Part 4:

**Q.1**

**What is the major difference between an ​ abstract class and an ​ interface?**

|  |  |
| --- | --- |
| accepted | Technically, the differences aren't really significant but, conceptually, they are entirely different things and that leads to the technical differences others have mentioned.  An abstract superclass is exactly what it sounds like, it's a common type that is shared by many other types, like Cats and Dogs are Animals.  An interface is also exactly what it sounds like, it's an interface through which other classes can communicate with the object. If you want to make a Cat Walk, you're ok, cause Cat implements a CanWalk interface. Same for a Lizard, though they walk very differently. A Snake, on the other hand, does not implement CanWalk, so you can't tell it to Walk. Meanwhile, Lizard and Snake (or possibly more explicit subclasses -- I'm not an expert) might both shed their skin, and thus implement CanShed, while a Cat couldn't do that.  But they're all still Animals and have some common properties, like whether they're alive or dead.  This is why all methods on an interface must be implemented as public (or explicitly, in C#). Cause what's the point in an interface that's hidden from the class interfacing with the object? It's also why you can have multiple interfaces to an object, even when a language doesn't support multiple inheritance. |

**Abstract Class:**

1. **Abstract class can have abstract and non-abstract methods.**
2. **Abstract class can have final, non-final, static and non-static variables.**
3. **Abstract class doesn't support multiple inheritance.**
4. **Abstract class can have static methods, main method and constructor.**
5. **The abstract keyword is used to declare abstract class.**
6. **Abstract class can provide the implementation of interface**

**Interface:**

1. **Interface can have only abstract methods.**
2. **Interface supports multiple inheritance.**
3. **Interface has only static and final variables.**
4. **Interface can't have static methods, main method or constructor.**
5. **Interface can't provide the implementation of abstract class.**
6. **The interface keyword is used to declare interface.**

Difference by Example Wise

Interface and Abstract Class Example:

public interface Engine

{

   void start();

   void stop();

   double useFuel(int distance, int speed);

   void breakDown() throws Exception;

}//end interface

public abstract class BasicEngine implements Engine

{

   public void breakDown() throws BreakdownException

   {

       throw new BreakdownException("Oil pressure low.");

   }

   protected class BreakdownException extends Exception

   {

       public BreakdownException()

       {

       }

       public BreakdownException(String message)

       {

             super(message);

       }

       public BreakdownException(String message, Throwable cause)

       {

             super(message, cause);

       }

       public BreakdownException(Throwable cause)

       {

             super(cause);

       }

   }

}//end class

**Q.2**

**Why is Java 7’s class inheritance flawed?**

Java doesn't allow multiple inheritance, but it allows implementing multiple interfaces.

Because interfaces specify only what the class is doing, not how it is doing it.

The problem with multiple inheritance is that two classes may define different ways of doing the same thing, and the subclass can't choose which one to pick.

C++ , Common lisp and few other languages supports multiple inheritance while java doesn’t support it. It is just to **remove ambiguity**, because **multiple inheritance** can cause ambiguity in few scenarios.

**achieve multiple inheritance in Java using interfaces?**

interface X

{

public void myMethod();

}

interface Y

{

public void myMethod();

}

class Demo implements X, Y

{

public void myMethod(){

System.out.println(" Multiple inheritance example using interfaces");

}}

**Q.3**

**What are the major differences between ​ Activities and ​ Fragments?**

Fragment is a part of an activity, which contributes its own UI to that activity. Fragment can be thought like a sub activity. Whereas the complete screen with which user interacts is called as activity. An activity can contain multiple fragments. Fragments are mostly a sub part of an activity.

An activity may contain 0 or multiple number of fragments based on the screen size. A fragment can be reused in multiple activities, so it acts like a reusable component in activities.

A fragment can't exist independently. It should be always part of an activity. Whereas activity can exist without any fragment in it.

**One More Clarification between Fragment and Activity**

A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity which enable a more modular activity design. It will not be wrong if we say a fragment is a kind of sub activity.

Following are important points about a fragment:

1. A fragment has its own layout and its own behavior with its own lifecycle callbacks.
2. You can add or remove fragments in an activity while the activity is running.
3. You can combine multiple fragments in a single activity to build a multi-pane UI.
4. A fragment can be used in multiple activities.
5. The fragment life cycle is closely related to the lifecycle of its host activity.
6. When the activity is paused, all the fragments available in the activity will also be stopped.
7. A fragment can implement a behavior that has no user interface component.
8. Fragments were added to the Android API in [Android 3](http://en.wikipedia.org/wiki/Android_version_history#Android_3.0_Honeycomb_.28API_level_11.29) (Honeycomb) with API version 11.

For more details, please visit the official site

**Q 4**

**When using ​ Fragments how you do communicate back to their hosting ​ Activity?**

Use an interface as a call back to the activity. Once you get the message in the activity there is no need to click the button just replace the existing fragment in the container.

Implement the interface in the activity

FragmentB newFragment = new FragmentB();

Bundle args = new Bundle();

args.putInt("key", "message");

newFragment.setArguments(args);

FragmentTransaction transaction = getFragmentManager().beginTransaction();

transaction.replace(R.id.fragment\_container, newFragment);

// replace with fragmentb. no need to perform click again.

// based on the message you decide which fragment you want to replace with

transaction.commit();

**Q.5**

**Can you make an entire app without ever using ​ Fragments ​ ? Why or why not? Are there any special cases when you absolutely have to use or should use Fragments?**

Yes we can built Apps without fragments**.**

**Because** Activity is a part of the four core components of Android, along with Service, Broadcast Receiver and Content provider. But fragment is not a core component, it's there to be used as a part or 'fragment' of activity. Your app certainly can be built using only activities, just like many other apps.

Fragments are more of a UI benefit in my opinion. It's convenient for the user sometimes to see two different views of two different classes on the same screen. If, in your moment of creativity, you decide it would be nice to display your application with, say, a listView that takes up half the screen and a webView that takes up the other half - so that when you click on a list item in fragment A it passes an intent to the webView in fragment B, and suddenly you see what you just clicked without the app switching activities - then you could use a fragment. That's just an example I came up with off the top of my head.

Fragments are two or more activities on the screen at the same time.

**Q.6**

**What makes an ​ AsyncTask ​ such an annoyance to Android developers? Detail some of the issues with ​ AsyncTask ​ , and how to potentially solve them.**

**You're doing it wrong**  
A common way for new developers to use AsyncTask could be something like the following:

|  |
| --- |
| public class MainActivity extends Activity {      @Override protected void onCreate(Bundle savedInstanceState) {      super.onCreate(savedInstanceState);      setContentView(R.layout.activity\_main);    }      // Somewhere the AsyncTask is started      public class MyAsyncTask extends AsyncTask<Void, Void, String> {        @Override protected String doInBackground(Void... params) {        // Do work        return result;      }        @Override protected void onPostExecute(String result) {        Log.d("MyAsyncTask", "Received result: " + result);      }    }  } |

The problem with this is that the AsyncTask has an implicit reference to the enclosing Activity. If a configuration change happens the Activity instance that started the AsyncTask would be destroyed, but not GCd until the AsyncTask finishes. Since Activities are heavy this could lead to memory issues if several AsyncTasks are started. Another issue is that the result of the AsyncTask could be lost, if it's intended to act on the state of the Activity.

This leads to two issues we have to fix:  
- Ensuring the Activity isn't kept in memory when destroyed by the framework  
- Ensuring the result of the AsyncTask is delivered to the current Activity instance

# Dealing with AsyncTask and Screen Orientation

A common task in Android is to perform some background activity in another thread, meanwhile displaying a ProgressDialog to the user. Example of these tasks include downloading some data from internet, logging into an application, etc. Implementing an AsyncTask is fairly a simple job, the big challenge is how to handle it properly when an orientation change occurs.

Solution:

Do **NOT** use android:configChanges to address this issue. This is very bad practice.

Do **NOT** use Activity#onRetainNonConfigurationInstance() either. This is less modular and not well-suited for Fragment-based applications.

You can [**read my article**](http://www.androiddesignpatterns.com/2013/04/retaining-objects-across-config-changes.html) describing how to handle configuration changes using retained Fragments. It solves the problem of retaining an AsyncTask across a rotation change nicely. You basically need to host your AsyncTask inside a Fragment, call setRetainInstance(true) on the Fragment, and report the AsyncTask's progress/results back to it's Activity through the retained Fragment

**​**

**​**