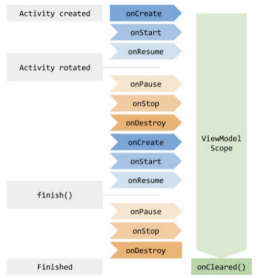
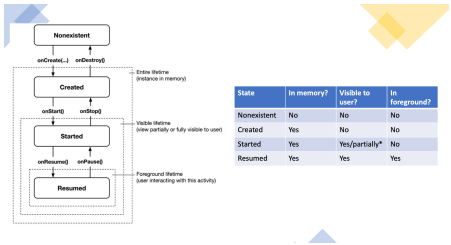
Native Executable are unbounded, can go anywhere on the computer and fo anything. The OS offers some minniaml protection. C++ source->C++ compiler->Native EXE **Virtual Machines** is a software application that emulates a physical machine. Java source->Java compiler->java byte code->JVM **Managed code** is compiled into an executable. Run directly on the CPU, but only with a manager running with it (ART,replaced Dalvik VM) **DVM** is similar to JVM, but is lightweight and can spawn itself quickly. Every Android app wun in its own DVM. Java source->java compiler->java byte code->Dalvik compiler->DVM.**ART**: Takes longer to install but runs and starts up quicker.1.Ahead-of-time(AOT)complication
->improve app performance.2.Imporve garbage collection.3.Debugging and development improvements.**GUI Components**:Display data/capture user input/ Allow the user to interact with.**SetContentView()**-> Inflating views and putting them on Screen. **findVIewByID()-**>Getting references to inflated views. **Rotating** the device changes the configuration.**JVM test** executes on your development machine through JVM.(faster)**Instrument test** excutes directly on an Android device.(more correct/ can only interact Android SDK).Both use JUnit framework.Test:
1.Setup environment(given).2.Test unit of code(when).3.Verify code behavior(then).**Second Activity:Use startActivity(Intent)**: an Intent is an object that a component can use to communicate with OS.**Explict intent:** start specific activities.**Implicit intent:** start in another. **build.giadle(module:app) file**->SDK version-> minSdkVersion(oldest version can install)/targetSdkVersion(expects to run version).**Jetpack:** A set of libraries from Google to make developmenteasier(Need to add to file).**Fragments{**Reuseable GUI components gtouped together in a layout/ An activity can have multiple fragments/ Activity host the fragments/ Why?{Reuse of code/ Excellent way to hangle multiple layouts and devices}};**Resouce** is the part of your file which is not code(image/audio/XML) live in app/res. Access read Resource ID.Andriod application are "**event driven**", so before click, we are "listening for" the object to respond a event is called "**listener**" .Fragment.onCreate(Bundle?) and other Fragment lifecycle functions must be **public**, because they will be called by whichever activity is hosting the fragment.
Activities: An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. A different app can start any one of these activities if the email app allows it. For example, a camera app might start the activity in the email app for composing a new email to let the user share a picture. An activity facilitates the following key interactions between system and app: Keeping track of what the user currently cares about—what is on-screen—so that the system keeps running the process that is hosting the activity. Knowing which previously used processes contain stopped activities the user might return to and prioritizing those processes more highly to keep them available. Helping the app handle having its process killed so the user can return to activities with their previous state restored. Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. The primary example of this is sharing. You implement an activity as a subclass of the `Activity` class
The manifest file: Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, AndroidManifest.xml. Your app declares all its components in this file, which is at the root of the app project directory. The manifest does a number of things in addition to declaring the app's components, such as the following: Identifies any user permissions the app requires, such as internet access or read-access to the user's contacts. Declares the minimum API level required by the app, based on which APIs the app uses. Declares hardware and software features used or required by the app, such as a camera, Bluetooth services, or a multitouch screen. Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.
Services: Services are this part which users can't see directly but It's very important and one of the main parts of your app. Service runs in the background to perform long-running operations even if the users are not active and the application is destroyed. A service interacts with a content provider, performs multiplayer in the background threads.
Broadcast Receivers: Broadcast receivers help your app to communicate with android systems. With the help of a broadcast receiver, your app can communicate with other app events and system events. One has to be very aware of broadcast receivers. It can help to build a fast and user-friendly app by communicating with the system.
Content Providers: Content Provider is a very important part of android. It handles the data part of your app. It helps to provide the main content of your app with the main data that you want to show in your app. Not only you can access your app data you can also access the data of other apps as per your need and app permissions.
Java Folder: The JAVA folder consists of the java files that are required to perform the background task of the app. It consists of the functionality of the buttons, calculation, storing, variables, toast(small popup message), programming function, etc. The number of these files depends upon the type of activities created.
Resource Folder: The res or Resource folder consists of the various resources that are used in the app. This consists of sub-folders like drawable, layout, mipmap, raw, and values. The drawable consists of the images. The layout consists of the XML files that define the user interface layout. These are stored in res.layout and are accessed as R.layout class. The raw consists of the Resources files like audio files or music files, etc. These are accessed through R.raw.filename. values are used to store the hardcoded strings(considered safe to store string values) values, integers, and colors. It consists of various other directories
Gradle Files: Gradle is an advanced toolkit, which is used to manage the build process, that allows defining the flexible custom build configurations. Each build configuration can define its own set of code and resources while reusing the parts common to all versions of your app. The Android plugin for Gradle works with the build toolkit to provide processes and configurable settings that are specific to building and testing Android applications. Gradle and the Android plugin run independently of Android Studio. This means that you can build your Android apps from within Android Studio. The flexibility of the Android build system enables you to perform custom build configurations without modifying your app's core source files.
LogCat Window is the place where various messages can be printed when an application runs. Suppose, you are running your application and the program crashes, unfortunately. Then, Logcat Window is going to help you to debug the output by collecting and viewing all the messages that your emulator throws. So, this is a very useful component for the app development because this Logcat dumps a lot of system messages and these messages are actually thrown by the emulator.This means, that when you run your app in your emulator, then you will see a lot of messages which include all the information, all the verbose messages, and all the errors that you are getting in your application. Suppose, an application of about 10000 lines of code gets an error. So, in that 10000 line codes, to detect the error Logcat helps by displaying the error messages.Errors in different modules and methods can be easily detected with the help of the LogCat window.Using LogCat window:The LogCat prints an error using a Log Class. The class which is used to print the log messages is actually known as a Log Class. So, this class is responsible to print messages in the Logcat terminal.
Toast : It is an Android UI component that is used to show message or notification that does not require any user action. It is independent to the activity in which it is being shown and disappears automatically after the set duration.
SnackBar: It is used to show popup message to user that requires some user action. It can disappear automatically after set time or can be dismissed by user.
Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states. The following sections introduce these callbacks.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.When onCreate() finishes, the next callback is always onStart().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.The onPause() callback always follows onResume().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.An activity in the Paused state may continue to update the UI if the user is expecting the UI to update. Examples of such an activity include one showing a navigation map screen or a media player playing. Even if such activities lose focus, the user expects their UI to continue updating.
You should not use onPause() to save application or user data, make network calls, or execute database transactions. For information about saving data, see Saving and restoring activity state.Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused 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.The next callback that the system calls is either onRestart(), if the activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating. 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.This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.
View represent the basic building blocks for user interface components • Rectangular area on screen• Base class for widgets which are used to create interactive UI
components (buttons, text fields, etc) **ViewGroup** is the base class for layouts, invisible containers that hold other Views or ViewGroups and define their layout properties. ViewGroups don't display, they orchestrate.
How Viewmodel Works: When the user finishes an activity, they expect their UI state to be reset.• When the user rotates an activity, they expect their UI state to be the same after rotation.ViewModel offers a way to keep an activity's UI state data in memory across configuration changes.• Its lifecycle mirrors the user's expectations - it survives configuration changes and is destroyed only when its associated activity is finished.• When you associate a ViewModel instance with an activity's lifecycle, it is said to be scoped to that activity's lifecycle.• This means the ViewModel will remain in memory, regardless of the activity's state, until the activity is finished.• Once the activity is finished (such as by the user closing the app from the overview screen),the ViewModel instance is destroyed.• This means that the ViewModel stays in memory during a configuration change, such as rotation.• During the configuration change, the activity instance is destroyed and re-created, but any ViewModels scoped to the activity stay in memory.
AndroidX Jetpack: Android Jetpack is a set of libraries from Google to make development easier. Encourages best practices, reduces code,
and creates consistency across versions. Need to add in the build.gradle(module:app) file. Some Jetpack Components are…
Fragment: A fragment is like a kind of subactivity that's used to control part of the screen. **Navigation:** This includes tools to navigate from screen to screen, and safely pass arguments between them. **LiveData:** This lets you build apps that respond to data changes as soon as they happen. **Data binding:** This lets you build responsive layouts that can access Kotlin methods and properties. **Compose:** A toolkit for building native Android apps without using layout files. **Room:** A persistence library that lets you create and interact with databases. **Recycler view:** An efficient list you can use to display data and navigate between screens.**Constraint layout:** Use this to build complex, flexible layouts without the overheads of nesting. **View Model:** This lets you move a screen's business logic into a separate class.Jetpack provides backward compatibility on older devices for consistent behavior **AppCompat** consistent look and feel. **WorkManager** consistent environment for essential tasks
Fragments: Reusable GUI components grouped together in a Layout.Think of it as a way to group views and logic into a reusable component. An Activity can have multiple fragments Activities host the fragments. Fragments cannot run without a host. Fragments should not talk to each other directly. If they did they would be tightly coupled. **Tight Coupling** is bad practice. Communicate through an interface to the Activity (e.g. ViewModel) instead. Activities responsible for determining how to layout fragments as well as providing the communication facility between fragments. Why use Fragments? Reuse of code, Excellent way to handle multiple layouts and devices, Eg, Big vs. Small Screen
Fragment Manager: FragmentManager Class responsible for performing actions on your app's fragments, such as adding, removing, or
replacing them, and adding them to the back stack Don't interact with the FragmentManager if using Jetpack – it does it for you. Can host one or more child
Fragments
Backstack: Activities are arranged in a stack— the back stack—in the order in which each activity is opened. For example, an email app
might have one activity to show a list of new messages When the user selects a message, a new activity opens to view that message. This new activity is added to the back
Stack Then, if the user presses or gestures Back, that new activity is finished and popped off the stack
Fragment Lifecycle: Like actives but with important differences Similar in that fragments are created, started, and resume with functions we can override. Different in that the lifecycle functions are call by FragmentManager of the hosting activity and not the OS
RecyclerView: RecyclerView Efficiently displays large sets of data You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're
Needed. As the name implies, RecyclerView recycles those individual elements. When an item scrolls off the screen, RecyclerView doesn't destroy its view, it reuses the view for new items that have scrolled onscreen. RecyclerView improves performance and your app's responsiveness, and it reduces power consumption. The RecyclerView doesn't create the ViewHolders – it asks an adapter • An adapter is just a controller object that is between the RecycleView and the data it should display • The adapter: • Creates the needed ViewHolders • Binds data from the model layer • The RecyclerView • Asks the adapter to create ViewHolders • Asks the adapter to bind ViewHolders to the item from the data at a given position
Styles:a collection of view attribute values for a single view(font color, font size, background color) ;a Map that the keys are all view attributes;Styles are specific to a single type of widget because different widgets support different sets of attributes. It can be set in the layout but it is best to do it in a separate file
Themes:A theme is a collection of named resources which can be referenced later by styles, layouts etc • Can be applied to an entire app, activity, or view hierarch – not just a single view • When applying a theme, every view in the app or activity applies the theme's resources • Can also apply to non-view elements like status bar and window background • They provide semantic names to Android resources so you can refer to them later e.g. colorPrimary is a semantic name for a given color. These named resources are known as theme attributes, so a theme is Map • Theme attributes are different from view attributes because they're not properties specific to an individual view type but semantically named pointers to values which are applicable more broadly in an app • A theme provides concrete values for these named resources • In the previous example the colorPrimary attribute specifies that the primary color for this theme is teal • By abstracting the resource with a theme, we can provide different concrete values (such as colorPrimary=orange) in different themes. A theme is like an interface • Programming to an interface allows you to decouple the public contract from the implementation allowing you to provide different implementations • Themes play a similar role • By writing our layouts and styles against theme attributes, we can use them under different themes, providing different concrete resources.The power of themes really comes from how you use them; you can build more flexible widgets by referencing theme attributes • Different themes provide concrete values later

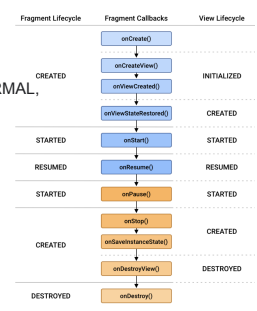
Theme Scope: A Theme is accessed as a property of a Context and can be obtained from any object which is or has a Context e.g. Activity, View or ViewGroup • These objects exist in a tree, where an Activity contains ViewGroups which contain Views etc • Specifying a theme at any level of this tree cascades to descendent nodes e.g. setting a theme on a ViewGroup applies to all the Views within it (in contrast to styles which only apply to a single view)
Scope:This can be extremely useful, say if you want a dark themed section of an otherwise light screen • Note that this behavior only applies at layout inflation time • While Context offers a setTheme method, or Theme offers an applyStyle method, these need to be called before inflation • Setting a new theme or applying a style after inflation will not update existing views
Example of Theme:Say you have a blue theme for your app, but some Pro screens get a fancy purple look and you want to provide dark themes with tweaked colors • If we tried to achieve this using only styles, we would have to create 4 styles for the permutations of Pro/non-Pro and light/dark • As styles are specific to a type of view (Button, Switch etc) you'd need to create these permutations for each view type in your app • By using styles and themes we can isolate the parts which alter by theme as theme attributes so we only need to define a single style per view type • For the above example we might define 4 themes which each provide different values for the colorPrimary theme attribute, which these styles then refer to and automatically reflect the correct value from the theme
Coroutines:Android runs Linux which (like all modern operating systems) support multiple threads • Recall that a thread is path of execution through code. • Coroutines are Kotlin's first-party solution for defining work that will run asynchronously and are fully supported on Android • They are based on the idea of functions being able to suspend, meaning that a function can be paused until a long-running operation completes • When the code running in a coroutine is suspended, the thread that the coroutine was executing on is free to work on other things, like drawing your UI, responding to touch events, or making more expensive calculations. Coroutines provide a high-level and safer set of tools to help you build asynchronous code • Under the hood, Kotlin's coroutines use threads to perform work in parallel, but you often do not have to worry about this detail • Coroutines make it easy to start work on the main thread, hop over to a background thread to perform asynchronous work, and then return the result back to the main thread
Using Coroutines:To run your code in a coroutine, you use a coroutine builder • A coroutine builder is a function that creates a new coroutine. Most coroutine builders also start executing the code within the coroutine immediately after creating it • Several builders are defined for you in the Coroutines library • The most commonly used coroutine builder is launch, a function that is defined as an extension to a class called CoroutineScope.
CoroutineScope:Every coroutine builder launches its coroutines inside a coroutine scope • A coroutine's scope has control over how the coroutine's code executes • This includes setting up the coroutine, canceling the coroutine, and choosing which thread will be used to run the code • On Android, this idea of scopes maps neatly onto the various lifecycles • The Activity, Fragment, and ViewModel classes have unique lifecycles and coroutine scopes to match • For ViewModels, you have access to the viewModelScope class property. • This viewModelScope is available from the time your ViewModel is initialized, and it cancels any coroutine work still running when the ViewModel is cleared out from memory
viewLifecycleView:viewLifecycleScope is active for as long as the view is in memory (in other words, after onViewCreated() but before onDestroyView()) • After the view is destroyed, the coroutine scope – and all work within it – is canceled • Only update the UI while the Fragment is in the started lifecycle state or higher • It does not make sense to update the UI when it is not visible
repeatOnLifecycle:With the repeatOnLifecycle(…) function, you can execute coroutine code while your fragment is in a specified lifecycle state • For example, you only want this coroutine code to execute while your fragment is in the started or resumed state • Also, repeatOnLifecycle is itself a suspending function • You will launch it in your view lifecycle scope, which will cause your work to be canceled permanently when your view is destroyed. • You are not required to call the repeatOnLifecycle(…) function in the onStart() callback • Normally, you use the onViewCreated(…) callback to hook up listeners to views and to set the data within those views • This is the perfect spot to handle your coroutine code, too
Room Component Library: Room is a set of API, annotations, and a compiler • The API contains classes you extend to define your database and build an instance of it • We use the annotations to indicate things like which classes need to be stored in the database, which class represents your database, and which class specifies the accessor functions to your database tables • The compiler processes the annotated classes and generates the implementation of your database. Creating a Database: First, add the required dependencies in the Gradle file • Three steps to creating a DB • annotate your model class to make it a database entity • create the class that will represent the database itself • create a type converter so that your database can handle your model data
SQLite and Type Converter: The DB in Android • Room does the object-relational mapping (ORM) between the Kotlin code and the SQL DB • Can store primitive types, enum classes and UUID but not the Date type – need to convert – use a type converter Data Access Object (DAO): To access the contents in the DB need to create a DAO • The DAO is an interface that has functions for each operation
Access the DB using Repo Pattern:A repository class encapsulates the logic for accessing data from a single source or a set of sources • It determines how to fetch and store a particular set of data, whether locally in a database or from a remote server • The UI code will request all the data from the repository, because the UI does not care how the data is stored or fetched • Those are implementation details of the repository itself. Observing Changes to the DB: • Use Flow and StateFlow from coroutines • a flow represents an asynchronous stream of data • Throughout their lifetime, flows emit a sequence of values over an indefinite period of time that get sent to a collector • The collector will observe the flow and will be notified every time a new value is emitted in the flow. • Flows are a great tool for observing changes to a database.
Stateflow: StateFlow is a specialized version of Flow that is designed specifically to share application state • StateFlow always has a value that observers can collect from its stream • It starts with an initial value and caches the latest value that was emitted into the stream • It is the perfect companion to the ViewModel class, because a StateFlow will always have a value to provide to fragments and activities as they get re-created
Single Activity Architecture: • This makes sure the app is in control of everything being shown on the screen • Using multiple activities will have the system take control of navigation and animations – may not want that and no way to customize those behaviors • Using a single activity, we maintain control on how the app behaves
NavController: Manages navigation with a host • Navigation is determined by the navigation graph • Don't create an instance of the class, just need to find it using findNavController • Will search the hierarchy and fragment for the NavController and return it • Then can call the navigate function passing in a resource ID either for the destination or a navigation action
SafeArgs: The Safe Args plugin not only generates code to perform type-safe navigation but also allows you to safely access navigation arguments once the user is at their destination • By using the navArgs property delegate, you can access the navigation arguments for a particular destination in a type-safe manner • The Safe Args plugin generates classes that hold all the arguments for a destination, naming them with the name of the destination plus "Args."
Unidirectional Data Flow: Applications must respond to input from multiple sources: data being loaded from the back end as well as inputs from the user • If you do not have a plan in place to combine these sources, you could code yourself into a mess of complex logic that is difficult to maintain • Unidirectional data flow is an architecture pattern that has risen to prominence and that plays nicely with the reactive patterns you have been using with Flow and StateFlow • Unidirectional data flow tries to simplify application architecture by encapsulating these two forces – data from the back end and input from the user – and clarifying their responsibilities
Kotlin:val(read-only);var(mutable);**listOf**(read-only list);**MutableList**<String>(mutable list)->to protect use **List**<String>(read-only);**Set**(can repeat/unordered);**Map**(use "**to**"/**.conrainsKey()**);**when**{a<0->"a" else->"b"} 1.2->1,2;1..<2->1;2 **downTo** 1->2,1;1..3 **step** 2->1,3;**Int**:Byte8/Short16/Int32/Long64 **Floats**:Float(32/24significant)/Double(64/53)/
ConstraintLayout->(app:layoyt_constraintBottom_toBottomOf="parent")**LinearLayout->**(android: orientation="horizontal"/"vertical") (android:layout_width/layout_height="match_parent/wrap_content")(android:text=)
Resource ID(android: id="@+id/button")**Design preview/Blueprint preview**. **onCreated(Bundle?)** function is called when an instance of the activity subclass is created.**Activity.setContentView(IsyoutResID:Int)**/ R.layout.activity_main(inflates a layout and put it on screen);**Activity.findViewById(Int)**;**lateinit**(not-null value)

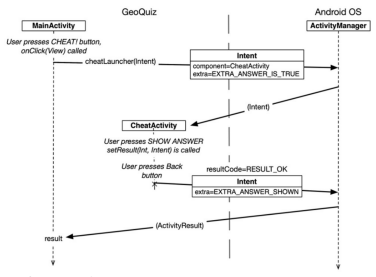



```
true.setOnClickListener{view:View->
            Toast.makeText(context, R,string.correct,Toast.LENGTH_SHORT).show()
            Snackbar.make(this,R,string.correct,Snack.LENGTH_SHORT).show())}
Binding=ActivityMainBinding.inflate(layoutInflater)Log:private const val TAG = "MainActivity" Log.d(TAG,"called?")
companion object {
    fun newIntent(packageContext: Context, answerIsTrue: Boolean): Intent {
      return Intent(packageContext, CheatActivity::class.java).apply {putExtra(EXTRA_ANSWER_IS_TRUE, answerIsTrue)} }}
val answerIsTrue = quizViewModel.currentQuestionAnswer
val intent = CheatActivity.newIntent(this@MainActivity, answerIsTrue)
answerIsTrue = intent.getBooleanExtra(EXTRA_ANSWER_IS_TRUE, false)
val intent = Intent(context, SecondActivity::class.java)
startActivity(intent)
Create a data object:
data class Crime{
  val id: UUID,val title: String}
sbcel.setOnSeekBarChangeListener(object: SeekBar.OnSeekBarChangeListener{
      override fun onProgressChanged(seekBar: SeekBar?, progress: Int, fromUser: Boolean) {
        val fah = Trans().celToFah(progress/1.0)
        sbfah.progress = fah.toInt()}
      override fun onStartTrackingTouch(seekBar: SeekBar?) {}
      override fun onStopTrackingTouch(seekBar: SeekBar?) {}
class MainActivity : AppCompatActivity(),SensorEventListener {
        private lateinit var sensorManager: SensorManager
        private fun setUpSensorStuff(){
                sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
                sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)?.also{sensorManager.registerListener(this,it,SensorManager.SENSOR_DELAY_NORMAL,
                                                                                SensorManager.SENSOR_DELAY_NORMAL) }}
        override fun onSensorChanged(event: SensorEvent?) {
            if(event?.sensor?.type == Sensor.TYPE_ACCELEROMETER){
                val x = kotlin.math.abs(lastX-event.values[0])
                if (x>=sbacc.progress){
```



State	In memory?	Visible to user?	In foreground?
Nonexistent	No	No	No
Created	Yes	No	No
Started	Yes	Yes/partially*	No
Resumed	Yes	Yes	Yes





Sequence diagram for intents and extras in GeoQuiz