

```
1: package com.delta.bartalk.util;
2:
3: import android.app.Activity;
4: import android.os.Build;
5: import android.view.View;
6:
7: /**
8:  * A utility class that helps with showing and hiding system UI such as the
9:  * status bar and navigation/system bar. This class uses backward-compatibility
10:  * techniques described in <a href=
11:  * "http://developer.android.com/training/backward-compatible-ui/index.html">
12:  * Creating Backward-Compatible UIs</a> to ensure that devices running any
13:  * version of ndroid OS are supported. More specifically, there are separate
14:  * implementations of this abstract class: for newer devices,
15:  * {@link #getInstance} will return a {@link SystemUiHiderHoneycomb} instance,
16:  * while on older devices {@link #getInstance} will return a
17:  * {@link SystemUiHiderBase} instance.
18:  * <p>
19:  * For more on system bars, see <a href=
20:  * "http://developer.android.com/design/get-started/ui-overview.html#system-bars"
21:  * > System Bars</a>.
22:  *
23:  * @see android.view.View#setSystemUiVisibility(int)
24:  * @see android.view.WindowManager.LayoutParams#FLAG_FULLSCREEN
25:  */
26: public abstract class SystemUiHider {
27:     /**
28:      * When this flag is set, the
29:      * {@link android.view.WindowManager.LayoutParams#FLAG_LAYOUT_IN_SCREEN}
30:      * flag will be set on older devices, making the status bar "float" on top
31:      * of the activity layout. This is most useful when there are no controls at
32:      * the top of the activity layout.
33:      * <p>
34:      * This flag isn't used on newer devices because the <a
35:      * href="http://developer.android.com/design/patterns/actionbar.html">action
36:      * bar</a>, the most important structural element of an Android app, should
37:      * be visible and not obscured by the system UI.
38:      */
39:     public static final int FLAG_LAYOUT_IN_SCREEN_OLDER_DEVICES = 0x1;
40:
41:     /**
42:      * When this flag is set, {@link #show()} and {@link #hide()} will toggle
43:      * the visibility of the status bar. If there is a navigation bar, show and
44:      * hide will toggle low profile mode.
45:      */
46:     public static final int FLAG_FULLSCREEN = 0x2;
47:
48:     /**
```

```
49:      * When this flag is set, {@link #show()} and {@link #hide()} will toggle
50:      * the visibility of the navigation bar, if it's present on the device and
51:      * the device allows hiding it. In cases where the navigation bar is present
52:      * but cannot be hidden, show and hide will toggle low profile mode.
53:      */
54:      public static final int FLAG_HIDE_NAVIGATION = FLAG_FULLSCREEN | 0x4;
55:
56:      /**
57:       * The activity associated with this UI hider object.
58:       */
59:      protected Activity mActivity;
60:
61:      /**
62:       * The view on which {@link View#setSystemUiVisibility(int)} will be called.
63:       */
64:      protected View mAnchorView;
65:
66:      /**
67:       * The current UI hider flags.
68:       *
69:       * @see #FLAG_FULLSCREEN
70:       * @see #FLAG_HIDE_NAVIGATION
71:       * @see #FLAG_LAYOUT_IN_SCREEN_OLDER_DEVICES
72:       */
73:      protected int mFlags;
74:
75:      /**
76:       * The current visibility callback.
77:       */
78:      protected OnVisibilityChangeListener mOnVisibilityChangeListener = sDummyListener;
79:
80:      /**
81:       * Creates and returns an instance of {@link SystemUiHider} that is
82:       * appropriate for this device. The object will be either a
83:       * {@link SystemUiHiderBase} or {@link SystemUiHiderHoneycomb} depending on
84:       * the device.
85:       *
86:       * @param activity The activity whose window's system UI should be
87:       *                 controlled by this class.
88:       * @param anchorView The view on which
89:       *                  {@link View#setSystemUiVisibility(int)} will be called.
90:       * @param flags Either 0 or any combination of {@link #FLAG_FULLSCREEN},
91:       *             {@link #FLAG_HIDE_NAVIGATION}, and
92:       *             {@link #FLAG_LAYOUT_IN_SCREEN_OLDER_DEVICES}.
93:       */
94:      public static SystemUiHider getInstance(Activity activity, View anchorView, int flags) {
95:          if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
96:              return new SystemUiHiderHoneycomb(activity, anchorView, flags);
```

```
97:         } else {
98:             return new SystemUiHiderBase(activity, anchorView, flags);
99:         }
100:     }
101:
102:     protected SystemUiHider(Activity activity, View anchorView, int flags) {
103:         mActivity = activity;
104:         mAnchorView = anchorView;
105:         mFlags = flags;
106:     }
107:
108:     /**
109:      * Sets up the system UI hider. Should be called from
110:      * {@link Activity#onCreate}.
111:      */
112:     public abstract void setup();
113:
114:     /**
115:      * Returns whether or not the system UI is visible.
116:      */
117:     public abstract boolean isVisible();
118:
119:     /**
120:      * Hide the system UI.
121:      */
122:     public abstract void hide();
123:
124:     /**
125:      * Show the system UI.
126:      */
127:     public abstract void show();
128:
129:     /**
130:      * Toggle the visibility of the system UI.
131:      */
132:     public void toggle() {
133:         if (isVisible()) {
134:             hide();
135:         } else {
136:             show();
137:         }
138:     }
139:
140:     /**
141:      * Registers a callback, to be triggered when the system UI visibility
142:      * changes.
143:      */
144:     public void setOnVisibilityChangeListener(OnVisibilityChangeListener listener) {
```

```
145:         if (listener == null) {
146:             listener = sDummyListener;
147:         }
148:
149:         mOnVisibilityChangeListener = listener;
150:     }
151:
152:     /**
153:      * A dummy no-op callback for use when there is no other listener set.
154:      */
155:     private static OnVisibilityChangeListener sDummyListener = new OnVisibilityChangeListener() {
156:         @Override
157:         public void onVisibilityChange(boolean visible) {
158:         }
159:     };
160:
161:     /**
162:      * A callback interface used to listen for system UI visibility changes.
163:      */
164:     public interface OnVisibilityChangeListener {
165:         /**
166:          * Called when the system UI visibility has changed.
167:          *
168:          * @param visible True if the system UI is visible.
169:          */
170:         public void onVisibilityChange(boolean visible);
171:     }
172: }
```

```
1: package com.delta.bartalk.util;
2:
3: import android.app.Activity;
4: import android.view.View;
5: import android.view.WindowManager;
6:
7: /**
8:  * A base implementation of {@link SystemUiHider}. Uses APIs available in all
9:  * API levels to show and hide the status bar.
10: */
11: public class SystemUiHiderBase extends SystemUiHider {
12:     /**
13:      * Whether or not the system UI is currently visible. This is a cached value
14:      * from calls to {@link #hide()} and {@link #show()}.
15:      */
16:     private boolean mVisible = true;
17:
18:     /**
19:      * Constructor not intended to be called by clients. Use
20:      * {@link SystemUiHider#getInstance} to obtain an instance.
21:      */
22:     protected SystemUiHiderBase(Activity activity, View anchorView, int flags) {
23:         super(activity, anchorView, flags);
24:     }
25:
26:     @Override
27:     public void setup() {
28:         if ((mFlags & FLAG_LAYOUT_IN_SCREEN_OLDER_DEVICES) == 0) {
29:             mActivity.getWindow().setFlags(
30:                 WindowManager.LayoutParams.FLAG_LAYOUT_IN_SCREEN
31:                 | WindowManager.LayoutParams.FLAG_LAYOUT_NO_LIMITS,
32:                 WindowManager.LayoutParams.FLAG_LAYOUT_IN_SCREEN
33:                 | WindowManager.LayoutParams.FLAG_LAYOUT_NO_LIMITS);
34:         }
35:     }
36:
37:     @Override
38:     public boolean isVisible() {
39:         return mVisible;
40:     }
41:
42:     @Override
43:     public void hide() {
44:         if ((mFlags & FLAG_FULLSCREEN) != 0) {
45:             mActivity.getWindow().setFlags(
46:                 WindowManager.LayoutParams.FLAG_FULLSCREEN,
47:                 WindowManager.LayoutParams.FLAG_FULLSCREEN);
48:         }
```

```
49:         mOnVisibilityChangeListener.onVisibilityChange(false);
50:         mVisible = false;
51:     }
52:
53:     @Override
54:     public void show() {
55:         if ((mFlags & FLAG_FULLSCREEN) != 0) {
56:             mActivity.getWindow().setFlags(
57:                 0,
58:                 WindowManager.LayoutParams.FLAG_FULLSCREEN);
59:         }
60:         mOnVisibilityChangeListener.onVisibilityChange(true);
61:         mVisible = true;
62:     }
63: }
```

```
1: package com.delta.bartalk.util;
2:
3: import android.annotation.TargetApi;
4: import android.app.Activity;
5: import android.os.Build;
6: import android.view.View;
7: import android.view.WindowManager;
8:
9: /**
10:  * An API 11+ implementation of {@link SystemUiHider}. Uses APIs available in
11:  * Honeycomb and later (specifically {@link View#setSystemUiVisibility(int)}) to
12:  * show and hide the system UI.
13:  */
14: @TargetApi(Build.VERSION_CODES.HONEYCOMB)
15: public class SystemUiHiderHoneycomb extends SystemUiHiderBase {
16:     /**
17:      * Flags for {@link View#setSystemUiVisibility(int)} to use when showing the
18:      * system UI.
19:      */
20:     private int mShowFlags;
21:
22:     /**
23:      * Flags for {@link View#setSystemUiVisibility(int)} to use when hiding the
24:      * system UI.
25:      */
26:     private int mHideFlags;
27:
28:     /**
29:      * Flags to test against the first parameter in
30:      * {@link android.view.View.OnSystemUiVisibilityChangeListener#onSystemUiVisibilityChange(int)}
31:      * to determine the system UI visibility state.
32:      */
33:     private int mTestFlags;
34:
35:     /**
36:      * Whether or not the system UI is currently visible. This is cached from
37:      * {@link android.view.View.OnSystemUiVisibilityChangeListener}.
38:      */
39:     private boolean mVisible = true;
40:
41:     /**
42:      * Constructor not intended to be called by clients. Use
43:      * {@link SystemUiHider#getInstance} to obtain an instance.
44:      */
45:     protected SystemUiHiderHoneycomb(Activity activity, View anchorView, int flags) {
46:         super(activity, anchorView, flags);
47:
48:         mShowFlags = View.SYSTEM_UI_FLAG_VISIBLE;
```

```
49:         mHideFlags = View.SYSTEM_UI_FLAG_LOW_PROFILE;
50:         mTestFlags = View.SYSTEM_UI_FLAG_LOW_PROFILE;
51:
52:         if ((mFlags & FLAG_FULLSCREEN) != 0) {
53:             // If the client requested fullscreen, add flags relevant to hiding
54:             // the status bar. Note that some of these constants are new as of
55:             // API 16 (Jelly Bean). It is safe to use them, as they are inlined
56:             // at compile-time and do nothing on pre-Jelly Bean devices.
57:             mShowFlags |= View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN;
58:             mHideFlags |= View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
59:                 | View.SYSTEM_UI_FLAG_FULLSCREEN;
60:         }
61:
62:         if ((mFlags & FLAG_HIDE_NAVIGATION) != 0) {
63:             // If the client requested hiding navigation, add relevant flags.
64:             mShowFlags |= View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION;
65:             mHideFlags |= View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
66:                 | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION;
67:             mTestFlags |= View.SYSTEM_UI_FLAG_HIDE_NAVIGATION;
68:         }
69:     }
70:
71:     /** {@inheritDoc} */
72:     @Override
73:     public void setup() {
74:         mAnchorView.setOnSystemUiVisibilityChangeListener(mSystemUiVisibilityChangeListener);
75:     }
76:
77:     /** {@inheritDoc} */
78:     @Override
79:     public void hide() {
80:         mAnchorView.setSystemUiVisibility(mHideFlags);
81:     }
82:
83:     /** {@inheritDoc} */
84:     @Override
85:     public void show() {
86:         mAnchorView.setSystemUiVisibility(mShowFlags);
87:     }
88:
89:     /** {@inheritDoc} */
90:     @Override
91:     public boolean isVisible() {
92:         return mVisible;
93:     }
94:
95:     private View.OnSystemUiVisibilityChangeListener mSystemUiVisibilityChangeListener
96:         = new View.OnSystemUiVisibilityChangeListener() {
```



```
97:         @Override
98:         public void onSystemUiVisibilityChange(int vis) {
99:             // Test against mTestFlags to see if the system UI is visible.
100:            if ((vis & mTestFlags) != 0) {
101:                if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN) {
102:                    // Pre-Jelly Bean, we must manually hide the action bar
103:                    // and use the old window flags API.
104:                    mActivity.getActionBar().hide();
105:                    mActivity.getWindow().setFlags(
106:                        WindowManager.LayoutParams.FLAG_FULLSCREEN,
107:                        WindowManager.LayoutParams.FLAG_FULLSCREEN);
108:                }
109:
110:                // Trigger the registered listener and cache the visibility
111:                // state.
112:                mOnVisibilityChangeListener.onVisibilityChange(false);
113:                mVisible = false;
114:
115:            } else {
116:                mAnchorView.setSystemUiVisibility(mShowFlags);
117:                if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN) {
118:                    // Pre-Jelly Bean, we must manually show the action bar
119:                    // and use the old window flags API.
120:                    mActivity.getActionBar().show();
121:                    mActivity.getWindow().setFlags(
122:                        0,
123:                        WindowManager.LayoutParams.FLAG_FULLSCREEN);
124:                }
125:
126:                // Trigger the registered listener and cache the visibility
127:                // state.
128:                mOnVisibilityChangeListener.onVisibilityChange(true);
129:                mVisible = true;
130:            }
131:        }
132:    };
133: }
```

```
1: package com.delta.bartalk;
2:
3: import android.annotation.TargetApi;
4: import android.content.Context;
5: import android.content.res.Resources;
6: import android.graphics.RectF;
7: import android.os.Build;
8: import android.text.Layout.Alignment;
9: import android.text.StaticLayout;
10: import android.text.TextPaint;
11: import android.util.AttributeSet;
12: import android.util.SparseIntArray;
13: import android.util.TypedValue;
14: import android.widget.TextView;
15:
16: /*
17:  * Taken from http://stackoverflow.com/users/1112882/m-wajeeh 's code on Stack Overflow
18:  * here:
19:  * http://stackoverflow.com/questions/5033012/auto-scale-textview-text-to-fit-within-bounds/17782522#17782522
20:  * m-wajeeh provided a completely amazingly robust widget to handle auto-resizing!
21:  */
22:
23:
24:
25: */
26:
27: public class AutoResizeTextView extends TextView {
28:     private interface SizeTester {
29:         /**
30:          *
31:          * @param suggestedSize
32:          *         Size of text to be tested
33:          * @param availableSpace
34:          *         available space in which text must fit
35:          * @return an integer < 0 if after applying {@code suggestedSize} to
36:          *         text, it takes less space than {@code availableSpace}, > 0
37:          *         otherwise
38:          */
39:         public int onTestSize(int suggestedSize, RectF availableSpace);
40:     }
41:
42:     private RectF mTextRect = new RectF();
43:
44:     private RectF mAvailableSpaceRect;
45:
46:     private SparseIntArray mTextCachedSizes;
47:
48:     private TextPaint mPaint;
```

```
49:
50:     private float mMaxTextSize;
51:
52:     private float mSpacingMult = 1.0f;
53:
54:     private float mSpacingAdd = 0.0f;
55:
56:     private float mMinTextSize = 20;
57:
58:     private int mWidthLimit;
59:
60:     private static final int NO_LINE_LIMIT = -1;
61:     private int mMaxLines;
62:
63:     private boolean mEnableSizeCache = true;
64:     private boolean mInitallized;
65:
66:     public AutoResizeTextView(Context context) {
67:         super(context);
68:         initialize();
69:     }
70:
71:     public AutoResizeTextView(Context context, AttributeSet attrs) {
72:         super(context, attrs);
73:         initialize();
74:     }
75:
76:     public AutoResizeTextView(Context context, AttributeSet attrs, int defStyle) {
77:         super(context, attrs, defStyle);
78:         initialize();
79:     }
80:
81:     private void initialize() {
82:         mPaint = new TextPaint(getPaint());
83:         mMaxTextSize = getTextSize();
84:         mAvailableSpaceRect = new RectF();
85:         mTextCachedSizes = new SparseIntArray();
86:         if (mMaxLines == 0) {
87:             // no value was assigned during construction
88:             mMaxLines = NO_LINE_LIMIT;
89:         }
90:         mInitallized = true;
91:     }
92:
93:     @Override
94:     public void setText(final CharSequence text, BufferType type) {
95:         super.setText(text, type);
96:         adjustTextSize(text.toString());
```

```
97:     }
98:
99:     @Override
100:    public void setTextSize(float size) {
101:        mMaxTextSize = size;
102:        mTextCachedSizes.clear();
103:        adjustTextSize(getText().toString());
104:    }
105:
106:    @Override
107:    public void setMaxLines(int maxlines) {
108:        super.setMaxLines(maxlines);
109:        mMaxLines = maxlines;
110:        reAdjust();
111:    }
112:
113:    public int getMaxLines() {
114:        return mMaxLines;
115:    }
116:
117:    @Override
118:    public void setSingleLine() {
119:        super.setSingleLine();
120:        mMaxLines = 1;
121:        reAdjust();
122:    }
123:
124:    @Override
125:    public void setSingleLine(boolean singleLine) {
126:        super.setSingleLine(singleLine);
127:        if (singleLine) {
128:            mMaxLines = 1;
129:        } else {
130:            mMaxLines = NO_LINE_LIMIT;
131:        }
132:        reAdjust();
133:    }
134:
135:    @Override
136:    public void setLines(int lines) {
137:        super.setLines(lines);
138:        mMaxLines = lines;
139:        reAdjust();
140:    }
141:
142:    @Override
143:    public void setTextSize(int unit, float size) {
144:        Context c = getContext();
```

```
145:         Resources r;
146:
147:         if (c == null)
148:             r = Resources.getSystem();
149:         else
150:             r = c.getResources();
151:         mMaxTextSize = TypedValue.applyDimension(unit, size,
152:             r.getDisplayMetrics());
153:         mTextCachedSizes.clear();
154:         adjustTextSize(getText().toString());
155:     }
156:
157:     @Override
158:     public void setLineSpacing(float add, float mult) {
159:         super.setLineSpacing(add, mult);
160:         mSpacingMult = mult;
161:         mSpacingAdd = add;
162:     }
163:
164:     /**
165:      * Set the lower text size limit and invalidate the view
166:      *
167:      * @param minTextSize
168:      */
169:     public void setMinTextSize(float minTextSize) {
170:         mMinTextSize = minTextSize;
171:         reAdjust();
172:     }
173:
174:     private void reAdjust() {
175:         adjustTextSize(getText().toString());
176:     }
177:
178:     private void adjustTextSize(String string) {
179:         if (!mInitiallized) {
180:             return;
181:         }
182:         int startSize = (int) mMinTextSize;
183:         int heightLimit = getMeasuredHeight() - getCompoundPaddingBottom()
184:             - getCompoundPaddingTop();
185:         mWidthLimit = getMeasuredWidth() - getCompoundPaddingLeft()
186:             - getCompoundPaddingRight();
187:         mAvailableSpaceRect.right = mWidthLimit;
188:         mAvailableSpaceRect.bottom = heightLimit;
189:         super.setTextSize(
190:             TypedValue.COMPLEX_UNIT_PX,
191:             efficientTextSizeSearch(startSize, (int) mMaxTextSize,
192:                 mSizeTester, mAvailableSpaceRect));
```

```
193:     }
194:
195:     private final SizeTester mSizeTester = new SizeTester() {
196:         @TargetApi(Build.VERSION_CODES.JELLY_BEAN)
197:         @Override
198:         public int onTestSize(int suggestedSize, RectF availableSpace) {
199:             mPaint.setTextSize(suggestedSize);
200:             String text = getText().toString();
201:             boolean singleline = getMaxLines() == 1;
202:             if (singleline) {
203:                 mTextRect.bottom = mPaint.getFontSpacing();
204:                 mTextRect.right = mPaint.measureText(text);
205:             } else {
206:                 StaticLayout layout = new StaticLayout(text, mPaint,
207:                     mWidthLimit, Alignment.ALIGN_NORMAL, mSpacingMult,
208:                     mSpacingAdd, true);
209:                 // return early if we have more lines
210:                 if (getMaxLines() != NO_LINE_LIMIT
211:                     && layout.getLineCount() > getMaxLines()) {
212:                     return 1;
213:                 }
214:                 mTextRect.bottom = layout.getHeight();
215:                 int maxWidth = -1;
216:                 for (int i = 0; i < layout.getLineCount(); i++) {
217:                     if (maxWidth < layout.getLineWidth(i)) {
218:                         maxWidth = (int) layout.getLineWidth(i);
219:                     }
220:                 }
221:                 mTextRect.right = maxWidth;
222:             }
223:
224:             mTextRect.offsetTo(0, 0);
225:             if (availableSpace.contains(mTextRect)) {
226:                 // may be too small, don't worry we will find the best match
227:                 return -1;
228:             } else {
229:                 // too big
230:                 return 1;
231:             }
232:         }
233:     };
234:
235:     /**
236:      * Enables or disables size caching, enabling it will improve performance
237:      * where you are animating a value inside TextView. This stores the font
238:      * size against getText().length() Be careful though while enabling it as 0
239:      * takes more space than 1 on some fonts and so on.
240:      *
```

```
241:      * @param enable
242:      *           enable font size caching
243:      */
244:      public void enableSizeCache(boolean enable) {
245:          mEnableSizeCache = enable;
246:          mTextCachedSizes.clear();
247:          adjustTextSize(getText().toString());
248:      }
249:
250:      private int efficientTextSizeSearch(int start, int end,
251:                                          SizeTester sizeTester, RectF availableSpace) {
252:          if (!mEnableSizeCache) {
253:              return binarySearch(start, end, sizeTester, availableSpace);
254:          }
255:          String text = getText().toString();
256:          int key = text == null ? 0 : text.length();
257:          int size = mTextCachedSizes.get(key);
258:          if (size != 0) {
259:              return size;
260:          }
261:          size = binarySearch(start, end, sizeTester, availableSpace);
262:          mTextCachedSizes.put(key, size);
263:          return size;
264:      }
265:
266:      private static int binarySearch(int start, int end, SizeTester sizeTester,
267:                                      RectF availableSpace) {
268:          int lastBest = start;
269:          int lo = start;
270:          int hi = end - 1;
271:          int mid = 0;
272:          while (lo <= hi) {
273:              mid = (lo + hi) >>> 1;
274:              int midValCmp = sizeTester.onTestSize(mid, availableSpace);
275:              if (midValCmp < 0) {
276:                  lastBest = lo;
277:                  lo = mid + 1;
278:              } else if (midValCmp > 0) {
279:                  hi = mid - 1;
280:                  lastBest = hi;
281:              } else {
282:                  return mid;
283:              }
284:          }
285:          // make sure to return last best
286:          // this is what should always be returned
287:          return lastBest;
288:
```

```
289:     }
290:
291:     @Override
292:     protected void onTextChanged(final CharSequence text, final int start,
293:                                   final int before, final int after) {
294:         super.onTextChanged(text, start, before, after);
295:         reAdjust();
296:     }
297:
298:     @Override
299:     protected void onSizeChanged(int width, int height, int oldwidth,
300:                                   int oldheight) {
301:         mTextCachedSizes.clear();
302:         super.onSizeChanged(width, height, oldwidth, oldheight);
303:         if (width != oldwidth || height != oldheight) {
304:             reAdjust();
305:         }
306:     }
307: }
```



```
1: package com.delta.bartalk;
2:
3: import com.delta.bartalk.util.SystemUiHider;
4:
5: import android.annotation.TargetApi;
6: import android.app.ActionBar;
7: import android.app.Activity;
8: import android.os.Build;
9: import android.os.Bundle;
10: import android.os.Handler;
11: import android.text.Editable;
12: import android.text.TextWatcher;
13: import android.view.MotionEvent;
14: import android.view.View;
15: import android.widget.EditText;
16: import android.widget.TextView;
17:
18: /**
19:  * An example full-screen activity that shows and hides the system UI (i.e.
20:  * status bar and navigation/system bar) with user interaction.
21:  *
22:  * @see SystemUiHider
23:  */
24:
25: public class BartalkActivity extends Activity {
26:     /**
27:      * Whether or not the system UI should be auto-hidden after
28:      * {@link #AUTO_HIDE_DELAY_MILLIS} milliseconds.
29:      */
30:     private static final boolean AUTO_HIDE = true;
31:
32:     /**
33:      * If {@link #AUTO_HIDE} is set, the number of milliseconds to wait after
34:      * user interaction before hiding the system UI.
35:      */
36:     private static final int AUTO_HIDE_DELAY_MILLIS = 3000;
37:
38:     /**
39:      * If set, will toggle the system UI visibility upon interaction. Otherwise,
40:      * will show the system UI visibility upon interaction.
41:      */
42:     private static final boolean TOGGLE_ON_CLICK = true;
43:
44:     /**
45:      * The flags to pass to {@link SystemUiHider#getInstance}.
46:      */
47:     private static final int HIDER_FLAGS = SystemUiHider.FLAG_HIDE_NAVIGATION;
48:
```

```
49:      /**
50:       * The instance of the {@link SystemUiHider} for this activity.
51:       */
52:     private SystemUiHider mSystemUiHider;
53:
54:     //
55:     // Bartalk instance variables
56:     //
57:     TextView mOutputText;
58:     EditText mInputText;
59:
60:     @Override
61:     protected void onCreate(Bundle savedInstanceState) {
62:         super.onCreate(savedInstanceState);
63:
64:         setContentView(R.layout.activity_bartalk);
65:
66:         //
67:         // Bartalk code
68:         //
69:         mOutputText = (TextView) findViewById(R.id.fullscreen_content);
70:         mInputText = (EditText) findViewById(R.id.input_text);
71:
72:         mInputText.addTextChangedListener(new TextWatcher() {
73:             public void afterTextChanged(Editable s) {}
74:             public void beforeTextChanged(CharSequence s, int start, int count, int after) {}
75:             public void onTextChanged(CharSequence s, int start, int before, int count) {
76:                 mOutputText.setText(mInputText.getText());
77:             }
78:         });
79:
80:         ActionBar actionBar = getActionBar();
81:         actionBar.hide();
82:
83:         final View controlsView = findViewById(R.id.fullscreen_content_controls);
84:         final View contentView = findViewById(R.id.fullscreen_content);
85:
86:         // Set up an instance of SystemUiHider to control the system UI for
87:         // this activity.
88:         mSystemUiHider = SystemUiHider.getInstance(this, contentView, HIDER_FLAGS);
89:         mSystemUiHider.setup();
90:         mSystemUiHider
91:             .setOnVisibilityChangeListener(new SystemUiHider.OnVisibilityChangeListener() {
92:                 // Cached values.
93:                 int mControlsHeight;
94:                 int mShortAnimTime;
95:
96:                 @Override
```

```
97:         @TargetApi(Build.VERSION_CODES.HONEYCOMB_MR2)
98:         public void onVisibilityChange(boolean visible) {
99:             if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {
100:                 // If the ViewPropertyAnimator API is available
101:                 // (Honeycomb MR2 and later), use it to animate the
102:                 // in-layout UI controls at the bottom of the
103:                 // screen.
104:                 if (mControlsHeight == 0) {
105:                     mControlsHeight = controlsView.getHeight();
106:                 }
107:                 if (mShortAnimTime == 0) {
108:                     mShortAnimTime = getResources().getInteger(
109:                         android.R.integer.config_shortAnimTime);
110:                 }
111:                 controlsView.animate()
112:                     .translationY(visible ? 0 : mControlsHeight)
113:                     .setDuration(mShortAnimTime);
114:             } else {
115:                 // If the ViewPropertyAnimator APIs aren't
116:                 // available, simply show or hide the in-layout UI
117:                 // controls.
118:                 controlsView.setVisibility(visible ? View.VISIBLE : View.GONE);
119:             }
120:
121:             if (visible && AUTO_HIDE) {
122:                 // Schedule a hide().
123:                 delayedHide(AUTO_HIDE_DELAY_MILLIS);
124:             }
125:         }
126:     });
127:
128:     // Set up the user interaction to manually show or hide the system UI.
129:     contentView.setOnClickListener(new View.OnClickListener() {
130:         @Override
131:         public void onClick(View view) {
132:
133:             if (TOGGLE_ON_CLICK) {
134:                 mSystemUiHider.toggle();
135:             } else {
136:                 mSystemUiHider.show();
137:             }
138:         }
139:     });
140:
141:     // Upon interacting with UI controls, delay any scheduled hide()
142:     // operations to prevent the jarring behavior of controls going away
143:     // while interacting with the UI.
144:     findViewById(R.id.input_text).setOnTouchListener(mDelayHideTouchListener);
```

```
145:     }
146:
147:     @Override
148:     protected void onCreate(Bundle savedInstanceState) {
149:         super.onCreate(savedInstanceState);
150:
151:         // Trigger the initial hide() shortly after the activity has been
152:         // created, to briefly hint to the user that UI controls
153:         // are available.
154:         delayedHide(100);
155:     }
156:
157:
158:     /**
159:      * Touch listener to use for in-layout UI controls to delay hiding the
160:      * system UI. This is to prevent the jarring behavior of controls going away
161:      * while interacting with activity UI.
162:      */
163:     View.OnTouchListener mDelayHideTouchListener = new View.OnTouchListener() {
164:         @Override
165:         public boolean onTouch(View view, MotionEvent motionEvent) {
166:             if (AUTO_HIDE) {
167:                 delayedHide(AUTO_HIDE_DELAY_MILLIS);
168:             }
169:
170:             return false;
171:         }
172:     };
173:
174:     Handler mHideHandler = new Handler();
175:     Runnable mHideRunnable = new Runnable() {
176:         @Override
177:         public void run() {
178:             mSystemUiHider.hide();
179:         }
180:     };
181:
182:     /**
183:      * Schedules a call to hide() in [delay] milliseconds, canceling any
184:      * previously scheduled calls.
185:      */
186:     private void delayedHide(int delayMillis) {
187:         mHideHandler.removeCallbacks(mHideRunnable);
188:         mHideHandler.postDelayed(mHideRunnable, delayMillis);
189:     }
190: }
```

1	bartalk/src/main/java/com/delta/bartalk/util/SystemUiHider.java	4	pages	172	lines	14/03/12	16:02:03
2	bartalk/src/main/java/com/delta/bartalk/util/SystemUiHiderBase.java	2	pages	63	lines	14/03/12	16:02:03
3	bartalk/src/main/java/com/delta/bartalk/util/SystemUiHiderHoneycomb.java	3	pages	133	lines	14/03/12	16:02:03
4	bartalk/src/main/java/com/delta/bartalk/AutoResizeTextView.java	7	pages	306	lines	14/03/12	16:20:57
5	bartalk/src/main/java/com/delta/bartalk/BartalkActivity.java	4	pages	189	lines	14/03/12	16:23:47