The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.

For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the **manifest**.

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.

You can take advantage of this feature by declaring an <intent-filter> attribute in the <activity> element.

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

Intent filter inside activity

```
<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>
```

Managing the activity lifecycle

onCreate()

You must implement this callback, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here. Most importantly, this is where you must call `setContentView()` to define the layout for the activity's user interface.

onStart()

As `onCreate()` exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity’s final preparations for coming to the foreground and becoming interactive.

onResume()

The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input. Most of an app’s core functionality is implemented in the `onResume()` method.

onPause()

The system calls `onPause()` when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button. When the system calls `onPause()` for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

onStop()

The system calls `onStop()` when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible at all.

onRestart()

The system invokes this callback when an activity in the Stopped state is about to restart. `onRestart()` restores the state of the activity from the time that it was stopped.

This callback is always followed by `onStart()`.

onDestroy()

The system invokes this callback before an activity is destroyed.

App architecture

Separation of concerns

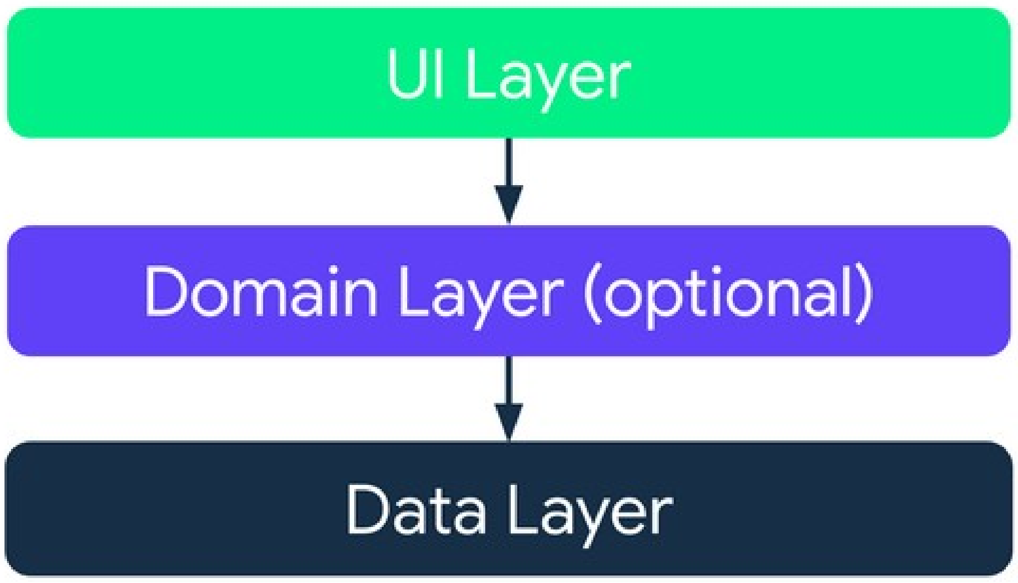
The most important principle to follow is separation of concerns. It's a common mistake to write all your code in an `Activity` or a `Fragment`. These UI-based classes should only contain logic that handles UI and operating system interactions. By keeping these classes as lean as possible, you can avoid many problems related to the component lifecycle, and improve the testability of these classes.

Drive UI from data models

Another important principle is that you should drive your UI from data models, preferably persistent models. Data models represent the data of an app. They're independent from the UI elements and other components in your app. This means that they are not tied to the UI and app component lifecycle, but will still be destroyed when the OS decides to remove the app's process from memory.

- The *UI layer* that displays application data on the screen.
- The *data layer* that contains the business logic of your app and exposes application data.

You can add an additional layer called the *domain layer* to simplify and reuse the interactions between the UI and data layers.



UI layer

- UI elements that render the data on the screen. You build these elements using Views or Jetpack Compose functions.
- State holders (such as ViewModel classes) that hold data, expose it to the UI, and handle logic.

Data layer

The data layer of an app contains the *business logic*. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.

The data layer is made of *repositories* that each can contain zero to many *data sources*. You should create a repository class for each different type of data you handle in your app. For example, you might create a `MoviesRepository` class for data related to movies, or a `PaymentsRepository` class for data related to payments.

Repository classes are responsible for the following tasks:

- Exposing data to the rest of the app.
- Centralizing changes to the data.
- Resolving conflicts between multiple data sources.
- Abstracting sources of data from the rest of the app.
- Containing business logic.

Domain layer

The domain layer is an optional layer that sits between the UI and data layers.

The domain layer is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. You should use it only when needed—for example, to handle complexity or favor reusability.

Manage dependencies between components

Classes in your app depend on other classes in order to function properly. You can use either of the following design patterns to gather the dependencies of a particular class:

- Dependency injection (DI): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies.

These patterns allow you to scale your code because they provide clear patterns for managing dependencies without duplicating code or adding complexity. Furthermore, these patterns allow you to quickly switch between test and production implementations.

General best practices

Don't store data in app components.

Reduce dependencies on Android classes.

Your app components should be the only classes that rely on Android framework SDK APIs such as `Context`, or `Toast`. Abstracting other classes in your app away from them helps with testability and reduces coupling within your app.

Create well-defined boundaries of responsibility between various modules in your app.

Consider how to make each part of your app testable in isolation.

Persist as much relevant and fresh data as possible.

NAVIGATION

NavHost — view thath have to consist fragments

- The `app:navGraph` attribute associates the `NavHostFragment` with a navigation graph. The navigation graph specifies all of the destinations in this `NavHostFragment` to which users can navigate.
- The `app:defaultNavHost="true"` attribute ensures that your `NavHostFragment` intercepts the system Back button
- The `android:name` attribute contains the class name of your `NavHost` implementation.

Navigate to a destination

Navigating to a destination is done using a `NavController`, an object that manages app navigation within a `NavHost`. Each `NavHost` has its own corresponding `NavController`. You can retrieve a `NavController` by using one of the following methods:

```
val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment)
as NavHostFragment
val navController = navHostFragment.navController
```

SafeArgs

Это обёртка над Boundle для передачи данных между фрагментами Он создаст класс `GameFragmentDirections` в котором будут все наши акшкны на переход между экранами, в эти функции передадим значение которое хотим передать. Использывание SafeArgs поможет увеличить сохранность данных при передачи и обеспечит получение defaultArguments если данные переданы не будут
1) Передаём данные из первого фрагмента как параметры актион

```
view.findNavController()
    .navigate(GameFragmentDirections
        .actionGameFragmentToGameWonFragment(numQuestions,
questionIndex))
```

2) Получаем данные в конструторе второго фрагмента

```
val args
=GameWonFragmentArgs.fromBundle(requireArguments())
Toast.makeText(context, "NumCorrect: ${args.numCorrect},
NumQuestions: ${args.numQuestions}",
Toast.LENGTH_LONG).show()
```

Navigation state is represented as a stack of destinations

When your app is first launched, a new task is created for the user, and app displays its start destination. This becomes the base destination of what is known as the *back stack* and is the basis for your app's navigation state. The top of the stack is the current screen, and the previous destinations in the stack represent the history of where you've been. The back stack always has the start destination of the app at the bottom of the stack.

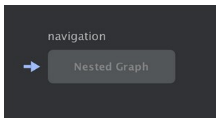
Operations that change the back stack always operate on the top of the stack, either by pushing a new destination onto the top of the stack or popping the top-most destination off the stack. Navigating to a destination pushes that destination on top of the stack.

Navigation state is represented as a stack of destinations

Global actions

You can use a *global action* to create a common action that multiple destinations can use. For example, you might want buttons in different destinations to navigate to the same main app screen.

A global action is represented in the Navigation Editor by a small arrow that points to the associated destination, as shown in figure 1.



DeepLink and nested gaph on my notebook

Android Conditional Navigation

This meen that you haven`t to build destination and navigate for screens by id

Create an explicit deep link

An [explicit deep link](#) is a single instance of a deep link that uses a [`PendingIntent`](#) to take users to a specific location within your app. You might surface an explicit deep link as part of a notification or an app widget,

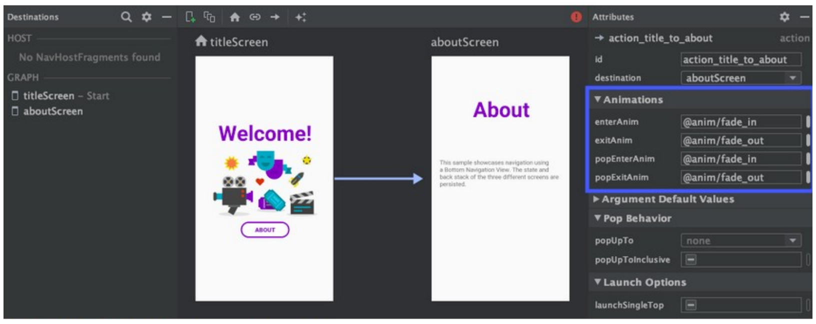
Create an implicit deep link

An [implicit deep link](#) refers to a specific destination in an app. When the deep link is invoked—for example, when a user clicks a link—Android can then open your app to the corresponding destination.

Deep links can be matched by URI, intent actions, and MIME types. You can specify multiple match types for a single deep link, but note that URI argument matching is prioritized first, followed by action, and then MIME type.

Animate transitions between destinations

The Navigation component lets you add both property and view animations to actions. To create your own animations, check out [Animation resources](#).



action_title_to_about action
id action_title_to_about
destination aboutScreen
Animations
enterAnim @anim/fade_in
exitAnim @anim/fade_out
popEnterAnim @anim/fade_in
popExitAnim @anim/fade_out
Argument Default Values
Pop Behavior
popUpTo none
popUpToInclusive
Launch Options
launchSingleTop

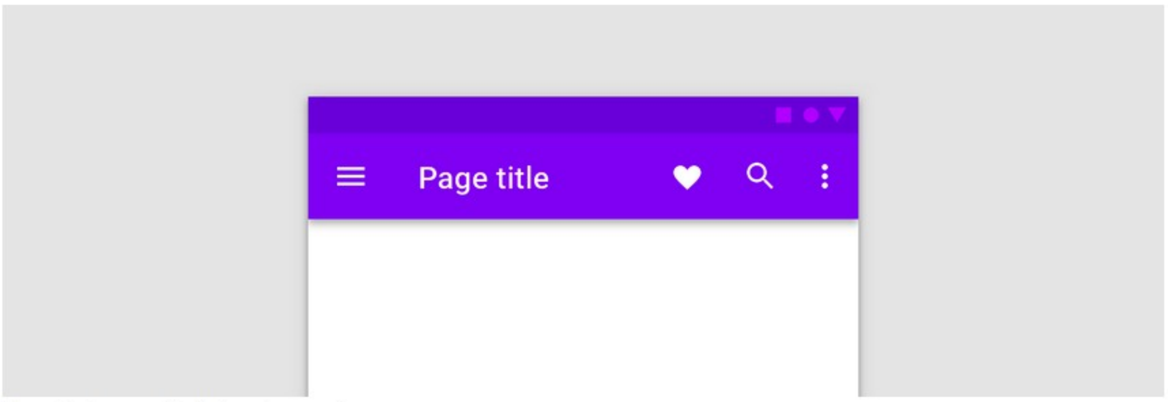


Figure 1. A screen displaying a top app bar.

`NavigationUI` contains methods that automatically update content in your top app bar as users navigate through your app. For example, `NavigationUI` uses the destination labels from your navigation graph to keep the title of the top app bar up-to-date.

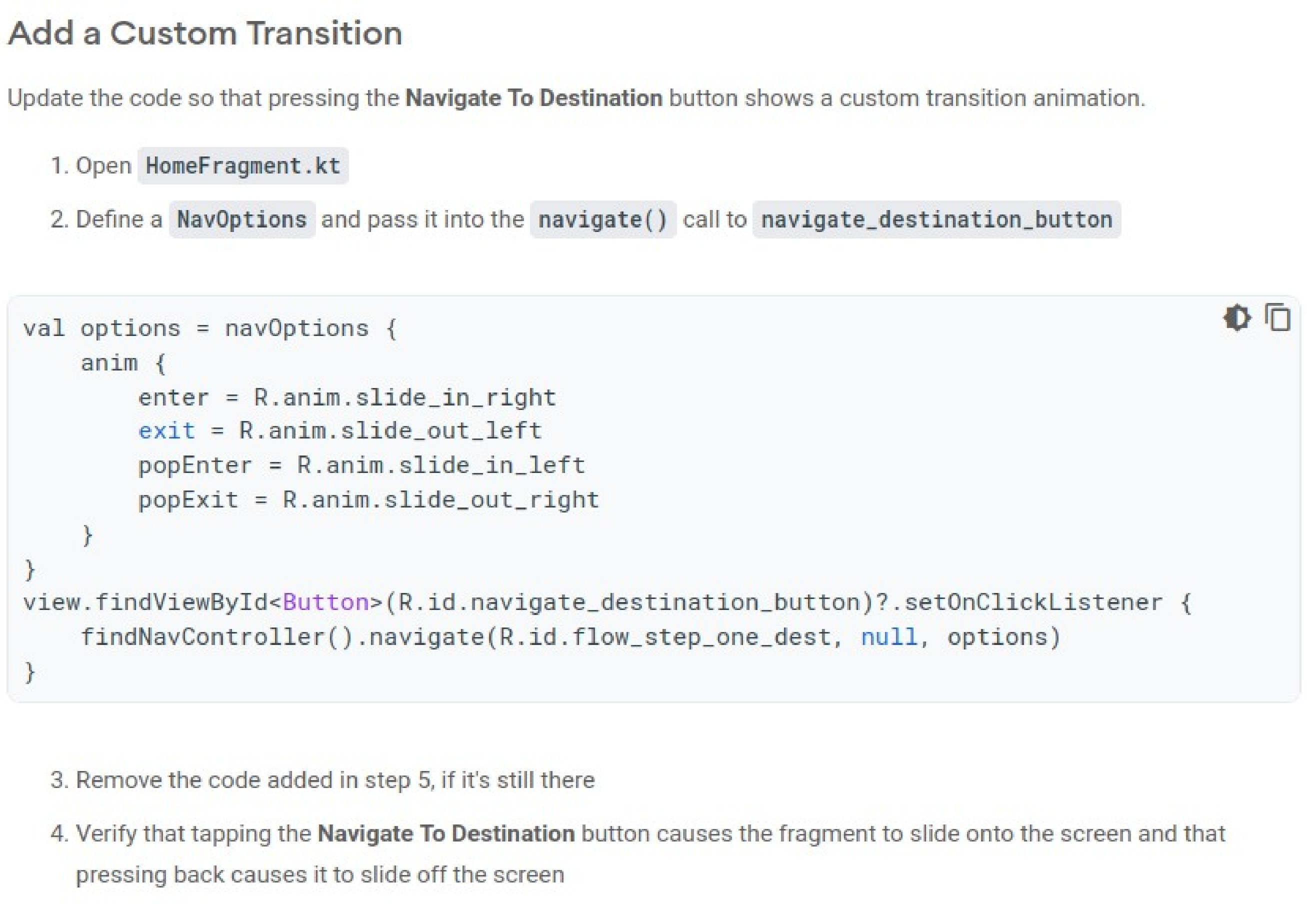
```
<navigation>
    <fragment ...
              android:label="Page title">
      ...
    </fragment>
</navigation>
```

Test fragment Navigation

```
@RunWith(AndroidJUnit4::class)
class TitleScreenTest {

    @Test
    fun testNavigationToInGameScreen() {
        // Create a TestNavHostController
        val navController = TestNavHostController(
            ApplicationProvider.getApplicationContext())

        // Create a graphical FragmentScenario for the TitleScreen
        val titleScenario = launchFragmentInContainer<TitleScreen>()

        titleScenario.onFragment { fragment ->
            // Set the graph on the TestNavHostController
            navController.setGraph(R.navigation.trivia)

            // Make the NavController available via the findNavController() APIs
            Navigation.setViewNavController(fragment.requireView(), navController)
        }

        // Verify that performing a click changes the NavController's state
        onView(ViewMatchers.withId(R.id.play_btn)).perform(ViewActions.click())
        assertThat(navController.currentDestination?.id).isEqualTo(R.id.in_game)
    }
}
```

```
  // This callback will only be called when MyFragment is at least Started.
val callback = requireActivity().onBackPressedDispatcher.addCallback(this) {
    // Handle the back button event
}
```

Add a Custom Transition



Add a Custom Transition

Update the code so that pressing the **Navigate To Destination** button shows a custom transition animation.

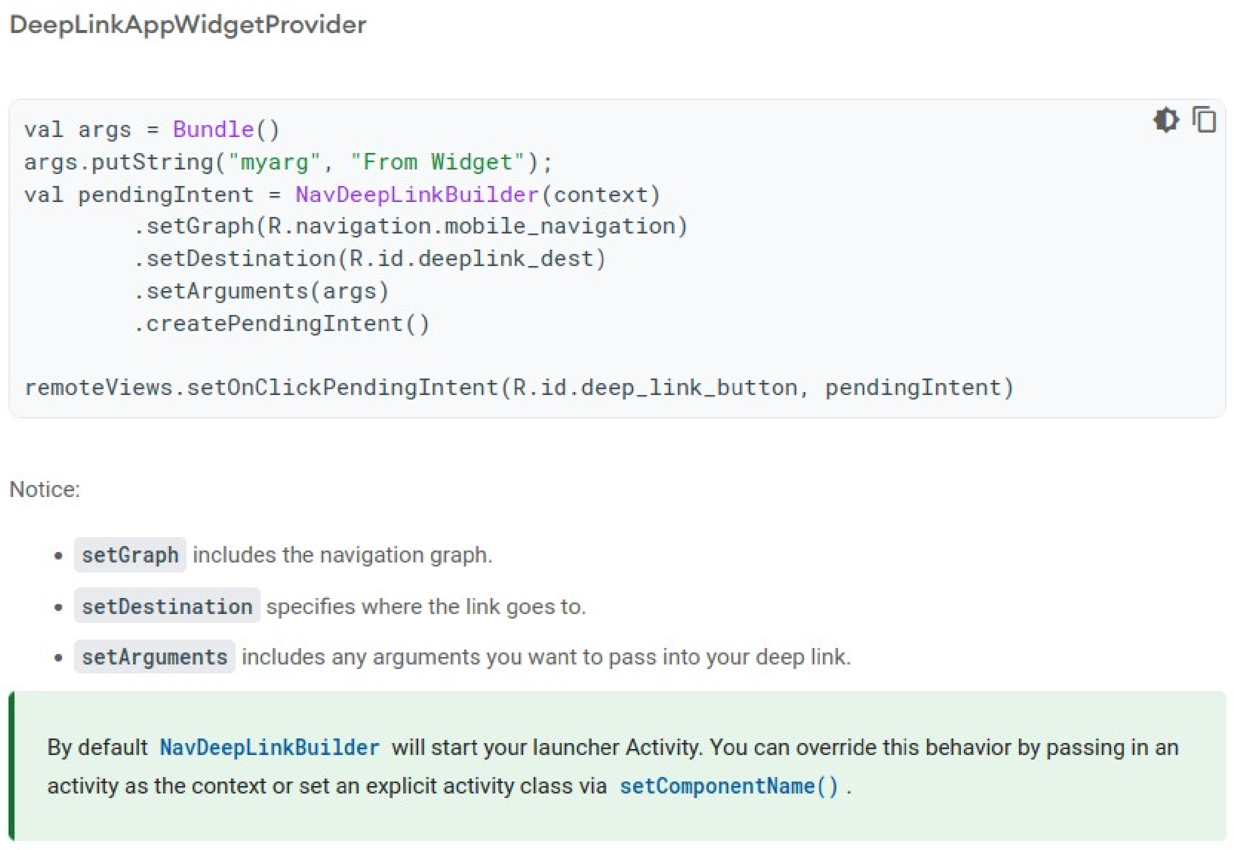
1. Open `HomeFragment.kt`
2. Define a `NavOptions` and pass it into the `navigate()` call to `navigate_destination_button`

```
val options = navOptions {
    anim {
        enter = R.anim.slide_in_right
        exit = R.anim.slide_out_left
        popEnter = R.anim.slide_in_left
        popExit = R.anim.slide_out_right
    }
}
view.findViewById<Button>(R.id.navigate_destination_button)?.setOnClickListener {
    findNavController().navigate(R.id.flow_step_one_dest, null, options)
}
```

3. Remove the code added in step 5, if it's still there
4. Verify that tapping the **Navigate To Destination** button causes the fragment to slide onto the screen and that pressing back causes it to slide off the screen

Navigation by **actions** (NOT ID) has the following benefits over navigation by destination:

- You can visualize the navigation paths through your app
- Actions can contain additional associated attributes you can set, such as a transition animation, arguments values, and backstack behavior
- You can use the plugin safe args to navigate, which you'll see shortly


DeepLinkAppWidgetProvider

```
val args = Bundle()
args.putString("myarg", "From Widget");
val pendingIntent = NavDeepLinkBuilder(context)
        .setGraph(R.navigation.mobile_navigation)
        .setDestination(R.id.deeplink_dest)
        .setArguments(args)
        .createPendingIntent()

remoteViews.setOnClickPendingIntent(R.id.deep_link_button, pendingIntent)
```

Notice:

- `setGraph` includes the navigation graph.
- `setDestination` specifies where the link goes to.
- `setArguments` includes any arguments you want to pass into your deep link.

By default `NavDeepLinkBuilder` will start your launcher Activity. You can override this behavior by passing in an activity as the context or set an explicit activity class via `setComponentName()` .

Must read

https://developer.android.com/codelabs/android-navigation#10 — Deeplink in browser

Intents and Intent Filters

An `Intent` is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

- **Starting an activity**

  An `Activity` represents a single screen in an app. You can start a new instance of an `Activity` by passing an `Intent` to `startActivity()`. The `Intent` describes the activity to start and carries any necessary data.

  If you want to receive a result from the activity when it finishes, call `startActivityForResult()`. Your activity receives the result as a separate `Intent` object in your activity's `onActivityResult()` callback. For more information, see the Activities guide.

- **Starting a service**

A `Service` is a component that performs operations in the background without a user interface. With Android 5.0 (API level 21) and later, you can start a service with `JobScheduler`. For more information about `JobScheduler`, see its `API-reference documentation`.

For versions earlier than Android 5.0 (API level 21), you can start a service by using methods of the `Service` class. You can start a service to perform a one-time operation (such as downloading a file) by passing an `Intent` to `startService()`. The `Intent` describes the service to start and carries any necessary data.

If the service is designed with a client-server interface, you can bind to the service from another component by passing an `Intent` to `bindService()`. For more information, see the Services guide.

- **Delivering a broadcast**

  A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an `Intent` to `sendBroadcast()` or `sendOrderedBroadcast()`.

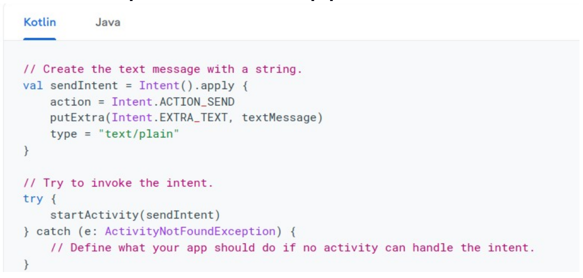
Intent types

There are two types of 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.
- **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.

Example implicit intent

An implicit intent specifies an action that can invoke any app on the device able to perform the action. Using an implicit intent is useful when your app cannot perform the action, but other apps probably can and you'd like the user to pick which app to use.



```
// Create the text message with a string.
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, textMessage)
    type = "text/plain"
}

// Try to invoke the intent.
try {
    startActivity(sendIntent)
} catch (e: ActivityNotFoundException) {
    // Define what your app should do if no activity can handle the intent.
}
```

Receiving an implicit intent

To advertise which implicit intents your app can receive, declare one or more intent filters for each of your app components with an `<intent-filter>` element in your manifest file. Each intent filter specifies the type of intents it accepts based on the intent's action, data, and category. The system delivers an implicit intent to your app component only if the intent can pass through one of your intent filters.

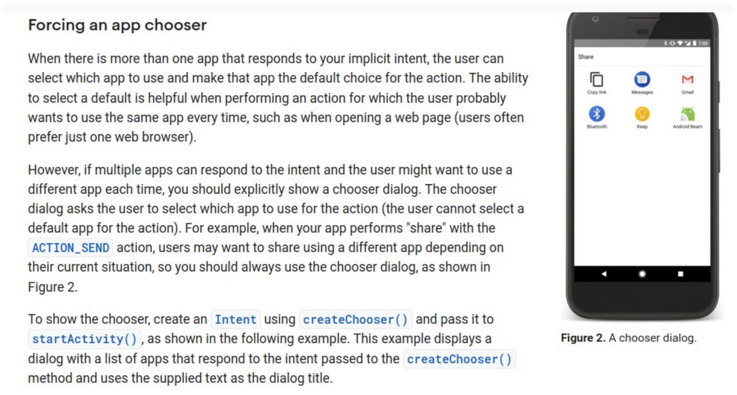
```
<activity android:name="ShareActivity" android:exported="false">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

Use explicit intents within pending intents

To better define how other apps can use your app's pending intents, always wrap a pending intent around an explicit intent. To help follow this best practice, do the following:

1. Check that the action, package, and component fields of the base intent are set.
2. Use `FLAG_IMMUTABLE`, added in Android 6.0 (API level 23), to create pending intents. This flag prevents apps that receive a `PendingIntent` from filling in unpopulated properties. If your app's `minSdkVersion` is 22 or lower, you can provide safety and compatibility together using the following code:

```
if (Build.VERSION.SDK_INT >= 23) {
  // Create a PendingIntent using FLAG_IMMUTABLE.
} else {
  // Existing code that creates a PendingIntent.
}
```



Forcing an app chooser

When there is more than one app that responds to your implicit intent, the user can select which app to use and make that app the default choice for the action. The ability to select a default is helpful when performing an action for which the user probably wants to use the same app every time, such as when opening a web page (users often prefer just one web browser).

However, if multiple apps can respond to the intent and the user might want to use a different app each time, you should explicitly show a chooser dialog. The chooser dialog asks the user to select which app to use for the action (the user cannot select a default app for the action). For example, when your app performs "share" with the `ACTION_SEND` action, users may want to share using a different app depending on their current situation, so you should always use the chooser dialog, as shown in Figure 2.

To show the chooser, create an `Intent` using `createChooser()` and pass it to `startActivity()`, as shown in the following example. This example displays a dialog with a list of apps that respond to the intent passed to the `createChooser()` method and uses the supplied text as the dialog title.

Figure 2. A chooser dialog.

Java Collections

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack. ArrayList

The **ArrayList** class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized.

- *add()* – takes *O(1)* time
- *get()* – is always a constant time *O(1)* operation
- *remove()* – runs in linear *O(n)* time. We have to iterate the entire array to find the element qualifying for removal.
- **indexOf()** – also runs in linear time. It iterates through the internal array and checks each element one by one, so the time complexity for this operation always requires *O(n)* time.

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

- *add()* – appends an element to the end of the list. It only updates a tail, and therefore, it's *O(1)* constant-time complexity.
- **add(index, element)** – on average runs in *O(n)* time
- *get()* – searching for an element takes *O(n)* time.
- *remove(element)* – to remove an element, we first need to find it. This operation is *O(n).*
- **remove(index)** – to remove an element by index, we first need to follow the links from the beginning; therefore, the overall complexity is *O(n).*
- *contains()* – also has *O(n)* time complexity

***Vector** uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.*

Stack The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Set Interface

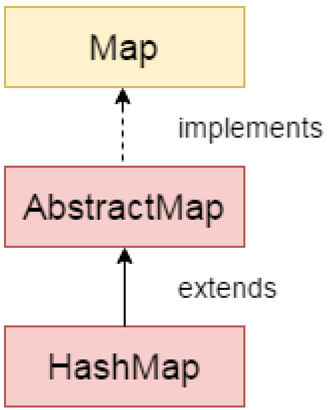
Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.



Java HashMap

Java HashMap class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

In the worst case, a `HashMap` has an O(n) lookup due to walking through all entries in the same hash bucket (e.g. if they all have the same hash code). Fortunately, that worst case scenario doesn't come up very often in real life, in my experience. So no, O(1) certainly isn't guaranteed - but it's usually what you should assume when considering which algorithms and data structures to use.

In JDK 8, `HashMap` has been tweaked so that if keys can be compared for ordering, then any densely-populated bucket is implemented as a tree, so that even if there are lots of entries with the same hash code, the complexity is O(log n). That can cause issues if you have a key type where equality and ordering are different, of course.

hashset vs. treeset vs. linkedhashset

hashset is implemented using a hash table. elements are not ordered. the add, remove, and contains methods has constant time complexity o(1). treeset is implemented using a tree structure(red-black tree in algorithm book). the elements in a set are sorted, but the add, remove, and contains methods has time complexity of o(log (n)). it offers several methods to deal with the ordered set like first(), last(), headset(), tailset(), etc. linkedhashset is between hashset and treeset. it is implemented as a hash table with a linked list running through it, so it provides the order of insertion. the time complexity of basic methods is o(1).

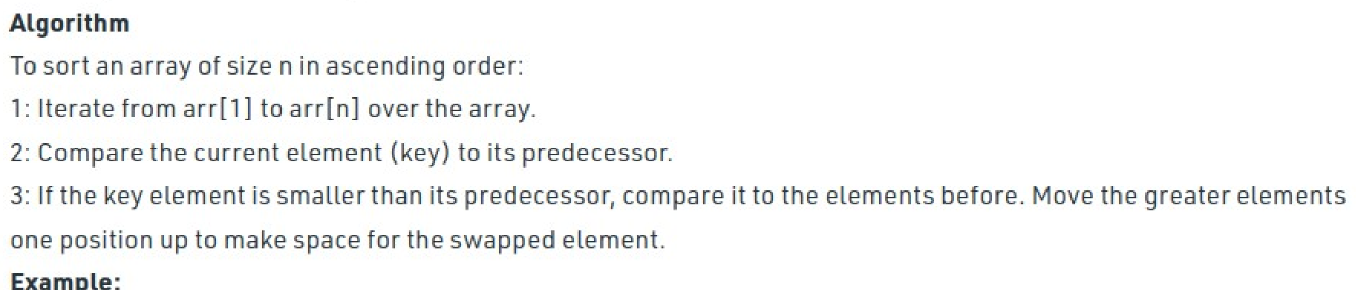
Binary search O(logn)

Binary search works on sorted arrays. Binary search begins by comparing an element in the middle of the array with the target value. If the target value matches the element, its position in the array is returned. If the target value is less than the element, the search continues in the lower half of the array. If the target value is greater than the element, the search continues in the upper half of the array. By doing this, the algorithm eliminates the half in which the target value cannot lie in each iteration.

Bubble sort n^2

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Bubble sort n^2 / n — (better performance)



Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:

version control system (VCS)

A VCS keeps track of the contributions of the developers working as a team on the projects. They maintain the history of code changes done and with project evolution, it gives an upper hand to the developers to introduce new code, fixes bugs, and run tests with confidence that their previously working copy could be restored at any moment in case things go wrong

git repository is a repository is a file structure where git stores all the project-based files. Git can either stores the files on the local or the remote repository

Show istory of commit	Git log
Откатывает изменение до последнего коммита	Git restore
Показывааeт изменение в local file	Git diff
Added all to stageing area and comit it	Git commit **-am** «message»
Delete file in commit but save in local machine	Git restore --staged Nameofapp
Create new branch	Git banch Name
Checkout me breanches	Git checkout Name
Show all branch in account	Git branch -a

Delete breanch	Git branch -d Name