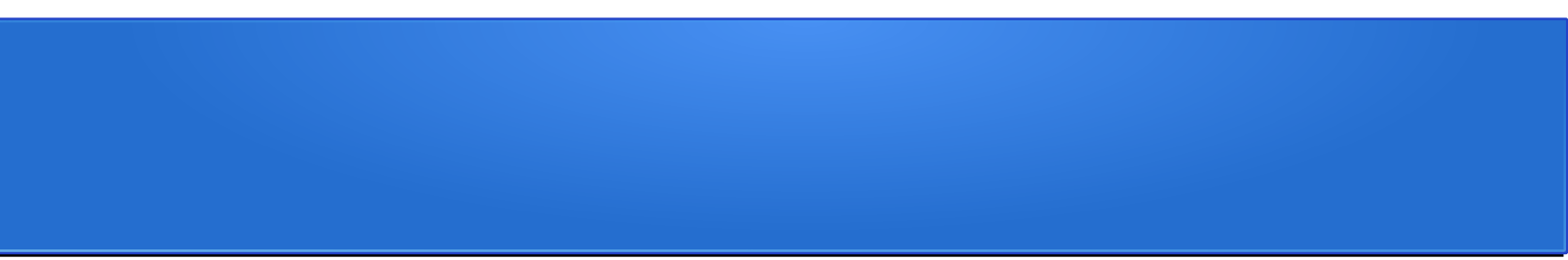


# Fragments

A Fragment represents a behavior or a portion of user interface in a `FragmentActivity`. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

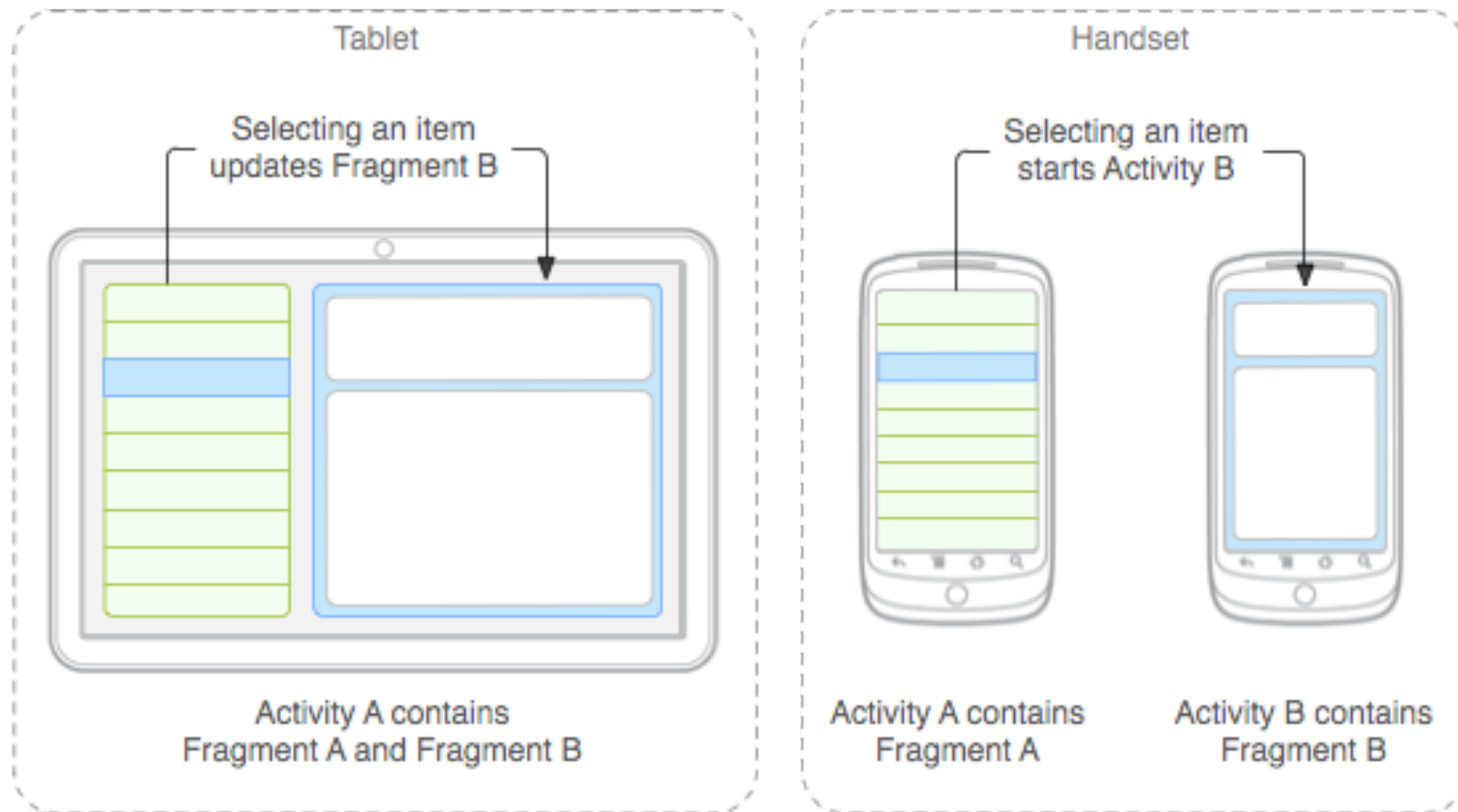


A fragment must always be hosted in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments. However, while an activity is running (it is in the resumed lifecycle), you can manipulate each fragment independently, such as add or remove them. When you perform such a fragment transaction, you can also add it to a back stack that's managed by the activity—each back stack entry in the activity is a record of the fragment transaction that occurred. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the Back button.

# Design Philosophy

Android introduced fragments in Android 3.0 (API level 11), primarily to support more dynamic and flexible UI designs on large screens, such as tablets. Because a tablet's screen is much larger than that of a handset, there's more room to combine and interchange UI components. Fragments allow such designs without the need for you to manage complex changes to the view hierarchy. By dividing the layout of an activity into fragments, you become able to modify the activity's appearance at runtime and preserve those changes in a back stack that's managed by the activity.

# Diagram



## Example

For example, a news application can use one fragment to show a list of articles on the left and another fragment to display an article on the right—both fragments appear in one activity, side by side, and each fragment has its own set of lifecycle callback methods and handle their own user input events. Thus, instead of using one activity to select an article and another activity to read the article, the user can select an article and read it all within the same activity

## Design as Modular & Reusable

You should design each fragment as a modular and reusable activity component. That is, because each fragment defines its own layout and its own behavior with its own lifecycle callbacks, you can include one fragment in multiple activities, so you should design for reuse and avoid directly manipulating one fragment from another fragment. This is especially important because a modular fragment allows you to change your fragment combinations for different screen sizes.

## Example

For example—to continue with the news application example—the application can embed two fragments in Activity A, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so Activity A includes only the fragment for the list of articles, and when the user selects an article, it starts Activity B, which includes the second fragment to read the article. Thus, the application supports both tablets and handsets by reusing fragments in different combinations

# Creating a Fragment

To create a fragment, you must create a subclass of `Fragment` (or an existing subclass of it). The `Fragment` class has code that looks a lot like an `Activity`. It contains callback methods similar to an activity, such as `onCreate()`, `onStart()`, `onPause()`, and `onStop()`. In fact, if you're converting an existing Android application to use fragments, you might simply move code from your activity's callback methods into the respective callback methods of your fragment.

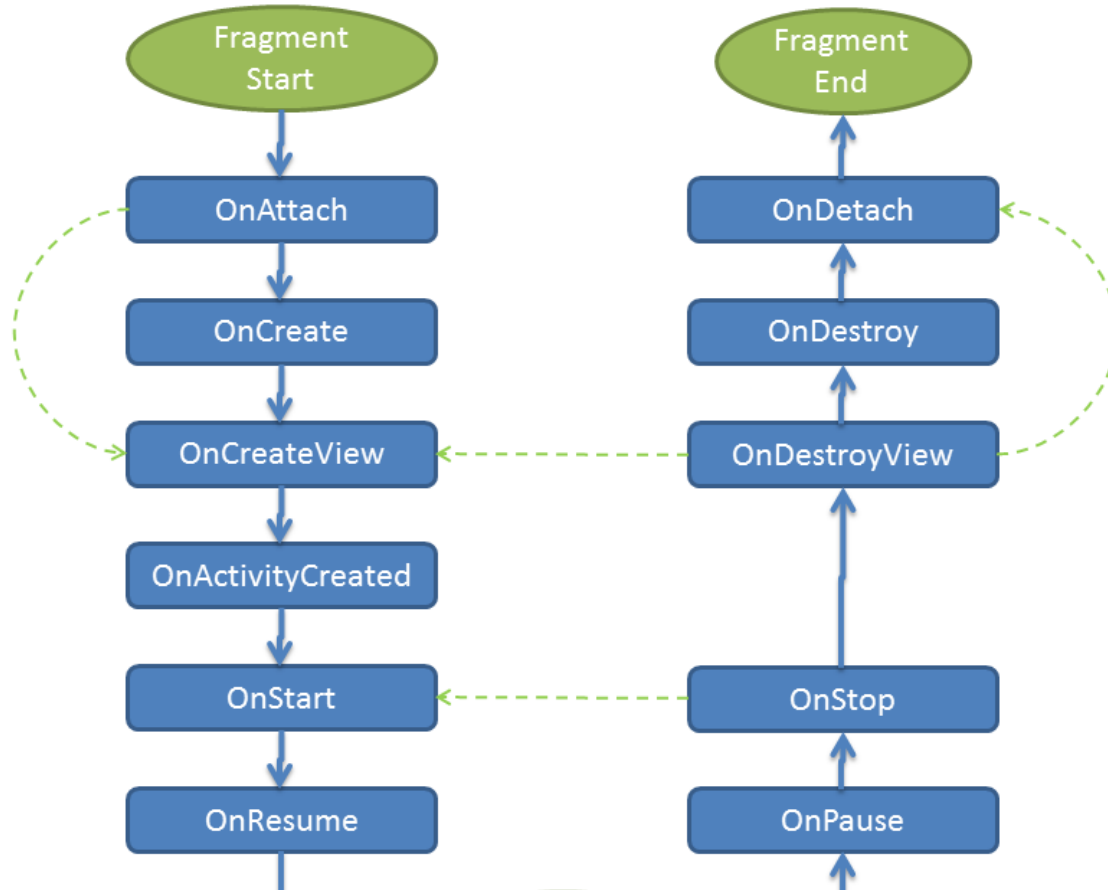


# Creating a Fragment Demo

Creating the android project and add fragments to activity with  
Static and dynamic methods using android studio

:

# Fragment life cycle Diagram



# Fragment life cycle methods

Most applications should implement at least these three methods for every fragment, but there are several other callback methods you should also use to handle various stages of the fragment lifecycle.

- `Oncreate()`
- `OnCreateView()`
- `OnPause()`

# Importance of Fragments

There are many use cases for fragments but the most common use cases include:

- Reusing View and Logic Components - Fragments enable re-use of parts of your screen including views and event logic over and over in different ways across many disparate activities. For example, using the same list across different data sources within an app.
- Tablet Support - Often within apps, the tablet version of an activity has a substantially different layout from the phone version which is different from the TV version. Fragments enable device-specific activities to reuse shared elements while also having differences.
- Screen Orientation - Often within apps, the portrait version of an activity has a substantially different layout from the landscape version. Fragments enable both orientations to reuse shared elements while also having differences.

# Sub classes of Fragment

There are also a few subclasses that you might want to extend, instead of the base Fragment class.

- DialogFragment
- ListFragment
- PreferenceFragmentCompat

# Looking Up a Fragment Instance

Often we need to lookup or find a fragment instance within an activity layout file. There are a few methods for looking up an existing fragment instance:

- ID - Lookup a fragment by calling `findFragmentById` on the `FragmentManager`
- Tag - Lookup a fragment by calling `findFragmentByTag` on the `FragmentManager`
- Pager - Lookup a fragment by calling `getRegisteredFragment` on a `PagerAdapter` (not part of the Android APIs but there is a custom implementation here <https://stackoverflow.com/a/20504487>)

# Looking Up a Fragment Instance

- **Finding Fragment By ID** If the fragment was statically embedded in the XML within an activity and given an android:id such as fragmentDemo then we can lookup this fragment by id by calling `findFragmentById` on the `FragmentManager`:

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        if (savedInstanceState == null) {  
            DemoFragment fragmentDemo = (DemoFragment)  
                getSupportFragmentManager().findFragmentById(R.id.fragmentDemo);  
        }  
    }  
}
```

# Looking Up a Fragment Instance

- **Finding Fragment By Tag** If the fragment was dynamically added at runtime within an activity then we can lookup this fragment by tag by calling `findFragmentByTag` on the `FragmentManager`:

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        if (savedInstanceState == null) {  
            // Let's first dynamically add a fragment into a frame container  
            getSupportFragmentManager().beginTransaction().  
                replace(R.id.flContainer, new DemoFragment(), "SOMETAG").  
                commit();  
            // Now let's lookup the fragment that
```



# Looking Up a Fragment Instance

**Finding Fragment Within Pager:** If the fragment was dynamically added at runtime within an activity into a ViewPager using a FragmentPagerAdapter then we can lookup the fragment by upgrading to a SmartFragmentStatePagerAdapter as described in the ViewPager guide. Now with the adapter in place, we can also easily access any fragments within the ViewPager using `getRegisteredFragment`:

```
adapterViewPager.getRegisteredFragment(0);
```

# Communicating with Fragments

Fragments should generally only communicate with their direct parent activity. Fragments communicate through their parent activity allowing the activity to manage the inputs and outputs of data from that fragment coordinating with other fragments or activities.

The important thing to keep in mind is that fragments should not directly communicate with each other and should generally only communicate with their parent activity. Fragments should be modular, standalone and reusable components. The fragments allow their parent activity to respond to intents and callbacks in most cases.

There are three ways a fragment and an activity can communicate:

1. Bundle - Activity can construct a fragment and set arguments
2. Methods - Activity can call methods on a fragment instance
3. Listener - Fragment can fire listener events on an activity via an interface

# Fragment with Arguments

In certain cases, your fragment may want to accept certain arguments. A common pattern is to create a static newInstance method for creating a Fragment with arguments. This is because a Fragment must have only a constructor with no arguments

```
public static SecondFragment newInstance(int someInt, String someTitle) {  
    SecondFragment fragmentDemo = new SecondFragment();  
    Bundle args = new Bundle();  
    args.putInt("someInt", someInt);  
    args.putString("someTitle", someTitle);  
    fragmentDemo.setArguments(args);  
    return fragmentDemo;  
}
```

# Fragment Methods

If an activity needs to make a fragment perform an action after initialization, the easiest way is by having the activity invoke a method on the fragment instance. In the fragment, add a method:

```
FirstFragment fragmentDemo = (FirstFragment)
    getSupportFragmentManager().findFragmentById(R.id.fragment);
fragmentDemo.doSomething("some param");
```

# Fragment Listener

If a fragment needs to communicate events to the activity, the fragment should define an interface as an inner type and require that the activity must implement this interface:

```
// Define the events that the fragment will use to communicate
public interface OnItemSelectedListener {
    // This can be any number of events to be sent to the activity
    public void onRssItemSelected(String link);
}

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnItemSelectedListener) {
        listener = (OnItemSelectedListener) context;
    } else {
        throw new ClassCastException(context.toString()
            + " must implement MyListFragment.OnItemSelectedListener");
    }
}

// Now we can fire the event when the user selects something in the fragment
public void onSomeClick(View v) {
    listener.onRssItemSelected("some link");
}
```

# Understanding the FragmentManager

The FragmentManager is responsible for all runtime management of fragments including adding, removing, hiding, showing, or otherwise navigating between fragments. As shown above, the fragment manager is also responsible for finding fragments within an activity. Important available methods are outlined below:

# Understanding the FragmentManager

## Method

- **AddOnBackStackChangeListener:-** Add a new listener for changes to the fragment back stack.
- **beginTransaction():-** Creates a new transaction to change fragments at runtime.
- **findFragmentById(int id):-** Finds a fragment by id usually inflated from activity XML layout.
- **findFragmentByTag(String tag):-** Finds a fragment by tag usually for a runtime added fragment.
- **PopBackStack():-** Remove a fragment from the backstack.
- **executePendingTransactions()** Forces committed transactions to be applied.

# Managing Fragment Backstack

A record of all Fragment transactions is kept for each Activity by the `FragmentManager`. When used properly, this allows the user to hit the device's back button to remove previously added Fragments (not unlike how the back button removes an Activity). Simply call `addToBackStack` on each `FragmentTransaction` that should be recorded:

```
FragmentTransaction fts = getSupportFragmentManager().beginTransaction();  
// Replace the content of the container  
    fts.replace(R.id.flContainer, new FirstFragment());  
// Append this transaction to the backstack  
    fts.addToBackStack("optional tag");  
// Commit the changes  
    fts.commit();
```



# Managing Fragment Backstack

Programmatically, you can also pop from the back stack at any time through the manager:

```
FragmentManager fragmentManager = getSupportFragmentManager();  
if (fragmentManager.getBackStackEntryCount() > 0) {  
    fragmentManager.popBackStack();  
}
```

# Fragment Hiding vs Replace

In many of the examples above, we call `transaction.replace(...)` to load a dynamic fragment which first removes the existing fragment from the activity invoking `onStop` and `onDestroy` for that fragment before adding the new fragment to the container. This can be good because this will release memory and make the UI snappier. However, in many cases, we may want to keep both fragments around in the container and simply toggle their visibility. This allows all fragments to maintain their state because they are never removed from the container. To do this, we might modify this code:

# Fragment Hiding vs Replace

```
private FragmentA fragmentA;  
private FragmentB fragmentB;  
private FragmentC fragmentC;
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    if (savedInstanceState == null) {  
        fragmentA = FragmentA.newInstance("foo");  
        fragmentB = FragmentB.newInstance("bar");  
        fragmentC = FragmentC.newInstance("baz");  
    }  
}
```

```
protected void displayFragmentA() {  
    FragmentTransaction ft = getSupportFragmentManager().beginTransaction();  
    // removes the existing fragment calling onDestroy  
    ft.replace(R.id.flContainer, fragmentA);  
}
```

# Fragment Hiding vs Replace

// ...onCreate stays the same

// Replace the switch method

```
protected void displayFragmentA() {  
    FragmentTransaction ft = getSupportFragmentManager().beginTransaction();  
    if (fragmentA.isAdded()) { // if the fragment is already in container  
        ft.show(fragmentA);  
    } else { // fragment needs to be added to frame container  
        ft.add(R.id.flContainer, fragmentA, "A");  
    }  
    // Hide fragment B  
    if (fragmentB.isAdded()) { ft.hide(fragmentB); }  
    // Hide fragment C  
    if (fragmentC.isAdded()) { ft.hide(fragmentC); }  
    // Commit changes  
    ft.commit();  
}
```