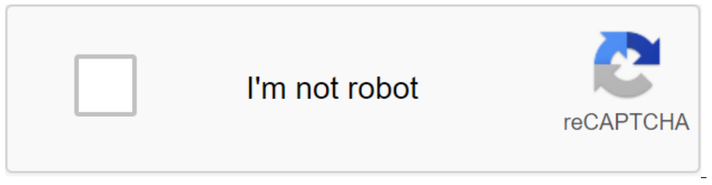
Android studio create tabbed activity



Continue

Creating an Android app isn't as hard as it may seem, as long as you're focused on creating a simple app at first. Android Authority shares a tutorial that covers 10 basic things you have to do to develop your first app. Although it's presented as a 10-day program that will take you from idea to publishing Play Store, if you don't have any background coding, you'll probably need more time than a couple of hours for each of the ten days. Learning the basics of Java alone can take days or weeks. However, it's a good plan with related resources to get you started. Ten Steps: Idea (inventing the concept of the app)Drawing wireframeChoosing your IDE and tweaking it By learning the basics of JavaCreating or purchasing images Building layoutWriting codeImbre advanced functionalityAdding some additional polishPublishing your applicationNeoy between steps 7 and 10, you can also add Debugging because you will work in bugs. In addition, steps from 7 to 9 will include a lot of googling and visits to the Stack Exchange.Check from the guide below to get started. A 10-day program to create your first Android Authority Hot app on the heels of Android 11 Developer Preview, Android Studio 3.6 is now available on a stable channel, meaning developers can start making confident use of it for their projects. This brings a number of useful features and updates, including the new Split View's Design Editor for faster design and preview of XML layouts. Another exciting new feature is the support of several displays in the Android emulator. Automatic detection of memory leakage meanwhile promises to make debugging much easier. You can check out the full array of features from the Android Developers blog, or get the highlights below. Split View and EditingPerhaps The most interesting new feature in Android Studio 3.6 is Split View for Design Editors. This allows you to see the XML code side by side along with the preview render. It's a small thing, but in fact it makes life a lot easier to view the effect that changes the code right away (and vice versa). The view you choose will also be saved on a case-by-case basis, which means you can easily download the preferred setup depending on the file you're editing. While we discuss design, we should also note the new color collector, making it much easier to select and fill color values without a set of values. It's available through the XML editor and design tools. Faster development When it comes to development, a few new changes should make life easier for Android Developers in Android Studio 3.6.View is a particularly welcome inclusion that will offer a compilation of security time when linking to opinions. With this option, you'll create a binding class for each XML layout file in the module. This will actually replace the need for findViewByID: findViewByID: be able to easily refer to any view with ID without risking zero pointer exceptions or exceptions to the class. This can prove very useful and reduce a lot of patterns. Other new updates include the release of the IntelliJ 2019.2 platform with better launch time and new tool services, as well as support for Kotlin for more Android NDK features. Updates to the Android Gradle plug-in include support for the Maven Publish Gradle plug-in. This allows you to create artifacts in the Apache Maven repository. Testing and debugging Android Emulator 29.2.12 makes it easier for developers to interact with the location of the emulation device. Google Maps is now built into the advanced control menu, making it easy to specify locations and create routes. Perhaps more appropriate is still the support of several virtual displays that will be useful for those designing for devices such as Samsung Galaxy Fold.Read also: Development for folding devices: What you need to know, memory detection Profiler will detect activity and instances of a snippet that may have leaked. Build time has also improved for debugging builds thanks to the use of zipflinger. A better quality of life changesIt's only a small selection of updates available in Android Studio 3.6. You'll find plenty of other small updates as you use the new software too: including the resumed SDK downloads, which is perfect for those who don't always have an hour spare to download the latest Android image! Grab Android Studio 3.6 here. Of course, on the Canary Channel you can already get your hands on Android Studio 4.1. What do you think of these new features? What would you like to see come to Android Studio in the future? Welcome to the two parts of this custom launcher tutorial! If you haven't read part of one of this series, read it and come back. Even more so than Part 1, it's a somewhat advanced project. If you're not familiar with classes, Android SDK, and Java, I recommend you also do some more background reading first. Still with me? Ok. If you followed along with Part 1, you must now have a launcher that is loaded when you download your phone. It should also have an app work drawer. At this point, that app drawer is a bit slow, and there is only one page displaying one app though. In other words, we have a job! First, it's time to put the icons in a box in a separate thread. This avoids the main user interface flow, which means that the list will be downloaded in the background, ready to use. To do this, we'll use what's called ASyncTask.Speeding up the drawerHere's plan applications. Make the list of apps public and create a method in our class to add new items to this list: Public RAdapter (Context c) - appsList - new ArrayList We don't need to create our list in the designer anymore, so we'll just announce Add the next subclass to AppsDrawer.java to do the same with AsyncTask. This will perform the same action in a separate thread, so the app can still deal with user interaction while working through it. The code should look familiar: the myThread community class expands AsyncTask'lt;Void, void, string and @Override doInBackground (Void... Params) - PackageManager pm - getPackageManager appsList - the new ArrayList Intention i - a new intention (Intent.ACTION_MAIN, zero); i.addCategory (Intent.CATEGORY_LAUNCHER); List allApps and pm.queryIntentActivities (i, 0); For (ResolveInfo ri:allApps) - AppInfo app - new AppInfo (); app.label - ri.loadLabel (pm); app.packageImia and ri.activityInfo.packageImia; app.icon - ri.activityInfo.loadIcon (pm); radapter.addApp - Return Success; - @Override protected void on PostExecute (result of the line) - super.onPostExecute (result); UpdateStuff(); Of course, you also need to remove a duplicate of the code from the adapter class. We can then simply run our ASync class in onCreate () AppsDawer.java file: new myThread ().;) Try to operate the launcher and the application drawer should now spring to life pretty seamlessly. Eagle eyes among you also noticed that I have created another new method: public invalid updateStuff () - radapter.notifyItemInserted (radapter.getItemCount ()-1); Note radaptor.notifiyItemInserted. This allows you to dynamically add items to our list of processors. This will be useful in the future for you serious launcher designers because it can listen to newly installed or remote applications and update the view accordingly. It all looks a lot better, but something is wrong yet. At the moment, we call onCreate and make a new app drawer every time an activity is created. To prevent this from happening, we want to add a line to our manifest in the lt'applicationgt;tag for AppsDrawer:android:launchModesingleTaskTo be safe, we can also override the onBackPressed () method in our AppsDrawer.java file. Using the fragmentsThe app box got faster, but it would be even better if it was created when the app launches, rather than when the user first presses the app box button. So he will be ready before he has been pressed. We could bend over to do this, but the best solution is to put our app drawer in a piece - a shelf that for a moment, we'll go back to it. The fragments are incredibly powerful to create dynamic UIs, and they are perfect for our launcher! Fragments also provide the best way to create a solution. A series of home screens to hold when selecting our apps! We'll create snippets and then swiping through them with ViewPager. It has its own life cycle and can contain many views, but more than one fragment can be seen on the screen at once (as opposed to activity). Fragments can also behave like objects, in that multiple instances of the same fragment can exist simultaneously. This again lends itself well to the homepage, because users can add and delete home pages as needed for the home of many different apps and widgets. The fragments are incredibly powerful to create dynamic UIs, and they are perfect for our launcher! To create a snippet, go to the New Fragment File. Then you will have the opportunity to create a new fragment, which we will call Homescreen. Untick factory techniques and callback boxes and click finish. This should create a new XML file, fragment_homescreen.xml, and a new Java file, Homescreen.java, just like the activity. At this point, add another kind of image and place it in the center of the screen using gravity. Give it an icon ID and give a snippet of the house's own ID. To make this work inside our snippet, we unfortunately can't just drag and drop the onClick code from before. Instead, examine the code below to see how it all should work: Public Classroom Homescreen Expands Fragment Implements View.OnClickListener Public Home Screen () / / Mandatory Empty Public Designer - @Override Public View atCreateView (LayoutInflater Inflatable, ViewGroup Container, Bundle SavedInstanceState) - View v. inflater.inflate (R.layout.fragment_homescreen, container, false); ImageView v.findViewById (R.id.icon); Icon.setImageDrawable (MainActivity.getActivityIcon(this.getContext(),), com.android.chrome, com.google.android.apps.chrome.Main)); Icon.setOnClickListener (this); Return v @Override public void onClick (View v) - Switch (v.getId()) - case R.id.icon: Intention to launchIntent - MainActivity.baseContext.getPackageManager (.getLaunchIntentForPackage); StartActivity (launchIntent); A break; It's a little more uncomfortable, but you should be able to draw an engineer to work it out as you require it. Just override the various onClicks.Notice that I was able to use getActivityIcon from MainActivity because I made the method static. Static methods from other classes can be worked without creating multiple instances of this class. You see, there is a method of my madness (and my methods too)! Add a snippet to And arrange it nicely over the app box button. Now you can see the Chrome icon button the same way you did before. It's a lot of code to achieve exactly the same result, but it's programming for you! Of course, the real reason we went to all this effort is because it will allow us to do more interesting things in the future. Now we can create multiple fragments using exactly the same Java code and exactly the same XML. That we could run two instances of the same screen and change the icons that are displayed based on the ID we give to everyone in XML! Getting better, too. ViewPagerUsing snippets also mean that we can use ViewPager to scroll through our home screens as would be normal in any launcher application. ViewPager also gives us the opportunity to spice up screens as we transition between them. Using snippets also means we can use ViewPager to scroll through our home screens as you would expect to be able to in any launcher application. The official documentation for ViewPager can be found here. It's not too hard, thankfully. First, we have to drag and drop our ViewPager into activity_main.xml, just like in any other species. Just stick to it where the snippet is now. Now we need to create a different class. This will be called HomescreenAdapter and will be distributed by FragmentStatePageAdapter. This adapter will put our fragments inside the ViewPager.It looks like this: the private HomescreenAdapter class expands FragmentStatePagerAdapter - public HomescreenAdapter (FragmentManager fm) - super (fm); - @Override a snippet of getItem (int position) - the return of a new home screen (); - @Override int getCount - return NUM_PAGES; We need a global variable as a static final int NUM_PAGES determine how many pages you want. You may not want this to be final in the future, though, as most apps allow your users to add extra home pages. Set up an adapter in onCreate () your MainActivity:mPager (ViewPager) findViewById (R.id.homescreenPager); mPagerAdapter - new HomescreenAdapter (getSupportFragmentManager); mPager.setAdapter (mPagerAdapter); Download this, and now you have to have a swipe-capable part of the screen, with each of which shows our Chrome icon. The app box button should also stay where it is at the bottom of the screen. In the future, you may need to adapt this to show different icons on each page. You would do that by passing int positions from getItem () as a bundle and using a switch statement to load different icons or layouts. With that, you now have a number of screens through which you can As well as a beautifully fast box app! It's starting to look and feel a lot like a real launcher. At the bottom of that documentation, you can even add a number of trendy animations just like the best launchers out there! Displaying Widgets Doesnaries doesn't just show icons though: they show widgets too. The first thing you need to do to make it work is to add this permission to your manifest: zlt;uses-permission android:name,android.permission.BIND_APPWIDGET/uses-permission. These days, you also need to provide permission for the app while performing through dialogue. You're going to use the AppWidgetHost class to control and display widgets that will have your own ID. This ID is important and should remain constant so that widgets know they are communicating with your app. Each widget will also have its own ID when it is tied to the host, which will happen every time the app launches. AppWidgetHostView will be a container with a host display and widget. You will use the options package to transmit information to and from widgets, how the size at which they need to be displayed and what information from the app they will show, among other things. It's an incredibly involved process, especially after you start doing things like saving the widgets the user wants to use and the settings they've chosen. You'll need to use multiple xML files and classes just to make the basics work. It is too active to go step by step in this post. You can find more information on how to place widgets here, but this is somewhat brief. You can also find the working code for the full launcher here. The code used in the tutorial comes from this, so if you read through this and pick up the snippets from the project, you can change the engineer of it to the point where it will work. Reverse engineering and hunting for clues is very often the reality of programming on Android, especially when you're trying to do something rare and not required for the vast majority of applications. I recommend you start by testing this in a separate activity within your project (or even a separate project entirely) and move it to homepage snippets only when you have everything working beautifully. Reverse engineering and hunting for clues is very often the reality of programming on Android, especially when you're trying to do something rare, or unnecessary for most apps. You will also need to check the section at the bottom of the documentation in order to update this process for Android 4.0 and above. There is a lot more to do! As I said, building a launcher is a big challenge. If you manage to work your way through Pain adding widgets, there are many other things worth adding: Icon packsHandling screen rotation (if you choose to do so!) allowing users to drag and drop their icons around screenCustomizationsFoldersPlus all that will make your app unique! It's not a little little But it can be a particularly fun and useful job to take on and the results will be something that you (and any users) will use every day. Good luck, share your thoughts on the process in the comments below, and let me know if you want to see the addition of widgets (or anything else, for that matter) handled in a separate post! Post!