **Better support for external devices**: Android 4.1 introduced improved support for external devices, including USB devices and Bluetooth devices.

Devices that ran Android 4.1:

Android 4.1 was available on a range of devices, including:

1. Nexus 7 (2012)
2. Galaxy Nexus
3. Nexus S
4. Motorola Xoom
5. Asus Transformer Prime

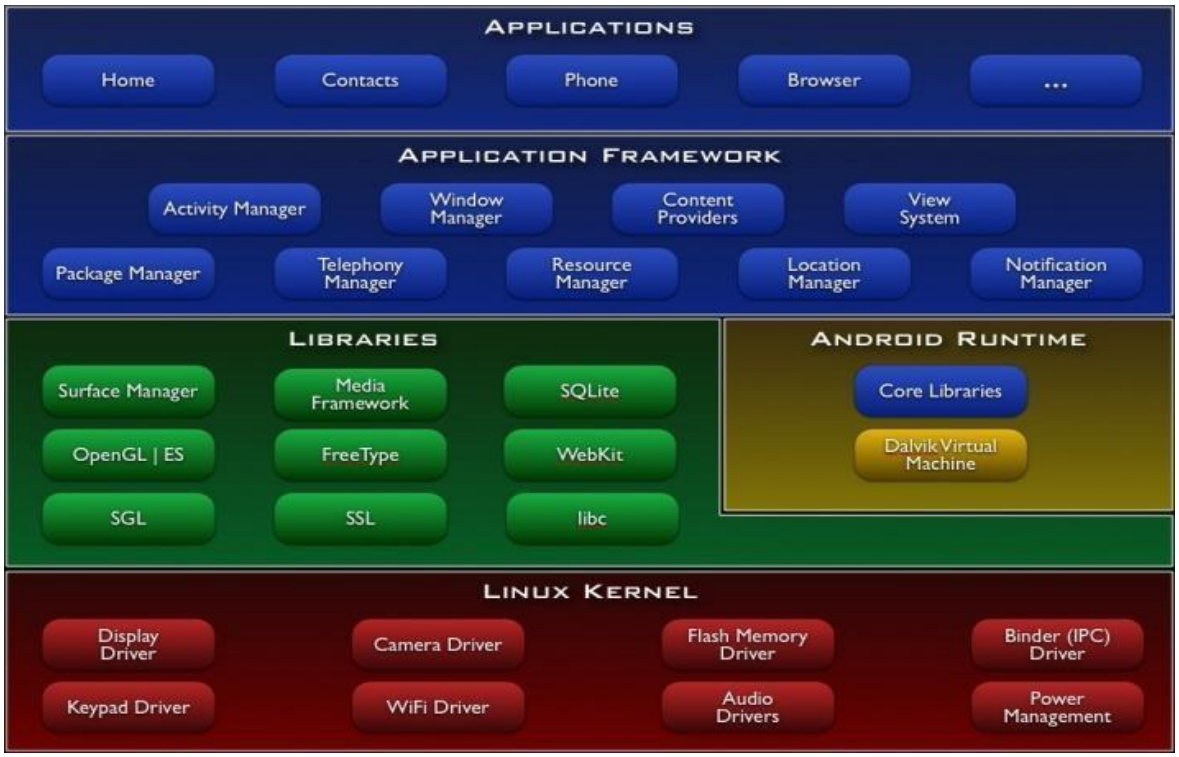
Overall, Android 4.1 was an important update that laid the foundation for future versions of the OS. Its performance improvements, new features, and enhanced security made it a popular choice among Android users.

THE ANDROID SOFTWARE STACK/ ANDROID ARCHITECTURE

The Android operating system is built on top of a modified Linux kernel. The software stack contains Java applications running on top of a virtual machine. Components of the system are written in Java, C, C++, and XML. Android operating system is a stack of software components which is roughly divided into five sections

1) Linux kernel
2) Native libraries (middleware),
3) Android Runtime
4) Application Framework

5) Applications



1) Linux kernel

It is the heart of android architecture that exists at the root of android architecture. Linux kernel is responsible for device drivers, power management, memory management, device management and resource access. This layer is the foundation of the Android Platform.

- Contains all low level drivers for various hardware components support.
- Android Runtime relies on Linux Kernel for core system services like,
- Memory, process management, threading etc.
- Network stack
- Driver model
- Security and more.

2) Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

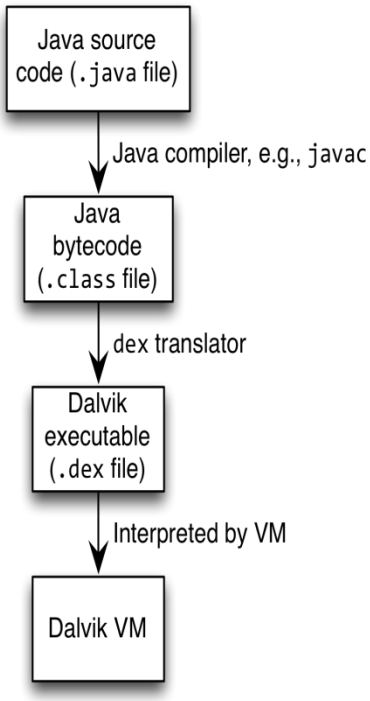
- **SQLite** Library used for data storage and light in terms of mobile memory footprints and task execution.
- **WebKit** Library mainly provides Web Browsing engine and a lot more related features.
- The **surface manager** library is responsible for rendering windows and drawing surfaces of various apps on the screen.
- The **media framework** library provides media codecs for audio and video.
- The **OpenGl** (Open Graphics Library) and **SGL**(Scalable Graphics Library) are the graphics

libraries for 3D and 2D rendering, respectively.

- The **FreeType** Library is used for rendering fonts.

3) Android Runtime

In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application. DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance. The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.



- Dalvik is a specialized virtual machine designed specifically for Android and optimized for battery-powered mobile devices with limited memory and CPU.
- Android apps execute on Dalvik VM, a "clean-room" implementation of JVM
- Dalvik optimized for efficient execution
- Dalvik: register-based VM, unlike Oracle's stack-based JVM
- Java .class bytecode translated to Dalvik EXecutable (DEX) bytecode, which Dalvik interprets

4) Android Framework

On the top of Native libraries and android runtime, there is android framework. Android framework includes Android API's such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers. It provides a lot of classes and interfaces for android application development.

- **Activity Manager**: manages the life cycle of an applications and maintains the back stack as well so that the applications running on different processes has smooth navigations.
- **Package Manager**: keeps track of which applications are installed in your device.
- **Window Manager** : Manages windows which are java programming abstractions on top of lower level surfaces provided by surface manager.
- **Telephony Managers**: manages the API which is use to build the phone applications

- **Content Providers:** Provide feature where one application can share the data with another application. like phone number , address, etc
- **View Manager** : Buttons , Edit text , all the building blocks of UI, event dispatching etc.

5) Applications

On the top of android framework, there are applications. All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries. Android runtime and native libraries are using linux kernel. Any applications that you write are located at this layer.

Android Studio Installation

First of all, Download android studio from this link: https://developer.android.com/studio/index.html

JDK 8 is required when developing for Android 5.0 and higher (JRE is not enough). To check if you have JDK installed (and which version), open a terminal and type javac -version. If the JDK is not available or the version islower than 6, download it from this link.

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

1)To set up Android Studio on **Windows**:

1. Launch the .exe file you just downloaded.
2. Follow the setup wizard to install Android Studio and any necessary SDK tools.

On some Windows systems, the launcher script does not find where Java is installed. If you encounter thisproblem, you need to set an environment variable indicating the correct location.

Select Start menu > Computer > System Properties > Advanced System Properties. Then open Advanced tab > Environment Variables and add a new system variable JAVA_HOME that points to your JDK folder, for example C:\Program Files\Java\jdk1.8.x.(where x is version number)

To set up Android Studio on Mac OSX:

Launch the .dmg file you just downloaded.

Drag and drop Android Studio into the Applications folder.

Open Android Studio and follow the setup wizard to install any necessary SDK tools.

Depending on your security settings, when you attempt to open Android Studio, you might see a warningthat says the package is damaged and should be moved to the trash. If this happens, go to System Preferences > Security & Privacy and under Allow applications downloaded from, select Anywhere. Then open Android Studio again.

If you need use the Android SDK tools from a command line, you can access them at:

/Users/<user>/Library/Android/sdk/ To set up Android Studio on **Linux**:

1. Unpack the downloaded ZIP file into an appropriate location for your applications.
   1. To launch Android Studio, navigate to the android-studio/bin/ directory in a terminal and execute studio.sh. You may want to add android-studio/bin/ to your PATH environmental variable so that you can start Android Studio from any directory.

   2. Follow the setup wizard to install any necessary SDK tools.

   Android Studio is now ready and loaded with the Android developer tools, but there are still a couple packages you should add to make your Android SDK complete.

   2)The SDK separates tools, platforms, and other components into packages you can download as needed using the Android SDK Manager. Make sure that you have downloaded all these packages.

   To start adding packages, launch the Android SDK Manager in one of the following ways:

   1. In Android Studio, click SDK Manager in the toolbar.

   2. If you're not using Android Studio:
   Windows: Double-click the SDK Manager.exe file at the root of the Android SDK directory.

   Mac/Linux: Open a terminal and navigate to the tools/ directory in the Android SDK, then execute android sdk.

   3)Now get all the SDK tools, support libraries for additional APIs, Google Play services (if you need to use them).
   6)Once you've selected all the desired packages, continue to install:

INSTALLING THE ANDROID SDK

For developing native Android applications that you can publish on the Google Play marketplace, you need to install the following four applications:

• **The Java Development Kit (JDK)** can be downloaded from http://oracle.com/technetwork/java/javase/downloads/index.html.

• **The Eclipse IDE** can be downloaded from http://www.eclipse.org/downloads/.

• **The Android Platform SDK Starter Package** can be download from http://developer.android.com/sdk/index.html.

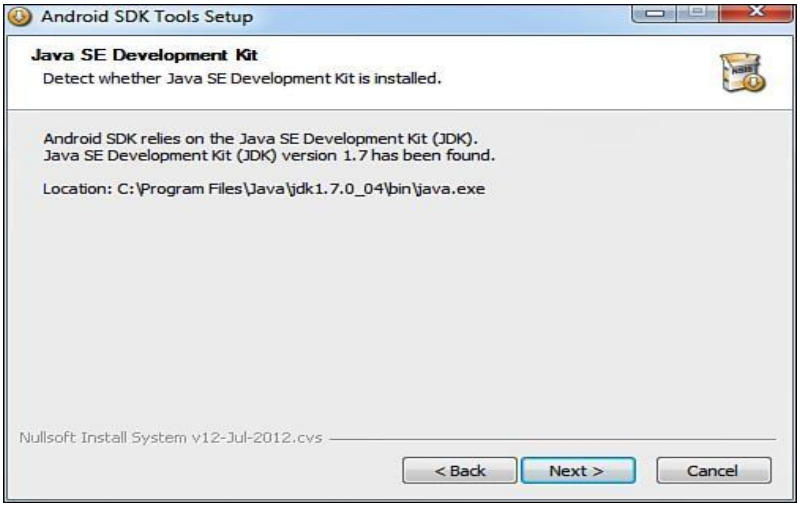
• **The Android Development Tools (ADT) Plug-in** can be downloaded from http://developer.android.com/sdk/eclipse-adt.html. The plug-in contains project templates and Eclipse tools that help in creating and managing Android projects.

The Android SDK is not a full development environment and includes only the core SDK Tools, which are used to download the rest of the SDK components. This means that after installing the Android SDK Tools, you need to install Android platform tools and the other components required for developing Android applications. Go to

http://developer.android.com/sdk/index.html and download the package by selecting the link for your operating system.

The first screen is a Welcome screen. Select the Next button to move to the next screen. Because the Android SDK requires the Java SE Development Kit for its operation, it checks for the presence of JDK on your computer.

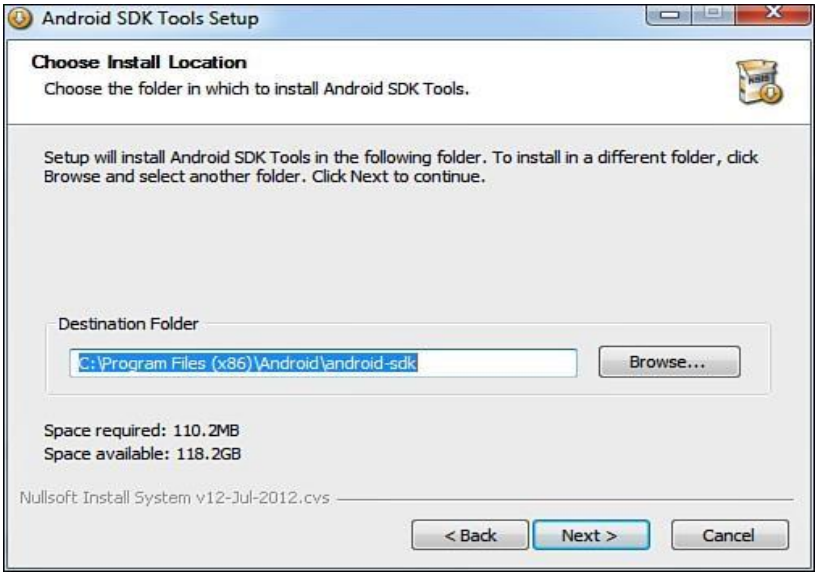
If Java is already installed on your computer before beginning with Android SDK installation, the wizard detects its presence and displays the version number of the JDK found on the machine, as shown in Figure.



Select the Next button. You get a dialog box asking you to choose the users for which Android SDK is being installed. The following two options are displayed in the dialog box:

- Install for anyone using this computer
- Install just for me

Select the Install for anyone using this computer option and click Next. The next dialog prompts you for the location to install the Android SDK Tools, as shown in below Figure . The dialog also displays the default directory location for installing Android SDK Tools as C:\Program Files (x86)\Android\android-sdk, which you can change by selecting the Browse button. Keep the default directory for installing Android SDK Tools unchanged; then select the Next button to continue.



The next dialog box asks you to specify the Start Menu folder where you want the program's shortcuts to appear, as shown.

Figure: Dialog box to select the Start menu shortcut folder

A default folder name appears called Android SDK Tools. If you do not want to make a Start Menu folder, select the Do not create shortcuts check box. Let's create the Start Menu folder by keeping the default folder name and selecting the Install button to begin the installation of the Android SDK Tools. After all the files have been downloaded and installed on the computer, select the Next button. The next dialog box tells you that the Android SDK Tools Setup Wizard is complete and the Android SDK Tools have successfully installed on the computer. Select Finish to exit the wizard, as shown in Figure

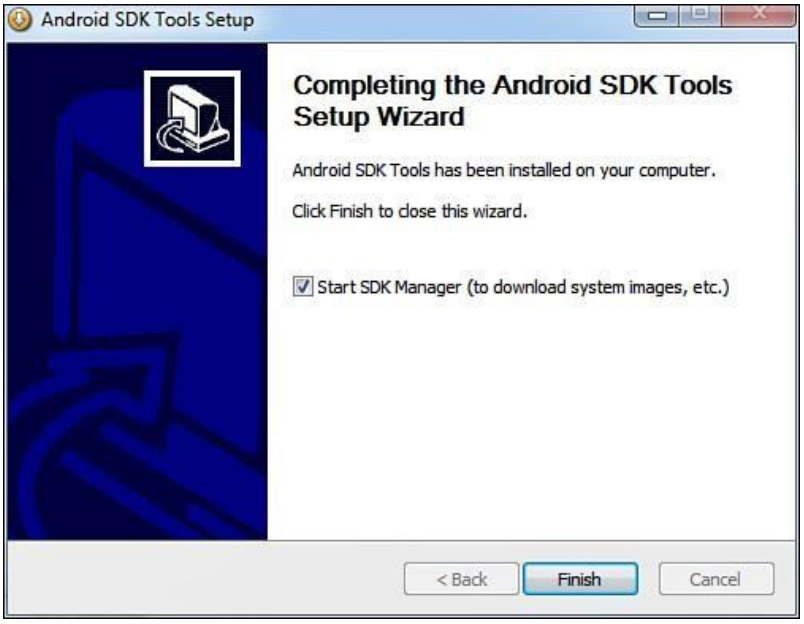


Figure: Successful installation of the Android SDK Tools dialog box

Note that the check box Start SDK Manager (to download system images) is checked by default. It means that after the Finish button is clicked, the Android SDK Manager, one of the tools in the Android SDK Tools package, will be launched. Android SDK is installed in two phases. The first phase is the installation of the SDK, which installs the Android SDK Tools, and the second phase is installation of the Android platforms and other components.

An Android application is a combination of several small components that include Java files, XML resource and layout files, manifest files, and much more. It would be very time-consuming to create all these components manually. So, you can use the following applications to help you:

• **Eclipse IDE**—An IDE that makes the task of creating Java applications easy. It provides a complete platform for developing Java applications with compiling, debugging, and testing support.

• **Android Development Tools (ADT) plug-in**—A plug-in that's added to the Eclipse IDE and automatically creates the necessary Android files so you can concentrate on the process of application development.

Before you begin the installation of Eclipse IDE, first set the path of the JDK that you installed, as it will be required for compiling the applications. To set the JDK path on Windows, right-click on My Computer and select the Properties option. From the System Properties dialog box that appears, select the Advanced tab, followed by the Environment Variables button. A dialog box, Environment Variables, pops up. In the System variables section, double-click on the Path variable. Add the full path of the JDK (C:\Program Files\Java\jdk1.7.0_04\bin\java.exe) to the path variable and select OK to close the windows.

CREATING ANDROID VIRTUAL DEVICES

An Android Virtual Device (AVD) represents a device configuration. There are many Android devices, each with different configuration. To test whether the Android application is compatible with a set of Android devices, you can create AVDs that represent their configuration. For example, you can create an AVD that represents an Android device running version 4.1 of the SDK with a 64MB SD card. After creating AVDs, you point the emulator to each one when developing and testing the application. AVDs are the easiest way of testing the application with various configurations.

To create AVDs in Eclipse, select the Window, AVD Manager option. An Android Virtual Device Manager dialog opens, as shown in Figure. The dialog box displays a list of existing AVDs, letting you create new AVDs and manage existing AVDs. Because you haven't yet defined an AVD, an empty list is displayed.

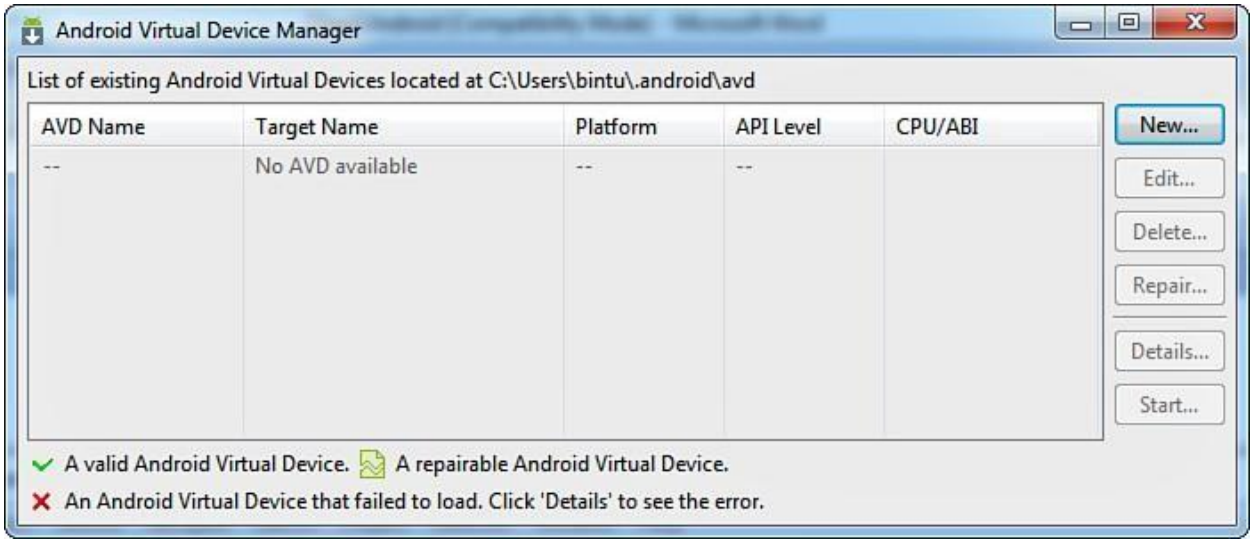


Figure: The AVD Manager dialog

Select the New button to define a new AVD. A Create new Android Virtual Device (AVD)dialog box, appears The fields are as follows:

• Name—Used to specify the name of the AVD.

• Target—Used to specify the target API level. Our application will be tested against the specified API level.

• CPU/ABI—Determines the processor that we want to emulate on our device.

• SD Card—Used for extending the storage capacity of the device. Large data files such as audio and video for which the built-in flash memory is insufficient are stored on the SD card.

• Snapshot—Enable this option to avoid booting of the emulator and start it from the last saved snapshot. Hence, this option is used to start the Android emulator quickly.

• Skin—Used for setting the screen size. Each built-in skin represents a specific screen size. You can try multiple skins to see if your application works across different devices.

• Hardware—Used to set properties representing various optional hardware that may be present in the target device.

In the AVD, set the Name of the AVD to demoAVD, choose Android 4.1—API Level 16 for the Target, set SD Card to 64 MiB, and leave the Default (WVGA800) for Skin.

In the Hardware section, three properties are already set for you depending on the selected target. The Abstracted LCD density is set to 240; the Max VM application heap size is set to 48, and the Device RAM size is set to 512.

You can select these properties and edit their values, delete them, and add new properties by selecting the New button. New properties can include Abstracted LCD density, DPad support, Accelerometer, Maximum horizontal camera pixels, Cache partition size, Audio playback support, and Track-ball support, among others.

Note

The larger the allocated SD Card space, the longer it takes to create the AVD. Unless it is really required, keep the SD Card space as low as possible. I would recommend keeping this small, like 64MiB.

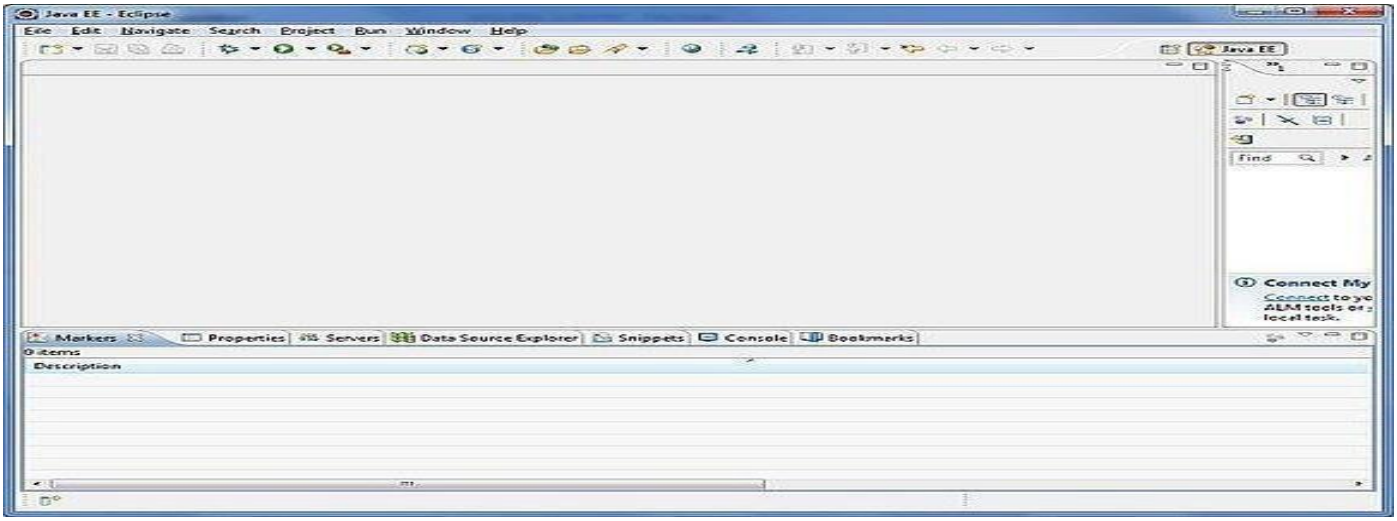
Finally, select the Create AVD button (see Figure 1.20—right) to see how to create the virtual device called demoAVD.

You now have everything ready for developing Android applications—the Android SDK, the Android platform, the Eclipse IDE, the ADT plug-in, and an AVD for testing Android applications. You can now create your first Android application.

Creating the First Android Project

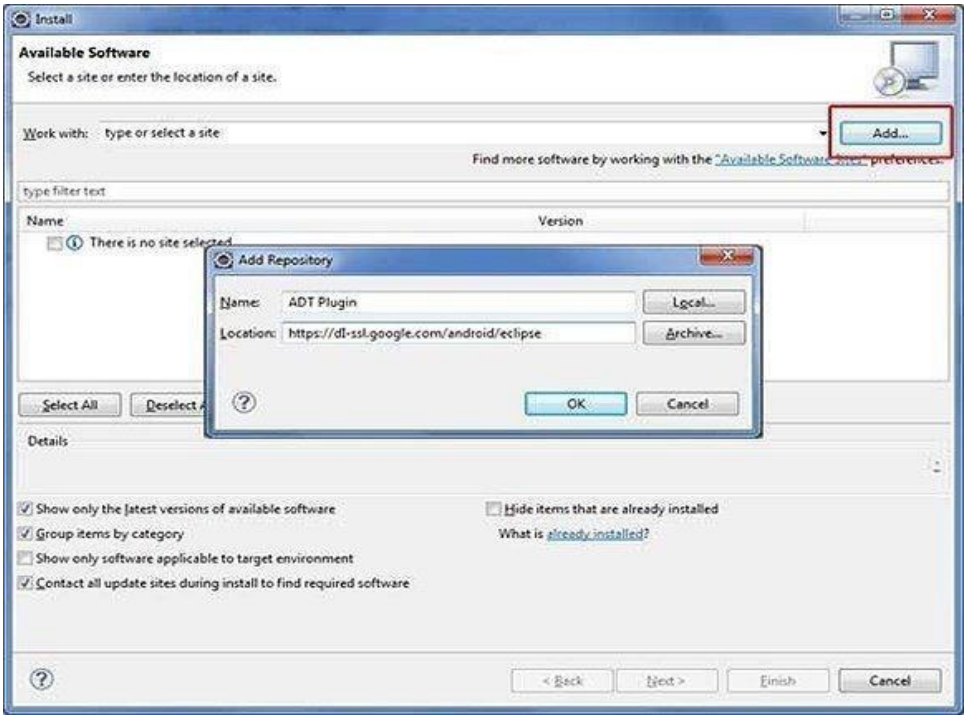
Now let's go over how to set up your first project so all you'll have left to do is write! you'll start a new Android Studio project and get to know the project workspace, including the project editor that you'll use to code the app.

Step 1: Setup Eclipse IDE:Install the latest version of Eclipse. After successful installation, it should display a

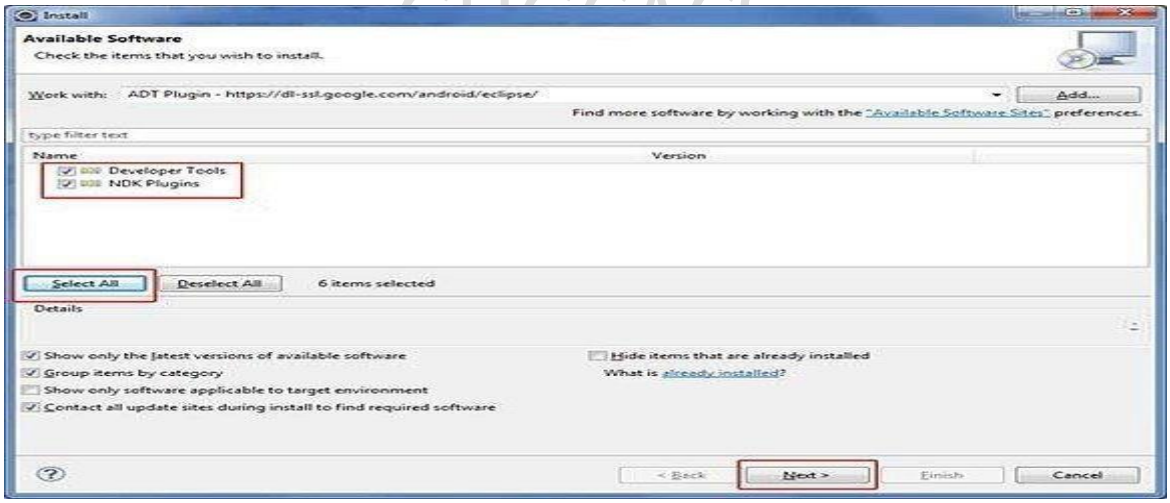


Step 2: Setup Android Development Tools (ADT) Plugin

Here you will learn to install the Android Development Tool plugin for Eclipse. To do this, you have to click on **Help > Software Updates > Install New Software**. This will display the following dialogue box.



Just click on the Add button as shown in the picture and add **https://dl-ssl.google.com/android/eclipse/** as the location. When you press OK, Eclipse will start to search for the required plug-in and finally it will list the found plug-ins.



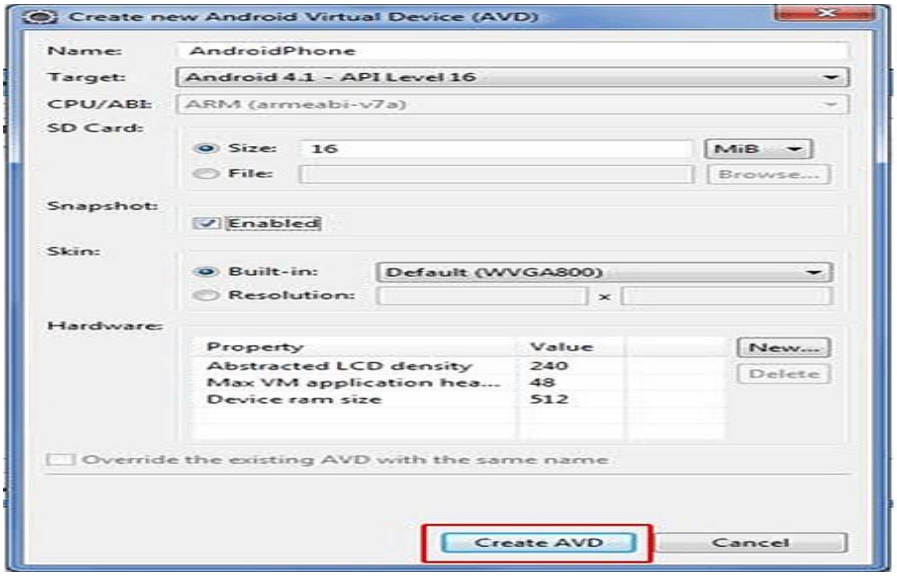
Step 3: Configuring the ADT plugin

After the installing ADT plugin, now tell the eclipse IDE for your android SDK location. To do so:

1. Select the **Window menu > preferences**
2. Now select the android from the left panel. Here you may see a dialog box asking if you want to send the statistics to the google. Click **proceed**.
3. Click on the browse button and locate your SDK directory e.g. my SDK location is C:\Program Files\Android\android-sdk .
4. Click the apply button then OK.

Step 4: Create Android Virtual Device:

The last step is to create Android Virtual Device, which you will use to test your Android applications. To do this, open Eclipse and Launch Android AVD Manager from options **Window > AVD Manager** and click on **New** which will create a successful Android Virtual Device. Use the screenshot below to enter the correct values.



USING THE TEXTVIEW CONTROL

In android **ui** or **input** controls are the interactive or View components which are used to design the user interface of an application. In android we have a wide variety of UI or input controls available, those are TextView, EditText, Buttons, Checkbox, Progressbar, Spinners, etc.

In android, **TextView** is a user interface control which is used to set and display the text to the user based on our requirements. The TextView control will act as like label control and it won't allow users to edit the text. A good example of TextView control usage would be to display textual labels for other controls, like "Enter a Date:", "Enter a Name:" or "Enter a Password:".

In android, we can create a TextView control in two ways either in XML layout file or create it in Activity file programmatically.

Specific attributes of TextView controls you will want to be aware of:

- Give the TextView control a unique name using the id property.
- Set the text displayed within the TextView control using the text property; programmatically set with the setText() method.
- Set the layout height and layout width properties of the control as appropriate.
- Set any other attributes you desire to adjust the control's appearance. For example, adjust the text size, color, font or other style settings.
- By default, text contents of a TextView control are left-aligned. However, you can position the text using the gravity attribute. This setting positions your text relative to the control's overall width and height and only really makes sense to use if there is whitespace within the TextView control.
- In XML, this property would appear within your TextView control as:

  android:gravity="center"

- By default, the background of a TextView control is transparent. That is, whatever is behind the control is shown. However, you can set the background of a control explicitly, to a color resource, or a drawable (picture). In XML, this property would appear within your TextView control as:

  android:background="#0000ff"
- By default, any text contents within a TextView control is displayed as plain text. However, by setting one simple attribute called autoLink, all you can enable automatic detection of web, email, phone and address information within the text. In XML, this property would appear within your TextView control as:

  android:autoLink="all"
- You can control the color of the text within the TextView control by using the textColor attribute. This attribute can be set to a color resource, or a specific color by hex value. In XML, this property would appear within your TextView control as:

  android:textColor="#ff0000"
- You can control the style of the text (bold, italic) and font family (sans, serif, monospace) within the TextView control by using the textStyle and typeface attributes. In XML, these properties would appear within your TextView control as:

  android:textStyle="bold"
  android:typeface="monospace"

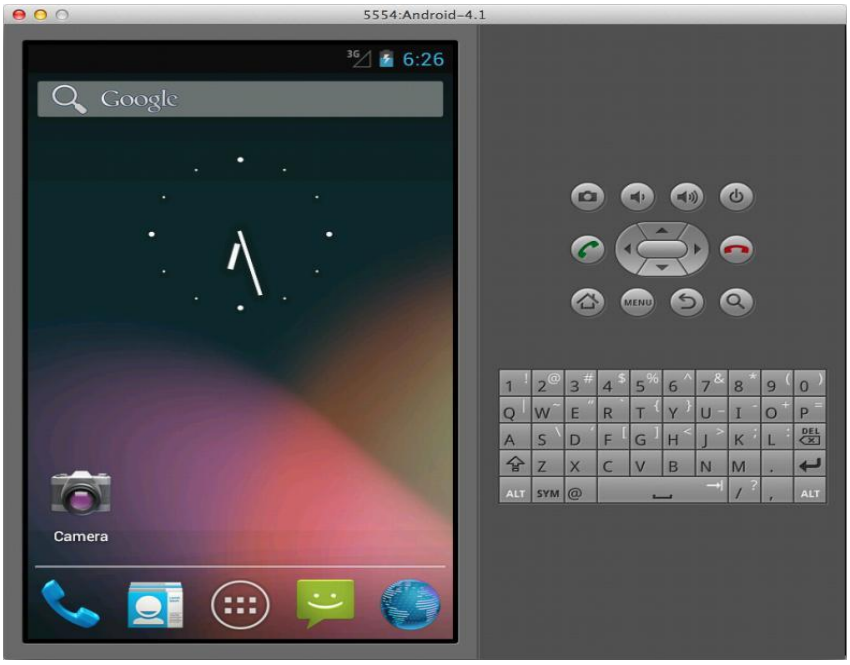
Example:


```
<TextView
   android:id="@+id/message"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   tools:context=".HelloWorldAppActivity"
   android:typeface="serif"
   android:textColor="#0F0"
   android:textSize="25dp"
   android:textStyle="italic"
   android:gravity="center_horizontal" />
```

This code makes the text of the TextView control appear in serif font, green color, 25dp size, italic, and at the horizontal center of the container

USING THE ANDROID EMULATOR

The Android emulator is used for testing and debugging applications before they are loaded onto a real handset. Android emulator is typically used for deploying apps that are developed in your IDE without actually installing it in a device. Android emulators such as Bluestacks can run android apps where in which emulators like AVD and genymotion are used to emulate an entire operating system. The Android emulator is integrated into Eclipse through the ADT plug-in.



Limitations of the Android Emulator

The Android emulator is useful to test Android applications for compatibility with devices of different configurations. But still, it is a piece of software and not an actual device and has several limitations:

- Emulators no doubt help in knowing how an application may operate within a given environment, but they still don't provide the actual environment to an application. For example, an actual device has memory, CPU, or other physical limitations that an emulator doesn't reveal.
- Emulators just simulate certain handset behavior. Features such as GPS, sensors, battery, power settings, and network connectivity can be easily simulated on a computer.
- SMS messages are also simulated and do not use a real network.
- Phone calls cannot be placed or received but are simulated.
- No support for device-attached headphones is available.
- Peripherals such as camera/video capture are not fully functional.
- No USB or Bluetooth support is available.

The emulator provides some facilities too. You can use the mouse and keyboard to interact with the emulator when it is running. For example, you can use your computer mouse to click, scroll, and drag items on the emulator. You can also use it to simulate finger touch on the soft keyboard or a physical emulator keyboard. You can use your computer keyboard to input text into UI controls and to execute specific emulator commands. Some of the most commonly used commands are

- Back [ESC button]
- Call [F3]
- End [F4]
- Volume Up [KEYPAD_PLUS, Ctrl-5]
- Volume down [KEYPAD_MINUS, Ctrl-F6]
- Switching orientations [KEYPAD_7, Ctrl-F11/KEYPAD_9, Ctrl-F12]

You can also interact with an emulator from within the DDMS tool. Eclipse IDE provides three perspectives to work with: *Java perspective, Debug perspective,* and *DDMS perspective.* The Java perspective is the default and the one with which you have been working up to now. You can switch between perspectives by choosing the appropriate icon in the top-right corner of the Eclipse environment. The three perspectives are as follows:

• **The Java perspective**—It's the default perspective in Eclipse where you spend most of the time. It shows the panes where you can write code and navigate around the project.

• **The Debug perspective**—Enables application debugging. You can set breakpoints; step through the code; view LogCat logging information, threads, and so on.

• **The Dalvik Debug Monitor Service (DDMS) perspective**—Enables you to monitor and manipulate emulator and device status. It also provides screen capture and simulates incoming phone calls, SMS sending, and GPS coordinates. To manage content in the device or emulator, you can use the ADB (Android Debug Bridge).

THE ANDROID DEBUG BRIDGE(ADB)

The Android-Debug-Bridge (abbreviated as adb) is a software-interface for the android system, which can be used to connect an android device with a computer using an USB cable or a wireless connection. It can be used to execute commands on the phone or transfer data between the device and the computer.[1]

The tool is part of the Android SDK (Android Software Development Kit) and is located in the subdirectory platform-tools. In previous versions of the SDK it was located in the subdirectory tools.

The Android Debug Bridge is a software interface between the device and the local computer, which allows the direct communication of both components. This includes the possibility to transfer files from one component to the other one, as well as executing commands from the computer on the connected device. The ADB can be used through a command line windows, terminal/shell in Linux-based systems, a command line (cmd) for Windows. The main advantage is to execute commands on the phone directly out of the computer, without any direct user interaction to the phone, which makes especially debugging a lot easier.

It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
- **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

Launching Android Applications on a Handset

To load an application onto a real handset, you need to plug a handset into your computer, using the USB data cable. You first confirm whether the configurations for debugging your application are correct and then launch the application as described here:

1. In Eclipse, choose the Run, Debug Configurations option.

2. Select the configuration HelloWorldApp_configuration, which you created for the HelloWorldApp application.

3. Select the Target tab, set the Deployment Target Selection Mode to Manual. The Manual option allows us to choose the device or AVD to connect to when using this launch configuration.

4. Apply the changes to the configuration file by clicking the Apply button.

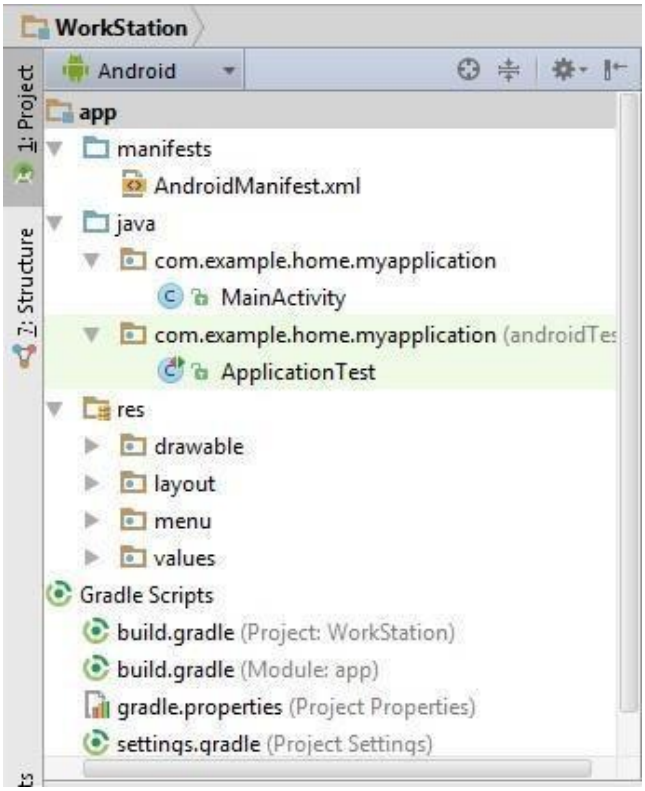
5. Plug an Android device into your computer, using a USB cable.

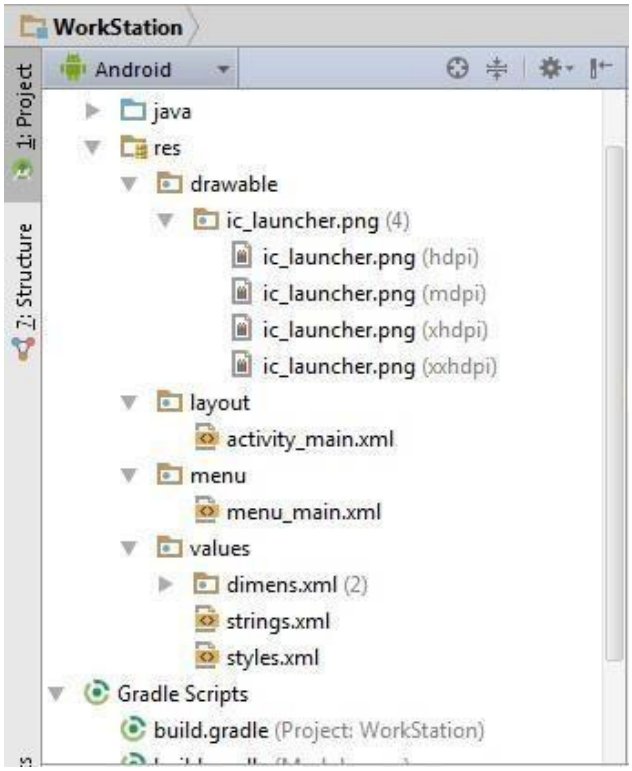
6. Select Run, Debug in Eclipse or press the F11 key. A dialog box appears, showing all available configurations for running and debugging your application. The physical device(s) connected to the computer are also listed. Double-click the running Android device. Eclipse now installs the Android application on the handset, attaches a debugger, and runs the application.

Android Studio Project Structure

Android studio shows a structured view of the project files and also provides a quick access to source files also. Android studio groups all the required source files, build files, resource files of all the modules at the top of project view.

Let`s first understand the anatomy of the Android Studio:





1) **.idea:** This folder contains the directories (subfolders) for IntelliJ IDEA settings.

2) **app:** It contains the actual application code(source files, resource file and manifest files). It further contains the following sub-directories :

a) **bulid:** This folder has sub-folders for the build-variants. The app/build/output/apk directory contains packages named app-<flavor>-<built-type>.apk. So different variant of the single app resides here.

b) **libs:** As the name suggests, this folder has all the .jar files and the library files.

3) **src:** Here you will see two sub-directories androidTest and main.

In **androidTest** the application code for testing purpose is created by android automatically.

This helps in building testing packages without modifying the building files and the application code.

main contains the all the source .java files including the stub mainactivity.java.

The **res** directory is where you will find the resources like drawable files, menu, values (style files, string values, dimension values).

Sub-directories of res folder

- anim/: For XML files that are compiled into animation objects.

- color/: For XML files that describe colors.

- drawable/:For image files (PNG, JPEG, or GIF), 9-Patch image files, and XML files that describe Drawable shapes or Drawable objects that contain multiple states (normal, pressed, or focused).
- mipmap/: For app launcher icons. This folder contains additional sub-folders for different screen resolutions. Once an image is imported into this folder, Studio automatically resizes the image into different resolutions and places them in the appropirate folders. This behavior allows launcher apps to pick the best resolution icon for your app to display on the home screen. To see how to import an image into the mipmap folder, please read about using 'Image asset'.
- layout/: XML files that are compiled into screen layouts (or parts of a parent layout).
- menu/: For XML files that define application menus.
- values/: For XML files that define constants that can be accessed in other XML and Java files.

R.java

R.java is an uneditable file auto-generated by Studio, whenever we modify the XML content. This file links the XML values to Java files. Whenever we need to use XML values in a Java file, we call these values using the R class.

The following images show a sample XML file, where a string variable named 'title' is assigned a value of 'Internshala - Tic Tac Toe'. This variable is then accessed in a java file using the R class.

4) **Gradle scripts:** With Android studio, Google switched to the new advanced building system, Gradle. It is a JVM based build system. If you want to make the package building task automatically, then you can write your own script in java or groovy and distribute it. Gradle allows to create different variants apk files for the same application project. We will learn about Gradles in the later chapters.

5) **External Libraries:** This is the place where all the referenced libraries and the information about the targeted platform SDK are stored.

What is Build system and Gradle

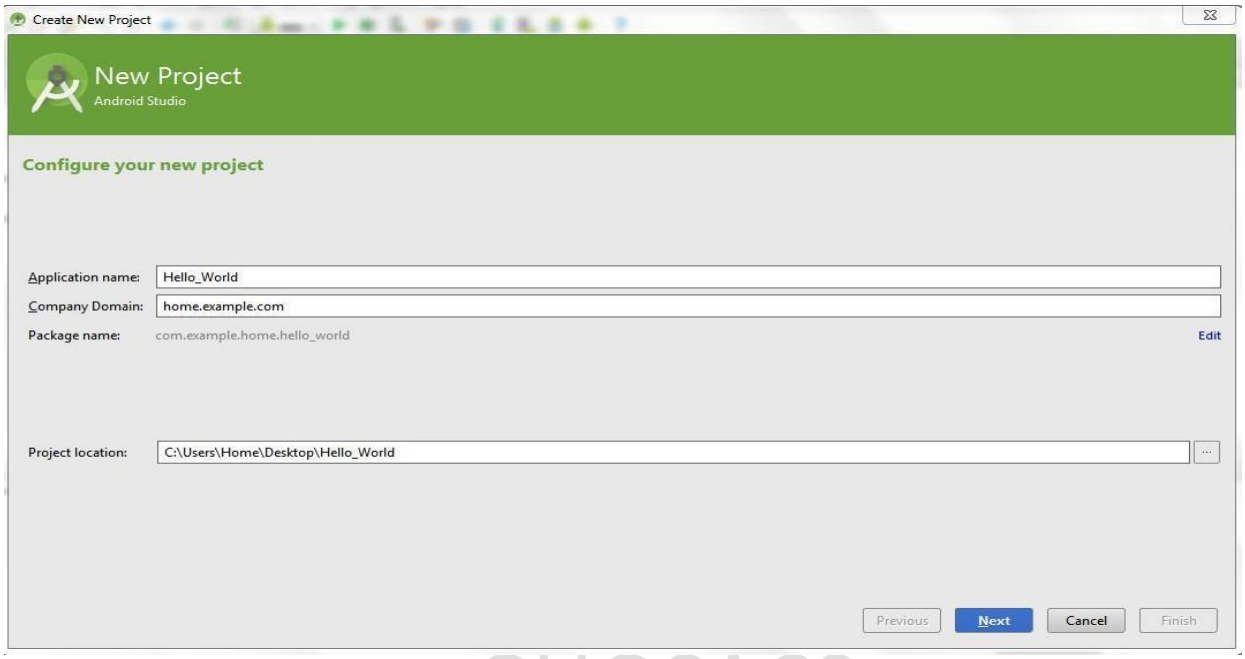
The build system is responsible to build, test an android system and also prepare the deployable files for the specified platform. In simple words, build system generates the .apk files for the application project.

With Android Studio, the advanced build system allows the developers to configure the build systems manually, create different variant APK files from single project (without modifying the execution code) and share the code and resources from other modules. Before Android Studio, Eclipse was the IDE used for android development. Eclipse kept all the .java files (in src directory) and resource files (in res directory) in the same directory. That allows the build system to group all the files into an intermediate code and finally generates .apk file.

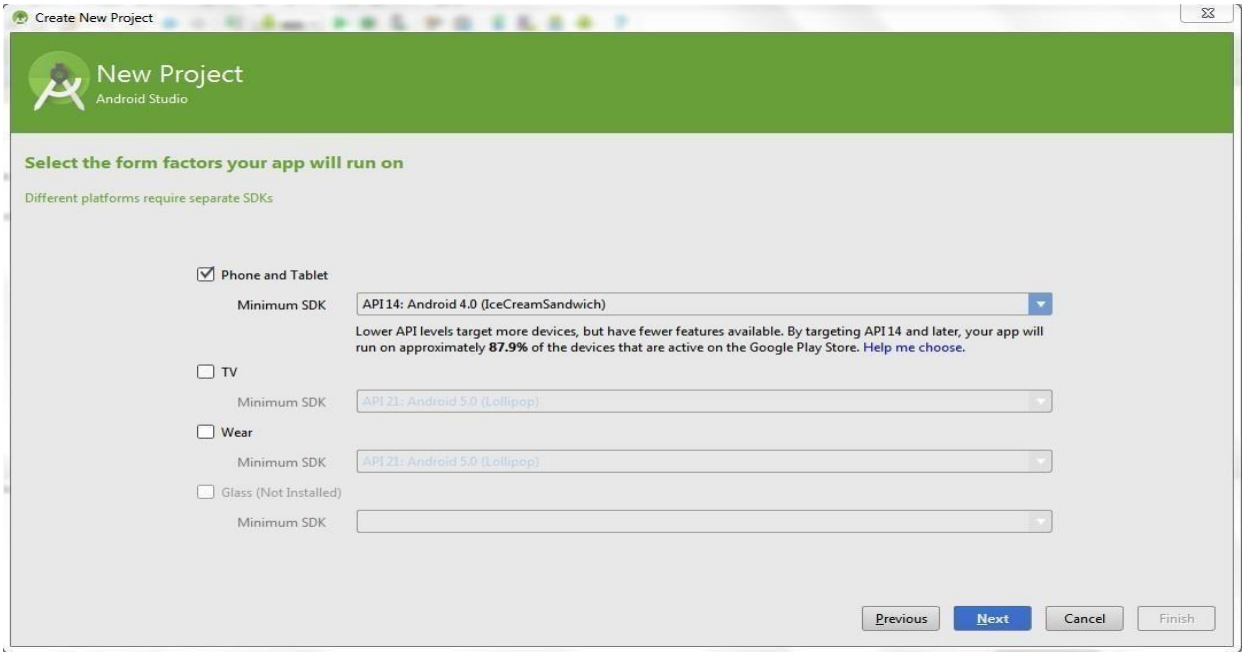
Android Studio uses Gradle as its build system. One intriguing feature that Gradle offers is that it allows you to write your own script to automate the task of building an app. As Gradle is plug-in based system, if you have your own programming language then you can write the plug-in in the script using java or groovy and share it with other developers too. Isn`t that cool?

‘Hello World’ APP

The first you will be creating will be “Hello World”! We will see how to start a project in Android Studio. First of all, launch Android Studio. Goto File-> New Project. Give the name of the application. Here we name it “Hello_World”. Hit next.



Select the category of the target devices and the desire minimum target platform of android:

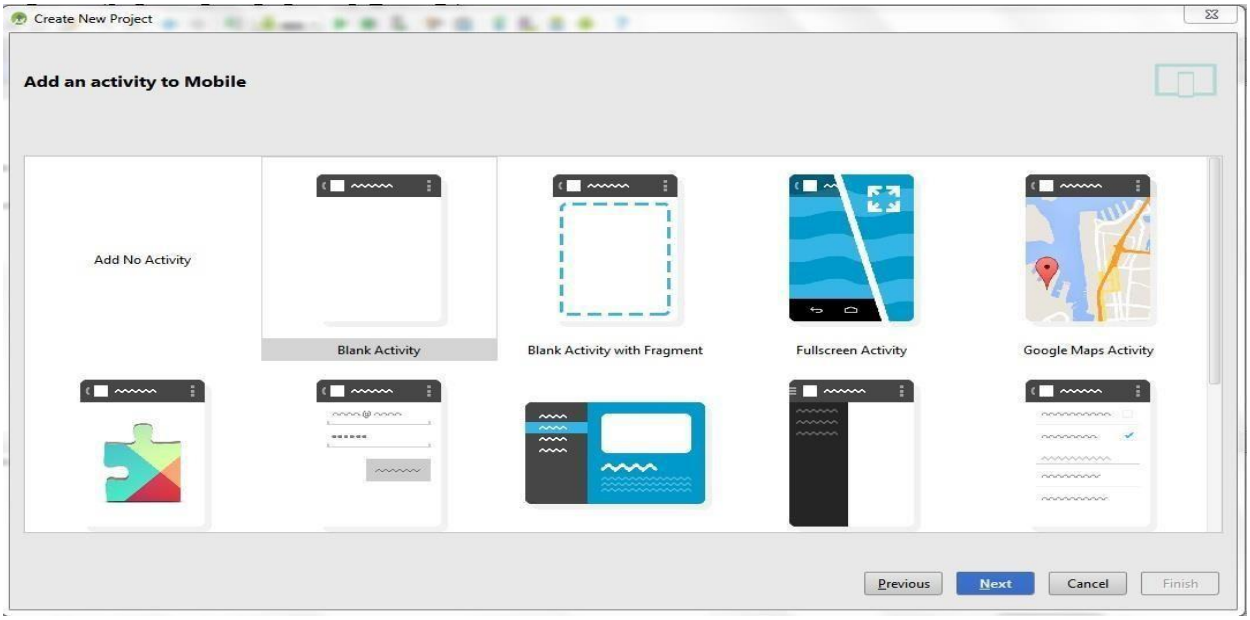


Now, select the main activity style depending upon the needs. Here for “Hello_world”,

select blank activity.

Name the main_activity files, which are the first activity to be displayed in the application. Click 'finish'.Run the Application.

Building and Running an App



First you need to build your application project before running it on the device. Click on the "build" icon fromthe toolbar and then select "Make project".

To Run an application:

Select "Run" from the toolbar and then "run app".

You can run your app on the emulator or on a real-device from Android Studio. This is done using the debugversion of the app. Run configuration defines which module will run, which activity is start, and about all theAVD settings. If you run the android application for the first time, android studio will automatically create a run configuration and choose AVD to run it. Though you can create or modify the run configuration.

Run an app on Emulator:

When you run an app on the emulator, first make sure about the AVD (Android Virtual Device). You canchoose from the available AVD or create a new AVD.
Go to Tools >Android > AVD Manager. Click on 'create virtual device'.
Choose Hardware category and configuration, Click next. Select the desired system version and create theAVD.
To run application on the desired AVD, click on the launch button.

Run an App on the real-device :

First, you have to make sure that the real-device in which you desire to run your app, is debuggable.
1) Check if the android:debuggable attribute is set to true in the build.gradle file. If not then, you cannotdebug it. Make it true in case you want to run it on real-device.
2) Please enable the USB Debugging, in the device. In android 4.2 or above, the developer option can beenabled by going to Settings > About Phone and

tapping Build Number 7 times. Now you can see the Developer Option in the previous screen.
Now, when you have your AVD set up or the real-devices ready, go to Run > Run (OR Run > Debug). If you don't see your device in the AVD window, then you need to download appropriate device drivers for your device from the internet.

Anatomy of an Android Application

Anatomy of an android application refers to the essential structural component that make up the application as a whole. The anatomy of an android application refers to its fundamental building blocks or the components of an android application and how they work together to create a functional software program.

Components are:

1. **Activities:** Activities are said to be the presentation layer of our applications. The UI of our application is built around one or more extensions of the Activity class. By using Fragments and Views, activities set the layout and display the output and also respond to the user's actions. An activity is implemented as a subclass of class Activity.

```
public class MainActivity extends Activity {
}
```

2. **Services**: Services are like invisible workers of our app. Tasks that do not require user interaction can be encapsulated in a service. These components run at the backend, updating your data sources and Activities, triggering Notification, and also broadcast Intents. They also perform some tasks when applications are not active. A service can be used as a subclass of class Service:

```
public class ServiceName extends Service {
}
```

3. **Content Providers:** It is used to manage and persist the application data also typically interacts with the SQL database. They are also responsible for sharing the data beyond the application boundaries. The Content Providers of a particular application can be configured to allow access from other applications, and the Content Providers exposed by other applications can also be configured. A content provider should be a sub-class of the class ContentProvider.

```
public class contentProviderName extends ContentProvider {
   public void onCreate(){}
}
```

4 **Broadcast Receiver**: They are known to be intent listener as they enable your application to listen to the Intents that satisfy the matching criteria specified by us.

Broad cast receiver make our application react to any received intent thereby making them perfect for creating event-driven application.

5. **Intents:** It is a powerful inter-application message-passing framework. They are extensively used throughout Android. Intents can be used to start and stop Activities and Services, to broadcast messages system-wide or to an explicit Activity, Service or Broadcast Receiver or to request action be performed on a particular piece of data.
6. **Widgets:** These are the small visual application components that you can find on the home screen of the devices. They are a special variation of Broadcast Receivers that allow us to create dynamic, interactive application components for users to embed on their Home Screen.
7. **Notifications:** Notifications are the application alerts that are used to draw the user's attention to some particular app event without stealing focus or interrupting the current activity of the user. They are generally used to grab user's attention when the application is not visible or active, particularly from within a Service or Broadcast Receiver. Examples: E-mail popups, Messenger popups, etc.

Android Terminologies

- **XML:** In Android, XML is used for designing the application's UI like creating layouts, views, buttons, text fields etc. and also used in passing data feeds from the internet.
- **View:** A view is an UI which occupies rectangular area on the screen to draw and handle user events.
- **Layout:** Layout is the parent of View it arrange all the views in a proper manner on the screen.
- Activity: An activity can be referred as your device's screen which you see user can place UI elements in any order in the created window of user's choice.
- **Emulator:** An emulator is an Android virtual device through which you can select the target Android version or platform to run and test your developed application.
- Manifest file: Manifesto file acts as a metadata for every application this file contains all the essential information about the application like app icon, app name, launcher activity and required permissions etc.
- **Service:** Service is an application component that can be used for long running background processes. It is not bounded with any activity as there is no UI. Any other

application component can start a service and this service will continue to run even when the user switches from one application to another.

- **Broadcast receiver:** Broadcast receiver is another building block of Android application development which allows you to register for system and application events. It works in such a way that even the event triggers for the first time all the registered receivers through this broadcast receiver will get notified for all the events by Android runtime.
- **Content providers:** Content providers are used to share data between two applications this can be implemented in two ways:
  1. When you want to implement the existing content provider is another application.
  2. When you want to create a new content provider that can share its data with other applications.
- **Intent:** Intent is a messaging object which can be used to communicate between two or more components like activities, services, broadcast receiver etc. Intent can also be used to start an activity or service or to deliver a broadcast message.
- **APK:** The APK (Android Package) file is the package file format used by the Android operating system for distributing and installing applications. It contains all the necessary files and resources required to run the application on an Android device.
- **Gradle:** Gradle is the build automation tool used for Android app development. It manages project dependencies, compiles source code, and produces the APK (Android Package) file for deployment.

Android Context

"Context" in Android refers to the environment in which your app is currently running. It provides essential information about your app's surroundings and allows your app to interact with the system and other apps.

Imagine you're at a party. The "context" would be everything around you at that party—the people, the music, the decorations, etc. Similarly, in an Android app, the context provides information about things like the current screen, user settings, resources, and more.

Here's a real-time example:

Let's say you're developing a weather app. When your app is launched, it needs to know various things like the user's location, the current time, and maybe even the user's preferences

(like temperature units—Celsius or Fahrenheit). All this information is provided by the context.

So, in this case:

- The user's location could be obtained from the context to fetch accurate weather data.
- The current time could be retrieved from the context to display the correct forecast.
- User preferences, such as temperature units, could be accessed from the context to customize the app's display.

The context in Android is like the background information your app needs to understand its current situation and function properly, much like you need to know what's going on at a party to have a good time!

Types of Android Context

1. Application Context
2. Activity Context
3. Service Context
4. Broadcast receiver Context
5. Fragment Receiver
6. View Receiver

In Android, there are several types of contexts, each serving different purposes and having different lifecycles. Here are some of the main types:

1. **Application Context**: This context represents the entire application and is tied to the lifecycle of the application. It's commonly used for global access to application-level resources, such as assets and preferences. You can obtain it using getApplicationContext(). For example, if we want to access a variable throughout the

android app, one has to use it via getApplicationContext().

```
import android.app.Application;
public class GlobalExampleClass extends Application {
    private String globalName;
    private String globalEmail;
    public String getName() {
        return globalName;
    }
    public void setName(String aName) {
        globalName = aName;
    }
public String getEmail() {
        return globalEmail;
    }
    public void setEmail(String aEmail) {
        globalEmail = aEmail;
    }
}
```

List of functionalities of Application Context:

- Load Resource Values
- Start a Service
- Bind to a Service
- Send a Broadcast

2. **Activity Context**: This context is tied to the lifecycle of an activity. It's typically used when you need access to resources that are tied to a specific activity, like layouts or localized strings. You can obtain it using this inside an activity or by calling getContext() from within a view. Suppose you have an activity where you want to inflate a layout or start a new activity. You would use the activity context for these operations. Here's an example:

   For example, EnquiryActivity refers to EnquiryActivity only and AddActivity refers to AddActivity only. It is tied to the life cycle of activity. It is used for the current Context. The method of invoking the Activity Context is getContext(). Some use cases of Activity Context are:

   i. The user is creating an object whose lifecycle is attached to an activity.
   ii. Whenever inside an activity for UI related kind of operations like toast, dialogue, etc.,

getContext(): It returns the Context which is linked to the Activity from which it is called. This is useful when we want to call the Context from only the current running activity.

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int pos,
long id) {

    // view.getContext() refers to the current activity view
    // Here it is used to start the activity
    Intent intent = new Intent(view.getContext(), <your java
 classname>::class.java);
    intent.putExtra(pid, ID);
    view.getContext().startActivity(intent);
}
```

List of Functionalities of Activity Context:

- Load Resource Values
- Layout Inflation
- Start an Activity
- Show a Dialog

- Start a Service
- Bind to a Service
- Send a Broadcast
- Register BroadcastReceiver

3. **Service Context**: This context is similar to the application context but is tied to a specific service. It's often used in background services for accessing resources or performing operations that require a context. You can obtain it using this inside a service or by calling getContext() from within a service.
4. **Broadcast Receiver Context**: This context is passed to a BroadcastReceiver's onReceive() method when it's triggered by a broadcast. It's typically used to perform operations in response to system-wide events, such as receiving a message or connectivity changes.

Activ

5. **Fragment Context**: Each fragment in an Android app has its own context, which is tied to the lifecycle of the fragment. It's used for accessing resources or performing operations within the fragment. You can obtain it using getContext() within a fragment.
6. **View Context:** This context is typically used within custom views or view elements. It provides access to resources and allows for inflating layouts specific to that view. You can obtain it using getContext() from within a view.

Android Activity

In Android, an activity is a fundamental component of an application's user interface. It represents a single screen with a user interface that the user can interact with. Activities play a crucial role in the navigation and flow of an Android application, as they provide the entry point for users to perform various tasks and interact with the app's functionality.

Each activity goes through various stages or a lifecycle and is managed by activity stacks. So, when a new activity starts, the previous one always remains below it. There are four stages of an activity.

1. 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.
2. 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.
3. 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.
4. 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.

Activity Lifecycle

Activities in Android have a well-defined lifecycle, consisting of several states including Created, Started, Resumed, Paused, Stopped, and Destroyed. Understanding this lifecycle is crucial for managing the behavior of your app across different states. For each stage, android provides us with a set of 7 methods that have their own significance for each stage in the life cycle. Below figure shows different stages and methods that shows the path of migration whenever an app switches from one state to another.

In Android, the Activity lifecycle refers to the series of states an Activity goes through from its creation to its destruction. Here are the different states in the Activity lifecycle:

 # 1. onCreate()

- Called when the Activity is first created.

- Used to initialize the Activity's UI and perform other setup tasks.

 # 2. onStart()

- Called when the Activity becomes visible to the user.

- Used to perform tasks that should occur when the Activity becomes visible.

 # 3. onResume()

- Called when the Activity starts interacting with the user.

- Used to perform tasks that should occur when the Activity is in the foreground.

 # 4. onPause()

- Called when the Activity is paused or partially covered by another Activity.

- Used to perform tasks that should occur when the Activity is paused.

 # 5. onStop()

- Called when the Activity is no longer visible to the user.

- Used to perform tasks that should occur when the Activity is stopped.

 # 6. onDestroy()

- Called when the Activity is destroyed.

- Used to perform tasks that should occur when the Activity is destroyed.

 # 7. onRestart()

- Called when the Activity is restarted after being stopped.

- Used to perform tasks that should occur when the Activity is restarted.

 # Activity Lifecycle Methods

Here is a summary of the Activity lifecycle methods:

Method	**Description**
onCreate()	Called when the Activity is created
onStart()	Called when the Activity becomes visible
onResume()	Called when the Activity starts interacting with the user
onPause()	Called when the Activity is paused
onStop()	Called when the Activity is stopped
onDestroy()	Called when the Activity is destroyed
onRestart()	Called when the Activity is restarted

Understanding the Activity lifecycle is crucial for building robust and efficient Android apps.

Android – Services

A service is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states:

1. **Started:** A service is started when an application component, such as an activity, starts it by calling startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
2. **Bound:** A service is bound when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

Android Service Lifecycle

A service has life cycle callback methods that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage. The following diagram on the left shows the life cycle when the service is created with startService() and the diagram on the right shows the life cycle when the service is created with bindService(): (image courtesy : android.com)

Here is the Android Service lifecycle:

 # 1. onCreate()

- Called when the service is created.

- Used to initialize the service and perform setup tasks.

 # 2. onStartCommand()

- Called when the service is started.

- Used to start the service and perform tasks that should occur when the service is started.

- Returns an integer value that indicates how the service should behave if the system kills it.

 # 3. onBind()

- Called when the service is bound to an application component.

- Used to return an IBinder object that defines the interface for the service.

 # 4. onUnbind()

- Called when the service is unbound from an application component.

- Used to clean up resources and perform tasks that should occur when the service is unbound.

 # 5. onDestroy()

- Called when the service is destroyed.

- Used to clean up resources and perform tasks that should occur when the service is destroyed.

 # Service Lifecycle Methods

| Method | Description |

| --- | --- |

| onCreate() | Called when the service is created |

| onStartCommand() | Called when the service is started |

| onBind() | Called when the service is bound to an application component |

| onUnbind() | Called when the service is unbound from an application component |

| onDestroy() | Called when the service is destroyed |

 # Service Lifecycle Scenarios

Started Service

1. onCreate()

2. onStartCommand()

3. onDestroy()

Bound Service

1. onCreate()

2. onBind()

3. onUnbind()

4. onDestroy()

Started and Bound Service

1. onCreate()

2. onStartCommand()

3. onBind()

4. onUnbind()

5. onDestroy()

Understanding the Android Service lifecycle is crucial for building robust and efficient Android apps.

Android Intents

Intents are a fundamental component of the Android system, used for communication between different components of an application and between different applications. They facilitate asynchronous communication and enable various actions such as starting activities, services, or broadcasting messages.

An intent is an abstract description of an operation to be performed. It can be used to request an action from another component of the Android system, either within the same application or across different applications.

Intent filters are a very powerful feature of the Android platform. They provide the ability to launch an activity based not only on an explicit request, but also an implicit one. For example, an explicit request might tell the system to "Start the Send Email activity in the Gmail app". By contrast, an implicit request tells the system to "Start a Send Email screen in any activity that can do the job." When the system UI asks a user which app to use in performing a task, that's an intent filter at work. You can take advantage of this feature by declaring an attribute in the element. The definition of this element includes an element and, optionally, a element and/or an element. These elements combine to specify the type of intent to which your activity can respond. For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```
<activity android:name=".ExampleActivity"
android:icon="@drawable/app_icon">
<intent-filter>
<action android:name="android.intent.action.SEND" />
<category android:name="android.intent.category.DEFAULT" />
<data android:mimeType="text/plain" />
</intent-filter>
</activity>
```

In this example, the element specifies that this activity sends data. Declaring the element as DEFAULT enables the activity to receive launch requests. The element specifies the type of data that this activity can send. The following code snippet shows how to call the activity described above:

```
val sendIntent = Intent().apply {
action = Intent.ACTION_SEND
type = "text/plain"
putExtra(Intent.EXTRA_TEXT, textMessage)
}
startActivity(sendIntent)
```

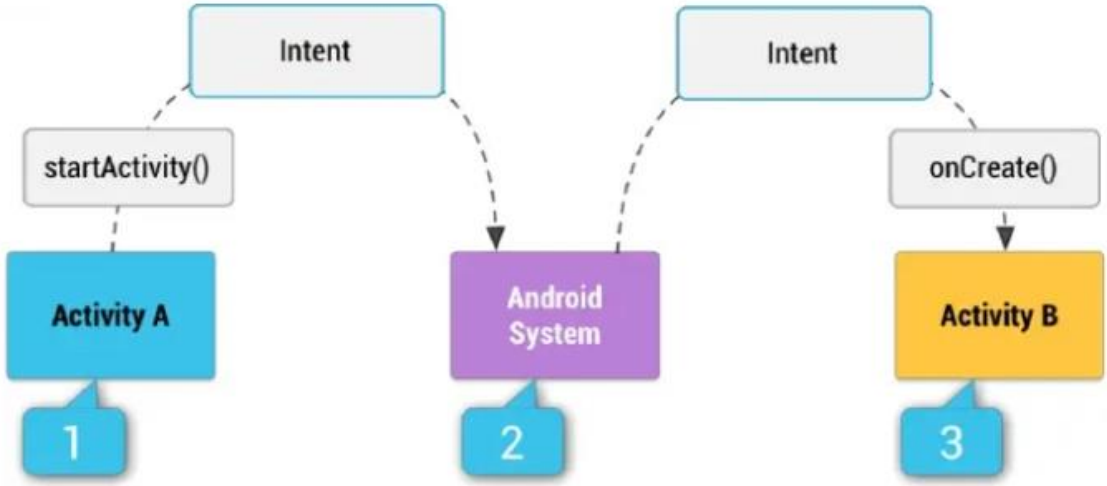
Types of Android Intents

There are two types of intents:

1. **Explicit intents:** Explicit intents specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, you might start a new activity within your app in response to a user action, or start a service to download a file in the background.

2. **Implicit intents**: Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

The following figure shows how an intent is used when starting an activity. When the Intent object names a specific activity component explicitly, the system immediately starts that component. It also shows how an implicit is delivered through the system to start another activity.



- Activity A: Creates an Intent with an action description and passes it to startActivity().
- The Android system searches all apps for an intent filter that matches the intent. When a match is found.
- The system starts the matching activity (Activity B) by invoking it onCreate() method and passing it to the intent.

Using Intent Filters

Intent filters in Android are used to declare the types of intents that a component (such as an activity, service, or broadcast receiver) can respond to. By specifying intent filters, you define the conditions under which your component should be activated.

Receiving and Broadcasting Intents

Receiving and broadcasting intents in Android is a fundamental part of building interactive and responsive applications. Here's a breakdown of how you can receive and broadcast intents in Android:

1. **Activity Intent Filter:** An activity intent filter in Android is a declaration within the AndroidManifest.xml file that specifies the types of intents that an activity can respond to. By defining intent filters for activities, you determine the conditions under which the activity should be launched or activated.

```
<activity android:name=".MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
      <category    android:name="android.intent.category.LAUNCHER"
/>
    </intent-filter>
</activity>
```

In this example,

- <action> element specifies the action that the activity can handle.
- android.intent.action.MAIN indicates that the activity is the main entry point of the application.
- <category> element specifies the category of the intent.
- android.intent.category.LAUNCHER indicates that the activity should appear in the launcher as an entry point for the application.

2. **Service Intent Filter:** In Android, service intent filters are not commonly used in the same way as activity intent filters. Services typically do not have intent filters declared in the manifest file. Instead, they are typically started or bound explicitly using intents from other components (such as activities or broadcast receivers) or system events.

```
<service android:name=".MyService">
    <intent-filter>
        <action    android:name="com.example.ACTION_START_SERVICE"
/>
    </intent-filter>
</service>
```

In this example:

- The service component is declared to handle intents with the action com.example.ACTION_START_SERVICE. This means that when an intent with this action is sent, it will activate the specified service.

3. **Broadcast Receiver Intent Filter:** In Android, broadcast receivers often utilize intent filters to specify the types of broadcast intents they are interested in receiving. Here's how you can define a broadcast receiver intent filter within the AndroidManifest.xml file:

```
<receiver android:name=".MyBroadcastReceiver">
  <intent-filter>
    <action
android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

In this example:

- The broadcast receiver is registered to listen for the android.intent.action.BOOT_COMPLETED action. This means that when the device completes booting, the specified broadcast receiver will be activated.

Receiving Intents:

1. **Broadcast Receivers:** A broadcast receiver is a component that listens for and responds to broadcasted intents. To create a broadcast receiver:
   - Extend the BroadcastReceiver class.
   - Override the onReceive() method to define the behavior when the receiver receives an intent.
   - Register the receiver either statically in the manifest file or dynamically in code using registerReceiver().

2. **Manifest-registered Receivers:** If you register a receiver in the manifest file, it will automatically receive intents broadcasted by the system, even if your app is not currently running.

```
<receiver android:name=".MyBroadcastReceiver">
    <intent-filter>
        <action
android:name="com.example.custom.intent.ACTION_NAME" />
    </intent-filter>
</receiver>
```

3. **Dynamic Receivers:** You can also register a receiver dynamically in code. This allows you to register and unregister the receiver based on the lifecycle of your app.

```
IntentFilter filter = new
IntentFilter("com.example.custom.intent.ACTION_NAME");
BroadcastReceiver receiver = new MyBroadcastReceiver();
context.registerReceiver(receiver, filter);
```

Broadcasting Intents:

1. **Creating Intents:** To broadcast an intent, create an Intent object with the appropriate action, data, and extras if needed.

```
Intent intent = new Intent("com.example.custom.intent.ACTION_NAME");
intent.putExtra("key", "value");
```

2. **Sending Broadcasts:** You can broadcast the intent using one of the following methods:
    - sendBroadcast(Intent): Sends the intent to all interested broadcast receivers.
    - sendOrderedBroadcast(Intent): Sends the intent to receivers in a specific order defined by their priority.
    - sendStickyBroadcast(Intent): Sends a sticky broadcast, which stays active even after the broadcast is complete and can be received by future broadcast receivers.

Example: Let's say you want to create a broadcast receiver that listens for a custom action and displays a toast message when it receives the intent.

1. **Create BroadcastReceiver:**

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String message = intent.getStringExtra("key");
        Toast.makeText(context, message,

Toast.LENGTH_SHORT).show();
    }
}
```

2. **Register BroadcastReceiver:**
    - Manifest registration: Add the receiver to the manifest file with the appropriate intent filter.
    - Dynamic registration: Register the receiver in your activity's onCreate() method and unregister it in onDestroy().

```
Intent intent = new
Intent("com.example.custom.intent.ACTION_NAME");
intent.putExtra("key", "Hello from Broadcast!");
sendBroadcast(intent);
```

With these steps, your broadcast receiver will listen for the custom intent and display a toast message when it receives the broadcasted intent.

Android Manifest File and its common settings

Every project in Android includes a Manifest XML file, which is AndroidManifest.xml, located in the root directory of its project hierarchy. The manifest file is an important part of our app because it defines the structure and metadata of our application, its components, and its requirements. This file includes nodes for each of the Activities, Services, Content Providers, and Broadcast Receivers that make the application, and using Intent Filters and Permissions determines how they coordinate with each other and other applications. The manifest file also specifies the application metadata, which includes its icon, version number, themes, etc. Here are some common settings and elements found in the AndroidManifest.xml file:

1. **Package Name:** The package attribute in the <manifest> element specifies the unique identifier for the application package. It must be unique across all applications installed on the device.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
```

2. **Application Components:** Activities, services, broadcast receivers, and content providers are declared as child elements of the <application> element.

```
<application
    android:allowBackup="true"
    android:icon="@drawable/app_icon"
    android:label="@string/app_name">

    <!-- Activity declarations -->
    <!-- Service declarations -->
    <!-- Receiver declarations -->
    <!-- Provider declarations -->
</application>
```

3. **Activity Declaration:** Each <activity> element defines an activity component of the application. Activities can have intent filters to specify how they are launched.

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4. **Service Declaration:** Services are declared using the <service> element. They perform background tasks without a user interface.

```
<receiver
  android:name=".MyBroadcastReceiver"
  android:exported="true">
   <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
   </intent-filter>
</receiver>
```

5. **Broadcast Receiver Declaration:** Broadcast receivers listen for and respond to broadcast messages. They are declared using the <receiver> element.

```
<receiver
    android:name=".MyBroadcastReceiver"
    android:exported="true">
    <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

6. **Content Provider Declaration:** Content providers manage access to a shared set of application data. They are declared using the <provider> element.

```
<provider
    android:name=".MyContentProvider"
    android:authorities="com.example.myapp.provider"
    android:exported="false">
</provider>
```

7. **Permissions:** The <uses-permission> element specifies the permissions required by the application to access certain features or resources on the device.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```

8. **Application Metadata:** Metadata tags can be used to provide additional information about the application.

```
<meta-data
    android:name="com.google.android.gms.ads.APPLICATION_ID"
    android:value="ca-app-pub-1234567890"/>
```

9. **Versioning and Targeting:** Attributes like android:versionCode and android:versionName specify the version information of the application.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp"
    android:versionCode="1"
    android:versionName="1.0">
```

10. **Minimum SDK Version:** The android:minSdkVersion attribute specifies the minimum API level required by the application to run.

```
<uses-sdk
    android:minSdkVersion="16"
    android:targetSdkVersion="31" />
```

Android Permissions

Permissions in Android are a crucial aspect of security and privacy management, allowing applications to access sensitive data and perform restricted operations. Android applications need to declare the permissions they require in their manifest files.

Declaring Permissions

Permissions are declared in the AndroidManifest.xml file using the <uses-permission> element. Each permission specifies a specific operation or resource that the application needs access to.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Permission Levels

Permissions in Android are categorized into different levels based on their sensitivity and potential risk to user privacy:

- **Normal Permissions:** These permissions are granted automatically when the application is installed and do not require explicit user consent.
- **Dangerous Permissions:** These permissions are considered sensitive and require the user to grant permission at runtime on devices running Android 6.0 (API level 23) and higher.

Requesting Dangerous Permissions

For dangerous permissions, developers need to request permission from the user at runtime. This involves checking whether the permission is already granted and, if not, requesting it from the user.

```
// Check if permission is not granted

if (ContextCompat.checkSelfPermission(this,
Manifest.permission.CAMERA)

 if (ContextCompat.checkSelfPermission(this,
 Manifest.permission.CAMERA)

         != PackageManager.PERMISSION_GRANTED) {

     // Request permission

     ActivityCompat.requestPermissions(this,

             new String[]{Manifest.permission.CAMERA},

             MY_PERMISSIONS_REQUEST_CAMERA);

 }
```

Handling Permission Results

After requesting permissions at runtime, the application receives the result in the onRequestPermissionsResult() method. Developers should handle permission grant or denial accordingly.

```
@Override
public void onRequestPermissionsResult(int requestCode,
@NonNull String[] permissions,@NonNull int[] grantResults) {
    if (requestCode == MY_PERMISSIONS_REQUEST_CAMERA) {
        if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
            // Permission granted, proceed with the operation
        } else {
            // Permission denied, handle accordingly
        }
    }
}
```

Common Android Permissions

Common Android Permissions

Here are some common permissions used in Android applications:

- android.permission.INTERNET: Allows the application to access the internet.
- android.permission.ACCESS_NETWORK_STATE: Allows the application to access information about networks.
- android.permission.READ_EXTERNAL_STORAGE: Allows the application to read from external storage.
- android.permission.WRITE_EXTERNAL_STORAGE: Allows the application to write to external storage.
- android.permission.CAMERA: Allows the application to access the device camera.
- android.permission.ACCESS_FINE_LOCATION: Allows the application to access precise location information.

Note:

- It's essential to request permissions only when they are necessary for the application's functionality to avoid unnecessary requests and user inconvenience.
- Carefully consider the implications of requesting sensitive permissions and ensure transparent communication with users regarding why the permissions are needed.