Q) When does the multi-touch gesture happen? A) It happens when more than one pointer (finger) touches the screen.

Q) How to keep track of each pointer within a gesture? A) You have to use the pointer's index and ID.

Q) What is the purpose of having an Index and ID for each pointer? A) Each pointer gets a unique ID during the gesture's lifetime and it is used to track the pointers within the gesture. This ID is generated once the pointer touches the screen and joins the gesture. Now, the MotionEvent object saves all the pointers' data in a special array and uses indices to access the pointers' entries. These entries might shift up (change) if a pointer leaves the screen (the gesture) and this will lead to changes in pointers' indices.
Reference: https://developer.android.com/training/gestures/multi
Android system generates touch events every time multiple pointers touches the screen at the same time: (Click [HERE]() for more details)

Event	
ACTION_DOWN	The first pointer (finger) touches the screen. The
ACTION_POINTER_UP	A non-primary (secondary) pointer leaves the scre
ACTION_POINTER_DOWN	A non-primary (secondary) pointer touches the sc
ACTION_MOVE	A motion happened to primary or non-primary po
ACTION_UP	The last pointer leaves the screen

Android system provides several methods (MotionEvent methods) to deal with multi-touch events, such as:

METHOD	
getPointerCount()	The current number of pointers (fingers) on the scre
getPointerId(int pointerIndex)	get the pointer id associated with a particular pointe
findPointerIndex(int pointerId)	find the pointer index for the given id
getX(int pointerIndex)	find the x coordinate for the given pointer index
getY(int pointerIndex)	find the y coordinate for the given pointer index

Gesture Detectors

The gesture detector classes are used to detect common gestures through a set of motion events. There are three steps required for the gesture detector to work:

1. Create an instance of Gesture Detector class
2. Implements the required methods

3. Intercept the touch events and pass them to the gesture detector

Mainly, there are two classes of detectors GestureDetector and ScaleGestureDetector.

Class	interface	methods	
GestureDetector	OnGestureListener	onDown(MotionEvent e)	Notified when a
		onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)	• Notified • e1 is the • e2 is the • velocity • velocity
		onLongPress(MotionEvent e)	Notified when a
		onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)	• Notified • e1 is the • e2 is the • **distance** • **distance**
		onShowPress(MotionEvent e)	The user has pe
		onSingleTapUp(MotionEvent e)	Notified when a

class	interface	methods	
GestureDetector	OnDoubleTapListener	onDoubleTap(MotionEvent e)	Notified wh
		onDoubleTapEvent(MotionEvent e)	Notified wh
		onSingleTapConfirmed(MotionEvent e)	Notified wh

class	interface	
ScaleGestureDetector	OnScaleGestureListener	onScale(ScaleGestureDet
		onScaleBegin(ScaleGestu
		onScaleEnd(ScaleGesture

Q) Some methods return boolean values, what does that mean? A) If a callback method returns true, it informs the parent that the event has been consumed and ready to accept further events from the current gesture. But, if a callback returns false, it indicates that the event is not consumed and it is not interested in the remainder of the gesture.

Q) What are the differences between onFling() and onScroll() callbacks?
A)   1) OnFling() needs some velocity in the movement (like swipe to unlock the phone). While, onScroll(), is invoked one you move your finger with normal speed (when you scroll a list or drag and drop) 2) onFlicg() will be called only once at the end of the gesture, while onScroll() will be called multiple times as you move your finger on the screen.

Instantiate a Gesture Detector instance

```
private GestureDetectorCompat mDetector;
private ScaleGestureDetector mScaleDetector;

...
...
mDetector = new GestureDetectorCompat(this, this);
mScaleDetector = new ScaleGestureDetector(this, this);
```

Where the first parameter is the context and the second one is a reference to the callbacks object.

Implements the required methods

It is possible to create a separate class to handle the implementation of all the callbacks or simply add them to the activity as shown below:

```
public class MainActivity extends AppCompatActivity implements
GestureDetector.OnGestureListener,
GestureDetector.OnDoubleTapListener {

}

@Override
   public boolean onDown(MotionEvent e) {
       return false;
   }

   @Override
   public void onShowPress(MotionEvent e) {

   }
   @Override
   public boolean onSingleTapUp(MotionEvent e) {
      return false;
   }

   @Override
   public boolean onScroll(MotionEvent e1, MotionEvent e2, float
distanceX, float distanceY) {
      return false;
   }

   @Override
   public void onLongPress(MotionEvent e) {
      return false;

   }

   @Override
   public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY) {
      return false;
   }

   @Override
   public boolean onSingleTapConfirmed(MotionEvent e) {
       return false;
   }

   @Override
   public boolean onDoubleTap(MotionEvent e) {
      return false;
   }

   @Override
   public boolean onDoubleTapEvent(MotionEvent e) {
       return false;
   }


   @Override
   public boolean onScale(ScaleGestureDetector detector) {
       return false;
   }
   @Override
   public boolean onScaleBegin(ScaleGestureDetector detector) {
        return true;
   }
   @Override
   public void onScaleEnd(ScaleGestureDetector detector) {

   }
```

Intercept the touch events and pass them to the gesture detector

For the gesture detectors to work, you must override the onTouch callback method and forward the MotionEvent object to the detectors.
Q) What will happen if your onTouch() callback returns false instead of true? Change it and observe the results.

```
@Override
public boolean onTouch(View v, MotionEvent event) {
       mDetector.onTouchEvent(event));
       mScaleDetector.onTouchEvent(event);

       return true;
    }
```

Now, the activity in one piece:

```
package com.fit2081.week11gesturesdetectors;

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.view.GestureDetectorCompat;

import android.os.Bundle;
import android.view.GestureDetector;
import android.view.MotionEvent;
import android.view.ScaleGestureDetector;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity implements
View.OnTouchListener, GestureDetector.OnGestureListener,
GestureDetector.OnDoubleTapListener,
ScaleGestureDetector.OnScaleGestureListener {

    TextView tV;
    private GestureDetectorCompat mDetector;
    private ScaleGestureDetector mScaleDetector;
```

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

    tV = findViewById(R.id.textview_id);
    mScaleDetector = new ScaleGestureDetector(this, this);

    mDetector = new GestureDetectorCompat(this, this);
    mDetector.setOnDoubleTapListener(this);

    View myLayout = findViewById(R.id.myLayout);
    myLayout.setOnTouchListener(this);


}

@Override
public boolean onTouch(View v, MotionEvent event) {
    mDetector.onTouchEvent(event);
    mScaleDetector.onTouchEvent(event);

    return true;
}

@Override
public boolean onSingleTapConfirmed(MotionEvent e) {
    tV.setText("onSingleTapConfirmed");
    return true;
}

@Override
public boolean onDoubleTap(MotionEvent e) {
    tV.setText("onDoubleTap");
    return true;
}

@Override
public boolean onDoubleTapEvent(MotionEvent e) {
    tV.setText("onDoubleTapEvent");

    return true;
}
```

```
    @Override
    public boolean onDown(MotionEvent e) {
        tV.setText("onDown");

        return true;
    }

    @Override
    public void onShowPress(MotionEvent e) {
        tV.setText("onShowPress");

    }

    @Override
    public boolean onSingleTapUp(MotionEvent e) {
        tV.setText("onSingleTapUp");

        return true;
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float
distanceX, float distanceY) {
        tV.setText("onScroll");

        return true;
    }

    @Override
    public void onLongPress(MotionEvent e) {
        tV.setText("onLongPress");

    }

    @Override
    public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY) {
        tV.setText("onFling");

        return true;
    }
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        tV.setText("onScale");

        return true;
    }

    @Override
    public boolean onScaleBegin(ScaleGestureDetector detector) {
        tV.setText("onScaleBegin");

        return true;
    }

    @Override
    public void onScaleEnd(ScaleGestureDetector detector) {
        tV.setText("onScaleEnd");

    }
}
```

Q) What will happen if onScaleBegin() callback returns false instead of true? A) The two methods onScale() and onScaleEnd() will not be invoked. Q) Why? A) if the callback onScaleBegin() returns false, this means it is not interested in the current gesture.

Q) How can I get the pinch (zoom/scale) size at the end of the scale gesture? A) you can return the event span using **detector.getCurrentSpan()** where detector is the input parameter to onScaleEnd() callback. Click HERE for more information about the ScaleGestureDetector.

Q) What if want to implement only a subset of the callback? in other words, my app requires one or two callbacks only. A) You need to implement a class that extends the convenience classes SimpleOnGestureListener or SimpleOnScaleGestureListener.

Build a class the extends SimpleOnGestureListener

```
private class MyGestureListener extends
GestureDetector.SimpleOnGestureListener {

       @Override
       public boolean onSingleTapConfirmed(MotionEvent e) {
           tV.setText("onSingleTapConfirmed");
```

```
            return true;
        }

        @Override
        public boolean onDoubleTap(MotionEvent e) {
            tV.setText("onDoubleTap");
            return true;
        }

        @Override
        public boolean onDown(MotionEvent e) {
            tV.setText("onDown");

            return true;
        }

        @Override
        public boolean onScroll(MotionEvent e1, MotionEvent e2, float
distanceX, float distanceY) {
            tV.setText("onScroll");

            return true;
        }

        @Override
        public void onLongPress(MotionEvent e) {
            tV.setText("onLongPress");

        }
    }
```

As you can see from the code above, the class MyGestureListener which works as a listener implements a subset of the interface OnGestureListener callbacks.

Build a class the extends SimpleOnScaleGestureListener

```
private class MyScaleListener extends
ScaleGestureDetector.SimpleOnScaleGestureListener{
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        tV.setText("onScale");

        return true;
    }
```

```
        @Override
        public boolean onScaleBegin(ScaleGestureDetector detector) {
            tV.setText("onScaleBegin");

            return true;
        }

    }
```

Again, extending the convenience class SimpleOnScaleGestureListener allows you to implement only the callbacks which your application needs.   Now, let's provide them to the detectors:

```
MyScaleListener MyScaleListener=new MyScaleListener();
mScaleDetector = new ScaleGestureDetector(this, MyScaleListener);

MyGestureListener myGestureListener = new MyGestureListener();
mDetector = new GestureDetectorCompat(this, myGestureListener);
mDetector.setOnDoubleTapListener(myGestureListener);
```

Detect common gestures

A "touch gesture" occurs when a user places one or more fingers on the touch screen, and your application interprets that pattern of touches as a particular gesture. There are correspondingly two phases to gesture detection:

1. Gather data about touch events.
2. Interpret the data to see if it meets the criteria for any of the gestures your app supports.

Refer to the following related resources:

- Input Events API Guide
- Sensors Overview
- Making the View Interactive

Support library classes

The examples in this lesson use the GestureDetectorCompat and MotionEventCompat classes. These classes are in the Support Library. You should use Support Library classes where possible to provide compatibility with devices running Android 1.6 and higher. Note that MotionEventCompat is *not* a replacement for the MotionEvent class. Rather, it provides static utility methods to which you pass your MotionEvent object in order to receive the desired action associated with that event.

Gather data

When a user places one or more fingers on the screen, this triggers the callback onTouchEvent() on the View that received the touch events. For each sequence of touch events (position, pressure, size, addition of another finger, etc.) that is ultimately identified as a gesture, onTouchEvent() is fired several times.

The gesture starts when the user first touches the screen, continues as the system tracks the position of the user's finger(s), and ends by capturing the final event of the user's fingers leaving the screen. Throughout this interaction, the MotionEvent delivered to onTouchEvent() provides the details of every interaction. Your app can use the data provided by the MotionEvent to determine if a gesture it cares about happened.

Capture touch events for an Activity or View

To intercept touch events in an Activity or View, override the onTouchEvent() callback.

The following snippet uses getActionMasked() to extract the action the user performed from the `event` parameter. This gives you the raw data you need to determine if a gesture you care about occurred:

KotlinJava

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same approach if
// you were subclassing a View.
@Override
public boolean onTouchEvent(MotionEvent event){

    int action = MotionEventCompat.getActionMasked(event);
    switch(action) {
        case (MotionEvent.ACTION_DOWN) :
            Log.d(DEBUG_TAG,"Action was DOWN");
            return true;
        case (MotionEvent.ACTION_MOVE) :
            Log.d(DEBUG_TAG,"Action was MOVE");
            return true;
        case (MotionEvent.ACTION_UP) :
            Log.d(DEBUG_TAG,"Action was UP");
            return true;
        case (MotionEvent.ACTION_CANCEL) :
            Log.d(DEBUG_TAG,"Action was CANCEL");
            return true;
        case (MotionEvent.ACTION_OUTSIDE) :
            Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
            return true;
        default :
            return super.onTouchEvent(event);
    }
}
```

You can then do your own processing on these events to determine if a gesture occurred. This is the kind of processing you would have to do for a custom gesture. However, if your app uses common gestures such as double tap, long press, fling, and so on, you can take advantage of the GestureDetector class. GestureDetector makes it easy for you to detect common gestures without processing the individual touch events yourself. This is discussed below in Detect Gestures.

Capture touch events for a single view

As an alternative to onTouchEvent(), you can attach an View.OnTouchListener object to any View object using the setOnTouchListener() method. This makes it possible to to listen for touch events without subclassing an existing View. For example:

KotlinJava

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

Beware of creating a listener that returns false for the ACTION_DOWN event. If you do this, the listener will not be called for the subsequent ACTION_MOVE and ACTION_UP string of events. This is because ACTION_DOWN is the starting point for all touch events.

If you are creating a custom View, you can override onTouchEvent(), as described above.

Detect gestures

Android provides the GestureDetector class for detecting common gestures. Some of the gestures it supports include onDown(), onLongPress(), onFling(), and so on. You can use GestureDetector in conjunction with the onTouchEvent() method described above.

Detect all supported gestures

When you instantiate a GestureDetectorCompat object, one of the parameters it takes is a class that implements the GestureDetector.OnGestureListener interface. GestureDetector.OnGestureListener notifies users when a particular touch event has occurred. To make it possible for your GestureDetector object to receive events, you override the View or Activity's onTouchEvent() method, and pass along all observed events to the detector instance.

In the following snippet, a return value of true from the individual on*<TouchEvent>* methods indicates that you have handled the touch event. A return value of false passes events down through the view stack until the touch has been successfully handled.

Run the following snippet to get a feel for how actions are triggered when you interact with the touch screen, and what the contents of the MotionEvent are for each touch event. You will realize how much data is being generated for even simple interactions.

KotlinJava

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;
```

```
// Called when the activity is first created.
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // Instantiate the gesture detector with the
    // application context and an implementation of
    // GestureDetector.OnGestureListener
    mDetector = new GestureDetectorCompat(this,this);
    // Set the gesture detector as the double tap
    // listener.
    mDetector.setOnDoubleTapListener(this);
}

@Override
public boolean onTouchEvent(MotionEvent event){
    if (this.mDetector.onTouchEvent(event)) {
        return true;
    }
    return super.onTouchEvent(event);
}

@Override
public boolean onDown(MotionEvent event) {
    Log.d(DEBUG_TAG,"onDown: " + event.toString());
    return true;
}

@Override
public boolean onFling(MotionEvent event1, MotionEvent event2,
        float velocityX, float velocityY) {
    Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
    return true;
}

@Override
public void onLongPress(MotionEvent event) {
    Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
}

@Override
public boolean onScroll(MotionEvent event1, MotionEvent event2, float distanceX,
        float distanceY) {
```

```
        Log.d(DEBUG_TAG, "onScroll: " + event1.toString() + event2.toString());
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapUp: " + event.toString());
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTap: " + event.toString());
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTapEvent: " + event.toString());
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapConfirmed: " + event.toString());
        return true;
    }
}
```

Detect a subset of supported gestures

If you only want to process a few gestures, you can extend GestureDetector.SimpleOnGestureListener instead of implementing the GestureDetector.OnGestureListener interface.

GestureDetector.SimpleOnGestureListener provides an implementation for all of the on<*TouchEvent*> methods by returning false for all of them. Thus you can override only the methods you care about. For example, the snippet below creates a class that extends GestureDetector.SimpleOnGestureListener and overrides onFling() and onDown().

Whether or not you use GestureDetector.OnGestureListener, it's best practice to implement an onDown() method that returns true. This is because all gestures begin with an onDown() message. If you return false from onDown(), as GestureDetector.SimpleOnGestureListener does by default, the system assumes that you want to ignore the rest of the gesture, and the other methods of GestureDetector.OnGestureListener never get called. This has the potential to cause unexpected problems in your app. The only time you should return false from onDown() is if you truly want to ignore an entire gesture.

KotlinJava

```
public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG,"onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent event2,
                float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
            return true;
        }
    }
}
```

Handle multi-touch gestures

A multi-touch gesture is when multiple pointers (fingers) touch the screen at the same time. This lesson describes how to detect gestures that involve multiple pointers.

Refer to the following related resources:

- Input Events API Guide
- Sensors Overview
- Making the View Interactive

Track multiple pointers

When multiple pointers touch the screen at the same time, the system generates the following touch events:

- ACTION_DOWN—For the first pointer that touches the screen. This starts the gesture. The pointer data for this pointer is always at index 0 in the MotionEvent.
- ACTION_POINTER_DOWN—For extra pointers that enter the screen beyond the first. The pointer data for this pointer is at the index returned by getActionIndex().
- ACTION_MOVE—A change has happened during a press gesture.
- ACTION_POINTER_UP—Sent when a non-primary pointer goes up.
- ACTION_UP—Sent when the last pointer leaves the screen.

You keep track of individual pointers within a MotionEvent via each pointer's index and ID:

- **Index**: A MotionEvent effectively stores information about each pointer in an array. The index of a pointer is its position within this array. Most of the MotionEvent methods you use to interact with pointers take the pointer index as a parameter, not the pointer ID.

- **ID**: Each pointer also has an ID mapping that stays persistent across touch events to allow tracking an individual pointer across the entire gesture.

The order in which individual pointers appear within a motion event is undefined. Thus the index of a pointer can change from one event to the next, but the pointer ID of a pointer is guaranteed to remain constant as long as the pointer remains active. Use the getPointerId() method to obtain a pointer's ID to track the pointer across all subsequent motion events in a gesture. Then for successive motion events, use the findPointerIndex() method to obtain the pointer index for a given pointer ID in that motion event. For example:

KotlinJava

```
private int mActivePointerId;

public boolean onTouchEvent(MotionEvent event) {
    ...
    // Get the pointer ID
    mActivePointerId = event.getPointerId(0);

    // ... Many touch events later...

    // Use the pointer ID to find the index of the active pointer
    // and fetch its position
    int pointerIndex = event.findPointerIndex(mActivePointerId);
    // Get the pointer's current position
    float x = event.getX(pointerIndex);
    float y = event.getY(pointerIndex);
    ...
}
```

Get a MotionEvent's action

You should always use the method getActionMasked() (or better yet, the compatibility version MotionEventCompat.getActionMasked()) to retrieve the action of a MotionEvent. Unlike the older getAction() method, getActionMasked() is designed to work with multiple pointers. It returns the masked action being performed, without including the pointer index bits. You can then use getActionIndex() to return the index of the pointer associated with the action. This is illustrated in the snippet below.

Note: This example uses the `MotionEventCompat` class. This class is in the Support Library. You should use `MotionEventCompat` to provide the best support for a wide range of platforms. Note that `MotionEventCompat` is *not* a replacement for the `MotionEvent` class. Rather, it provides static utility methods to which you pass your `MotionEvent` object in order to receive the desired action associated with that event.

KotlinJava

```
int action = MotionEventCompat.getActionMasked(event);
// Get the index of the pointer associated with the action.
int index = MotionEventCompat.getActionIndex(event);
int xPos = -1;
int yPos = -1;

Log.d(DEBUG_TAG,"The action is " + actionToString(action));

if (event.getPointerCount() > 1) {
    Log.d(DEBUG_TAG,"Multitouch event");
    // The coordinates of the current screen contact, relative to
    // the responding View or Activity.
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);

} else {
    // Single touch event
    Log.d(DEBUG_TAG,"Single touch event");
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);
}
...

// Given an action int, returns a string description
public static String actionToString(int action) {
    switch (action) {

        case MotionEvent.ACTION_DOWN: return "Down";
        case MotionEvent.ACTION_MOVE: return "Move";
        case MotionEvent.ACTION_POINTER_DOWN: return "Pointer Down";
        case MotionEvent.ACTION_UP: return "Up";
        case MotionEvent.ACTION_POINTER_UP: return "Pointer Up";
        case MotionEvent.ACTION_OUTSIDE: return "Outside";
        case MotionEvent.ACTION_CANCEL: return "Cancel";
    }
    return "";
}
```

Drag and scale

This lesson describes how to use touch gestures to drag and scale on-screen objects, using onTouchEvent() to intercept touch events.

Refer to the following related resources:

- Input Events API Guide
- Sensors Overview
- Making the View Interactive

Drag an object

If you are targeting Android 3.0 or higher, you can use the built-in drag-and-drop event listeners with `View.OnDragListener`, as described in Drag and Drop.

A common operation for a touch gesture is to use it to drag an object across the screen. The following snippet lets the user drag an on-screen image. Note the following:

- In a drag (or scroll) operation, the app has to keep track of the original pointer (finger), even if additional fingers get placed on the screen. For example, imagine that while dragging the image around, the user places a second finger on the touch screen and lifts the first finger. If your app is just tracking individual pointers, it will regard the second pointer as the default and move the image to that location.
- To prevent this from happening, your app needs to distinguish between the original pointer and any follow-on pointers. To do this, it tracks the ACTION_POINTER_DOWN and ACTION_POINTER_UP events described in Handling Multi-Touch Gestures. ACTION_POINTER_DOWN and ACTION_POINTER_UP are passed to the onTouchEvent() callback whenever a secondary pointer goes down or up.
- In the ACTION_POINTER_UP case, the example extracts this index and ensures that the active pointer ID is not referring to a pointer that is no longer touching the screen. If it is, the app selects a different pointer to be active and saves its current X and Y position. Since this saved position is used in the ACTION_MOVE case to calculate the distance to move the onscreen object, the app will always calculate the distance to move using data from the correct pointer.

The following snippet enables a user to drag an object around on the screen. It records the initial position of the active pointer, calculates the distance the pointer traveled, and moves the object to the new position. It correctly manages the possibility of additional pointers, as described above.

Notice that the snippet uses the getActionMasked() method. You should always use this method (or better yet, the compatibility version MotionEventCompat.getActionMasked()) to retrieve the action of a MotionEvent. Unlike the older getAction() method, getActionMasked() is designed to work with multiple pointers. It returns the masked action being performed, without including the pointer index bits.

KotlinJava

```
// The ‘active pointer’ is the one currently moving our object.
private int mActivePointerId = INVALID_POINTER_ID;

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);

    final int action = MotionEventCompat.getActionMasked(ev);

    switch (action) {
    case MotionEvent.ACTION_DOWN: {
        final int pointerIndex = MotionEventCompat.getActionIndex(ev);
        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);

        // Remember where we started (for dragging)
        mLastTouchX = x;
        mLastTouchY = y;
        // Save the ID of this pointer (for dragging)
        mActivePointerId = MotionEventCompat.getPointerId(ev, 0);
        break;
    }

    case MotionEvent.ACTION_MOVE: {
        // Find the index of the active pointer and fetch its position
        final int pointerIndex =
                MotionEventCompat.findPointerIndex(ev, mActivePointerId);

        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);
        // Calculate the distance moved
        final float dx = x - mLastTouchX;
        final float dy = y - mLastTouchY;

        mPosX += dx;
        mPosY += dy;

        invalidate();

        // Remember this touch position for the next move event
        mLastTouchX = x;
        mLastTouchY = y;

        break;
    }

    case MotionEvent.ACTION_UP: {
        mActivePointerId = INVALID_POINTER_ID;
        break;
    }

    case MotionEvent.ACTION_CANCEL: {
        mActivePointerId = INVALID_POINTER_ID;
        break;
    }

    case MotionEvent.ACTION_POINTER_UP: {

        final int pointerIndex = MotionEventCompat.getActionIndex(ev);
        final int pointerId = MotionEventCompat.getPointerId(ev, pointerIndex);

        if (pointerId == mActivePointerId) {
            // This was our active pointer going up. Choose a new
            // active pointer and adjust accordingly.
            final int newPointerIndex = pointerIndex == 0 ? 1 : 0;
            mLastTouchX = MotionEventCompat.getX(ev, newPointerIndex);
            mLastTouchY = MotionEventCompat.getY(ev, newPointerIndex);
            mActivePointerId = MotionEventCompat.getPointerId(ev, newPointerIndex);
        }
        break;
    }
    }
    return true;
```

```
}
```

Drag to pan

The previous section showed an example of dragging an object around the screen. Another common scenario is *panning*, which is when a user's dragging motion causes scrolling in both the x and y axes. The above snippet directly intercepted the MotionEvent actions to implement dragging. The snippet in this section takes advantage of the platform's built-in support for common gestures. It overrides onScroll() in GestureDetector.SimpleOnGestureListener.

To provide a little more context, onScroll() is called when a user is dragging a finger to pan the content. onScroll() is only called when a finger is down; as soon as the finger is lifted from the screen, the gesture either ends, or a fling gesture is started (if the finger was moving with some speed just before it was lifted). For more discussion of scrolling vs. flinging, see Animating a Scroll Gesture.

Here is the snippet for onScroll():

KotlinJava

```
// The current viewport. This rectangle represents the currently visible
// chart domain and range.
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);

// The current destination rectangle (in pixel coordinates) into which the
// chart data should be drawn.
private Rect mContentRect;

private final GestureDetector.SimpleOnGestureListener mGestureListener
            = new GestureDetector.SimpleOnGestureListener() {
...

@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
            float distanceX, float distanceY) {
    // Scrolling uses math based on the viewport (as opposed to math using pixels).

    // Pixel offset is the offset in screen pixels, while viewport offset is the
    // offset within the current viewport.
    float viewportOffsetX = distanceX * mCurrentViewport.width()
            / mContentRect.width();
    float viewportOffsetY = -distanceY * mCurrentViewport.height()
            / mContentRect.height();
    ...
    // Updates the viewport, refreshes the display.
    setViewportBottomLeft(
            mCurrentViewport.left + viewportOffsetX,
            mCurrentViewport.bottom + viewportOffsetY);
    ...
    return true;
}
```

The implementation of onScroll() scrolls the viewport in response to the touch gesture:

KotlinJava

```
/**
 * Sets the current viewport (defined by mCurrentViewport) to the given
 * X and Y positions. Note that the Y value represents the topmost pixel position,
 * and thus the bottom of the mCurrentViewport rectangle.
 */
private void setViewportBottomLeft(float x, float y) {
    /*
     * Constrains within the scroll range. The scroll range is simply the viewport
     * extremes (AXIS_X_MAX, etc.) minus the viewport size. For example, if the
     * extremes were 0 and 10, and the viewport size was 2, the scroll range would
     * be 0 to 8.
     */

    float curWidth = mCurrentViewport.width();
    float curHeight = mCurrentViewport.height();
    x = Math.max(AXIS_X_MIN, Math.min(x, AXIS_X_MAX - curWidth));
    y = Math.max(AXIS_Y_MIN + curHeight, Math.min(y, AXIS_Y_MAX));

    mCurrentViewport.set(x, y - curHeight, x + curWidth, y);

    // Invalidates the View to update the display.
    ViewCompat.postInvalidateOnAnimation(this);
}
```

Use touch to perform scaling

As discussed in Detecting Common Gestures, GestureDetector helps you detect common gestures used by Android such as scrolling, flinging, and long press. For scaling, Android
provides ScaleGestureDetector. GestureDetector and ScaleGestureDetector can be used together when you want a view to recognize additional gestures.

To report detected gesture events, gesture detectors use listener objects passed to their
constructors. ScaleGestureDetector uses ScaleGestureDetector.OnScaleGestureListener. Android provides ScaleGestureDetector.SimpleOnScaleGestureListener as a helper class that you can extend if you don't care about all of the reported events.

Basic scaling example

Here is a snippet that illustrates the basic ingredients involved in scaling.

KotlinJava

```
private ScaleGestureDetector mScaleDetector;
private float mScaleFactor = 1.f;

public MyCustomView(Context mContext){
    ...
    // View code goes here
    ...
    mScaleDetector = new ScaleGestureDetector(context, new ScaleListener());
}

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);
    return true;
}

@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    canvas.save();
    canvas.scale(mScaleFactor, mScaleFactor);
    ...
    // onDraw() code goes here
    ...
    canvas.restore();
}

private class ScaleListener
        extends ScaleGestureDetector.SimpleOnScaleGestureListener {
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        mScaleFactor *= detector.getScaleFactor();

        // Don't let the object get too small or too large.
        mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

        invalidate();
        return true;
    }
}
```

More complex scaling example

Here is a more complex example from the InteractiveChart sample provided with this class. The InteractiveChart sample supports both scrolling (panning) and scaling with multiple fingers, using the ScaleGestureDetector "span" (getCurrentSpanX/Y) and "focus" (getFocusX/Y) features:

KotlinJava

```
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);
private Rect mContentRect;
private ScaleGestureDetector mScaleGestureDetector;
...
@Override
public boolean onTouchEvent(MotionEvent event) {
    boolean retVal = mScaleGestureDetector.onTouchEvent(event);
    retVal = mGestureDetector.onTouchEvent(event) || retVal;
    return retVal || super.onTouchEvent(event);
}

/**
 * The scale listener, used for handling multi-finger scale gestures.
 */
private final ScaleGestureDetector.OnScaleGestureListener mScaleGestureListener
        = new ScaleGestureDetector.SimpleOnScaleGestureListener() {
    /**
 * This is the active focal point in terms of the viewport. Could be a local
 * variable but kept here to minimize per-frame allocations.
 */
private PointF viewportFocus = new PointF();
private float lastSpanX;
private float lastSpanY;

// Detects that new pointers are going down.
@Override
public boolean onScaleBegin(ScaleGestureDetector scaleGestureDetector) {
    lastSpanX = ScaleGestureDetectorCompat.
            getCurrentSpanX(scaleGestureDetector);
    lastSpanY = ScaleGestureDetectorCompat.
            getCurrentSpanY(scaleGestureDetector);
    return true;
}

@Override
public boolean onScale(ScaleGestureDetector scaleGestureDetector) {

    float spanX = ScaleGestureDetectorCompat.
            getCurrentSpanX(scaleGestureDetector);
    float spanY = ScaleGestureDetectorCompat.
            getCurrentSpanY(scaleGestureDetector);

    float newWidth = lastSpanX / spanX * mCurrentViewport.width();
    float newHeight = lastSpanY / spanY * mCurrentViewport.height();

    float focusX = scaleGestureDetector.getFocusX();
    float focusY = scaleGestureDetector.getFocusY();
    // Makes sure that the chart point is within the chart region.
    // See the sample for the implementation of hitTest().
    hitTest(scaleGestureDetector.getFocusX(),
            scaleGestureDetector.getFocusY(),
            viewportFocus);

    mCurrentViewport.set(
            viewportFocus.x
                    - newWidth * (focusX - mContentRect.left)
                    / mContentRect.width(),
            viewportFocus.y
                    - newHeight * (mContentRect.bottom - focusY)
                    / mContentRect.height(),
            0,
                0);
            mCurrentViewport.right = mCurrentViewport.left + newWidth;
            mCurrentViewport.bottom = mCurrentViewport.top + newHeight;
            ...
            // Invalidates the View to update the display.
            ViewCompat.postInvalidateOnAnimation(InteractiveLineGraphView.this);

            lastSpanX = spanX;
            lastSpanY = spanY;
            return true;
        }
    };
```