Following

Introduction to Android Activities with Kotlin

Learn about one of the most important concepts within Android apps with this introduction to Android activities tutorial, using Kotlin!

By Joe Howard & Namrata Bandekar Jul 26 2017 · Beginner · Article · 30 mins

Download Materials

Bookmark

Version

- Kotlin 1.1, Android 4.4, Android Studio 3

Update Note: This tutorial has been updated to Kotlin and Android Studio 3.0 by Joe Howard. The original tutorial was written by Namrata Bandekar.

Note: The sample code for this tutorial has been updated to be compatible with Android Studio 3.2.1 or later.

When you make an Android app, it's pretty likely that the first thing you'll do is plot and plan how you'll take over the world. Just kidding. Actually, the first thing you do is create an *Activity*. Activities are where all the action happens, because they are the screens that allow the user to interact with your app.

In short, activities are one of the basic building blocks of an Android application.

In this Introduction to Android Activities Tutorial, you'll dive into how to work with activities. You'll create a to-do list app named *Forget Me Not* and along the way learn:

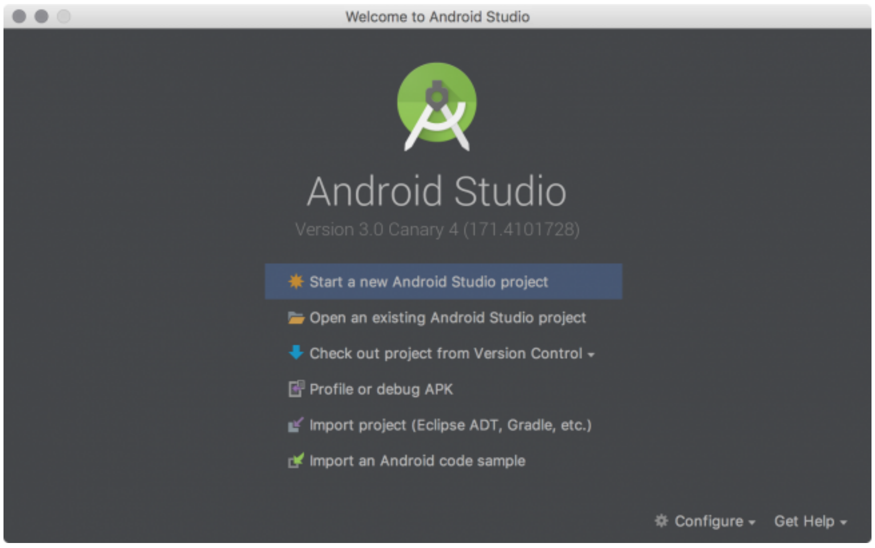
- The process for creating, starting and stopping an activity, and how to handle navigation between activities.
- The various stages in the lifecycle of an activity and how to handle each stage gracefully.
- The way to manage configuration changes and persist data within your activity.

In addition, you'll use the (newly official) *Kotlin* programming language and Android Studio 3.2. You'll need to use Kotlin 1.2.71 or later and Android Studio 3.2.1 or later.

This tutorial assumes you're familiar with the basics of Android development. If you're completely new to Kotlin, XML or Android Studio, you should take a look at the Beginning Android Development Series and Kotlin For Android: An Introduction before you start.

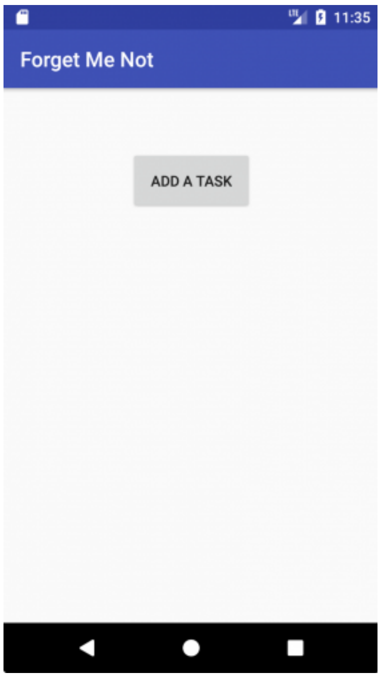
Getting Started

Download the starter project using the *Download materials* button at the top or bottom of the tutorial. Open *Android Studio* 3.2.1 or later, and choose *Open an existing Android Studio project*. Then select the project you just extracted.


Welcome to Android Studio
Android Studio
Version 3.0 Canary 4 (171.4101728)
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Eclipse ADT, Gradle, etc.)
Import an Android code sample
Configure
Get Help

Your subject and new best friend for the rest of the tutorial is *Forget Me Not*, which is a simple app that lets you add and delete tasks from a list. It also displays the current date and time so you're always on top of your game!

Run the project as it is now and you'll see a very basic screen:



At this point, there isn't much you can do. Your to-do list is empty and the "ADD A TASK" button does nothing when you tap it. Your mission is to liven up this screen by making the to-do list modifiable.

Activity Lifecycle

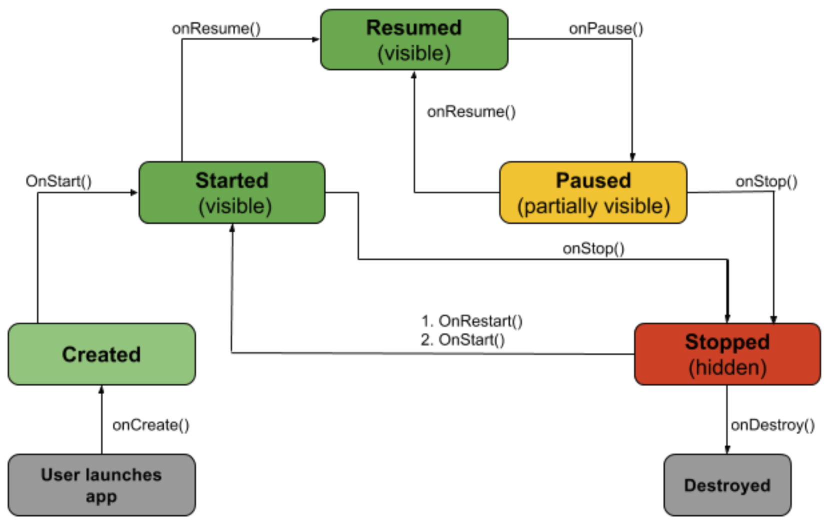
Before firing up your fingers, indulge yourself in a bit of theory.

As mentioned earlier, activities are the foundations upon which you build screens for your app. They encompass multiple components the user can interact with, and it's likely that your app will have multiple activities to handle the various screens you create.

Having all these activities doing different things makes it sound like developing an Android app is a complicated undertaking.

Fortunately, Android handles this with specific *callback methods* that initiate code only when it's needed. This system is known as the *activity lifecycle*.

Handling the various lifecycle stages of your activities is crucial to creating a robust and reliable app. The lifecycle of an activity is best illustrated as a step pyramid of different stages linked by the core callback methods:



Following the diagram above, you can picture the lifecycle in action as it courses through your code. Take a closer look at each of the callbacks:

- *onCreate()*: Called by the OS when the activity is first created. This is where you initialize any UI elements or data objects. You also have the `savedInstanceState` of the activity that contains its previously saved state, and you can use it to recreate that state.
- *onStart()*: Just before presenting the user with an activity, this method is called. It's always followed by `onResume()`. In here, you generally should start UI animations, audio based content or anything else that requires the activity's contents to be on screen.
- *onResume()*: As an activity enters the foreground, this method is called. Here you have a good place to restart animations, update UI elements, restart camera previews, resume audio/video playback or initialize any components that you release during `onPause()`.
- *onPause()*: This method is called before sliding into the background. Here you should stop any visuals or audio associated with the activity such as UI animations, music playback or the camera. This method is followed by `onResume()` if the activity returns to the foreground or by `onStop()` if it becomes hidden.
- *onStop()*: This method is called right after `onPause()`, when the activity is no longer visible to the user, and it's a good place to save data that you want to commit to the disk. It's followed by either `onRestart()`, if this activity is coming back to the foreground, or `onDestroy()` if it's being released from memory.
- *onRestart()*: Called after stopping an activity, but just before starting it again. It's always followed by `onStart()`.
- *onDestroy()*: This is the final callback you'll receive from the OS before the activity is destroyed. You can trigger an activity's desctruction by calling `finish()`, or it can be triggered by the system when the system needs to recoup memory. If your activity includes any background threads or other long-running resources, destruction could lead to a memory leak if they're not released, so you need to remember to stop these processes here as well.

Note: You do not call any of the above callback methods directly in your own code (other than superclass invocations) — you only override them as needed in your activity subclasses. They are called by the OS when a user opens, hides or exits the activity.

So many methods to remember! In the next section, you'll see some of these lifecycle methods in action, and then it'll be a lot easier to remember what everything does.

Configuring an Activity

Keeping the activity lifecycle in mind, take a look at an activity in the sample project. Open *MainActivity.kt*, and you'll see that the class with its `onCreate()` override looks like this:

```
class MainActivity : AppCompatActivity() {
  // 1
  private val taskList: MutableList<String> = mutableListOf()
  private val adapter by lazy { makeAdapter(taskList) }

  override fun onCreate(savedInstanceState: Bundle?) {
    // 2
    super.onCreate(savedInstanceState)
    // 3
    setContentView(R.layout.activity_main)

    // 4
    taskListView.adapter = adapter

    // 5
    taskListView.onItemClickListener = AdapterView.OnItemClickListener { parent, view, position, id -> }
  }

  // 6
  fun addTaskClicked(view: View) {

  }

  // 7
  private fun makeAdapter(list: List<String>): ArrayAdapter<String> =
      ArrayAdapter(this, android.R.layout.simple_list_item_1, list)
}
```

Here's a play-by-play of what's happening above:

1. You initialize the activity's properties, which include an empty mutable list of tasks and an adapter initialized using `by lazy`.
2. You call `onCreate()` on the superclass — remember that this is (usually) the first thing you should do in a callback method. There are some advanced cases in which you may call code prior to calling the superclass.
3. You set the content view of your activity with the corresponding layout file resource.
4. Here you set up the adapter for `taskListView`. The reference to `taskListView` is initialized using *Kotlin Android Extensions*, on which you can find more info here. This replaces `findViewById()` calls and the need for other view-binding libraries.

5. You add an empty `OnItemClickListener()` to the `ListView` to capture the user’s taps on individual list entries. The listener is a Kotlin lambda.
6. An empty on-click method for the “ADD A TASK” button, designated by the *activity_main.xml* layout.
7. A private function that initializes the adapter for the list view. Here you are using the Kotlin = syntax for a single-expression function.

Note: To learn more about `ListViews` and `Adapters`, refer to the Android Developer docs.

Your implementation follows the theory in the previous section — you’re doing the layout, adapter and click listener initialization for your activity during creation.

Starting an Activity

In its current state, the app is a fairly useless lump of ones and zeros because you can’t add anything to the to-do list. You have the power to change that, and that’s exactly what you’ll do next.

In the *MainActivity.kt* file you have open, add a property to the top of the class:

```
private val ADD_TASK_REQUEST = 1
```

You’ll use this immutable value to reference your request to add new tasks later on.

Then add this import statement at the top of the file:

```
import android.content.Intent
```

And add the following implementation for `addTaskClicked()`:

```
val intent = Intent(this, TaskDescriptionActivity::class.java)
startActivityForResult(intent, ADD_TASK_REQUEST)
```

When the user taps the “ADD A TASK” button, the Android OS calls `addTaskClicked()`. Here you create an `Intent` to launch the `TaskDescriptionActivity` from `MainActivity`.

Note: There will be a compile error since you have yet to define `TaskDescriptionActivity`.

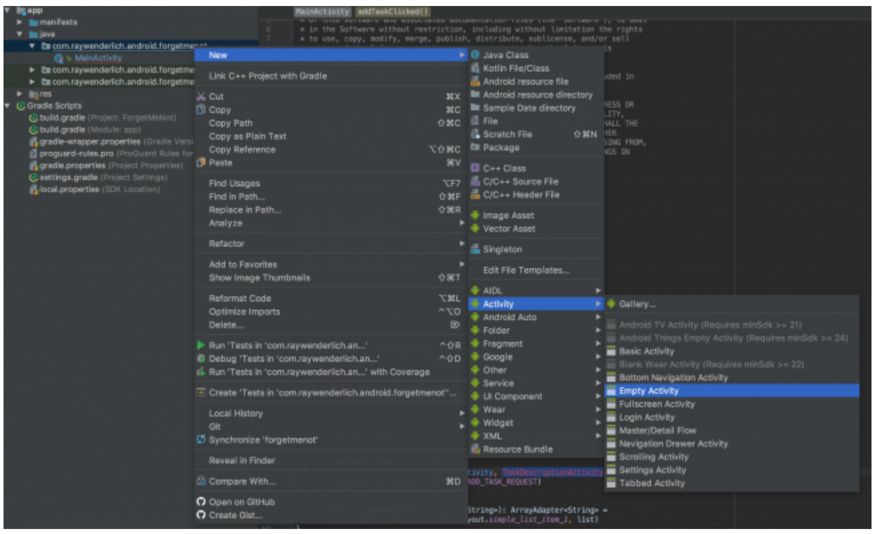
You can start an activity with either `startActivity()` or `startActivityForResult()`. They are similar except that `startActivityForResult()` will result in `onActivityResult()` being called once the `TaskDescriptionActivity` finishes. You’ll implement this callback later so you can know if there is a new task to add to your list or not.

But first, you need a way to enter new tasks in Forget Me Not — you’ll do so by creating the `TaskDescriptionActivity`.

Note: `Intents` are used to start activities and pass data between them. For more information, check out the Android: Intents Tutorial

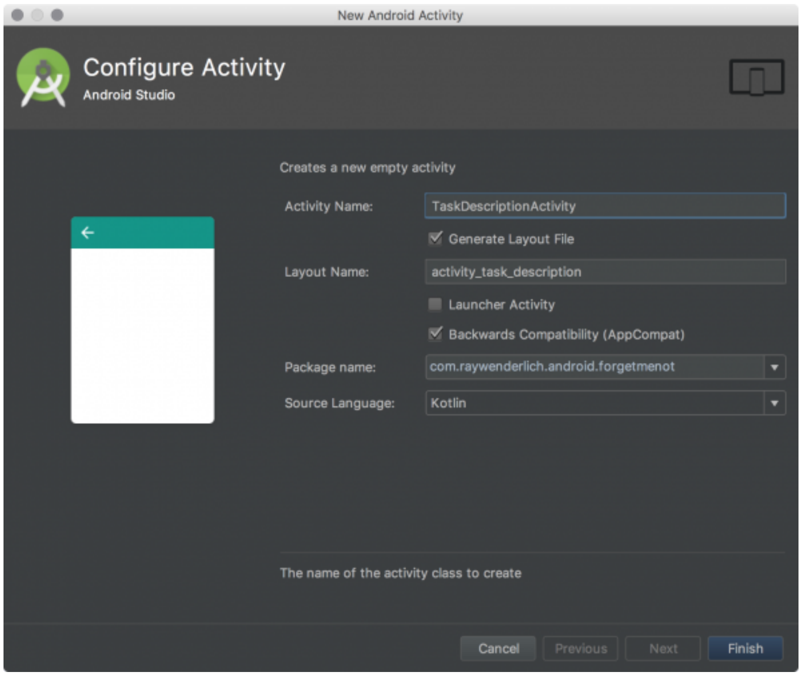
Creating an Activity

Android Studio makes it very easy to create an activity. Just right-click on the package where you want to add the activity — in this case, the package is *com.raywenderlich.android.forgetmenot*. Then navigate to *New\Activity*, and choose *Empty Activity*, which is a basic template for an activity:



On the next screen, enter *TaskDescriptionActivity* as the *Activity Name* and Android Studio will automatically fill the other fields based on that.

Click *Finish* and put your hands in the air to celebrate. You’ve just created your first activity!



Creates a new empty activity

Activity Name: TaskDescriptionActivity

☑ Generate Layout File

Layout Name: activity_task_description

☐ Launcher Activity

☑ Backwards Compatibility (AppCompat)

Package name: com.raywenderlich.android.forgetmenot

Source Language: Kotlin

The name of the activity class to create

Android Studio will automatically generate the corresponding resources needed to create the activity. These are:

- *Class*: The class file is named *TaskDescriptionActivity.kt*. This is where you implement the activity's behavior. This class must subclass the Activity class or an existing subclass of it, such as AppCompatActivity.
- *Layout*: The layout file is located under `res/layout/` and named *activity_task_description.xml*. It defines the placement of different UI elements on the screen when the activity is created.

The layout file created from the *Empty Activity* template in Android Studio 3.0 and above defaults to using a *ConstraintLayout* for the root view group. For more information on ConstraintLayout, please see the android developer docs here.

In addition to this, you will see a new addition to your app's *AndroidManifest.xml* file:

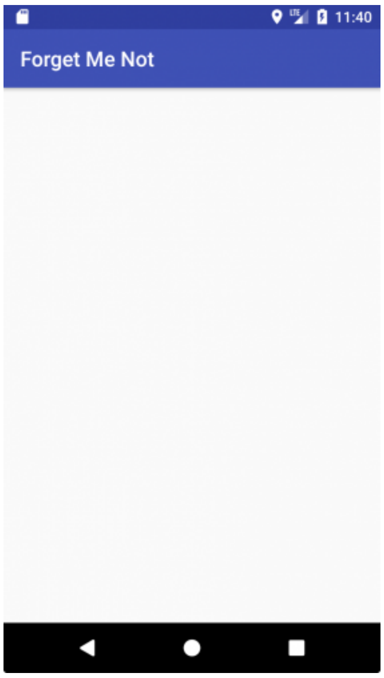
```
<activity android:name=".TaskDescriptionActivity"></activity>
```

The `activity` element declares the activity. Android apps have a strong sense of order, so all available activities must be declared in the manifest to ensure the app only has control of activities declared here. You don't want your app to accidentally use the wrong activity, or even worse, have it use activities that are used by other apps without explicit permission.

There are several attributes that you can include in this element to define properties for the activity, such as a label or icon, or a theme to style the activity's UI.

`android:name` is the only required attribute. It specifies the activity's class name relative to the app package (hence the period at the beginning).

Now build and run the app. When you tap on *ADD A TASK*, you're presented with your newly generated activity!



Looking good, except for the fact that it's lacking substance. Now for a quick remedy to that!

In your newly generated *TaskDescriptionActivity*, paste the following into your class file, overwriting anything else except the class declaration and its brackets.

```
// 1
companion object {
  val EXTRA_TASK_DESCRIPTION = "task"
}

// 2
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_task_description)
}

// 3
fun doneClicked(view: View) {

}
```

Note: You can quickly fix any missing imports that Android Studio complains about by placing your cursor on the item and pressing option+return on Mac or Alt+Enter on PC.

Here, you've accomplished the following:

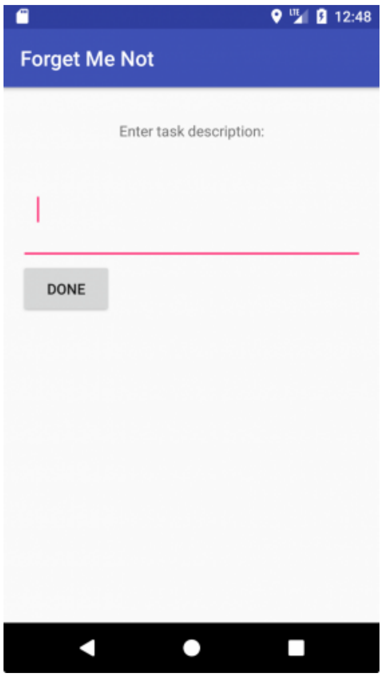
1. Used the Kotlin companion object for the class to define attributes common across the class, similar to *static* members in Java.
2. Overriden the `onCreate()` lifecycle method to set the content view for the activity from the layout file.
3. Added an empty click handler that will be used to finish the activity.

Jump over to your associated layout in *res/layout/activity_task_description.xml* and replace everything with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="@dimen/default_padding"
  tools:context="com.raywenderlich.android.forgetmenot.TaskDescriptionActivity">

  <TextView
    android:id="@+id/descriptionLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/default_padding"
    android:text="@string/description"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <EditText
    android:id="@+id/descriptionText"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:inputType="textMultiLine"
    android:padding="@dimen/default_padding"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/descriptionLabel" />

  <Button
    android:id="@+id/doneButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="doneClicked"
    android:padding="@dimen/default_padding"
    android:text="@string/done"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/descriptionText" />

</android.support.constraint.ConstraintLayout>
```

Here you change the layout so there is a `TextView` prompting for a task description, an `EditText` for the user to input the description, and a Done button to save the new task.

Run the app again, tap *ADD A TASK* and your new screen will look a lot more interesting.



Stopping an Activity

Just as important as starting an activity with all the right methods is properly stopping it.

In *TaskDescriptionActivity.kt*, add these imports, including one for Kotlin Android Extensions:

```
import android.app.Activity
import android.content.Intent
import kotlinx.android.synthetic.main.activity_task_description.*
```

Then add the following to `doneClicked()`, which is called when the *Done* button is tapped in this activity:

```
// 1
val taskDescription = descriptionText.text.toString()

if (!taskDescription.isEmpty()) {
  // 2
  val result = Intent()
  result.putExtra(EXTRA_TASK_DESCRIPTION, taskDescription)
  setResult(Activity.RESULT_OK, result)
} else {
  // 3
  setResult(Activity.RESULT_CANCELED)
}

// 4
finish()
```

You can see a few things happening here:

1. You retrieve the task description from the `descriptionText` `EditText`, where Kotlin Android Extensions has again been used to get references to view fields.
2. You create a result `Intent` to pass back to `MainActivity` if the task description retrieved in step one is not empty. Then you bundle the task description with the intent and set the activity result to `RESULT_OK`, indicating that the user successfully entered a task.
3. If the user has not entered a task description, you set the activity result to `RESULT_CANCELED`.
4. Here you close the activity.

Once you call `finish()` in step four, the callback `onActivityResult()` will be called in `MainActivity` — in turn, you need to add the task to the to-do list.

Add the following method to *MainActivity.kt*, right after `onCreate()`:

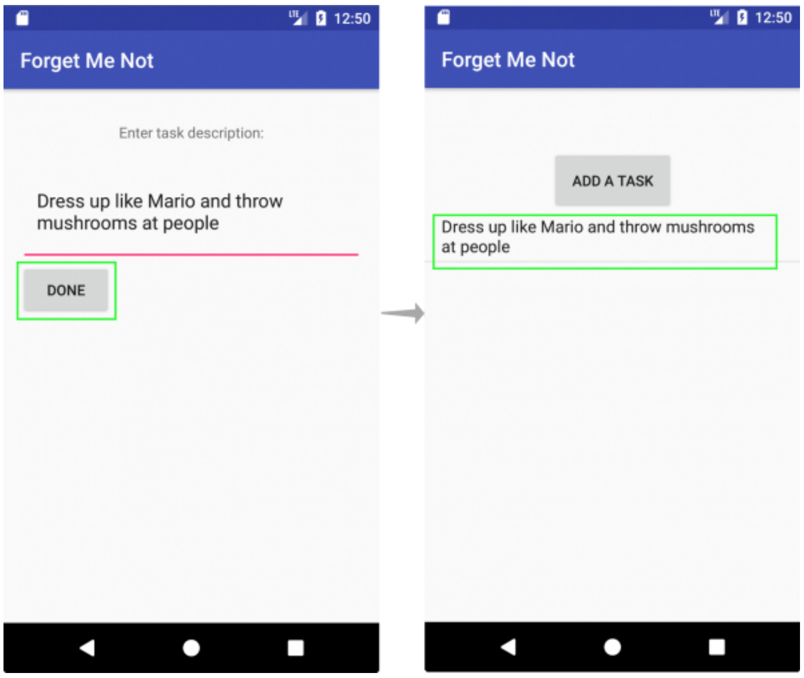
```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  // 1
  if (requestCode == ADD_TASK_REQUEST) {
    // 2
    if (resultCode == Activity.RESULT_OK) {
      // 3
      val task = data?.getStringExtra(TaskDescriptionActivity.EXTRA_TASK_DESCRIPTION)
      task?.let {
        taskList.add(task)
        // 4
        adapter.notifyDataSetChanged()
      }
    }
  }
}
```

Let’s take this step-by-step:

1. You check the `requestCode` to ensure the activity result is indeed for your add task request you started with `TaskDescriptionActivity`.
2. You make sure the `resultCode` is `RESULT_OK` — the standard activity result for a successful operation.
3. Here you extract the task description from the result intent and, after a null check with the `let` function, add it to your list.

4. Finally, you call `notifyDataSetChanged()` on your list adapter. In turn, it notifies the `ListView` about changes in your data model so it can trigger a refresh of its view.

Build and run the project to see it in action. After the app starts, tap *ADD A TASK*. This will bring up a new screen that lets you enter a task. Now add a description and tap *Done*. The screen will close and the new task will be on your list:


12:50
Forget Me Not
Enter task description:
Dress up like Mario and throw mushrooms at people
DONE
12:50
Forget Me Not
ADD A TASK
Dress up like Mario and throw mushrooms at people

Registering Broadcast Receivers

Every to-do list needs to have a good grasp on date and time, so a time display should be the next thing you add to your app. Open *MainActivity.kt* and add the following after the existing property declarations at the top:

```
private val tickReceiver by lazy { makeBroadcastReceiver() }
```

Then add a *companion object* near the top of `MainActivity`:

```
companion object {
  private const val LOG_TAG = "MainActivityLog"

  private fun getCurrentTimeStamp(): String {
    val simpleDateFormat = SimpleDateFormat("yyyy-MM-dd HH:mm", Locale.US)
    val now = Date()
    return simpleDateFormat.format(now)
  }
}
```

And initialize the `tickReceiver` by adding the following to the bottom of `MainActivity`:

```
private fun makeBroadcastReceiver(): BroadcastReceiver {
  return object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent?) {
      if (intent?.action == Intent.ACTION_TIME_TICK) {
        dateTimeTextView.text = getCurrentTimeStamp()
      }
    }
  }
}
```

Here, you create a *BroadcastReceiver* that sets the date and time on the screen if it receives a time change broadcast from the system. You use `getCurrentTimeStamp()`, which is a utility method in your activity, to format the current date and time.

Note: If you're not familiar with `BroadcastReceivers`, you should refer to the Android Developer documentation.

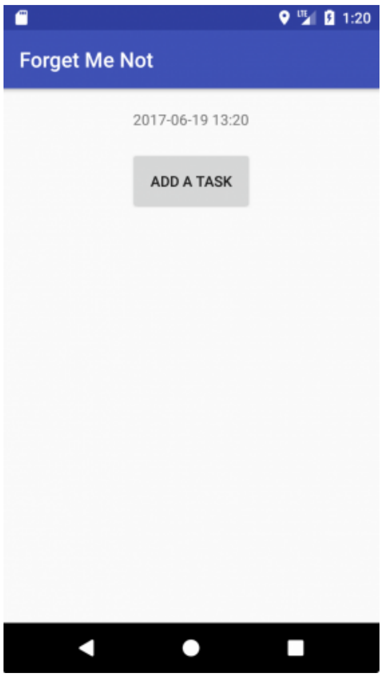
Next add the following methods to `MainActivity` underneath `onCreate()`

```
override fun onResume() {
  // 1
  super.onResume()
  // 2
  dateTimeTextView.text = getCurrentTimeStamp()
  // 3
  registerReceiver(tickReceiver, IntentFilter(Intent.ACTION_TIME_TICK))
}

override fun onPause() {
  // 4
  super.onPause()
  // 5
  try {
    unregisterReceiver(tickReceiver)
  } catch (e: IllegalArgumentException) {
    Log.e(MainActivity.LOG_TAG, "Time tick Receiver not registered", e)
  }
}
```

Here you do a few things:

1. You call `onResume()` on the superclass.
2. You update the date and time `TextView` with the current time stamp, because the broadcast receiver is not currently registered.
3. You then register the broadcast receiver in `onResume()`. This ensures it will receive the broadcasts for ACTION_TIME_TICK. These are sent every minute after the time changes.
4. In `onPause()`, you first call `onPause()` on the superclass.
5. You then unregister the broadcast receiver in `onPause()`, so the activity no longer receives the time change broadcasts while paused. This cuts down unnecessary system overhead.

Build and run the app. Now you should now see the current date and time at the top of the screen. Even if you navigate to the add task screen and come back, the time still gets updated.



Persisting State

Every to-do list is good at remembering what you need to do, except for your friend Forget Me Not. Unfortunately, the app is quite forgetful at the moment. See it for yourself.

Open the app and follow these steps.

1. Tap *ADD A TASK*.
2. Enter “Replace regular with decaf in the breakroom” as the task description and tap *Done*. You’ll see your new task in the list.
3. Close the app from the recent apps.
4. Open the app again.

You can see that it forgot about your evil plans.

Persisting Data Between Launches

Open *MainActivity.kt*, and add the following properties to the top of the class:

```
private val PREFS_TASKS = "prefs_tasks"
private val KEY_TASKS_LIST = "tasks_list"
```

And add the following underneath the rest of your activity lifecycle methods.

```
override fun onStop() {
  super.onStop()

  // Save all data which you want to persist.
  val savedList = StringBuilder()
  for (task in taskList) {
    savedList.append(task)
    savedList.append(",")
  }

  getSharedPreferences(PREFS_TASKS, Context.MODE_PRIVATE).edit()
      .putString(KEY_TASKS_LIST, savedList.toString()).apply()
}
```

Here you build a comma separated string with all the task descriptions in your list, and then you save the string to SharedPreferences in the `onStop()` callback. As mentioned earlier, `onStop()` is a good place to save data that you want to persist across app uses.

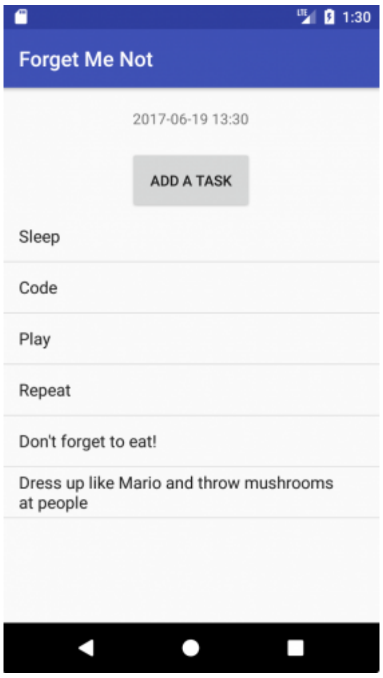
Next add the following to `onCreate()` below the existing initialization code:

```
val savedList = getSharedPreferences(PREFS_TASKS, Context.MODE_PRIVATE).getString(KEY_TASKS_LIST, null)
if (savedList != null) {
  val items = savedList.split(",".toRegex()).dropLastWhile { it.isEmpty() }.toTypedArray()
  taskList.addAll(items)
}
```

Here you read the saved list from the `SharedPreferences` and initialize `taskList` by converting the retrieved comma separated string to a typed array.

Note: You used `SharedPreferences` since you were only saving primitive data types. For more complex data you can use a variety of storage options available on Android.

Now, build and run the app. Add a task, close and reopen the app. Do you see a difference? You are now able to retain tasks in your list!



Configuration Changes

You need the ability to delete entries from Forget Me Not.

Still in *MainActivity.kt*, at the bottom of the class add:

```
private fun taskSelected(position: Int) {
  // 1
  AlertDialog.Builder(this)
    // 2
    .setTitle(R.string.alert_title)
    // 3
    .setMessage(taskList[position])
    .setPositiveButton(R.string.delete, { _, _ ->
      taskList.removeAt(position)
      adapter.notifyDataSetChanged()
    })
    .setNegativeButton(R.string.cancel, {
      dialog, _ -> dialog.cancel()
    })
    // 4
    .create()
    // 5
    .show()
}
```

In a nutshell, you're creating and showing an alert dialog when you select a task from the list. Here is the step-by-step explanation:

1. You create an `AlertDialog.Builder` which facilitates the creation of an `AlertDialog`.
2. You set the alert dialog title.
3. You set the alert dialog message to be the description of the selected task. Then you also implement the `PositiveButton` to remove the item from the list and refresh it, and the `NegativeButton` to dismiss the dialog.
4. You create the alert dialog.
5. You display the alert dialog to the user.

Update the `OnItemClickListener` of the `taskListView` in `onCreate()`:

```
taskListView.onItemClickListener = AdapterView.OnItemClickListener { _, _, position, _ ->
  taskSelected(position)
}
```

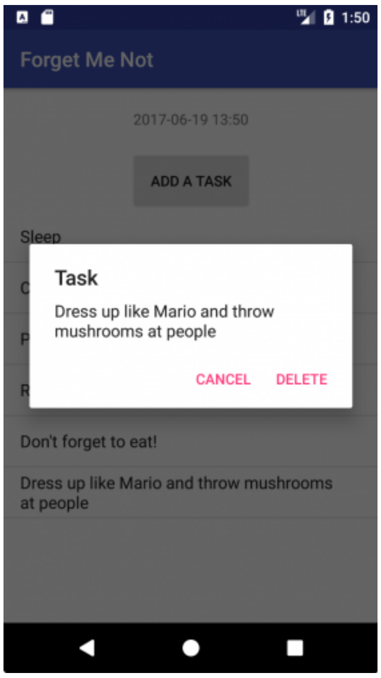
Your app won't compile though until you define some strings. It's good practice to keep text you want in your app separate from the code. The reason is so that you can easily change it, which is especially useful for text that you use in multiple places. It's also handy for those times you need to translate your app into another language.

To configure the strings, open *res/values/strings.xml* and within the `resources` element add:

```
<string name="alert_title">Task</string>
<string name="delete">Delete</string>
```

```
<string name="cancel">Cancel</string>
```

Build and run the app. Tap on one of the tasks. You'll see an alert dialog with options to *CANCEL* or *DELETE* the task from the list:


1:50
Forget Me Not
2017-06-19 13:50
ADD A TASK
Sleep
Task
Dress up like Mario and throw mushrooms at people
CANCEL
DELETE
Don't forget to eat!
Dress up like Mario and throw mushrooms at people

Now try rotating the device. (Make sure you have rotation in the device settings set to auto-rotate.)

As soon as you rotate the device, the alert dialog is dismissed. This makes for an unreliable, undesirable user experience — users don't like it when things just vanish from their screen without reason.

Handling Configuration Changes

Configuration changes, such as rotation, keyboard visibility and so on, cause an activity to shut down and restart. You can find the full list of system events that cause an activity to be recreated here.

There are a couple of ways you can handle a configuration change.

One way is as follows. In *AndroidManifest.xml*, find the start tag:

```
<activity android:name=".MainActivity">
```

And change it to:

```
<activity android:name=".MainActivity" android:configChanges="orientation|screenSize">
```

Here, you declare that your `MainActivity` will handle any configuration changes that arise from a change in orientation or screen size. This simple line prevents a restart of your activity by the system, and it passes the work to `MainActivity`.

You can then handle these configuration changes by implementing `onConfigurationChanged()`. In *MainActivity.kt*, add the following method after `onStop()`:

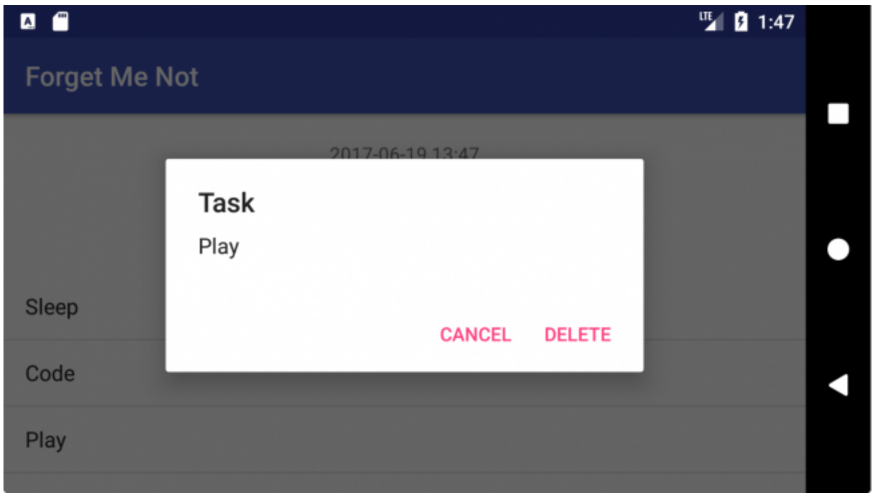
```
override fun onConfigurationChanged(newConfig: Configuration?) {
  super.onConfigurationChanged(newConfig)
}
```

Here you're just calling the superclass's `onConfigurationChanged()` method since you're not updating or resetting any elements based on screen rotation or size.

`onConfigurationChanged()` is passed a `Configuration` object that contains the updated device configuration.

By reading fields in this `newConfig`, you can determine the new configuration and make appropriate changes to update the resources used in your interface.

Now, build and run the app and rotate the device again. This time, the dialog stays in place until you dismiss it.



An alternative way to handle configuration changes is to retain a stateful object that's carried forward to the recreated instance of your activity. You can accomplish this by implementing the `onSaveInstanceState()` callback.

When you do this, the system saves your activity's state in a `Bundle`, and you can restore it when you implement the corresponding `onRestoreInstanceState()` callback. However, the `Bundle` is not designed to hold large data sets such as bitmaps, and it can only store data that is serializable.

The downside of the stateful object solution is that serializing and deserializing the data during a configuration change can come at a high cost. It can consume a lot of memory and slow down the activity restart process.

In such instances, retaining a `Fragment` is currently the most preferable way to handle a configuration change. You can learn more about fragments and how to use them to retain information when your activity is restarted in our Android Fragments Tutorial.

Where To Go From Here?

Congratulations! You have just learned the basics of using activities in Android and now have an excellent understanding of the ever-important activity lifecycle.

You covered quite a few concepts, including:

- How to create an activity
- How to stop an activity
- How to persist data when an activity stops
- How to work around a configuration change

You can download the completed project using the *Download materials* button at the top or bottom of the tutorial. If you're still hungry for more, check out Google's documentation.

I hope you enjoyed this introduction to Android activities tutorial, and if you have any questions, comments, or awesome modifications to this project app please join the forum discussion and comment below!

Core Concepts Android & Kotlin Tutorials