Handling Runtime Changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android restarts the running Activity (`onDestroy()` is called, followed by `onCreate()`). The restart behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources.

To properly handle a restart, it is important that your Activity restores its previous state through the normal Activity lifecycle, in which Android calls `onSaveInstanceState()` before it destroys your Activity so that you can save data about the application state. You can then restore the state during `onCreate()` or `onRestoreInstanceState()`. To test that your application restarts itself with the application state intact, you should invoke configuration changes (such as changing the screen orientation) while performing various tasks in your application.

Your application should be able to restart at any time without loss of user data or state in order to handle events such as when the user receives an incoming phone call and then returns to your application (read about the Activity lifecycle).

However, you might encounter a situation in which restarting your application and restoring significant amounts of data can be costly and create a poor user experience. In such a situation, you have two options:

a. Retain an object during a configuration change

   Allow your Activity to restart when a configuration changes, but carry a stateful `Object` to the new instance of your Activity.

b. Handle the configuration change yourself

   Prevent the system from restarting your Activity during certain configuration changes and receive a callback when the configurations do change, so that you can manually update your Activity as necessary.

Retaining an Object During a Configuration Change

If restarting your Activity requires that you recover large sets of data, re-establish a network connection, or perform other intensive operations, then a full restart due to a configuration change might be an unpleasant user experience. Also, it may not be possible for you to completely maintain your Activity state with the `Bundle` that the system saves for you during the Activity lifecycle—it is not designed to carry large objects (such as bitmaps) and the data within it must be serialized then deserialized, which can consume a lot of memory and make the configuration change slow. In such a situation, you can alleviate the burden of reinitializing your Activity by retaining a stateful Object when your Activity is restarted due to a configuration change.

To retain an Object during a runtime configuration change:

1. Override the `onRetainNonConfigurationInstance()` method to return the Object you would like to retain.
2. When your Activity is created again, call `getLastNonConfigurationInstance()` to recover your Object.

Android calls `onRetainNonConfigurationInstance()` between `onStop()` and `onDestroy()` when it shuts down your Activity due to a configuration change. In your implementation of `onRetainNonConfigurationInstance()`, you can return any `Object` that you need in order to efficiently restore your state after the configuration change.

A scenario in which this can be valuable is if your application loads a lot of data from the web. If the user changes the orientation of the device and the Activity restarts, your application must re-fetch the data, which could be slow. What you can do instead is implement `onRetainNonConfigurationInstance()` to return an object carrying your data and then retrieve the data when your Activity starts again with `getLastNonConfigurationInstance()`. For example:

```
@Override
public Object onRetainNonConfigurationInstance() {
    final MyDataObject data = collectMyLoadedData();
    return data;
}
```

> **Caution:** While you can return any object, you should never pass an object that is tied to the `Activity`, such as a `Drawable`, an `Adapter`, a `View` or any other object that's associated with a `Context`. If you do, it will leak all the Views and resources of the original Activity instance. (To leak the resources means that your application maintains a hold on them and they cannot be garbage-collected, so lots of memory can be lost.)

Then retrieve the `data` when your Activity starts again:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    final MyDataObject data = (MyDataObject) getLastNonConfigurationInstance();
    if (data == null) {
        data = loadMyData();
    }
    ...
}
```

In this case, `getLastNonConfigurationInstance()` retrieves the data saved by `onRetainNonConfigurationInstance()`. If `data` is null (which happens when the Activity starts due to any reason other than a configuration change) then the data object is loaded from the original source.

Handling the Configuration Change Yourself

If your application doesn't need to update resources during a specific configuration change *and* you have a performance limitation that requires you to avoid the Activity restart, then you can declare that your Activity handles the configuration change itself, which prevents the system from restarting your Activity.

> **Note:** Handling the configuration change yourself can make it much more difficult to use alternative resources, because the system does not automatically apply them for you. This technique should be considered a last resort and is not recommended for most applications.

To declare that your Activity handles a configuration change, edit the appropriate `<activity>` element in your manifest file to include the `android:configChanges` attribute with a string value that represents the configuration that you want to handle. Possible values are listed in the documentation for the `android:configChanges` attribute (the most commonly used values are `orientation` to handle when the screen orientation changes and `keyboardHidden` to handle when the keyboard availability changes). You can declare multiple configuration values in the attribute by separating them with a pipe character ("|").

For example, the following manifest snippet declares an Activity that handles both the screen orientation change and keyboard availability change:

```
<activity android:name=".MyActivity"
          android:configChanges="orientation|keyboardHidden"
          android:label="@string/app_name">
```

Now when one of these configurations change, `MyActivity` is not restarted. Instead, the Activity receives a call to `onConfigurationChanged()`. This method is passed a `Configuration` object that specifies the new device configuration. By reading fields in the `Configuration`, you can determine the new configuration and make appropriate changes by updating the resources used in your interface. At the time this method is called, your Activity's `Resources` object is updated to return resources based on the new configuration, so you can easily reset elements of your UI without the system restarting your Activity.

For example, the following `onConfigurationChanged()` implementation checks the availability of a hardware keyboard and the current device orientation:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    // Checks the orientation of the screen
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        Toast.makeText(this, "landscape", Toast.LENGTH_SHORT).show();
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT){
        Toast.makeText(this, "portrait", Toast.LENGTH_SHORT).show();
    }
    // Checks whether a hardware keyboard is available
    if (newConfig.hardKeyboardHidden == Configuration.HARDKEYBOARDHIDDEN_NO) {
        Toast.makeText(this, "keyboard visible", Toast.LENGTH_SHORT).show();
    } else if (newConfig.hardKeyboardHidden ==
Configuration.HARDKEYBOARDHIDDEN_YES) {
        Toast.makeText(this, "keyboard hidden", Toast.LENGTH_SHORT).show();
    }
}
```

The `Configuration` object represents all of the current configurations, not just the ones that have changed. Most of the time, you won't care exactly how the configuration has changed and can simply re-assign all your resources that provide alternatives to the configuration that you're handling. For example, because the `Resources` object is now updated, you can reset any `ImageView`s with `setImageResource(int)` and the appropriate resource for the new configuration is used (as described in Providing Resources).

Notice that the values from the `Configuration` fields are integers that are matched to specific constants from the `Configuration` class. For documentation about which constants to use with each field, refer to the appropriate field in the `Configuration` reference.

> **Remember:** When you declare your Activity to handle a configuration change, you are responsible for resetting any elements for which you provide alternatives. If you declare your Activity to handle the orientation change and have images that should change between landscape and portrait, you must re-assign each resource to each element during `onConfigurationChanged()`.

If you don't need to update your application based on these configuration changes, you can instead *not* implement `onConfigurationChanged()`. In which case, all of the resources used before the configuration change are still used and you've only avoided the restart of your Activity. However, your application should always be able to shutdown and restart with its previous state intact. Not only because there are other configuration changes that you cannot prevent from restarting your application but also in order to handle events such as when the user receives an incoming phone call and then returns to your application.

For more about which configuration changes you can handle in your Activity, see the `android:configChanges` documentation and the `Configuration` class.

← Back to Application Resources ↑ Go to top