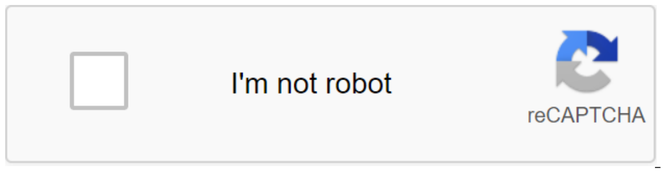
Android studio java learning



Continue

There are many ways to create an Android game and one important way to do it from scratch is in Android Studio with Java. This gives you maximum control over how you want your game to look and behave, and this process will teach you skills that you can use in a number of other scenarios too - whether you create a splash screen for the app or you just want to add some animation. With that in mind, this tutorial is going to show you how to create a simple 2D game using Android Studio and Java. You can find all the code and resources on Github if you want to follow along. Setting up In order to create our game, we will deal with several specific concepts: game cycles, themes and canvases. To begin with, run Android Studio. If you don't have it installed, then check out our full introduction to Android Studio, which goes over the installation process. Now start a new project and make sure you choose the Empty Activity template. This is a game, so of course you don't need items such as the FAB button complicating matters. The first thing you want to do is change AppCompatActivity to activity. This means that we will not use the functions of the action bar. Similarly, we also want to make our game on full screen. Add the following code to onCreate before calling to installContentView ().setWindow (WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN); this.requestWindowFeature (Window.FEATURE_NO_TITLE); Note that if you write some code and it is highlighted in red, it probably means that you need to import the class. In other words, you have to tell Android Studio that you want to use certain statements and make them available. If you just click anywhere on the stressed word and then click Alt'Enter, then it will be done for you automatically! Creating a game viewY can be used for apps that use the XML script to determine the layout of views such as buttons, images, and tags. That's what setContentView line does for us. But then again, this game means that it doesn't have to have a browser window or scroll the recycler views. Instead, we want to show the canvas. In Android Studio, the canvas is the same as in art: it's an environment we can build on. So change this line to read both:setContentView (new GameView (it)) you'll find that it's once again highlighted red. But now, if you click Alt'Enter, you don't have the ability to import a class. Instead, you have the opportunity to create a class. In other words, we're going to make our own class that will determine what will go on canvas. This is something that will allow us to draw to the screen, and not just show ready-made views. So that's what I'm the right button on the package name in the hierarchy on the left and select the New Class. Now you will be presented with a window to create a class and you you call it GameView. Under SuperClass, write: android.view.SurfaceView which means that the class will inherit the methods - its capabilities - from the SurfaceView.In interface (s) field, you will write android.view.SurfaceHolder.Callback. As in any class, now we need to create our designer. Use this code: MainThread private stream; Public GameView - Super (context); getHolder (). Every time our class is called to make a new object (in this case our surface), it will work as a constructor and it will create a new surface. The line calls superclass, and in our case, that is, SurfaceView.By adding Callback, we can intercept events. Now override some methods:@Override public void surfaceChanged (SurfaceHolder holder, int format, int width, int height) - @Override public void surfaceCreated (owner of SurfaceHolder) - @Override public void surfaceDestroyed (owner of SurfaceHolder) - They basically allow us to redefine (hence the name) methods in the superclass (SurfaceView). Now there should be no reds in the code. Nice.You have just created a new class and every time we refer to this, it will build a canvas for your game to get painted on. Classes create objects, and we need another one. Creating Streams Our new class will be called MainThread. And his job will be to create a stream. The thread is essentially like a parallel code plug that can work simultaneously next to the main part of the code. You can have a lot of threads running all at once, thereby allowing things to happen simultaneously rather than sticking to a strict consistency. This is important for the game because we have to make sure that it continues to run smoothly, even when a lot is happening. Create your new class just like you did before, and this time it's going to expand the flow. As a designer, we'll just call it super. Remember that this is a super class that thread, and that can do all the hard work for us. It's like creating a dishwashing program that just calls washing machine. When this class is called, it is going to create a separate thread that works as an offshoot of the main. And that's where we want to create our GameView. This means that we also have to refer to the GameView class, and we also use SurfaceHolder, which contains canvas. So if the canvas is a surface, surfaceHolder is an easel. And GameView is what puts it all together. The full thing should look like this: MainThread's public class expands the thread - a private surfaceHolder surfaceHolder; Private gameView gameView; MainThread (SurfaceHolder surfaceHolder, GameView gameView) - super() this.surfaceHolder - surfaceHolder; this.gameView - gameView; It's shvit. We have GameView and Thread! Creating a game cycle we now have the raw materials needed for our game, but nothing Happens. This is where the game loop comes in. Our goal is to make it as consistent as possible, so there are no stutters or hiccups in the frame, which I will study a little later. At the moment we are still in the MainThread class and are going to redefine the method from the superclass. This one works. And it goes a little something like this: @Override public void perspective () try canvas - this.surfaceHolder.lockCanvas (); synchronised (surfaceHolder) - this.gameView.update this.gameView.draw You'll see a lot of add-ons, so we need to add a few more variables and links. Go back to the top and add: private SurfaceHolder surfaceHolder; Private game GameViewView; Private boolean works; public static canvas canvas; Don't forget to import canvas. Canvas is something we're actually relying on. As for lockCanvas, this is important because it is something that essentially freezes the canvas to allow us to rely on it. This is important because otherwise, you may have multiple threads trying to rely on it at once. Just know that in order to edit the canvas, you must first block the canvas. Upgrading is a method that we are going to create and this is where the fun things will happen later. Trying to catch meanwhile just Java requirements that show that we are willing to try and handle exceptions (errors) that may occur if the canvas is not ready, etc. Finally, we want to be able to start our flow when we need it. To do this, we will need another method that will allow us to set things in motion. This is what works variable for (note that Boolean is a type of variable that is only ever true or false). Add this method to mainThread class: public set of voidsRunning -- launch - isRunning; But for now, one thing still needs to be highlighted, and that's an update. That's because we haven't created an upgrade method yet. So pop back into GameView and now add method.public invalid update () We also have to start streaming! We're going to do this in our surfaceCreated method: @Override public void surfaceCreated (SurfaceHolder holder) - thread.setRunning (admittedly); thread.start (); We also have to stop the flow when the surface is destroyed. As you might have guessed, we are coping with this in the surfaceDestroyed method. But seeing how it can really take a few tries to stop the flow, we're going to put it in a loop and use to try to catch again. As so: @Override public void surfaceDestroyed (SurfaceHolder holder) - boolean retry - while (repeat) (repeat) Try q thread.setRunning (false); thread.join(); - Catch (InterruptedException e) - e.printStackTrace(); - repeat - false; And finally, head up to the designer and make sure to create a new instance of your flow, otherwise you will get a terrible zero pointer exception! And then we're going to make GameView focus, meaning it can handle events.thread and the new MainThread (getHolder); setFocusable (true); Now you can finally actually check out this thing! That's right, click run and it should actually work without any mistakes. Get ready to be blown away! This is... This is... Blank screen! All this code. For an empty screen. But, it's a blank screen of possibilities. You have a surface and works with the game cycle to handle events. Now all that's left is to make things happen. It doesn't even matter if you don't follow everything in the tutorial until now. The fact is that you can just recycle this code to start making glorious games! Doing graphicsRight, now we have a blank screen to draw on all we need to do is build on it. Fortunately, this is the simple part. All you have to do is override the toss method in our GameView class and then add some beautiful photos:@Override public void to draw (canvas canvas) - super.draw (canvas); if (canvas ! - null) - canvas.drawColor (Color.WHITE); Paint paint - new paint (); paint.setColor (Color.rgb(250, 0, 0)); canvas.drawRect (100, 100, 200, 200, paint); You run this and now you have to have a pretty red square in the top left left of the otherwise white screen. This is definitely an improvement. Theoretically, you can create almost the entire game by sticking it inside this method (and redefining AttouchEvent to process input), but it won't be a very good way to go about things. Placing a new paint inside our cycle will significantly slow down the process, and even if we put it elsewhere, adding too much code to the drawing method will become ugly and difficult to follow. Instead, it makes a lot more sense to handle game objects with their own classes. Let's start with a character that will be called CharacterSprite. Go and do it. This class is going to draw a sprite on canvas and will look like a sopublic of the CharacterSprite class - a private image of Bitmap; Public CharacterSprite (Bitmap bmp) - public void draw (canvas canvas) - canvas.drawBitmap (image, 100, 100, null); Now, to use this, you need to download a bit card and then call the class from GameView. Add a link to the private characterSprite characterSprite, and then to the surfaceCreated method, add the line:characterSprite - new You can see that the bit card we're downloading is stored in resources and is called avdgreen (this was from the previous game). Now all you have to do is pass this bit card to a new class in the toss method with:characterSprite.draw (canvas); Now click run and you have to see your graphic appear on the screen! It's B.B.B. I draw it in my school textbooks. What if we wanted to get this little guy to move? Simply: we simply create x and y variables for its positions, and then change those values in the update method. So add help to your CharacterSprite and then draw your bitmap to x, y. Create an update method here, and for now we're just going to try: Every time the game cycle works, we'll move the character down the screen. Remember that the coordinates are measured from above so 0 at the top of the screen. Of course, we should call the upgrade method in CharacterSprite from the update method in GameView.Press play again, and now you'll see that your image is slowly traced across the screen. We don't win a single awards game just yet, but that's the beginning! Ok to make things a little more interesting, I'm just going to give up some inflatable ball code here. This will make our graphic bounce across the screen from the edges like those old Windows screensavers. You know, strangely hypnotic ones.public invalid update () - xVelocity; y yVelocity; If (x (x lt; 0)) xStop - xStop - -1; If (y and gt; screenHeight - image.getHeight (y and lt; 0)) - yVelocity - yVelocity -1; You will also need to identify these variables: private int xVelocity No 10; private int yVelocity No 5; privately int screenWidth and Resources.getSystem (.getDisplayMetrics ().widthPixels; Private int screenHeight and Resources.getSystem (.getDisplayMetrics ().).heightPixels;OptimizationThere is a lot more to delve into here, from processing player input, to scaling images, to managing with lots of characters all moving around the screen at once. Right now, the character is bouncing, but if you look very closely there is a slight stutter. It's not scary, but the fact that you can see it with the naked eye is something of a warning sign. Speed also varies greatly on the emulator compared to the physical device. Now imagine what happens when you have a ton going on the screen right away! There are several solutions to this problem. What I want to do to start with is create a private integer in MainThread and call that targetFPS. It will have a value of 60. I'm going to try to get my game to run at that speed, and meanwhile, I'll check to make sure that's the case. Для этого я также хочу, чтобы частный двойной называется averageFPS.I'i также собирается обновить Start in order to measure how long each cycle of the game takes and then suspend that game cycle temporarily if it is ahead of targetFPS. Then we're going to how long it took and then print that so we could see it in log.@Override public emptiness run () - a long start time; For a long timeMillis; Long waittime Long total time No 0; int frameCount No 0; Long TargetTime No. 1000 / Target FPS; While (launch) - startTime - System.nanoTime (); Canvas - zero; Try canvas - this.surfaceHolder.lockCanvas Synchronized (surfaceholder) - this.gameView.update this.gameView.draw - Catch (Exception e) - after all, if (canvas !) - null) - try surfaceHolder.unlockCanvasAndPost (canvas); - Catch (Exception e) - e.printStackTrace (); TimeMillis (System.nanoTime() - startTime) / 1000000; waitTime - target time - timeMillis; Try - this.sleep (waitTime); Catch (Exception e) - totalTime and System.nanoTime () - startTime; frameCount; If (frameCount - targetFPS) - an average of 1000 euros / (total time / frameCount) / 10,000,000 euros); frameCount No 0; Total time No 0; System.out.println (averageFPS); Now our game is trying to lock it FPS to 60, and you have to find that it usually measures a fairly steady 58-62 FPS on a modern device. On the emulator though you can get a different result. Try changing that 60 to 30 and see what happens. The game is slowing down and now it has to read 30 in your logcat. Closing ThoughtsThere are some other things we can do to optimize performance too. There's a great blog post on the subject here. Try to refrain from creating new paint or bitmaps inside the loop and do all the initializations outside before the game starts. If you're planning to create the next hit Android game, then there are definitely easier and more efficient ways to go about it these days. But there are certainly still use scripts in order to be able to draw on canvas, and it is a very useful skill to add to your repertoire. I hope this guide has helped a few and wish you the best of luck in upcoming coding ventures! Next - Java Java Beginners' Guide learning java for android studio