```java
/*
 * Copyright (C) 2006 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package psycho.euphoria.tools;

import static java.lang.Math.max;

import android.animation.AnimatorInflater;
import android.animation.StateListAnimator;
import android.annotation.CallSuper;
import android.annotation.ColorInt;
import android.annotation.DrawableRes;
import android.annotation.FloatRange;
import android.annotation.IdRes;
import android.annotation.IntDef;
import android.annotation.IntRange;
import android.annotation.LayoutRes;
import android.annotation.NonNull;
import android.annotation.Nullable;
import android.annotation.Size;
import android.annotation.TestApi;
import android.annotation.UiThread;
import android.content.ClipData;
import android.content.Context;
import android.content.ContextWrapper;
import android.content.Intent;
import android.content.res.ColorStateList;
import android.content.res.Configuration;
import android.content.res.Resources;
import android.content.res.TypedArray;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Insets;
import android.graphics.Interpolator;
import android.graphics.LinearGradient;
import android.graphics.Matrix;
import android.graphics.Outline;
import android.graphics.Paint;
import android.graphics.PixelFormat;
import android.graphics.Point;
import android.graphics.PorterDuff;
import android.graphics.PorterDuffXfermode;
import android.graphics.Rect;
import android.graphics.RectF;
import android.graphics.Region;
import android.graphics.Shader;
import android.graphics.drawable.ColorDrawable;
import android.graphics.drawable.Drawable;
import android.hardware.display.DisplayManagerGlobal;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Parcel;
import android.os.Parcelable;
import android.os.RemoteException;
import android.os.SystemClock;
import android.os.SystemProperties;
import android.os.Trace;
import android.text.TextUtils;
import android.util.AttributeSet;
import android.util.FloatProperty;
import android.util.LayoutDirection;
```

```
import android.util.Log;
import android.util.LongSparseLongArray;
import android.util.Pools.SynchronizedPool;
import android.util.Property;
import android.util.SparseArray;
import android.util.StateSet;
import android.util.SuperNotCalledException;
import android.util.TypedValue;
import android.view.AccessibilityIterators.CharacterTextSegmentIterator;
import android.view.AccessibilityIterators.ParagraphTextSegmentIterator;
import android.view.AccessibilityIterators.TextSegmentIterator;
import android.view.AccessibilityIterators.WordTextSegmentIterator;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.accessibility.AccessibilityEvent;
import android.view.accessibility.AccessibilityEventSource;
import android.view.accessibility.AccessibilityManager;
import android.view.accessibility.AccessibilityNodeInfo;
import android.view.accessibility.AccessibilityNodeInfo.AccessibilityAction;
import android.view.accessibility.AccessibilityNodeProvider;
import android.view.accessibility.AccessibilityWindowInfo;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.view.animation.Transformation;
import android.view.autofill.AutofillId;
import android.view.autofill.AutofillManager;
import android.view.autofill.AutofillValue;
import android.view.inputmethod.EditorInfo;
import android.view.inputmethod.InputConnection;
import android.view.inputmethod.InputMethodManager;
import android.widget.Checkable;
import android.widget.FrameLayout;
import android.widget.ScrollBarDrawable;

import com.android.internal.R;
import com.android.internal.view.TooltipPopup;
import com.android.internal.view.menu.MenuBuilder;
import com.android.internal.widget.ScrollBarUtils;

import com.google.android.collect.Lists;
import com.google.android.collect.Maps;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.ref.WeakReference;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Calendar;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Predicate;

/**
 * <p>
 * This class represents the basic building block for user interface components. A View
 * occupies a rectangular area on the screen and is responsible for drawing and
 * event handling. View is the base class for <em>widgets</em>, which are
 * used to create interactive UI components (buttons, text fields, etc.). The
 * {@link android.view.ViewGroup} subclass is the base class for <em>layouts</em>, which
 * are invisible containers that hold other Views (or other ViewGroups) and define
 * their layout properties.
 * </p>
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For information about using this class to develop your application's user interface,
 * read the <a href="{@docRoot}guide/topics/ui/index.html">User Interface</a> developer guide.
 * </div>
 *
 * <a name="Using"></a>
 * <h3>Using Views</h3>
 * <p>
 * All of the views in a window are arranged in a single tree. You can add views
```

```
* either from code or by specifying a tree of views in one or more XML layout
* files. There are many specialized subclasses of views that act as controls or
* are capable of displaying text, images, or other content.
* </p>
* <p>
* Once you have created a tree of views, there are typically a few types of
* common operations you may wish to perform:
* <ul>
* <li><strong>Set properties:</strong> for example setting the text of a
* {@link android.widget.TextView}. The available properties and the methods
* that set them will vary among the different subclasses of views. Note that
* properties that are known at build time can be set in the XML layout
* files.</li>
* <li><strong>Set focus:</strong> The framework will handle moving focus in
* response to user input. To force focus to a specific view, call
* {@link #requestFocus}.</li>
* <li><strong>Set up listeners:</strong> Views allow clients to set listeners
* that will be notified when something interesting happens to the view. For
* example, all views will let you set a listener to be notified when the view
* gains or loses focus. You can register such a listener using
* {@link #setOnFocusChangeListener(android.view.View.OnFocusChangeListener)}.
* Other view subclasses offer more specialized listeners. For example, a Button
* exposes a listener to notify clients when the button is clicked.</li>
* <li><strong>Set visibility:</strong> You can hide or show views using
* {@link #setVisibility(int)}.</li>
* </ul>
* </p>
* <p><em>
* Note: The Android framework is responsible for measuring, laying out and
* drawing views. You should not call methods that perform these actions on
* views yourself unless you are actually implementing a
* {@link android.view.ViewGroup}.
* </em></p>
*
* <a name="Lifecycle"></a>
* <h3>Implementing a Custom View</h3>
*
* <p>
* To implement a custom view, you will usually begin by providing overrides for
* some of the standard methods that the framework calls on all views. You do
* not need to override all of these methods. In fact, you can start by just
* overriding {@link #onDraw(android.graphics.Canvas)}.
* <table border="2" width="85%" align="center" cellpadding="5">
*     <thead>
*         <tr><th>Category</th> <th>Methods</th> <th>Description</th></tr>
*     </thead>
*
*     <tbody>
*     <tr>
*         <td rowspan="2">Creation</td>
*         <td>Constructors</td>
*         <td>There is a form of the constructor that are called when the view
*         is created from code and a form that is called when the view is
*         inflated from a layout file. The second form should parse and apply
*         any attributes defined in the layout file.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onFinishInflate()}</code></td>
*         <td>Called after a view and all of its children has been inflated
*         from XML.</td>
*     </tr>
*
*     <tr>
*         <td rowspan="3">Layout</td>
*         <td><code>{@link #onMeasure(int, int)}</code></td>
*         <td>Called to determine the size requirements for this view and all
*         of its children.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onLayout(boolean, int, int, int, int)}</code></td>
*         <td>Called when this view should assign a size and position to all
*         of its children.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onSizeChanged(int, int, int, int)}</code></td>
*         <td>Called when the size of this view has changed.
*         </td>
*     </tr>
*
```

```
*     <tr>
*         <td>Drawing</td>
*         <td><code>{@link #onDraw(android.graphics.Canvas)}</code></td>
*         <td>Called when the view should render its content.
*         </td>
*     </tr>
*
*     <tr>
*         <td rowspan="4">Event processing</td>
*         <td><code>{@link #onKeyDown(int, KeyEvent)}</code></td>
*         <td>Called when a new hardware key event occurs.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onKeyUp(int, KeyEvent)}</code></td>
*         <td>Called when a hardware key up event occurs.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onTrackballEvent(MotionEvent)}</code></td>
*         <td>Called when a trackball motion event occurs.
*         </td>
*     </tr>
*     <tr>
*         <td><code>{@link #onTouchEvent(MotionEvent)}</code></td>
*         <td>Called when a touch screen motion event occurs.
*         </td>
*     </tr>
*
*     <tr>
*         <td rowspan="2">Focus</td>
*         <td><code>{@link #onFocusChanged(boolean, int, android.graphics.Rect)}</code></td>
*         <td>Called when the view gains or loses focus.
*         </td>
*     </tr>
*
*     <tr>
*         <td><code>{@link #onWindowFocusChanged(boolean)}</code></td>
*         <td>Called when the window containing the view gains or loses focus.
*         </td>
*     </tr>
*
*     <tr>
*         <td rowspan="3">Attaching</td>
*         <td><code>{@link #onAttachedToWindow()}</code></td>
*         <td>Called when the view is attached to a window.
*         </td>
*     </tr>
*
*     <tr>
*         <td><code>{@link #onDetachedFromWindow}</code></td>
*         <td>Called when the view is detached from its window.
*         </td>
*     </tr>
*
*     <tr>
*         <td><code>{@link #onWindowVisibilityChanged(int)}</code></td>
*         <td>Called when the visibility of the window containing the view
*         has changed.
*         </td>
*     </tr>
*     </tbody>
*
* </table>
* </p>
*
* <a name="IDs"></a>
* <h3>IDs</h3>
* Views may have an integer id associated with them. These ids are typically
* assigned in the layout XML files, and are used to find specific views within
* the view tree. A common pattern is to:
* <ul>
* <li>Define a Button in the layout file and assign it a unique ID.
* <pre>
* &lt;Button
*     android:id="@+id/my_button"
*     android:layout_width="wrap_content"
*     android:layout_height="wrap_content"
*     android:text="@string/my_button_text"/&gt;
* </pre></li>
* <li>From the onCreate method of an Activity, find the Button
* <pre class="prettyprint">
```

```
*      Button myButton = findViewById(R.id.my_button);
* </pre></li>
* </ul>
* <p>
* View IDs need not be unique throughout the tree, but it is good practice to
* ensure that they are at least unique within the part of the tree you are
* searching.
* </p>
*
* <a name="Position"></a>
* <h3>Position</h3>
* <p>
* The geometry of a view is that of a rectangle. A view has a location,
* expressed as a pair of <em>left</em> and <em>top</em> coordinates, and
* two dimensions, expressed as a width and a height. The unit for location
* and dimensions is the pixel.
* </p>
*
* <p>
* It is possible to retrieve the location of a view by invoking the methods
* {@link #getLeft()} and {@link #getTop()}. The former returns the left, or X,
* coordinate of the rectangle representing the view. The latter returns the
* top, or Y, coordinate of the rectangle representing the view. These methods
* both return the location of the view relative to its parent. For instance,
* when getLeft() returns 20, that means the view is located 20 pixels to the
* right of the left edge of its direct parent.
* </p>
*
* <p>
* In addition, several convenience methods are offered to avoid unnecessary
* computations, namely {@link #getRight()} and {@link #getBottom()}.
* These methods return the coordinates of the right and bottom edges of the
* rectangle representing the view. For instance, calling {@link #getRight()}
* is similar to the following computation: <code>getLeft() + getWidth()</code>
* (see <a href="#SizePaddingMargins">Size</a> for more information about the width.)
* </p>
*
* <a name="SizePaddingMargins"></a>
* <h3>Size, padding and margins</h3>
* <p>
* The size of a view is expressed with a width and a height. A view actually
* possess two pairs of width and height values.
* </p>
*
* <p>
* The first pair is known as <em>measured width</em> and
* <em>measured height</em>. These dimensions define how big a view wants to be
* within its parent (see <a href="#Layout">Layout</a> for more details.) The
* measured dimensions can be obtained by calling {@link #getMeasuredWidth()}
* and {@link #getMeasuredHeight()}.
* </p>
*
* <p>
* The second pair is simply known as <em>width</em> and <em>height</em>, or
* sometimes <em>drawing width</em> and <em>drawing height</em>. These
* dimensions define the actual size of the view on screen, at drawing time and
* after layout. These values may, but do not have to, be different from the
* measured width and height. The width and height can be obtained by calling
* {@link #getWidth()} and {@link #getHeight()}.
* </p>
*
* <p>
* To measure its dimensions, a view takes into account its padding. The padding
* is expressed in pixels for the left, top, right and bottom parts of the view.
* Padding can be used to offset the content of the view by a specific amount of
* pixels. For instance, a left padding of 2 will push the view's content by
* 2 pixels to the right of the left edge. Padding can be set using the
* {@link #setPadding(int, int, int, int)} or {@link #setPaddingRelative(int, int, int, int)}
* method and queried by calling {@link #getPaddingLeft()}, {@link #getPaddingTop()},
* {@link #getPaddingRight()}, {@link #getPaddingBottom()}, {@link #getPaddingStart()},
* {@link #getPaddingEnd()}.
* </p>
*
* <p>
* Even though a view can define a padding, it does not provide any support for
* margins. However, view groups provide such a support. Refer to
* {@link android.view.ViewGroup} and
* {@link android.view.ViewGroup.MarginLayoutParams} for further information.
* </p>
*
* <a name="Layout"></a>
* <h3>Layout</h3>
```

```
* <p>
* Layout is a two pass process: a measure pass and a layout pass. The measuring
* pass is implemented in {@link #measure(int, int)} and is a top-down traversal
* of the view tree. Each view pushes dimension specifications down the tree
* during the recursion. At the end of the measure pass, every view has stored
* its measurements. The second pass happens in
* {@link #layout(int,int,int,int)} and is also top-down. During
* this pass each parent is responsible for positioning all of its children
* using the sizes computed in the measure pass.
* </p>
*
* <p>
* When a view's measure() method returns, its {@link #getMeasuredWidth()} and
* {@link #getMeasuredHeight()} values must be set, along with those for all of
* that view's descendants. A view's measured width and measured height values
* must respect the constraints imposed by the view's parents. This guarantees
* that at the end of the measure pass, all parents accept all of their
* children's measurements. A parent view may call measure() more than once on
* its children. For example, the parent may measure each child once with
* unspecified dimensions to find out how big they want to be, then call
* measure() on them again with actual numbers if the sum of all the children's
* unconstrained sizes is too big or too small.
* </p>
*
* <p>
* The measure pass uses two classes to communicate dimensions. The
* {@link MeasureSpec} class is used by views to tell their parents how they
* want to be measured and positioned. The base LayoutParams class just
* describes how big the view wants to be for both width and height. For each
* dimension, it can specify one of:
* <ul>
* <li> an exact number
* <li>MATCH_PARENT, which means the view wants to be as big as its parent
* (minus padding)
* <li> WRAP_CONTENT, which means that the view wants to be just big enough to
* enclose its content (plus padding).
* </ul>
* There are subclasses of LayoutParams for different subclasses of ViewGroup.
* For example, AbsoluteLayout has its own subclass of LayoutParams which adds
* an X and Y value.
* </p>
*
* <p>
* MeasureSpecs are used to push requirements down the tree from parent to
* child. A MeasureSpec can be in one of three modes:
* <ul>
* <li>UNSPECIFIED: This is used by a parent to determine the desired dimension
* of a child view. For example, a LinearLayout may call measure() on its child
* with the height set to UNSPECIFIED and a width of EXACTLY 240 to find out how
* tall the child view wants to be given a width of 240 pixels.
* <li>EXACTLY: This is used by the parent to impose an exact size on the
* child. The child must use this size, and guarantee that all of its
* descendants will fit within this size.
* <Li>AT_MOST: This is used by the parent to impose a maximum size on the
* child. The child must guarantee that it and all of its descendants will fit
* within this size.
* </ul>
* </p>
*
* <p>
* To initiate a layout, call {@link #requestLayout}. This method is typically
* called by a view on itself when it believes that is can no longer fit within
* its current bounds.
* </p>
*
* <a name="Drawing"></a>
* <h3>Drawing</h3>
* <p>
* Drawing is handled by walking the tree and recording the drawing commands of
* any View that needs to update. After this, the drawing commands of the
* entire tree are issued to screen, clipped to the newly damaged area.
* </p>
*
* <p>
* The tree is largely recorded and drawn in order, with parents drawn before
* (i.e., behind) their children, with siblings drawn in the order they appear
* in the tree. If you set a background drawable for a View, then the View will
* draw it before calling back to its <code>onDraw()</code> method. The child
* drawing order can be overridden with
* {@link ViewGroup#setChildrenDrawingOrderEnabled(boolean) custom child drawing order}
* in a ViewGroup, and with {@link #setZ(float)} custom Z values} set on Views.
* </p>
```

```
*
* <p>
* To force a view to draw, call {@link #invalidate()}.
* </p>
*
* <a name="EventHandlingThreading"></a>
* <h3>Event Handling and Threading</h3>
* <p>
* The basic cycle of a view is as follows:
* <ol>
* <li>An event comes in and is dispatched to the appropriate view. The view
* handles the event and notifies any listeners.</li>
* <li>If in the course of processing the event, the view's bounds may need
* to be changed, the view will call {@link #requestLayout()}.</li>
* <li>Similarly, if in the course of processing the event the view's appearance
* may need to be changed, the view will call {@link #invalidate()}.</li>
* <li>If either {@link #requestLayout()} or {@link #invalidate()} were called,
* the framework will take care of measuring, laying out, and drawing the tree
* as appropriate.</li>
* </ol>
* </p>
*
* <p><em>Note: The entire view tree is single threaded. You must always be on
* the UI thread when calling any method on any view.</em>
* If you are doing work on other threads and want to update the state of a view
* from that thread, you should use a {@link Handler}.
* </p>
*
* <a name="FocusHandling"></a>
* <h3>Focus Handling</h3>
* <p>
* The framework will handle routine focus movement in response to user input.
* This includes changing the focus as views are removed or hidden, or as new
* views become available. Views indicate their willingness to take focus
* through the {@link #isFocusable} method. To change whether a view can take
* focus, call {@link #setFocusable(boolean)}.  When in touch mode (see notes below)
* views indicate whether they still would like focus via {@link #isFocusableInTouchMode}
* and can change this via {@link #setFocusableInTouchMode(boolean)}.
* </p>
* <p>
* Focus movement is based on an algorithm which finds the nearest neighbor in a
* given direction. In rare cases, the default algorithm may not match the
* intended behavior of the developer. In these situations, you can provide
* explicit overrides by using these XML attributes in the layout file:
* <pre>
* nextFocusDown
* nextFocusLeft
* nextFocusRight
* nextFocusUp
* </pre>
* </p>
*
*
* <p>
* To get a particular view to take focus, call {@link #requestFocus()}.
* </p>
*
* <a name="TouchMode"></a>
* <h3>Touch Mode</h3>
* <p>
* When a user is navigating a user interface via directional keys such as a D-pad, it is
* necessary to give focus to actionable items such as buttons so the user can see
* what will take input.  If the device has touch capabilities, however, and the user
* begins interacting with the interface by touching it, it is no longer necessary to
* always highlight, or give focus to, a particular view.  This motivates a mode
* for interaction named 'touch mode'.
* </p>
* <p>
* For a touch capable device, once the user touches the screen, the device
* will enter touch mode.  From this point onward, only views for which
* {@link #isFocusableInTouchMode} is true will be focusable, such as text editing widgets.
* Other views that are touchable, like buttons, will not take focus when touched; they will
* only fire the on click listeners.
* </p>
* <p>
* Any time a user hits a directional key, such as a D-pad direction, the view device will
* exit touch mode, and find a view to take focus, so that the user may resume interacting
* with the user interface without touching the screen again.
* </p>
* <p>
* The touch mode state is maintained across {@link android.app.Activity}s.  Call
* {@link #isInTouchMode} to see whether the device is currently in touch mode.
```

```
* </p>
*
* <a name="Scrolling"></a>
* <h3>Scrolling</h3>
* <p>
* The framework provides basic support for views that wish to internally
* scroll their content. This includes keeping track of the X and Y scroll
* offset as well as mechanisms for drawing scrollbars. See
* {@link #scrollBy(int, int)}, {@link #scrollTo(int, int)}, and
* {@link #awakenScrollBars()} for more details.
* </p>
*
* <a name="Tags"></a>
* <h3>Tags</h3>
* <p>
* Unlike IDs, tags are not used to identify views. Tags are essentially an
* extra piece of information that can be associated with a view. They are most
* often used as a convenience to store data related to views in the views
* themselves rather than by putting them in a separate structure.
* </p>
* <p>
* Tags may be specified with character sequence values in layout XML as either
* a single tag using the {@link android.R.styleable#View_tag android:tag}
* attribute or multiple tags using the {@code <tag>} child element:
* <pre>
*     &lt;View ...
*           android:tag="@string/mytag_value" /&gt;
*     &lt;View ...&gt;
*         &lt;tag android:id="@+id/mytag"
*              android:value="@string/mytag_value" /&gt;
*     &lt;/View>
* </pre>
* </p>
* <p>
* Tags may also be specified with arbitrary objects from code using
* {@link #setTag(Object)} or {@link #setTag(int, Object)}.
* </p>
*
* <a name="Themes"></a>
* <h3>Themes</h3>
* <p>
* By default, Views are created using the theme of the Context object supplied
* to their constructor; however, a different theme may be specified by using
* the {@link android.R.styleable#View_theme android:theme} attribute in layout
* XML or by passing a {@link ContextThemeWrapper} to the constructor from
* code.
* </p>
* <p>
* When the {@link android.R.styleable#View_theme android:theme} attribute is
* used in XML, the specified theme is applied on top of the inflation
* context's theme (see {@link LayoutInflater}) and used for the view itself as
* well as any child elements.
* </p>
* <p>
* In the following example, both views will be created using the Material dark
* color scheme; however, because an overlay theme is used which only defines a
* subset of attributes, the value of
* {@link android.R.styleable#Theme_colorAccent android:colorAccent} defined on
* the inflation context's theme (e.g. the Activity theme) will be preserved.
* <pre>
*     &lt;LinearLayout
*             ...
*             android:theme="@android:theme/ThemeOverlay.Material.Dark"&gt;
*         &lt;View ...&gt;
*     &lt;/LinearLayout&gt;
* </pre>
* </p>
*
* <a name="Properties"></a>
* <h3>Properties</h3>
* <p>
* The View class exposes an {@link #ALPHA} property, as well as several transform-related
* properties, such as {@link #TRANSLATION_X} and {@link #TRANSLATION_Y}. These properties are
* available both in the {@link Property} form as well as in similarly-named setter/getter
* methods (such as {@link #setAlpha(float)} for {@link #ALPHA}). These properties can
* be used to set persistent state associated with these rendering-related properties on the view.
* The properties and methods can also be used in conjunction with
* {@link android.animation.Animator Animator}-based animations, described more in the
* <a href="#Animation">Animation</a> section.
* </p>
*
* <a name="Animation"></a>
```

```
* <h3>Animation</h3>
* <p>
* Starting with Android 3.0, the preferred way of animating views is to use the
* {@link android.animation} package APIs. These {@link android.animation.Animator Animator}-based
* classes change actual properties of the View object, such as {@link #setAlpha(float) alpha} and
* {@link #setTranslationX(float) translationX}. This behavior is contrasted to that of the pre-3.0
* {@link android.view.animation.Animation Animation}-based classes, which instead animate only
* how the view is drawn on the display. In particular, the {@link ViewPropertyAnimator} class
* makes animating these View properties particularly easy and efficient.
* </p>
* <p>
* Alternatively, you can use the pre-3.0 animation classes to animate how Views are rendered.
* You can attach an {@link Animation} object to a view using
* {@link #setAnimation(Animation)} or
* {@link #startAnimation(Animation)}. The animation can alter the scale,
* rotation, translation and alpha of a view over time. If the animation is
* attached to a view that has children, the animation will affect the entire
* subtree rooted by that node. When an animation is started, the framework will
* take care of redrawing the appropriate views until the animation completes.
* </p>
*
* <a name="Security"></a>
* <h3>Security</h3>
* <p>
* Sometimes it is essential that an application be able to verify that an action
* is being performed with the full knowledge and consent of the user, such as
* granting a permission request, making a purchase or clicking on an advertisement.
* Unfortunately, a malicious application could try to spoof the user into
* performing these actions, unaware, by concealing the intended purpose of the view.
* As a remedy, the framework offers a touch filtering mechanism that can be used to
* improve the security of views that provide access to sensitive functionality.
* </p><p>
* To enable touch filtering, call {@link #setFilterTouchesWhenObscured(boolean)} or set the
* android:filterTouchesWhenObscured layout attribute to true.  When enabled, the framework
* will discard touches that are received whenever the view's window is obscured by
* another visible window.  As a result, the view will not receive touches whenever a
* toast, dialog or other window appears above the view's window.
* </p><p>
* For more fine-grained control over security, consider overriding the
* {@link #onFilterTouchEventForSecurity(MotionEvent)} method to implement your own
* security policy. See also {@link MotionEvent#FLAG_WINDOW_IS_OBSCURED}.
* </p>
*
* @attr ref android.R.styleable#View_alpha
* @attr ref android.R.styleable#View_background
* @attr ref android.R.styleable#View_clickable
* @attr ref android.R.styleable#View_contentDescription
* @attr ref android.R.styleable#View_drawingCacheQuality
* @attr ref android.R.styleable#View_duplicateParentState
* @attr ref android.R.styleable#View_id
* @attr ref android.R.styleable#View_requiresFadingEdge
* @attr ref android.R.styleable#View_fadeScrollbars
* @attr ref android.R.styleable#View_fadingEdgeLength
* @attr ref android.R.styleable#View_filterTouchesWhenObscured
* @attr ref android.R.styleable#View_fitsSystemWindows
* @attr ref android.R.styleable#View_isScrollContainer
* @attr ref android.R.styleable#View_focusable
* @attr ref android.R.styleable#View_focusableInTouchMode
* @attr ref android.R.styleable#View_focusedByDefault
* @attr ref android.R.styleable#View_hapticFeedbackEnabled
* @attr ref android.R.styleable#View_keepScreenOn
* @attr ref android.R.styleable#View_keyboardNavigationCluster
* @attr ref android.R.styleable#View_layerType
* @attr ref android.R.styleable#View_layoutDirection
* @attr ref android.R.styleable#View_longClickable
* @attr ref android.R.styleable#View_minHeight
* @attr ref android.R.styleable#View_minWidth
* @attr ref android.R.styleable#View_nextClusterForward
* @attr ref android.R.styleable#View_nextFocusDown
* @attr ref android.R.styleable#View_nextFocusLeft
* @attr ref android.R.styleable#View_nextFocusRight
* @attr ref android.R.styleable#View_nextFocusUp
* @attr ref android.R.styleable#View_onClick
* @attr ref android.R.styleable#View_padding
* @attr ref android.R.styleable#View_paddingHorizontal
* @attr ref android.R.styleable#View_paddingVertical
* @attr ref android.R.styleable#View_paddingBottom
* @attr ref android.R.styleable#View_paddingLeft
* @attr ref android.R.styleable#View_paddingRight
* @attr ref android.R.styleable#View_paddingTop
* @attr ref android.R.styleable#View_paddingStart
* @attr ref android.R.styleable#View_paddingEnd
```

```
 * @attr ref android.R.styleable#View_saveEnabled
 * @attr ref android.R.styleable#View_rotation
 * @attr ref android.R.styleable#View_rotationX
 * @attr ref android.R.styleable#View_rotationY
 * @attr ref android.R.styleable#View_scaleX
 * @attr ref android.R.styleable#View_scaleY
 * @attr ref android.R.styleable#View_scrollX
 * @attr ref android.R.styleable#View_scrollY
 * @attr ref android.R.styleable#View_scrollbarSize
 * @attr ref android.R.styleable#View_scrollbarStyle
 * @attr ref android.R.styleable#View_scrollbars
 * @attr ref android.R.styleable#View_scrollbarDefaultDelayBeforeFade
 * @attr ref android.R.styleable#View_scrollbarFadeDuration
 * @attr ref android.R.styleable#View_scrollbarTrackHorizontal
 * @attr ref android.R.styleable#View_scrollbarThumbHorizontal
 * @attr ref android.R.styleable#View_scrollbarThumbVertical
 * @attr ref android.R.styleable#View_scrollbarTrackVertical
 * @attr ref android.R.styleable#View_scrollbarAlwaysDrawHorizontalTrack
 * @attr ref android.R.styleable#View_scrollbarAlwaysDrawVerticalTrack
 * @attr ref android.R.styleable#View_stateListAnimator
 * @attr ref android.R.styleable#View_transitionName
 * @attr ref android.R.styleable#View_soundEffectsEnabled
 * @attr ref android.R.styleable#View_tag
 * @attr ref android.R.styleable#View_textAlignment
 * @attr ref android.R.styleable#View_textDirection
 * @attr ref android.R.styleable#View_transformPivotX
 * @attr ref android.R.styleable#View_transformPivotY
 * @attr ref android.R.styleable#View_translationX
 * @attr ref android.R.styleable#View_translationY
 * @attr ref android.R.styleable#View_translationZ
 * @attr ref android.R.styleable#View_visibility
 * @attr ref android.R.styleable#View_theme
 *
 * @see android.view.ViewGroup
 */
@UiThread
public class View implements Drawable.Callback, KeyEvent.Callback,
        AccessibilityEventSource {
    private static final boolean DBG = false;

    /** @hide */
    public static boolean DEBUG_DRAW = false;

    /**
     * The logging tag used by this class with android.util.Log.
     */
    protected static final String VIEW_LOG_TAG = "View";

    /**
     * When set to true, apps will draw debugging information about their layouts.
     *
     * @hide
     */
    public static final String DEBUG_LAYOUT_PROPERTY = "debug.layout";

    /**
     * When set to true, this view will save its attribute data.
     *
     * @hide
     */
    public static boolean mDebugViewAttributes = false;

    /**
     * Used to mark a View that has no ID.
     */
    public static final int NO_ID = -1;

    /**
     * Last ID that is given to Views that are no part of activities.
     *
     * {@hide}
     */
    public static final int LAST_APP_AUTOFILL_ID = Integer.MAX_VALUE / 2;

    /**
     * Attribute to find the autofilled highlight
     *
     * @see #getAutofilledDrawable()
     */
    private static final int[] AUTOFILL_HIGHLIGHT_ATTR =
            new int[]{android.R.attr.autofilledHighlight};
```

```
/**
 * Signals that compatibility booleans have been initialized according to
 * target SDK versions.
 */
private static boolean sCompatibilityDone = false;

/**
 * Use the old (broken) way of building MeasureSpecs.
 */
private static boolean sUseBrokenMakeMeasureSpec = false;

/**
 * Always return a size of 0 for MeasureSpec values with a mode of UNSPECIFIED
 */
static boolean sUseZeroUnspecifiedMeasureSpec = false;

/**
 * Ignore any optimizations using the measure cache.
 */
private static boolean sIgnoreMeasureCache = false;

/**
 * Ignore an optimization that skips unnecessary EXACTLY layout passes.
 */
private static boolean sAlwaysRemeasureExactly = false;

/**
 * Relax constraints around whether setLayoutParams() must be called after
 * modifying the layout params.
 */
private static boolean sLayoutParamsAlwaysChanged = false;

/**
 * Allow setForeground/setBackground to be called (and ignored) on a textureview,
 * without throwing
 */
static boolean sTextureViewIgnoresDrawableSetters = false;

/**
 * Prior to N, some ViewGroups would not convert LayoutParams properly even though both extend
 * MarginLayoutParams. For instance, converting LinearLayout.LayoutParams to
 * RelativeLayout.LayoutParams would lose margin information. This is fixed on N but target API
 * check is implemented for backwards compatibility.
 *
 * {@hide}
 */
protected static boolean sPreserveMarginParamsInLayoutParamConversion;

/**
 * Prior to N, when drag enters into child of a view that has already received an
 * ACTION_DRAG_ENTERED event, the parent doesn't get a ACTION_DRAG_EXITED event.
 * ACTION_DRAG_LOCATION and ACTION_DROP were delivered to the parent of a view that returned
 * false from its event handler for these events.
 * Starting from N, the parent will get ACTION_DRAG_EXITED event before the child gets its
 * ACTION_DRAG_ENTERED. ACTION_DRAG_LOCATION and ACTION_DROP are never propagated to the parent.
 * sCascadedDragDrop is true for pre-N apps for backwards compatibility implementation.
 */
static boolean sCascadedDragDrop;

/**
 * Prior to O, auto-focusable didn't exist and widgets such as ListView use hasFocusable
 * to determine things like whether or not to permit item click events. We can't break
 * apps that do this just because more things (clickable things) are now auto-focusable
 * and they would get different results, so give old behavior to old apps.
 */
static boolean sHasFocusableExcludeAutoFocusable;

/**
 * Prior to O, auto-focusable didn't exist and views marked as clickable weren't implicitly
 * made focusable by default. As a result, apps could (incorrectly) change the clickable
 * setting of views off the UI thread. Now that clickable can effect the focusable state,
 * changing the clickable attribute off the UI thread will cause an exception (since changing
 * the focusable state checks). In order to prevent apps from crashing, we will handle this
 * specific case and just not notify parents on new focusables resulting from marking views
 * clickable from outside the UI thread.
 */
private static boolean sAutoFocusableOffUIThreadWontNotifyParents;

/** @hide */
@IntDef({NOT_FOCUSABLE, FOCUSABLE, FOCUSABLE_AUTO})
@Retention(RetentionPolicy.SOURCE)
public @interface Focusable {}
```

```java
/**
 * This view does not want keystrokes.
 * <p>
 * Use with {@link #setFocusable(int)} and <a href="#attr_android:focusable">{@code
 * android:focusable}.
 */
public static final int NOT_FOCUSABLE = 0x00000000;

/**
 * This view wants keystrokes.
 * <p>
 * Use with {@link #setFocusable(int)} and <a href="#attr_android:focusable">{@code
 * android:focusable}.
 */
public static final int FOCUSABLE = 0x00000001;

/**
 * This view determines focusability automatically. This is the default.
 * <p>
 * Use with {@link #setFocusable(int)} and <a href="#attr_android:focusable">{@code
 * android:focusable}.
 */
public static final int FOCUSABLE_AUTO = 0x00000010;

/**
 * Mask for use with setFlags indicating bits used for focus.
 */
private static final int FOCUSABLE_MASK = 0x00000011;

/**
 * This view will adjust its padding to fit sytem windows (e.g. status bar)
 */
private static final int FITS_SYSTEM_WINDOWS = 0x00000002;

/** @hide */
@IntDef({VISIBLE, INVISIBLE, GONE})
@Retention(RetentionPolicy.SOURCE)
public @interface Visibility {}

/**
 * This view is visible.
 * Use with {@link #setVisibility} and <a href="#attr_android:visibility">{@code
 * android:visibility}.
 */
public static final int VISIBLE = 0x00000000;

/**
 * This view is invisible, but it still takes up space for layout purposes.
 * Use with {@link #setVisibility} and <a href="#attr_android:visibility">{@code
 * android:visibility}.
 */
public static final int INVISIBLE = 0x00000004;

/**
 * This view is invisible, and it doesn't take any space for layout
 * purposes. Use with {@link #setVisibility} and <a href="#attr_android:visibility">{@code
 * android:visibility}.
 */
public static final int GONE = 0x00000008;

/**
 * Mask for use with setFlags indicating bits used for visibility.
 * {@hide}
 */
static final int VISIBILITY_MASK = 0x0000000C;

private static final int[] VISIBILITY_FLAGS = {VISIBLE, INVISIBLE, GONE};

/**
 * Hint indicating that this view can be autofilled with an email address.
 *
 * <p>Can be used with either {@link #setAutofillHints(String[])} or
 * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
 * value should be <code>{@value #AUTOFILL_HINT_EMAIL_ADDRESS}</code>).
 *
 * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
 */
public static final String AUTOFILL_HINT_EMAIL_ADDRESS = "emailAddress";

/**
 * Hint indicating that this view can be autofilled with a user's real name.
```

```
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_NAME}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_NAME = "name";

    /**
     * Hint indicating that this view can be autofilled with a username.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_USERNAME}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_USERNAME = "username";

    /**
     * Hint indicating that this view can be autofilled with a password.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_PASSWORD}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_PASSWORD = "password";

    /**
     * Hint indicating that this view can be autofilled with a phone number.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_PHONE}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_PHONE = "phone";

    /**
     * Hint indicating that this view can be autofilled with a postal address.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_POSTAL_ADDRESS}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_POSTAL_ADDRESS = "postalAddress";

    /**
     * Hint indicating that this view can be autofilled with a postal code.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_POSTAL_CODE}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_POSTAL_CODE = "postalCode";

    /**
     * Hint indicating that this view can be autofilled with a credit card number.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_NUMBER}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_NUMBER = "creditCardNumber";

    /**
     * Hint indicating that this view can be autofilled with a credit card security code.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_SECURITY_CODE}</code>).
     *
```

```
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_SECURITY_CODE = "creditCardSecurityCode";

    /**
     * Hint indicating that this view can be autofilled with a credit card expiration date.
     *
     * <p>It should be used when the credit card expiration date is represented by just one view;
     * if it is represented by more than one (for example, one view for the month and another view
     * for the year), then each of these views should use the hint specific for the unit
     * ({@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DAY},
     * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_MONTH},
     * or {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_YEAR}).
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DATE}</code>).
     *
     * <p>When annotating a view with this hint, it's recommended to use a date autofill value to
     * avoid ambiguity when the autofill service provides a value for it. To understand why a
     * value can be ambiguous, consider "April of 2020", which could be represented as either of
     * the following options:
     *
     * <ul>
     *   <li>{@code "04/2020"}
     *   <li>{@code "4/2020"}
     *   <li>{@code "2020/04"}
     *   <li>{@code "2020/4"}
     *   <li>{@code "April/2020"}
     *   <li>{@code "Apr/2020"}
     * </ul>
     *
     * <p>You define a date autofill value for the view by overriding the following methods:
     *
     * <ol>
     *   <li>{@link #getAutofillType()} to return {@link #AUTOFILL_TYPE_DATE}.
     *   <li>{@link #getAutofillValue()} to return a
     *       {@link AutofillValue#forDate(long) date autofillvalue}.
     *   <li>{@link #autofill(AutofillValue)} to expect a data autofillvalue.
     * </ol>
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DATE =
            "creditCardExpirationDate";

    /**
     * Hint indicating that this view can be autofilled with a credit card expiration month.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_MONTH}</code>).
     *
     * <p>When annotating a view with this hint, it's recommended to use a text autofill value
     * whose value is the numerical representation of the month, starting on {@code 1} to avoid
     * ambiguity when the autofill service provides a value for it. To understand why a
     * value can be ambiguous, consider "January", which could be represented as either of
     *
     * <ul>
     *   <li>{@code "1"}: recommended way.
     *   <li>{@code "0"}: if following the {@link Calendar#MONTH} convention.
     *   <li>{@code "January"}: full name, in English.
     *   <li>{@code "jan"}: abbreviated name, in English.
     *   <li>{@code "Janeiro"}: full name, in another language.
     * </ul>
     *
     * <p>Another recommended approach is to use a date autofill value - see
     * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DATE} for more details.
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_MONTH =
            "creditCardExpirationMonth";

    /**
     * Hint indicating that this view can be autofilled with a credit card expiration year.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_YEAR}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
```

```
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_YEAR =
            "creditCardExpirationYear";

    /**
     * Hint indicating that this view can be autofilled with a credit card expiration day.
     *
     * <p>Can be used with either {@link #setAutofillHints(String[])} or
     * <a href="#attr_android:autofillHint"> {@code android:autofillHint}</a> (in which case the
     * value should be <code>{@value #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DAY}</code>).
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about autofill hints.
     */
    public static final String AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DAY = "creditCardExpirationDay";

    /**
     * Hints for the autofill services that describes the content of the view.
     */
    private @Nullable String[] mAutofillHints;

    /**
     * Autofill id, lazily created on calls to {@link #getAutofillId()}.
     */
    private AutofillId mAutofillId;

    /** @hide */
    @IntDef({
            AUTOFILL_TYPE_NONE,
            AUTOFILL_TYPE_TEXT,
            AUTOFILL_TYPE_TOGGLE,
            AUTOFILL_TYPE_LIST,
            AUTOFILL_TYPE_DATE
    })
    @Retention(RetentionPolicy.SOURCE)
    public @interface AutofillType {}

    /**
     * Autofill type for views that cannot be autofilled.
     *
     * <p>Typically used when the view is read-only; for example, a text label.
     *
     * @see #getAutofillType()
     */
    public static final int AUTOFILL_TYPE_NONE = 0;

    /**
     * Autofill type for a text field, which is filled by a {@link CharSequence}.
     *
     * <p>{@link AutofillValue} instances for autofilling a {@link View} can be obtained through
     * {@link AutofillValue#forText(CharSequence)}, and the value passed to autofill a
     * {@link View} can be fetched through {@link AutofillValue#getTextValue()}.
     *
     * @see #getAutofillType()
     */
    public static final int AUTOFILL_TYPE_TEXT = 1;

    /**
     * Autofill type for a togglable field, which is filled by a {@code boolean}.
     *
     * <p>{@link AutofillValue} instances for autofilling a {@link View} can be obtained through
     * {@link AutofillValue#forToggle(boolean)}, and the value passed to autofill a
     * {@link View} can be fetched through {@link AutofillValue#getToggleValue()}.
     *
     * @see #getAutofillType()
     */
    public static final int AUTOFILL_TYPE_TOGGLE = 2;

    /**
     * Autofill type for a selection list field, which is filled by an {@code int}
     * representing the element index inside the list (starting at {@code 0}).
     *
     * <p>{@link AutofillValue} instances for autofilling a {@link View} can be obtained through
     * {@link AutofillValue#forList(int)}, and the value passed to autofill a
     * {@link View} can be fetched through {@link AutofillValue#getListValue()}.
     *
     * <p>The available options in the selection list are typically provided by
     * {@link android.app.assist.AssistStructure.ViewNode#getAutofillOptions()}.
     *
     * @see #getAutofillType()
     */
    public static final int AUTOFILL_TYPE_LIST = 3;
```

```
/**
 * Autofill type for a field that contains a date, which is represented by a long representing
 * the number of milliseconds since the standard base time known as "the epoch", namely
 * January 1, 1970, 00:00:00 GMT (see {@link java.util.Date#getTime()}.
 *
 * <p>{@link AutofillValue} instances for autofilling a {@link View} can be obtained through
 * {@link AutofillValue#forDate(long)}, and the values passed to
 * autofill a {@link View} can be fetched through {@link AutofillValue#getDateValue()}.
 *
 * @see #getAutofillType()
 */
public static final int AUTOFILL_TYPE_DATE = 4;

/** @hide */
@IntDef({
        IMPORTANT_FOR_AUTOFILL_AUTO,
        IMPORTANT_FOR_AUTOFILL_YES,
        IMPORTANT_FOR_AUTOFILL_NO,
        IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS,
        IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS
})
@Retention(RetentionPolicy.SOURCE)
public @interface AutofillImportance {}

/**
 * Automatically determine whether a view is important for autofill.
 *
 * @see #isImportantForAutofill()
 * @see #setImportantForAutofill(int)
 */
public static final int IMPORTANT_FOR_AUTOFILL_AUTO = 0x0;

/**
 * The view is important for autofill, and its children (if any) will be traversed.
 *
 * @see #isImportantForAutofill()
 * @see #setImportantForAutofill(int)
 */
public static final int IMPORTANT_FOR_AUTOFILL_YES = 0x1;

/**
 * The view is not important for autofill, but its children (if any) will be traversed.
 *
 * @see #isImportantForAutofill()
 * @see #setImportantForAutofill(int)
 */
public static final int IMPORTANT_FOR_AUTOFILL_NO = 0x2;

/**
 * The view is important for autofill, but its children (if any) will not be traversed.
 *
 * @see #isImportantForAutofill()
 * @see #setImportantForAutofill(int)
 */
public static final int IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS = 0x4;

/**
 * The view is not important for autofill, and its children (if any) will not be traversed.
 *
 * @see #isImportantForAutofill()
 * @see #setImportantForAutofill(int)
 */
public static final int IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS = 0x8;

/** @hide */
@IntDef(
        flag = true,
        value = {AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS})
@Retention(RetentionPolicy.SOURCE)
public @interface AutofillFlags {}

/**
 * Flag requesting you to add views that are marked as not important for autofill
 * (see {@link #setImportantForAutofill(int)}) to a {@link ViewStructure}.
 */
public static final int AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS = 0x1;

/**
 * This view is enabled. Interpretation varies by subclass.
 * Use with ENABLED_MASK when calling setFlags.
 * {@hide}
```

```
     */
    static final int ENABLED = 0x00000000;

    /**
     * This view is disabled. Interpretation varies by subclass.
     * Use with ENABLED_MASK when calling setFlags.
     * {@hide}
     */
    static final int DISABLED = 0x00000020;

   /**
    * Mask for use with setFlags indicating bits used for indicating whether
    * this view is enabled
    * {@hide}
    */
    static final int ENABLED_MASK = 0x00000020;

    /**
     * This view won't draw. {@link #onDraw(android.graphics.Canvas)} won't be
     * called and further optimizations will be performed. It is okay to have
     * this flag set and a background. Use with DRAW_MASK when calling setFlags.
     * {@hide}
     */
    static final int WILL_NOT_DRAW = 0x00000080;

    /**
     * Mask for use with setFlags indicating bits used for indicating whether
     * this view is will draw
     * {@hide}
     */
    static final int DRAW_MASK = 0x00000080;

    /**
     * <p>This view doesn't show scrollbars.</p>
     * {@hide}
     */
    static final int SCROLLBARS_NONE = 0x00000000;

    /**
     * <p>This view shows horizontal scrollbars.</p>
     * {@hide}
     */
    static final int SCROLLBARS_HORIZONTAL = 0x00000100;

    /**
     * <p>This view shows vertical scrollbars.</p>
     * {@hide}
     */
    static final int SCROLLBARS_VERTICAL = 0x00000200;

    /**
     * <p>Mask for use with setFlags indicating bits used for indicating which
     * scrollbars are enabled.</p>
     * {@hide}
     */
    static final int SCROLLBARS_MASK = 0x00000300;

    /**
     * Indicates that the view should filter touches when its window is obscured.
     * Refer to the class comments for more information about this security feature.
     * {@hide}
     */
    static final int FILTER_TOUCHES_WHEN_OBSCURED = 0x00000400;

    /**
     * Set for framework elements that use FITS_SYSTEM_WINDOWS, to indicate
     * that they are optional and should be skipped if the window has
     * requested system UI flags that ignore those insets for layout.
     */
    static final int OPTIONAL_FITS_SYSTEM_WINDOWS = 0x00000800;

    /**
     * <p>This view doesn't show fading edges.</p>
     * {@hide}
     */
    static final int FADING_EDGE_NONE = 0x00000000;

    /**
     * <p>This view shows horizontal fading edges.</p>
     * {@hide}
     */
    static final int FADING_EDGE_HORIZONTAL = 0x00001000;
```

```java
/**
 * <p>This view shows vertical fading edges.</p>
 * {@hide}
 */
static final int FADING_EDGE_VERTICAL = 0x00002000;

/**
 * <p>Mask for use with setFlags indicating bits used for indicating which
 * fading edges are enabled.</p>
 * {@hide}
 */
static final int FADING_EDGE_MASK = 0x00003000;

/**
 * <p>Indicates this view can be clicked. When clickable, a View reacts
 * to clicks by notifying the OnClickListener.<p>
 * {@hide}
 */
static final int CLICKABLE = 0x00004000;

/**
 * <p>Indicates this view is caching its drawing into a bitmap.</p>
 * {@hide}
 */
static final int DRAWING_CACHE_ENABLED = 0x00008000;

/**
 * <p>Indicates that no icicle should be saved for this view.<p>
 * {@hide}
 */
static final int SAVE_DISABLED = 0x000010000;

/**
 * <p>Mask for use with setFlags indicating bits used for the saveEnabled
 * property.</p>
 * {@hide}
 */
static final int SAVE_DISABLED_MASK = 0x000010000;

/**
 * <p>Indicates that no drawing cache should ever be created for this view.<p>
 * {@hide}
 */
static final int WILL_NOT_CACHE_DRAWING = 0x000020000;

/**
 * <p>Indicates this view can take / keep focus when int touch mode.</p>
 * {@hide}
 */
static final int FOCUSABLE_IN_TOUCH_MODE = 0x00040000;

/** @hide */
@Retention(RetentionPolicy.SOURCE)
@IntDef({DRAWING_CACHE_QUALITY_LOW, DRAWING_CACHE_QUALITY_HIGH, DRAWING_CACHE_QUALITY_AUTO})
public @interface DrawingCacheQuality {}

/**
 * <p>Enables low quality mode for the drawing cache.</p>
 */
public static final int DRAWING_CACHE_QUALITY_LOW = 0x00080000;

/**
 * <p>Enables high quality mode for the drawing cache.</p>
 */
public static final int DRAWING_CACHE_QUALITY_HIGH = 0x00100000;

/**
 * <p>Enables automatic quality mode for the drawing cache.</p>
 */
public static final int DRAWING_CACHE_QUALITY_AUTO = 0x00000000;

private static final int[] DRAWING_CACHE_QUALITY_FLAGS = {
        DRAWING_CACHE_QUALITY_AUTO, DRAWING_CACHE_QUALITY_LOW, DRAWING_CACHE_QUALITY_HIGH
};

/**
 * <p>Mask for use with setFlags indicating bits used for the cache
 * quality property.</p>
 * {@hide}
 */
static final int DRAWING_CACHE_QUALITY_MASK = 0x00180000;
```

```java
/**
 * <p>
 * Indicates this view can be long clicked. When long clickable, a View
 * reacts to long clicks by notifying the OnLongClickListener or showing a
 * context menu.
 * </p>
 * {@hide}
 */
static final int LONG_CLICKABLE = 0x00200000;

/**
 * <p>Indicates that this view gets its drawable states from its direct parent
 * and ignores its original internal states.</p>
 *
 * @hide
 */
static final int DUPLICATE_PARENT_STATE = 0x00400000;

/**
 * <p>
 * Indicates this view can be context clicked. When context clickable, a View reacts to a
 * context click (e.g. a primary stylus button press or right mouse click) by notifying the
 * OnContextClickListener.
 * </p>
 * {@hide}
 */
static final int CONTEXT_CLICKABLE = 0x00800000;


/** @hide */
@IntDef({
    SCROLLBARS_INSIDE_OVERLAY,
    SCROLLBARS_INSIDE_INSET,
    SCROLLBARS_OUTSIDE_OVERLAY,
    SCROLLBARS_OUTSIDE_INSET
})
@Retention(RetentionPolicy.SOURCE)
public @interface ScrollBarStyle {}

/**
 * The scrollbar style to display the scrollbars inside the content area,
 * without increasing the padding. The scrollbars will be overlaid with
 * translucency on the view's content.
 */
public static final int SCROLLBARS_INSIDE_OVERLAY = 0;

/**
 * The scrollbar style to display the scrollbars inside the padded area,
 * increasing the padding of the view. The scrollbars will not overlap the
 * content area of the view.
 */
public static final int SCROLLBARS_INSIDE_INSET = 0x01000000;

/**
 * The scrollbar style to display the scrollbars at the edge of the view,
 * without increasing the padding. The scrollbars will be overlaid with
 * translucency.
 */
public static final int SCROLLBARS_OUTSIDE_OVERLAY = 0x02000000;

/**
 * The scrollbar style to display the scrollbars at the edge of the view,
 * increasing the padding of the view. The scrollbars will only overlap the
 * background, if any.
 */
public static final int SCROLLBARS_OUTSIDE_INSET = 0x03000000;

/**
 * Mask to check if the scrollbar style is overlay or inset.
 * {@hide}
 */
static final int SCROLLBARS_INSET_MASK = 0x01000000;

/**
 * Mask to check if the scrollbar style is inside or outside.
 * {@hide}
 */
static final int SCROLLBARS_OUTSIDE_MASK = 0x02000000;

/**
 * Mask for scrollbar style.
```

```
     * {@hide}
     */
    static final int SCROLLBARS_STYLE_MASK = 0x03000000;

    /**
     * View flag indicating that the screen should remain on while the
     * window containing this view is visible to the user.  This effectively
     * takes care of automatically setting the WindowManager's
     * {@link WindowManager.LayoutParams#FLAG_KEEP_SCREEN_ON}.
     */
    public static final int KEEP_SCREEN_ON = 0x04000000;

    /**
     * View flag indicating whether this view should have sound effects enabled
     * for events such as clicking and touching.
     */
    public static final int SOUND_EFFECTS_ENABLED = 0x08000000;

    /**
     * View flag indicating whether this view should have haptic feedback
     * enabled for events such as long presses.
     */
    public static final int HAPTIC_FEEDBACK_ENABLED = 0x10000000;

    /**
     * <p>Indicates that the view hierarchy should stop saving state when
     * it reaches this view.  If state saving is initiated immediately at
     * the view, it will be allowed.
     * {@hide}
     */
    static final int PARENT_SAVE_DISABLED = 0x20000000;

    /**
     * <p>Mask for use with setFlags indicating bits used for PARENT_SAVE_DISABLED.</p>
     * {@hide}
     */
    static final int PARENT_SAVE_DISABLED_MASK = 0x20000000;

    private static Paint sDebugPaint;

    /**
     * <p>Indicates this view can display a tooltip on hover or long press.</p>
     * {@hide}
     */
    static final int TOOLTIP = 0x40000000;

    /** @hide */
    @IntDef(flag = true,
            value = {
                FOCUSABLES_ALL,
                FOCUSABLES_TOUCH_MODE
            })
    @Retention(RetentionPolicy.SOURCE)
    public @interface FocusableMode {}

    /**
     * View flag indicating whether {@link #addFocusables(ArrayList, int, int)}
     * should add all focusable Views regardless if they are focusable in touch mode.
     */
    public static final int FOCUSABLES_ALL = 0x00000000;

    /**
     * View flag indicating whether {@link #addFocusables(ArrayList, int, int)}
     * should add only Views focusable in touch mode.
     */
    public static final int FOCUSABLES_TOUCH_MODE = 0x00000001;

    /** @hide */
    @IntDef({
            FOCUS_BACKWARD,
            FOCUS_FORWARD,
            FOCUS_LEFT,
            FOCUS_UP,
            FOCUS_RIGHT,
            FOCUS_DOWN
    })
    @Retention(RetentionPolicy.SOURCE)
    public @interface FocusDirection {}

    /** @hide */
    @IntDef({
            FOCUS_LEFT,
```

```
        FOCUS_UP,
        FOCUS_RIGHT,
        FOCUS_DOWN
})
@Retention(RetentionPolicy.SOURCE)
public @interface FocusRealDirection {} // Like @FocusDirection, but without forward/backward

/**
 * Use with {@link #focusSearch(int)}. Move focus to the previous selectable
 * item.
 */
public static final int FOCUS_BACKWARD = 0x00000001;

/**
 * Use with {@link #focusSearch(int)}. Move focus to the next selectable
 * item.
 */
public static final int FOCUS_FORWARD = 0x00000002;

/**
 * Use with {@link #focusSearch(int)}. Move focus to the left.
 */
public static final int FOCUS_LEFT = 0x00000011;

/**
 * Use with {@link #focusSearch(int)}. Move focus up.
 */
public static final int FOCUS_UP = 0x00000021;

/**
 * Use with {@link #focusSearch(int)}. Move focus to the right.
 */
public static final int FOCUS_RIGHT = 0x00000042;

/**
 * Use with {@link #focusSearch(int)}. Move focus down.
 */
public static final int FOCUS_DOWN = 0x00000082;

/**
 * Bits of {@link #getMeasuredWidthAndState()} and
 * {@link #getMeasuredWidthAndState()} that provide the actual measured size.
 */
public static final int MEASURED_SIZE_MASK = 0x00ffffff;

/**
 * Bits of {@link #getMeasuredWidthAndState()} and
 * {@link #getMeasuredWidthAndState()} that provide the additional state bits.
 */
public static final int MEASURED_STATE_MASK = 0xff000000;

/**
 * Bit shift of {@link #MEASURED_STATE_MASK} to get to the height bits
 * for functions that combine both width and height into a single int,
 * such as {@link #getMeasuredState()} and the childState argument of
 * {@link #resolveSizeAndState(int, int, int)}.
 */
public static final int MEASURED_HEIGHT_STATE_SHIFT = 16;

/**
 * Bit of {@link #getMeasuredWidthAndState()} and
 * {@link #getMeasuredWidthAndState()} that indicates the measured size
 * is smaller that the space the view would like to have.
 */
public static final int MEASURED_STATE_TOO_SMALL = 0x01000000;

/**
 * Base View state sets
 */
// Singles
/**
 * Indicates the view has no states set. States are used with
 * {@link android.graphics.drawable.Drawable} to change the drawing of the
 * view depending on its state.
 *
 * @see android.graphics.drawable.Drawable
 * @see #getDrawableState()
 */
protected static final int[] EMPTY_STATE_SET;
/**
 * Indicates the view is enabled. States are used with
 * {@link android.graphics.drawable.Drawable} to change the drawing of the
```

```
     * view depending on its state.
     *
     * @see android.graphics.drawable.Drawable
     * @see #getDrawableState()
     */
    protected static final int[] ENABLED_STATE_SET;
    /**
     * Indicates the view is focused. States are used with
     * {@link android.graphics.drawable.Drawable} to change the drawing of the
     * view depending on its state.
     *
     * @see android.graphics.drawable.Drawable
     * @see #getDrawableState()
     */
    protected static final int[] FOCUSED_STATE_SET;
    /**
     * Indicates the view is selected. States are used with
     * {@link android.graphics.drawable.Drawable} to change the drawing of the
     * view depending on its state.
     *
     * @see android.graphics.drawable.Drawable
     * @see #getDrawableState()
     */
    protected static final int[] SELECTED_STATE_SET;
    /**
     * Indicates the view is pressed. States are used with
     * {@link android.graphics.drawable.Drawable} to change the drawing of the
     * view depending on its state.
     *
     * @see android.graphics.drawable.Drawable
     * @see #getDrawableState()
     */
    protected static final int[] PRESSED_STATE_SET;
    /**
     * Indicates the view's window has focus. States are used with
     * {@link android.graphics.drawable.Drawable} to change the drawing of the
     * view depending on its state.
     *
     * @see android.graphics.drawable.Drawable
     * @see #getDrawableState()
     */
    protected static final int[] WINDOW_FOCUSED_STATE_SET;
    // Doubles
    /**
     * Indicates the view is enabled and has the focus.
     *
     * @see #ENABLED_STATE_SET
     * @see #FOCUSED_STATE_SET
     */
    protected static final int[] ENABLED_FOCUSED_STATE_SET;
    /**
     * Indicates the view is enabled and selected.
     *
     * @see #ENABLED_STATE_SET
     * @see #SELECTED_STATE_SET
     */
    protected static final int[] ENABLED_SELECTED_STATE_SET;
    /**
     * Indicates the view is enabled and that its window has focus.
     *
     * @see #ENABLED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] ENABLED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is focused and selected.
     *
     * @see #FOCUSED_STATE_SET
     * @see #SELECTED_STATE_SET
     */
    protected static final int[] FOCUSED_SELECTED_STATE_SET;
    /**
     * Indicates the view has the focus and that its window has the focus.
     *
     * @see #FOCUSED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] FOCUSED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is selected and that its window has the focus.
     *
     * @see #SELECTED_STATE_SET
```

```
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] SELECTED_WINDOW_FOCUSED_STATE_SET;
    // Triples
    /**
     * Indicates the view is enabled, focused and selected.
     *
     * @see #ENABLED_STATE_SET
     * @see #FOCUSED_STATE_SET
     * @see #SELECTED_STATE_SET
     */
    protected static final int[] ENABLED_FOCUSED_SELECTED_STATE_SET;
    /**
     * Indicates the view is enabled, focused and its window has the focus.
     *
     * @see #ENABLED_STATE_SET
     * @see #FOCUSED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] ENABLED_FOCUSED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is enabled, selected and its window has the focus.
     *
     * @see #ENABLED_STATE_SET
     * @see #SELECTED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] ENABLED_SELECTED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is focused, selected and its window has the focus.
     *
     * @see #FOCUSED_STATE_SET
     * @see #SELECTED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is enabled, focused, selected and its window
     * has the focus.
     *
     * @see #ENABLED_STATE_SET
     * @see #FOCUSED_STATE_SET
     * @see #SELECTED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] ENABLED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is pressed and its window has the focus.
     *
     * @see #PRESSED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] PRESSED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is pressed and selected.
     *
     * @see #PRESSED_STATE_SET
     * @see #SELECTED_STATE_SET
     */
    protected static final int[] PRESSED_SELECTED_STATE_SET;
    /**
     * Indicates the view is pressed, selected and its window has the focus.
     *
     * @see #PRESSED_STATE_SET
     * @see #SELECTED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] PRESSED_SELECTED_WINDOW_FOCUSED_STATE_SET;
    /**
     * Indicates the view is pressed and focused.
     *
     * @see #PRESSED_STATE_SET
     * @see #FOCUSED_STATE_SET
     */
    protected static final int[] PRESSED_FOCUSED_STATE_SET;
    /**
     * Indicates the view is pressed, focused and its window has the focus.
     *
     * @see #PRESSED_STATE_SET
     * @see #FOCUSED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
```

```
protected static final int[] PRESSED_FOCUSED_WINDOW_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed, focused and selected.
 *
 * @see #PRESSED_STATE_SET
 * @see #SELECTED_STATE_SET
 * @see #FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_FOCUSED_SELECTED_STATE_SET;
/**
 * Indicates the view is pressed, focused, selected and its window has the focus.
 *
 * @see #PRESSED_STATE_SET
 * @see #FOCUSED_STATE_SET
 * @see #SELECTED_STATE_SET
 * @see #WINDOW_FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed and enabled.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_STATE_SET;
/**
 * Indicates the view is pressed, enabled and its window has the focus.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #WINDOW_FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_WINDOW_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed, enabled and selected.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #SELECTED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_SELECTED_STATE_SET;
/**
 * Indicates the view is pressed, enabled, selected and its window has the
 * focus.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #SELECTED_STATE_SET
 * @see #WINDOW_FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_SELECTED_WINDOW_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed, enabled and focused.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed, enabled, focused and its window has the
 * focus.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #FOCUSED_STATE_SET
 * @see #WINDOW_FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_FOCUSED_WINDOW_FOCUSED_STATE_SET;
/**
 * Indicates the view is pressed, enabled, focused and selected.
 *
 * @see #PRESSED_STATE_SET
 * @see #ENABLED_STATE_SET
 * @see #SELECTED_STATE_SET
 * @see #FOCUSED_STATE_SET
 */
protected static final int[] PRESSED_ENABLED_FOCUSED_SELECTED_STATE_SET;
/**
 * Indicates the view is pressed, enabled, focused, selected and its window
 * has the focus.
 *
```

```
     * @see #PRESSED_STATE_SET
     * @see #ENABLED_STATE_SET
     * @see #SELECTED_STATE_SET
     * @see #FOCUSED_STATE_SET
     * @see #WINDOW_FOCUSED_STATE_SET
     */
    protected static final int[] PRESSED_ENABLED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET;

    static {
        EMPTY_STATE_SET = StateSet.get(0);

        WINDOW_FOCUSED_STATE_SET = StateSet.get(StateSet.VIEW_STATE_WINDOW_FOCUSED);

        SELECTED_STATE_SET = StateSet.get(StateSet.VIEW_STATE_SELECTED);
        SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED);

        FOCUSED_STATE_SET = StateSet.get(StateSet.VIEW_STATE_FOCUSED);
        FOCUSED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_FOCUSED);
        FOCUSED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_FOCUSED);
        FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_FOCUSED);

        ENABLED_STATE_SET = StateSet.get(StateSet.VIEW_STATE_ENABLED);
        ENABLED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_ENABLED);
        ENABLED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_ENABLED);
        ENABLED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_ENABLED);
        ENABLED_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_FOCUSED | StateSet.VIEW_STATE_ENABLED);
        ENABLED_FOCUSED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_FOCUSED
                        | StateSet.VIEW_STATE_ENABLED);
        ENABLED_FOCUSED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_FOCUSED
                        | StateSet.VIEW_STATE_ENABLED);
        ENABLED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_FOCUSED| StateSet.VIEW_STATE_ENABLED);

        PRESSED_STATE_SET = StateSet.get(StateSet.VIEW_STATE_PRESSED);
        PRESSED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_FOCUSED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_FOCUSED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_FOCUSED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_FOCUSED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_FOCUSED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_FOCUSED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_ENABLED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_ENABLED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_SELECTED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_ENABLED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                        | StateSet.VIEW_STATE_ENABLED | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_FOCUSED | StateSet.VIEW_STATE_ENABLED
                        | StateSet.VIEW_STATE_PRESSED);
        PRESSED_ENABLED_FOCUSED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
                StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_FOCUSED
                        | StateSet.VIEW_STATE_ENABLED | StateSet.VIEW_STATE_PRESSED);
```

```
    PRESSED_ENABLED_FOCUSED_SELECTED_STATE_SET = StateSet.get(
            StateSet.VIEW_STATE_SELECTED | StateSet.VIEW_STATE_FOCUSED
                    | StateSet.VIEW_STATE_ENABLED | StateSet.VIEW_STATE_PRESSED);
    PRESSED_ENABLED_FOCUSED_SELECTED_WINDOW_FOCUSED_STATE_SET = StateSet.get(
            StateSet.VIEW_STATE_WINDOW_FOCUSED | StateSet.VIEW_STATE_SELECTED
                    | StateSet.VIEW_STATE_FOCUSED| StateSet.VIEW_STATE_ENABLED
                    | StateSet.VIEW_STATE_PRESSED);
}

/**
 * Accessibility event types that are dispatched for text population.
 */
private static final int POPULATING_ACCESSIBILITY_EVENT_TYPES =
        AccessibilityEvent.TYPE_VIEW_CLICKED
        | AccessibilityEvent.TYPE_VIEW_LONG_CLICKED
        | AccessibilityEvent.TYPE_VIEW_SELECTED
        | AccessibilityEvent.TYPE_VIEW_FOCUSED
        | AccessibilityEvent.TYPE_WINDOW_STATE_CHANGED
        | AccessibilityEvent.TYPE_VIEW_HOVER_ENTER
        | AccessibilityEvent.TYPE_VIEW_HOVER_EXIT
        | AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED
        | AccessibilityEvent.TYPE_VIEW_TEXT_SELECTION_CHANGED
        | AccessibilityEvent.TYPE_VIEW_ACCESSIBILITY_FOCUSED
        | AccessibilityEvent.TYPE_VIEW_TEXT_TRAVERSED_AT_MOVEMENT_GRANULARITY;

static final int DEBUG_CORNERS_COLOR = Color.rgb(63, 127, 255);

static final int DEBUG_CORNERS_SIZE_DIP = 8;

/**
 * Temporary Rect currently for use in setBackground().  This will probably
 * be extended in the future to hold our own class with more than just
 * a Rect. :)
 */
static final ThreadLocal<Rect> sThreadLocal = new ThreadLocal<Rect>();

/**
 * Map used to store views' tags.
 */
private SparseArray<Object> mKeyedTags;

/**
 * The next available accessibility id.
 */
private static int sNextAccessibilityViewId;

/**
 * The animation currently associated with this view.
 * @hide
 */
protected Animation mCurrentAnimation = null;

/**
 * Width as measured during measure pass.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "measurement")
int mMeasuredWidth;

/**
 * Height as measured during measure pass.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "measurement")
int mMeasuredHeight;

/**
 * Flag to indicate that this view was marked INVALIDATED, or had its display list
 * invalidated, prior to the current drawing iteration. If true, the view must re-draw
 * its display list. This flag, used only when hw accelerated, allows us to clear the
 * flag while retaining this information until it's needed (at getDisplayList() time and
 * in drawChild(), when we decide to draw a view's children's display lists into our own).
 *
 * {@hide}
 */
boolean mRecreateDisplayList = false;

/**
 * The view's identifier.
 * {@hide}
 *
 * @see #setId(int)
```

```
     * @see #getId()
     */
    @IdRes
    @ViewDebug.ExportedProperty(resolveId = true)
    int mID = NO_ID;

    /** The ID of this view for autofill purposes.
     * <ul>
     *     <li>== {@link #NO_ID}: ID has not been assigned yet
     *     <li>&le; {@link #LAST_APP_AUTOFILL_ID}: View is not part of a activity. The ID is
     *                                             unique in the process. This might change
     *                                             over activity lifecycle events.
     *     <li>&gt; {@link #LAST_APP_AUTOFILL_ID}: View is part of a activity. The ID is
     *                                             unique in the activity. This stays the same
     *                                             over activity lifecycle events.
     */
    private int mAutofillViewId = NO_ID;

    // ID for accessibility purposes. This ID must be unique for every window
    private int mAccessibilityViewId = NO_ID;

    private int mAccessibilityCursorPosition = ACCESSIBILITY_CURSOR_POSITION_UNDEFINED;

    /**
     * The view's tag.
     * {@hide}
     *
     * @see #setTag(Object)
     * @see #getTag()
     */
    protected Object mTag = null;

    // for mPrivateFlags:
    /** {@hide} */
    static final int PFLAG_WANTS_FOCUS                 = 0x00000001;
    /** {@hide} */
    static final int PFLAG_FOCUSED                     = 0x00000002;
    /** {@hide} */
    static final int PFLAG_SELECTED                    = 0x00000004;
    /** {@hide} */
    static final int PFLAG_IS_ROOT_NAMESPACE           = 0x00000008;
    /** {@hide} */
    static final int PFLAG_HAS_BOUNDS                  = 0x00000010;
    /** {@hide} */
    static final int PFLAG_DRAWN                       = 0x00000020;
    /**
     * When this flag is set, this view is running an animation on behalf of its
     * children and should therefore not cancel invalidate requests, even if they
     * lie outside of this view's bounds.
     *
     * {@hide}
     */
    static final int PFLAG_DRAW_ANIMATION              = 0x00000040;
    /** {@hide} */
    static final int PFLAG_SKIP_DRAW                   = 0x00000080;
    /** {@hide} */
    static final int PFLAG_REQUEST_TRANSPARENT_REGIONS = 0x00000200;
    /** {@hide} */
    static final int PFLAG_DRAWABLE_STATE_DIRTY        = 0x00000400;
    /** {@hide} */
    static final int PFLAG_MEASURED_DIMENSION_SET      = 0x00000800;
    /** {@hide} */
    static final int PFLAG_FORCE_LAYOUT                = 0x00001000;
    /** {@hide} */
    static final int PFLAG_LAYOUT_REQUIRED             = 0x00002000;

    private static final int PFLAG_PRESSED             = 0x00004000;

    /** {@hide} */
    static final int PFLAG_DRAWING_CACHE_VALID         = 0x00008000;
    /**
     * Flag used to indicate that this view should be drawn once more (and only once
     * more) after its animation has completed.
     * {@hide}
     */
    static final int PFLAG_ANIMATION_STARTED           = 0x00010000;

    private static final int PFLAG_SAVE_STATE_CALLED   = 0x00020000;

    /**
     * Indicates that the View returned true when onSetAlpha() was called and that
     * the alpha must be restored.
```

```
     * {@hide}
     */
    static final int PFLAG_ALPHA_SET                    = 0x00040000;

    /**
     * Set by {@link #setScrollContainer(boolean)}.
     */
    static final int PFLAG_SCROLL_CONTAINER             = 0x00080000;

    /**
     * Set by {@link #setScrollContainer(boolean)}.
     */
    static final int PFLAG_SCROLL_CONTAINER_ADDED       = 0x00100000;

    /**
     * View flag indicating whether this view was invalidated (fully or partially.)
     *
     * @hide
     */
    static final int PFLAG_DIRTY                        = 0x00200000;

    /**
     * View flag indicating whether this view was invalidated by an opaque
     * invalidate request.
     *
     * @hide
     */
    static final int PFLAG_DIRTY_OPAQUE                 = 0x00400000;

    /**
     * Mask for {@link #PFLAG_DIRTY} and {@link #PFLAG_DIRTY_OPAQUE}.
     *
     * @hide
     */
    static final int PFLAG_DIRTY_MASK                   = 0x00600000;

    /**
     * Indicates whether the background is opaque.
     *
     * @hide
     */
    static final int PFLAG_OPAQUE_BACKGROUND            = 0x00800000;

    /**
     * Indicates whether the scrollbars are opaque.
     *
     * @hide
     */
    static final int PFLAG_OPAQUE_SCROLLBARS            = 0x01000000;

    /**
     * Indicates whether the view is opaque.
     *
     * @hide
     */
    static final int PFLAG_OPAQUE_MASK                  = 0x01800000;

    /**
     * Indicates a prepressed state;
     * the short time between ACTION_DOWN and recognizing
     * a 'real' press. Prepressed is used to recognize quick taps
     * even when they are shorter than ViewConfiguration.getTapTimeout().
     *
     * @hide
     */
    private static final int PFLAG_PREPRESSED           = 0x02000000;

    /**
     * Indicates whether the view is temporarily detached.
     *
     * @hide
     */
    static final int PFLAG_CANCEL_NEXT_UP_EVENT         = 0x04000000;

    /**
     * Indicates that we should awaken scroll bars once attached
     *
     * PLEASE NOTE: This flag is now unused as we now send onVisibilityChanged
     * during window attachment and it is no longer needed. Feel free to repurpose it.
     *
     * @hide
     */
```

```
private static final int PFLAG_AWAKEN_SCROLL_BARS_ON_ATTACH = 0x08000000;

/**
 * Indicates that the view has received HOVER_ENTER.  Cleared on HOVER_EXIT.
 * @hide
 */
private static final int PFLAG_HOVERED             = 0x10000000;

/**
 * no longer needed, should be reused
 */
private static final int PFLAG_DOES_NOTHING_REUSE_PLEASE = 0x20000000;

/** {@hide} */
static final int PFLAG_ACTIVATED                   = 0x40000000;

/**
 * Indicates that this view was specifically invalidated, not just dirtied because some
 * child view was invalidated. The flag is used to determine when we need to recreate
 * a view's display list (as opposed to just returning a reference to its existing
 * display list).
 *
 * @hide
 */
static final int PFLAG_INVALIDATED                 = 0x80000000;

/**
 * Masks for mPrivateFlags2, as generated by dumpFlags():
 *
 * |-------|-------|-------|-------|
 *                                 1 PFLAG2_DRAG_CAN_ACCEPT
 *                                1  PFLAG2_DRAG_HOVERED
 *                              11   PFLAG2_LAYOUT_DIRECTION_MASK
 *                             1     PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL
 *                            1      PFLAG2_LAYOUT_DIRECTION_RESOLVED
 *                            11     PFLAG2_LAYOUT_DIRECTION_RESOLVED_MASK
 *                           1       PFLAG2_TEXT_DIRECTION_FLAGS[1]
 *                          1        PFLAG2_TEXT_DIRECTION_FLAGS[2]
 *                          11       PFLAG2_TEXT_DIRECTION_FLAGS[3]
 *                         1         PFLAG2_TEXT_DIRECTION_FLAGS[4]
 *                         1 1       PFLAG2_TEXT_DIRECTION_FLAGS[5]
 *                         11        PFLAG2_TEXT_DIRECTION_FLAGS[6]
 *                         111       PFLAG2_TEXT_DIRECTION_FLAGS[7]
 *                         111       PFLAG2_TEXT_DIRECTION_MASK
 *                        1          PFLAG2_TEXT_DIRECTION_RESOLVED
 *                       1           PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT
 *                      111          PFLAG2_TEXT_DIRECTION_RESOLVED_MASK
 *                     1             PFLAG2_TEXT_ALIGNMENT_FLAGS[1]
 *                     1             PFLAG2_TEXT_ALIGNMENT_FLAGS[2]
 *                     11            PFLAG2_TEXT_ALIGNMENT_FLAGS[3]
 *                    1              PFLAG2_TEXT_ALIGNMENT_FLAGS[4]
 *                    1 1            PFLAG2_TEXT_ALIGNMENT_FLAGS[5]
 *                    11             PFLAG2_TEXT_ALIGNMENT_FLAGS[6]
 *                    111            PFLAG2_TEXT_ALIGNMENT_MASK
 *                   1               PFLAG2_TEXT_ALIGNMENT_RESOLVED
 *                  1                PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT
 *                 111               PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK
 *              111                  PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK
 *           11                      PFLAG2_ACCESSIBILITY_LIVE_REGION_MASK
 *         1                         PFLAG2_ACCESSIBILITY_FOCUSED
 *        1                          PFLAG2_SUBTREE_ACCESSIBILITY_STATE_CHANGED
 *       1                           PFLAG2_VIEW_QUICK_REJECTED
 *      1                            PFLAG2_PADDING_RESOLVED
 *     1                             PFLAG2_DRAWABLE_RESOLVED
 *    1                              PFLAG2_HAS_TRANSIENT_STATE
 * |-------|-------|-------|-------|
 */

/**
 * Indicates that this view has reported that it can accept the current drag's content.
 * Cleared when the drag operation concludes.
 * @hide
 */
static final int PFLAG2_DRAG_CAN_ACCEPT            = 0x00000001;

/**
 * Indicates that this view is currently directly under the drag location in a
 * drag-and-drop operation involving content that it can accept.  Cleared when
 * the drag exits the view, or when the drag operation concludes.
 * @hide
 */
static final int PFLAG2_DRAG_HOVERED               = 0x00000002;
```

```java
/** @hide */
@IntDef({
    LAYOUT_DIRECTION_LTR,
    LAYOUT_DIRECTION_RTL,
    LAYOUT_DIRECTION_INHERIT,
    LAYOUT_DIRECTION_LOCALE
})
@Retention(RetentionPolicy.SOURCE)
// Not called LayoutDirection to avoid conflict with android.util.LayoutDirection
public @interface LayoutDir {}

/** @hide */
@IntDef({
    LAYOUT_DIRECTION_LTR,
    LAYOUT_DIRECTION_RTL
})
@Retention(RetentionPolicy.SOURCE)
public @interface ResolvedLayoutDir {}

/**
 * A flag to indicate that the layout direction of this view has not been defined yet.
 * @hide
 */
public static final int LAYOUT_DIRECTION_UNDEFINED = LayoutDirection.UNDEFINED;

/**
 * Horizontal layout direction of this view is from Left to Right.
 * Use with {@link #setLayoutDirection}.
 */
public static final int LAYOUT_DIRECTION_LTR = LayoutDirection.LTR;

/**
 * Horizontal layout direction of this view is from Right to Left.
 * Use with {@link #setLayoutDirection}.
 */
public static final int LAYOUT_DIRECTION_RTL = LayoutDirection.RTL;

/**
 * Horizontal layout direction of this view is inherited from its parent.
 * Use with {@link #setLayoutDirection}.
 */
public static final int LAYOUT_DIRECTION_INHERIT = LayoutDirection.INHERIT;

/**
 * Horizontal layout direction of this view is from deduced from the default language
 * script for the locale. Use with {@link #setLayoutDirection}.
 */
public static final int LAYOUT_DIRECTION_LOCALE = LayoutDirection.LOCALE;

/**
 * Bit shift to get the horizontal layout direction. (bits after DRAG_HOVERED)
 * @hide
 */
static final int PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT = 2;

/**
 * Mask for use with private flags indicating bits used for horizontal layout direction.
 * @hide
 */
static final int PFLAG2_LAYOUT_DIRECTION_MASK = 0x00000003 << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT;

/**
 * Indicates whether the view horizontal layout direction has been resolved and drawn to the
 * right-to-left direction.
 * @hide
 */
static final int PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL = 4 << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT;

/**
 * Indicates whether the view horizontal layout direction has been resolved.
 * @hide
 */
static final int PFLAG2_LAYOUT_DIRECTION_RESOLVED = 8 << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT;

/**
 * Mask for use with private flags indicating bits used for resolved horizontal layout direction.
 * @hide
 */
static final int PFLAG2_LAYOUT_DIRECTION_RESOLVED_MASK = 0x0000000C
        << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT;
```

```
/*
 * Array of horizontal layout direction flags for mapping attribute "layoutDirection" to correct
 * flag value.
 * @hide
 */
private static final int[] LAYOUT_DIRECTION_FLAGS = {
        LAYOUT_DIRECTION_LTR,
        LAYOUT_DIRECTION_RTL,
        LAYOUT_DIRECTION_INHERIT,
        LAYOUT_DIRECTION_LOCALE
};

/**
 * Default horizontal layout direction.
 */
private static final int LAYOUT_DIRECTION_DEFAULT = LAYOUT_DIRECTION_INHERIT;

/**
 * Default horizontal layout direction.
 * @hide
 */
static final int LAYOUT_DIRECTION_RESOLVED_DEFAULT = LAYOUT_DIRECTION_LTR;

/**
 * Text direction is inherited through {@link ViewGroup}
 */
public static final int TEXT_DIRECTION_INHERIT = 0;

/**
 * Text direction is using "first strong algorithm". The first strong directional character
 * determines the paragraph direction. If there is no strong directional character, the
 * paragraph direction is the view's resolved layout direction.
 */
public static final int TEXT_DIRECTION_FIRST_STRONG = 1;

/**
 * Text direction is using "any-RTL" algorithm. The paragraph direction is RTL if it contains
 * any strong RTL character, otherwise it is LTR if it contains any strong LTR characters.
 * If there are neither, the paragraph direction is the view's resolved layout direction.
 */
public static final int TEXT_DIRECTION_ANY_RTL = 2;

/**
 * Text direction is forced to LTR.
 */
public static final int TEXT_DIRECTION_LTR = 3;

/**
 * Text direction is forced to RTL.
 */
public static final int TEXT_DIRECTION_RTL = 4;

/**
 * Text direction is coming from the system Locale.
 */
public static final int TEXT_DIRECTION_LOCALE = 5;

/**
 * Text direction is using "first strong algorithm". The first strong directional character
 * determines the paragraph direction. If there is no strong directional character, the
 * paragraph direction is LTR.
 */
public static final int TEXT_DIRECTION_FIRST_STRONG_LTR = 6;

/**
 * Text direction is using "first strong algorithm". The first strong directional character
 * determines the paragraph direction. If there is no strong directional character, the
 * paragraph direction is RTL.
 */
public static final int TEXT_DIRECTION_FIRST_STRONG_RTL = 7;

/**
 * Default text direction is inherited
 */
private static final int TEXT_DIRECTION_DEFAULT = TEXT_DIRECTION_INHERIT;

/**
 * Default resolved text direction
 * @hide
 */
static final int TEXT_DIRECTION_RESOLVED_DEFAULT = TEXT_DIRECTION_FIRST_STRONG;
```

```
/**
 * Bit shift to get the horizontal layout direction. (bits after LAYOUT_DIRECTION_RESOLVED)
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_MASK_SHIFT = 6;

/**
 * Mask for use with private flags indicating bits used for text direction.
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_MASK = 0x00000007
        << PFLAG2_TEXT_DIRECTION_MASK_SHIFT;

/**
 * Array of text direction flags for mapping attribute "textDirection" to correct
 * flag value.
 * @hide
 */
private static final int[] PFLAG2_TEXT_DIRECTION_FLAGS = {
        TEXT_DIRECTION_INHERIT << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_FIRST_STRONG << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_ANY_RTL << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_LTR << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_RTL << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_LOCALE << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_FIRST_STRONG_LTR << PFLAG2_TEXT_DIRECTION_MASK_SHIFT,
        TEXT_DIRECTION_FIRST_STRONG_RTL << PFLAG2_TEXT_DIRECTION_MASK_SHIFT
};

/**
 * Indicates whether the view text direction has been resolved.
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_RESOLVED = 0x00000008
        << PFLAG2_TEXT_DIRECTION_MASK_SHIFT;

/**
 * Bit shift to get the horizontal layout direction. (bits after DRAG_HOVERED)
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT = 10;

/**
 * Mask for use with private flags indicating bits used for resolved text direction.
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_RESOLVED_MASK = 0x00000007
        << PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT;

/**
 * Indicates whether the view text direction has been resolved to the "first strong" heuristic.
 * @hide
 */
static final int PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT =
        TEXT_DIRECTION_RESOLVED_DEFAULT << PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT;

/** @hide */
@IntDef({
    TEXT_ALIGNMENT_INHERIT,
    TEXT_ALIGNMENT_GRAVITY,
    TEXT_ALIGNMENT_CENTER,
    TEXT_ALIGNMENT_TEXT_START,
    TEXT_ALIGNMENT_TEXT_END,
    TEXT_ALIGNMENT_VIEW_START,
    TEXT_ALIGNMENT_VIEW_END
})
@Retention(RetentionPolicy.SOURCE)
public @interface TextAlignment {}

/**
 * Default text alignment. The text alignment of this View is inherited from its parent.
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_INHERIT = 0;

/**
 * Default for the root view. The gravity determines the text alignment, ALIGN_NORMAL,
 * ALIGN_CENTER, or ALIGN_OPPOSITE, which are relative to each paragraph’s text direction.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_GRAVITY = 1;
```

```
/**
 * Align to the start of the paragraph, e.g. ALIGN_NORMAL.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_TEXT_START = 2;

/**
 * Align to the end of the paragraph, e.g. ALIGN_OPPOSITE.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_TEXT_END = 3;

/**
 * Center the paragraph, e.g. ALIGN_CENTER.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_CENTER = 4;

/**
 * Align to the start of the view, which is ALIGN_LEFT if the view's resolved
 * layoutDirection is LTR, and ALIGN_RIGHT otherwise.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_VIEW_START = 5;

/**
 * Align to the end of the view, which is ALIGN_RIGHT if the view's resolved
 * layoutDirection is LTR, and ALIGN_LEFT otherwise.
 *
 * Use with {@link #setTextAlignment(int)}
 */
public static final int TEXT_ALIGNMENT_VIEW_END = 6;

/**
 * Default text alignment is inherited
 */
private static final int TEXT_ALIGNMENT_DEFAULT = TEXT_ALIGNMENT_GRAVITY;

/**
 * Default resolved text alignment
 * @hide
 */
static final int TEXT_ALIGNMENT_RESOLVED_DEFAULT = TEXT_ALIGNMENT_GRAVITY;

/**
  * Bit shift to get the horizontal layout direction. (bits after DRAG_HOVERED)
  * @hide
  */
static final int PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT = 13;

/**
  * Mask for use with private flags indicating bits used for text alignment.
  * @hide
  */
static final int PFLAG2_TEXT_ALIGNMENT_MASK = 0x00000007 << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT;

/**
 * Array of text direction flags for mapping attribute "textAlignment" to correct
 * flag value.
 * @hide
 */
private static final int[] PFLAG2_TEXT_ALIGNMENT_FLAGS = {
        TEXT_ALIGNMENT_INHERIT << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_GRAVITY << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_TEXT_START << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_TEXT_END << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_CENTER << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_VIEW_START << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT,
        TEXT_ALIGNMENT_VIEW_END << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT
};

/**
 * Indicates whether the view text alignment has been resolved.
 * @hide
 */
static final int PFLAG2_TEXT_ALIGNMENT_RESOLVED = 0x00000008 << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT;
```

```
/**
 * Bit shift to get the resolved text alignment.
 * @hide
 */
static final int PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT = 17;

/**
 * Mask for use with private flags indicating bits used for text alignment.
 * @hide
 */
static final int PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK = 0x00000007
        << PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT;

/**
 * Indicates whether if the view text alignment has been resolved to gravity
 */
private static final int PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT =
        TEXT_ALIGNMENT_RESOLVED_DEFAULT << PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT;

// Accessiblity constants for mPrivateFlags2

/**
 * Shift for the bits in {@link #mPrivateFlags2} related to the
 * "importantForAccessibility" attribute.
 */
static final int PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT = 20;

/**
 * Automatically determine whether a view is important for accessibility.
 */
public static final int IMPORTANT_FOR_ACCESSIBILITY_AUTO = 0x00000000;

/**
 * The view is important for accessibility.
 */
public static final int IMPORTANT_FOR_ACCESSIBILITY_YES = 0x00000001;

/**
 * The view is not important for accessibility.
 */
public static final int IMPORTANT_FOR_ACCESSIBILITY_NO = 0x00000002;

/**
 * The view is not important for accessibility, nor are any of its
 * descendant views.
 */
public static final int IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS = 0x00000004;

/**
 * The default whether the view is important for accessibility.
 */
static final int IMPORTANT_FOR_ACCESSIBILITY_DEFAULT = IMPORTANT_FOR_ACCESSIBILITY_AUTO;

/**
 * Mask for obtaining the bits which specify how to determine
 * whether a view is important for accessibility.
 */
static final int PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK = (IMPORTANT_FOR_ACCESSIBILITY_AUTO
    | IMPORTANT_FOR_ACCESSIBILITY_YES | IMPORTANT_FOR_ACCESSIBILITY_NO
    | IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS)
    << PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT;

/**
 * Shift for the bits in {@link #mPrivateFlags2} related to the
 * "accessibilityLiveRegion" attribute.
 */
static final int PFLAG2_ACCESSIBILITY_LIVE_REGION_SHIFT = 23;

/**
 * Live region mode specifying that accessibility services should not
 * automatically announce changes to this view. This is the default live
 * region mode for most views.
 * <p>
 * Use with {@link #setAccessibilityLiveRegion(int)}.
 */
public static final int ACCESSIBILITY_LIVE_REGION_NONE = 0x00000000;

/**
 * Live region mode specifying that accessibility services should announce
 * changes to this view.
 * <p>
 * Use with {@link #setAccessibilityLiveRegion(int)}.
```

```
     */
    public static final int ACCESSIBILITY_LIVE_REGION_POLITE = 0x00000001;

    /**
     * Live region mode specifying that accessibility services should interrupt
     * ongoing speech to immediately announce changes to this view.
     * <p>
     * Use with {@link #setAccessibilityLiveRegion(int)}.
     */
    public static final int ACCESSIBILITY_LIVE_REGION_ASSERTIVE = 0x00000002;

    /**
     * The default whether the view is important for accessibility.
     */
    static final int ACCESSIBILITY_LIVE_REGION_DEFAULT = ACCESSIBILITY_LIVE_REGION_NONE;

    /**
     * Mask for obtaining the bits which specify a view's accessibility live
     * region mode.
     */
    static final int PFLAG2_ACCESSIBILITY_LIVE_REGION_MASK = (ACCESSIBILITY_LIVE_REGION_NONE
            | ACCESSIBILITY_LIVE_REGION_POLITE | ACCESSIBILITY_LIVE_REGION_ASSERTIVE)
            << PFLAG2_ACCESSIBILITY_LIVE_REGION_SHIFT;

    /**
     * Flag indicating whether a view has accessibility focus.
     */
    static final int PFLAG2_ACCESSIBILITY_FOCUSED = 0x04000000;

    /**
     * Flag whether the accessibility state of the subtree rooted at this view changed.
     */
    static final int PFLAG2_SUBTREE_ACCESSIBILITY_STATE_CHANGED = 0x08000000;

    /**
     * Flag indicating whether a view failed the quickReject() check in draw(). This condition
     * is used to check whether later changes to the view's transform should invalidate the
     * view to force the quickReject test to run again.
     */
    static final int PFLAG2_VIEW_QUICK_REJECTED = 0x10000000;

    /**
     * Flag indicating that start/end padding has been resolved into left/right padding
     * for use in measurement, layout, drawing, etc. This is set by {@link #resolvePadding()}
     * and checked by {@link #measure(int, int)} to determine if padding needs to be resolved
     * during measurement. In some special cases this is required such as when an adapter-based
     * view measures prospective children without attaching them to a window.
     */
    static final int PFLAG2_PADDING_RESOLVED = 0x20000000;

    /**
     * Flag indicating that the start/end drawables has been resolved into left/right ones.
     */
    static final int PFLAG2_DRAWABLE_RESOLVED = 0x40000000;

    /**
     * Indicates that the view is tracking some sort of transient state
     * that the app should not need to be aware of, but that the framework
     * should take special care to preserve.
     */
    static final int PFLAG2_HAS_TRANSIENT_STATE = 0x80000000;

    /**
     * Group of bits indicating that RTL properties resolution is done.
     */
    static final int ALL_RTL_PROPERTIES_RESOLVED = PFLAG2_LAYOUT_DIRECTION_RESOLVED |
            PFLAG2_TEXT_DIRECTION_RESOLVED |
            PFLAG2_TEXT_ALIGNMENT_RESOLVED |
            PFLAG2_PADDING_RESOLVED |
            PFLAG2_DRAWABLE_RESOLVED;

    // There are a couple of flags left in mPrivateFlags2

    /* End of masks for mPrivateFlags2 */

    /**
     * Masks for mPrivateFlags3, as generated by dumpFlags():
     *
     * |-------|-------|-------|-------|
     *                                 1 PFLAG3_VIEW_IS_ANIMATING_TRANSFORM
     *                                1  PFLAG3_VIEW_IS_ANIMATING_ALPHA
     *                               1   PFLAG3_IS_LAID_OUT
```

```
 *                             1    PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT
 *                            1     PFLAG3_CALLED_SUPER
 *                           1      PFLAG3_APPLYING_INSETS
 *                          1       PFLAG3_FITTING_SYSTEM_WINDOWS
 *                         1        PFLAG3_NESTED_SCROLLING_ENABLED
 *                        1         PFLAG3_SCROLL_INDICATOR_TOP
 *                       1          PFLAG3_SCROLL_INDICATOR_BOTTOM
 *                      1           PFLAG3_SCROLL_INDICATOR_LEFT
 *                     1            PFLAG3_SCROLL_INDICATOR_RIGHT
 *                    1             PFLAG3_SCROLL_INDICATOR_START
 *                   1              PFLAG3_SCROLL_INDICATOR_END
 *                  1               PFLAG3_ASSIST_BLOCKED
 *                 1                PFLAG3_CLUSTER
 *                1                 PFLAG3_IS_AUTOFILLED
 *               1                  PFLAG3_FINGER_DOWN
 *              1                   PFLAG3_FOCUSED_BY_DEFAULT
 *           1111                   PFLAG3_IMPORTANT_FOR_AUTOFILL
 *          1                       PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE
 *         1                        PFLAG3_HAS_OVERLAPPING_RENDERING_FORCED
 *        1                         PFLAG3_TEMPORARY_DETACH
 *       1                          PFLAG3_NO_REVEAL_ON_FOCUS
 *      1                           PFLAG3_NOTIFY_AUTOFILL_ENTER_ON_LAYOUT
 * |-------|-------|-------|-------|
 */

/**
 * Flag indicating that view has a transform animation set on it. This is used to track whether
 * an animation is cleared between successive frames, in order to tell the associated
 * DisplayList to clear its animation matrix.
 */
static final int PFLAG3_VIEW_IS_ANIMATING_TRANSFORM = 0x1;

/**
 * Flag indicating that view has an alpha animation set on it. This is used to track whether an
 * animation is cleared between successive frames, in order to tell the associated
 * DisplayList to restore its alpha value.
 */
static final int PFLAG3_VIEW_IS_ANIMATING_ALPHA = 0x2;

/**
 * Flag indicating that the view has been through at least one layout since it
 * was last attached to a window.
 */
static final int PFLAG3_IS_LAID_OUT = 0x4;

/**
 * Flag indicating that a call to measure() was skipped and should be done
 * instead when layout() is invoked.
 */
static final int PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT = 0x8;

/**
 * Flag indicating that an overridden method correctly called down to
 * the superclass implementation as required by the API spec.
 */
static final int PFLAG3_CALLED_SUPER = 0x10;

/**
 * Flag indicating that we're in the process of applying window insets.
 */
static final int PFLAG3_APPLYING_INSETS = 0x20;

/**
 * Flag indicating that we're in the process of fitting system windows using the old method.
 */
static final int PFLAG3_FITTING_SYSTEM_WINDOWS = 0x40;

/**
 * Flag indicating that nested scrolling is enabled for this view.
 * The view will optionally cooperate with views up its parent chain to allow for
 * integrated nested scrolling along the same axis.
 */
static final int PFLAG3_NESTED_SCROLLING_ENABLED = 0x80;

/**
 * Flag indicating that the bottom scroll indicator should be displayed
 * when this view can scroll up.
 */
static final int PFLAG3_SCROLL_INDICATOR_TOP = 0x0100;

/**
 * Flag indicating that the bottom scroll indicator should be displayed
```

```
     * when this view can scroll down.
     */
    static final int PFLAG3_SCROLL_INDICATOR_BOTTOM = 0x0200;

    /**
     * Flag indicating that the left scroll indicator should be displayed
     * when this view can scroll left.
     */
    static final int PFLAG3_SCROLL_INDICATOR_LEFT = 0x0400;

    /**
     * Flag indicating that the right scroll indicator should be displayed
     * when this view can scroll right.
     */
    static final int PFLAG3_SCROLL_INDICATOR_RIGHT = 0x0800;

    /**
     * Flag indicating that the start scroll indicator should be displayed
     * when this view can scroll in the start direction.
     */
    static final int PFLAG3_SCROLL_INDICATOR_START = 0x1000;

    /**
     * Flag indicating that the end scroll indicator should be displayed
     * when this view can scroll in the end direction.
     */
    static final int PFLAG3_SCROLL_INDICATOR_END = 0x2000;

    static final int DRAG_MASK = PFLAG2_DRAG_CAN_ACCEPT | PFLAG2_DRAG_HOVERED;

    static final int SCROLL_INDICATORS_NONE = 0x0000;

    /**
     * Mask for use with setFlags indicating bits used for indicating which
     * scroll indicators are enabled.
     */
    static final int SCROLL_INDICATORS_PFLAG3_MASK = PFLAG3_SCROLL_INDICATOR_TOP
            | PFLAG3_SCROLL_INDICATOR_BOTTOM | PFLAG3_SCROLL_INDICATOR_LEFT
            | PFLAG3_SCROLL_INDICATOR_RIGHT | PFLAG3_SCROLL_INDICATOR_START
            | PFLAG3_SCROLL_INDICATOR_END;

    /**
     * Left-shift required to translate between public scroll indicator flags
     * and internal PFLAGS3 flags. When used as a right-shift, translates
     * PFLAGS3 flags to public flags.
     */
    static final int SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT = 8;

    /** @hide */
    @Retention(RetentionPolicy.SOURCE)
    @IntDef(flag = true,
            value = {
                    SCROLL_INDICATOR_TOP,
                    SCROLL_INDICATOR_BOTTOM,
                    SCROLL_INDICATOR_LEFT,
                    SCROLL_INDICATOR_RIGHT,
                    SCROLL_INDICATOR_START,
                    SCROLL_INDICATOR_END,
            })
    public @interface ScrollIndicators {}

    /**
     * Scroll indicator direction for the top edge of the view.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_TOP =
            PFLAG3_SCROLL_INDICATOR_TOP >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
     * Scroll indicator direction for the bottom edge of the view.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_BOTTOM =
            PFLAG3_SCROLL_INDICATOR_BOTTOM >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
```

```
     * Scroll indicator direction for the left edge of the view.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_LEFT =
            PFLAG3_SCROLL_INDICATOR_LEFT >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
     * Scroll indicator direction for the right edge of the view.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_RIGHT =
            PFLAG3_SCROLL_INDICATOR_RIGHT >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
     * Scroll indicator direction for the starting edge of the view.
     * <p>
     * Resolved according to the view's layout direction, see
     * {@link #getLayoutDirection()} for more information.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_START =
            PFLAG3_SCROLL_INDICATOR_START >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
     * Scroll indicator direction for the ending edge of the view.
     * <p>
     * Resolved according to the view's layout direction, see
     * {@link #getLayoutDirection()} for more information.
     *
     * @see #setScrollIndicators(int)
     * @see #setScrollIndicators(int, int)
     * @see #getScrollIndicators()
     */
    public static final int SCROLL_INDICATOR_END =
            PFLAG3_SCROLL_INDICATOR_END >> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;

    /**
     * <p>Indicates that we are allowing {@link ViewStructure} to traverse
     * into this view.<p>
     */
    static final int PFLAG3_ASSIST_BLOCKED = 0x4000;

    /**
     * Flag indicating that the view is a root of a keyboard navigation cluster.
     *
     * @see #isKeyboardNavigationCluster()
     * @see #setKeyboardNavigationCluster(boolean)
     */
    private static final int PFLAG3_CLUSTER = 0x8000;

    /**
     * Flag indicating that the view is autofilled
     *
     * @see #isAutofilled()
     * @see #setAutofilled(boolean)
     */
    private static final int PFLAG3_IS_AUTOFILLED = 0x10000;

    /**
     * Indicates that the user is currently touching the screen.
     * Currently used for the tooltip positioning only.
     */
    private static final int PFLAG3_FINGER_DOWN = 0x20000;

    /**
     * Flag indicating that this view is the default-focus view.
     *
     * @see #isFocusedByDefault()
     * @see #setFocusedByDefault(boolean)
     */
    private static final int PFLAG3_FOCUSED_BY_DEFAULT = 0x40000;

    /**
```

```
     * Shift for the bits in {@link #mPrivateFlags3} related to the
     * "importantForAutofill" attribute.
     */
    static final int PFLAG3_IMPORTANT_FOR_AUTOFILL_SHIFT = 19;

    /**
     * Mask for obtaining the bits which specify how to determine
     * whether a view is important for autofill.
     */
    static final int PFLAG3_IMPORTANT_FOR_AUTOFILL_MASK = (IMPORTANT_FOR_AUTOFILL_AUTO
            | IMPORTANT_FOR_AUTOFILL_YES | IMPORTANT_FOR_AUTOFILL_NO
            | IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS
            | IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS)
            << PFLAG3_IMPORTANT_FOR_AUTOFILL_SHIFT;

    /**
     * Whether this view has rendered elements that overlap (see {@link
     * #hasOverlappingRendering()}, {@link #forceHasOverlappingRendering(boolean)}, and
     * {@link #getHasOverlappingRendering()} ). The value in this bit is only valid when
     * PFLAG3_HAS_OVERLAPPING_RENDERING_FORCED has been set. Otherwise, the value is
     * determined by whatever {@link #hasOverlappingRendering()} returns.
     */
    private static final int PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE = 0x800000;

    /**
     * Whether {@link #forceHasOverlappingRendering(boolean)} has been called. When true, value
     * in PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE is valid.
     */
    private static final int PFLAG3_HAS_OVERLAPPING_RENDERING_FORCED = 0x1000000;

    /**
     * Flag indicating that the view is temporarily detached from the parent view.
     *
     * @see #onStartTemporaryDetach()
     * @see #onFinishTemporaryDetach()
     */
    static final int PFLAG3_TEMPORARY_DETACH = 0x2000000;

    /**
     * Flag indicating that the view does not wish to be revealed within its parent
     * hierarchy when it gains focus. Expressed in the negative since the historical
     * default behavior is to reveal on focus; this flag suppresses that behavior.
     *
     * @see #setRevealOnFocusHint(boolean)
     * @see #getRevealOnFocusHint()
     */
    private static final int PFLAG3_NO_REVEAL_ON_FOCUS = 0x4000000;

    /**
     * Flag indicating that when layout is completed we should notify
     * that the view was entered for autofill purposes. To minimize
     * showing autofill for views not visible to the user we evaluate
     * user visibility which cannot be done until the view is laid out.
     */
    static final int PFLAG3_NOTIFY_AUTOFILL_ENTER_ON_LAYOUT = 0x8000000;

    /* End of masks for mPrivateFlags3 */

    /**
     * Always allow a user to over-scroll this view, provided it is a
     * view that can scroll.
     *
     * @see #getOverScrollMode()
     * @see #setOverScrollMode(int)
     */
    public static final int OVER_SCROLL_ALWAYS = 0;

    /**
     * Allow a user to over-scroll this view only if the content is large
     * enough to meaningfully scroll, provided it is a view that can scroll.
     *
     * @see #getOverScrollMode()
     * @see #setOverScrollMode(int)
     */
    public static final int OVER_SCROLL_IF_CONTENT_SCROLLS = 1;

    /**
     * Never allow a user to over-scroll this view.
     *
     * @see #getOverScrollMode()
     * @see #setOverScrollMode(int)
     */
```

```
public static final int OVER_SCROLL_NEVER = 2;

/**
 * Special constant for {@link #setSystemUiVisibility(int)}: View has
 * requested the system UI (status bar) to be visible (the default).
 *
 * @see #setSystemUiVisibility(int)
 */
public static final int SYSTEM_UI_FLAG_VISIBLE = 0;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: View has requested the
 * system UI to enter an unobtrusive "low profile" mode.
 *
 * <p>This is for use in games, book readers, video players, or any other
 * "immersive" application where the usual system chrome is deemed too distracting.
 *
 * <p>In low profile mode, the status bar and/or navigation icons may dim.
 *
 * @see #setSystemUiVisibility(int)
 */
public static final int SYSTEM_UI_FLAG_LOW_PROFILE = 0x00000001;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: View has requested that the
 * system navigation be temporarily hidden.
 *
 * <p>This is an even less obtrusive state than that called for by
 * {@link #SYSTEM_UI_FLAG_LOW_PROFILE}; on devices that draw essential navigation controls
 * (Home, Back, and the like) on screen, <code>SYSTEM_UI_FLAG_HIDE_NAVIGATION</code> will cause
 * those to disappear. This is useful (in conjunction with the
 * {@link android.view.WindowManager.LayoutParams#FLAG_FULLSCREEN FLAG_FULLSCREEN} and
 * {@link android.view.WindowManager.LayoutParams#FLAG_LAYOUT_IN_SCREEN FLAG_LAYOUT_IN_SCREEN}
 * window flags) for displaying content using every last pixel on the display.
 *
 * <p>There is a limitation: because navigation controls are so important, the least user
 * interaction will cause them to reappear immediately.  When this happens, both
 * this flag and {@link #SYSTEM_UI_FLAG_FULLSCREEN} will be cleared automatically,
 * so that both elements reappear at the same time.
 *
 * @see #setSystemUiVisibility(int)
 */
public static final int SYSTEM_UI_FLAG_HIDE_NAVIGATION = 0x00000002;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: View has requested to go
 * into the normal fullscreen mode so that its content can take over the screen
 * while still allowing the user to interact with the application.
 *
 * <p>This has the same visual effect as
 * {@link android.view.WindowManager.LayoutParams#FLAG_FULLSCREEN
 * WindowManager.LayoutParams.FLAG_FULLSCREEN},
 * meaning that non-critical screen decorations (such as the status bar) will be
 * hidden while the user is in the View's window, focusing the experience on
 * that content.  Unlike the window flag, if you are using ActionBar in
 * overlay mode with {@link Window#FEATURE_ACTION_BAR_OVERLAY
 * Window.FEATURE_ACTION_BAR_OVERLAY}, then enabling this flag will also
 * hide the action bar.
 *
 * <p>This approach to going fullscreen is best used over the window flag when
 * it is a transient state -- that is, the application does this at certain
 * points in its user interaction where it wants to allow the user to focus
 * on content, but not as a continuous state.  For situations where the application
 * would like to simply stay full screen the entire time (such as a game that
 * wants to take over the screen), the
 * {@link android.view.WindowManager.LayoutParams#FLAG_FULLSCREEN window flag}
 * is usually a better approach.  The state set here will be removed by the system
 * in various situations (such as the user moving to another application) like
 * the other system UI states.
 *
 * <p>When using this flag, the application should provide some easy facility
 * for the user to go out of it.  A common example would be in an e-book
 * reader, where tapping on the screen brings back whatever screen and UI
 * decorations that had been hidden while the user was immersed in reading
 * the book.
 *
 * @see #setSystemUiVisibility(int)
 */
public static final int SYSTEM_UI_FLAG_FULLSCREEN = 0x00000004;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: When using other layout
```

```
     * flags, we would like a stable view of the content insets given to
     * {@link #fitSystemWindows(Rect)}.  This means that the insets seen there
     * will always represent the worst case that the application can expect
     * as a continuous state.  In the stock Android UI this is the space for
     * the system bar, nav bar, and status bar, but not more transient elements
     * such as an input method.
     *
     * The stable layout your UI sees is based on the system UI modes you can
     * switch to.  That is, if you specify {@link #SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN}
     * then you will get a stable layout for changes of the
     * {@link #SYSTEM_UI_FLAG_FULLSCREEN} mode; if you specify
     * {@link #SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN} and
     * {@link #SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION}, then you can transition
     * to {@link #SYSTEM_UI_FLAG_FULLSCREEN} and {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}
     * with a stable layout.  (Note that you should avoid using
     * {@link #SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION} by itself.)
     *
     * If you have set the window flag {@link WindowManager.LayoutParams#FLAG_FULLSCREEN}
     * to hide the status bar (instead of using {@link #SYSTEM_UI_FLAG_FULLSCREEN}),
     * then a hidden status bar will be considered a "stable" state for purposes
     * here.  This allows your UI to continually hide the status bar, while still
     * using the system UI flags to hide the action bar while still retaining
     * a stable layout.  Note that changing the window fullscreen flag will never
     * provide a stable layout for a clean transition.
     *
     * <p>If you are using ActionBar in
     * overlay mode with {@link Window#FEATURE_ACTION_BAR_OVERLAY
     * Window.FEATURE_ACTION_BAR_OVERLAY}, this flag will also impact the
     * insets it adds to those given to the application.
     */
    public static final int SYSTEM_UI_FLAG_LAYOUT_STABLE = 0x00000100;

    /**
     * Flag for {@link #setSystemUiVisibility(int)}: View would like its window
     * to be laid out as if it has requested
     * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, even if it currently hasn't.  This
     * allows it to avoid artifacts when switching in and out of that mode, at
     * the expense that some of its user interface may be covered by screen
     * decorations when they are shown.  You can perform layout of your inner
     * UI elements to account for the navigation system UI through the
     * {@link #fitSystemWindows(Rect)} method.
     */
    public static final int SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION = 0x00000200;

    /**
     * Flag for {@link #setSystemUiVisibility(int)}: View would like its window
     * to be laid out as if it has requested
     * {@link #SYSTEM_UI_FLAG_FULLSCREEN}, even if it currently hasn't.  This
     * allows it to avoid artifacts when switching in and out of that mode, at
     * the expense that some of its user interface may be covered by screen
     * decorations when they are shown.  You can perform layout of your inner
     * UI elements to account for non-fullscreen system UI through the
     * {@link #fitSystemWindows(Rect)} method.
     */
    public static final int SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN = 0x00000400;

    /**
     * Flag for {@link #setSystemUiVisibility(int)}: View would like to remain interactive when
     * hiding the navigation bar with {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}.  If this flag is
     * not set, {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION} will be force cleared by the system on any
     * user interaction.
     * <p>Since this flag is a modifier for {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, it only
     * has an effect when used in combination with that flag.</p>
     */
    public static final int SYSTEM_UI_FLAG_IMMERSIVE = 0x00000800;

    /**
     * Flag for {@link #setSystemUiVisibility(int)}: View would like to remain interactive when
     * hiding the status bar with {@link #SYSTEM_UI_FLAG_FULLSCREEN} and/or hiding the navigation
     * bar with {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}.  Use this flag to create an immersive
     * experience while also hiding the system bars.  If this flag is not set,
     * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION} will be force cleared by the system on any user
     * interaction, and {@link #SYSTEM_UI_FLAG_FULLSCREEN} will be force-cleared by the system
     * if the user swipes from the top of the screen.
     * <p>When system bars are hidden in immersive mode, they can be revealed temporarily with
     * system gestures, such as swiping from the top of the screen.  These transient system bars
     * will overlay app's content, may have some degree of transparency, and will automatically
     * hide after a short timeout.
     * </p><p>Since this flag is a modifier for {@link #SYSTEM_UI_FLAG_FULLSCREEN} and
     * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, it only has an effect when used in combination
     * with one or both of those flags.</p>
     */
```

```
public static final int SYSTEM_UI_FLAG_IMMERSIVE_STICKY = 0x00001000;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: Requests the status bar to draw in a mode that
 * is compatible with light status bar backgrounds.
 *
 * <p>For this to take effect, the window must request
 * {@link android.view.WindowManager.LayoutParams#FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS
 *         FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS} but not
 * {@link android.view.WindowManager.LayoutParams#FLAG_TRANSLUCENT_STATUS
 *         FLAG_TRANSLUCENT_STATUS}.
 *
 * @see android.R.attr#windowLightStatusBar
 */
public static final int SYSTEM_UI_FLAG_LIGHT_STATUS_BAR = 0x00002000;

/**
 * This flag was previously used for a private API. DO NOT reuse it for a public API as it might
 * trigger undefined behavior on older platforms with apps compiled against a new SDK.
 */
private static final int SYSTEM_UI_RESERVED_LEGACY1 = 0x00004000;

/**
 * This flag was previously used for a private API. DO NOT reuse it for a public API as it might
 * trigger undefined behavior on older platforms with apps compiled against a new SDK.
 */
private static final int SYSTEM_UI_RESERVED_LEGACY2 = 0x00010000;

/**
 * Flag for {@link #setSystemUiVisibility(int)}: Requests the navigation bar to draw in a mode
 * that is compatible with light navigation bar backgrounds.
 *
 * <p>For this to take effect, the window must request
 * {@link android.view.WindowManager.LayoutParams#FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS
 *         FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS} but not
 * {@link android.view.WindowManager.LayoutParams#FLAG_TRANSLUCENT_NAVIGATION
 *         FLAG_TRANSLUCENT_NAVIGATION}.
 */
public static final int SYSTEM_UI_FLAG_LIGHT_NAVIGATION_BAR = 0x00000010;

/**
 * @deprecated Use {@link #SYSTEM_UI_FLAG_LOW_PROFILE} instead.
 */
@Deprecated
public static final int STATUS_BAR_HIDDEN = SYSTEM_UI_FLAG_LOW_PROFILE;

/**
 * @deprecated Use {@link #SYSTEM_UI_FLAG_VISIBLE} instead.
 */
@Deprecated
public static final int STATUS_BAR_VISIBLE = SYSTEM_UI_FLAG_VISIBLE;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to make the status bar not expandable.  Unless you also
 * set {@link #STATUS_BAR_DISABLE_NOTIFICATION_ICONS}, new notifications will continue to show.
 */
public static final int STATUS_BAR_DISABLE_EXPAND = 0x00010000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide notification icons and scrolling ticker text.
 */
public static final int STATUS_BAR_DISABLE_NOTIFICATION_ICONS = 0x00020000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to disable incoming notification alerts.  This will not block
 * icons, but it will block sound, vibrating and other visual or aural notifications.
 */
```

```
public static final int STATUS_BAR_DISABLE_NOTIFICATION_ALERTS = 0x00040000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide only the scrolling ticker.  Note that
 * {@link #STATUS_BAR_DISABLE_NOTIFICATION_ICONS} implies
 * {@link #STATUS_BAR_DISABLE_NOTIFICATION_TICKER}.
 */
public static final int STATUS_BAR_DISABLE_NOTIFICATION_TICKER = 0x00080000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide the center system info area.
 */
public static final int STATUS_BAR_DISABLE_SYSTEM_INFO = 0x00100000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide only the home button.  Don't use this
 * unless you're a special part of the system UI (i.e., setup wizard, keyguard).
 */
public static final int STATUS_BAR_DISABLE_HOME = 0x00200000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide only the back button. Don't use this
 * unless you're a special part of the system UI (i.e., setup wizard, keyguard).
 */
public static final int STATUS_BAR_DISABLE_BACK = 0x00400000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide only the clock.  You might use this if your activity has
 * its own clock making the status bar's clock redundant.
 */
public static final int STATUS_BAR_DISABLE_CLOCK = 0x00800000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to hide only the recent apps button. Don't use this
 * unless you're a special part of the system UI (i.e., setup wizard, keyguard).
 */
public static final int STATUS_BAR_DISABLE_RECENT = 0x01000000;

/**
 * @hide
 *
 * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
 * out of the public fields to keep the undefined bits out of the developer's way.
 *
 * Flag to disable the global search gesture. Don't use this
 * unless you're a special part of the system UI (i.e., setup wizard, keyguard).
 */
public static final int STATUS_BAR_DISABLE_SEARCH = 0x02000000;

/**
 * @hide
 *
```

```
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the status bar is displayed in transient mode.
     */
    public static final int STATUS_BAR_TRANSIENT = 0x04000000;

    /**
     * @hide
     *
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the navigation bar is displayed in transient mode.
     */
    public static final int NAVIGATION_BAR_TRANSIENT = 0x08000000;

    /**
     * @hide
     *
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the hidden status bar would like to be shown.
     */
    public static final int STATUS_BAR_UNHIDE = 0x10000000;

    /**
     * @hide
     *
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the hidden navigation bar would like to be shown.
     */
    public static final int NAVIGATION_BAR_UNHIDE = 0x20000000;

    /**
     * @hide
     *
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the status bar is displayed in translucent mode.
     */
    public static final int STATUS_BAR_TRANSLUCENT = 0x40000000;

    /**
     * @hide
     *
     * NOTE: This flag may only be used in subtreeSystemUiVisibility. It is masked
     * out of the public fields to keep the undefined bits out of the developer's way.
     *
     * Flag to specify that the navigation bar is displayed in translucent mode.
     */
    public static final int NAVIGATION_BAR_TRANSLUCENT = 0x80000000;

    /**
     * @hide
     *
     * Makes navigation bar transparent (but not the status bar).
     */
    public static final int NAVIGATION_BAR_TRANSPARENT = 0x00008000;

    /**
     * @hide
     *
     * Makes status bar transparent (but not the navigation bar).
     */
    public static final int STATUS_BAR_TRANSPARENT = 0x00000008;

    /**
     * @hide
     *
     * Makes both status bar and navigation bar transparent.
     */
    public static final int SYSTEM_UI_TRANSPARENT = NAVIGATION_BAR_TRANSPARENT
            | STATUS_BAR_TRANSPARENT;

    /**
     * @hide
     */
```

```
public static final int PUBLIC_STATUS_BAR_VISIBILITY_MASK = 0x00003FF7;

/**
 * These are the system UI flags that can be cleared by events outside
 * of an application.  Currently this is just the ability to tap on the
 * screen while hiding the navigation bar to have it return.
 * @hide
 */
public static final int SYSTEM_UI_CLEARABLE_FLAGS =
        SYSTEM_UI_FLAG_LOW_PROFILE | SYSTEM_UI_FLAG_HIDE_NAVIGATION
        | SYSTEM_UI_FLAG_FULLSCREEN;

/**
 * Flags that can impact the layout in relation to system UI.
 */
public static final int SYSTEM_UI_LAYOUT_FLAGS =
        SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
        | SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN;

/** @hide */
@IntDef(flag = true,
        value = { FIND_VIEWS_WITH_TEXT, FIND_VIEWS_WITH_CONTENT_DESCRIPTION })
@Retention(RetentionPolicy.SOURCE)
public @interface FindViewFlags {}

/**
 * Find views that render the specified text.
 *
 * @see #findViewsWithText(ArrayList, CharSequence, int)
 */
public static final int FIND_VIEWS_WITH_TEXT = 0x00000001;

/**
 * Find find views that contain the specified content description.
 *
 * @see #findViewsWithText(ArrayList, CharSequence, int)
 */
public static final int FIND_VIEWS_WITH_CONTENT_DESCRIPTION = 0x00000002;

/**
 * Find views that contain {@link AccessibilityNodeProvider}. Such
 * a View is a root of virtual view hierarchy and may contain the searched
 * text. If this flag is set Views with providers are automatically
 * added and it is a responsibility of the client to call the APIs of
 * the provider to determine whether the virtual tree rooted at this View
 * contains the text, i.e. getting the list of {@link AccessibilityNodeInfo}s
 * representing the virtual views with this text.
 *
 * @see #findViewsWithText(ArrayList, CharSequence, int)
 *
 * @hide
 */
public static final int FIND_VIEWS_WITH_ACCESSIBILITY_NODE_PROVIDERS = 0x00000004;

/**
 * The undefined cursor position.
 *
 * @hide
 */
public static final int ACCESSIBILITY_CURSOR_POSITION_UNDEFINED = -1;

/**
 * Indicates that the screen has changed state and is now off.
 *
 * @see #onScreenStateChanged(int)
 */
public static final int SCREEN_STATE_OFF = 0x0;

/**
 * Indicates that the screen has changed state and is now on.
 *
 * @see #onScreenStateChanged(int)
 */
public static final int SCREEN_STATE_ON = 0x1;

/**
 * Indicates no axis of view scrolling.
 */
public static final int SCROLL_AXIS_NONE = 0;

/**
 * Indicates scrolling along the horizontal axis.
```

```
 */
public static final int SCROLL_AXIS_HORIZONTAL = 1 << 0;

/**
 * Indicates scrolling along the vertical axis.
 */
public static final int SCROLL_AXIS_VERTICAL = 1 << 1;

/**
 * Controls the over-scroll mode for this view.
 * See {@link #overScrollBy(int, int, int, int, int, int, int, int, boolean)},
 * {@link #OVER_SCROLL_ALWAYS}, {@link #OVER_SCROLL_IF_CONTENT_SCROLLS},
 * and {@link #OVER_SCROLL_NEVER}.
 */
private int mOverScrollMode;

/**
 * The parent this view is attached to.
 * {@hide}
 *
 * @see #getParent()
 */
protected ViewParent mParent;

/**
 * {@hide}
 */
AttachInfo mAttachInfo;

/**
 * {@hide}
 */
@ViewDebug.ExportedProperty(flagMapping = {
    @ViewDebug.FlagToString(mask = PFLAG_FORCE_LAYOUT, equals = PFLAG_FORCE_LAYOUT,
            name = "FORCE_LAYOUT"),
    @ViewDebug.FlagToString(mask = PFLAG_LAYOUT_REQUIRED, equals = PFLAG_LAYOUT_REQUIRED,
            name = "LAYOUT_REQUIRED"),
    @ViewDebug.FlagToString(mask = PFLAG_DRAWING_CACHE_VALID, equals = PFLAG_DRAWING_CACHE_VALID,
        name = "DRAWING_CACHE_INVALID", outputIf = false),
    @ViewDebug.FlagToString(mask = PFLAG_DRAWN, equals = PFLAG_DRAWN, name = "DRAWN", outputIf = true),
    @ViewDebug.FlagToString(mask = PFLAG_DRAWN, equals = PFLAG_DRAWN, name = "NOT_DRAWN", outputIf = false),
    @ViewDebug.FlagToString(mask = PFLAG_DIRTY_MASK, equals = PFLAG_DIRTY_OPAQUE, name = "DIRTY_OPAQUE"),
    @ViewDebug.FlagToString(mask = PFLAG_DIRTY_MASK, equals = PFLAG_DIRTY, name = "DIRTY")
}, formatToHexString = true)

/* @hide */
public int mPrivateFlags;
int mPrivateFlags2;
int mPrivateFlags3;

/**
 * This view's request for the visibility of the status bar.
 * @hide
 */
@ViewDebug.ExportedProperty(flagMapping = {
    @ViewDebug.FlagToString(mask = SYSTEM_UI_FLAG_LOW_PROFILE,
                            equals = SYSTEM_UI_FLAG_LOW_PROFILE,
                            name = "SYSTEM_UI_FLAG_LOW_PROFILE", outputIf = true),
    @ViewDebug.FlagToString(mask = SYSTEM_UI_FLAG_HIDE_NAVIGATION,
                            equals = SYSTEM_UI_FLAG_HIDE_NAVIGATION,
                            name = "SYSTEM_UI_FLAG_HIDE_NAVIGATION", outputIf = true),
    @ViewDebug.FlagToString(mask = PUBLIC_STATUS_BAR_VISIBILITY_MASK,
                            equals = SYSTEM_UI_FLAG_VISIBLE,
                            name = "SYSTEM_UI_FLAG_VISIBLE", outputIf = true)
}, formatToHexString = true)
int mSystemUiVisibility;

/**
 * Reference count for transient state.
 * @see #setHasTransientState(boolean)
 */
int mTransientStateCount = 0;

/**
 * Count of how many windows this view has been attached to.
 */
int mWindowAttachCount;

/**
 * The layout parameters associated with this view and used by the parent
 * {@link android.view.ViewGroup} to determine how this view should be
 * laid out.
```

```
     * {@hide}
     */
    protected ViewGroup.LayoutParams mLayoutParams;

    /**
     * The view flags hold various views states.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(formatToHexString = true)
    int mViewFlags;

    static class TransformationInfo {
        /**
         * The transform matrix for the View. This transform is calculated internally
         * based on the translation, rotation, and scale properties.
         *
         * Do *not* use this variable directly; instead call getMatrix(), which will
         * load the value from the View's RenderNode.
         */
        private final Matrix mMatrix = new Matrix();

        /**
         * The inverse transform matrix for the View. This transform is calculated
         * internally based on the translation, rotation, and scale properties.
         *
         * Do *not* use this variable directly; instead call getInverseMatrix(),
         * which will load the value from the View's RenderNode.
         */
        private Matrix mInverseMatrix;

        /**
         * The opacity of the View. This is a value from 0 to 1, where 0 means
         * completely transparent and 1 means completely opaque.
         */
        @ViewDebug.ExportedProperty
        float mAlpha = 1f;

        /**
         * The opacity of the view as manipulated by the Fade transition. This is a hidden
         * property only used by transitions, which is composited with the other alpha
         * values to calculate the final visual alpha value.
         */
        float mTransitionAlpha = 1f;
    }

    /** @hide */
    public TransformationInfo mTransformationInfo;

    /**
     * Current clip bounds. to which all drawing of this view are constrained.
     */
    Rect mClipBounds = null;

    private boolean mLastIsOpaque;

    /**
     * The distance in pixels from the left edge of this view's parent
     * to the left edge of this view.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(category = "layout")
    protected int mLeft;
    /**
     * The distance in pixels from the left edge of this view's parent
     * to the right edge of this view.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(category = "layout")
    protected int mRight;
    /**
     * The distance in pixels from the top edge of this view's parent
     * to the top edge of this view.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(category = "layout")
    protected int mTop;
    /**
     * The distance in pixels from the top edge of this view's parent
     * to the bottom edge of this view.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(category = "layout")
```

```
protected int mBottom;

/**
 * The offset, in pixels, by which the content of this view is scrolled
 * horizontally.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "scrolling")
protected int mScrollX;
/**
 * The offset, in pixels, by which the content of this view is scrolled
 * vertically.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "scrolling")
protected int mScrollY;

/**
 * The left padding in pixels, that is the distance in pixels between the
 * left edge of this view and the left edge of its content.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mPaddingLeft = 0;
/**
 * The right padding in pixels, that is the distance in pixels between the
 * right edge of this view and the right edge of its content.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mPaddingRight = 0;
/**
 * The top padding in pixels, that is the distance in pixels between the
 * top edge of this view and the top edge of its content.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mPaddingTop;
/**
 * The bottom padding in pixels, that is the distance in pixels between the
 * bottom edge of this view and the bottom edge of its content.
 * {@hide}
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mPaddingBottom;

/**
 * The layout insets in pixels, that is the distance in pixels between the
 * visible edges of this view its bounds.
 */
private Insets mLayoutInsets;

/**
 * Briefly describes the view and is primarily used for accessibility support.
 */
private CharSequence mContentDescription;

/**
 * Specifies the id of a view for which this view serves as a label for
 * accessibility purposes.
 */
private int mLabelForId = View.NO_ID;

/**
 * Predicate for matching labeled view id with its label for
 * accessibility purposes.
 */
private MatchLabelForPredicate mMatchLabelForPredicate;

/**
 * Specifies a view before which this one is visited in accessibility traversal.
 */
private int mAccessibilityTraversalBeforeId = NO_ID;

/**
 * Specifies a view after which this one is visited in accessibility traversal.
 */
private int mAccessibilityTraversalAfterId = NO_ID;

/**
 * Predicate for matching a view by its id.
 */
```

```
private MatchIdPredicate mMatchIdPredicate;

/**
 * Cache the paddingRight set by the user to append to the scrollbar's size.
 *
 * @hide
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mUserPaddingRight;

/**
 * Cache the paddingBottom set by the user to append to the scrollbar's size.
 *
 * @hide
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mUserPaddingBottom;

/**
 * Cache the paddingLeft set by the user to append to the scrollbar's size.
 *
 * @hide
 */
@ViewDebug.ExportedProperty(category = "padding")
protected int mUserPaddingLeft;

/**
 * Cache the paddingStart set by the user to append to the scrollbar's size.
 *
 */
@ViewDebug.ExportedProperty(category = "padding")
int mUserPaddingStart;

/**
 * Cache the paddingEnd set by the user to append to the scrollbar's size.
 *
 */
@ViewDebug.ExportedProperty(category = "padding")
int mUserPaddingEnd;

/**
 * Cache initial left padding.
 *
 * @hide
 */
int mUserPaddingLeftInitial;

/**
 * Cache initial right padding.
 *
 * @hide
 */
int mUserPaddingRightInitial;

/**
 * Default undefined padding
 */
private static final int UNDEFINED_PADDING = Integer.MIN_VALUE;

/**
 * Cache if a left padding has been defined
 */
private boolean mLeftPaddingDefined = false;

/**
 * Cache if a right padding has been defined
 */
private boolean mRightPaddingDefined = false;

/**
 * @hide
 */
int mOldWidthMeasureSpec = Integer.MIN_VALUE;
/**
 * @hide
 */
int mOldHeightMeasureSpec = Integer.MIN_VALUE;

private LongSparseLongArray mMeasureCache;

@ViewDebug.ExportedProperty(deepExport = true, prefix = "bg_")
private Drawable mBackground;
```

```
private TintInfo mBackgroundTint;

@ViewDebug.ExportedProperty(deepExport = true, prefix = "fg_")
private ForegroundInfo mForegroundInfo;

private Drawable mScrollIndicatorDrawable;

/**
 * RenderNode used for backgrounds.
 * <p>
 * When non-null and valid, this is expected to contain an up-to-date copy
 * of the background drawable. It is cleared on temporary detach, and reset
 * on cleanup.
 */
private RenderNode mBackgroundRenderNode;

private int mBackgroundResource;
private boolean mBackgroundSizeChanged;

/** The default focus highlight.
 * @see #mDefaultFocusHighlightEnabled
 * @see Drawable#hasFocusStateSpecified()
 */
private Drawable mDefaultFocusHighlight;
private Drawable mDefaultFocusHighlightCache;
private boolean mDefaultFocusHighlightSizeChanged;
/**
 * True if the default focus highlight is needed on the target device.
 */
private static boolean sUseDefaultFocusHighlight;

private String mTransitionName;

static class TintInfo {
    ColorStateList mTintList;
    PorterDuff.Mode mTintMode;
    boolean mHasTintMode;
    boolean mHasTintList;
}

private static class ForegroundInfo {
    private Drawable mDrawable;
    private TintInfo mTintInfo;
    private int mGravity = Gravity.FILL;
    private boolean mInsidePadding = true;
    private boolean mBoundsChanged = true;
    private final Rect mSelfBounds = new Rect();
    private final Rect mOverlayBounds = new Rect();
}

static class ListenerInfo {
    /**
     * Listener used to dispatch focus change events.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    protected OnFocusChangeListener mOnFocusChangeListener;

    /**
     * Listeners for layout change events.
     */
    private ArrayList<OnLayoutChangeListener> mOnLayoutChangeListeners;

    protected OnScrollChangeListener mOnScrollChangeListener;

    /**
     * Listeners for attach events.
     */
    private CopyOnWriteArrayList<OnAttachStateChangeListener> mOnAttachStateChangeListeners;

    /**
     * Listener used to dispatch click events.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    public OnClickListener mOnClickListener;

    /**
     * Listener used to dispatch long click events.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
```

```
        protected OnLongClickListener mOnLongClickListener;

        /**
         * Listener used to dispatch context click events. This field should be made private, so it
         * is hidden from the SDK.
         * {@hide}
         */
        protected OnContextClickListener mOnContextClickListener;

        /**
         * Listener used to build the context menu.
         * This field should be made private, so it is hidden from the SDK.
         * {@hide}
         */
        protected OnCreateContextMenuListener mOnCreateContextMenuListener;

        private OnKeyListener mOnKeyListener;

        private OnTouchListener mOnTouchListener;

        private OnHoverListener mOnHoverListener;

        private OnGenericMotionListener mOnGenericMotionListener;

        private OnDragListener mOnDragListener;

        private OnSystemUiVisibilityChangeListener mOnSystemUiVisibilityChangeListener;

        OnApplyWindowInsetsListener mOnApplyWindowInsetsListener;

        OnCapturedPointerListener mOnCapturedPointerListener;
    }

    ListenerInfo mListenerInfo;

    private static class TooltipInfo {
        /**
         * Text to be displayed in a tooltip popup.
         */
        @Nullable
        CharSequence mTooltipText;

        /**
         * View-relative position of the tooltip anchor point.
         */
        int mAnchorX;
        int mAnchorY;

        /**
         * The tooltip popup.
         */
        @Nullable
        TooltipPopup mTooltipPopup;

        /**
         * Set to true if the tooltip was shown as a result of a long click.
         */
        boolean mTooltipFromLongClick;

        /**
         * Keep these Runnables so that they can be used to reschedule.
         */
        Runnable mShowTooltipRunnable;
        Runnable mHideTooltipRunnable;
    }

    TooltipInfo mTooltipInfo;

    // Temporary values used to hold (x,y) coordinates when delegating from the
    // two-arg performLongClick() method to the legacy no-arg version.
    private float mLongClickX = Float.NaN;
    private float mLongClickY = Float.NaN;

    /**
     * The application environment this view lives in.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(deepExport = true)
    protected Context mContext;

    private final Resources mResources;
```

```
private ScrollabilityCache mScrollCache;

private int[] mDrawableState = null;

ViewOutlineProvider mOutlineProvider = ViewOutlineProvider.BACKGROUND;

/**
 * Animator that automatically runs based on state changes.
 */
private StateListAnimator mStateListAnimator;

/**
 * When this view has focus and the next focus is {@link #FOCUS_LEFT},
 * the user may specify which view to go to next.
 */
private int mNextFocusLeftId = View.NO_ID;

/**
 * When this view has focus and the next focus is {@link #FOCUS_RIGHT},
 * the user may specify which view to go to next.
 */
private int mNextFocusRightId = View.NO_ID;

/**
 * When this view has focus and the next focus is {@link #FOCUS_UP},
 * the user may specify which view to go to next.
 */
private int mNextFocusUpId = View.NO_ID;

/**
 * When this view has focus and the next focus is {@link #FOCUS_DOWN},
 * the user may specify which view to go to next.
 */
private int mNextFocusDownId = View.NO_ID;

/**
 * When this view has focus and the next focus is {@link #FOCUS_FORWARD},
 * the user may specify which view to go to next.
 */
int mNextFocusForwardId = View.NO_ID;

/**
 * User-specified next keyboard navigation cluster in the {@link #FOCUS_FORWARD} direction.
 *
 * @see #findUserSetNextKeyboardNavigationCluster(View, int)
 */
int mNextClusterForwardId = View.NO_ID;

/**
 * Whether this View should use a default focus highlight when it gets focused but doesn't
 * have {@link android.R.attr#state_focused} defined in its background.
 */
boolean mDefaultFocusHighlightEnabled = true;

private CheckForLongPress mPendingCheckForLongPress;
private CheckForTap mPendingCheckForTap = null;
private PerformClick mPerformClick;
private SendViewScrolledAccessibilityEvent mSendViewScrolledAccessibilityEvent;

private UnsetPressedState mUnsetPressedState;

/**
 * Whether the long press's action has been invoked.  The tap's action is invoked on the
 * up event while a long press is invoked as soon as the long press duration is reached, so
 * a long press could be performed before the tap is checked, in which case the tap's action
 * should not be invoked.
 */
private boolean mHasPerformedLongPress;

/**
 * Whether a context click button is currently pressed down. This is true when the stylus is
 * touching the screen and the primary button has been pressed, or if a mouse's right button is
 * pressed. This is false once the button is released or if the stylus has been lifted.
 */
private boolean mInContextButtonPress;

/**
 * Whether the next up event should be ignored for the purposes of gesture recognition. This is
 * true after a stylus button press has occured, when the next up event should not be recognized
 * as a tap.
 */
```

```
private boolean mIgnoreNextUpEvent;

/**
 * The minimum height of the view. We'll try our best to have the height
 * of this view to at least this amount.
 */
@ViewDebug.ExportedProperty(category = "measurement")
private int mMinHeight;

/**
 * The minimum width of the view. We'll try our best to have the width
 * of this view to at least this amount.
 */
@ViewDebug.ExportedProperty(category = "measurement")
private int mMinWidth;

/**
 * The delegate to handle touch events that are physically in this view
 * but should be handled by another view.
 */
private TouchDelegate mTouchDelegate = null;

/**
 * Solid color to use as a background when creating the drawing cache. Enables
 * the cache to use 16 bit bitmaps instead of 32 bit.
 */
private int mDrawingCacheBackgroundColor = 0;

/**
 * Special tree observer used when mAttachInfo is null.
 */
private ViewTreeObserver mFloatingTreeObserver;

/**
 * Cache the touch slop from the context that created the view.
 */
private int mTouchSlop;

/**
 * Object that handles automatic animation of view properties.
 */
private ViewPropertyAnimator mAnimator = null;

/**
 * List of registered FrameMetricsObservers.
 */
private ArrayList<FrameMetricsObserver> mFrameMetricsObservers;

/**
 * Flag indicating that a drag can cross window boundaries.  When
 * {@link #startDragAndDrop(ClipData, DragShadowBuilder, Object, int)} is called
 * with this flag set, all visible applications with targetSdkVersion >=
 * {@link android.os.Build.VERSION_CODES#N API 24} will be able to participate
 * in the drag operation and receive the dragged content.
 *
 * <p>If this is the only flag set, then the drag recipient will only have access to text data
 * and intents contained in the {@link ClipData} object. Access to URIs contained in the
 * {@link ClipData} is determined by other DRAG_FLAG_GLOBAL_* flags</p>
 */
public static final int DRAG_FLAG_GLOBAL = 1 << 8;  // 256

/**
 * When this flag is used with {@link #DRAG_FLAG_GLOBAL}, the drag recipient will be able to
 * request read access to the content URI(s) contained in the {@link ClipData} object.
 * @see android.content.Intent#FLAG_GRANT_READ_URI_PERMISSION
 */
public static final int DRAG_FLAG_GLOBAL_URI_READ = Intent.FLAG_GRANT_READ_URI_PERMISSION;

/**
 * When this flag is used with {@link #DRAG_FLAG_GLOBAL}, the drag recipient will be able to
 * request write access to the content URI(s) contained in the {@link ClipData} object.
 * @see android.content.Intent#FLAG_GRANT_WRITE_URI_PERMISSION
 */
public static final int DRAG_FLAG_GLOBAL_URI_WRITE = Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

/**
 * When this flag is used with {@link #DRAG_FLAG_GLOBAL_URI_READ} and/or {@link
 * #DRAG_FLAG_GLOBAL_URI_WRITE}, the URI permission grant can be persisted across device
 * reboots until explicitly revoked with
 * {@link android.content.Context#revokeUriPermission(Uri, int)} Context.revokeUriPermission}.
 * @see android.content.Intent#FLAG_GRANT_PERSISTABLE_URI_PERMISSION
 */
```

```
public static final int DRAG_FLAG_GLOBAL_PERSISTABLE_URI_PERMISSION =
        Intent.FLAG_GRANT_PERSISTABLE_URI_PERMISSION;

/**
 * When this flag is used with {@link #DRAG_FLAG_GLOBAL_URI_READ} and/or {@link
 * #DRAG_FLAG_GLOBAL_URI_WRITE}, the URI permission grant applies to any URI that is a prefix
 * match against the original granted URI.
 * @see android.content.Intent#FLAG_GRANT_PREFIX_URI_PERMISSION
 */
public static final int DRAG_FLAG_GLOBAL_PREFIX_URI_PERMISSION =
        Intent.FLAG_GRANT_PREFIX_URI_PERMISSION;

/**
 * Flag indicating that the drag shadow will be opaque.  When
 * {@link #startDragAndDrop(ClipData, DragShadowBuilder, Object, int)} is called
 * with this flag set, the drag shadow will be opaque, otherwise, it will be semitransparent.
 */
public static final int DRAG_FLAG_OPAQUE = 1 << 9;

/**
 * Vertical scroll factor cached by {@link #getVerticalScrollFactor}.
 */
private float mVerticalScrollFactor;

/**
 * Position of the vertical scroll bar.
 */
private int mVerticalScrollbarPosition;

/**
 * Position the scroll bar at the default position as determined by the system.
 */
public static final int SCROLLBAR_POSITION_DEFAULT = 0;

/**
 * Position the scroll bar along the left edge.
 */
public static final int SCROLLBAR_POSITION_LEFT = 1;

/**
 * Position the scroll bar along the right edge.
 */
public static final int SCROLLBAR_POSITION_RIGHT = 2;

/**
 * Indicates that the view does not have a layer.
 *
 * @see #getLayerType()
 * @see #setLayerType(int, android.graphics.Paint)
 * @see #LAYER_TYPE_SOFTWARE
 * @see #LAYER_TYPE_HARDWARE
 */
public static final int LAYER_TYPE_NONE = 0;

/**
 * <p>Indicates that the view has a software layer. A software layer is backed
 * by a bitmap and causes the view to be rendered using Android's software
 * rendering pipeline, even if hardware acceleration is enabled.</p>
 *
 * <p>Software layers have various usages:</p>
 * <p>When the application is not using hardware acceleration, a software layer
 * is useful to apply a specific color filter and/or blending mode and/or
 * translucency to a view and all its children.</p>
 * <p>When the application is using hardware acceleration, a software layer
 * is useful to render drawing primitives not supported by the hardware
 * accelerated pipeline. It can also be used to cache a complex view tree
 * into a texture and reduce the complexity of drawing operations. For instance,
 * when animating a complex view tree with a translation, a software layer can
 * be used to render the view tree only once.</p>
 * <p>Software layers should be avoided when the affected view tree updates
 * often. Every update will require to re-render the software layer, which can
 * potentially be slow (particularly when hardware acceleration is turned on
 * since the layer will have to be uploaded into a hardware texture after every
 * update.)</p>
 *
 * @see #getLayerType()
 * @see #setLayerType(int, android.graphics.Paint)
 * @see #LAYER_TYPE_NONE
 * @see #LAYER_TYPE_HARDWARE
 */
public static final int LAYER_TYPE_SOFTWARE = 1;
```

```
/**
 * <p>Indicates that the view has a hardware layer. A hardware layer is backed
 * by a hardware specific texture (generally Frame Buffer Objects or FBO on
 * OpenGL hardware) and causes the view to be rendered using Android's hardware
 * rendering pipeline, but only if hardware acceleration is turned on for the
 * view hierarchy. When hardware acceleration is turned off, hardware layers
 * behave exactly as {@link #LAYER_TYPE_SOFTWARE software layers}.</p>
 *
 * <p>A hardware layer is useful to apply a specific color filter and/or
 * blending mode and/or translucency to a view and all its children.</p>
 * <p>A hardware layer can be used to cache a complex view tree into a
 * texture and reduce the complexity of drawing operations. For instance,
 * when animating a complex view tree with a translation, a hardware layer can
 * be used to render the view tree only once.</p>
 * <p>A hardware layer can also be used to increase the rendering quality when
 * rotation transformations are applied on a view. It can also be used to
 * prevent potential clipping issues when applying 3D transforms on a view.</p>
 *
 * @see #getLayerType()
 * @see #setLayerType(int, android.graphics.Paint)
 * @see #LAYER_TYPE_NONE
 * @see #LAYER_TYPE_SOFTWARE
 */
public static final int LAYER_TYPE_HARDWARE = 2;

@ViewDebug.ExportedProperty(category = "drawing", mapping = {
        @ViewDebug.IntToString(from = LAYER_TYPE_NONE, to = "NONE"),
        @ViewDebug.IntToString(from = LAYER_TYPE_SOFTWARE, to = "SOFTWARE"),
        @ViewDebug.IntToString(from = LAYER_TYPE_HARDWARE, to = "HARDWARE")
})
int mLayerType = LAYER_TYPE_NONE;
Paint mLayerPaint;

/**
 * Set to true when drawing cache is enabled and cannot be created.
 *
 * @hide
 */
public boolean mCachingFailed;
private Bitmap mDrawingCache;
private Bitmap mUnscaledDrawingCache;

/**
 * RenderNode holding View properties, potentially holding a DisplayList of View content.
 * <p>
 * When non-null and valid, this is expected to contain an up-to-date copy
 * of the View content. Its DisplayList content is cleared on temporary detach and reset on
 * cleanup.
 */
final RenderNode mRenderNode;

/**
 * Set to true when the view is sending hover accessibility events because it
 * is the innermost hovered view.
 */
private boolean mSendingHoverAccessibilityEvents;

/**
 * Delegate for injecting accessibility functionality.
 */
AccessibilityDelegate mAccessibilityDelegate;

/**
 * The view's overlay layer. Developers get a reference to the overlay via getOverlay()
 * and add/remove objects to/from the overlay directly through the Overlay methods.
 */
ViewOverlay mOverlay;

/**
 * The currently active parent view for receiving delegated nested scrolling events.
 * This is set by {@link #startNestedScroll(int)} during a touch interaction and cleared
 * by {@link #stopNestedScroll()} at the same point where we clear
 * requestDisallowInterceptTouchEvent.
 */
private ViewParent mNestedScrollingParent;

/**
 * Consistency verifier for debugging purposes.
 * @hide
 */
protected final InputEventConsistencyVerifier mInputEventConsistencyVerifier =
        InputEventConsistencyVerifier.isInstrumentationEnabled() ?
```

```
                new InputEventConsistencyVerifier(this, 0) : null;

    private static final AtomicInteger sNextGeneratedId = new AtomicInteger(1);

    private int[] mTempNestedScrollConsumed;

    /**
     * An overlay is going to draw this View instead of being drawn as part of this
     * View's parent. mGhostView is the View in the Overlay that must be invalidated
     * when this view is invalidated.
     */
    GhostView mGhostView;

    /**
     * Holds pairs of adjacent attribute data: attribute name followed by its value.
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "attributes", hasAdjacentMapping = true)
    public String[] mAttributes;

    /**
     * Maps a Resource id to its name.
     */
    private static SparseArray<String> mAttributeMap;

    /**
     * Queue of pending runnables. Used to postpone calls to post() until this
     * view is attached and has a handler.
     */
    private HandlerActionQueue mRunQueue;

    /**
     * The pointer icon when the mouse hovers on this view. The default is null.
     */
    private PointerIcon mPointerIcon;

    /**
     * @hide
     */
    String mStartActivityRequestWho;

    @Nullable
    private RoundScrollbarRenderer mRoundScrollbarRenderer;

    /** Used to delay visibility updates sent to the autofill manager */
    private Handler mVisibilityChangeForAutofillHandler;

    /**
     * Simple constructor to use when creating a view from code.
     *
     * @param context The Context the view is running in, through which it can
     *        access the current theme, resources, etc.
     */
    public View(Context context) {
        mContext = context;
        mResources = context != null ? context.getResources() : null;
        mViewFlags = SOUND_EFFECTS_ENABLED | HAPTIC_FEEDBACK_ENABLED | FOCUSABLE_AUTO;
        // Set some flags defaults
        mPrivateFlags2 =
                (LAYOUT_DIRECTION_DEFAULT << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT) |
                (TEXT_DIRECTION_DEFAULT << PFLAG2_TEXT_DIRECTION_MASK_SHIFT) |
                (PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT) |
                (TEXT_ALIGNMENT_DEFAULT << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT) |
                (PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT) |
                (IMPORTANT_FOR_ACCESSIBILITY_DEFAULT << PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT);
        mTouchSlop = ViewConfiguration.get(context).getScaledTouchSlop();
        setOverScrollMode(OVER_SCROLL_IF_CONTENT_SCROLLS);
        mUserPaddingStart = UNDEFINED_PADDING;
        mUserPaddingEnd = UNDEFINED_PADDING;
        mRenderNode = RenderNode.create(getClass().getName(), this);

        if (!sCompatibilityDone && context != null) {
            final int targetSdkVersion = context.getApplicationInfo().targetSdkVersion;

            // Older apps may need this compatibility hack for measurement.
            sUseBrokenMakeMeasureSpec = targetSdkVersion <= Build.VERSION_CODES.JELLY_BEAN_MR1;

            // Older apps expect onMeasure() to always be called on a layout pass, regardless
            // of whether a layout was requested on that View.
            sIgnoreMeasureCache = targetSdkVersion < Build.VERSION_CODES.KITKAT;

            Canvas.sCompatibilityRestore = targetSdkVersion < Build.VERSION_CODES.M;
```

```
        Canvas.sCompatibilitySetBitmap = targetSdkVersion < Build.VERSION_CODES.O;

        // In M and newer, our widgets can pass a "hint" value in the size
        // for UNSPECIFIED MeasureSpecs. This lets child views of scrolling containers
        // know what the expected parent size is going to be, so e.g. list items can size
        // themselves at 1/3 the size of their container. It breaks older apps though,
        // specifically apps that use some popular open source libraries.
        sUseZeroUnspecifiedMeasureSpec = targetSdkVersion < Build.VERSION_CODES.M;

        // Old versions of the platform would give different results from
        // LinearLayout measurement passes using EXACTLY and non-EXACTLY
        // modes, so we always need to run an additional EXACTLY pass.
        sAlwaysRemeasureExactly = targetSdkVersion <= Build.VERSION_CODES.M;

        // Prior to N, layout params could change without requiring a
        // subsequent call to setLayoutParams() and they would usually
        // work. Partial layout breaks this assumption.
        sLayoutParamsAlwaysChanged = targetSdkVersion <= Build.VERSION_CODES.M;

        // Prior to N, TextureView would silently ignore calls to setBackground/setForeground.
        // On N+, we throw, but that breaks compatibility with apps that use these methods.
        sTextureViewIgnoresDrawableSetters = targetSdkVersion <= Build.VERSION_CODES.M;

        // Prior to N, we would drop margins in LayoutParam conversions. The fix triggers bugs
        // in apps so we target check it to avoid breaking existing apps.
        sPreserveMarginParamsInLayoutParamConversion =
                targetSdkVersion >= Build.VERSION_CODES.N;

        sCascadedDragDrop = targetSdkVersion < Build.VERSION_CODES.N;

        sHasFocusableExcludeAutoFocusable = targetSdkVersion < Build.VERSION_CODES.O;

        sAutoFocusableOffUIThreadWontNotifyParents = targetSdkVersion < Build.VERSION_CODES.O;

        sUseDefaultFocusHighlight = context.getResources().getBoolean(
                com.android.internal.R.bool.config_useDefaultFocusHighlight);

        sCompatibilityDone = true;
    }
}

/**
 * Constructor that is called when inflating a view from XML. This is called
 * when a view is being constructed from an XML file, supplying attributes
 * that were specified in the XML file. This version uses a default style of
 * 0, so the only attribute values applied are those in the Context's Theme
 * and the given AttributeSet.
 *
 * <p>
 * The method onFinishInflate() will be called after all children have been
 * added.
 *
 * @param context The Context the view is running in, through which it can
 *        access the current theme, resources, etc.
 * @param attrs The attributes of the XML tag that is inflating the view.
 * @see #View(Context, AttributeSet, int)
 */
public View(Context context, @Nullable AttributeSet attrs) {
    this(context, attrs, 0);
}

/**
 * Perform inflation from XML and apply a class-specific base style from a
 * theme attribute. This constructor of View allows subclasses to use their
 * own base style when they are inflating. For example, a Button class's
 * constructor would call this version of the super class constructor and
 * supply <code>R.attr.buttonStyle</code> for <var>defStyleAttr</var>; this
 * allows the theme's button style to modify all of the base view attributes
 * (in particular its background) as well as the Button class's attributes.
 *
 * @param context The Context the view is running in, through which it can
 *        access the current theme, resources, etc.
 * @param attrs The attributes of the XML tag that is inflating the view.
 * @param defStyleAttr An attribute in the current theme that contains a
 *        reference to a style resource that supplies default values for
 *        the view. Can be 0 to not look for defaults.
 * @see #View(Context, AttributeSet)
 */
public View(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
    this(context, attrs, defStyleAttr, 0);
}
```

```java
/**
 * Perform inflation from XML and apply a class-specific base style from a
 * theme attribute or style resource. This constructor of View allows
 * subclasses to use their own base style when they are inflating.
 * <p>
 * When determining the final value of a particular attribute, there are
 * four inputs that come into play:
 * <ol>
 * <li>Any attribute values in the given AttributeSet.
 * <li>The style resource specified in the AttributeSet (named "style").
 * <li>The default style specified by <var>defStyleAttr</var>.
 * <li>The default style specified by <var>defStyleRes</var>.
 * <li>The base values in this theme.
 * </ol>
 * <p>
 * Each of these inputs is considered in-order, with the first listed taking
 * precedence over the following ones. In other words, if in the
 * AttributeSet you have supplied <code>&lt;Button * textColor="#ff000000"&gt;</code>
 * , then the button's text will <em>always</em> be black, regardless of
 * what is specified in any of the styles.
 *
 * @param context The Context the view is running in, through which it can
 *        access the current theme, resources, etc.
 * @param attrs The attributes of the XML tag that is inflating the view.
 * @param defStyleAttr An attribute in the current theme that contains a
 *        reference to a style resource that supplies default values for
 *        the view. Can be 0 to not look for defaults.
 * @param defStyleRes A resource identifier of a style resource that
 *        supplies default values for the view, used only if
 *        defStyleAttr is 0 or can not be found in the theme. Can be 0
 *        to not look for defaults.
 * @see #View(Context, AttributeSet, int)
 */
public View(Context context, @Nullable AttributeSet attrs, int defStyleAttr, int defStyleRes) {
    this(context);

    final TypedArray a = context.obtainStyledAttributes(
            attrs, com.android.internal.R.styleable.View, defStyleAttr, defStyleRes);

    if (mDebugViewAttributes) {
        saveAttributeData(attrs, a);
    }

    Drawable background = null;

    int leftPadding = -1;
    int topPadding = -1;
    int rightPadding = -1;
    int bottomPadding = -1;
    int startPadding = UNDEFINED_PADDING;
    int endPadding = UNDEFINED_PADDING;

    int padding = -1;
    int paddingHorizontal = -1;
    int paddingVertical = -1;

    int viewFlagValues = 0;
    int viewFlagMasks = 0;

    boolean setScrollContainer = false;

    int x = 0;
    int y = 0;

    float tx = 0;
    float ty = 0;
    float tz = 0;
    float elevation = 0;
    float rotation = 0;
    float rotationX = 0;
    float rotationY = 0;
    float sx = 1f;
    float sy = 1f;
    boolean transformSet = false;

    int scrollbarStyle = SCROLLBARS_INSIDE_OVERLAY;
    int overScrollMode = mOverScrollMode;
    boolean initializeScrollbars = false;
    boolean initializeScrollIndicators = false;

    boolean startPaddingDefined = false;
    boolean endPaddingDefined = false;
```

```
boolean leftPaddingDefined = false;
boolean rightPaddingDefined = false;

final int targetSdkVersion = context.getApplicationInfo().targetSdkVersion;

// Set default values.
viewFlagValues |= FOCUSABLE_AUTO;
viewFlagMasks |= FOCUSABLE_AUTO;

final int N = a.getIndexCount();
for (int i = 0; i < N; i++) {
    int attr = a.getIndex(i);
    switch (attr) {
        case com.android.internal.R.styleable.View_background:
            background = a.getDrawable(attr);
            break;
        case com.android.internal.R.styleable.View_padding:
            padding = a.getDimensionPixelSize(attr, -1);
            mUserPaddingLeftInitial = padding;
            mUserPaddingRightInitial = padding;
            leftPaddingDefined = true;
            rightPaddingDefined = true;
            break;
        case com.android.internal.R.styleable.View_paddingHorizontal:
            paddingHorizontal = a.getDimensionPixelSize(attr, -1);
            mUserPaddingLeftInitial = paddingHorizontal;
            mUserPaddingRightInitial = paddingHorizontal;
            leftPaddingDefined = true;
            rightPaddingDefined = true;
            break;
        case com.android.internal.R.styleable.View_paddingVertical:
            paddingVertical = a.getDimensionPixelSize(attr, -1);
            break;
         case com.android.internal.R.styleable.View_paddingLeft:
            leftPadding = a.getDimensionPixelSize(attr, -1);
            mUserPaddingLeftInitial = leftPadding;
            leftPaddingDefined = true;
            break;
        case com.android.internal.R.styleable.View_paddingTop:
            topPadding = a.getDimensionPixelSize(attr, -1);
            break;
        case com.android.internal.R.styleable.View_paddingRight:
            rightPadding = a.getDimensionPixelSize(attr, -1);
            mUserPaddingRightInitial = rightPadding;
            rightPaddingDefined = true;
            break;
        case com.android.internal.R.styleable.View_paddingBottom:
            bottomPadding = a.getDimensionPixelSize(attr, -1);
            break;
        case com.android.internal.R.styleable.View_paddingStart:
            startPadding = a.getDimensionPixelSize(attr, UNDEFINED_PADDING);
            startPaddingDefined = (startPadding != UNDEFINED_PADDING);
            break;
        case com.android.internal.R.styleable.View_paddingEnd:
            endPadding = a.getDimensionPixelSize(attr, UNDEFINED_PADDING);
            endPaddingDefined = (endPadding != UNDEFINED_PADDING);
            break;
        case com.android.internal.R.styleable.View_scrollX:
            x = a.getDimensionPixelOffset(attr, 0);
            break;
        case com.android.internal.R.styleable.View_scrollY:
            y = a.getDimensionPixelOffset(attr, 0);
            break;
        case com.android.internal.R.styleable.View_alpha:
            setAlpha(a.getFloat(attr, 1f));
            break;
        case com.android.internal.R.styleable.View_transformPivotX:
            setPivotX(a.getDimension(attr, 0));
            break;
        case com.android.internal.R.styleable.View_transformPivotY:
            setPivotY(a.getDimension(attr, 0));
            break;
        case com.android.internal.R.styleable.View_translationX:
            tx = a.getDimension(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_translationY:
            ty = a.getDimension(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_translationZ:
            tz = a.getDimension(attr, 0);
```

```
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_elevation:
            elevation = a.getDimension(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_rotation:
            rotation = a.getFloat(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_rotationX:
            rotationX = a.getFloat(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_rotationY:
            rotationY = a.getFloat(attr, 0);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_scaleX:
            sx = a.getFloat(attr, 1f);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_scaleY:
            sy = a.getFloat(attr, 1f);
            transformSet = true;
            break;
        case com.android.internal.R.styleable.View_id:
            mID = a.getResourceId(attr, NO_ID);
            break;
        case com.android.internal.R.styleable.View_tag:
            mTag = a.getText(attr);
            break;
        case com.android.internal.R.styleable.View_fitsSystemWindows:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= FITS_SYSTEM_WINDOWS;
                viewFlagMasks |= FITS_SYSTEM_WINDOWS;
            }
            break;
        case com.android.internal.R.styleable.View_focusable:
            viewFlagValues = (viewFlagValues & ~FOCUSABLE_MASK) | getFocusableAttribute(a);
            if ((viewFlagValues & FOCUSABLE_AUTO) == 0) {
                viewFlagMasks |= FOCUSABLE_MASK;
            }
            break;
        case com.android.internal.R.styleable.View_focusableInTouchMode:
            if (a.getBoolean(attr, false)) {
                // unset auto focus since focusableInTouchMode implies explicit focusable
                viewFlagValues &= ~FOCUSABLE_AUTO;
                viewFlagValues |= FOCUSABLE_IN_TOUCH_MODE | FOCUSABLE;
                viewFlagMasks |= FOCUSABLE_IN_TOUCH_MODE | FOCUSABLE_MASK;
            }
            break;
        case com.android.internal.R.styleable.View_clickable:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= CLICKABLE;
                viewFlagMasks |= CLICKABLE;
            }
            break;
        case com.android.internal.R.styleable.View_longClickable:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= LONG_CLICKABLE;
                viewFlagMasks |= LONG_CLICKABLE;
            }
            break;
        case com.android.internal.R.styleable.View_contextClickable:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= CONTEXT_CLICKABLE;
                viewFlagMasks |= CONTEXT_CLICKABLE;
            }
            break;
        case com.android.internal.R.styleable.View_saveEnabled:
            if (!a.getBoolean(attr, true)) {
                viewFlagValues |= SAVE_DISABLED;
                viewFlagMasks |= SAVE_DISABLED_MASK;
            }
            break;
        case com.android.internal.R.styleable.View_duplicateParentState:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= DUPLICATE_PARENT_STATE;
                viewFlagMasks |= DUPLICATE_PARENT_STATE;
            }
            break;
```

```
case com.android.internal.R.styleable.View_visibility:
    final int visibility = a.getInt(attr, 0);
    if (visibility != 0) {
        viewFlagValues |= VISIBILITY_FLAGS[visibility];
        viewFlagMasks |= VISIBILITY_MASK;
    }
    break;
case com.android.internal.R.styleable.View_layoutDirection:
    // Clear any layout direction flags (included resolved bits) already set
    mPrivateFlags2 &=
            ~(PFLAG2_LAYOUT_DIRECTION_MASK | PFLAG2_LAYOUT_DIRECTION_RESOLVED_MASK);
    // Set the layout direction flags depending on the value of the attribute
    final int layoutDirection = a.getInt(attr, -1);
    final int value = (layoutDirection != -1) ?
            LAYOUT_DIRECTION_FLAGS[layoutDirection] : LAYOUT_DIRECTION_DEFAULT;
    mPrivateFlags2 |= (value << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT);
    break;
case com.android.internal.R.styleable.View_drawingCacheQuality:
    final int cacheQuality = a.getInt(attr, 0);
    if (cacheQuality != 0) {
        viewFlagValues |= DRAWING_CACHE_QUALITY_FLAGS[cacheQuality];
        viewFlagMasks |= DRAWING_CACHE_QUALITY_MASK;
    }
    break;
case com.android.internal.R.styleable.View_contentDescription:
    setContentDescription(a.getString(attr));
    break;
case com.android.internal.R.styleable.View_accessibilityTraversalBefore:
    setAccessibilityTraversalBefore(a.getResourceId(attr, NO_ID));
    break;
case com.android.internal.R.styleable.View_accessibilityTraversalAfter:
    setAccessibilityTraversalAfter(a.getResourceId(attr, NO_ID));
    break;
case com.android.internal.R.styleable.View_labelFor:
    setLabelFor(a.getResourceId(attr, NO_ID));
    break;
case com.android.internal.R.styleable.View_soundEffectsEnabled:
    if (!a.getBoolean(attr, true)) {
        viewFlagValues &= ~SOUND_EFFECTS_ENABLED;
        viewFlagMasks |= SOUND_EFFECTS_ENABLED;
    }
    break;
case com.android.internal.R.styleable.View_hapticFeedbackEnabled:
    if (!a.getBoolean(attr, true)) {
        viewFlagValues &= ~HAPTIC_FEEDBACK_ENABLED;
        viewFlagMasks |= HAPTIC_FEEDBACK_ENABLED;
    }
    break;
case R.styleable.View_scrollbars:
    final int scrollbars = a.getInt(attr, SCROLLBARS_NONE);
    if (scrollbars != SCROLLBARS_NONE) {
        viewFlagValues |= scrollbars;
        viewFlagMasks |= SCROLLBARS_MASK;
        initializeScrollbars = true;
    }
    break;
//noinspection deprecation
case R.styleable.View_fadingEdge:
    if (targetSdkVersion >= Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
        // Ignore the attribute starting with ICS
        break;
    }
    // With builds < ICS, fall through and apply fading edges
case R.styleable.View_requiresFadingEdge:
    final int fadingEdge = a.getInt(attr, FADING_EDGE_NONE);
    if (fadingEdge != FADING_EDGE_NONE) {
        viewFlagValues |= fadingEdge;
        viewFlagMasks |= FADING_EDGE_MASK;
        initializeFadingEdgeInternal(a);
    }
    break;
case R.styleable.View_scrollbarStyle:
    scrollbarStyle = a.getInt(attr, SCROLLBARS_INSIDE_OVERLAY);
    if (scrollbarStyle != SCROLLBARS_INSIDE_OVERLAY) {
        viewFlagValues |= scrollbarStyle & SCROLLBARS_STYLE_MASK;
        viewFlagMasks |= SCROLLBARS_STYLE_MASK;
    }
    break;
case R.styleable.View_isScrollContainer:
    setScrollContainer = true;
    if (a.getBoolean(attr, false)) {
        setScrollContainer(true);
```

```
            }
            break;
        case com.android.internal.R.styleable.View_keepScreenOn:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= KEEP_SCREEN_ON;
                viewFlagMasks |= KEEP_SCREEN_ON;
            }
            break;
        case R.styleable.View_filterTouchesWhenObscured:
            if (a.getBoolean(attr, false)) {
                viewFlagValues |= FILTER_TOUCHES_WHEN_OBSCURED;
                viewFlagMasks |= FILTER_TOUCHES_WHEN_OBSCURED;
            }
            break;
        case R.styleable.View_nextFocusLeft:
            mNextFocusLeftId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_nextFocusRight:
            mNextFocusRightId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_nextFocusUp:
            mNextFocusUpId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_nextFocusDown:
            mNextFocusDownId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_nextFocusForward:
            mNextFocusForwardId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_nextClusterForward:
            mNextClusterForwardId = a.getResourceId(attr, View.NO_ID);
            break;
        case R.styleable.View_minWidth:
            mMinWidth = a.getDimensionPixelSize(attr, 0);
            break;
        case R.styleable.View_minHeight:
            mMinHeight = a.getDimensionPixelSize(attr, 0);
            break;
        case R.styleable.View_onClick:
            if (context.isRestricted()) {
                throw new IllegalStateException("The android:onClick attribute cannot "
                        + "be used within a restricted context");
            }

            final String handlerName = a.getString(attr);
            if (handlerName != null) {
                setOnClickListener(new DeclaredOnClickListener(this, handlerName));
            }
            break;
        case R.styleable.View_overScrollMode:
            overScrollMode = a.getInt(attr, OVER_SCROLL_IF_CONTENT_SCROLLS);
            break;
        case R.styleable.View_verticalScrollbarPosition:
            mVerticalScrollbarPosition = a.getInt(attr, SCROLLBAR_POSITION_DEFAULT);
            break;
        case R.styleable.View_layerType:
            setLayerType(a.getInt(attr, LAYER_TYPE_NONE), null);
            break;
        case R.styleable.View_textDirection:
            // Clear any text direction flag already set
            mPrivateFlags2 &= ~PFLAG2_TEXT_DIRECTION_MASK;
            // Set the text direction flags depending on the value of the attribute
            final int textDirection = a.getInt(attr, -1);
            if (textDirection != -1) {
                mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_FLAGS[textDirection];
            }
            break;
        case R.styleable.View_textAlignment:
            // Clear any text alignment flag already set
            mPrivateFlags2 &= ~PFLAG2_TEXT_ALIGNMENT_MASK;
            // Set the text alignment flag depending on the value of the attribute
            final int textAlignment = a.getInt(attr, TEXT_ALIGNMENT_DEFAULT);
            mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_FLAGS[textAlignment];
            break;
        case R.styleable.View_importantForAccessibility:
            setImportantForAccessibility(a.getInt(attr,
                    IMPORTANT_FOR_ACCESSIBILITY_DEFAULT));
            break;
        case R.styleable.View_accessibilityLiveRegion:
            setAccessibilityLiveRegion(a.getInt(attr, ACCESSIBILITY_LIVE_REGION_DEFAULT));
            break;
        case R.styleable.View_transitionName:
```

```
            setTransitionName(a.getString(attr));
            break;
        case R.styleable.View_nestedScrollingEnabled:
            setNestedScrollingEnabled(a.getBoolean(attr, false));
            break;
        case R.styleable.View_stateListAnimator:
            setStateListAnimator(AnimatorInflater.loadStateListAnimator(context,
                    a.getResourceId(attr, 0)));
            break;
        case R.styleable.View_backgroundTint:
            // This will get applied later during setBackground().
            if (mBackgroundTint == null) {
                mBackgroundTint = new TintInfo();
            }
            mBackgroundTint.mTintList = a.getColorStateList(
                    R.styleable.View_backgroundTint);
            mBackgroundTint.mHasTintList = true;
            break;
        case R.styleable.View_backgroundTintMode:
            // This will get applied later during setBackground().
            if (mBackgroundTint == null) {
                mBackgroundTint = new TintInfo();
            }
            mBackgroundTint.mTintMode = Drawable.parseTintMode(a.getInt(
                    R.styleable.View_backgroundTintMode, -1), null);
            mBackgroundTint.mHasTintMode = true;
            break;
        case R.styleable.View_outlineProvider:
            setOutlineProviderFromAttribute(a.getInt(R.styleable.View_outlineProvider,
                    PROVIDER_BACKGROUND));
            break;
        case R.styleable.View_foreground:
            if (targetSdkVersion >= Build.VERSION_CODES.M || this instanceof FrameLayout) {
                setForeground(a.getDrawable(attr));
            }
            break;
        case R.styleable.View_foregroundGravity:
            if (targetSdkVersion >= Build.VERSION_CODES.M || this instanceof FrameLayout) {
                setForegroundGravity(a.getInt(attr, Gravity.NO_GRAVITY));
            }
            break;
        case R.styleable.View_foregroundTintMode:
            if (targetSdkVersion >= Build.VERSION_CODES.M || this instanceof FrameLayout) {
                setForegroundTintMode(Drawable.parseTintMode(a.getInt(attr, -1), null));
            }
            break;
        case R.styleable.View_foregroundTint:
            if (targetSdkVersion >= Build.VERSION_CODES.M || this instanceof FrameLayout) {
                setForegroundTintList(a.getColorStateList(attr));
            }
            break;
        case R.styleable.View_foregroundInsidePadding:
            if (targetSdkVersion >= Build.VERSION_CODES.M || this instanceof FrameLayout) {
                if (mForegroundInfo == null) {
                    mForegroundInfo = new ForegroundInfo();
                }
                mForegroundInfo.mInsidePadding = a.getBoolean(attr,
                        mForegroundInfo.mInsidePadding);
            }
            break;
        case R.styleable.View_scrollIndicators:
            final int scrollIndicators =
                    (a.getInt(attr, 0) << SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT)
                            & SCROLL_INDICATORS_PFLAG3_MASK;
            if (scrollIndicators != 0) {
                mPrivateFlags3 |= scrollIndicators;
                initializeScrollIndicators = true;
            }
            break;
        case R.styleable.View_pointerIcon:
            final int resourceId = a.getResourceId(attr, 0);
            if (resourceId != 0) {
                setPointerIcon(PointerIcon.load(
                        context.getResources(), resourceId));
            } else {
                final int pointerType = a.getInt(attr, PointerIcon.TYPE_NOT_SPECIFIED);
                if (pointerType != PointerIcon.TYPE_NOT_SPECIFIED) {
                    setPointerIcon(PointerIcon.getSystemIcon(context, pointerType));
                }
            }
            break;
        case R.styleable.View_forceHasOverlappingRendering:
```

```
            if (a.peekValue(attr) != null) {
                forceHasOverlappingRendering(a.getBoolean(attr, true));
            }
            break;
        case R.styleable.View_tooltipText:
            setTooltipText(a.getText(attr));
            break;
        case R.styleable.View_keyboardNavigationCluster:
            if (a.peekValue(attr) != null) {
                setKeyboardNavigationCluster(a.getBoolean(attr, true));
            }
            break;
        case R.styleable.View_focusedByDefault:
            if (a.peekValue(attr) != null) {
                setFocusedByDefault(a.getBoolean(attr, true));
            }
            break;
        case R.styleable.View_autofillHints:
            if (a.peekValue(attr) != null) {
                CharSequence[] rawHints = null;
                String rawString = null;

                if (a.getType(attr) == TypedValue.TYPE_REFERENCE) {
                    int resId = a.getResourceId(attr, 0);

                    try {
                        rawHints = a.getTextArray(attr);
                    } catch (Resources.NotFoundException e) {
                        rawString = getResources().getString(resId);
                    }
                } else {
                    rawString = a.getString(attr);
                }

                if (rawHints == null) {
                    if (rawString == null) {
                        throw new IllegalArgumentException(
                                "Could not resolve autofillHints");
                    } else {
                        rawHints = rawString.split(",");
                    }
                }

                String[] hints = new String[rawHints.length];

                int numHints = rawHints.length;
                for (int rawHintNum = 0; rawHintNum < numHints; rawHintNum++) {
                    hints[rawHintNum] = rawHints[rawHintNum].toString().trim();
                }
                setAutofillHints(hints);
            }
            break;
        case R.styleable.View_importantForAutofill:
            if (a.peekValue(attr) != null) {
                setImportantForAutofill(a.getInt(attr, IMPORTANT_FOR_AUTOFILL_AUTO));
            }
            break;
        case R.styleable.View_defaultFocusHighlightEnabled:
            if (a.peekValue(attr) != null) {
                setDefaultFocusHighlightEnabled(a.getBoolean(attr, true));
            }
            break;
    }
}

setOverScrollMode(overScrollMode);

// Cache start/end user padding as we cannot fully resolve padding here (we dont have yet
// the resolved layout direction). Those cached values will be used later during padding
// resolution.
mUserPaddingStart = startPadding;
mUserPaddingEnd = endPadding;

if (background != null) {
    setBackground(background);
}

// setBackground above will record that padding is currently provided by the background.
// If we have padding specified via xml, record that here instead and use it.
mLeftPaddingDefined = leftPaddingDefined;
mRightPaddingDefined = rightPaddingDefined;
```

```
if (padding >= 0) {
    leftPadding = padding;
    topPadding = padding;
    rightPadding = padding;
    bottomPadding = padding;
    mUserPaddingLeftInitial = padding;
    mUserPaddingRightInitial = padding;
} else {
    if (paddingHorizontal >= 0) {
        leftPadding = paddingHorizontal;
        rightPadding = paddingHorizontal;
        mUserPaddingLeftInitial = paddingHorizontal;
        mUserPaddingRightInitial = paddingHorizontal;
    }
    if (paddingVertical >= 0) {
        topPadding = paddingVertical;
        bottomPadding = paddingVertical;
    }
}

if (isRtlCompatibilityMode()) {
    // RTL compatibility mode: pre Jelly Bean MR1 case OR no RTL support case.
    // left / right padding are used if defined (meaning here nothing to do). If they are not
    // defined and start / end padding are defined (e.g. in Frameworks resources), then we use
    // start / end and resolve them as left / right (layout direction is not taken into account).
    // Padding from the background drawable is stored at this point in mUserPaddingLeftInitial
    // and mUserPaddingRightInitial) so drawable padding will be used as ultimate default if
    // defined.
    if (!mLeftPaddingDefined && startPaddingDefined) {
        leftPadding = startPadding;
    }
    mUserPaddingLeftInitial = (leftPadding >= 0) ? leftPadding : mUserPaddingLeftInitial;
    if (!mRightPaddingDefined && endPaddingDefined) {
        rightPadding = endPadding;
    }
    mUserPaddingRightInitial = (rightPadding >= 0) ? rightPadding : mUserPaddingRightInitial;
} else {
    // Jelly Bean MR1 and after case: if start/end defined, they will override any left/right
    // values defined. Otherwise, left /right values are used.
    // Padding from the background drawable is stored at this point in mUserPaddingLeftInitial
    // and mUserPaddingRightInitial) so drawable padding will be used as ultimate default if
    // defined.
    final boolean hasRelativePadding = startPaddingDefined || endPaddingDefined;

    if (mLeftPaddingDefined && !hasRelativePadding) {
        mUserPaddingLeftInitial = leftPadding;
    }
    if (mRightPaddingDefined && !hasRelativePadding) {
        mUserPaddingRightInitial = rightPadding;
    }
}

internalSetPadding(
        mUserPaddingLeftInitial,
        topPadding >= 0 ? topPadding : mPaddingTop,
        mUserPaddingRightInitial,
        bottomPadding >= 0 ? bottomPadding : mPaddingBottom);

if (viewFlagMasks != 0) {
    setFlags(viewFlagValues, viewFlagMasks);
}

if (initializeScrollbars) {
    initializeScrollbarsInternal(a);
}

if (initializeScrollIndicators) {
    initializeScrollIndicatorsInternal();
}

a.recycle();

// Needs to be called after mViewFlags is set
if (scrollbarStyle != SCROLLBARS_INSIDE_OVERLAY) {
    recomputePadding();
}

if (x != 0 || y != 0) {
    scrollTo(x, y);
}

if (transformSet) {
```

```
        setTranslationX(tx);
        setTranslationY(ty);
        setTranslationZ(tz);
        setElevation(elevation);
        setRotation(rotation);
        setRotationX(rotationX);
        setRotationY(rotationY);
        setScaleX(sx);
        setScaleY(sy);
    }

    if (!setScrollContainer && (viewFlagValues&SCROLLBARS_VERTICAL) != 0) {
        setScrollContainer(true);
    }

    computeOpaqueFlags();
}

/**
 * An implementation of OnClickListener that attempts to lazily load a
 * named click handling method from a parent or ancestor context.
 */
private static class DeclaredOnClickListener implements OnClickListener {
    private final View mHostView;
    private final String mMethodName;

    private Method mResolvedMethod;
    private Context mResolvedContext;

    public DeclaredOnClickListener(@NonNull View hostView, @NonNull String methodName) {
        mHostView = hostView;
        mMethodName = methodName;
    }

    @Override
    public void onClick(@NonNull View v) {
        if (mResolvedMethod == null) {
            resolveMethod(mHostView.getContext(), mMethodName);
        }

        try {
            mResolvedMethod.invoke(mResolvedContext, v);
        } catch (IllegalAccessException e) {
            throw new IllegalStateException(
                    "Could not execute non-public method for android:onClick", e);
        } catch (InvocationTargetException e) {
            throw new IllegalStateException(
                    "Could not execute method for android:onClick", e);
        }
    }

    @NonNull
    private void resolveMethod(@Nullable Context context, @NonNull String name) {
        while (context != null) {
            try {
                if (!context.isRestricted()) {
                    final Method method = context.getClass().getMethod(mMethodName, View.class);
                    if (method != null) {
                        mResolvedMethod = method;
                        mResolvedContext = context;
                        return;
                    }
                }
            } catch (NoSuchMethodException e) {
                // Failed to find method, keep searching up the hierarchy.
            }

            if (context instanceof ContextWrapper) {
                context = ((ContextWrapper) context).getBaseContext();
            } else {
                // Can't search up the hierarchy, null out and fail.
                context = null;
            }
        }

        final int id = mHostView.getId();
        final String idText = id == NO_ID ? "" : " with id '"
                + mHostView.getContext().getResources().getResourceEntryName(id) + "'";
        throw new IllegalStateException("Could not find method " + mMethodName
                + "(View) in a parent or ancestor Context for android:onClick "
                + "attribute defined on view " + mHostView.getClass() + idText);
    }
```

```
}

/**
 * Non-public constructor for use in testing
 */
View() {
    mResources = null;
    mRenderNode = RenderNode.create(getClass().getName(), this);
}

final boolean debugDraw() {
    return DEBUG_DRAW || mAttachInfo != null && mAttachInfo.mDebugLayout;
}

private static SparseArray<String> getAttributeMap() {
    if (mAttributeMap == null) {
        mAttributeMap = new SparseArray<>();
    }
    return mAttributeMap;
}

private void saveAttributeData(@Nullable AttributeSet attrs, @NonNull TypedArray t) {
    final int attrsCount = attrs == null ? 0 : attrs.getAttributeCount();
    final int indexCount = t.getIndexCount();
    final String[] attributes = new String[(attrsCount + indexCount) * 2];

    int i = 0;

    // Store raw XML attributes.
    for (int j = 0; j < attrsCount; ++j) {
        attributes[i] = attrs.getAttributeName(j);
        attributes[i + 1] = attrs.getAttributeValue(j);
        i += 2;
    }

    // Store resolved styleable attributes.
    final Resources res = t.getResources();
    final SparseArray<String> attributeMap = getAttributeMap();
    for (int j = 0; j < indexCount; ++j) {
        final int index = t.getIndex(j);
        if (!t.hasValueOrEmpty(index)) {
            // Value is undefined. Skip it.
            continue;
        }

        final int resourceId = t.getResourceId(index, 0);
        if (resourceId == 0) {
            // Value is not a reference. Skip it.
            continue;
        }

        String resourceName = attributeMap.get(resourceId);
        if (resourceName == null) {
            try {
                resourceName = res.getResourceName(resourceId);
            } catch (Resources.NotFoundException e) {
                resourceName = "0x" + Integer.toHexString(resourceId);
            }
            attributeMap.put(resourceId, resourceName);
        }

        attributes[i] = resourceName;
        attributes[i + 1] = t.getString(index);
        i += 2;
    }

    // Trim to fit contents.
    final String[] trimmed = new String[i];
    System.arraycopy(attributes, 0, trimmed, 0, i);
    mAttributes = trimmed;
}

public String toString() {
    StringBuilder out = new StringBuilder(128);
    out.append(getClass().getName());
    out.append('{');
    out.append(Integer.toHexString(System.identityHashCode(this)));
    out.append(' ');
    switch (mViewFlags&VISIBILITY_MASK) {
        case VISIBLE: out.append('V'); break;
        case INVISIBLE: out.append('I'); break;
        case GONE: out.append('G'); break;
```

```
            default: out.append('.'); break;
        }
        out.append((mViewFlags & FOCUSABLE) == FOCUSABLE ? 'F' : '.');
        out.append((mViewFlags&ENABLED_MASK) == ENABLED ? 'E' : '.');
        out.append((mViewFlags&DRAW_MASK) == WILL_NOT_DRAW ? '.' : 'D');
        out.append((mViewFlags&SCROLLBARS_HORIZONTAL) != 0 ? 'H' : '.');
        out.append((mViewFlags&SCROLLBARS_VERTICAL) != 0 ? 'V' : '.');
        out.append((mViewFlags&CLICKABLE) != 0 ? 'C' : '.');
        out.append((mViewFlags&LONG_CLICKABLE) != 0 ? 'L' : '.');
        out.append((mViewFlags&CONTEXT_CLICKABLE) != 0 ? 'X' : '.');
        out.append(' ');
        out.append((mPrivateFlags&PFLAG_IS_ROOT_NAMESPACE) != 0 ? 'R' : '.');
        out.append((mPrivateFlags&PFLAG_FOCUSED) != 0 ? 'F' : '.');
        out.append((mPrivateFlags&PFLAG_SELECTED) != 0 ? 'S' : '.');
        if ((mPrivateFlags&PFLAG_PREPRESSED) != 0) {
            out.append('p');
        } else {
            out.append((mPrivateFlags&PFLAG_PRESSED) != 0 ? 'P' : '.');
        }
        out.append((mPrivateFlags&PFLAG_HOVERED) != 0 ? 'H' : '.');
        out.append((mPrivateFlags&PFLAG_ACTIVATED) != 0 ? 'A' : '.');
        out.append((mPrivateFlags&PFLAG_INVALIDATED) != 0 ? 'I' : '.');
        out.append((mPrivateFlags&PFLAG_DIRTY_MASK) != 0 ? 'D' : '.');
        out.append(' ');
        out.append(mLeft);
        out.append(',');
        out.append(mTop);
        out.append('-');
        out.append(mRight);
        out.append(',');
        out.append(mBottom);
        final int id = getId();
        if (id != NO_ID) {
            out.append(" #");
            out.append(Integer.toHexString(id));
            final Resources r = mResources;
            if (id > 0 && Resources.resourceHasPackage(id) && r != null) {
                try {
                    String pkgname;
                    switch (id&0xff000000) {
                        case 0x7f000000:
                            pkgname="app";
                            break;
                        case 0x01000000:
                            pkgname="android";
                            break;
                        default:
                            pkgname = r.getResourcePackageName(id);
                            break;
                    }
                    String typename = r.getResourceTypeName(id);
                    String entryname = r.getResourceEntryName(id);
                    out.append(" ");
                    out.append(pkgname);
                    out.append(":");
                    out.append(typename);
                    out.append("/");
                    out.append(entryname);
                } catch (Resources.NotFoundException e) {
                }
            }
        }
        out.append("}");
        return out.toString();
    }

    /**
     * <p>
     * Initializes the fading edges from a given set of styled attributes. This
     * method should be called by subclasses that need fading edges and when an
     * instance of these subclasses is created programmatically rather than
     * being inflated from XML. This method is automatically called when the XML
     * is inflated.
     * </p>
     *
     * @param a the styled attributes set to initialize the fading edges from
     *
     * @removed
     */
    protected void initializeFadingEdge(TypedArray a) {
        // This method probably shouldn't have been included in the SDK to begin with.
        // It relies on 'a' having been initialized using an attribute filter array that is
```

```
    // not publicly available to the SDK. The old method has been renamed
    // to initializeFadingEdgeInternal and hidden for framework use only;
    // this one initializes using defaults to make it safe to call for apps.

    TypedArray arr = mContext.obtainStyledAttributes(com.android.internal.R.styleable.View);

    initializeFadingEdgeInternal(arr);

    arr.recycle();
}

/**
 * <p>
 * Initializes the fading edges from a given set of styled attributes. This
 * method should be called by subclasses that need fading edges and when an
 * instance of these subclasses is created programmatically rather than
 * being inflated from XML. This method is automatically called when the XML
 * is inflated.
 * </p>
 *
 * @param a the styled attributes set to initialize the fading edges from
 * @hide This is the real method; the public one is shimmed to be safe to call from apps.
 */
protected void initializeFadingEdgeInternal(TypedArray a) {
    initScrollCache();

    mScrollCache.fadingEdgeLength = a.getDimensionPixelSize(
            R.styleable.View_fadingEdgeLength,
            ViewConfiguration.get(mContext).getScaledFadingEdgeLength());
}

/**
 * Returns the size of the vertical faded edges used to indicate that more
 * content in this view is visible.
 *
 * @return The size in pixels of the vertical faded edge or 0 if vertical
 *         faded edges are not enabled for this view.
 * @attr ref android.R.styleable#View_fadingEdgeLength
 */
public int getVerticalFadingEdgeLength() {
    if (isVerticalFadingEdgeEnabled()) {
        ScrollabilityCache cache = mScrollCache;
        if (cache != null) {
            return cache.fadingEdgeLength;
        }
    }
    return 0;
}

/**
 * Set the size of the faded edge used to indicate that more content in this
 * view is available.  Will not change whether the fading edge is enabled; use
 * {@link #setVerticalFadingEdgeEnabled(boolean)} or
 * {@link #setHorizontalFadingEdgeEnabled(boolean)} to enable the fading edge
 * for the vertical or horizontal fading edges.
 *
 * @param length The size in pixels of the faded edge used to indicate that more
 *        content in this view is visible.
 */
public void setFadingEdgeLength(int length) {
    initScrollCache();
    mScrollCache.fadingEdgeLength = length;
}

/**
 * Returns the size of the horizontal faded edges used to indicate that more
 * content in this view is visible.
 *
 * @return The size in pixels of the horizontal faded edge or 0 if horizontal
 *         faded edges are not enabled for this view.
 * @attr ref android.R.styleable#View_fadingEdgeLength
 */
public int getHorizontalFadingEdgeLength() {
    if (isHorizontalFadingEdgeEnabled()) {
        ScrollabilityCache cache = mScrollCache;
        if (cache != null) {
            return cache.fadingEdgeLength;
        }
    }
    return 0;
}
```

```
/**
 * Returns the width of the vertical scrollbar.
 *
 * @return The width in pixels of the vertical scrollbar or 0 if there
 *         is no vertical scrollbar.
 */
public int getVerticalScrollbarWidth() {
    ScrollabilityCache cache = mScrollCache;
    if (cache != null) {
        ScrollBarDrawable scrollBar = cache.scrollBar;
        if (scrollBar != null) {
            int size = scrollBar.getSize(true);
            if (size <= 0) {
                size = cache.scrollBarSize;
            }
            return size;
        }
        return 0;
    }
    return 0;
}

/**
 * Returns the height of the horizontal scrollbar.
 *
 * @return The height in pixels of the horizontal scrollbar or 0 if
 *         there is no horizontal scrollbar.
 */
protected int getHorizontalScrollbarHeight() {
    ScrollabilityCache cache = mScrollCache;
    if (cache != null) {
        ScrollBarDrawable scrollBar = cache.scrollBar;
        if (scrollBar != null) {
            int size = scrollBar.getSize(false);
            if (size <= 0) {
                size = cache.scrollBarSize;
            }
            return size;
        }
        return 0;
    }
    return 0;
}

/**
 * <p>
 * Initializes the scrollbars from a given set of styled attributes. This
 * method should be called by subclasses that need scrollbars and when an
 * instance of these subclasses is created programmatically rather than
 * being inflated from XML. This method is automatically called when the XML
 * is inflated.
 * </p>
 *
 * @param a the styled attributes set to initialize the scrollbars from
 *
 * @removed
 */
protected void initializeScrollbars(TypedArray a) {
    // It's not safe to use this method from apps. The parameter 'a' must have been obtained
    // using the View filter array which is not available to the SDK. As such, internal
    // framework usage now uses initializeScrollbarsInternal and we grab a default
    // TypedArray with the right filter instead here.
    TypedArray arr = mContext.obtainStyledAttributes(com.android.internal.R.styleable.View);

    initializeScrollbarsInternal(arr);

    // We ignored the method parameter. Recycle the one we actually did use.
    arr.recycle();
}

/**
 * <p>
 * Initializes the scrollbars from a given set of styled attributes. This
 * method should be called by subclasses that need scrollbars and when an
 * instance of these subclasses is created programmatically rather than
 * being inflated from XML. This method is automatically called when the XML
 * is inflated.
 * </p>
 *
 * @param a the styled attributes set to initialize the scrollbars from
 * @hide
 */
```

```
protected void initializeScrollbarsInternal(TypedArray a) {
    initScrollCache();

    final ScrollabilityCache scrollabilityCache = mScrollCache;

    if (scrollabilityCache.scrollBar == null) {
        scrollabilityCache.scrollBar = new ScrollBarDrawable();
        scrollabilityCache.scrollBar.setState(getDrawableState());
        scrollabilityCache.scrollBar.setCallback(this);
    }

    final boolean fadeScrollbars = a.getBoolean(R.styleable.View_fadeScrollbars, true);

    if (!fadeScrollbars) {
        scrollabilityCache.state = ScrollabilityCache.ON;
    }
    scrollabilityCache.fadeScrollBars = fadeScrollbars;


    scrollabilityCache.scrollBarFadeDuration = a.getInt(
            R.styleable.View_scrollbarFadeDuration, ViewConfiguration
                    .getScrollBarFadeDuration());
    scrollabilityCache.scrollBarDefaultDelayBeforeFade = a.getInt(
            R.styleable.View_scrollbarDefaultDelayBeforeFade,
            ViewConfiguration.getScrollDefaultDelay());


    scrollabilityCache.scrollBarSize = a.getDimensionPixelSize(
            com.android.internal.R.styleable.View_scrollbarSize,
            ViewConfiguration.get(mContext).getScaledScrollBarSize());

    Drawable track = a.getDrawable(R.styleable.View_scrollbarTrackHorizontal);
    scrollabilityCache.scrollBar.setHorizontalTrackDrawable(track);

    Drawable thumb = a.getDrawable(R.styleable.View_scrollbarThumbHorizontal);
    if (thumb != null) {
        scrollabilityCache.scrollBar.setHorizontalThumbDrawable(thumb);
    }

    boolean alwaysDraw = a.getBoolean(R.styleable.View_scrollbarAlwaysDrawHorizontalTrack,
            false);
    if (alwaysDraw) {
        scrollabilityCache.scrollBar.setAlwaysDrawHorizontalTrack(true);
    }

    track = a.getDrawable(R.styleable.View_scrollbarTrackVertical);
    scrollabilityCache.scrollBar.setVerticalTrackDrawable(track);

    thumb = a.getDrawable(R.styleable.View_scrollbarThumbVertical);
    if (thumb != null) {
        scrollabilityCache.scrollBar.setVerticalThumbDrawable(thumb);
    }

    alwaysDraw = a.getBoolean(R.styleable.View_scrollbarAlwaysDrawVerticalTrack,
            false);
    if (alwaysDraw) {
        scrollabilityCache.scrollBar.setAlwaysDrawVerticalTrack(true);
    }

    // Apply layout direction to the new Drawables if needed
    final int layoutDirection = getLayoutDirection();
    if (track != null) {
        track.setLayoutDirection(layoutDirection);
    }
    if (thumb != null) {
        thumb.setLayoutDirection(layoutDirection);
    }

    // Re-apply user/background padding so that scrollbar(s) get added
    resolvePadding();
}

private void initializeScrollIndicatorsInternal() {
    // Some day maybe we'll break this into top/left/start/etc. and let the
    // client control it. Until then, you can have any scroll indicator you
    // want as long as it's a 1dp foreground-colored rectangle.
    if (mScrollIndicatorDrawable == null) {
        mScrollIndicatorDrawable = mContext.getDrawable(R.drawable.scroll_indicator_material);
    }
}

/**
```

```
     * <p>
     * Initalizes the scrollability cache if necessary.
     * </p>
     */
    private void initScrollCache() {
        if (mScrollCache == null) {
            mScrollCache = new ScrollabilityCache(ViewConfiguration.get(mContext), this);
        }
    }

    private ScrollabilityCache getScrollCache() {
        initScrollCache();
        return mScrollCache;
    }

    /**
     * Set the position of the vertical scroll bar. Should be one of
     * {@link #SCROLLBAR_POSITION_DEFAULT}, {@link #SCROLLBAR_POSITION_LEFT} or
     * {@link #SCROLLBAR_POSITION_RIGHT}.
     *
     * @param position Where the vertical scroll bar should be positioned.
     */
    public void setVerticalScrollbarPosition(int position) {
        if (mVerticalScrollbarPosition != position) {
            mVerticalScrollbarPosition = position;
            computeOpaqueFlags();
            resolvePadding();
        }
    }

    /**
     * @return The position where the vertical scroll bar will show, if applicable.
     * @see #setVerticalScrollbarPosition(int)
     */
    public int getVerticalScrollbarPosition() {
        return mVerticalScrollbarPosition;
    }

    boolean isOnScrollbar(float x, float y) {
        if (mScrollCache == null) {
            return false;
        }
        x += getScrollX();
        y += getScrollY();
        if (isVerticalScrollBarEnabled() && !isVerticalScrollBarHidden()) {
            final Rect touchBounds = mScrollCache.mScrollBarTouchBounds;
            getVerticalScrollBarBounds(null, touchBounds);
            if (touchBounds.contains((int) x, (int) y)) {
                return true;
            }
        }
        if (isHorizontalScrollBarEnabled()) {
            final Rect touchBounds = mScrollCache.mScrollBarTouchBounds;
            getHorizontalScrollBarBounds(null, touchBounds);
            if (touchBounds.contains((int) x, (int) y)) {
                return true;
            }
        }
        return false;
    }

    boolean isOnScrollbarThumb(float x, float y) {
        return isOnVerticalScrollbarThumb(x, y) || isOnHorizontalScrollbarThumb(x, y);
    }

    private boolean isOnVerticalScrollbarThumb(float x, float y) {
        if (mScrollCache == null) {
            return false;
        }
        if (isVerticalScrollBarEnabled() && !isVerticalScrollBarHidden()) {
            x += getScrollX();
            y += getScrollY();
            final Rect bounds = mScrollCache.mScrollBarBounds;
            final Rect touchBounds = mScrollCache.mScrollBarTouchBounds;
            getVerticalScrollBarBounds(bounds, touchBounds);
            final int range = computeVerticalScrollRange();
            final int offset = computeVerticalScrollOffset();
            final int extent = computeVerticalScrollExtent();
            final int thumbLength = ScrollBarUtils.getThumbLength(bounds.height(), bounds.width(),
                    extent, range);
            final int thumbOffset = ScrollBarUtils.getThumbOffset(bounds.height(), thumbLength,
                    extent, range, offset);
```

```
        final int thumbTop = bounds.top + thumbOffset;
        final int adjust = Math.max(mScrollCache.scrollBarMinTouchTarget - thumbLength, 0) / 2;
        if (x >= touchBounds.left && x <= touchBounds.right
                && y >= thumbTop - adjust && y <= thumbTop + thumbLength + adjust) {
            return true;
        }
    }
    return false;
}

private boolean isOnHorizontalScrollbarThumb(float x, float y) {
    if (mScrollCache == null) {
        return false;
    }
    if (isHorizontalScrollBarEnabled()) {
        x += getScrollX();
        y += getScrollY();
        final Rect bounds = mScrollCache.mScrollBarBounds;
        final Rect touchBounds = mScrollCache.mScrollBarTouchBounds;
        getHorizontalScrollBarBounds(bounds, touchBounds);
        final int range = computeHorizontalScrollRange();
        final int offset = computeHorizontalScrollOffset();
        final int extent = computeHorizontalScrollExtent();
        final int thumbLength = ScrollBarUtils.getThumbLength(bounds.width(), bounds.height(),
                extent, range);
        final int thumbOffset = ScrollBarUtils.getThumbOffset(bounds.width(), thumbLength,
                extent, range, offset);
        final int thumbLeft = bounds.left + thumbOffset;
        final int adjust = Math.max(mScrollCache.scrollBarMinTouchTarget - thumbLength, 0) / 2;
        if (x >= thumbLeft - adjust && x <= thumbLeft + thumbLength + adjust
                && y >= touchBounds.top && y <= touchBounds.bottom) {
            return true;
        }
    }
    return false;
}

boolean isDraggingScrollBar() {
    return mScrollCache != null
            && mScrollCache.mScrollBarDraggingState != ScrollabilityCache.NOT_DRAGGING;
}

/**
 * Sets the state of all scroll indicators.
 * <p>
 * See {@link #setScrollIndicators(int, int)} for usage information.
 *
 * @param indicators a bitmask of indicators that should be enabled, or
 *                   {@code 0} to disable all indicators
 * @see #setScrollIndicators(int, int)
 * @see #getScrollIndicators()
 * @attr ref android.R.styleable#View_scrollIndicators
 */
public void setScrollIndicators(@ScrollIndicators int indicators) {
    setScrollIndicators(indicators,
            SCROLL_INDICATORS_PFLAG3_MASK >>> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT);
}

/**
 * Sets the state of the scroll indicators specified by the mask. To change
 * all scroll indicators at once, see {@link #setScrollIndicators(int)}.
 * <p>
 * When a scroll indicator is enabled, it will be displayed if the view
 * can scroll in the direction of the indicator.
 * <p>
 * Multiple indicator types may be enabled or disabled by passing the
 * logical OR of the desired types. If multiple types are specified, they
 * will all be set to the same enabled state.
 * <p>
 * For example, to enable the top scroll indicatorExample: {@code setScrollIndicators
 *
 * @param indicators the indicator direction, or the logical OR of multiple
 *             indicator directions. One or more of:
 *             <ul>
 *               <li>{@link #SCROLL_INDICATOR_TOP}</li>
 *               <li>{@link #SCROLL_INDICATOR_BOTTOM}</li>
 *               <li>{@link #SCROLL_INDICATOR_LEFT}</li>
 *               <li>{@link #SCROLL_INDICATOR_RIGHT}</li>
 *               <li>{@link #SCROLL_INDICATOR_START}</li>
 *               <li>{@link #SCROLL_INDICATOR_END}</li>
 *             </ul>
 * @see #setScrollIndicators(int)
```

```
     * @see #getScrollIndicators()
     * @attr ref android.R.styleable#View_scrollIndicators
     */
    public void setScrollIndicators(@ScrollIndicators int indicators, @ScrollIndicators int mask) {
        // Shift and sanitize mask.
        mask <<= SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;
        mask &= SCROLL_INDICATORS_PFLAG3_MASK;

        // Shift and mask indicators.
        indicators <<= SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;
        indicators &= mask;

        // Merge with non-masked flags.
        final int updatedFlags = indicators | (mPrivateFlags3 & ~mask);

        if (mPrivateFlags3 != updatedFlags) {
            mPrivateFlags3 = updatedFlags;

            if (indicators != 0) {
                initializeScrollIndicatorsInternal();
            }
            invalidate();
        }
    }

    /**
     * Returns a bitmask representing the enabled scroll indicators.
     * <p>
     * For example, if the top and left scroll indicators are enabled and all
     * other indicators are disabled, the return value will be
     * {@code View.SCROLL_INDICATOR_TOP | View.SCROLL_INDICATOR_LEFT}.
     * <p>
     * To check whether the bottom scroll indicator is enabled, use the value
     * of {@code (getScrollIndicators() & View.SCROLL_INDICATOR_BOTTOM) != 0}.
     *
     * @return a bitmask representing the enabled scroll indicators
     */
    @ScrollIndicators
    public int getScrollIndicators() {
        return (mPrivateFlags3 & SCROLL_INDICATORS_PFLAG3_MASK)
                >>> SCROLL_INDICATORS_TO_PFLAGS3_LSHIFT;
    }

    ListenerInfo getListenerInfo() {
        if (mListenerInfo != null) {
            return mListenerInfo;
        }
        mListenerInfo = new ListenerInfo();
        return mListenerInfo;
    }

    /**
     * Register a callback to be invoked when the scroll X or Y positions of
     * this view change.
     * <p>
     * <b>Note:</b> Some views handle scrolling independently from View and may
     * have their own separate listeners for scroll-type events. For example,
     * {@link android.widget.ListView ListView} allows clients to register an
     * {@link android.widget.ListView#setOnScrollListener(android.widget.AbsListView.OnScrollListener) AbsListView.OnScrol
     * to listen for changes in list scroll position.
     *
     * @param l The listener to notify when the scroll X or Y position changes.
     * @see android.view.View#getScrollX()
     * @see android.view.View#getScrollY()
     */
    public void setOnScrollChangeListener(OnScrollChangeListener l) {
        getListenerInfo().mOnScrollChangeListener = l;
    }

    /**
     * Register a callback to be invoked when focus of this view changed.
     *
     * @param l The callback that will run.
     */
    public void setOnFocusChangeListener(OnFocusChangeListener l) {
        getListenerInfo().mOnFocusChangeListener = l;
    }

    /**
     * Add a listener that will be called when the bounds of the view change due to
     * layout processing.
     *
```

```
     * @param listener The listener that will be called when layout bounds change.
     */
    public void addOnLayoutChangeListener(OnLayoutChangeListener listener) {
        ListenerInfo li = getListenerInfo();
        if (li.mOnLayoutChangeListeners == null) {
            li.mOnLayoutChangeListeners = new ArrayList<OnLayoutChangeListener>();
        }
        if (!li.mOnLayoutChangeListeners.contains(listener)) {
            li.mOnLayoutChangeListeners.add(listener);
        }
    }

    /**
     * Remove a listener for layout changes.
     *
     * @param listener The listener for layout bounds change.
     */
    public void removeOnLayoutChangeListener(OnLayoutChangeListener listener) {
        ListenerInfo li = mListenerInfo;
        if (li == null || li.mOnLayoutChangeListeners == null) {
            return;
        }
        li.mOnLayoutChangeListeners.remove(listener);
    }

    /**
     * Add a listener for attach state changes.
     *
     * This listener will be called whenever this view is attached or detached
     * from a window. Remove the listener using
     * {@link #removeOnAttachStateChangeListener(OnAttachStateChangeListener)}.
     *
     * @param listener Listener to attach
     * @see #removeOnAttachStateChangeListener(OnAttachStateChangeListener)
     */
    public void addOnAttachStateChangeListener(OnAttachStateChangeListener listener) {
        ListenerInfo li = getListenerInfo();
        if (li.mOnAttachStateChangeListeners == null) {
            li.mOnAttachStateChangeListeners
                    = new CopyOnWriteArrayList<OnAttachStateChangeListener>();
        }
        li.mOnAttachStateChangeListeners.add(listener);
    }

    /**
     * Remove a listener for attach state changes. The listener will receive no further
     * notification of window attach/detach events.
     *
     * @param listener Listener to remove
     * @see #addOnAttachStateChangeListener(OnAttachStateChangeListener)
     */
    public void removeOnAttachStateChangeListener(OnAttachStateChangeListener listener) {
        ListenerInfo li = mListenerInfo;
        if (li == null || li.mOnAttachStateChangeListeners == null) {
            return;
        }
        li.mOnAttachStateChangeListeners.remove(listener);
    }

    /**
     * Returns the focus-change callback registered for this view.
     *
     * @return The callback, or null if one is not registered.
     */
    public OnFocusChangeListener getOnFocusChangeListener() {
        ListenerInfo li = mListenerInfo;
        return li != null ? li.mOnFocusChangeListener : null;
    }

    /**
     * Register a callback to be invoked when this view is clicked. If this view is not
     * clickable, it becomes clickable.
     *
     * @param l The callback that will run
     *
     * @see #setClickable(boolean)
     */
    public void setOnClickListener(@Nullable OnClickListener l) {
        if (!isClickable()) {
            setClickable(true);
        }
        getListenerInfo().mOnClickListener = l;
```

```
}

/**
 * Return whether this view has an attached OnClickListener.  Returns
 * true if there is a listener, false if there is none.
 */
public boolean hasOnClickListeners() {
    ListenerInfo li = mListenerInfo;
    return (li != null && li.mOnClickListener != null);
}

/**
 * Register a callback to be invoked when this view is clicked and held. If this view is not
 * long clickable, it becomes long clickable.
 *
 * @param l The callback that will run
 *
 * @see #setLongClickable(boolean)
 */
public void setOnLongClickListener(@Nullable OnLongClickListener l) {
    if (!isLongClickable()) {
        setLongClickable(true);
    }
    getListenerInfo().mOnLongClickListener = l;
}

/**
 * Register a callback to be invoked when this view is context clicked. If the view is not
 * context clickable, it becomes context clickable.
 *
 * @param l The callback that will run
 * @see #setContextClickable(boolean)
 */
public void setOnContextClickListener(@Nullable OnContextClickListener l) {
    if (!isContextClickable()) {
        setContextClickable(true);
    }
    getListenerInfo().mOnContextClickListener = l;
}

/**
 * Register a callback to be invoked when the context menu for this view is
 * being built. If this view is not long clickable, it becomes long clickable.
 *
 * @param l The callback that will run
 *
 */
public void setOnCreateContextMenuListener(OnCreateContextMenuListener l) {
    if (!isLongClickable()) {
        setLongClickable(true);
    }
    getListenerInfo().mOnCreateContextMenuListener = l;
}

/**
 * Set an observer to collect stats for each frame rendered for this view.
 *
 * @hide
 */
public void addFrameMetricsListener(Window window,
        Window.OnFrameMetricsAvailableListener listener,
        Handler handler) {
    if (mAttachInfo != null) {
        if (mAttachInfo.mThreadedRenderer != null) {
            if (mFrameMetricsObservers == null) {
                mFrameMetricsObservers = new ArrayList<>();
            }

            FrameMetricsObserver fmo = new FrameMetricsObserver(window,
                    handler.getLooper(), listener);
            mFrameMetricsObservers.add(fmo);
            mAttachInfo.mThreadedRenderer.addFrameMetricsObserver(fmo);
        } else {
            Log.w(VIEW_LOG_TAG, "View not hardware-accelerated. Unable to observe frame stats");
        }
    } else {
        if (mFrameMetricsObservers == null) {
            mFrameMetricsObservers = new ArrayList<>();
        }

        FrameMetricsObserver fmo = new FrameMetricsObserver(window,
                handler.getLooper(), listener);
```

```
            mFrameMetricsObservers.add(fmo);
        }
    }

    /**
     * Remove observer configured to collect frame stats for this view.
     *
     * @hide
     */
    public void removeFrameMetricsListener(
            Window.OnFrameMetricsAvailableListener listener) {
        ThreadedRenderer renderer = getThreadedRenderer();
        FrameMetricsObserver fmo = findFrameMetricsObserver(listener);
        if (fmo == null) {
            throw new IllegalArgumentException(
                    "attempt to remove OnFrameMetricsAvailableListener that was never added");
        }

        if (mFrameMetricsObservers != null) {
            mFrameMetricsObservers.remove(fmo);
            if (renderer != null) {
                renderer.removeFrameMetricsObserver(fmo);
            }
        }
    }

    private void registerPendingFrameMetricsObservers() {
        if (mFrameMetricsObservers != null) {
            ThreadedRenderer renderer = getThreadedRenderer();
            if (renderer != null) {
                for (FrameMetricsObserver fmo : mFrameMetricsObservers) {
                    renderer.addFrameMetricsObserver(fmo);
                }
            } else {
                Log.w(VIEW_LOG_TAG, "View not hardware-accelerated. Unable to observe frame stats");
            }
        }
    }

    private FrameMetricsObserver findFrameMetricsObserver(
            Window.OnFrameMetricsAvailableListener listener) {
        for (int i = 0; i < mFrameMetricsObservers.size(); i++) {
            FrameMetricsObserver observer = mFrameMetricsObservers.get(i);
            if (observer.mListener == listener) {
                return observer;
            }
        }

        return null;
    }

    /**
     * Call this view's OnClickListener, if it is defined.  Performs all normal
     * actions associated with clicking: reporting accessibility event, playing
     * a sound, etc.
     *
     * @return True there was an assigned OnClickListener that was called, false
     *         otherwise is returned.
     */
    public boolean performClick() {
        final boolean result;
        final ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnClickListener != null) {
            playSoundEffect(SoundEffectConstants.CLICK);
            li.mOnClickListener.onClick(this);
            result = true;
        } else {
            result = false;
        }

        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);

        notifyEnterOrExitForAutoFillIfNeeded(true);

        return result;
    }

    /**
     * Directly call any attached OnClickListener.  Unlike {@link #performClick()},
     * this only calls the listener, and does not do any associated clicking
     * actions like reporting an accessibility event.
     *
```

```
     * @return True there was an assigned OnClickListener that was called, false
     *         otherwise is returned.
     */
    public boolean callOnClick() {
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnClickListener != null) {
            li.mOnClickListener.onClick(this);
            return true;
        }
        return false;
    }

    /**
     * Calls this view's OnLongClickListener, if it is defined. Invokes the
     * context menu if the OnLongClickListener did not consume the event.
     *
     * @return {@code true} if one of the above receivers consumed the event,
     *         {@code false} otherwise
     */
    public boolean performLongClick() {
        return performLongClickInternal(mLongClickX, mLongClickY);
    }

    /**
     * Calls this view's OnLongClickListener, if it is defined. Invokes the
     * context menu if the OnLongClickListener did not consume the event,
     * anchoring it to an (x,y) coordinate.
     *
     * @param x x coordinate of the anchoring touch event, or {@link Float#NaN}
     *          to disable anchoring
     * @param y y coordinate of the anchoring touch event, or {@link Float#NaN}
     *          to disable anchoring
     * @return {@code true} if one of the above receivers consumed the event,
     *         {@code false} otherwise
     */
    public boolean performLongClick(float x, float y) {
        mLongClickX = x;
        mLongClickY = y;
        final boolean handled = performLongClick();
        mLongClickX = Float.NaN;
        mLongClickY = Float.NaN;
        return handled;
    }

    /**
     * Calls this view's OnLongClickListener, if it is defined. Invokes the
     * context menu if the OnLongClickListener did not consume the event,
     * optionally anchoring it to an (x,y) coordinate.
     *
     * @param x x coordinate of the anchoring touch event, or {@link Float#NaN}
     *          to disable anchoring
     * @param y y coordinate of the anchoring touch event, or {@link Float#NaN}
     *          to disable anchoring
     * @return {@code true} if one of the above receivers consumed the event,
     *         {@code false} otherwise
     */
    private boolean performLongClickInternal(float x, float y) {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_LONG_CLICKED);

        boolean handled = false;
        final ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnLongClickListener != null) {
            handled = li.mOnLongClickListener.onLongClick(View.this);
        }
        if (!handled) {
            final boolean isAnchored = !Float.isNaN(x) && !Float.isNaN(y);
            handled = isAnchored ? showContextMenu(x, y) : showContextMenu();
        }
        if ((mViewFlags & TOOLTIP) == TOOLTIP) {
            if (!handled) {
                handled = showLongClickTooltip((int) x, (int) y);
            }
        }
        if (handled) {
            performHapticFeedback(HapticFeedbackConstants.LONG_PRESS);
        }
        return handled;
    }

    /**
     * Call this view's OnContextClickListener, if it is defined.
     *
```

```
     * @param x the x coordinate of the context click
     * @param y the y coordinate of the context click
     * @return True if there was an assigned OnContextClickListener that consumed the event, false
     *         otherwise.
     */
    public boolean performContextClick(float x, float y) {
        return performContextClick();
    }

    /**
     * Call this view's OnContextClickListener, if it is defined.
     *
     * @return True if there was an assigned OnContextClickListener that consumed the event, false
     *         otherwise.
     */
    public boolean performContextClick() {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CONTEXT_CLICKED);

        boolean handled = false;
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnContextClickListener != null) {
            handled = li.mOnContextClickListener.onContextClick(View.this);
        }
        if (handled) {
            performHapticFeedback(HapticFeedbackConstants.CONTEXT_CLICK);
        }
        return handled;
    }

    /**
     * Performs button-related actions during a touch down event.
     *
     * @param event The event.
     * @return True if the down was consumed.
     *
     * @hide
     */
    protected boolean performButtonActionOnTouchDown(MotionEvent event) {
        if (event.isFromSource(InputDevice.SOURCE_MOUSE) &&
            (event.getButtonState() & MotionEvent.BUTTON_SECONDARY) != 0) {
            showContextMenu(event.getX(), event.getY());
            mPrivateFlags |= PFLAG_CANCEL_NEXT_UP_EVENT;
            return true;
        }
        return false;
    }

    /**
     * Shows the context menu for this view.
     *
     * @return {@code true} if the context menu was shown, {@code false}
     *         otherwise
     * @see #showContextMenu(float, float)
     */
    public boolean showContextMenu() {
        return getParent().showContextMenuForChild(this);
    }

    /**
     * Shows the context menu for this view anchored to the specified
     * view-relative coordinate.
     *
     * @param x the X coordinate in pixels relative to the view to which the
     *          menu should be anchored, or {@link Float#NaN} to disable anchoring
     * @param y the Y coordinate in pixels relative to the view to which the
     *          menu should be anchored, or {@link Float#NaN} to disable anchoring
     * @return {@code true} if the context menu was shown, {@code false}
     *         otherwise
     */
    public boolean showContextMenu(float x, float y) {
        return getParent().showContextMenuForChild(this, x, y);
    }

    /**
     * Start an action mode with the default type {@link ActionMode#TYPE_PRIMARY}.
     *
     * @param callback Callback that will control the lifecycle of the action mode
     * @return The new action mode if it is started, null otherwise
     *
     * @see ActionMode
     * @see #startActionMode(android.view.ActionMode.Callback, int)
     */
```

```
public ActionMode startActionMode(ActionMode.Callback callback) {
    return startActionMode(callback, ActionMode.TYPE_PRIMARY);
}

/**
 * Start an action mode with the given type.
 *
 * @param callback Callback that will control the lifecycle of the action mode
 * @param type One of {@link ActionMode#TYPE_PRIMARY} or {@link ActionMode#TYPE_FLOATING}.
 * @return The new action mode if it is started, null otherwise
 *
 * @see ActionMode
 */
public ActionMode startActionMode(ActionMode.Callback callback, int type) {
    ViewParent parent = getParent();
    if (parent == null) return null;
    try {
        return parent.startActionModeForChild(this, callback, type);
    } catch (AbstractMethodError ame) {
        // Older implementations of custom views might not implement this.
        return parent.startActionModeForChild(this, callback);
    }
}

/**
 * Call {@link Context#startActivityForResult(String, Intent, int, Bundle)} for the View's
 * Context, creating a unique View identifier to retrieve the result.
 *
 * @param intent The Intent to be started.
 * @param requestCode The request code to use.
 * @hide
 */
public void startActivityForResult(Intent intent, int requestCode) {
    mStartActivityRequestWho = "@android:view:" + System.identityHashCode(this);
    getContext().startActivityForResult(mStartActivityRequestWho, intent, requestCode, null);
}

/**
 * If this View corresponds to the calling who, dispatches the activity result.
 * @param who The identifier for the targeted View to receive the result.
 * @param requestCode The integer request code originally supplied to
 *                    startActivityForResult(), allowing you to identify who this
 *                    result came from.
 * @param resultCode The integer result code returned by the child activity
 *                   through its setResult().
 * @param data An Intent, which can return result data to the caller
 *               (various data can be attached to Intent "extras").
 * @return {@code true} if the activity result was dispatched.
 * @hide
 */
public boolean dispatchActivityResult(
        String who, int requestCode, int resultCode, Intent data) {
    if (mStartActivityRequestWho != null && mStartActivityRequestWho.equals(who)) {
        onActivityResult(requestCode, resultCode, data);
        mStartActivityRequestWho = null;
        return true;
    }
    return false;
}

/**
 * Receive the result from a previous call to {@link #startActivityForResult(Intent, int)}.
 *
 * @param requestCode The integer request code originally supplied to
 *                    startActivityForResult(), allowing you to identify who this
 *                    result came from.
 * @param resultCode The integer result code returned by the child activity
 *                   through its setResult().
 * @param data An Intent, which can return result data to the caller
 *               (various data can be attached to Intent "extras").
 * @hide
 */
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    // Do nothing.
}

/**
 * Register a callback to be invoked when a hardware key is pressed in this view.
 * Key presses in software input methods will generally not trigger the methods of
 * this listener.
 * @param l the key listener to attach to this view
 */
```

```
public void setOnKeyListener(OnKeyListener l) {
    getListenerInfo().mOnKeyListener = l;
}

/**
 * Register a callback to be invoked when a touch event is sent to this view.
 * @param l the touch listener to attach to this view
 */
public void setOnTouchListener(OnTouchListener l) {
    getListenerInfo().mOnTouchListener = l;
}

/**
 * Register a callback to be invoked when a generic motion event is sent to this view.
 * @param l the generic motion listener to attach to this view
 */
public void setOnGenericMotionListener(OnGenericMotionListener l) {
    getListenerInfo().mOnGenericMotionListener = l;
}

/**
 * Register a callback to be invoked when a hover event is sent to this view.
 * @param l the hover listener to attach to this view
 */
public void setOnHoverListener(OnHoverListener l) {
    getListenerInfo().mOnHoverListener = l;
}

/**
 * Register a drag event listener callback object for this View. The parameter is
 * an implementation of {@link android.view.View.OnDragListener}. To send a drag event to a
 * View, the system calls the
 * {@link android.view.View.OnDragListener#onDrag(View,DragEvent)} method.
 * @param l An implementation of {@link android.view.View.OnDragListener}.
 */
public void setOnDragListener(OnDragListener l) {
    getListenerInfo().mOnDragListener = l;
}

/**
 * Give this view focus. This will cause
 * {@link #onFocusChanged(boolean, int, android.graphics.Rect)} to be called.
 *
 * Note: this does not check whether this {@link View} should get focus, it just
 * gives it focus no matter what.  It should only be called internally by framework
 * code that knows what it is doing, namely {@link #requestFocus(int, Rect)}.
 *
 * @param direction values are {@link View#FOCUS_UP}, {@link View#FOCUS_DOWN},
 *        {@link View#FOCUS_LEFT} or {@link View#FOCUS_RIGHT}. This is the direction which
 *        focus moved when requestFocus() is called. It may not always
 *        apply, in which case use the default View.FOCUS_DOWN.
 * @param previouslyFocusedRect The rectangle of the view that had focus
 *        prior in this View's coordinate system.
 */
void handleFocusGainInternal(@FocusRealDirection int direction, Rect previouslyFocusedRect) {
    if (DBG) {
        System.out.println(this + " requestFocus()");
    }

    if ((mPrivateFlags & PFLAG_FOCUSED) == 0) {
        mPrivateFlags |= PFLAG_FOCUSED;

        View oldFocus = (mAttachInfo != null) ? getRootView().findFocus() : null;

        if (mParent != null) {
            mParent.requestChildFocus(this, this);
            updateFocusedInCluster(oldFocus, direction);
        }

        if (mAttachInfo != null) {
            mAttachInfo.mTreeObserver.dispatchOnGlobalFocusChange(oldFocus, this);
        }

        onFocusChanged(true, direction, previouslyFocusedRect);
        refreshDrawableState();
    }
}

/**
 * Sets this view's preference for reveal behavior when it gains focus.
 *
 * <p>When set to true, this is a signal to ancestor views in the hierarchy that
```

```
     * this view would prefer to be brought fully into view when it gains focus.
     * For example, a text field that a user is meant to type into. Other views such
     * as scrolling containers may prefer to opt-out of this behavior.</p>
     *
     * <p>The default value for views is true, though subclasses may change this
     * based on their preferred behavior.</p>
     *
     * @param revealOnFocus true to request reveal on focus in ancestors, false otherwise
     *
     * @see #getRevealOnFocusHint()
     */
    public final void setRevealOnFocusHint(boolean revealOnFocus) {
        if (revealOnFocus) {
            mPrivateFlags3 &= ~PFLAG3_NO_REVEAL_ON_FOCUS;
        } else {
            mPrivateFlags3 |= PFLAG3_NO_REVEAL_ON_FOCUS;
        }
    }

    /**
     * Returns this view's preference for reveal behavior when it gains focus.
     *
     * <p>When this method returns true for a child view requesting focus, ancestor
     * views responding to a focus change in {@link ViewParent#requestChildFocus(View, View)}
     * should make a best effort to make the newly focused child fully visible to the user.
     * When it returns false, ancestor views should preferably not disrupt scroll positioning or
     * other properties affecting visibility to the user as part of the focus change.</p>
     *
     * @return true if this view would prefer to become fully visible when it gains focus,
     *         false if it would prefer not to disrupt scroll positioning
     *
     * @see #setRevealOnFocusHint(boolean)
     */
    public final boolean getRevealOnFocusHint() {
        return (mPrivateFlags3 & PFLAG3_NO_REVEAL_ON_FOCUS) == 0;
    }

    /**
     * Populates <code>outRect</code> with the hotspot bounds. By default,
     * the hotspot bounds are identical to the screen bounds.
     *
     * @param outRect rect to populate with hotspot bounds
     * @hide Only for internal use by views and widgets.
     */
    public void getHotspotBounds(Rect outRect) {
        final Drawable background = getBackground();
        if (background != null) {
            background.getHotspotBounds(outRect);
        } else {
            getBoundsOnScreen(outRect);
        }
    }

    /**
     * Request that a rectangle of this view be visible on the screen,
     * scrolling if necessary just enough.
     *
     * <p>A View should call this if it maintains some notion of which part
     * of its content is interesting.  For example, a text editing view
     * should call this when its cursor moves.
     * <p>The Rectangle passed into this method should be in the View's content coordinate space.
     * It should not be affected by which part of the View is currently visible or its scroll
     * position.
     *
     * @param rectangle The rectangle in the View's content coordinate space
     * @return Whether any parent scrolled.
     */
    public boolean requestRectangleOnScreen(Rect rectangle) {
        return requestRectangleOnScreen(rectangle, false);
    }

    /**
     * Request that a rectangle of this view be visible on the screen,
     * scrolling if necessary just enough.
     *
     * <p>A View should call this if it maintains some notion of which part
     * of its content is interesting.  For example, a text editing view
     * should call this when its cursor moves.
     * <p>The Rectangle passed into this method should be in the View's content coordinate space.
     * It should not be affected by which part of the View is currently visible or its scroll
     * position.
     * <p>When <code>immediate</code> is set to true, scrolling will not be
```

```
     * animated.
     *
     * @param rectangle The rectangle in the View's content coordinate space
     * @param immediate True to forbid animated scrolling, false otherwise
     * @return Whether any parent scrolled.
     */
    public boolean requestRectangleOnScreen(Rect rectangle, boolean immediate) {
        if (mParent == null) {
            return false;
        }

        View child = this;

        RectF position = (mAttachInfo != null) ? mAttachInfo.mTmpTransformRect : new RectF();
        position.set(rectangle);

        ViewParent parent = mParent;
        boolean scrolled = false;
        while (parent != null) {
            rectangle.set((int) position.left, (int) position.top,
                    (int) position.right, (int) position.bottom);

            scrolled |= parent.requestChildRectangleOnScreen(child, rectangle, immediate);

            if (!(parent instanceof View)) {
                break;
            }

            // move it from child's content coordinate space to parent's content coordinate space
            position.offset(child.mLeft - child.getScrollX(), child.mTop -child.getScrollY());

            child = (View) parent;
            parent = child.getParent();
        }

        return scrolled;
    }

    /**
     * Called when this view wants to give up focus. If focus is cleared
     * {@link #onFocusChanged(boolean, int, android.graphics.Rect)} is called.
     * <p>
     * <strong>Note:</strong> When a View clears focus the framework is trying
     * to give focus to the first focusable View from the top. Hence, if this
     * View is the first from the top that can take focus, then all callbacks
     * related to clearing focus will be invoked after which the framework will
     * give focus to this view.
     * </p>
     */
    public void clearFocus() {
        if (DBG) {
            System.out.println(this + " clearFocus()");
        }

        clearFocusInternal(null, true, true);
    }

    /**
     * Clears focus from the view, optionally propagating the change up through
     * the parent hierarchy and requesting that the root view place new focus.
     *
     * @param propagate whether to propagate the change up through the parent
     *            hierarchy
     * @param refocus when propagate is true, specifies whether to request the
     *            root view place new focus
     */
    void clearFocusInternal(View focused, boolean propagate, boolean refocus) {
        if ((mPrivateFlags & PFLAG_FOCUSED) != 0) {
            mPrivateFlags &= ~PFLAG_FOCUSED;

            if (propagate && mParent != null) {
                mParent.clearChildFocus(this);
            }

            onFocusChanged(false, 0, null);
            refreshDrawableState();

            if (propagate && (!refocus || !rootViewRequestFocus())) {
                notifyGlobalFocusCleared(this);
            }
        }
    }
```

```
void notifyGlobalFocusCleared(View oldFocus) {
    if (oldFocus != null && mAttachInfo != null) {
        mAttachInfo.mTreeObserver.dispatchOnGlobalFocusChange(oldFocus, null);
    }
}

boolean rootViewRequestFocus() {
    final View root = getRootView();
    return root != null && root.requestFocus();
}

/**
 * Called internally by the view system when a new view is getting focus.
 * This is what clears the old focus.
 * <p>
 * <b>NOTE:</b> The parent view's focused child must be updated manually
 * after calling this method. Otherwise, the view hierarchy may be left in
 * an inconstent state.
 */
void unFocus(View focused) {
    if (DBG) {
        System.out.println(this + " unFocus()");
    }

    clearFocusInternal(focused, false, false);
}

/**
 * Returns true if this view has focus itself, or is the ancestor of the
 * view that has focus.
 *
 * @return True if this view has or contains focus, false otherwise.
 */
@ViewDebug.ExportedProperty(category = "focus")
public boolean hasFocus() {
    return (mPrivateFlags & PFLAG_FOCUSED) != 0;
}

/**
 * Returns true if this view is focusable or if it contains a reachable View
 * for which {@link #hasFocusable()} returns {@code true}. A "reachable hasFocusable()"
 * is a view whose parents do not block descendants focus.
 * Only {@link #VISIBLE} views are considered focusable.
 *
 * <p>As of {@link Build.VERSION_CODES#O} views that are determined to be focusable
 * through {@link #FOCUSABLE_AUTO} will also cause this method to return {@code true}.
 * Apps that declare a {@link android.content.pm.ApplicationInfo#targetSdkVersion} of
 * earlier than {@link Build.VERSION_CODES#O} will continue to see this method return
 * {@code false} for views not explicitly marked as focusable.
 * Use {@link #hasExplicitFocusable()} if you require the pre-{@link Build.VERSION_CODES#O}
 * behavior.</p>
 *
 * @return {@code true} if the view is focusable or if the view contains a focusable
 *         view, {@code false} otherwise
 *
 * @see ViewGroup#FOCUS_BLOCK_DESCENDANTS
 * @see ViewGroup#getTouchscreenBlocksFocus()
 * @see #hasExplicitFocusable()
 */
public boolean hasFocusable() {
    return hasFocusable(!sHasFocusableExcludeAutoFocusable, false);
}

/**
 * Returns true if this view is focusable or if it contains a reachable View
 * for which {@link #hasExplicitFocusable()} returns {@code true}.
 * A "reachable hasExplicitFocusable()" is a view whose parents do not block descendants focus.
 * Only {@link #VISIBLE} views for which {@link #getFocusable()} would return
 * {@link #FOCUSABLE} are considered focusable.
 *
 * <p>This method preserves the pre-{@link Build.VERSION_CODES#O} behavior of
 * {@link #hasFocusable()} in that only views explicitly set focusable will cause
 * this method to return true. A view set to {@link #FOCUSABLE_AUTO} that resolves
 * to focusable will not.</p>
 *
 * @return {@code true} if the view is focusable or if the view contains a focusable
 *         view, {@code false} otherwise
 *
 * @see #hasFocusable()
 */
public boolean hasExplicitFocusable() {
```

```
    return hasFocusable(false, true);
}

boolean hasFocusable(boolean allowAutoFocus, boolean dispatchExplicit) {
    if (!isFocusableInTouchMode()) {
        for (ViewParent p = mParent; p instanceof ViewGroup; p = p.getParent()) {
            final ViewGroup g = (ViewGroup) p;
            if (g.shouldBlockFocusForTouchscreen()) {
                return false;
            }
        }
    }

    // Invisible and gone views are never focusable.
    if ((mViewFlags & VISIBILITY_MASK) != VISIBLE) {
        return false;
    }

    // Only use effective focusable value when allowed.
    if ((allowAutoFocus || getFocusable() != FOCUSABLE_AUTO) && isFocusable()) {
        return true;
    }

    return false;
}

/**
 * Called by the view system when the focus state of this view changes.
 * When the focus change event is caused by directional navigation, direction
 * and previouslyFocusedRect provide insight into where the focus is coming from.
 * When overriding, be sure to call up through to the super class so that
 * the standard focus handling will occur.
 *
 * @param gainFocus True if the View has focus; false otherwise.
 * @param direction The direction focus has moved when requestFocus()
 *                  is called to give this view focus. Values are
 *                  {@link #FOCUS_UP}, {@link #FOCUS_DOWN}, {@link #FOCUS_LEFT},
 *                  {@link #FOCUS_RIGHT}, {@link #FOCUS_FORWARD}, or {@link #FOCUS_BACKWARD}.
 *                  It may not always apply, in which case use the default.
 * @param previouslyFocusedRect The rectangle, in this view's coordinate
 *        system, of the previously focused view.  If applicable, this will be
 *        passed in as finer grained information about where the focus is coming
 *        from (in addition to direction).  Will be <code>null</code> otherwise.
 */
@CallSuper
protected void onFocusChanged(boolean gainFocus, @FocusDirection int direction,
        @Nullable Rect previouslyFocusedRect) {
    if (gainFocus) {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_FOCUSED);
    } else {
        notifyViewAccessibilityStateChangedIfNeeded(
                AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
    }

    // Here we check whether we still need the default focus highlight, and switch it on/off.
    switchDefaultFocusHighlight();

    InputMethodManager imm = InputMethodManager.peekInstance();
    if (!gainFocus) {
        if (isPressed()) {
            setPressed(false);
        }
        if (imm != null && mAttachInfo != null && mAttachInfo.mHasWindowFocus) {
            imm.focusOut(this);
        }
        onFocusLost();
    } else if (imm != null && mAttachInfo != null && mAttachInfo.mHasWindowFocus) {
        imm.focusIn(this);
    }

    invalidate(true);
    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnFocusChangeListener != null) {
        li.mOnFocusChangeListener.onFocusChange(this, gainFocus);
    }

    if (mAttachInfo != null) {
        mAttachInfo.mKeyDispatchState.reset(this);
    }

    notifyEnterOrExitForAutoFillIfNeeded(gainFocus);
}
```

```
private void notifyEnterOrExitForAutoFillIfNeeded(boolean enter) {
    if (isAutofillable() && isAttachedToWindow()) {
        AutofillManager afm = getAutofillManager();
        if (afm != null) {
            if (enter && hasWindowFocus() && isFocused()) {
                // We have not been laid out yet, hence cannot evaluate
                // whether this view is visible to the user, we will do
                // the evaluation once layout is complete.
                if (!isLaidOut()) {
                    mPrivateFlags3 |= PFLAG3_NOTIFY_AUTOFILL_ENTER_ON_LAYOUT;
                } else if (isVisibleToUser()) {
                    afm.notifyViewEntered(this);
                }
            } else if (!hasWindowFocus() || !isFocused()) {
                afm.notifyViewExited(this);
            }
        }
    }
}

/**
 * Sends an accessibility event of the given type. If accessibility is
 * not enabled this method has no effect. The default implementation calls
 * {@link #onInitializeAccessibilityEvent(AccessibilityEvent)} first
 * to populate information about the event source (this View), then calls
 * {@link #dispatchPopulateAccessibilityEvent(AccessibilityEvent)} to
 * populate the text content of the event source including its descendants,
 * and last calls
 * {@link ViewParent#requestSendAccessibilityEvent(View, AccessibilityEvent)}
 * on its parent to request sending of the event to interested parties.
 * <p>
 * If an {@link AccessibilityDelegate} has been specified via calling
 * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
 * {@link AccessibilityDelegate#sendAccessibilityEvent(View, int)} is
 * responsible for handling this call.
 * </p>
 *
 * @param eventType The type of the event to send, as defined by several types from
 * {@link android.view.accessibility.AccessibilityEvent}, such as
 * {@link android.view.accessibility.AccessibilityEvent#TYPE_VIEW_CLICKED} or
 * {@link android.view.accessibility.AccessibilityEvent#TYPE_VIEW_HOVER_ENTER}.
 *
 * @see #onInitializeAccessibilityEvent(AccessibilityEvent)
 * @see #dispatchPopulateAccessibilityEvent(AccessibilityEvent)
 * @see ViewParent#requestSendAccessibilityEvent(View, AccessibilityEvent)
 * @see AccessibilityDelegate
 */
public void sendAccessibilityEvent(int eventType) {
    if (mAccessibilityDelegate != null) {
        mAccessibilityDelegate.sendAccessibilityEvent(this, eventType);
    } else {
        sendAccessibilityEventInternal(eventType);
    }
}

/**
 * Convenience method for sending a {@link AccessibilityEvent#TYPE_ANNOUNCEMENT}
 * {@link AccessibilityEvent} to make an announcement which is related to some
 * sort of a context change for which none of the events representing UI transitions
 * is a good fit. For example, announcing a new page in a book. If accessibility
 * is not enabled this method does nothing.
 *
 * @param text The announcement text.
 */
public void announceForAccessibility(CharSequence text) {
    if (AccessibilityManager.getInstance(mContext).isEnabled() && mParent != null) {
        AccessibilityEvent event = AccessibilityEvent.obtain(
                AccessibilityEvent.TYPE_ANNOUNCEMENT);
        onInitializeAccessibilityEvent(event);
        event.getText().add(text);
        event.setContentDescription(null);
        mParent.requestSendAccessibilityEvent(this, event);
    }
}

/**
 * @see #sendAccessibilityEvent(int)
 *
 * Note: Called from the default {@link AccessibilityDelegate}.
 *
 * @hide
```

```
     */
    public void sendAccessibilityEventInternal(int eventType) {
        if (AccessibilityManager.getInstance(mContext).isEnabled()) {
            sendAccessibilityEventUnchecked(AccessibilityEvent.obtain(eventType));
        }
    }

    /**
     * This method behaves exactly as {@link #sendAccessibilityEvent(int)} but
     * takes as an argument an empty {@link AccessibilityEvent} and does not
     * perform a check whether accessibility is enabled.
     * <p>
     * If an {@link AccessibilityDelegate} has been specified via calling
     * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
     * {@link AccessibilityDelegate#sendAccessibilityEventUnchecked(View, AccessibilityEvent)}
     * is responsible for handling this call.
     * </p>
     *
     * @param event The event to send.
     *
     * @see #sendAccessibilityEvent(int)
     */
    public void sendAccessibilityEventUnchecked(AccessibilityEvent event) {
        if (mAccessibilityDelegate != null) {
            mAccessibilityDelegate.sendAccessibilityEventUnchecked(this, event);
        } else {
            sendAccessibilityEventUncheckedInternal(event);
        }
    }

    /**
     * @see #sendAccessibilityEventUnchecked(AccessibilityEvent)
     *
     * Note: Called from the default {@link AccessibilityDelegate}.
     *
     * @hide
     */
    public void sendAccessibilityEventUncheckedInternal(AccessibilityEvent event) {
        if (!isShown()) {
            return;
        }
        onInitializeAccessibilityEvent(event);
        // Only a subset of accessibility events populates text content.
        if ((event.getEventType() & POPULATING_ACCESSIBILITY_EVENT_TYPES) != 0) {
            dispatchPopulateAccessibilityEvent(event);
        }
        // In the beginning we called #isShown(), so we know that getParent() is not null.
        ViewParent parent = getParent();
        if (parent != null) {
            getParent().requestSendAccessibilityEvent(this, event);
        }
    }

    /**
     * Dispatches an {@link AccessibilityEvent} to the {@link View} first and then
     * to its children for adding their text content to the event. Note that the
     * event text is populated in a separate dispatch path since we add to the
     * event not only the text of the source but also the text of all its descendants.
     * A typical implementation will call
     * {@link #onPopulateAccessibilityEvent(AccessibilityEvent)} on the this view
     * and then call the {@link #dispatchPopulateAccessibilityEvent(AccessibilityEvent)}
     * on each child. Override this method if custom population of the event text
     * content is required.
     * <p>
     * If an {@link AccessibilityDelegate} has been specified via calling
     * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
     * {@link AccessibilityDelegate#dispatchPopulateAccessibilityEvent(View, AccessibilityEvent)}
     * is responsible for handling this call.
     * </p>
     * <p>
     * <em>Note:</em> Accessibility events of certain types are not dispatched for
     * populating the event text via this method. For details refer to {@link AccessibilityEvent}.
     * </p>
     *
     * @param event The event.
     *
     * @return True if the event population was completed.
     */
    public boolean dispatchPopulateAccessibilityEvent(AccessibilityEvent event) {
        if (mAccessibilityDelegate != null) {
            return mAccessibilityDelegate.dispatchPopulateAccessibilityEvent(this, event);
        } else {
```

```
        return dispatchPopulateAccessibilityEventInternal(event);
    }
}

/**
 * @see #dispatchPopulateAccessibilityEvent(AccessibilityEvent)
 *
 * Note: Called from the default {@link AccessibilityDelegate}.
 *
 * @hide
 */
public boolean dispatchPopulateAccessibilityEventInternal(AccessibilityEvent event) {
    onPopulateAccessibilityEvent(event);
    return false;
}

/**
 * Called from {@link #dispatchPopulateAccessibilityEvent(AccessibilityEvent)}
 * giving a chance to this View to populate the accessibility event with its
 * text content. While this method is free to modify event
 * attributes other than text content, doing so should normally be performed in
 * {@link #onInitializeAccessibilityEvent(AccessibilityEvent)}.
 * <p>
 * Example: Adding formatted date string to an accessibility event in addition
 *          to the text added by the super implementation:
 * <pre> public void onPopulateAccessibilityEvent(AccessibilityEvent event) {
 *     super.onPopulateAccessibilityEvent(event);
 *     final int flags = DateUtils.FORMAT_SHOW_DATE | DateUtils.FORMAT_SHOW_WEEKDAY;
 *     String selectedDateUtterance = DateUtils.formatDateTime(mContext,
 *         mCurrentDate.getTimeInMillis(), flags);
 *     event.getText().add(selectedDateUtterance);
 * }</pre>
 * <p>
 * If an {@link AccessibilityDelegate} has been specified via calling
 * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
 * {@link AccessibilityDelegate#onPopulateAccessibilityEvent(View, AccessibilityEvent)}
 * is responsible for handling this call.
 * </p>
 * <p class="note"><strong>Note:</strong> Always call the super implementation before adding
 * information to the event, in case the default implementation has basic information to add.
 * </p>
 *
 * @param event The accessibility event which to populate.
 *
 * @see #sendAccessibilityEvent(int)
 * @see #dispatchPopulateAccessibilityEvent(AccessibilityEvent)
 */
@CallSuper
public void onPopulateAccessibilityEvent(AccessibilityEvent event) {
    if (mAccessibilityDelegate != null) {
        mAccessibilityDelegate.onPopulateAccessibilityEvent(this, event);
    } else {
        onPopulateAccessibilityEventInternal(event);
    }
}

/**
 * @see #onPopulateAccessibilityEvent(AccessibilityEvent)
 *
 * Note: Called from the default {@link AccessibilityDelegate}.
 *
 * @hide
 */
public void onPopulateAccessibilityEventInternal(AccessibilityEvent event) {
}

/**
 * Initializes an {@link AccessibilityEvent} with information about
 * this View which is the event source. In other words, the source of
 * an accessibility event is the view whose state change triggered firing
 * the event.
 * <p>
 * Example: Setting the password property of an event in addition
 *          to properties set by the super implementation:
 * <pre> public void onInitializeAccessibilityEvent(AccessibilityEvent event) {
 *     super.onInitializeAccessibilityEvent(event);
 *     event.setPassword(true);
 * }</pre>
 * <p>
 * If an {@link AccessibilityDelegate} has been specified via calling
 * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
 * {@link AccessibilityDelegate#onInitializeAccessibilityEvent(View, AccessibilityEvent)}
```

```
     * is responsible for handling this call.
     * </p>
     * <p class="note"><strong>Note:</strong> Always call the super implementation before adding
     * information to the event, in case the default implementation has basic information to add.
     * </p>
     * @param event The event to initialize.
     *
     * @see #sendAccessibilityEvent(int)
     * @see #dispatchPopulateAccessibilityEvent(AccessibilityEvent)
     */
    @CallSuper
    public void onInitializeAccessibilityEvent(AccessibilityEvent event) {
        if (mAccessibilityDelegate != null) {
            mAccessibilityDelegate.onInitializeAccessibilityEvent(this, event);
        } else {
            onInitializeAccessibilityEventInternal(event);
        }
    }

    /**
     * @see #onInitializeAccessibilityEvent(AccessibilityEvent)
     *
     * Note: Called from the default {@link AccessibilityDelegate}.
     *
     * @hide
     */
    public void onInitializeAccessibilityEventInternal(AccessibilityEvent event) {
        event.setSource(this);
        event.setClassName(getAccessibilityClassName());
        event.setPackageName(getContext().getPackageName());
        event.setEnabled(isEnabled());
        event.setContentDescription(mContentDescription);

        switch (event.getEventType()) {
            case AccessibilityEvent.TYPE_VIEW_FOCUSED: {
                ArrayList<View> focusablesTempList = (mAttachInfo != null)
                        ? mAttachInfo.mTempArrayList : new ArrayList<View>();
                getRootView().addFocusables(focusablesTempList, View.FOCUS_FORWARD, FOCUSABLES_ALL);
                event.setItemCount(focusablesTempList.size());
                event.setCurrentItemIndex(focusablesTempList.indexOf(this));
                if (mAttachInfo != null) {
                    focusablesTempList.clear();
                }
            } break;
            case AccessibilityEvent.TYPE_VIEW_TEXT_SELECTION_CHANGED: {
                CharSequence text = getIterableTextForAccessibility();
                if (text != null && text.length() > 0) {
                    event.setFromIndex(getAccessibilitySelectionStart());
                    event.setToIndex(getAccessibilitySelectionEnd());
                    event.setItemCount(text.length());
                }
            } break;
        }
    }

    /**
     * Returns an {@link AccessibilityNodeInfo} representing this view from the
     * point of view of an {@link android.accessibilityservice.AccessibilityService}.
     * This method is responsible for obtaining an accessibility node info from a
     * pool of reusable instances and calling
     * {@link #onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)} on this view to
     * initialize the former.
     * <p>
     * Note: The client is responsible for recycling the obtained instance by calling
     *       {@link AccessibilityNodeInfo#recycle()} to minimize object creation.
     * </p>
     *
     * @return A populated {@link AccessibilityNodeInfo}.
     *
     * @see AccessibilityNodeInfo
     */
    public AccessibilityNodeInfo createAccessibilityNodeInfo() {
        if (mAccessibilityDelegate != null) {
            return mAccessibilityDelegate.createAccessibilityNodeInfo(this);
        } else {
            return createAccessibilityNodeInfoInternal();
        }
    }

    /**
     * @see #createAccessibilityNodeInfo()
     *
```

```
     * @hide
     */
    public AccessibilityNodeInfo createAccessibilityNodeInfoInternal() {
        AccessibilityNodeProvider provider = getAccessibilityNodeProvider();
        if (provider != null) {
            return provider.createAccessibilityNodeInfo(AccessibilityNodeProvider.HOST_VIEW_ID);
        } else {
            AccessibilityNodeInfo info = AccessibilityNodeInfo.obtain(this);
            onInitializeAccessibilityNodeInfo(info);
            return info;
        }
    }

    /**
     * Initializes an {@link AccessibilityNodeInfo} with information about this view.
     * The base implementation sets:
     * <ul>
     *   <li>{@link AccessibilityNodeInfo#setParent(View)},</li>
     *   <li>{@link AccessibilityNodeInfo#setBoundsInParent(Rect)},</li>
     *   <li>{@link AccessibilityNodeInfo#setBoundsInScreen(Rect)},</li>
     *   <li>{@link AccessibilityNodeInfo#setPackageName(CharSequence)},</li>
     *   <li>{@link AccessibilityNodeInfo#setClassName(CharSequence)},</li>
     *   <li>{@link AccessibilityNodeInfo#setContentDescription(CharSequence)},</li>
     *   <li>{@link AccessibilityNodeInfo#setEnabled(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setClickable(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setFocusable(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setFocused(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setLongClickable(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setSelected(boolean)},</li>
     *   <li>{@link AccessibilityNodeInfo#setContextClickable(boolean)}</li>
     * </ul>
     * <p>
     * Subclasses should override this method, call the super implementation,
     * and set additional attributes.
     * </p>
     * <p>
     * If an {@link AccessibilityDelegate} has been specified via calling
     * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
     * {@link AccessibilityDelegate#onInitializeAccessibilityNodeInfo(View, AccessibilityNodeInfo)}
     * is responsible for handling this call.
     * </p>
     *
     * @param info The instance to initialize.
     */
    @CallSuper
    public void onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo info) {
        if (mAccessibilityDelegate != null) {
            mAccessibilityDelegate.onInitializeAccessibilityNodeInfo(this, info);
        } else {
            onInitializeAccessibilityNodeInfoInternal(info);
        }
    }

    /**
     * Gets the location of this view in screen coordinates.
     *
     * @param outRect The output location
     * @hide
     */
    public void getBoundsOnScreen(Rect outRect) {
        getBoundsOnScreen(outRect, false);
    }

    /**
     * Gets the location of this view in screen coordinates.
     *
     * @param outRect The output location
     * @param clipToParent Whether to clip child bounds to the parent ones.
     * @hide
     */
    public void getBoundsOnScreen(Rect outRect, boolean clipToParent) {
        if (mAttachInfo == null) {
            return;
        }

        RectF position = mAttachInfo.mTmpTransformRect;
        position.set(0, 0, mRight - mLeft, mBottom - mTop);
        mapRectFromViewToScreenCoords(position, clipToParent);
        outRect.set(Math.round(position.left), Math.round(position.top),
                Math.round(position.right), Math.round(position.bottom));
    }
```

```java
/**
 * Map a rectangle from view-relative coordinates to screen-relative coordinates
 *
 * @param rect The rectangle to be mapped
 * @param clipToParent Whether to clip child bounds to the parent ones.
 * @hide
 */
public void mapRectFromViewToScreenCoords(RectF rect, boolean clipToParent) {
    if (!hasIdentityMatrix()) {
        getMatrix().mapRect(rect);
    }

    rect.offset(mLeft, mTop);

    ViewParent parent = mParent;
    while (parent instanceof View) {
        View parentView = (View) parent;

        rect.offset(-parentView.mScrollX, -parentView.mScrollY);

        if (clipToParent) {
            rect.left = Math.max(rect.left, 0);
            rect.top = Math.max(rect.top, 0);
            rect.right = Math.min(rect.right, parentView.getWidth());
            rect.bottom = Math.min(rect.bottom, parentView.getHeight());
        }

        if (!parentView.hasIdentityMatrix()) {
            parentView.getMatrix().mapRect(rect);
        }

        rect.offset(parentView.mLeft, parentView.mTop);

        parent = parentView.mParent;
    }

    if (parent instanceof ViewRootImpl) {
        ViewRootImpl viewRootImpl = (ViewRootImpl) parent;
        rect.offset(0, -viewRootImpl.mCurScrollY);
    }

    rect.offset(mAttachInfo.mWindowLeft, mAttachInfo.mWindowTop);
}

/**
 * Return the class name of this object to be used for accessibility purposes.
 * Subclasses should only override this if they are implementing something that
 * should be seen as a completely new class of view when used by accessibility,
 * unrelated to the class it is deriving from.  This is used to fill in
 * {@link AccessibilityNodeInfo#setClassName AccessibilityNodeInfo.setClassName}.
 */
public CharSequence getAccessibilityClassName() {
    return View.class.getName();
}

/**
 * Called when assist structure is being retrieved from a view as part of
 * {@link android.app.Activity#onProvideAssistData Activity.onProvideAssistData}.
 * @param structure Fill in with structured view data.  The default implementation
 * fills in all data that can be inferred from the view itself.
 */
public void onProvideStructure(ViewStructure structure) {
    onProvideStructureForAssistOrAutofill(structure, false, 0);
}

/**
 * Populates a {@link ViewStructure} to fullfil an autofill request.
 *
 * <p>The structure should contain at least the following properties:
 * <ul>
 *   <li>Autofill id ({@link ViewStructure#setAutofillId(AutofillId, int)}).
 *   <li>Autofill type ({@link ViewStructure#setAutofillType(int)}).
 *   <li>Autofill value ({@link ViewStructure#setAutofillValue(AutofillValue)}).
 *   <li>Whether the data is sensitive ({@link ViewStructure#setDataIsSensitive(boolean)}).
 * </ul>
 *
 * <p>It's also recommended to set the following properties - the more properties the structure
 * has, the higher the changes of an {@link android.service.autofill.AutofillService} properly
 * using the structure:
 *
 * <ul>
 *   <li>Autofill hints ({@link ViewStructure#setAutofillHints(String[])}).
```

```
     *   <li>Autofill options ({@link ViewStructure#setAutofillOptions(CharSequence[])}) when the
     *       view can only be filled with predefined values (typically used when the autofill type
     *       is {@link #AUTOFILL_TYPE_LIST}).
     *   <li>Resource id ({@link ViewStructure#setId(int, String, String, String)}).
     *   <li>Class name ({@link ViewStructure#setClassName(String)}).
     *   <li>Content description ({@link ViewStructure#setContentDescription(CharSequence)}).
     *   <li>Visual properties such as visibility ({@link ViewStructure#setVisibility(int)}),
     *       dimensions ({@link ViewStructure#setDimens(int, int, int, int, int, int)}), and
     *       opacity ({@link ViewStructure#setOpaque(boolean)}).
     *   <li>For views representing text fields, text properties such as the text itself
     *       ({@link ViewStructure#setText(CharSequence)}), text hints
     *       ({@link ViewStructure#setHint(CharSequence)}, input type
     *       ({@link ViewStructure#setInputType(int)}),
     *   <li>For views representing HTML nodes, its web domain
     *       ({@link ViewStructure#setWebDomain(String)}) and HTML properties
     *       (({@link ViewStructure#setHtmlInfo(android.view.ViewStructure.HtmlInfo)}).
     * </ul>
     *
     * <p>The default implementation of this method already sets most of these properties based on
     * related {@link View} methods (for example, the autofill id is set using
     * {@link #getAutofillId()}, the autofill type set using {@link #getAutofillType()}, etc.),
     * and views in the standard Android widgets library also override it to set their
     * relevant properties (for example, {@link android.widget.TextView} already sets the text
     * properties), so it's recommended to only override this method
     * (and call {@code super.onProvideAutofillStructure()}) when:
     *
     * <ul>
     *   <li>The view contents does not include PII (Personally Identifiable Information), so it
     *       can call {@link ViewStructure#setDataIsSensitive(boolean)} passing {@code false}.
     *   <li>The view can only be autofilled with predefined options, so it can call
     *       {@link ViewStructure#setAutofillOptions(CharSequence[])}.
     * </ul>
     *
     * <p><b>Note:</b> The {@code left} and {@code top} values set in
     * {@link ViewStructure#setDimens(int, int, int, int, int, int)} must be relative to the next
     * {@link ViewGroup#isImportantForAutofill()} predecessor view included in the structure.
     *
     * <p>Views support the Autofill Framework mainly by:
     * <ul>
     *   <li>Providing the metadata defining what the view means and how it can be autofilled.
     *   <li>Notifying the Android System when the view value changed by calling
     *       {@link AutofillManager#notifyValueChanged(View)}.
     *   <li>Implementing the methods that autofill the view.
     * </ul>
     * <p>This method is responsible for the former; {@link #autofill(AutofillValue)} is responsible
     * for the latter.
     *
     * @param structure fill in with structured view data for autofill purposes.
     * @param flags optional flags.
     *
     * @see #AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS
     */
    public void onProvideAutofillStructure(ViewStructure structure, @AutofillFlags int flags) {
        onProvideStructureForAssistOrAutofill(structure, true, flags);
    }

    private void onProvideStructureForAssistOrAutofill(ViewStructure structure,
            boolean forAutofill, @AutofillFlags int flags) {
        final int id = mID;
        if (id != NO_ID && !isViewIdGenerated(id)) {
            String pkg, type, entry;
            try {
                final Resources res = getResources();
                entry = res.getResourceEntryName(id);
                type = res.getResourceTypeName(id);
                pkg = res.getResourcePackageName(id);
            } catch (Resources.NotFoundException e) {
                entry = type = pkg = null;
            }
            structure.setId(id, pkg, type, entry);
        } else {
            structure.setId(id, null, null, null);
        }

        if (forAutofill) {
            final @AutofillType int autofillType = getAutofillType();
            // Don't need to fill autofill info if view does not support it.
            // For example, only TextViews that are editable support autofill
            if (autofillType != AUTOFILL_TYPE_NONE) {
                structure.setAutofillType(autofillType);
                structure.setAutofillHints(getAutofillHints());
                structure.setAutofillValue(getAutofillValue());
```

```
        }
    }

    int ignoredParentLeft = 0;
    int ignoredParentTop = 0;
    if (forAutofill && (flags & AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS) == 0) {
        View parentGroup = null;

        ViewParent viewParent = getParent();
        if (viewParent instanceof View) {
            parentGroup = (View) viewParent;
        }

        while (parentGroup != null && !parentGroup.isImportantForAutofill()) {
            ignoredParentLeft += parentGroup.mLeft;
            ignoredParentTop += parentGroup.mTop;

            viewParent = parentGroup.getParent();
            if (viewParent instanceof View) {
                parentGroup = (View) viewParent;
            } else {
                break;
            }
        }
    }

    structure.setDimens(ignoredParentLeft + mLeft, ignoredParentTop + mTop, mScrollX, mScrollY,
            mRight - mLeft, mBottom - mTop);
    if (!forAutofill) {
        if (!hasIdentityMatrix()) {
            structure.setTransformation(getMatrix());
        }
        structure.setElevation(getZ());
    }
    structure.setVisibility(getVisibility());
    structure.setEnabled(isEnabled());
    if (isClickable()) {
        structure.setClickable(true);
    }
    if (isFocusable()) {
        structure.setFocusable(true);
    }
    if (isFocused()) {
        structure.setFocused(true);
    }
    if (isAccessibilityFocused()) {
        structure.setAccessibilityFocused(true);
    }
    if (isSelected()) {
        structure.setSelected(true);
    }
    if (isActivated()) {
        structure.setActivated(true);
    }
    if (isLongClickable()) {
        structure.setLongClickable(true);
    }
    if (this instanceof Checkable) {
        structure.setCheckable(true);
        if (((Checkable)this).isChecked()) {
            structure.setChecked(true);
        }
    }
    if (isOpaque()) {
        structure.setOpaque(true);
    }
    if (isContextClickable()) {
        structure.setContextClickable(true);
    }
    structure.setClassName(getAccessibilityClassName().toString());
    structure.setContentDescription(getContentDescription());
}

/**
 * Called when assist structure is being retrieved from a view as part of
 * {@link android.app.Activity#onProvideAssistData Activity.onProvideAssistData} to
 * generate additional virtual structure under this view.  The defaullt implementation
 * uses {@link #getAccessibilityNodeProvider()} to try to generate this from the
 * view's virtual accessibility nodes, if any.  You can override this for a more
 * optimal implementation providing this data.
 */
public void onProvideVirtualStructure(ViewStructure structure) {
```

```
    AccessibilityNodeProvider provider = getAccessibilityNodeProvider();
    if (provider != null) {
        AccessibilityNodeInfo info = createAccessibilityNodeInfo();
        structure.setChildCount(1);
        ViewStructure root = structure.newChild(0);
        populateVirtualStructure(root, provider, info);
        info.recycle();
    }
}

/**
 * Populates a {@link ViewStructure} containing virtual children to fullfil an autofill
 * request.
 *
 * <p>This method should be used when the view manages a virtual structure under this view. For
 * example, a view that draws input fields using {@link #draw(Canvas)}.
 *
 * <p>When implementing this method, subclasses must follow the rules below:
 *
 * <ul>
 *   <li>Add virtual children by calling the {@link ViewStructure#newChild(int)} or
 *       {@link ViewStructure#asyncNewChild(int)} methods, where the {@code id} is an unique id
 *       identifying the children in the virtual structure.
 *   <li>The children hierarchy can have multiple levels if necessary, but ideally it should
 *       exclude intermediate levels that are irrelevant for autofill; that would improve the
 *       autofill performance.
 *   <li>Also implement {@link #autofill(SparseArray)} to autofill the virtual
 *       children.
 *   <li>Set the autofill properties of the child structure as defined by
 *       {@link #onProvideAutofillStructure(ViewStructure, int)}, using
 *       {@link ViewStructure#setAutofillId(AutofillId, int)} to set its autofill id.
 *   <li>Call {@link android.view.autofill.AutofillManager#notifyViewEntered(View, int, Rect)}
 *       and/or {@link android.view.autofill.AutofillManager#notifyViewExited(View, int)}
 *       when the focused virtual child changed.
 *   <li>Call
 *    {@link android.view.autofill.AutofillManager#notifyValueChanged(View, int, AutofillValue)}
 *       when the value of a virtual child changed.
 *   <li>Call
 *    {@link
 *    android.view.autofill.AutofillManager#notifyViewVisibilityChanged(View, int, boolean)}
 *       when the visibility of a virtual child changed.
 *   <li>Call {@link AutofillManager#commit()} when the autofill context of the view structure
 *       changed and the current context should be committed (for example, when the user tapped
 *       a {@code SUBMIT} button in an HTML page).
 *   <li>Call {@link AutofillManager#cancel()} when the autofill context of the view structure
 *       changed and the current context should be canceled (for example, when the user tapped
 *       a {@code CANCEL} button in an HTML page).
 *   <li>Provide ways for users to manually request autofill by calling
 *       {@link AutofillManager#requestAutofill(View, int, Rect)}.
 *   <li>The {@code left} and {@code top} values set in
 *       {@link ViewStructure#setDimens(int, int, int, int, int, int)} must be relative to the
 *       next {@link ViewGroup#isImportantForAutofill()} predecessor view included in the
 *       structure.
 * </ul>
 *
 * <p>Views with virtual children support the Autofill Framework mainly by:
 * <ul>
 *   <li>Providing the metadata defining what the virtual children mean and how they can be
 *       autofilled.
 *   <li>Implementing the methods that autofill the virtual children.
 * </ul>
 * <p>This method is responsible for the former; {@link #autofill(SparseArray)} is responsible
 * for the latter.
 *
 * @param structure fill in with virtual children data for autofill purposes.
 * @param flags optional flags.
 *
 * @see #AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS
 */
public void onProvideAutofillVirtualStructure(ViewStructure structure, int flags) {
}

/**
 * Automatically fills the content of this view with the {@code value}.
 *
 * <p>Views support the Autofill Framework mainly by:
 * <ul>
 *   <li>Providing the metadata defining what the view means and how it can be autofilled.
 *   <li>Implementing the methods that autofill the view.
 * </ul>
 * <p>{@link #onProvideAutofillStructure(ViewStructure, int)} is responsible for the former,
 * this method is responsible for latter.
```

```
     *
     * <p>This method does nothing by default, but when overridden it typically:
     * <ol>
     *   <li>Checks if the provided value matches the expected type (which is defined by
     *       {@link #getAutofillType()}).
     *   <li>Checks if the view is editable - if it isn't, it should return right away.
     *   <li>Call the proper getter method on {@link AutofillValue} to fetch the actual value.
     *   <li>Pass the actual value to the equivalent setter in the view.
     * </ol>
     *
     * <p>For example, a text-field view could implement the method this way:
     *
     * <pre class="prettyprint">
     * &#64;Override
     * public void autofill(AutofillValue value) {
     *   if (!value.isText() || !this.isEditable()) {
     *      return;
     *   }
     *   CharSequence text = value.getTextValue();
     *   if (text != null) {
     *     this.setText(text);
     *   }
     * }
     * </pre>
     *
     * <p>If the value is updated asynchronously, the next call to
     * {@link AutofillManager#notifyValueChanged(View)} must happen <b>after</b> the value was
     * changed to the autofilled value. If not, the view will not be considered autofilled.
     *
     * <p><b>Note:</b> After this method is called, the value returned by
     * {@link #getAutofillValue()} must be equal to the {@code value} passed to it, otherwise the
     * view will not be highlighted as autofilled.
     *
     * @param value value to be autofilled.
     */
    public void autofill(@SuppressWarnings("unused") AutofillValue value) {
    }

    /**
     * Automatically fills the content of the virtual children within this view.
     *
     * <p>Views with virtual children support the Autofill Framework mainly by:
     * <ul>
     *   <li>Providing the metadata defining what the virtual children mean and how they can be
     *       autofilled.
     *   <li>Implementing the methods that autofill the virtual children.
     * </ul>
     * <p>{@link #onProvideAutofillVirtualStructure(ViewStructure, int)} is responsible for the
     * former, this method is responsible for the latter - see {@link #autofill(AutofillValue)} and
     * {@link #onProvideAutofillVirtualStructure(ViewStructure, int)} for more info about autofill.
     *
     * <p>If a child value is updated asynchronously, the next call to
     * {@link AutofillManager#notifyValueChanged(View, int, AutofillValue)} must happen
     * <b>after</b> the value was changed to the autofilled value. If not, the child will not be
     * considered autofilled.
     *
     * <p><b>Note:</b> To indicate that a virtual view was autofilled,
     * <code>?android:attr/autofilledHighlight</code> should be drawn over it until the data
     * changes.
     *
     * @param values map of values to be autofilled, keyed by virtual child id.
     *
     * @attr ref android.R.styleable#Theme_autofilledHighlight
     */
    public void autofill(@NonNull @SuppressWarnings("unused") SparseArray<AutofillValue> values) {
    }

    /**
     * Gets the unique identifier of this view in the screen, for autofill purposes.
     *
     * @return The View's autofill id.
     */
    public final AutofillId getAutofillId() {
        if (mAutofillId == null) {
            // The autofill id needs to be unique, but its value doesn't matter,
            // so it's better to reuse the accessibility id to save space.
            mAutofillId = new AutofillId(getAutofillViewId());
        }
        return mAutofillId;
    }

    /**
```

```
     * Describes the autofill type of this view, so an
     * {@link android.service.autofill.AutofillService} can create the proper {@link AutofillValue}
     * when autofilling the view.
     *
     * <p>By default returns {@link #AUTOFILL_TYPE_NONE}, but views should override it to properly
     * support the Autofill Framework.
     *
     * @return either {@link #AUTOFILL_TYPE_NONE}, {@link #AUTOFILL_TYPE_TEXT},
     * {@link #AUTOFILL_TYPE_LIST}, {@link #AUTOFILL_TYPE_DATE}, or {@link #AUTOFILL_TYPE_TOGGLE}.
     *
     * @see #onProvideAutofillStructure(ViewStructure, int)
     * @see #autofill(AutofillValue)
     */
    public @AutofillType int getAutofillType() {
        return AUTOFILL_TYPE_NONE;
    }

    /**
     * Gets the hints that help an {@link android.service.autofill.AutofillService} determine how
     * to autofill the view with the user's data.
     *
     * <p>See {@link #setAutofillHints(String...)} for more info about these hints.
     *
     * @return The hints set via the attribute or {@link #setAutofillHints(String...)}, or
     * {@code null} if no hints were set.
     *
     * @attr ref android.R.styleable#View_autofillHints
     */
    @ViewDebug.ExportedProperty()
    @Nullable public String[] getAutofillHints() {
        return mAutofillHints;
    }

    /**
     * @hide
     */
    public boolean isAutofilled() {
        return (mPrivateFlags3 & PFLAG3_IS_AUTOFILLED) != 0;
    }

    /**
     * Gets the {@link View}'s current autofill value.
     *
     * <p>By default returns {@code null}, but subclasses should override it and return an
     * appropriate value to properly support the Autofill Framework.
     *
     * @see #onProvideAutofillStructure(ViewStructure, int)
     * @see #autofill(AutofillValue)
     */
    @Nullable
    public AutofillValue getAutofillValue() {
        return null;
    }

    /**
     * Gets the mode for determining whether this view is important for autofill.
     *
     * <p>See {@link #setImportantForAutofill(int)} and {@link #isImportantForAutofill()} for more
     * info about this mode.
     *
     * @return {@link #IMPORTANT_FOR_AUTOFILL_AUTO} by default, or value passed to
     * {@link #setImportantForAutofill(int)}.
     *
     * @attr ref android.R.styleable#View_importantForAutofill
     */
    @ViewDebug.ExportedProperty(mapping = {
            @ViewDebug.IntToString(from = IMPORTANT_FOR_AUTOFILL_AUTO, to = "auto"),
            @ViewDebug.IntToString(from = IMPORTANT_FOR_AUTOFILL_YES, to = "yes"),
            @ViewDebug.IntToString(from = IMPORTANT_FOR_AUTOFILL_NO, to = "no"),
            @ViewDebug.IntToString(from = IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS,
                    to = "yesExcludeDescendants"),
            @ViewDebug.IntToString(from = IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS,
                    to = "noExcludeDescendants")})
    public @AutofillImportance int getImportantForAutofill() {
        return (mPrivateFlags3
                & PFLAG3_IMPORTANT_FOR_AUTOFILL_MASK) >> PFLAG3_IMPORTANT_FOR_AUTOFILL_SHIFT;
    }

    /**
     * Sets the mode for determining whether this view is considered important for autofill.
     *
     * <p>The platform determines the importance for autofill automatically but you
```

```
     * can use this method to customize the behavior. For example:
     *
     * <ol>
     *   <li>When the view contents is irrelevant for autofill (for example, a text field used in a
     *       "Captcha" challenge), it should be {@link #IMPORTANT_FOR_AUTOFILL_NO}.
     *   <li>When both the view and its children are irrelevant for autofill (for example, the root
     *       view of an activity containing a spreadhseet editor), it should be
     *       {@link #IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS}.
     *   <li>When the view content is relevant for autofill but its children aren't (for example,
     *       a credit card expiration date represented by a custom view that overrides the proper
     *       autofill methods and has 2 children representing the month and year), it should
     *       be {@link #IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS}.
     * </ol>
     *
     * <p><b>Note:</b> Setting the mode as {@link #IMPORTANT_FOR_AUTOFILL_NO} or
     * {@link #IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS} does not guarantee the view (and its
     * children) will be always be considered not important; for example, when the user explicitly
     * makes an autofill request, all views are considered important. See
     * {@link #isImportantForAutofill()} for more details about how the View's importance for
     * autofill is used.
     *
     * @param mode {@link #IMPORTANT_FOR_AUTOFILL_AUTO}, {@link #IMPORTANT_FOR_AUTOFILL_YES},
     * {@link #IMPORTANT_FOR_AUTOFILL_NO}, {@link #IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS},
     * or {@link #IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS}.
     *
     * @attr ref android.R.styleable#View_importantForAutofill
     */
    public void setImportantForAutofill(@AutofillImportance int mode) {
        mPrivateFlags3 &= ~PFLAG3_IMPORTANT_FOR_AUTOFILL_MASK;
        mPrivateFlags3 |= (mode << PFLAG3_IMPORTANT_FOR_AUTOFILL_SHIFT)
                & PFLAG3_IMPORTANT_FOR_AUTOFILL_MASK;
    }

    /**
     * Hints the Android System whether the {@link android.app.assist.AssistStructure.ViewNode}
     * associated with this view is considered important for autofill purposes.
     *
     * <p>Generally speaking, a view is important for autofill if:
     * <ol>
     * <li>The view can be autofilled by an {@link android.service.autofill.AutofillService}.
     * <li>The view contents can help an {@link android.service.autofill.AutofillService}
     *     determine how other views can be autofilled.
     * <ol>
     *
     * <p>For example, view containers should typically return {@code false} for performance reasons
     * (since the important info is provided by their children), but if its properties have relevant
     * information (for example, a resource id called {@code credentials}, it should return
     * {@code true}. On the other hand, views representing labels or editable fields should
     * typically return {@code true}, but in some cases they could return {@code false}
     * (for example, if they're part of a "Captcha" mechanism).
     *
     * <p>The value returned by this method depends on the value returned by
     * {@link #getImportantForAutofill()}:
     *
     * <ol>
     *   <li>if it returns {@link #IMPORTANT_FOR_AUTOFILL_YES} or
     *       {@link #IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS}, then it returns {@code true}
     *   <li>if it returns {@link #IMPORTANT_FOR_AUTOFILL_NO} or
     *       {@link #IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS}, then it returns {@code false}
     *   <li>if it returns {@link #IMPORTANT_FOR_AUTOFILL_AUTO}, then it uses some simple heuristics
     *       that can return {@code true} in some cases (like a container with a resource id),
     *       but {@code false} in most.
     *   <li>otherwise, it returns {@code false}.
     * </ol>
     *
     * <p>When a view is considered important for autofill:
     * <ul>
     *   <li>The view might automatically trigger an autofill request when focused on.
     *   <li>The contents of the view are included in the {@link ViewStructure} used in an autofill
     *       request.
     * </ul>
     *
     * <p>On the other hand, when a view is considered not important for autofill:
     * <ul>
     *   <li>The view never automatically triggers autofill requests, but it can trigger a manual
     *       request through {@link AutofillManager#requestAutofill(View)}.
     *   <li>The contents of the view are not included in the {@link ViewStructure} used in an
     *       autofill request, unless the request has the
     *       {@link #AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS} flag.
     * </ul>
     *
     * @return whether the view is considered important for autofill.
```

```
     *
     * @see #setImportantForAutofill(int)
     * @see #IMPORTANT_FOR_AUTOFILL_AUTO
     * @see #IMPORTANT_FOR_AUTOFILL_YES
     * @see #IMPORTANT_FOR_AUTOFILL_NO
     * @see #IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS
     * @see #IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS
     * @see AutofillManager#requestAutofill(View)
     */
    public final boolean isImportantForAutofill() {
        // Check parent mode to ensure we're not hidden.
        ViewParent parent = mParent;
        while (parent instanceof View) {
            final int parentImportance = ((View) parent).getImportantForAutofill();
            if (parentImportance == IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS
                    || parentImportance == IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS) {
                return false;
            }
            parent = parent.getParent();
        }

        final int importance = getImportantForAutofill();

        // First, check the explicit states.
        if (importance == IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS
                || importance == IMPORTANT_FOR_AUTOFILL_YES) {
            return true;
        }
        if (importance == IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS
                || importance == IMPORTANT_FOR_AUTOFILL_NO) {
            return false;
        }

        // Then use some heuristics to handle AUTO.

        // Always include views that have an explicit resource id.
        final int id = mID;
        if (id != NO_ID && !isViewIdGenerated(id)) {
            final Resources res = getResources();
            String entry = null;
            String pkg = null;
            try {
                entry = res.getResourceEntryName(id);
                pkg = res.getResourcePackageName(id);
            } catch (Resources.NotFoundException e) {
                // ignore
            }
            if (entry != null && pkg != null && pkg.equals(mContext.getPackageName())) {
                return true;
            }
        }

        // Otherwise, assume it's not important...
        return false;
    }

    @Nullable
    private AutofillManager getAutofillManager() {
        return mContext.getSystemService(AutofillManager.class);
    }

    private boolean isAutofillable() {
        return getAutofillType() != AUTOFILL_TYPE_NONE && isImportantForAutofill()
                && getAutofillViewId() > LAST_APP_AUTOFILL_ID;
    }

    private void populateVirtualStructure(ViewStructure structure,
            AccessibilityNodeProvider provider, AccessibilityNodeInfo info) {
        structure.setId(AccessibilityNodeInfo.getVirtualDescendantId(info.getSourceNodeId()),
                null, null, null);
        Rect rect = structure.getTempRect();
        info.getBoundsInParent(rect);
        structure.setDimens(rect.left, rect.top, 0, 0, rect.width(), rect.height());
        structure.setVisibility(VISIBLE);
        structure.setEnabled(info.isEnabled());
        if (info.isClickable()) {
            structure.setClickable(true);
        }
        if (info.isFocusable()) {
            structure.setFocusable(true);
        }
        if (info.isFocused()) {
```

```
        structure.setFocused(true);
    }
    if (info.isAccessibilityFocused()) {
        structure.setAccessibilityFocused(true);
    }
    if (info.isSelected()) {
        structure.setSelected(true);
    }
    if (info.isLongClickable()) {
        structure.setLongClickable(true);
    }
    if (info.isCheckable()) {
        structure.setCheckable(true);
        if (info.isChecked()) {
            structure.setChecked(true);
        }
    }
    if (info.isContextClickable()) {
        structure.setContextClickable(true);
    }
    CharSequence cname = info.getClassName();
    structure.setClassName(cname != null ? cname.toString() : null);
    structure.setContentDescription(info.getContentDescription());
    if ((info.getText() != null || info.getError() != null)) {
        structure.setText(info.getText(), info.getTextSelectionStart(),
                info.getTextSelectionEnd());
    }
    final int NCHILDREN = info.getChildCount();
    if (NCHILDREN > 0) {
        structure.setChildCount(NCHILDREN);
        for (int i=0; i<NCHILDREN; i++) {
            AccessibilityNodeInfo cinfo = provider.createAccessibilityNodeInfo(
                    AccessibilityNodeInfo.getVirtualDescendantId(info.getChildId(i)));
            ViewStructure child = structure.newChild(i);
            populateVirtualStructure(child, provider, cinfo);
            cinfo.recycle();
        }
    }
}

/**
 * Dispatch creation of {@link ViewStructure} down the hierarchy.  The default
 * implementation calls {@link #onProvideStructure} and
 * {@link #onProvideVirtualStructure}.
 */
public void dispatchProvideStructure(ViewStructure structure) {
    dispatchProvideStructureForAssistOrAutofill(structure, false, 0);
}

/**
 * Dispatches creation of a {@link ViewStructure}s for autofill purposes down the hierarchy,
 * when an Assist structure is being created as part of an autofill request.
 *
 * <p>The default implementation does the following:
 * <ul>
 *   <li>Sets the {@link AutofillId} in the structure.
 *   <li>Calls {@link #onProvideAutofillStructure(ViewStructure, int)}.
 *   <li>Calls {@link #onProvideAutofillVirtualStructure(ViewStructure, int)}.
 * </ul>
 *
 * <p>Typically, this method should only be overridden by subclasses that provide a view
 * hierarchy (such as {@link ViewGroup}) - other classes should override
 * {@link #onProvideAutofillStructure(ViewStructure, int)} or
 * {@link #onProvideAutofillVirtualStructure(ViewStructure, int)} instead.
 *
 * <p>When overridden, it must:
 *
 * <ul>
 *   <li>Either call
 *       {@code super.dispatchProvideAutofillStructure(structure, flags)} or explicitly
 *       set the {@link AutofillId} in the structure (for example, by calling
 *       {@code structure.setAutofillId(getAutofillId())}).
 *   <li>Decide how to handle the {@link #AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS} flag - when
 *       set, all views in the structure should be considered important for autofill,
 *       regardless of what {@link #isImportantForAutofill()} returns. We encourage you to
 *       respect this flag to provide a better user experience - this flag is typically used
 *       when an user explicitly requested autofill. If the flag is not set,
 *       then only views marked as important for autofill should be included in the
 *       structure - skipping non-important views optimizes the overall autofill performance.
 * </ul>
 *
 * @param structure fill in with structured view data for autofill purposes.
```

```
     * @param flags optional flags.
     *
     * @see #AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS
     */
    public void dispatchProvideAutofillStructure(@NonNull ViewStructure structure,
            @AutofillFlags int flags) {
        dispatchProvideStructureForAssistOrAutofill(structure, true, flags);
    }

    private void dispatchProvideStructureForAssistOrAutofill(ViewStructure structure,
            boolean forAutofill, @AutofillFlags int flags) {
        if (forAutofill) {
            structure.setAutofillId(getAutofillId());
            onProvideAutofillStructure(structure, flags);
            onProvideAutofillVirtualStructure(structure, flags);
        } else if (!isAssistBlocked()) {
            onProvideStructure(structure);
            onProvideVirtualStructure(structure);
        } else {
            structure.setClassName(getAccessibilityClassName().toString());
            structure.setAssistBlocked(true);
        }
    }

    /**
     * @see #onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)
     *
     * Note: Called from the default {@link AccessibilityDelegate}.
     *
     * @hide
     */
    public void onInitializeAccessibilityNodeInfoInternal(AccessibilityNodeInfo info) {
        if (mAttachInfo == null) {
            return;
        }

        Rect bounds = mAttachInfo.mTmpInvalRect;

        getDrawingRect(bounds);
        info.setBoundsInParent(bounds);

        getBoundsOnScreen(bounds, true);
        info.setBoundsInScreen(bounds);

        ViewParent parent = getParentForAccessibility();
        if (parent instanceof View) {
            info.setParent((View) parent);
        }

        if (mID != View.NO_ID) {
            View rootView = getRootView();
            if (rootView == null) {
                rootView = this;
            }

            View label = rootView.findLabelForView(this, mID);
            if (label != null) {
                info.setLabeledBy(label);
            }

            if ((mAttachInfo.mAccessibilityFetchFlags
                    & AccessibilityNodeInfo.FLAG_REPORT_VIEW_IDS) != 0
                    && Resources.resourceHasPackage(mID)) {
                try {
                    String viewId = getResources().getResourceName(mID);
                    info.setViewIdResourceName(viewId);
                } catch (Resources.NotFoundException nfe) {
                    /* ignore */
                }
            }
        }

        if (mLabelForId != View.NO_ID) {
            View rootView = getRootView();
            if (rootView == null) {
                rootView = this;
            }
            View labeled = rootView.findViewInsideOutShouldExist(this, mLabelForId);
            if (labeled != null) {
                info.setLabelFor(labeled);
            }
        }
```

```
if (mAccessibilityTraversalBeforeId != View.NO_ID) {
    View rootView = getRootView();
    if (rootView == null) {
        rootView = this;
    }
    View next = rootView.findViewInsideOutShouldExist(this,
            mAccessibilityTraversalBeforeId);
    if (next != null && next.includeForAccessibility()) {
        info.setTraversalBefore(next);
    }
}

if (mAccessibilityTraversalAfterId != View.NO_ID) {
    View rootView = getRootView();
    if (rootView == null) {
        rootView = this;
    }
    View next = rootView.findViewInsideOutShouldExist(this,
            mAccessibilityTraversalAfterId);
    if (next != null && next.includeForAccessibility()) {
        info.setTraversalAfter(next);
    }
}

info.setVisibleToUser(isVisibleToUser());

info.setImportantForAccessibility(isImportantForAccessibility());
info.setPackageName(mContext.getPackageName());
info.setClassName(getAccessibilityClassName());
info.setContentDescription(getContentDescription());

info.setEnabled(isEnabled());
info.setClickable(isClickable());
info.setFocusable(isFocusable());
info.setFocused(isFocused());
info.setAccessibilityFocused(isAccessibilityFocused());
info.setSelected(isSelected());
info.setLongClickable(isLongClickable());
info.setContextClickable(isContextClickable());
info.setLiveRegion(getAccessibilityLiveRegion());

// TODO: These make sense only if we are in an AdapterView but all
// views can be selected. Maybe from accessibility perspective
// we should report as selectable view in an AdapterView.
info.addAction(AccessibilityNodeInfo.ACTION_SELECT);
info.addAction(AccessibilityNodeInfo.ACTION_CLEAR_SELECTION);

if (isFocusable()) {
    if (isFocused()) {
        info.addAction(AccessibilityNodeInfo.ACTION_CLEAR_FOCUS);
    } else {
        info.addAction(AccessibilityNodeInfo.ACTION_FOCUS);
    }
}

if (!isAccessibilityFocused()) {
    info.addAction(AccessibilityNodeInfo.ACTION_ACCESSIBILITY_FOCUS);
} else {
    info.addAction(AccessibilityNodeInfo.ACTION_CLEAR_ACCESSIBILITY_FOCUS);
}

if (isClickable() && isEnabled()) {
    info.addAction(AccessibilityNodeInfo.ACTION_CLICK);
}

if (isLongClickable() && isEnabled()) {
    info.addAction(AccessibilityNodeInfo.ACTION_LONG_CLICK);
}

if (isContextClickable() && isEnabled()) {
    info.addAction(AccessibilityAction.ACTION_CONTEXT_CLICK);
}

CharSequence text = getIterableTextForAccessibility();
if (text != null && text.length() > 0) {
    info.setTextSelection(getAccessibilitySelectionStart(), getAccessibilitySelectionEnd());

    info.addAction(AccessibilityNodeInfo.ACTION_SET_SELECTION);
    info.addAction(AccessibilityNodeInfo.ACTION_NEXT_AT_MOVEMENT_GRANULARITY);
    info.addAction(AccessibilityNodeInfo.ACTION_PREVIOUS_AT_MOVEMENT_GRANULARITY);
    info.setMovementGranularities(AccessibilityNodeInfo.MOVEMENT_GRANULARITY_CHARACTER
```

```
                | AccessibilityNodeInfo.MOVEMENT_GRANULARITY_WORD
                | AccessibilityNodeInfo.MOVEMENT_GRANULARITY_PARAGRAPH);
    }

    info.addAction(AccessibilityAction.ACTION_SHOW_ON_SCREEN);
    populateAccessibilityNodeInfoDrawingOrderInParent(info);
}

/**
 * Adds extra data to an {@link AccessibilityNodeInfo} based on an explicit request for the
 * additional data.
 * <p>
 * This method only needs overloading if the node is marked as having extra data available.
 * </p>
 *
 * @param info The info to which to add the extra data. Never {@code null}.
 * @param extraDataKey A key specifying the type of extra data to add to the info. The
 *                     extra data should be added to the {@link Bundle} returned by
 *                     the info's {@link AccessibilityNodeInfo#getExtras} method. Never
 *                     {@code null}.
 * @param arguments A {@link Bundle} holding any arguments relevant for this request. May be
 *                  {@code null} if the service provided no arguments.
 *
 * @see AccessibilityNodeInfo#setAvailableExtraData(List)
 */
public void addExtraDataToAccessibilityNodeInfo(
        @NonNull AccessibilityNodeInfo info, @NonNull String extraDataKey,
        @Nullable Bundle arguments) {
}

/**
 * Determine the order in which this view will be drawn relative to its siblings for a11y
 *
 * @param info The info whose drawing order should be populated
 */
private void populateAccessibilityNodeInfoDrawingOrderInParent(AccessibilityNodeInfo info) {
    /*
     * If the view's bounds haven't been set yet, layout has not completed. In that situation,
     * drawing order may not be well-defined, and some Views with custom drawing order may
     * not be initialized sufficiently to respond properly getChildDrawingOrder.
     */
    if ((mPrivateFlags & PFLAG_HAS_BOUNDS) == 0) {
        info.setDrawingOrder(0);
        return;
    }
    int drawingOrderInParent = 1;
    // Iterate up the hierarchy if parents are not important for a11y
    View viewAtDrawingLevel = this;
    final ViewParent parent = getParentForAccessibility();
    while (viewAtDrawingLevel != parent) {
        final ViewParent currentParent = viewAtDrawingLevel.getParent();
        if (!(currentParent instanceof ViewGroup)) {
            // Should only happen for the Decor
            drawingOrderInParent = 0;
            break;
        } else {
            final ViewGroup parentGroup = (ViewGroup) currentParent;
            final int childCount = parentGroup.getChildCount();
            if (childCount > 1) {
                List<View> preorderedList = parentGroup.buildOrderedChildList();
                if (preorderedList != null) {
                    final int childDrawIndex = preorderedList.indexOf(viewAtDrawingLevel);
                    for (int i = 0; i < childDrawIndex; i++) {
                        drawingOrderInParent += numViewsForAccessibility(preorderedList.get(i));
                    }
                } else {
                    final int childIndex = parentGroup.indexOfChild(viewAtDrawingLevel);
                    final boolean customOrder = parentGroup.isChildrenDrawingOrderEnabled();
                    final int childDrawIndex = ((childIndex >= 0) && customOrder) ? parentGroup
                            .getChildDrawingOrder(childCount, childIndex) : childIndex;
                    final int numChildrenToIterate = customOrder ? childCount : childDrawIndex;
                    if (childDrawIndex != 0) {
                        for (int i = 0; i < numChildrenToIterate; i++) {
                            final int otherDrawIndex = (customOrder ?
                                    parentGroup.getChildDrawingOrder(childCount, i) : i);
                            if (otherDrawIndex < childDrawIndex) {
                                drawingOrderInParent +=
                                        numViewsForAccessibility(parentGroup.getChildAt(i));
                            }
                        }
                    }
                }
            }
```

```
            }
        }
        viewAtDrawingLevel = (View) currentParent;
    }
    info.setDrawingOrder(drawingOrderInParent);
}

private static int numViewsForAccessibility(View view) {
    if (view != null) {
        if (view.includeForAccessibility()) {
            return 1;
        } else if (view instanceof ViewGroup) {
            return ((ViewGroup) view).getNumChildrenForAccessibility();
        }
    }
    return 0;
}

private View findLabelForView(View view, int labeledId) {
    if (mMatchLabelForPredicate == null) {
        mMatchLabelForPredicate = new MatchLabelForPredicate();
    }
    mMatchLabelForPredicate.mLabeledId = labeledId;
    return findViewByPredicateInsideOut(view, mMatchLabelForPredicate);
}

/**
 * Computes whether this view is visible to the user. Such a view is
 * attached, visible, all its predecessors are visible, it is not clipped
 * entirely by its predecessors, and has an alpha greater than zero.
 *
 * @return Whether the view is visible on the screen.
 *
 * @hide
 */
protected boolean isVisibleToUser() {
    return isVisibleToUser(null);
}

/**
 * Computes whether the given portion of this view is visible to the user.
 * Such a view is attached, visible, all its predecessors are visible,
 * has an alpha greater than zero, and the specified portion is not
 * clipped entirely by its predecessors.
 *
 * @param boundInView the portion of the view to test; coordinates should be relative; may be
 *                    <code>null</code>, and the entire view will be tested in this case.
 *                    When <code>true</code> is returned by the function, the actual visible
 *                    region will be stored in this parameter; that is, if boundInView is fully
 *                    contained within the view, no modification will be made, otherwise regions
 *                    outside of the visible area of the view will be clipped.
 *
 * @return Whether the specified portion of the view is visible on the screen.
 *
 * @hide
 */
protected boolean isVisibleToUser(Rect boundInView) {
    if (mAttachInfo != null) {
        // Attached to invisible window means this view is not visible.
        if (mAttachInfo.mWindowVisibility != View.VISIBLE) {
            return false;
        }
        // An invisible predecessor or one with alpha zero means
        // that this view is not visible to the user.
        Object current = this;
        while (current instanceof View) {
            View view = (View) current;
            // We have attach info so this view is attached and there is no
            // need to check whether we reach to ViewRootImpl on the way up.
            if (view.getAlpha() <= 0 || view.getTransitionAlpha() <= 0 ||
                    view.getVisibility() != VISIBLE) {
                return false;
            }
            current = view.mParent;
        }
        // Check if the view is entirely covered by its predecessors.
        Rect visibleRect = mAttachInfo.mTmpInvalRect;
        Point offset = mAttachInfo.mPoint;
        if (!getGlobalVisibleRect(visibleRect, offset)) {
            return false;
        }
        // Check if the visible portion intersects the rectangle of interest.
```

```
        if (boundInView != null) {
            visibleRect.offset(-offset.x, -offset.y);
            return boundInView.intersect(visibleRect);
        }
        return true;
    }
    return false;
}

/**
 * Returns the delegate for implementing accessibility support via
 * composition. For more details see {@link AccessibilityDelegate}.
 *
 * @return The delegate, or null if none set.
 *
 * @hide
 */
public AccessibilityDelegate getAccessibilityDelegate() {
    return mAccessibilityDelegate;
}

/**
 * Sets a delegate for implementing accessibility support via composition
 * (as opposed to inheritance). For more details, see
 * {@link AccessibilityDelegate}.
 * <p>
 * <strong>Note:</strong> On platform versions prior to
 * {@link android.os.Build.VERSION_CODES#M API 23}, delegate methods on
 * views in the {@code android.widget.*} package are called <i>before</i>
 * host methods. This prevents certain properties such as class name from
 * being modified by overriding
 * {@link AccessibilityDelegate#onInitializeAccessibilityNodeInfo(View, AccessibilityNodeInfo)},
 * as any changes will be overwritten by the host class.
 * <p>
 * Starting in {@link android.os.Build.VERSION_CODES#M API 23}, delegate
 * methods are called <i>after</i> host methods, which all properties to be
 * modified without being overwritten by the host class.
 *
 * @param delegate the object to which accessibility method calls should be
 *                 delegated
 * @see AccessibilityDelegate
 */
public void setAccessibilityDelegate(@Nullable AccessibilityDelegate delegate) {
    mAccessibilityDelegate = delegate;
}

/**
 * Gets the provider for managing a virtual view hierarchy rooted at this View
 * and reported to {@link android.accessibilityservice.AccessibilityService}s
 * that explore the window content.
 * <p>
 * If this method returns an instance, this instance is responsible for managing
 * {@link AccessibilityNodeInfo}s describing the virtual sub-tree rooted at this
 * View including the one representing the View itself. Similarly the returned
 * instance is responsible for performing accessibility actions on any virtual
 * view or the root view itself.
 * </p>
 * <p>
 * If an {@link AccessibilityDelegate} has been specified via calling
 * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
 * {@link AccessibilityDelegate#getAccessibilityNodeProvider(View)}
 * is responsible for handling this call.
 * </p>
 *
 * @return The provider.
 *
 * @see AccessibilityNodeProvider
 */
public AccessibilityNodeProvider getAccessibilityNodeProvider() {
    if (mAccessibilityDelegate != null) {
        return mAccessibilityDelegate.getAccessibilityNodeProvider(this);
    } else {
        return null;
    }
}

/**
 * Gets the unique identifier of this view on the screen for accessibility purposes.
 *
 * @return The view accessibility id.
 *
 * @hide
```

```
     */
    public int getAccessibilityViewId() {
        if (mAccessibilityViewId == NO_ID) {
            mAccessibilityViewId = sNextAccessibilityViewId++;
        }
        return mAccessibilityViewId;
    }

    /**
     * Gets the unique identifier of this view on the screen for autofill purposes.
     *
     * @return The view autofill id.
     *
     * @hide
     */
    public int getAutofillViewId() {
        if (mAutofillViewId == NO_ID) {
            mAutofillViewId = mContext.getNextAutofillId();
        }
        return mAutofillViewId;
    }

    /**
     * Gets the unique identifier of the window in which this View reseides.
     *
     * @return The window accessibility id.
     *
     * @hide
     */
    public int getAccessibilityWindowId() {
        return mAttachInfo != null ? mAttachInfo.mAccessibilityWindowId
                : AccessibilityWindowInfo.UNDEFINED_WINDOW_ID;
    }

    /**
     * Returns the {@link View}'s content description.
     * <p>
     * <strong>Note:</strong> Do not override this method, as it will have no
     * effect on the content description presented to accessibility services.
     * You must call {@link #setContentDescription(CharSequence)} to modify the
     * content description.
     *
     * @return the content description
     * @see #setContentDescription(CharSequence)
     * @attr ref android.R.styleable#View_contentDescription
     */
    @ViewDebug.ExportedProperty(category = "accessibility")
    public CharSequence getContentDescription() {
        return mContentDescription;
    }

    /**
     * Sets the {@link View}'s content description.
     * <p>
     * A content description briefly describes the view and is primarily used
     * for accessibility support to determine how a view should be presented to
     * the user. In the case of a view with no textual representation, such as
     * {@link android.widget.ImageButton}, a useful content description
     * explains what the view does. For example, an image button with a phone
     * icon that is used to place a call may use "Call" as its content
     * description. An image of a floppy disk that is used to save a file may
     * use "Save".
     *
     * @param contentDescription The content description.
     * @see #getContentDescription()
     * @attr ref android.R.styleable#View_contentDescription
     */
    @RemotableViewMethod
    public void setContentDescription(CharSequence contentDescription) {
        if (mContentDescription == null) {
            if (contentDescription == null) {
                return;
            }
        } else if (mContentDescription.equals(contentDescription)) {
            return;
        }
        mContentDescription = contentDescription;
        final boolean nonEmptyDesc = contentDescription != null && contentDescription.length() > 0;
        if (nonEmptyDesc && getImportantForAccessibility() == IMPORTANT_FOR_ACCESSIBILITY_AUTO) {
            setImportantForAccessibility(IMPORTANT_FOR_ACCESSIBILITY_YES);
            notifySubtreeAccessibilityStateChangedIfNeeded();
        } else {
```

```
        notifyViewAccessibilityStateChangedIfNeeded(
                AccessibilityEvent.CONTENT_CHANGE_TYPE_CONTENT_DESCRIPTION);
    }
}

/**
 * Sets the id of a view before which this one is visited in accessibility traversal.
 * A screen-reader must visit the content of this view before the content of the one
 * it precedes. For example, if view B is set to be before view A, then a screen-reader
 * will traverse the entire content of B before traversing the entire content of A,
 * regardles of what traversal strategy it is using.
 * <p>
 * Views that do not have specified before/after relationships are traversed in order
 * determined by the screen-reader.
 * </p>
 * <p>
 * Setting that this view is before a view that is not important for accessibility
 * or if this view is not important for accessibility will have no effect as the
 * screen-reader is not aware of unimportant views.
 * </p>
 *
 * @param beforeId The id of a view this one precedes in accessibility traversal.
 *
 * @attr ref android.R.styleable#View_accessibilityTraversalBefore
 *
 * @see #setImportantForAccessibility(int)
 */
@RemotableViewMethod
public void setAccessibilityTraversalBefore(int beforeId) {
    if (mAccessibilityTraversalBeforeId == beforeId) {
        return;
    }
    mAccessibilityTraversalBeforeId = beforeId;
    notifyViewAccessibilityStateChangedIfNeeded(
            AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
}

/**
 * Gets the id of a view before which this one is visited in accessibility traversal.
 *
 * @return The id of a view this one precedes in accessibility traversal if
 *         specified, otherwise {@link #NO_ID}.
 *
 * @see #setAccessibilityTraversalBefore(int)
 */
public int getAccessibilityTraversalBefore() {
    return mAccessibilityTraversalBeforeId;
}

/**
 * Sets the id of a view after which this one is visited in accessibility traversal.
 * A screen-reader must visit the content of the other view before the content of this
 * one. For example, if view B is set to be after view A, then a screen-reader
 * will traverse the entire content of A before traversing the entire content of B,
 * regardles of what traversal strategy it is using.
 * <p>
 * Views that do not have specified before/after relationships are traversed in order
 * determined by the screen-reader.
 * </p>
 * <p>
 * Setting that this view is after a view that is not important for accessibility
 * or if this view is not important for accessibility will have no effect as the
 * screen-reader is not aware of unimportant views.
 * </p>
 *
 * @param afterId The id of a view this one succedees in accessibility traversal.
 *
 * @attr ref android.R.styleable#View_accessibilityTraversalAfter
 *
 * @see #setImportantForAccessibility(int)
 */
@RemotableViewMethod
public void setAccessibilityTraversalAfter(int afterId) {
    if (mAccessibilityTraversalAfterId == afterId) {
        return;
    }
    mAccessibilityTraversalAfterId = afterId;
    notifyViewAccessibilityStateChangedIfNeeded(
            AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
}

/**
```

```
     * Gets the id of a view after which this one is visited in accessibility traversal.
     *
     * @return The id of a view this one succeedes in accessibility traversal if
     *         specified, otherwise {@link #NO_ID}.
     *
     * @see #setAccessibilityTraversalAfter(int)
     */
    public int getAccessibilityTraversalAfter() {
        return mAccessibilityTraversalAfterId;
    }

    /**
     * Gets the id of a view for which this view serves as a label for
     * accessibility purposes.
     *
     * @return The labeled view id.
     */
    @ViewDebug.ExportedProperty(category = "accessibility")
    public int getLabelFor() {
        return mLabelForId;
    }

    /**
     * Sets the id of a view for which this view serves as a label for
     * accessibility purposes.
     *
     * @param id The labeled view id.
     */
    @RemotableViewMethod
    public void setLabelFor(@IdRes int id) {
        if (mLabelForId == id) {
            return;
        }
        mLabelForId = id;
        if (mLabelForId != View.NO_ID
                && mID == View.NO_ID) {
            mID = generateViewId();
        }
        notifyViewAccessibilityStateChangedIfNeeded(
                AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
    }

    /**
     * Invoked whenever this view loses focus, either by losing window focus or by losing
     * focus within its window. This method can be used to clear any state tied to the
     * focus. For instance, if a button is held pressed with the trackball and the window
     * loses focus, this method can be used to cancel the press.
     *
     * Subclasses of View overriding this method should always call super.onFocusLost().
     *
     * @see #onFocusChanged(boolean, int, android.graphics.Rect)
     * @see #onWindowFocusChanged(boolean)
     *
     * @hide pending API council approval
     */
    @CallSuper
    protected void onFocusLost() {
        resetPressedState();
    }

    private void resetPressedState() {
        if ((mViewFlags & ENABLED_MASK) == DISABLED) {
            return;
        }

        if (isPressed()) {
            setPressed(false);

            if (!mHasPerformedLongPress) {
                removeLongPressCallback();
            }
        }
    }

    /**
     * Returns true if this view has focus
     *
     * @return True if this view has focus, false otherwise.
     */
    @ViewDebug.ExportedProperty(category = "focus")
    public boolean isFocused() {
        return (mPrivateFlags & PFLAG_FOCUSED) != 0;
```

```
}

/**
 * Find the view in the hierarchy rooted at this view that currently has
 * focus.
 *
 * @return The view that currently has focus, or null if no focused view can
 *         be found.
 */
public View findFocus() {
    return (mPrivateFlags & PFLAG_FOCUSED) != 0 ? this : null;
}

/**
 * Indicates whether this view is one of the set of scrollable containers in
 * its window.
 *
 * @return whether this view is one of the set of scrollable containers in
 * its window
 *
 * @attr ref android.R.styleable#View_isScrollContainer
 */
public boolean isScrollContainer() {
    return (mPrivateFlags & PFLAG_SCROLL_CONTAINER_ADDED) != 0;
}

/**
 * Change whether this view is one of the set of scrollable containers in
 * its window.  This will be used to determine whether the window can
 * resize or must pan when a soft input area is open -- scrollable
 * containers allow the window to use resize mode since the container
 * will appropriately shrink.
 *
 * @attr ref android.R.styleable#View_isScrollContainer
 */
public void setScrollContainer(boolean isScrollContainer) {
    if (isScrollContainer) {
        if (mAttachInfo != null && (mPrivateFlags&PFLAG_SCROLL_CONTAINER_ADDED) == 0) {
            mAttachInfo.mScrollContainers.add(this);
            mPrivateFlags |= PFLAG_SCROLL_CONTAINER_ADDED;
        }
        mPrivateFlags |= PFLAG_SCROLL_CONTAINER;
    } else {
        if ((mPrivateFlags&PFLAG_SCROLL_CONTAINER_ADDED) != 0) {
            mAttachInfo.mScrollContainers.remove(this);
        }
        mPrivateFlags &= ~(PFLAG_SCROLL_CONTAINER|PFLAG_SCROLL_CONTAINER_ADDED);
    }
}

/**
 * Returns the quality of the drawing cache.
 *
 * @return One of {@link #DRAWING_CACHE_QUALITY_AUTO},
 *         {@link #DRAWING_CACHE_QUALITY_LOW}, or {@link #DRAWING_CACHE_QUALITY_HIGH}
 *
 * @see #setDrawingCacheQuality(int)
 * @see #setDrawingCacheEnabled(boolean)
 * @see #isDrawingCacheEnabled()
 *
 * @attr ref android.R.styleable#View_drawingCacheQuality
 */
@DrawingCacheQuality
public int getDrawingCacheQuality() {
    return mViewFlags & DRAWING_CACHE_QUALITY_MASK;
}

/**
 * Set the drawing cache quality of this view. This value is used only when the
 * drawing cache is enabled
 *
 * @param quality One of {@link #DRAWING_CACHE_QUALITY_AUTO},
 *        {@link #DRAWING_CACHE_QUALITY_LOW}, or {@link #DRAWING_CACHE_QUALITY_HIGH}
 *
 * @see #getDrawingCacheQuality()
 * @see #setDrawingCacheEnabled(boolean)
 * @see #isDrawingCacheEnabled()
 *
 * @attr ref android.R.styleable#View_drawingCacheQuality
 */
public void setDrawingCacheQuality(@DrawingCacheQuality int quality) {
    setFlags(quality, DRAWING_CACHE_QUALITY_MASK);
```

```
}

/**
 * Returns whether the screen should remain on, corresponding to the current
 * value of {@link #KEEP_SCREEN_ON}.
 *
 * @return Returns true if {@link #KEEP_SCREEN_ON} is set.
 *
 * @see #setKeepScreenOn(boolean)
 *
 * @attr ref android.R.styleable#View_keepScreenOn
 */
public boolean getKeepScreenOn() {
    return (mViewFlags & KEEP_SCREEN_ON) != 0;
}

/**
 * Controls whether the screen should remain on, modifying the
 * value of {@link #KEEP_SCREEN_ON}.
 *
 * @param keepScreenOn Supply true to set {@link #KEEP_SCREEN_ON}.
 *
 * @see #getKeepScreenOn()
 *
 * @attr ref android.R.styleable#View_keepScreenOn
 */
public void setKeepScreenOn(boolean keepScreenOn) {
    setFlags(keepScreenOn ? KEEP_SCREEN_ON : 0, KEEP_SCREEN_ON);
}

/**
 * Gets the id of the view to use when the next focus is {@link #FOCUS_LEFT}.
 * @return The next focus ID, or {@link #NO_ID} if the framework should decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusLeft
 */
public int getNextFocusLeftId() {
    return mNextFocusLeftId;
}

/**
 * Sets the id of the view to use when the next focus is {@link #FOCUS_LEFT}.
 * @param nextFocusLeftId The next focus ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusLeft
 */
public void setNextFocusLeftId(int nextFocusLeftId) {
    mNextFocusLeftId = nextFocusLeftId;
}

/**
 * Gets the id of the view to use when the next focus is {@link #FOCUS_RIGHT}.
 * @return The next focus ID, or {@link #NO_ID} if the framework should decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusRight
 */
public int getNextFocusRightId() {
    return mNextFocusRightId;
}

/**
 * Sets the id of the view to use when the next focus is {@link #FOCUS_RIGHT}.
 * @param nextFocusRightId The next focus ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusRight
 */
public void setNextFocusRightId(int nextFocusRightId) {
    mNextFocusRightId = nextFocusRightId;
}

/**
 * Gets the id of the view to use when the next focus is {@link #FOCUS_UP}.
 * @return The next focus ID, or {@link #NO_ID} if the framework should decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusUp
 */
public int getNextFocusUpId() {
    return mNextFocusUpId;
}
```

```
/**
 * Sets the id of the view to use when the next focus is {@link #FOCUS_UP}.
 * @param nextFocusUpId The next focus ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusUp
 */
public void setNextFocusUpId(int nextFocusUpId) {
    mNextFocusUpId = nextFocusUpId;
}

/**
 * Gets the id of the view to use when the next focus is {@link #FOCUS_DOWN}.
 * @return The next focus ID, or {@link #NO_ID} if the framework should decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusDown
 */
public int getNextFocusDownId() {
    return mNextFocusDownId;
}

/**
 * Sets the id of the view to use when the next focus is {@link #FOCUS_DOWN}.
 * @param nextFocusDownId The next focus ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusDown
 */
public void setNextFocusDownId(int nextFocusDownId) {
    mNextFocusDownId = nextFocusDownId;
}

/**
 * Gets the id of the view to use when the next focus is {@link #FOCUS_FORWARD}.
 * @return The next focus ID, or {@link #NO_ID} if the framework should decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusForward
 */
public int getNextFocusForwardId() {
    return mNextFocusForwardId;
}

/**
 * Sets the id of the view to use when the next focus is {@link #FOCUS_FORWARD}.
 * @param nextFocusForwardId The next focus ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextFocusForward
 */
public void setNextFocusForwardId(int nextFocusForwardId) {
    mNextFocusForwardId = nextFocusForwardId;
}

/**
 * Gets the id of the root of the next keyboard navigation cluster.
 * @return The next keyboard navigation cluster ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextClusterForward
 */
public int getNextClusterForwardId() {
    return mNextClusterForwardId;
}

/**
 * Sets the id of the view to use as the root of the next keyboard navigation cluster.
 * @param nextClusterForwardId The next cluster ID, or {@link #NO_ID} if the framework should
 * decide automatically.
 *
 * @attr ref android.R.styleable#View_nextClusterForward
 */
public void setNextClusterForwardId(int nextClusterForwardId) {
    mNextClusterForwardId = nextClusterForwardId;
}

/**
 * Returns the visibility of this view and all of its ancestors
 *
 * @return True if this view and all of its ancestors are {@link #VISIBLE}
 */
public boolean isShown() {
    View current = this;
```

```
    //noinspection ConstantConditions
    do {
        if ((current.mViewFlags & VISIBILITY_MASK) != VISIBLE) {
            return false;
        }
        ViewParent parent = current.mParent;
        if (parent == null) {
            return false; // We are not attached to the view root
        }
        if (!(parent instanceof View)) {
            return true;
        }
        current = (View) parent;
    } while (current != null);

    return false;
}

/**
 * Called by the view hierarchy when the content insets for a window have
 * changed, to allow it to adjust its content to fit within those windows.
 * The content insets tell you the space that the status bar, input method,
 * and other system windows infringe on the application's window.
 *
 * <p>You do not normally need to deal with this function, since the default
 * window decoration given to applications takes care of applying it to the
 * content of the window.  If you use {@link #SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN}
 * or {@link #SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION} this will not be the case,
 * and your content can be placed under those system elements.  You can then
 * use this method within your view hierarchy if you have parts of your UI
 * which you would like to ensure are not being covered.
 *
 * <p>The default implementation of this method simply applies the content
 * insets to the view's padding, consuming that content (modifying the
 * insets to be 0), and returning true.  This behavior is off by default, but can
 * be enabled through {@link #setFitsSystemWindows(boolean)}.
 *
 * <p>This function's traversal down the hierarchy is depth-first.  The same content
 * insets object is propagated down the hierarchy, so any changes made to it will
 * be seen by all following views (including potentially ones above in
 * the hierarchy since this is a depth-first traversal).  The first view
 * that returns true will abort the entire traversal.
 *
 * <p>The default implementation works well for a situation where it is
 * used with a container that covers the entire window, allowing it to
 * apply the appropriate insets to its content on all edges.  If you need
 * a more complicated layout (such as two different views fitting system
 * windows, one on the top of the window, and one on the bottom),
 * you can override the method and handle the insets however you would like.
 * Note that the insets provided by the framework are always relative to the
 * far edges of the window, not accounting for the location of the called view
 * within that window.  (In fact when this method is called you do not yet know
 * where the layout will place the view, as it is done before layout happens.)
 *
 * <p>Note: unlike many View methods, there is no dispatch phase to this
 * call.  If you are overriding it in a ViewGroup and want to allow the
 * call to continue to your children, you must be sure to call the super
 * implementation.
 *
 * <p>Here is a sample layout that makes use of fitting system windows
 * to have controls for a video view placed inside of the window decorations
 * that it hides and shows.  This can be used with code like the second
 * sample (video player) shown in {@link #setSystemUiVisibility(int)}.
 *
 * {@sample development/samples/ApiDemos/res/layout/video_player.xml complete}
 *
 * @param insets Current content insets of the window.  Prior to
 * {@link android.os.Build.VERSION_CODES#JELLY_BEAN} you must not modify
 * the insets or else you and Android will be unhappy.
 *
 * @return {@code true} if this view applied the insets and it should not
 * continue propagating further down the hierarchy, {@code false} otherwise.
 * @see #getFitsSystemWindows()
 * @see #setFitsSystemWindows(boolean)
 * @see #setSystemUiVisibility(int)
 *
 * @deprecated As of API 20 use {@link #dispatchApplyWindowInsets(WindowInsets)} to apply
 * insets to views. Views should override {@link #onApplyWindowInsets(WindowInsets)} or use
 * {@link #setOnApplyWindowInsetsListener(android.view.View.OnApplyWindowInsetsListener)}
 * to implement handling their own insets.
 */
@Deprecated
```

```java
protected boolean fitSystemWindows(Rect insets) {
    if ((mPrivateFlags3 & PFLAG3_APPLYING_INSETS) == 0) {
        if (insets == null) {
            // Null insets by definition have already been consumed.
            // This call cannot apply insets since there are none to apply,
            // so return false.
            return false;
        }
        // If we're not in the process of dispatching the newer apply insets call,
        // that means we're not in the compatibility path. Dispatch into the newer
        // apply insets path and take things from there.
        try {
            mPrivateFlags3 |= PFLAG3_FITTING_SYSTEM_WINDOWS;
            return dispatchApplyWindowInsets(new WindowInsets(insets)).isConsumed();
        } finally {
            mPrivateFlags3 &= ~PFLAG3_FITTING_SYSTEM_WINDOWS;
        }
    } else {
        // We're being called from the newer apply insets path.
        // Perform the standard fallback behavior.
        return fitSystemWindowsInt(insets);
    }
}

private boolean fitSystemWindowsInt(Rect insets) {
    if ((mViewFlags & FITS_SYSTEM_WINDOWS) == FITS_SYSTEM_WINDOWS) {
        mUserPaddingStart = UNDEFINED_PADDING;
        mUserPaddingEnd = UNDEFINED_PADDING;
        Rect localInsets = sThreadLocal.get();
        if (localInsets == null) {
            localInsets = new Rect();
            sThreadLocal.set(localInsets);
        }
        boolean res = computeFitSystemWindows(insets, localInsets);
        mUserPaddingLeftInitial = localInsets.left;
        mUserPaddingRightInitial = localInsets.right;
        internalSetPadding(localInsets.left, localInsets.top,
                localInsets.right, localInsets.bottom);
        return res;
    }
    return false;
}

/**
 * Called when the view should apply {@link WindowInsets} according to its internal policy.
 *
 * <p>This method should be overridden by views that wish to apply a policy different from or
 * in addition to the default behavior. Clients that wish to force a view subtree
 * to apply insets should call {@link #dispatchApplyWindowInsets(WindowInsets)}.</p>
 *
 * <p>Clients may supply an {@link OnApplyWindowInsetsListener} to a view. If one is set
 * it will be called during dispatch instead of this method. The listener may optionally
 * call this method from its own implementation if it wishes to apply the view's default
 * insets policy in addition to its own.</p>
 *
 * <p>Implementations of this method should either return the insets parameter unchanged
 * or a new {@link WindowInsets} cloned from the supplied insets with any insets consumed
 * that this view applied itself. This allows new inset types added in future platform
 * versions to pass through existing implementations unchanged without being erroneously
 * consumed.</p>
 *
 * <p>By default if a view's {@link #setFitsSystemWindows(boolean) fitsSystemWindows}
 * property is set then the view will consume the system window insets and apply them
 * as padding for the view.</p>
 *
 * @param insets Insets to apply
 * @return The supplied insets with any applied insets consumed
 */
public WindowInsets onApplyWindowInsets(WindowInsets insets) {
    if ((mPrivateFlags3 & PFLAG3_FITTING_SYSTEM_WINDOWS) == 0) {
        // We weren't called from within a direct call to fitSystemWindows,
        // call into it as a fallback in case we're in a class that overrides it
        // and has logic to perform.
        if (fitSystemWindows(insets.getSystemWindowInsets())) {
            return insets.consumeSystemWindowInsets();
        }
    } else {
        // We were called from within a direct call to fitSystemWindows.
        if (fitSystemWindowsInt(insets.getSystemWindowInsets())) {
            return insets.consumeSystemWindowInsets();
        }
    }
```

```
    return insets;
}

/**
 * Set an {@link OnApplyWindowInsetsListener} to take over the policy for applying
 * window insets to this view. The listener's
 * {@link OnApplyWindowInsetsListener#onApplyWindowInsets(View, WindowInsets) onApplyWindowInsets}
 * method will be called instead of the view's
 * {@link #onApplyWindowInsets(WindowInsets) onApplyWindowInsets} method.
 *
 * @param listener Listener to set
 *
 * @see #onApplyWindowInsets(WindowInsets)
 */
public void setOnApplyWindowInsetsListener(OnApplyWindowInsetsListener listener) {
    getListenerInfo().mOnApplyWindowInsetsListener = listener;
}

/**
 * Request to apply the given window insets to this view or another view in its subtree.
 *
 * <p>This method should be called by clients wishing to apply insets corresponding to areas
 * obscured by window decorations or overlays. This can include the status and navigation bars,
 * action bars, input methods and more. New inset categories may be added in the future.
 * The method returns the insets provided minus any that were applied by this view or its
 * children.</p>
 *
 * <p>Clients wishing to provide custom behavior should override the
 * {@link #onApplyWindowInsets(WindowInsets)} method or alternatively provide a
 * {@link OnApplyWindowInsetsListener} via the
 * {@link #setOnApplyWindowInsetsListener(View.OnApplyWindowInsetsListener) setOnApplyWindowInsetsListener}
 * method.</p>
 *
 * <p>This method replaces the older {@link #fitSystemWindows(Rect) fitSystemWindows} method.
 * </p>
 *
 * @param insets Insets to apply
 * @return The provided insets minus the insets that were consumed
 */
public WindowInsets dispatchApplyWindowInsets(WindowInsets insets) {
    try {
        mPrivateFlags3 |= PFLAG3_APPLYING_INSETS;
        if (mListenerInfo != null && mListenerInfo.mOnApplyWindowInsetsListener != null) {
            return mListenerInfo.mOnApplyWindowInsetsListener.onApplyWindowInsets(this, insets);
        } else {
            return onApplyWindowInsets(insets);
        }
    } finally {
        mPrivateFlags3 &= ~PFLAG3_APPLYING_INSETS;
    }
}

/**
 * Compute the view's coordinate within the surface.
 *
 * <p>Computes the coordinates of this view in its surface. The argument
 * must be an array of two integers. After the method returns, the array
 * contains the x and y location in that order.</p>
 * @hide
 * @param location an array of two integers in which to hold the coordinates
 */
public void getLocationInSurface(@Size(2) int[] location) {
    getLocationInWindow(location);
    if (mAttachInfo != null && mAttachInfo.mViewRootImpl != null) {
        location[0] += mAttachInfo.mViewRootImpl.mWindowAttributes.surfaceInsets.left;
        location[1] += mAttachInfo.mViewRootImpl.mWindowAttributes.surfaceInsets.top;
    }
}

/**
 * Provide original WindowInsets that are dispatched to the view hierarchy. The insets are
 * only available if the view is attached.
 *
 * @return WindowInsets from the top of the view hierarchy or null if View is detached
 */
public WindowInsets getRootWindowInsets() {
    if (mAttachInfo != null) {
        return mAttachInfo.mViewRootImpl.getWindowInsets(false /* forceConstruct */);
    }
    return null;
}
```

```java
/**
 * @hide Compute the insets that should be consumed by this view and the ones
 * that should propagate to those under it.
 */
protected boolean computeFitSystemWindows(Rect inoutInsets, Rect outLocalInsets) {
    if ((mViewFlags & OPTIONAL_FITS_SYSTEM_WINDOWS) == 0
            || mAttachInfo == null
            || ((mAttachInfo.mSystemUiVisibility & SYSTEM_UI_LAYOUT_FLAGS) == 0
                    && !mAttachInfo.mOverscanRequested)) {
        outLocalInsets.set(inoutInsets);
        inoutInsets.set(0, 0, 0, 0);
        return true;
    } else {
        // The application wants to take care of fitting system window for
        // the content...  however we still need to take care of any overscan here.
        final Rect overscan = mAttachInfo.mOverscanInsets;
        outLocalInsets.set(overscan);
        inoutInsets.left -= overscan.left;
        inoutInsets.top -= overscan.top;
        inoutInsets.right -= overscan.right;
        inoutInsets.bottom -= overscan.bottom;
        return false;
    }
}

/**
 * Compute insets that should be consumed by this view and the ones that should propagate
 * to those under it.
 *
 * @param in Insets currently being processed by this View, likely received as a parameter
 *           to {@link #onApplyWindowInsets(WindowInsets)}.
 * @param outLocalInsets A Rect that will receive the insets that should be consumed
 *                       by this view
 * @return Insets that should be passed along to views under this one
 */
public WindowInsets computeSystemWindowInsets(WindowInsets in, Rect outLocalInsets) {
    if ((mViewFlags & OPTIONAL_FITS_SYSTEM_WINDOWS) == 0
            || mAttachInfo == null
            || (mAttachInfo.mSystemUiVisibility & SYSTEM_UI_LAYOUT_FLAGS) == 0) {
        outLocalInsets.set(in.getSystemWindowInsets());
        return in.consumeSystemWindowInsets();
    } else {
        outLocalInsets.set(0, 0, 0, 0);
        return in;
    }
}

/**
 * Sets whether or not this view should account for system screen decorations
 * such as the status bar and inset its content; that is, controlling whether
 * the default implementation of {@link #fitSystemWindows(Rect)} will be
 * executed.  See that method for more details.
 *
 * <p>Note that if you are providing your own implementation of
 * {@link #fitSystemWindows(Rect)}, then there is no need to set this
 * flag to true -- your implementation will be overriding the default
 * implementation that checks this flag.
 *
 * @param fitSystemWindows If true, then the default implementation of
 * {@link #fitSystemWindows(Rect)} will be executed.
 *
 * @attr ref android.R.styleable#View_fitsSystemWindows
 * @see #getFitsSystemWindows()
 * @see #fitSystemWindows(Rect)
 * @see #setSystemUiVisibility(int)
 */
public void setFitsSystemWindows(boolean fitSystemWindows) {
    setFlags(fitSystemWindows ? FITS_SYSTEM_WINDOWS : 0, FITS_SYSTEM_WINDOWS);
}

/**
 * Check for state of {@link #setFitsSystemWindows(boolean)}. If this method
 * returns {@code true}, the default implementation of {@link #fitSystemWindows(Rect)}
 * will be executed.
 *
 * @return {@code true} if the default implementation of
 * {@link #fitSystemWindows(Rect)} will be executed.
 *
 * @attr ref android.R.styleable#View_fitsSystemWindows
 * @see #setFitsSystemWindows(boolean)
 * @see #fitSystemWindows(Rect)
 * @see #setSystemUiVisibility(int)
```

```
     */
    @ViewDebug.ExportedProperty
    public boolean getFitsSystemWindows() {
        return (mViewFlags & FITS_SYSTEM_WINDOWS) == FITS_SYSTEM_WINDOWS;
    }

    /** @hide */
    public boolean fitsSystemWindows() {
        return getFitsSystemWindows();
    }

    /**
     * Ask that a new dispatch of {@link #fitSystemWindows(Rect)} be performed.
     * @deprecated Use {@link #requestApplyInsets()} for newer platform versions.
     */
    @Deprecated
    public void requestFitSystemWindows() {
        if (mParent != null) {
            mParent.requestFitSystemWindows();
        }
    }

    /**
     * Ask that a new dispatch of {@link #onApplyWindowInsets(WindowInsets)} be performed.
     */
    public void requestApplyInsets() {
        requestFitSystemWindows();
    }

    /**
     * For use by PhoneWindow to make its own system window fitting optional.
     * @hide
     */
    public void makeOptionalFitsSystemWindows() {
        setFlags(OPTIONAL_FITS_SYSTEM_WINDOWS, OPTIONAL_FITS_SYSTEM_WINDOWS);
    }

    /**
     * Returns the outsets, which areas of the device that aren't a surface, but we would like to
     * treat them as such.
     * @hide
     */
    public void getOutsets(Rect outOutsetRect) {
        if (mAttachInfo != null) {
            outOutsetRect.set(mAttachInfo.mOutsets);
        } else {
            outOutsetRect.setEmpty();
        }
    }

    /**
     * Returns the visibility status for this view.
     *
     * @return One of {@link #VISIBLE}, {@link #INVISIBLE}, or {@link #GONE}.
     * @attr ref android.R.styleable#View_visibility
     */
    @ViewDebug.ExportedProperty(mapping = {
        @ViewDebug.IntToString(from = VISIBLE,   to = "VISIBLE"),
        @ViewDebug.IntToString(from = INVISIBLE, to = "INVISIBLE"),
        @ViewDebug.IntToString(from = GONE,      to = "GONE")
    })
    @Visibility
    public int getVisibility() {
        return mViewFlags & VISIBILITY_MASK;
    }

    /**
     * Set the visibility state of this view.
     *
     * @param visibility One of {@link #VISIBLE}, {@link #INVISIBLE}, or {@link #GONE}.
     * @attr ref android.R.styleable#View_visibility
     */
    @RemotableViewMethod
    public void setVisibility(@Visibility int visibility) {
        setFlags(visibility, VISIBILITY_MASK);
    }

    /**
     * Returns the enabled status for this view. The interpretation of the
     * enabled state varies by subclass.
     *
     * @return True if this view is enabled, false otherwise.
```

```
     */
    @ViewDebug.ExportedProperty
    public boolean isEnabled() {
        return (mViewFlags & ENABLED_MASK) == ENABLED;
    }

    /**
     * Set the enabled state of this view. The interpretation of the enabled
     * state varies by subclass.
     *
     * @param enabled True if this view is enabled, false otherwise.
     */
    @RemotableViewMethod
    public void setEnabled(boolean enabled) {
        if (enabled == isEnabled()) return;

        setFlags(enabled ? ENABLED : DISABLED, ENABLED_MASK);

        /*
         * The View most likely has to change its appearance, so refresh
         * the drawable state.
         */
        refreshDrawableState();

        // Invalidate too, since the default behavior for views is to be
        // be drawn at 50% alpha rather than to change the drawable.
        invalidate(true);

        if (!enabled) {
            cancelPendingInputEvents();
        }
    }

    /**
     * Set whether this view can receive the focus.
     * <p>
     * Setting this to false will also ensure that this view is not focusable
     * in touch mode.
     *
     * @param focusable If true, this view can receive the focus.
     *
     * @see #setFocusableInTouchMode(boolean)
     * @see #setFocusable(int)
     * @attr ref android.R.styleable#View_focusable
     */
    public void setFocusable(boolean focusable) {
        setFocusable(focusable ? FOCUSABLE : NOT_FOCUSABLE);
    }

    /**
     * Sets whether this view can receive focus.
     * <p>
     * Setting this to {@link #FOCUSABLE_AUTO} tells the framework to determine focusability
     * automatically based on the view's interactivity. This is the default.
     * <p>
     * Setting this to NOT_FOCUSABLE will ensure that this view is also not focusable
     * in touch mode.
     *
     * @param focusable One of {@link #NOT_FOCUSABLE}, {@link #FOCUSABLE},
     *                  or {@link #FOCUSABLE_AUTO}.
     * @see #setFocusableInTouchMode(boolean)
     * @attr ref android.R.styleable#View_focusable
     */
    public void setFocusable(@Focusable int focusable) {
        if ((focusable & (FOCUSABLE_AUTO | FOCUSABLE)) == 0) {
            setFlags(0, FOCUSABLE_IN_TOUCH_MODE);
        }
        setFlags(focusable, FOCUSABLE_MASK);
    }

    /**
     * Set whether this view can receive focus while in touch mode.
     *
     * Setting this to true will also ensure that this view is focusable.
     *
     * @param focusableInTouchMode If true, this view can receive the focus while
     *   in touch mode.
     *
     * @see #setFocusable(boolean)
     * @attr ref android.R.styleable#View_focusableInTouchMode
     */
    public void setFocusableInTouchMode(boolean focusableInTouchMode) {
```

```
    // Focusable in touch mode should always be set before the focusable flag
    // otherwise, setting the focusable flag will trigger a focusableViewAvailable()
    // which, in touch mode, will not successfully request focus on this view
    // because the focusable in touch mode flag is not set
    setFlags(focusableInTouchMode ? FOCUSABLE_IN_TOUCH_MODE : 0, FOCUSABLE_IN_TOUCH_MODE);

    // Clear FOCUSABLE_AUTO if set.
    if (focusableInTouchMode) {
        // Clears FOCUSABLE_AUTO if set.
        setFlags(FOCUSABLE, FOCUSABLE_MASK);
    }
}

/**
 * Sets the hints that help an {@link android.service.autofill.AutofillService} determine how
 * to autofill the view with the user's data.
 *
 * <p>Typically, there is only one way to autofill a view, but there could be more than one.
 * For example, if the application accepts either an username or email address to identify
 * an user.
 *
 * <p>These hints are not validated by the Android System, but passed "as is" to the service.
 * Hence, they can have any value, but it's recommended to use the {@code AUTOFILL_HINT_}
 * constants such as:
 * {@link #AUTOFILL_HINT_USERNAME}, {@link #AUTOFILL_HINT_PASSWORD},
 * {@link #AUTOFILL_HINT_EMAIL_ADDRESS},
 * {@link #AUTOFILL_HINT_NAME},
 * {@link #AUTOFILL_HINT_PHONE},
 * {@link #AUTOFILL_HINT_POSTAL_ADDRESS}, {@link #AUTOFILL_HINT_POSTAL_CODE},
 * {@link #AUTOFILL_HINT_CREDIT_CARD_NUMBER}, {@link #AUTOFILL_HINT_CREDIT_CARD_SECURITY_CODE},
 * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DATE},
 * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DAY},
 * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_MONTH} or
 * {@link #AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_YEAR}.
 *
 * @param autofillHints The autofill hints to set. If the array is emtpy, {@code null} is set.
 * @attr ref android.R.styleable#View_autofillHints
 */
public void setAutofillHints(@Nullable String... autofillHints) {
    if (autofillHints == null || autofillHints.length == 0) {
        mAutofillHints = null;
    } else {
        mAutofillHints = autofillHints;
    }
}

/**
 * @hide
 */
@TestApi
public void setAutofilled(boolean isAutofilled) {
    boolean wasChanged = isAutofilled != isAutofilled();

    if (wasChanged) {
        if (isAutofilled) {
            mPrivateFlags3 |= PFLAG3_IS_AUTOFILLED;
        } else {
            mPrivateFlags3 &= ~PFLAG3_IS_AUTOFILLED;
        }

        invalidate();
    }
}

/**
 * Set whether this view should have sound effects enabled for events such as
 * clicking and touching.
 *
 * <p>You may wish to disable sound effects for a view if you already play sounds,
 * for instance, a dial key that plays dtmf tones.
 *
 * @param soundEffectsEnabled whether sound effects are enabled for this view.
 * @see #isSoundEffectsEnabled()
 * @see #playSoundEffect(int)
 * @attr ref android.R.styleable#View_soundEffectsEnabled
 */
public void setSoundEffectsEnabled(boolean soundEffectsEnabled) {
    setFlags(soundEffectsEnabled ? SOUND_EFFECTS_ENABLED: 0, SOUND_EFFECTS_ENABLED);
}

/**
 * @return whether this view should have sound effects enabled for events such as
```

```
     *     clicking and touching.
     *
     * @see #setSoundEffectsEnabled(boolean)
     * @see #playSoundEffect(int)
     * @attr ref android.R.styleable#View_soundEffectsEnabled
     */
    @ViewDebug.ExportedProperty
    public boolean isSoundEffectsEnabled() {
        return SOUND_EFFECTS_ENABLED == (mViewFlags & SOUND_EFFECTS_ENABLED);
    }

    /**
     * Set whether this view should have haptic feedback for events such as
     * long presses.
     *
     * <p>You may wish to disable haptic feedback if your view already controls
     * its own haptic feedback.
     *
     * @param hapticFeedbackEnabled whether haptic feedback enabled for this view.
     * @see #isHapticFeedbackEnabled()
     * @see #performHapticFeedback(int)
     * @attr ref android.R.styleable#View_hapticFeedbackEnabled
     */
    public void setHapticFeedbackEnabled(boolean hapticFeedbackEnabled) {
        setFlags(hapticFeedbackEnabled ? HAPTIC_FEEDBACK_ENABLED: 0, HAPTIC_FEEDBACK_ENABLED);
    }

    /**
     * @return whether this view should have haptic feedback enabled for events
     * long presses.
     *
     * @see #setHapticFeedbackEnabled(boolean)
     * @see #performHapticFeedback(int)
     * @attr ref android.R.styleable#View_hapticFeedbackEnabled
     */
    @ViewDebug.ExportedProperty
    public boolean isHapticFeedbackEnabled() {
        return HAPTIC_FEEDBACK_ENABLED == (mViewFlags & HAPTIC_FEEDBACK_ENABLED);
    }

    /**
     * Returns the layout direction for this view.
     *
     * @return One of {@link #LAYOUT_DIRECTION_LTR},
     *   {@link #LAYOUT_DIRECTION_RTL},
     *   {@link #LAYOUT_DIRECTION_INHERIT} or
     *   {@link #LAYOUT_DIRECTION_LOCALE}.
     *
     * @attr ref android.R.styleable#View_layoutDirection
     *
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "layout", mapping = {
        @ViewDebug.IntToString(from = LAYOUT_DIRECTION_LTR,     to = "LTR"),
        @ViewDebug.IntToString(from = LAYOUT_DIRECTION_RTL,     to = "RTL"),
        @ViewDebug.IntToString(from = LAYOUT_DIRECTION_INHERIT, to = "INHERIT"),
        @ViewDebug.IntToString(from = LAYOUT_DIRECTION_LOCALE,  to = "LOCALE")
    })
    @LayoutDir
    public int getRawLayoutDirection() {
        return (mPrivateFlags2 & PFLAG2_LAYOUT_DIRECTION_MASK) >> PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT;
    }

    /**
     * Set the layout direction for this view. This will propagate a reset of layout direction
     * resolution to the view's children and resolve layout direction for this view.
     *
     * @param layoutDirection the layout direction to set. Should be one of:
     *
     * {@link #LAYOUT_DIRECTION_LTR},
     * {@link #LAYOUT_DIRECTION_RTL},
     * {@link #LAYOUT_DIRECTION_INHERIT},
     * {@link #LAYOUT_DIRECTION_LOCALE}.
     *
     * Resolution will be done if the value is set to LAYOUT_DIRECTION_INHERIT. The resolution
     * proceeds up the parent chain of the view to get the value. If there is no parent, then it
     * will return the default {@link #LAYOUT_DIRECTION_LTR}.
     *
     * @attr ref android.R.styleable#View_layoutDirection
     */
    @RemotableViewMethod
    public void setLayoutDirection(@LayoutDir int layoutDirection) {
```

```
    if (getRawLayoutDirection() != layoutDirection) {
        // Reset the current layout direction and the resolved one
        mPrivateFlags2 &= ~PFLAG2_LAYOUT_DIRECTION_MASK;
        resetRtlProperties();
        // Set the new layout direction (filtered)
        mPrivateFlags2 |=
                ((layoutDirection << PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT) & PFLAG2_LAYOUT_DIRECTION_MASK);
        // We need to resolve all RTL properties as they all depend on layout direction
        resolveRtlPropertiesIfNeeded();
        requestLayout();
        invalidate(true);
    }
}

/**
 * Returns the resolved layout direction for this view.
 *
 * @return {@link #LAYOUT_DIRECTION_RTL} if the layout direction is RTL or returns
 * {@link #LAYOUT_DIRECTION_LTR} if the layout direction is not RTL.
 *
 * For compatibility, this will return {@link #LAYOUT_DIRECTION_LTR} if API version
 * is lower than {@link android.os.Build.VERSION_CODES#JELLY_BEAN_MR1}.
 *
 * @attr ref android.R.styleable#View_layoutDirection
 */
@ViewDebug.ExportedProperty(category = "layout", mapping = {
    @ViewDebug.IntToString(from = LAYOUT_DIRECTION_LTR, to = "RESOLVED_DIRECTION_LTR"),
    @ViewDebug.IntToString(from = LAYOUT_DIRECTION_RTL, to = "RESOLVED_DIRECTION_RTL")
})
@ResolvedLayoutDir
public int getLayoutDirection() {
    final int targetSdkVersion = getContext().getApplicationInfo().targetSdkVersion;
    if (targetSdkVersion < Build.VERSION_CODES.JELLY_BEAN_MR1) {
        mPrivateFlags2 |= PFLAG2_LAYOUT_DIRECTION_RESOLVED;
        return LAYOUT_DIRECTION_RESOLVED_DEFAULT;
    }
    return ((mPrivateFlags2 & PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL) ==
            PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL) ? LAYOUT_DIRECTION_RTL : LAYOUT_DIRECTION_LTR;
}

/**
 * Indicates whether or not this view's layout is right-to-left. This is resolved from
 * layout attribute and/or the inherited value from the parent
 *
 * @return true if the layout is right-to-left.
 *
 * @hide
 */
@ViewDebug.ExportedProperty(category = "layout")
public boolean isLayoutRtl() {
    return (getLayoutDirection() == LAYOUT_DIRECTION_RTL);
}

/**
 * Indicates whether the view is currently tracking transient state that the
 * app should not need to concern itself with saving and restoring, but that
 * the framework should take special note to preserve when possible.
 *
 * <p>A view with transient state cannot be trivially rebound from an external
 * data source, such as an adapter binding item views in a list. This may be
 * because the view is performing an animation, tracking user selection
 * of content, or similar.</p>
 *
 * @return true if the view has transient state
 */
@ViewDebug.ExportedProperty(category = "layout")
public boolean hasTransientState() {
    return (mPrivateFlags2 & PFLAG2_HAS_TRANSIENT_STATE) == PFLAG2_HAS_TRANSIENT_STATE;
}

/**
 * Set whether this view is currently tracking transient state that the
 * framework should attempt to preserve when possible. This flag is reference counted,
 * so every call to setHasTransientState(true) should be paired with a later call
 * to setHasTransientState(false).
 *
 * <p>A view with transient state cannot be trivially rebound from an external
 * data source, such as an adapter binding item views in a list. This may be
 * because the view is performing an animation, tracking user selection
 * of content, or similar.</p>
 *
 * @param hasTransientState true if this view has transient state
```

```
     */
    public void setHasTransientState(boolean hasTransientState) {
        final boolean oldHasTransientState = hasTransientState();
        mTransientStateCount = hasTransientState ? mTransientStateCount + 1 :
                mTransientStateCount - 1;
        if (mTransientStateCount < 0) {
            mTransientStateCount = 0;
            Log.e(VIEW_LOG_TAG, "hasTransientState decremented below 0: " +
                    "unmatched pair of setHasTransientState calls");
        } else if ((hasTransientState && mTransientStateCount == 1) ||
                (!hasTransientState && mTransientStateCount == 0)) {
            // update flag if we've just incremented up from 0 or decremented down to 0
            mPrivateFlags2 = (mPrivateFlags2 & ~PFLAG2_HAS_TRANSIENT_STATE) |
                    (hasTransientState ? PFLAG2_HAS_TRANSIENT_STATE : 0);
            final boolean newHasTransientState = hasTransientState();
            if (mParent != null && newHasTransientState != oldHasTransientState) {
                try {
                    mParent.childHasTransientStateChanged(this, newHasTransientState);
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
            }
        }
    }

    /**
     * Returns true if this view is currently attached to a window.
     */
    public boolean isAttachedToWindow() {
        return mAttachInfo != null;
    }

    /**
     * Returns true if this view has been through at least one layout since it
     * was last attached to or detached from a window.
     */
    public boolean isLaidOut() {
        return (mPrivateFlags3 & PFLAG3_IS_LAID_OUT) == PFLAG3_IS_LAID_OUT;
    }

    /**
     * If this view doesn't do any drawing on its own, set this flag to
     * allow further optimizations. By default, this flag is not set on
     * View, but could be set on some View subclasses such as ViewGroup.
     *
     * Typically, if you override {@link #onDraw(android.graphics.Canvas)}
     * you should clear this flag.
     *
     * @param willNotDraw whether or not this View draw on its own
     */
    public void setWillNotDraw(boolean willNotDraw) {
        setFlags(willNotDraw ? WILL_NOT_DRAW : 0, DRAW_MASK);
    }

    /**
     * Returns whether or not this View draws on its own.
     *
     * @return true if this view has nothing to draw, false otherwise
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public boolean willNotDraw() {
        return (mViewFlags & DRAW_MASK) == WILL_NOT_DRAW;
    }

    /**
     * When a View's drawing cache is enabled, drawing is redirected to an
     * offscreen bitmap. Some views, like an ImageView, must be able to
     * bypass this mechanism if they already draw a single bitmap, to avoid
     * unnecessary usage of the memory.
     *
     * @param willNotCacheDrawing true if this view does not cache its
     *        drawing, false otherwise
     */
    public void setWillNotCacheDrawing(boolean willNotCacheDrawing) {
        setFlags(willNotCacheDrawing ? WILL_NOT_CACHE_DRAWING : 0, WILL_NOT_CACHE_DRAWING);
    }

    /**
     * Returns whether or not this View can cache its drawing or not.
     *
     * @return true if this view does not cache its drawing, false otherwise
```

```
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public boolean willNotCacheDrawing() {
        return (mViewFlags & WILL_NOT_CACHE_DRAWING) == WILL_NOT_CACHE_DRAWING;
    }

    /**
     * Indicates whether this view reacts to click events or not.
     *
     * @return true if the view is clickable, false otherwise
     *
     * @see #setClickable(boolean)
     * @attr ref android.R.styleable#View_clickable
     */
    @ViewDebug.ExportedProperty
    public boolean isClickable() {
        return (mViewFlags & CLICKABLE) == CLICKABLE;
    }

    /**
     * Enables or disables click events for this view. When a view
     * is clickable it will change its state to "pressed" on every click.
     * Subclasses should set the view clickable to visually react to
     * user's clicks.
     *
     * @param clickable true to make the view clickable, false otherwise
     *
     * @see #isClickable()
     * @attr ref android.R.styleable#View_clickable
     */
    public void setClickable(boolean clickable) {
        setFlags(clickable ? CLICKABLE : 0, CLICKABLE);
    }

    /**
     * Indicates whether this view reacts to long click events or not.
     *
     * @return true if the view is long clickable, false otherwise
     *
     * @see #setLongClickable(boolean)
     * @attr ref android.R.styleable#View_longClickable
     */
    public boolean isLongClickable() {
        return (mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE;
    }

    /**
     * Enables or disables long click events for this view. When a view is long
     * clickable it reacts to the user holding down the button for a longer
     * duration than a tap. This event can either launch the listener or a
     * context menu.
     *
     * @param longClickable true to make the view long clickable, false otherwise
     * @see #isLongClickable()
     * @attr ref android.R.styleable#View_longClickable
     */
    public void setLongClickable(boolean longClickable) {
        setFlags(longClickable ? LONG_CLICKABLE : 0, LONG_CLICKABLE);
    }

    /**
     * Indicates whether this view reacts to context clicks or not.
     *
     * @return true if the view is context clickable, false otherwise
     * @see #setContextClickable(boolean)
     * @attr ref android.R.styleable#View_contextClickable
     */
    public boolean isContextClickable() {
        return (mViewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;
    }

    /**
     * Enables or disables context clicking for this view. This event can launch the listener.
     *
     * @param contextClickable true to make the view react to a context click, false otherwise
     * @see #isContextClickable()
     * @attr ref android.R.styleable#View_contextClickable
     */
    public void setContextClickable(boolean contextClickable) {
        setFlags(contextClickable ? CONTEXT_CLICKABLE : 0, CONTEXT_CLICKABLE);
    }
```

```java
/**
 * Sets the pressed state for this view and provides a touch coordinate for
 * animation hinting.
 *
 * @param pressed Pass true to set the View's internal state to "pressed",
 *            or false to reverts the View's internal state from a
 *            previously set "pressed" state.
 * @param x The x coordinate of the touch that caused the press
 * @param y The y coordinate of the touch that caused the press
 */
private void setPressed(boolean pressed, float x, float y) {
    if (pressed) {
        drawableHotspotChanged(x, y);
    }

    setPressed(pressed);
}

/**
 * Sets the pressed state for this view.
 *
 * @see #isClickable()
 * @see #setClickable(boolean)
 *
 * @param pressed Pass true to set the View's internal state to "pressed", or false to reverts
 *        the View's internal state from a previously set "pressed" state.
 */
public void setPressed(boolean pressed) {
    final boolean needsRefresh = pressed != ((mPrivateFlags & PFLAG_PRESSED) == PFLAG_PRESSED);

    if (pressed) {
        mPrivateFlags |= PFLAG_PRESSED;
    } else {
        mPrivateFlags &= ~PFLAG_PRESSED;
    }

    if (needsRefresh) {
        refreshDrawableState();
    }
    dispatchSetPressed(pressed);
}

/**
 * Dispatch setPressed to all of this View's children.
 *
 * @see #setPressed(boolean)
 *
 * @param pressed The new pressed state
 */
protected void dispatchSetPressed(boolean pressed) {
}

/**
 * Indicates whether the view is currently in pressed state. Unless
 * {@link #setPressed(boolean)} is explicitly called, only clickable views can enter
 * the pressed state.
 *
 * @see #setPressed(boolean)
 * @see #isClickable()
 * @see #setClickable(boolean)
 *
 * @return true if the view is currently pressed, false otherwise
 */
@ViewDebug.ExportedProperty
public boolean isPressed() {
    return (mPrivateFlags & PFLAG_PRESSED) == PFLAG_PRESSED;
}

/**
 * @hide
 * Indicates whether this view will participate in data collection through
 * {@link ViewStructure}.  If true, it will not provide any data
 * for itself or its children.  If false, the normal data collection will be allowed.
 *
 * @return Returns false if assist data collection is not blocked, else true.
 *
 * @see #setAssistBlocked(boolean)
 * @attr ref android.R.styleable#View_assistBlocked
 */
public boolean isAssistBlocked() {
    return (mPrivateFlags3 & PFLAG3_ASSIST_BLOCKED) != 0;
}
```

```java
/**
 * @hide
 * Controls whether assist data collection from this view and its children is enabled
 * (that is, whether {@link #onProvideStructure} and
 * {@link #onProvideVirtualStructure} will be called).  The default value is false,
 * allowing normal assist collection.  Setting this to false will disable assist collection.
 *
 * @param enabled Set to true to <em>disable</em> assist data collection, or false
 * (the default) to allow it.
 *
 * @see #isAssistBlocked()
 * @see #onProvideStructure
 * @see #onProvideVirtualStructure
 * @attr ref android.R.styleable#View_assistBlocked
 */
public void setAssistBlocked(boolean enabled) {
    if (enabled) {
        mPrivateFlags3 |= PFLAG3_ASSIST_BLOCKED;
    } else {
        mPrivateFlags3 &= ~PFLAG3_ASSIST_BLOCKED;
    }
}

/**
 * Indicates whether this view will save its state (that is,
 * whether its {@link #onSaveInstanceState} method will be called).
 *
 * @return Returns true if the view state saving is enabled, else false.
 *
 * @see #setSaveEnabled(boolean)
 * @attr ref android.R.styleable#View_saveEnabled
 */
public boolean isSaveEnabled() {
    return (mViewFlags & SAVE_DISABLED_MASK) != SAVE_DISABLED;
}

/**
 * Controls whether the saving of this view's state is
 * enabled (that is, whether its {@link #onSaveInstanceState} method
 * will be called).  Note that even if freezing is enabled, the
 * view still must have an id assigned to it (via {@link #setId(int)})
 * for its state to be saved.  This flag can only disable the
 * saving of this view; any child views may still have their state saved.
 *
 * @param enabled Set to false to <em>disable</em> state saving, or true
 * (the default) to allow it.
 *
 * @see #isSaveEnabled()
 * @see #setId(int)
 * @see #onSaveInstanceState()
 * @attr ref android.R.styleable#View_saveEnabled
 */
public void setSaveEnabled(boolean enabled) {
    setFlags(enabled ? 0 : SAVE_DISABLED, SAVE_DISABLED_MASK);
}

/**
 * Gets whether the framework should discard touches when the view's
 * window is obscured by another visible window.
 * Refer to the {@link View} security documentation for more details.
 *
 * @return True if touch filtering is enabled.
 *
 * @see #setFilterTouchesWhenObscured(boolean)
 * @attr ref android.R.styleable#View_filterTouchesWhenObscured
 */
@ViewDebug.ExportedProperty
public boolean getFilterTouchesWhenObscured() {
    return (mViewFlags & FILTER_TOUCHES_WHEN_OBSCURED) != 0;
}

/**
 * Sets whether the framework should discard touches when the view's
 * window is obscured by another visible window.
 * Refer to the {@link View} security documentation for more details.
 *
 * @param enabled True if touch filtering should be enabled.
 *
 * @see #getFilterTouchesWhenObscured
 * @attr ref android.R.styleable#View_filterTouchesWhenObscured
 */
```

```
public void setFilterTouchesWhenObscured(boolean enabled) {
    setFlags(enabled ? FILTER_TOUCHES_WHEN_OBSCURED : 0,
            FILTER_TOUCHES_WHEN_OBSCURED);
}

/**
 * Indicates whether the entire hierarchy under this view will save its
 * state when a state saving traversal occurs from its parent.  The default
 * is true; if false, these views will not be saved unless
 * {@link #saveHierarchyState(SparseArray)} is called directly on this view.
 *
 * @return Returns true if the view state saving from parent is enabled, else false.
 *
 * @see #setSaveFromParentEnabled(boolean)
 */
public boolean isSaveFromParentEnabled() {
    return (mViewFlags & PARENT_SAVE_DISABLED_MASK) != PARENT_SAVE_DISABLED;
}

/**
 * Controls whether the entire hierarchy under this view will save its
 * state when a state saving traversal occurs from its parent.  The default
 * is true; if false, these views will not be saved unless
 * {@link #saveHierarchyState(SparseArray)} is called directly on this view.
 *
 * @param enabled Set to false to <em>disable</em> state saving, or true
 * (the default) to allow it.
 *
 * @see #isSaveFromParentEnabled()
 * @see #setId(int)
 * @see #onSaveInstanceState()
 */
public void setSaveFromParentEnabled(boolean enabled) {
    setFlags(enabled ? 0 : PARENT_SAVE_DISABLED, PARENT_SAVE_DISABLED_MASK);
}


/**
 * Returns whether this View is currently able to take focus.
 *
 * @return True if this view can take focus, or false otherwise.
 */
@ViewDebug.ExportedProperty(category = "focus")
public final boolean isFocusable() {
    return FOCUSABLE == (mViewFlags & FOCUSABLE);
}

/**
 * Returns the focusable setting for this view.
 *
 * @return One of {@link #NOT_FOCUSABLE}, {@link #FOCUSABLE}, or {@link #FOCUSABLE_AUTO}.
 * @attr ref android.R.styleable#View_focusable
 */
@ViewDebug.ExportedProperty(mapping = {
        @ViewDebug.IntToString(from = NOT_FOCUSABLE, to = "NOT_FOCUSABLE"),
        @ViewDebug.IntToString(from = FOCUSABLE, to = "FOCUSABLE"),
        @ViewDebug.IntToString(from = FOCUSABLE_AUTO, to = "FOCUSABLE_AUTO")
        }, category = "focus")
@Focusable
public int getFocusable() {
    return (mViewFlags & FOCUSABLE_AUTO) > 0 ? FOCUSABLE_AUTO : mViewFlags & FOCUSABLE;
}

/**
 * When a view is focusable, it may not want to take focus when in touch mode.
 * For example, a button would like focus when the user is navigating via a D-pad
 * so that the user can click on it, but once the user starts touching the screen,
 * the button shouldn't take focus
 * @return Whether the view is focusable in touch mode.
 * @attr ref android.R.styleable#View_focusableInTouchMode
 */
@ViewDebug.ExportedProperty(category = "focus")
public final boolean isFocusableInTouchMode() {
    return FOCUSABLE_IN_TOUCH_MODE == (mViewFlags & FOCUSABLE_IN_TOUCH_MODE);
}

/**
 * Find the nearest view in the specified direction that can take focus.
 * This does not actually give focus to that view.
 *
 * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT
 *
```

```
     * @return The nearest focusable in the specified direction, or null if none
     *         can be found.
     */
    public View focusSearch(@FocusRealDirection int direction) {
        if (mParent != null) {
            return mParent.focusSearch(this, direction);
        } else {
            return null;
        }
    }

    /**
     * Returns whether this View is a root of a keyboard navigation cluster.
     *
     * @return True if this view is a root of a cluster, or false otherwise.
     * @attr ref android.R.styleable#View_keyboardNavigationCluster
     */
    @ViewDebug.ExportedProperty(category = "focus")
    public final boolean isKeyboardNavigationCluster() {
        return (mPrivateFlags3 & PFLAG3_CLUSTER) != 0;
    }

    /**
     * Searches up the view hierarchy to find the top-most cluster. All deeper/nested clusters
     * will be ignored.
     *
     * @return the keyboard navigation cluster that this view is in (can be this view)
     *         or {@code null} if not in one
     */
    View findKeyboardNavigationCluster() {
        if (mParent instanceof View) {
            View cluster = ((View) mParent).findKeyboardNavigationCluster();
            if (cluster != null) {
                return cluster;
            } else if (isKeyboardNavigationCluster()) {
                return this;
            }
        }
        return null;
    }

    /**
     * Set whether this view is a root of a keyboard navigation cluster.
     *
     * @param isCluster If true, this view is a root of a cluster.
     *
     * @attr ref android.R.styleable#View_keyboardNavigationCluster
     */
    public void setKeyboardNavigationCluster(boolean isCluster) {
        if (isCluster) {
            mPrivateFlags3 |= PFLAG3_CLUSTER;
        } else {
            mPrivateFlags3 &= ~PFLAG3_CLUSTER;
        }
    }

    /**
     * Sets this View as the one which receives focus the next time cluster navigation jumps
     * to the cluster containing this View. This does NOT change focus even if the cluster
     * containing this view is current.
     *
     * @hide
     */
    @TestApi
    public final void setFocusedInCluster() {
        setFocusedInCluster(findKeyboardNavigationCluster());
    }

    private void setFocusedInCluster(View cluster) {
        if (this instanceof ViewGroup) {
            ((ViewGroup) this).mFocusedInCluster = null;
        }
        if (cluster == this) {
            return;
        }
        ViewParent parent = mParent;
        View child = this;
        while (parent instanceof ViewGroup) {
            ((ViewGroup) parent).mFocusedInCluster = child;
            if (parent == cluster) {
                break;
            }
```

```
            child = (View) parent;
            parent = parent.getParent();
        }
    }

    private void updateFocusedInCluster(View oldFocus, @FocusDirection int direction) {
        if (oldFocus != null) {
            View oldCluster = oldFocus.findKeyboardNavigationCluster();
            View cluster = findKeyboardNavigationCluster();
            if (oldCluster != cluster) {
                // Going from one cluster to another, so save last-focused.
                // This covers cluster jumps because they are always FOCUS_DOWN
                oldFocus.setFocusedInCluster(oldCluster);
                if (!(oldFocus.mParent instanceof ViewGroup)) {
                    return;
                }
                if (direction == FOCUS_FORWARD || direction == FOCUS_BACKWARD) {
                    // This is a result of ordered navigation so consider navigation through
                    // the previous cluster "complete" and clear its last-focused memory.
                    ((ViewGroup) oldFocus.mParent).clearFocusedInCluster(oldFocus);
                } else if (oldFocus instanceof ViewGroup
                        && ((ViewGroup) oldFocus).getDescendantFocusability()
                                == ViewGroup.FOCUS_AFTER_DESCENDANTS
                        && ViewRootImpl.isViewDescendantOf(this, oldFocus)) {
                    // This means oldFocus is not focusable since it obviously has a focusable
                    // child (this). Don't restore focus to it in the future.
                    ((ViewGroup) oldFocus.mParent).clearFocusedInCluster(oldFocus);
                }
            }
        }
    }

    /**
     * Returns whether this View should receive focus when the focus is restored for the view
     * hierarchy containing this view.
     * <p>
     * Focus gets restored for a view hierarchy when the root of the hierarchy gets added to a
     * window or serves as a target of cluster navigation.
     *
     * @see #restoreDefaultFocus()
     *
     * @return {@code true} if this view is the default-focus view, {@code false} otherwise
     * @attr ref android.R.styleable#View_focusedByDefault
     */
    @ViewDebug.ExportedProperty(category = "focus")
    public final boolean isFocusedByDefault() {
        return (mPrivateFlags3 & PFLAG3_FOCUSED_BY_DEFAULT) != 0;
    }

    /**
     * Sets whether this View should receive focus when the focus is restored for the view
     * hierarchy containing this view.
     * <p>
     * Focus gets restored for a view hierarchy when the root of the hierarchy gets added to a
     * window or serves as a target of cluster navigation.
     *
     * @param isFocusedByDefault {@code true} to set this view as the default-focus view,
     *                           {@code false} otherwise.
     *
     * @see #restoreDefaultFocus()
     *
     * @attr ref android.R.styleable#View_focusedByDefault
     */
    public void setFocusedByDefault(boolean isFocusedByDefault) {
        if (isFocusedByDefault == ((mPrivateFlags3 & PFLAG3_FOCUSED_BY_DEFAULT) != 0)) {
            return;
        }

        if (isFocusedByDefault) {
            mPrivateFlags3 |= PFLAG3_FOCUSED_BY_DEFAULT;
        } else {
            mPrivateFlags3 &= ~PFLAG3_FOCUSED_BY_DEFAULT;
        }

        if (mParent instanceof ViewGroup) {
            if (isFocusedByDefault) {
                ((ViewGroup) mParent).setDefaultFocus(this);
            } else {
                ((ViewGroup) mParent).clearDefaultFocus(this);
            }
        }
    }
```

```java
/**
 * Returns whether the view hierarchy with this view as a root contain a default-focus view.
 *
 * @return {@code true} if this view has default focus, {@code false} otherwise
 */
boolean hasDefaultFocus() {
    return isFocusedByDefault();
}

/**
 * Find the nearest keyboard navigation cluster in the specified direction.
 * This does not actually give focus to that cluster.
 *
 * @param currentCluster The starting point of the search. Null means the current cluster is not
 *                       found yet
 * @param direction Direction to look
 *
 * @return The nearest keyboard navigation cluster in the specified direction, or null if none
 *         can be found
 */
public View keyboardNavigationClusterSearch(View currentCluster,
        @FocusDirection int direction) {
    if (isKeyboardNavigationCluster()) {
        currentCluster = this;
    }
    if (isRootNamespace()) {
        // Root namespace means we should consider ourselves the top of the
        // tree for group searching; otherwise we could be group searching
        // into other tabs.  see LocalActivityManager and TabHost for more info.
        return FocusFinder.getInstance().findNextKeyboardNavigationCluster(
                this, currentCluster, direction);
    } else if (mParent != null) {
        return mParent.keyboardNavigationClusterSearch(currentCluster, direction);
    }
    return null;
}

/**
 * This method is the last chance for the focused view and its ancestors to
 * respond to an arrow key. This is called when the focused view did not
 * consume the key internally, nor could the view system find a new view in
 * the requested direction to give focus to.
 *
 * @param focused The currently focused view.
 * @param direction The direction focus wants to move. One of FOCUS_UP,
 *        FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT.
 * @return True if the this view consumed this unhandled move.
 */
public boolean dispatchUnhandledMove(View focused, @FocusRealDirection int direction) {
    return false;
}

/**
 * Sets whether this View should use a default focus highlight when it gets focused but doesn't
 * have {@link android.R.attr#state_focused} defined in its background.
 *
 * @param defaultFocusHighlightEnabled {@code true} to set this view to use a default focus
 *                                     highlight, {@code false} otherwise.
 *
 * @attr ref android.R.styleable#View_defaultFocusHighlightEnabled
 */
public void setDefaultFocusHighlightEnabled(boolean defaultFocusHighlightEnabled) {
    mDefaultFocusHighlightEnabled = defaultFocusHighlightEnabled;
}

/**

/**
 * Returns whether this View should use a default focus highlight when it gets focused but
 * doesn't have {@link android.R.attr#state_focused} defined in its background.
 *
 * @return True if this View should use a default focus highlight.
 * @attr ref android.R.styleable#View_defaultFocusHighlightEnabled
 */
@ViewDebug.ExportedProperty(category = "focus")
public final boolean getDefaultFocusHighlightEnabled() {
    return mDefaultFocusHighlightEnabled;
}

/**
 * If a user manually specified the next view id for a particular direction,
```

```
     * use the root to look up the view.
     * @param root The root view of the hierarchy containing this view.
     * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, FOCUS_RIGHT, FOCUS_FORWARD,
     * or FOCUS_BACKWARD.
     * @return The user specified next view, or null if there is none.
     */
    View findUserSetNextFocus(View root, @FocusDirection int direction) {
        switch (direction) {
            case FOCUS_LEFT:
                if (mNextFocusLeftId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusLeftId);
            case FOCUS_RIGHT:
                if (mNextFocusRightId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusRightId);
            case FOCUS_UP:
                if (mNextFocusUpId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusUpId);
            case FOCUS_DOWN:
                if (mNextFocusDownId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusDownId);
            case FOCUS_FORWARD:
                if (mNextFocusForwardId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextFocusForwardId);
            case FOCUS_BACKWARD: {
                if (mID == View.NO_ID) return null;
                final int id = mID;
                return root.findViewByPredicateInsideOut(this, new Predicate<View>() {
                    @Override
                    public boolean test(View t) {
                        return t.mNextFocusForwardId == id;
                    }
                });
            }
        }
        return null;
    }

    /**
     * If a user manually specified the next keyboard-navigation cluster for a particular direction,
     * use the root to look up the view.
     *
     * @param root the root view of the hierarchy containing this view
     * @param direction {@link #FOCUS_FORWARD} or {@link #FOCUS_BACKWARD}
     * @return the user-specified next cluster, or {@code null} if there is none
     */
    View findUserSetNextKeyboardNavigationCluster(View root, @FocusDirection int direction) {
        switch (direction) {
            case FOCUS_FORWARD:
                if (mNextClusterForwardId == View.NO_ID) return null;
                return findViewInsideOutShouldExist(root, mNextClusterForwardId);
            case FOCUS_BACKWARD: {
                if (mID == View.NO_ID) return null;
                final int id = mID;
                return root.findViewByPredicateInsideOut(this,
                        (Predicate<View>) t -> t.mNextClusterForwardId == id);
            }
        }
        return null;
    }

    private View findViewInsideOutShouldExist(View root, int id) {
        if (mMatchIdPredicate == null) {
            mMatchIdPredicate = new MatchIdPredicate();
        }
        mMatchIdPredicate.mId = id;
        View result = root.findViewByPredicateInsideOut(this, mMatchIdPredicate);
        if (result == null) {
            Log.w(VIEW_LOG_TAG, "couldn't find view with id " + id);
        }
        return result;
    }

    /**
     * Find and return all focusable views that are descendants of this view,
     * possibly including this view if it is focusable itself.
     *
     * @param direction The direction of the focus
     * @return A list of focusable views
     */
    public ArrayList<View> getFocusables(@FocusDirection int direction) {
        ArrayList<View> result = new ArrayList<View>(24);
        addFocusables(result, direction);
```

```
    return result;
}

/**
 * Add any focusable views that are descendants of this view (possibly
 * including this view if it is focusable itself) to views.  If we are in touch mode,
 * only add views that are also focusable in touch mode.
 *
 * @param views Focusable views found so far
 * @param direction The direction of the focus
 */
public void addFocusables(ArrayList<View> views, @FocusDirection int direction) {
    addFocusables(views, direction, isInTouchMode() ? FOCUSABLES_TOUCH_MODE : FOCUSABLES_ALL);
}

/**
 * Adds any focusable views that are descendants of this view (possibly
 * including this view if it is focusable itself) to views. This method
 * adds all focusable views regardless if we are in touch mode or
 * only views focusable in touch mode if we are in touch mode or
 * only views that can take accessibility focus if accessibility is enabled
 * depending on the focusable mode parameter.
 *
 * @param views Focusable views found so far or null if all we are interested is
 *        the number of focusables.
 * @param direction The direction of the focus.
 * @param focusableMode The type of focusables to be added.
 *
 * @see #FOCUSABLES_ALL
 * @see #FOCUSABLES_TOUCH_MODE
 */
public void addFocusables(ArrayList<View> views, @FocusDirection int direction,
        @FocusableMode int focusableMode) {
    if (views == null) {
        return;
    }
    if (!isFocusable()) {
        return;
    }
    if ((focusableMode & FOCUSABLES_TOUCH_MODE) == FOCUSABLES_TOUCH_MODE
            && !isFocusableInTouchMode()) {
        return;
    }
    views.add(this);
}

/**
 * Adds any keyboard navigation cluster roots that are descendants of this view (possibly
 * including this view if it is a cluster root itself) to views.
 *
 * @param views Keyboard navigation cluster roots found so far
 * @param direction Direction to look
 */
public void addKeyboardNavigationClusters(
        @NonNull Collection<View> views,
        int direction) {
    if (!isKeyboardNavigationCluster()) {
        return;
    }
    if (!hasFocusable()) {
        return;
    }
    views.add(this);
}

/**
 * Finds the Views that contain given text. The containment is case insensitive.
 * The search is performed by either the text that the View renders or the content
 * description that describes the view for accessibility purposes and the view does
 * not render or both. Clients can specify how the search is to be performed via
 * passing the {@link #FIND_VIEWS_WITH_TEXT} and
 * {@link #FIND_VIEWS_WITH_CONTENT_DESCRIPTION} flags.
 *
 * @param outViews The output list of matching Views.
 * @param searched The text to match against.
 *
 * @see #FIND_VIEWS_WITH_TEXT
 * @see #FIND_VIEWS_WITH_CONTENT_DESCRIPTION
 * @see #setContentDescription(CharSequence)
 */
public void findViewsWithText(ArrayList<View> outViews, CharSequence searched,
        @FindViewFlags int flags) {
```

```
    if (getAccessibilityNodeProvider() != null) {
        if ((flags & FIND_VIEWS_WITH_ACCESSIBILITY_NODE_PROVIDERS) != 0) {
            outViews.add(this);
        }
    } else if ((flags & FIND_VIEWS_WITH_CONTENT_DESCRIPTION) != 0
            && (searched != null && searched.length() > 0)
            && (mContentDescription != null && mContentDescription.length() > 0)) {
        String searchedLowerCase = searched.toString().toLowerCase();
        String contentDescriptionLowerCase = mContentDescription.toString().toLowerCase();
        if (contentDescriptionLowerCase.contains(searchedLowerCase)) {
            outViews.add(this);
        }
    }
}

/**
 * Find and return all touchable views that are descendants of this view,
 * possibly including this view if it is touchable itself.
 *
 * @return A list of touchable views
 */
public ArrayList<View> getTouchables() {
    ArrayList<View> result = new ArrayList<View>();
    addTouchables(result);
    return result;
}

/**
 * Add any touchable views that are descendants of this view (possibly
 * including this view if it is touchable itself) to views.
 *
 * @param views Touchable views found so far
 */
public void addTouchables(ArrayList<View> views) {
    final int viewFlags = mViewFlags;

    if (((viewFlags & CLICKABLE) == CLICKABLE || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE
            || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE)
            && (viewFlags & ENABLED_MASK) == ENABLED) {
        views.add(this);
    }
}

/**
 * Returns whether this View is accessibility focused.
 *
 * @return True if this View is accessibility focused.
 */
public boolean isAccessibilityFocused() {
    return (mPrivateFlags2 & PFLAG2_ACCESSIBILITY_FOCUSED) != 0;
}

/**
 * Call this to try to give accessibility focus to this view.
 *
 * A view will not actually take focus if {@link AccessibilityManager#isEnabled()}
 * returns false or the view is no visible or the view already has accessibility
 * focus.
 *
 * See also {@link #focusSearch(int)}, which is what you call to say that you
 * have focus, and you want your parent to look for the next one.
 *
 * @return Whether this view actually took accessibility focus.
 *
 * @hide
 */
public boolean requestAccessibilityFocus() {
    AccessibilityManager manager = AccessibilityManager.getInstance(mContext);
    if (!manager.isEnabled() || !manager.isTouchExplorationEnabled()) {
        return false;
    }
    if ((mViewFlags & VISIBILITY_MASK) != VISIBLE) {
        return false;
    }
    if ((mPrivateFlags2 & PFLAG2_ACCESSIBILITY_FOCUSED) == 0) {
        mPrivateFlags2 |= PFLAG2_ACCESSIBILITY_FOCUSED;
        ViewRootImpl viewRootImpl = getViewRootImpl();
        if (viewRootImpl != null) {
            viewRootImpl.setAccessibilityFocus(this, null);
        }
        invalidate();
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_ACCESSIBILITY_FOCUSED);
```

```
            return true;
        }
        return false;
    }

    /**
     * Call this to try to clear accessibility focus of this view.
     *
     * See also {@link #focusSearch(int)}, which is what you call to say that you
     * have focus, and you want your parent to look for the next one.
     *
     * @hide
     */
    public void clearAccessibilityFocus() {
        clearAccessibilityFocusNoCallbacks(0);

        // Clear the global reference of accessibility focus if this view or
        // any of its descendants had accessibility focus. This will NOT send
        // an event or update internal state if focus is cleared from a
        // descendant view, which may leave views in inconsistent states.
        final ViewRootImpl viewRootImpl = getViewRootImpl();
        if (viewRootImpl != null) {
            final View focusHost = viewRootImpl.getAccessibilityFocusedHost();
            if (focusHost != null && ViewRootImpl.isViewDescendantOf(focusHost, this)) {
                viewRootImpl.setAccessibilityFocus(null, null);
            }
        }
    }

    private void sendAccessibilityHoverEvent(int eventType) {
        // Since we are not delivering to a client accessibility events from not
        // important views (unless the clinet request that) we need to fire the
        // event from the deepest view exposed to the client. As a consequence if
        // the user crosses a not exposed view the client will see enter and exit
        // of the exposed predecessor followed by and enter and exit of that same
        // predecessor when entering and exiting the not exposed descendant. This
        // is fine since the client has a clear idea which view is hovered at the
        // price of a couple more events being sent. This is a simple and
        // working solution.
        View source = this;
        while (true) {
            if (source.includeForAccessibility()) {
                source.sendAccessibilityEvent(eventType);
                return;
            }
            ViewParent parent = source.getParent();
            if (parent instanceof View) {
                source = (View) parent;
            } else {
                return;
            }
        }
    }

    /**
     * Clears accessibility focus without calling any callback methods
     * normally invoked in {@link #clearAccessibilityFocus()}. This method
     * is used separately from that one for clearing accessibility focus when
     * giving this focus to another view.
     *
     * @param action The action, if any, that led to focus being cleared. Set to
     * AccessibilityNodeInfo#ACTION_ACCESSIBILITY_FOCUS to specify that focus is moving within
     * the window.
     */
    void clearAccessibilityFocusNoCallbacks(int action) {
        if ((mPrivateFlags2 & PFLAG2_ACCESSIBILITY_FOCUSED) != 0) {
            mPrivateFlags2 &= ~PFLAG2_ACCESSIBILITY_FOCUSED;
            invalidate();
            if (AccessibilityManager.getInstance(mContext).isEnabled()) {
                AccessibilityEvent event = AccessibilityEvent.obtain(
                        AccessibilityEvent.TYPE_VIEW_ACCESSIBILITY_FOCUS_CLEARED);
                event.setAction(action);
                if (mAccessibilityDelegate != null) {
                    mAccessibilityDelegate.sendAccessibilityEventUnchecked(this, event);
                } else {
                    sendAccessibilityEventUnchecked(event);
                }
            }
        }
    }

    /**
```

```
     * Call this to try to give focus to a specific view or to one of its
     * descendants.
     *
     * A view will not actually take focus if it is not focusable ({@link #isFocusable} returns
     * false), or if it is focusable and it is not focusable in touch mode
     * ({@link #isFocusableInTouchMode}) while the device is in touch mode.
     *
     * See also {@link #focusSearch(int)}, which is what you call to say that you
     * have focus, and you want your parent to look for the next one.
     *
     * This is equivalent to calling {@link #requestFocus(int, Rect)} with arguments
     * {@link #FOCUS_DOWN} and <code>null</code>.
     *
     * @return Whether this view or one of its descendants actually took focus.
     */
    public final boolean requestFocus() {
        return requestFocus(View.FOCUS_DOWN);
    }

    /**
     * This will request focus for whichever View was last focused within this
     * cluster before a focus-jump out of it.
     *
     * @hide
     */
    @TestApi
    public boolean restoreFocusInCluster(@FocusRealDirection int direction) {
        // Prioritize focusableByDefault over algorithmic focus selection.
        if (restoreDefaultFocus()) {
            return true;
        }
        return requestFocus(direction);
    }

    /**
     * This will request focus for whichever View not in a cluster was last focused before a
     * focus-jump to a cluster. If no non-cluster View has previously had focus, this will focus
     * the "first" focusable view it finds.
     *
     * @hide
     */
    @TestApi
    public boolean restoreFocusNotInCluster() {
        return requestFocus(View.FOCUS_DOWN);
    }

    /**
     * Gives focus to the default-focus view in the view hierarchy that has this view as a root.
     * If the default-focus view cannot be found, falls back to calling {@link #requestFocus(int)}.
     *
     * @return Whether this view or one of its descendants actually took focus
     */
    public boolean restoreDefaultFocus() {
        return requestFocus(View.FOCUS_DOWN);
    }

    /**
     * Call this to try to give focus to a specific view or to one of its
     * descendants and give it a hint about what direction focus is heading.
     *
     * A view will not actually take focus if it is not focusable ({@link #isFocusable} returns
     * false), or if it is focusable and it is not focusable in touch mode
     * ({@link #isFocusableInTouchMode}) while the device is in touch mode.
     *
     * See also {@link #focusSearch(int)}, which is what you call to say that you
     * have focus, and you want your parent to look for the next one.
     *
     * This is equivalent to calling {@link #requestFocus(int, Rect)} with
     * <code>null</code> set for the previously focused rectangle.
     *
     * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT
     * @return Whether this view or one of its descendants actually took focus.
     */
    public final boolean requestFocus(int direction) {
        return requestFocus(direction, null);
    }

    /**
     * Call this to try to give focus to a specific view or to one of its descendants
     * and give it hints about the direction and a specific rectangle that the focus
     * is coming from.  The rectangle can help give larger views a finer grained hint
     * about where focus is coming from, and therefore, where to show selection, or
```

```
     * forward focus change internally.
     *
     * A view will not actually take focus if it is not focusable ({@link #isFocusable} returns
     * false), or if it is focusable and it is not focusable in touch mode
     * ({@link #isFocusableInTouchMode}) while the device is in touch mode.
     *
     * A View will not take focus if it is not visible.
     *
     * A View will not take focus if one of its parents has
     * {@link android.view.ViewGroup#getDescendantFocusability()} equal to
     * {@link ViewGroup#FOCUS_BLOCK_DESCENDANTS}.
     *
     * See also {@link #focusSearch(int)}, which is what you call to say that you
     * have focus, and you want your parent to look for the next one.
     *
     * You may wish to override this method if your custom {@link View} has an internal
     * {@link View} that it wishes to forward the request to.
     *
     * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and FOCUS_RIGHT
     * @param previouslyFocusedRect The rectangle (in this View's coordinate system)
     *        to give a finer grained hint about where focus is coming from.  May be null
     *        if there is no hint.
     * @return Whether this view or one of its descendants actually took focus.
     */
    public boolean requestFocus(int direction, Rect previouslyFocusedRect) {
        return requestFocusNoSearch(direction, previouslyFocusedRect);
    }

    private boolean requestFocusNoSearch(int direction, Rect previouslyFocusedRect) {
        // need to be focusable
        if ((mViewFlags & FOCUSABLE) != FOCUSABLE
                || (mViewFlags & VISIBILITY_MASK) != VISIBLE) {
            return false;
        }

        // need to be focusable in touch mode if in touch mode
        if (isInTouchMode() &&
            (FOCUSABLE_IN_TOUCH_MODE != (mViewFlags & FOCUSABLE_IN_TOUCH_MODE))) {
               return false;
        }

        // need to not have any parents blocking us
        if (hasAncestorThatBlocksDescendantFocus()) {
            return false;
        }

        handleFocusGainInternal(direction, previouslyFocusedRect);
        return true;
    }

    /**
     * Call this to try to give focus to a specific view or to one of its descendants. This is a
     * special variant of {@link #requestFocus() } that will allow views that are not focusable in
     * touch mode to request focus when they are touched.
     *
     * @return Whether this view or one of its descendants actually took focus.
     *
     * @see #isInTouchMode()
     *
     */
    public final boolean requestFocusFromTouch() {
        // Leave touch mode if we need to
        if (isInTouchMode()) {
            ViewRootImpl viewRoot = getViewRootImpl();
            if (viewRoot != null) {
                viewRoot.ensureTouchMode(false);
            }
        }
        return requestFocus(View.FOCUS_DOWN);
    }

    /**
     * @return Whether any ancestor of this view blocks descendant focus.
     */
    private boolean hasAncestorThatBlocksDescendantFocus() {
        final boolean focusableInTouchMode = isFocusableInTouchMode();
        ViewParent ancestor = mParent;
        while (ancestor instanceof ViewGroup) {
            final ViewGroup vgAncestor = (ViewGroup) ancestor;
            if (vgAncestor.getDescendantFocusability() == ViewGroup.FOCUS_BLOCK_DESCENDANTS
                    || (!focusableInTouchMode && vgAncestor.shouldBlockFocusForTouchscreen())) {
                return true;
```

```
        } else {
            ancestor = vgAncestor.getParent();
        }
    }
    return false;
}

/**
 * Gets the mode for determining whether this View is important for accessibility.
 * A view is important for accessibility if it fires accessibility events and if it
 * is reported to accessibility services that query the screen.
 *
 * @return The mode for determining whether a view is important for accessibility, one
 * of {@link #IMPORTANT_FOR_ACCESSIBILITY_AUTO}, {@link #IMPORTANT_FOR_ACCESSIBILITY_YES},
 * {@link #IMPORTANT_FOR_ACCESSIBILITY_NO}, or
 * {@link #IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS}.
 *
 * @attr ref android.R.styleable#View_importantForAccessibility
 *
 * @see #IMPORTANT_FOR_ACCESSIBILITY_YES
 * @see #IMPORTANT_FOR_ACCESSIBILITY_NO
 * @see #IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS
 * @see #IMPORTANT_FOR_ACCESSIBILITY_AUTO
 */
@ViewDebug.ExportedProperty(category = "accessibility", mapping = {
        @ViewDebug.IntToString(from = IMPORTANT_FOR_ACCESSIBILITY_AUTO, to = "auto"),
        @ViewDebug.IntToString(from = IMPORTANT_FOR_ACCESSIBILITY_YES, to = "yes"),
        @ViewDebug.IntToString(from = IMPORTANT_FOR_ACCESSIBILITY_NO, to = "no"),
        @ViewDebug.IntToString(from = IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS,
                to = "noHideDescendants")
    })
public int getImportantForAccessibility() {
    return (mPrivateFlags2 & PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK)
            >> PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT;
}

/**
 * Sets the live region mode for this view. This indicates to accessibility
 * services whether they should automatically notify the user about changes
 * to the view's content description or text, or to the content descriptions
 * or text of the view's children (where applicable).
 * <p>
 * For example, in a login screen with a TextView that displays an "incorrect
 * password" notification, that view should be marked as a live region with
 * mode {@link #ACCESSIBILITY_LIVE_REGION_POLITE}.
 * <p>
 * To disable change notifications for this view, use
 * {@link #ACCESSIBILITY_LIVE_REGION_NONE}. This is the default live region
 * mode for most views.
 * <p>
 * To indicate that the user should be notified of changes, use
 * {@link #ACCESSIBILITY_LIVE_REGION_POLITE}.
 * <p>
 * If the view's changes should interrupt ongoing speech and notify the user
 * immediately, use {@link #ACCESSIBILITY_LIVE_REGION_ASSERTIVE}.
 *
 * @param mode The live region mode for this view, one of:
 *        <ul>
 *        <li>{@link #ACCESSIBILITY_LIVE_REGION_NONE}
 *        <li>{@link #ACCESSIBILITY_LIVE_REGION_POLITE}
 *        <li>{@link #ACCESSIBILITY_LIVE_REGION_ASSERTIVE}
 *        </ul>
 * @attr ref android.R.styleable#View_accessibilityLiveRegion
 */
public void setAccessibilityLiveRegion(int mode) {
    if (mode != getAccessibilityLiveRegion()) {
        mPrivateFlags2 &= ~PFLAG2_ACCESSIBILITY_LIVE_REGION_MASK;
        mPrivateFlags2 |= (mode << PFLAG2_ACCESSIBILITY_LIVE_REGION_SHIFT)
                & PFLAG2_ACCESSIBILITY_LIVE_REGION_MASK;
        notifyViewAccessibilityStateChangedIfNeeded(
                AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
    }
}

/**
 * Gets the live region mode for this View.
 *
 * @return The live region mode for the view.
 *
 * @attr ref android.R.styleable#View_accessibilityLiveRegion
 *
 * @see #setAccessibilityLiveRegion(int)
```

```
     */
    public int getAccessibilityLiveRegion() {
        return (mPrivateFlags2 & PFLAG2_ACCESSIBILITY_LIVE_REGION_MASK)
                >> PFLAG2_ACCESSIBILITY_LIVE_REGION_SHIFT;
    }

    /**
     * Sets how to determine whether this view is important for accessibility
     * which is if it fires accessibility events and if it is reported to
     * accessibility services that query the screen.
     *
     * @param mode How to determine whether this view is important for accessibility.
     *
     * @attr ref android.R.styleable#View_importantForAccessibility
     *
     * @see #IMPORTANT_FOR_ACCESSIBILITY_YES
     * @see #IMPORTANT_FOR_ACCESSIBILITY_NO
     * @see #IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS
     * @see #IMPORTANT_FOR_ACCESSIBILITY_AUTO
     */
    public void setImportantForAccessibility(int mode) {
        final int oldMode = getImportantForAccessibility();
        if (mode != oldMode) {
            final boolean hideDescendants =
                    mode == IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS;

            // If this node or its descendants are no longer important, try to
            // clear accessibility focus.
            if (mode == IMPORTANT_FOR_ACCESSIBILITY_NO || hideDescendants) {
                final View focusHost = findAccessibilityFocusHost(hideDescendants);
                if (focusHost != null) {
                    focusHost.clearAccessibilityFocus();
                }
            }

            // If we're moving between AUTO and another state, we might not need
            // to send a subtree changed notification. We'll store the computed
            // importance, since we'll need to check it later to make sure.
            final boolean maySkipNotify = oldMode == IMPORTANT_FOR_ACCESSIBILITY_AUTO
                    || mode == IMPORTANT_FOR_ACCESSIBILITY_AUTO;
            final boolean oldIncludeForAccessibility = maySkipNotify && includeForAccessibility();
            mPrivateFlags2 &= ~PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK;
            mPrivateFlags2 |= (mode << PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT)
                    & PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK;
            if (!maySkipNotify || oldIncludeForAccessibility != includeForAccessibility()) {
                notifySubtreeAccessibilityStateChangedIfNeeded();
            } else {
                notifyViewAccessibilityStateChangedIfNeeded(
                        AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
            }
        }
    }

    /**
     * Returns the view within this view's hierarchy that is hosting
     * accessibility focus.
     *
     * @param searchDescendants whether to search for focus in descendant views
     * @return the view hosting accessibility focus, or {@code null}
     */
    private View findAccessibilityFocusHost(boolean searchDescendants) {
        if (isAccessibilityFocusedViewOrHost()) {
            return this;
        }

        if (searchDescendants) {
            final ViewRootImpl viewRoot = getViewRootImpl();
            if (viewRoot != null) {
                final View focusHost = viewRoot.getAccessibilityFocusedHost();
                if (focusHost != null && ViewRootImpl.isViewDescendantOf(focusHost, this)) {
                    return focusHost;
                }
            }
        }

        return null;
    }

    /**
     * Computes whether this view should be exposed for accessibility. In
     * general, views that are interactive or provide information are exposed
     * while views that serve only as containers are hidden.
```

```
     * <p>
     * If an ancestor of this view has importance
     * {@link #IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS}, this method
     * returns <code>false</code>.
     * <p>
     * Otherwise, the value is computed according to the view's
     * {@link #getImportantForAccessibility()} value:
     * <ol>
     * <li>{@link #IMPORTANT_FOR_ACCESSIBILITY_NO} or
     * {@link #IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS}, return <code>false
     * </code>
     * <li>{@link #IMPORTANT_FOR_ACCESSIBILITY_YES}, return <code>true</code>
     * <li>{@link #IMPORTANT_FOR_ACCESSIBILITY_AUTO}, return <code>true</code> if
     * view satisfies any of the following:
     * <ul>
     * <li>Is actionable, e.g. {@link #isClickable()},
     * {@link #isLongClickable()}, or {@link #isFocusable()}
     * <li>Has an {@link AccessibilityDelegate}
     * <li>Has an interaction listener, e.g. {@link OnTouchListener},
     * {@link OnKeyListener}, etc.
     * <li>Is an accessibility live region, e.g.
     * {@link #getAccessibilityLiveRegion()} is not
     * {@link #ACCESSIBILITY_LIVE_REGION_NONE}.
     * </ul>
     * </ol>
     *
     * @return Whether the view is exposed for accessibility.
     * @see #setImportantForAccessibility(int)
     * @see #getImportantForAccessibility()
     */
    public boolean isImportantForAccessibility() {
        final int mode = (mPrivateFlags2 & PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_MASK)
                >> PFLAG2_IMPORTANT_FOR_ACCESSIBILITY_SHIFT;
        if (mode == IMPORTANT_FOR_ACCESSIBILITY_NO
                || mode == IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS) {
            return false;
        }

        // Check parent mode to ensure we're not hidden.
        ViewParent parent = mParent;
        while (parent instanceof View) {
            if (((View) parent).getImportantForAccessibility()
                    == IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS) {
                return false;
            }
            parent = parent.getParent();
        }

        return mode == IMPORTANT_FOR_ACCESSIBILITY_YES || isActionableForAccessibility()
                || hasListenersForAccessibility() || getAccessibilityNodeProvider() != null
                || getAccessibilityLiveRegion() != ACCESSIBILITY_LIVE_REGION_NONE;
    }

    /**
     * Gets the parent for accessibility purposes. Note that the parent for
     * accessibility is not necessary the immediate parent. It is the first
     * predecessor that is important for accessibility.
     *
     * @return The parent for accessibility purposes.
     */
    public ViewParent getParentForAccessibility() {
        if (mParent instanceof View) {
            View parentView = (View) mParent;
            if (parentView.includeForAccessibility()) {
                return mParent;
            } else {
                return mParent.getParentForAccessibility();
            }
        }
        return null;
    }

    /**
     * Adds the children of this View relevant for accessibility to the given list
     * as output. Since some Views are not important for accessibility the added
     * child views are not necessarily direct children of this view, rather they are
     * the first level of descendants important for accessibility.
     *
     * @param outChildren The output list that will receive children for accessibility.
     */
    public void addChildrenForAccessibility(ArrayList<View> outChildren) {
```

```
}

/**
 * Whether to regard this view for accessibility. A view is regarded for
 * accessibility if it is important for accessibility or the querying
 * accessibility service has explicitly requested that view not
 * important for accessibility are regarded.
 *
 * @return Whether to regard the view for accessibility.
 *
 * @hide
 */
public boolean includeForAccessibility() {
    if (mAttachInfo != null) {
        return (mAttachInfo.mAccessibilityFetchFlags
                & AccessibilityNodeInfo.FLAG_INCLUDE_NOT_IMPORTANT_VIEWS) != 0
                || isImportantForAccessibility();
    }
    return false;
}

/**
 * Returns whether the View is considered actionable from
 * accessibility perspective. Such view are important for
 * accessibility.
 *
 * @return True if the view is actionable for accessibility.
 *
 * @hide
 */
public boolean isActionableForAccessibility() {
    return (isClickable() || isLongClickable() || isFocusable());
}

/**
 * Returns whether the View has registered callbacks which makes it
 * important for accessibility.
 *
 * @return True if the view is actionable for accessibility.
 */
private boolean hasListenersForAccessibility() {
    ListenerInfo info = getListenerInfo();
    return mTouchDelegate != null || info.mOnKeyListener != null
            || info.mOnTouchListener != null || info.mOnGenericMotionListener != null
            || info.mOnHoverListener != null || info.mOnDragListener != null;
}

/**
 * Notifies that the accessibility state of this view changed. The change
 * is local to this view and does not represent structural changes such
 * as children and parent. For example, the view became focusable. The
 * notification is at at most once every
 * {@link ViewConfiguration#getSendRecurringAccessibilityEventsInterval()}
 * to avoid unnecessary load to the system. Also once a view has a pending
 * notification this method is a NOP until the notification has been sent.
 *
 * @hide
 */
public void notifyViewAccessibilityStateChangedIfNeeded(int changeType) {
    if (!AccessibilityManager.getInstance(mContext).isEnabled() || mAttachInfo == null) {
        return;
    }
    // If this is a live region, we should send a subtree change event
    // from this view immediately. Otherwise, we can let it propagate up.
    if (getAccessibilityLiveRegion() != ACCESSIBILITY_LIVE_REGION_NONE) {
        final AccessibilityEvent event = AccessibilityEvent.obtain();
        event.setEventType(AccessibilityEvent.TYPE_WINDOW_CONTENT_CHANGED);
        event.setContentChangeTypes(changeType);
        sendAccessibilityEventUnchecked(event);
    } else if (mParent != null) {
        try {
            mParent.notifySubtreeAccessibilityStateChanged(this, this, changeType);
        } catch (AbstractMethodError e) {
            Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                    " does not fully implement ViewParent", e);
        }
    }
}

/**
 * Notifies that the accessibility state of this view changed. The change
 * is *not* local to this view and does represent structural changes such
```

```
     * as children and parent. For example, the view size changed. The
     * notification is at at most once every
     * {@link ViewConfiguration#getSendRecurringAccessibilityEventsInterval()}
     * to avoid unnecessary load to the system. Also once a view has a pending
     * notification this method is a NOP until the notification has been sent.
     *
     * @hide
     */
    public void notifySubtreeAccessibilityStateChangedIfNeeded() {
        if (!AccessibilityManager.getInstance(mContext).isEnabled() || mAttachInfo == null) {
            return;
        }
        if ((mPrivateFlags2 & PFLAG2_SUBTREE_ACCESSIBILITY_STATE_CHANGED) == 0) {
            mPrivateFlags2 |= PFLAG2_SUBTREE_ACCESSIBILITY_STATE_CHANGED;
            if (mParent != null) {
                try {
                    mParent.notifySubtreeAccessibilityStateChanged(
                            this, this, AccessibilityEvent.CONTENT_CHANGE_TYPE_SUBTREE);
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
            }
        }
    }

    /**
     * Change the visibility of the View without triggering any other changes. This is
     * important for transitions, where visibility changes should not adjust focus or
     * trigger a new layout. This is only used when the visibility has already been changed
     * and we need a transient value during an animation. When the animation completes,
     * the original visibility value is always restored.
     *
     * @param visibility One of {@link #VISIBLE}, {@link #INVISIBLE}, or {@link #GONE}.
     * @hide
     */
    public void setTransitionVisibility(@Visibility int visibility) {
        mViewFlags = (mViewFlags & ~View.VISIBILITY_MASK) | visibility;
    }

    /**
     * Reset the flag indicating the accessibility state of the subtree rooted
     * at this view changed.
     */
    void resetSubtreeAccessibilityStateChanged() {
        mPrivateFlags2 &= ~PFLAG2_SUBTREE_ACCESSIBILITY_STATE_CHANGED;
    }

    /**
     * Report an accessibility action to this view's parents for delegated processing.
     *
     * <p>Implementations of {@link #performAccessibilityAction(int, Bundle)} may internally
     * call this method to delegate an accessibility action to a supporting parent. If the parent
     * returns true from its
     * {@link ViewParent#onNestedPrePerformAccessibilityAction(View, int, android.os.Bundle)}
     * method this method will return true to signify that the action was consumed.</p>
     *
     * <p>This method is useful for implementing nested scrolling child views. If
     * {@link #isNestedScrollingEnabled()} returns true and the action is a scrolling action
     * a custom view implementation may invoke this method to allow a parent to consume the
     * scroll first. If this method returns true the custom view should skip its own scrolling
     * behavior.</p>
     *
     * @param action Accessibility action to delegate
     * @param arguments Optional action arguments
     * @return true if the action was consumed by a parent
     */
    public boolean dispatchNestedPrePerformAccessibilityAction(int action, Bundle arguments) {
        for (ViewParent p = getParent(); p != null; p = p.getParent()) {
            if (p.onNestedPrePerformAccessibilityAction(this, action, arguments)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Performs the specified accessibility action on the view. For
     * possible accessibility actions look at {@link AccessibilityNodeInfo}.
     * <p>
     * If an {@link AccessibilityDelegate} has been specified via calling
     * {@link #setAccessibilityDelegate(AccessibilityDelegate)} its
```

```
     * {@link AccessibilityDelegate#performAccessibilityAction(View, int, Bundle)}
     * is responsible for handling this call.
     * </p>
     *
     * <p>The default implementation will delegate
     * {@link AccessibilityNodeInfo#ACTION_SCROLL_BACKWARD} and
     * {@link AccessibilityNodeInfo#ACTION_SCROLL_FORWARD} to nested scrolling parents if
     * {@link #isNestedScrollingEnabled() nested scrolling is enabled} on this view.</p>
     *
     * @param action The action to perform.
     * @param arguments Optional action arguments.
     * @return Whether the action was performed.
     */
    public boolean performAccessibilityAction(int action, Bundle arguments) {
      if (mAccessibilityDelegate != null) {
          return mAccessibilityDelegate.performAccessibilityAction(this, action, arguments);
      } else {
          return performAccessibilityActionInternal(action, arguments);
      }
    }

    /**
     * @see #performAccessibilityAction(int, Bundle)
     *
     * Note: Called from the default {@link AccessibilityDelegate}.
     *
     * @hide
     */
    public boolean performAccessibilityActionInternal(int action, Bundle arguments) {
        if (isNestedScrollingEnabled()
                && (action == AccessibilityNodeInfo.ACTION_SCROLL_BACKWARD
                || action == AccessibilityNodeInfo.ACTION_SCROLL_FORWARD
                || action == R.id.accessibilityActionScrollUp
                || action == R.id.accessibilityActionScrollLeft
                || action == R.id.accessibilityActionScrollDown
                || action == R.id.accessibilityActionScrollRight)) {
            if (dispatchNestedPrePerformAccessibilityAction(action, arguments)) {
                return true;
            }
        }

        switch (action) {
            case AccessibilityNodeInfo.ACTION_CLICK: {
                if (isClickable()) {
                    performClick();
                    return true;
                }
            } break;
            case AccessibilityNodeInfo.ACTION_LONG_CLICK: {
                if (isLongClickable()) {
                    performLongClick();
                    return true;
                }
            } break;
            case AccessibilityNodeInfo.ACTION_FOCUS: {
                if (!hasFocus()) {
                    // Get out of touch mode since accessibility
                    // wants to move focus around.
                    getViewRootImpl().ensureTouchMode(false);
                    return requestFocus();
                }
            } break;
            case AccessibilityNodeInfo.ACTION_CLEAR_FOCUS: {
                if (hasFocus()) {
                    clearFocus();
                    return !isFocused();
                }
            } break;
            case AccessibilityNodeInfo.ACTION_SELECT: {
                if (!isSelected()) {
                    setSelected(true);
                    return isSelected();
                }
            } break;
            case AccessibilityNodeInfo.ACTION_CLEAR_SELECTION: {
                if (isSelected()) {
                    setSelected(false);
                    return !isSelected();
                }
            } break;
            case AccessibilityNodeInfo.ACTION_ACCESSIBILITY_FOCUS: {
                if (!isAccessibilityFocused()) {
```

```
                return requestAccessibilityFocus();
            }
        } break;
        case AccessibilityNodeInfo.ACTION_CLEAR_ACCESSIBILITY_FOCUS: {
            if (isAccessibilityFocused()) {
                clearAccessibilityFocus();
                return true;
            }
        } break;
        case AccessibilityNodeInfo.ACTION_NEXT_AT_MOVEMENT_GRANULARITY: {
            if (arguments != null) {
                final int granularity = arguments.getInt(
                        AccessibilityNodeInfo.ACTION_ARGUMENT_MOVEMENT_GRANULARITY_INT);
                final boolean extendSelection = arguments.getBoolean(
                        AccessibilityNodeInfo.ACTION_ARGUMENT_EXTEND_SELECTION_BOOLEAN);
                return traverseAtGranularity(granularity, true, extendSelection);
            }
        } break;
        case AccessibilityNodeInfo.ACTION_PREVIOUS_AT_MOVEMENT_GRANULARITY: {
            if (arguments != null) {
                final int granularity = arguments.getInt(
                        AccessibilityNodeInfo.ACTION_ARGUMENT_MOVEMENT_GRANULARITY_INT);
                final boolean extendSelection = arguments.getBoolean(
                        AccessibilityNodeInfo.ACTION_ARGUMENT_EXTEND_SELECTION_BOOLEAN);
                return traverseAtGranularity(granularity, false, extendSelection);
            }
        } break;
        case AccessibilityNodeInfo.ACTION_SET_SELECTION: {
            CharSequence text = getIterableTextForAccessibility();
            if (text == null) {
                return false;
            }
            final int start = (arguments != null) ? arguments.getInt(
                    AccessibilityNodeInfo.ACTION_ARGUMENT_SELECTION_START_INT, -1) : -1;
            final int end = (arguments != null) ? arguments.getInt(
            AccessibilityNodeInfo.ACTION_ARGUMENT_SELECTION_END_INT, -1) : -1;
            // Only cursor position can be specified (selection length == 0)
            if ((getAccessibilitySelectionStart() != start
                    || getAccessibilitySelectionEnd() != end)
                    && (start == end)) {
                setAccessibilitySelection(start, end);
                notifyViewAccessibilityStateChangedIfNeeded(
                        AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
                return true;
            }
        } break;
        case R.id.accessibilityActionShowOnScreen: {
            if (mAttachInfo != null) {
                final Rect r = mAttachInfo.mTmpInvalRect;
                getDrawingRect(r);
                return requestRectangleOnScreen(r, true);
            }
        } break;
        case R.id.accessibilityActionContextClick: {
            if (isContextClickable()) {
                performContextClick();
                return true;
            }
        } break;
    }
    return false;
}

private boolean traverseAtGranularity(int granularity, boolean forward,
        boolean extendSelection) {
    CharSequence text = getIterableTextForAccessibility();
    if (text == null || text.length() == 0) {
        return false;
    }
    TextSegmentIterator iterator = getIteratorForGranularity(granularity);
    if (iterator == null) {
        return false;
    }
    int current = getAccessibilitySelectionEnd();
    if (current == ACCESSIBILITY_CURSOR_POSITION_UNDEFINED) {
        current = forward ? 0 : text.length();
    }
    final int[] range = forward ? iterator.following(current) : iterator.preceding(current);
    if (range == null) {
        return false;
    }
    final int segmentStart = range[0];
```

```
    final int segmentEnd = range[1];
    int selectionStart;
    int selectionEnd;
    if (extendSelection && isAccessibilitySelectionExtendable()) {
        selectionStart = getAccessibilitySelectionStart();
        if (selectionStart == ACCESSIBILITY_CURSOR_POSITION_UNDEFINED) {
            selectionStart = forward ? segmentStart : segmentEnd;
        }
        selectionEnd = forward ? segmentEnd : segmentStart;
    } else {
        selectionStart = selectionEnd= forward ? segmentEnd : segmentStart;
    }
    setAccessibilitySelection(selectionStart, selectionEnd);
    final int action = forward ? AccessibilityNodeInfo.ACTION_NEXT_AT_MOVEMENT_GRANULARITY
            : AccessibilityNodeInfo.ACTION_PREVIOUS_AT_MOVEMENT_GRANULARITY;
    sendViewTextTraversedAtGranularityEvent(action, granularity, segmentStart, segmentEnd);
    return true;
}

/**
 * Gets the text reported for accessibility purposes.
 *
 * @return The accessibility text.
 *
 * @hide
 */
public CharSequence getIterableTextForAccessibility() {
    return getContentDescription();
}

/**
 * Gets whether accessibility selection can be extended.
 *
 * @return If selection is extensible.
 *
 * @hide
 */
public boolean isAccessibilitySelectionExtendable() {
    return false;
}

/**
 * @hide
 */
public int getAccessibilitySelectionStart() {
    return mAccessibilityCursorPosition;
}

/**
 * @hide
 */
public int getAccessibilitySelectionEnd() {
    return getAccessibilitySelectionStart();
}

/**
 * @hide
 */
public void setAccessibilitySelection(int start, int end) {
    if (start ==  end && end == mAccessibilityCursorPosition) {
        return;
    }
    if (start >= 0 && start == end && end <= getIterableTextForAccessibility().length()) {
        mAccessibilityCursorPosition = start;
    } else {
        mAccessibilityCursorPosition = ACCESSIBILITY_CURSOR_POSITION_UNDEFINED;
    }
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_SELECTION_CHANGED);
}

private void sendViewTextTraversedAtGranularityEvent(int action, int granularity,
        int fromIndex, int toIndex) {
    if (mParent == null) {
        return;
    }
    AccessibilityEvent event = AccessibilityEvent.obtain(
            AccessibilityEvent.TYPE_VIEW_TEXT_TRAVERSED_AT_MOVEMENT_GRANULARITY);
    onInitializeAccessibilityEvent(event);
    onPopulateAccessibilityEvent(event);
    event.setFromIndex(fromIndex);
    event.setToIndex(toIndex);
    event.setAction(action);
```

```
    event.setMovementGranularity(granularity);
    mParent.requestSendAccessibilityEvent(this, event);
}

/**
 * @hide
 */
public TextSegmentIterator getIteratorForGranularity(int granularity) {
    switch (granularity) {
        case AccessibilityNodeInfo.MOVEMENT_GRANULARITY_CHARACTER: {
            CharSequence text = getIterableTextForAccessibility();
            if (text != null && text.length() > 0) {
                CharacterTextSegmentIterator iterator =
                    CharacterTextSegmentIterator.getInstance(
                            mContext.getResources().getConfiguration().locale);
                iterator.initialize(text.toString());
                return iterator;
            }
        } break;
        case AccessibilityNodeInfo.MOVEMENT_GRANULARITY_WORD: {
            CharSequence text = getIterableTextForAccessibility();
            if (text != null && text.length() > 0) {
                WordTextSegmentIterator iterator =
                    WordTextSegmentIterator.getInstance(
                            mContext.getResources().getConfiguration().locale);
                iterator.initialize(text.toString());
                return iterator;
            }
        } break;
        case AccessibilityNodeInfo.MOVEMENT_GRANULARITY_PARAGRAPH: {
            CharSequence text = getIterableTextForAccessibility();
            if (text != null && text.length() > 0) {
                ParagraphTextSegmentIterator iterator =
                    ParagraphTextSegmentIterator.getInstance();
                iterator.initialize(text.toString());
                return iterator;
            }
        } break;
    }
    return null;
}

/**
 * Tells whether the {@link View} is in the state between {@link #onStartTemporaryDetach()}
 * and {@link #onFinishTemporaryDetach()}.
 *
 * <p>This method always returns {@code true} when called directly or indirectly from
 * {@link #onStartTemporaryDetach()}. The return value when called directly or indirectly from
 * {@link #onFinishTemporaryDetach()}, however, depends on the OS version.
 * <ul>
 *     <li>{@code true} on {@link android.os.Build.VERSION_CODES#N API 24}</li>
 *     <li>{@code false} on {@link android.os.Build.VERSION_CODES#N_MR1 API 25}} and later</li>
 * </ul>
 * </p>
 *
 * @return {@code true} when the View is in the state between {@link #onStartTemporaryDetach()}
 * and {@link #onFinishTemporaryDetach()}.
 */
public final boolean isTemporarilyDetached() {
    return (mPrivateFlags3 & PFLAG3_TEMPORARY_DETACH) != 0;
}

/**
 * Dispatch {@link #onStartTemporaryDetach()} to this View and its direct children if this is
 * a container View.
 */
@CallSuper
public void dispatchStartTemporaryDetach() {
    mPrivateFlags3 |= PFLAG3_TEMPORARY_DETACH;
    notifyEnterOrExitForAutoFillIfNeeded(false);
    onStartTemporaryDetach();
}

/**
 * This is called when a container is going to temporarily detach a child, with
 * {@link ViewGroup#detachViewFromParent(View) ViewGroup.detachViewFromParent}.
 * It will either be followed by {@link #onFinishTemporaryDetach()} or
 * {@link #onDetachedFromWindow()} when the container is done.
 */
public void onStartTemporaryDetach() {
    removeUnsetPressCallback();
    mPrivateFlags |= PFLAG_CANCEL_NEXT_UP_EVENT;
```

```
}

/**
 * Dispatch {@link #onFinishTemporaryDetach()} to this View and its direct children if this is
 * a container View.
 */
@CallSuper
public void dispatchFinishTemporaryDetach() {
    mPrivateFlags3 &= ~PFLAG3_TEMPORARY_DETACH;
    onFinishTemporaryDetach();
    if (hasWindowFocus() && hasFocus()) {
        InputMethodManager.getInstance().focusIn(this);
    }
    notifyEnterOrExitForAutoFillIfNeeded(true);
}

/**
 * Called after {@link #onStartTemporaryDetach} when the container is done
 * changing the view.
 */
public void onFinishTemporaryDetach() {
}

/**
 * Return the global {@link KeyEvent.DispatcherState KeyEvent.DispatcherState}
 * for this view's window.  Returns null if the view is not currently attached
 * to the window.  Normally you will not need to use this directly, but
 * just use the standard high-level event callbacks like
 * {@link #onKeyDown(int, KeyEvent)}.
 */
public KeyEvent.DispatcherState getKeyDispatcherState() {
    return mAttachInfo != null ? mAttachInfo.mKeyDispatchState : null;
}

/**
 * Dispatch a key event before it is processed by any input method
 * associated with the view hierarchy.  This can be used to intercept
 * key events in special situations before the IME consumes them; a
 * typical example would be handling the BACK key to update the application's
 * UI instead of allowing the IME to see it and close itself.
 *
 * @param event The key event to be dispatched.
 * @return True if the event was handled, false otherwise.
 */
public boolean dispatchKeyEventPreIme(KeyEvent event) {
    return onKeyPreIme(event.getKeyCode(), event);
}

/**
 * Dispatch a key event to the next view on the focus path. This path runs
 * from the top of the view tree down to the currently focused view. If this
 * view has focus, it will dispatch to itself. Otherwise it will dispatch
 * the next node down the focus path. This method also fires any key
 * listeners.
 *
 * @param event The key event to be dispatched.
 * @return True if the event was handled, false otherwise.
 */
public boolean dispatchKeyEvent(KeyEvent event) {
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onKeyEvent(event, 0);
    }

    // Give any attached key listener a first crack at the event.
    //noinspection SimplifiableIfStatement
    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnKeyListener != null && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnKeyListener.onKey(this, event.getKeyCode(), event)) {
        return true;
    }

    if (event.dispatch(this, mAttachInfo != null
            ? mAttachInfo.mKeyDispatchState : null, this)) {
        return true;
    }

    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }
    return false;
}
```

```java
/**
 * Dispatches a key shortcut event.
 *
 * @param event The key event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
public boolean dispatchKeyShortcutEvent(KeyEvent event) {
    return onKeyShortcut(event.getKeyCode(), event);
}

/**
 * Pass the touch screen motion event down to the target view, or this
 * view if it is the target.
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
public boolean dispatchTouchEvent(MotionEvent event) {
    // If the event should be handled by accessibility focus first.
    if (event.isTargetAccessibilityFocus()) {
        // We don't have focus or no virtual descendant has it, do not handle the event.
        if (!isAccessibilityFocusedViewOrHost()) {
            return false;
        }
        // We have focus and got the event, then use normal event dispatch.
        event.setTargetAccessibilityFocus(false);
    }

    boolean result = false;

    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(event, 0);
    }

    final int actionMasked = event.getActionMasked();
    if (actionMasked == MotionEvent.ACTION_DOWN) {
        // Defensive cleanup for new gesture
        stopNestedScroll();
    }

    if (onFilterTouchEventForSecurity(event)) {
        if ((mViewFlags & ENABLED_MASK) == ENABLED && handleScrollBarDragging(event)) {
            result = true;
        }
        //noinspection SimplifiableIfStatement
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnTouchListener != null
                && (mViewFlags & ENABLED_MASK) == ENABLED
                && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }

        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }

    if (!result && mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }

    // Clean up after nested scrolls if this is the end of a gesture;
    // also cancel it if we tried an ACTION_DOWN but we didn't want the rest
    // of the gesture.
    if (actionMasked == MotionEvent.ACTION_UP ||
            actionMasked == MotionEvent.ACTION_CANCEL ||
            (actionMasked == MotionEvent.ACTION_DOWN && !result)) {
        stopNestedScroll();
    }

    return result;
}

boolean isAccessibilityFocusedViewOrHost() {
    return isAccessibilityFocused() || (getViewRootImpl() != null && getViewRootImpl()
            .getAccessibilityFocusedHost() == this);
}

/**
 * Filter the touch event to apply security policies.
 *
 * @param event The motion event to be filtered.
```

```
     * @return True if the event should be dispatched, false if the event should be dropped.
     *
     * @see #getFilterTouchesWhenObscured
     */
    public boolean onFilterTouchEventForSecurity(MotionEvent event) {
        //noinspection RedundantIfStatement
        if ((mViewFlags & FILTER_TOUCHES_WHEN_OBSCURED) != 0
                && (event.getFlags() & MotionEvent.FLAG_WINDOW_IS_OBSCURED) != 0) {
            // Window is obscured, drop this touch.
            return false;
        }
        return true;
    }

    /**
     * Pass a trackball motion event down to the focused view.
     *
     * @param event The motion event to be dispatched.
     * @return True if the event was handled by the view, false otherwise.
     */
    public boolean dispatchTrackballEvent(MotionEvent event) {
        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onTrackballEvent(event, 0);
        }

        return onTrackballEvent(event);
    }

    /**
     * Pass a captured pointer event down to the focused view.
     *
     * @param event The motion event to be dispatched.
     * @return True if the event was handled by the view, false otherwise.
     */
    public boolean dispatchCapturedPointerEvent(MotionEvent event) {
        if (!hasPointerCapture()) {
            return false;
        }
        //noinspection SimplifiableIfStatement
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnCapturedPointerListener != null
                && li.mOnCapturedPointerListener.onCapturedPointer(this, event)) {
            return true;
        }
        return onCapturedPointerEvent(event);
    }

    /**
     * Dispatch a generic motion event.
     * <p>
     * Generic motion events with source class {@link InputDevice#SOURCE_CLASS_POINTER}
     * are delivered to the view under the pointer.  All other generic motion events are
     * delivered to the focused view.  Hover events are handled specially and are delivered
     * to {@link #onHoverEvent(MotionEvent)}.
     * </p>
     *
     * @param event The motion event to be dispatched.
     * @return True if the event was handled by the view, false otherwise.
     */
    public boolean dispatchGenericMotionEvent(MotionEvent event) {
        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onGenericMotionEvent(event, 0);
        }

        final int source = event.getSource();
        if ((source & InputDevice.SOURCE_CLASS_POINTER) != 0) {
            final int action = event.getAction();
            if (action == MotionEvent.ACTION_HOVER_ENTER
                    || action == MotionEvent.ACTION_HOVER_MOVE
                    || action == MotionEvent.ACTION_HOVER_EXIT) {
                if (dispatchHoverEvent(event)) {
                    return true;
                }
            } else if (dispatchGenericPointerEvent(event)) {
                return true;
            }
        } else if (dispatchGenericFocusedEvent(event)) {
            return true;
        }

        if (dispatchGenericMotionEventInternal(event)) {
            return true;
```

```
    }

    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }
    return false;
}

private boolean dispatchGenericMotionEventInternal(MotionEvent event) {
    //noinspection SimplifiableIfStatement
    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnGenericMotionListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnGenericMotionListener.onGenericMotion(this, event)) {
        return true;
    }

    if (onGenericMotionEvent(event)) {
        return true;
    }

    final int actionButton = event.getActionButton();
    switch (event.getActionMasked()) {
        case MotionEvent.ACTION_BUTTON_PRESS:
            if (isContextClickable() && !mInContextButtonPress && !mHasPerformedLongPress
                    && (actionButton == MotionEvent.BUTTON_STYLUS_PRIMARY
                    || actionButton == MotionEvent.BUTTON_SECONDARY)) {
                if (performContextClick(event.getX(), event.getY())) {
                    mInContextButtonPress = true;
                    setPressed(true, event.getX(), event.getY());
                    removeTapCallback();
                    removeLongPressCallback();
                    return true;
                }
            }
            break;

        case MotionEvent.ACTION_BUTTON_RELEASE:
            if (mInContextButtonPress && (actionButton == MotionEvent.BUTTON_STYLUS_PRIMARY
                    || actionButton == MotionEvent.BUTTON_SECONDARY)) {
                mInContextButtonPress = false;
                mIgnoreNextUpEvent = true;
            }
            break;
    }

    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }
    return false;
}

/**
 * Dispatch a hover event.
 * <p>
 * Do not call this method directly.
 * Call {@link #dispatchGenericMotionEvent(MotionEvent)} instead.
 * </p>
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
protected boolean dispatchHoverEvent(MotionEvent event) {
    ListenerInfo li = mListenerInfo;
    //noinspection SimplifiableIfStatement
    if (li != null && li.mOnHoverListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnHoverListener.onHover(this, event)) {
        return true;
    }

    return onHoverEvent(event);
}

/**
 * Returns true if the view has a child to which it has recently sent
 * {@link MotionEvent#ACTION_HOVER_ENTER}.  If this view is hovered and
 * it does not have a hovered child, then it must be the innermost hovered view.
 * @hide
 */
protected boolean hasHoveredChild() {
    return false;
```

```
}

/**
 * Dispatch a generic motion event to the view under the first pointer.
 * <p>
 * Do not call this method directly.
 * Call {@link #dispatchGenericMotionEvent(MotionEvent)} instead.
 * </p>
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
protected boolean dispatchGenericPointerEvent(MotionEvent event) {
    return false;
}

/**
 * Dispatch a generic motion event to the currently focused view.
 * <p>
 * Do not call this method directly.
 * Call {@link #dispatchGenericMotionEvent(MotionEvent)} instead.
 * </p>
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
protected boolean dispatchGenericFocusedEvent(MotionEvent event) {
    return false;
}

/**
 * Dispatch a pointer event.
 * <p>
 * Dispatches touch related pointer events to {@link #onTouchEvent(MotionEvent)} and all
 * other events to {@link #onGenericMotionEvent(MotionEvent)}.  This separation of concerns
 * reinforces the invariant that {@link #onTouchEvent(MotionEvent)} is really about touches
 * and should not be expected to handle other pointing device features.
 * </p>
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 * @hide
 */
public final boolean dispatchPointerEvent(MotionEvent event) {
    if (event.isTouchEvent()) {
        return dispatchTouchEvent(event);
    } else {
        return dispatchGenericMotionEvent(event);
    }
}

/**
 * Called when the window containing this view gains or loses window focus.
 * ViewGroups should override to route to their children.
 *
 * @param hasFocus True if the window containing this view now has focus,
 *        false otherwise.
 */
public void dispatchWindowFocusChanged(boolean hasFocus) {
    onWindowFocusChanged(hasFocus);
}

/**
 * Called when the window containing this view gains or loses focus.  Note
 * that this is separate from view focus: to receive key events, both
 * your view and its window must have focus.  If a window is displayed
 * on top of yours that takes input focus, then your own window will lose
 * focus but the view focus will remain unchanged.
 *
 * @param hasWindowFocus True if the window containing this view now has
 *        focus, false otherwise.
 */
public void onWindowFocusChanged(boolean hasWindowFocus) {
    InputMethodManager imm = InputMethodManager.peekInstance();
    if (!hasWindowFocus) {
        if (isPressed()) {
            setPressed(false);
        }
        mPrivateFlags3 &= ~PFLAG3_FINGER_DOWN;
        if (imm != null && (mPrivateFlags & PFLAG_FOCUSED) != 0) {
            imm.focusOut(this);
        }
```

```
            removeLongPressCallback();
            removeTapCallback();
            onFocusLost();
        } else if (imm != null && (mPrivateFlags & PFLAG_FOCUSED) != 0) {
            imm.focusIn(this);
        }

        notifyEnterOrExitForAutoFillIfNeeded(hasWindowFocus);

        refreshDrawableState();
    }

    /**
     * Returns true if this view is in a window that currently has window focus.
     * Note that this is not the same as the view itself having focus.
     *
     * @return True if this view is in a window that currently has window focus.
     */
    public boolean hasWindowFocus() {
        return mAttachInfo != null && mAttachInfo.mHasWindowFocus;
    }

    /**
     * Dispatch a view visibility change down the view hierarchy.
     * ViewGroups should override to route to their children.
     * @param changedView The view whose visibility changed. Could be 'this' or
     * an ancestor view.
     * @param visibility The new visibility of changedView: {@link #VISIBLE},
     * {@link #INVISIBLE} or {@link #GONE}.
     */
    protected void dispatchVisibilityChanged(@NonNull View changedView,
            @Visibility int visibility) {
        onVisibilityChanged(changedView, visibility);
    }

    /**
     * Called when the visibility of the view or an ancestor of the view has
     * changed.
     *
     * @param changedView The view whose visibility changed. May be
     *                    {@code this} or an ancestor view.
     * @param visibility The new visibility, one of {@link #VISIBLE},
     *                   {@link #INVISIBLE} or {@link #GONE}.
     */
    protected void onVisibilityChanged(@NonNull View changedView, @Visibility int visibility) {
    }

    /**
     * Dispatch a hint about whether this view is displayed. For instance, when
     * a View moves out of the screen, it might receives a display hint indicating
     * the view is not displayed. Applications should not <em>rely</em> on this hint
     * as there is no guarantee that they will receive one.
     *
     * @param hint A hint about whether or not this view is displayed:
     * {@link #VISIBLE} or {@link #INVISIBLE}.
     */
    public void dispatchDisplayHint(@Visibility int hint) {
        onDisplayHint(hint);
    }

    /**
     * Gives this view a hint about whether is displayed or not. For instance, when
     * a View moves out of the screen, it might receives a display hint indicating
     * the view is not displayed. Applications should not <em>rely</em> on this hint
     * as there is no guarantee that they will receive one.
     *
     * @param hint A hint about whether or not this view is displayed:
     * {@link #VISIBLE} or {@link #INVISIBLE}.
     */
    protected void onDisplayHint(@Visibility int hint) {
    }

    /**
     * Dispatch a window visibility change down the view hierarchy.
     * ViewGroups should override to route to their children.
     *
     * @param visibility The new visibility of the window.
     *
     * @see #onWindowVisibilityChanged(int)
     */
    public void dispatchWindowVisibilityChanged(@Visibility int visibility) {
        onWindowVisibilityChanged(visibility);
```

```
}

/**
 * Called when the window containing has change its visibility
 * (between {@link #GONE}, {@link #INVISIBLE}, and {@link #VISIBLE}).  Note
 * that this tells you whether or not your window is being made visible
 * to the window manager; this does <em>not</em> tell you whether or not
 * your window is obscured by other windows on the screen, even if it
 * is itself visible.
 *
 * @param visibility The new visibility of the window.
 */
protected void onWindowVisibilityChanged(@Visibility int visibility) {
    if (visibility == VISIBLE) {
        initialAwakenScrollBars();
    }
}

/**
 * Internal dispatching method for {@link #onVisibilityAggregated}. Overridden by
 * ViewGroup. Intended to only be called when {@link #isAttachedToWindow()},
 * {@link #getWindowVisibility()} is {@link #VISIBLE} and this view's parent {@link #isShown()}.
 *
 * @param isVisible true if this view's visibility to the user is uninterrupted by its
 *                  ancestors or by window visibility
 * @return true if this view is visible to the user, not counting clipping or overlapping
 */
boolean dispatchVisibilityAggregated(boolean isVisible) {
    final boolean thisVisible = getVisibility() == VISIBLE;
    // If we're not visible but something is telling us we are, ignore it.
    if (thisVisible || !isVisible) {
        onVisibilityAggregated(isVisible);
    }
    return thisVisible && isVisible;
}

/**
 * Called when the user-visibility of this View is potentially affected by a change
 * to this view itself, an ancestor view or the window this view is attached to.
 *
 * @param isVisible true if this view and all of its ancestors are {@link #VISIBLE}
 *                  and this view's window is also visible
 */
@CallSuper
public void onVisibilityAggregated(boolean isVisible) {
    if (isVisible && mAttachInfo != null) {
        initialAwakenScrollBars();
    }

    final Drawable dr = mBackground;
    if (dr != null && isVisible != dr.isVisible()) {
        dr.setVisible(isVisible, false);
    }
    final Drawable hl = mDefaultFocusHighlight;
    if (hl != null && isVisible != hl.isVisible()) {
        hl.setVisible(isVisible, false);
    }
    final Drawable fg = mForegroundInfo != null ? mForegroundInfo.mDrawable : null;
    if (fg != null && isVisible != fg.isVisible()) {
        fg.setVisible(isVisible, false);
    }

    if (isAutofillable()) {
        AutofillManager afm = getAutofillManager();

        if (afm != null && getAutofillViewId() > LAST_APP_AUTOFILL_ID) {
            if (mVisibilityChangeForAutofillHandler != null) {
                mVisibilityChangeForAutofillHandler.removeMessages(0);
            }

            // If the view is in the background but still part of the hierarchy this is called
            // with isVisible=false. Hence visibility==false requires further checks
            if (isVisible) {
                afm.notifyViewVisibilityChanged(this, true);
            } else {
                if (mVisibilityChangeForAutofillHandler == null) {
                    mVisibilityChangeForAutofillHandler =
                            new VisibilityChangeForAutofillHandler(afm, this);
                }
                // Let current operation (e.g. removal of the view from the hierarchy)
                // finish before checking state
                mVisibilityChangeForAutofillHandler.obtainMessage(0, this).sendToTarget();
```

```
            }
        }
    }
}

/**
 * Returns the current visibility of the window this view is attached to
 * (either {@link #GONE}, {@link #INVISIBLE}, or {@link #VISIBLE}).
 *
 * @return Returns the current visibility of the view's window.
 */
@Visibility
public int getWindowVisibility() {
    return mAttachInfo != null ? mAttachInfo.mWindowVisibility : GONE;
}

/**
 * Retrieve the overall visible display size in which the window this view is
 * attached to has been positioned in.  This takes into account screen
 * decorations above the window, for both cases where the window itself
 * is being position inside of them or the window is being placed under
 * then and covered insets are used for the window to position its content
 * inside.  In effect, this tells you the available area where content can
 * be placed and remain visible to users.
 *
 * <p>This function requires an IPC back to the window manager to retrieve
 * the requested information, so should not be used in performance critical
 * code like drawing.
 *
 * @param outRect Filled in with the visible display frame.  If the view
 * is not attached to a window, this is simply the raw display size.
 */
public void getWindowVisibleDisplayFrame(Rect outRect) {
    if (mAttachInfo != null) {
        try {
            mAttachInfo.mSession.getDisplayFrame(mAttachInfo.mWindow, outRect);
        } catch (RemoteException e) {
            return;
        }
        // XXX This is really broken, and probably all needs to be done
        // in the window manager, and we need to know more about whether
        // we want the area behind or in front of the IME.
        final Rect insets = mAttachInfo.mVisibleInsets;
        outRect.left += insets.left;
        outRect.top += insets.top;
        outRect.right -= insets.right;
        outRect.bottom -= insets.bottom;
        return;
    }
    // The view is not attached to a display so we don't have a context.
    // Make a best guess about the display size.
    Display d = DisplayManagerGlobal.getInstance().getRealDisplay(Display.DEFAULT_DISPLAY);
    d.getRectSize(outRect);
}

/**
 * Like {@link #getWindowVisibleDisplayFrame}, but returns the "full" display frame this window
 * is currently in without any insets.
 *
 * @hide
 */
public void getWindowDisplayFrame(Rect outRect) {
    if (mAttachInfo != null) {
        try {
            mAttachInfo.mSession.getDisplayFrame(mAttachInfo.mWindow, outRect);
        } catch (RemoteException e) {
            return;
        }
        return;
    }
    // The view is not attached to a display so we don't have a context.
    // Make a best guess about the display size.
    Display d = DisplayManagerGlobal.getInstance().getRealDisplay(Display.DEFAULT_DISPLAY);
    d.getRectSize(outRect);
}

/**
 * Dispatch a notification about a resource configuration change down
 * the view hierarchy.
 * ViewGroups should override to route to their children.
 *
 * @param newConfig The new resource configuration.
```

```
     *
     * @see #onConfigurationChanged(android.content.res.Configuration)
     */
    public void dispatchConfigurationChanged(Configuration newConfig) {
        onConfigurationChanged(newConfig);
    }

    /**
     * Called when the current configuration of the resources being used
     * by the application have changed.  You can use this to decide when
     * to reload resources that can changed based on orientation and other
     * configuration characteristics.  You only need to use this if you are
     * not relying on the normal {@link android.app.Activity} mechanism of
     * recreating the activity instance upon a configuration change.
     *
     * @param newConfig The new resource configuration.
     */
    protected void onConfigurationChanged(Configuration newConfig) {
    }

    /**
     * Private function to aggregate all per-view attributes in to the view
     * root.
     */
    void dispatchCollectViewAttributes(AttachInfo attachInfo, int visibility) {
        performCollectViewAttributes(attachInfo, visibility);
    }

    void performCollectViewAttributes(AttachInfo attachInfo, int visibility) {
        if ((visibility & VISIBILITY_MASK) == VISIBLE) {
            if ((mViewFlags & KEEP_SCREEN_ON) == KEEP_SCREEN_ON) {
                attachInfo.mKeepScreenOn = true;
            }
            attachInfo.mSystemUiVisibility |= mSystemUiVisibility;
            ListenerInfo li = mListenerInfo;
            if (li != null && li.mOnSystemUiVisibilityChangeListener != null) {
                attachInfo.mHasSystemUiListeners = true;
            }
        }
    }

    void needGlobalAttributesUpdate(boolean force) {
        final AttachInfo ai = mAttachInfo;
        if (ai != null && !ai.mRecomputeGlobalAttributes) {
            if (force || ai.mKeepScreenOn || (ai.mSystemUiVisibility != 0)
                    || ai.mHasSystemUiListeners) {
                ai.mRecomputeGlobalAttributes = true;
            }
        }
    }

    /**
     * Returns whether the device is currently in touch mode.  Touch mode is entered
     * once the user begins interacting with the device by touch, and affects various
     * things like whether focus is always visible to the user.
     *
     * @return Whether the device is in touch mode.
     */
    @ViewDebug.ExportedProperty
    public boolean isInTouchMode() {
        if (mAttachInfo != null) {
            return mAttachInfo.mInTouchMode;
        } else {
            return ViewRootImpl.isInTouchMode();
        }
    }

    /**
     * Returns the context the view is running in, through which it can
     * access the current theme, resources, etc.
     *
     * @return The view's Context.
     */
    @ViewDebug.CapturedViewProperty
    public final Context getContext() {
        return mContext;
    }

    /**
     * Handle a key event before it is processed by any input method
     * associated with the view hierarchy.  This can be used to intercept
     * key events in special situations before the IME consumes them; a
```

```
     * typical example would be handling the BACK key to update the application's
     * UI instead of allowing the IME to see it and close itself.
     *
     * @param keyCode The value in event.getKeyCode().
     * @param event Description of the key event.
     * @return If you handled the event, return true. If you want to allow the
     *         event to be handled by the next receiver, return false.
     */
    public boolean onKeyPreIme(int keyCode, KeyEvent event) {
        return false;
    }

    /**
     * Default implementation of {@link KeyEvent.Callback#onKeyDown(int, KeyEvent)
     * KeyEvent.Callback.onKeyDown()}: perform press of the view
     * when {@link KeyEvent#KEYCODE_DPAD_CENTER} or {@link KeyEvent#KEYCODE_ENTER}
     * is released, if the view is enabled and clickable.
     * <p>
     * Key presses in software keyboards will generally NOT trigger this
     * listener, although some may elect to do so in some situations. Do not
     * rely on this to catch software key presses.
     *
     * @param keyCode a key code that represents the button pressed, from
     *                {@link android.view.KeyEvent}
     * @param event the KeyEvent object that defines the button action
     */
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (KeyEvent.isConfirmKey(keyCode)) {
            if ((mViewFlags & ENABLED_MASK) == DISABLED) {
                return true;
            }

            if (event.getRepeatCount() == 0) {
                // Long clickable items don't necessarily have to be clickable.
                final boolean clickable = (mViewFlags & CLICKABLE) == CLICKABLE
                        || (mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE;
                if (clickable || (mViewFlags & TOOLTIP) == TOOLTIP) {
                    // For the purposes of menu anchoring and drawable hotspots,
                    // key events are considered to be at the center of the view.
                    final float x = getWidth() / 2f;
                    final float y = getHeight() / 2f;
                    if (clickable) {
                        setPressed(true, x, y);
                    }
                    checkForLongClick(0, x, y);
                    return true;
                }
            }
        }

        return false;
    }

    /**
     * Default implementation of {@link KeyEvent.Callback#onKeyLongPress(int, KeyEvent)
     * KeyEvent.Callback.onKeyLongPress()}: always returns false (doesn't handle
     * the event).
     * <p>Key presses in software keyboards will generally NOT trigger this listener,
     * although some may elect to do so in some situations. Do not rely on this to
     * catch software key presses.
     */
    public boolean onKeyLongPress(int keyCode, KeyEvent event) {
        return false;
    }

    /**
     * Default implementation of {@link KeyEvent.Callback#onKeyUp(int, KeyEvent)
     * KeyEvent.Callback.onKeyUp()}: perform clicking of the view
     * when {@link KeyEvent#KEYCODE_DPAD_CENTER}, {@link KeyEvent#KEYCODE_ENTER}
     * or {@link KeyEvent#KEYCODE_SPACE} is released.
     * <p>Key presses in software keyboards will generally NOT trigger this listener,
     * although some may elect to do so in some situations. Do not rely on this to
     * catch software key presses.
     *
     * @param keyCode A key code that represents the button pressed, from
     *                {@link android.view.KeyEvent}.
     * @param event   The KeyEvent object that defines the button action.
     */
    public boolean onKeyUp(int keyCode, KeyEvent event) {
        if (KeyEvent.isConfirmKey(keyCode)) {
            if ((mViewFlags & ENABLED_MASK) == DISABLED) {
                return true;
```

```
        }
        if ((mViewFlags & CLICKABLE) == CLICKABLE && isPressed()) {
            setPressed(false);

            if (!mHasPerformedLongPress) {
                // This is a tap, so remove the longpress check
                removeLongPressCallback();
                if (!event.isCanceled()) {
                    return performClick();
                }
            }
        }
    }
    return false;
}

/**
 * Default implementation of {@link KeyEvent.Callback#onKeyMultiple(int, int, KeyEvent)
 * KeyEvent.Callback.onKeyMultiple()}: always returns false (doesn't handle
 * the event).
 * <p>Key presses in software keyboards will generally NOT trigger this listener,
 * although some may elect to do so in some situations. Do not rely on this to
 * catch software key presses.
 *
 * @param keyCode     A key code that represents the button pressed, from
 *                    {@link android.view.KeyEvent}.
 * @param repeatCount The number of times the action was made.
 * @param event       The KeyEvent object that defines the button action.
 */
public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event) {
    return false;
}

/**
 * Called on the focused view when a key shortcut event is not handled.
 * Override this method to implement local key shortcuts for the View.
 * Key shortcuts can also be implemented by setting the
 * {@link MenuItem#setShortcut(char, char) shortcut} property of menu items.
 *
 * @param keyCode The value in event.getKeyCode().
 * @param event Description of the key event.
 * @return If you handled the event, return true. If you want to allow the
 *         event to be handled by the next receiver, return false.
 */
public boolean onKeyShortcut(int keyCode, KeyEvent event) {
    return false;
}

/**
 * Check whether the called view is a text editor, in which case it
 * would make sense to automatically display a soft input window for
 * it.  Subclasses should override this if they implement
 * {@link #onCreateInputConnection(EditorInfo)} to return true if
 * a call on that method would return a non-null InputConnection, and
 * they are really a first-class editor that the user would normally
 * start typing on when the go into a window containing your view.
 *
 * <p>The default implementation always returns false.  This does
 * <em>not</em> mean that its {@link #onCreateInputConnection(EditorInfo)}
 * will not be called or the user can not otherwise perform edits on your
 * view; it is just a hint to the system that this is not the primary
 * purpose of this view.
 *
 * @return Returns true if this view is a text editor, else false.
 */
public boolean onCheckIsTextEditor() {
    return false;
}

/**
 * Create a new InputConnection for an InputMethod to interact
 * with the view.  The default implementation returns null, since it doesn't
 * support input methods.  You can override this to implement such support.
 * This is only needed for views that take focus and text input.
 *
 * <p>When implementing this, you probably also want to implement
 * {@link #onCheckIsTextEditor()} to indicate you will return a
 * non-null InputConnection.</p>
 *
 * <p>Also, take good care to fill in the {@link android.view.inputmethod.EditorInfo}
 * object correctly and in its entirety, so that the connected IME can rely
 * on its values. For example, {@link android.view.inputmethod.EditorInfo#initialSelStart}
```

```
     * and  {@link android.view.inputmethod.EditorInfo#initialSelEnd} members
     * must be filled in with the correct cursor position for IMEs to work correctly
     * with your application.</p>
     *
     * @param outAttrs Fill in with attribute information about the connection.
     */
    public InputConnection onCreateInputConnection(EditorInfo outAttrs) {
        return null;
    }

    /**
     * Called by the {@link android.view.inputmethod.InputMethodManager}
     * when a view who is not the current
     * input connection target is trying to make a call on the manager.  The
     * default implementation returns false; you can override this to return
     * true for certain views if you are performing InputConnection proxying
     * to them.
     * @param view The View that is making the InputMethodManager call.
     * @return Return true to allow the call, false to reject.
     */
    public boolean checkInputConnectionProxy(View view) {
        return false;
    }

    /**
     * Show the context menu for this view. It is not safe to hold on to the
     * menu after returning from this method.
     *
     * You should normally not overload this method. Overload
     * {@link #onCreateContextMenu(ContextMenu)} or define an
     * {@link OnCreateContextMenuListener} to add items to the context menu.
     *
     * @param menu The context menu to populate
     */
    public void createContextMenu(ContextMenu menu) {
        ContextMenuInfo menuInfo = getContextMenuInfo();

        // Sets the current menu info so all items added to menu will have
        // my extra info set.
        ((MenuBuilder)menu).setCurrentMenuInfo(menuInfo);

        onCreateContextMenu(menu);
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnCreateContextMenuListener != null) {
            li.mOnCreateContextMenuListener.onCreateContextMenu(menu, this, menuInfo);
        }

        // Clear the extra information so subsequent items that aren't mine don't
        // have my extra info.
        ((MenuBuilder)menu).setCurrentMenuInfo(null);

        if (mParent != null) {
            mParent.createContextMenu(menu);
        }
    }

    /**
     * Views should implement this if they have extra information to associate
     * with the context menu. The return result is supplied as a parameter to
     * the {@link OnCreateContextMenuListener#onCreateContextMenu(ContextMenu, View, ContextMenuInfo)}
     * callback.
     *
     * @return Extra information about the item for which the context menu
     *         should be shown. This information will vary across different
     *         subclasses of View.
     */
    protected ContextMenuInfo getContextMenuInfo() {
        return null;
    }

    /**
     * Views should implement this if the view itself is going to add items to
     * the context menu.
     *
     * @param menu the context menu to populate
     */
    protected void onCreateContextMenu(ContextMenu menu) {
    }

    /**
     * Implement this method to handle trackball motion events.  The
     * <em>relative</em> movement of the trackball since the last event
```

```
     * can be retrieve with {@link MotionEvent#getX MotionEvent.getX()} and
     * {@link MotionEvent#getY MotionEvent.getY()}.  These are normalized so
     * that a movement of 1 corresponds to the user pressing one DPAD key (so
     * they will often be fractional values, representing the more fine-grained
     * movement information available from a trackball).
     *
     * @param event The motion event.
     * @return True if the event was handled, false otherwise.
     */
    public boolean onTrackballEvent(MotionEvent event) {
        return false;
    }

    /**
     * Implement this method to handle generic motion events.
     * <p>
     * Generic motion events describe joystick movements, mouse hovers, track pad
     * touches, scroll wheel movements and other input events.  The
     * {@link MotionEvent#getSource() source} of the motion event specifies
     * the class of input that was received.  Implementations of this method
     * must examine the bits in the source before processing the event.
     * The following code example shows how this is done.
     * </p><p>
     * Generic motion events with source class {@link InputDevice#SOURCE_CLASS_POINTER}
     * are delivered to the view under the pointer.  All other generic motion events are
     * delivered to the focused view.
     * </p>
     * <pre> public boolean onGenericMotionEvent(MotionEvent event) {
     *     if (event.isFromSource(InputDevice.SOURCE_CLASS_JOYSTICK)) {
     *         if (event.getAction() == MotionEvent.ACTION_MOVE) {
     *             // process the joystick movement...
     *             return true;
     *         }
     *     }
     *     if (event.isFromSource(InputDevice.SOURCE_CLASS_POINTER)) {
     *         switch (event.getAction()) {
     *             case MotionEvent.ACTION_HOVER_MOVE:
     *                 // process the mouse hover movement...
     *                 return true;
     *             case MotionEvent.ACTION_SCROLL:
     *                 // process the scroll wheel movement...
     *                 return true;
     *         }
     *     }
     *     return super.onGenericMotionEvent(event);
     * }</pre>
     *
     * @param event The generic motion event being processed.
     * @return True if the event was handled, false otherwise.
     */
    public boolean onGenericMotionEvent(MotionEvent event) {
        return false;
    }

    /**
     * Implement this method to handle hover events.
     * <p>
     * This method is called whenever a pointer is hovering into, over, or out of the
     * bounds of a view and the view is not currently being touched.
     * Hover events are represented as pointer events with action
     * {@link MotionEvent#ACTION_HOVER_ENTER}, {@link MotionEvent#ACTION_HOVER_MOVE},
     * or {@link MotionEvent#ACTION_HOVER_EXIT}.
     * </p>
     * <ul>
     * <li>The view receives a hover event with action {@link MotionEvent#ACTION_HOVER_ENTER}
     * when the pointer enters the bounds of the view.</li>
     * <li>The view receives a hover event with action {@link MotionEvent#ACTION_HOVER_MOVE}
     * when the pointer has already entered the bounds of the view and has moved.</li>
     * <li>The view receives a hover event with action {@link MotionEvent#ACTION_HOVER_EXIT}
     * when the pointer has exited the bounds of the view or when the pointer is
     * about to go down due to a button click, tap, or similar user action that
     * causes the view to be touched.</li>
     * </ul>
     * <p>
     * The view should implement this method to return true to indicate that it is
     * handling the hover event, such as by changing its drawable state.
     * </p><p>
     * The default implementation calls {@link #setHovered} to update the hovered state
     * of the view when a hover enter or hover exit event is received, if the view
     * is enabled and is clickable.  The default implementation also sends hover
     * accessibility events.
     * </p>
```

```
     *
     * @param event The motion event that describes the hover.
     * @return True if the view handled the hover event.
     *
     * @see #isHovered
     * @see #setHovered
     * @see #onHoverChanged
     */
    public boolean onHoverEvent(MotionEvent event) {
        // The root view may receive hover (or touch) events that are outside the bounds of
        // the window.  This code ensures that we only send accessibility events for
        // hovers that are actually within the bounds of the root view.
        final int action = event.getActionMasked();
        if (!mSendingHoverAccessibilityEvents) {
            if ((action == MotionEvent.ACTION_HOVER_ENTER
                    || action == MotionEvent.ACTION_HOVER_MOVE)
                    && !hasHoveredChild()
                    && pointInView(event.getX(), event.getY())) {
                sendAccessibilityHoverEvent(AccessibilityEvent.TYPE_VIEW_HOVER_ENTER);
                mSendingHoverAccessibilityEvents = true;
            }
        } else {
            if (action == MotionEvent.ACTION_HOVER_EXIT
                    || (action == MotionEvent.ACTION_MOVE
                            && !pointInView(event.getX(), event.getY()))) {
                mSendingHoverAccessibilityEvents = false;
                sendAccessibilityHoverEvent(AccessibilityEvent.TYPE_VIEW_HOVER_EXIT);
            }
        }

        if ((action == MotionEvent.ACTION_HOVER_ENTER || action == MotionEvent.ACTION_HOVER_MOVE)
                && event.isFromSource(InputDevice.SOURCE_MOUSE)
                && isOnScrollbar(event.getX(), event.getY())) {
            awakenScrollBars();
        }

        // If we consider ourself hoverable, or if we we're already hovered,
        // handle changing state in response to ENTER and EXIT events.
        if (isHoverable() || isHovered()) {
            switch (action) {
                case MotionEvent.ACTION_HOVER_ENTER:
                    setHovered(true);
                    break;
                case MotionEvent.ACTION_HOVER_EXIT:
                    setHovered(false);
                    break;
            }

            // Dispatch the event to onGenericMotionEvent before returning true.
            // This is to provide compatibility with existing applications that
            // handled HOVER_MOVE events in onGenericMotionEvent and that would
            // break because of the new default handling for hoverable views
            // in onHoverEvent.
            // Note that onGenericMotionEvent will be called by default when
            // onHoverEvent returns false (refer to dispatchGenericMotionEvent).
            dispatchGenericMotionEventInternal(event);
            // The event was already handled by calling setHovered(), so always
            // return true.
            return true;
        }

        return false;
    }

    /**
     * Returns true if the view should handle {@link #onHoverEvent}
     * by calling {@link #setHovered} to change its hovered state.
     *
     * @return True if the view is hoverable.
     */
    private boolean isHoverable() {
        final int viewFlags = mViewFlags;
        if ((viewFlags & ENABLED_MASK) == DISABLED) {
            return false;
        }

        return (viewFlags & CLICKABLE) == CLICKABLE
                || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE
                || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;
    }

    /**
```

```
     * Returns true if the view is currently hovered.
     *
     * @return True if the view is currently hovered.
     *
     * @see #setHovered
     * @see #onHoverChanged
     */
    @ViewDebug.ExportedProperty
    public boolean isHovered() {
        return (mPrivateFlags & PFLAG_HOVERED) != 0;
    }

    /**
     * Sets whether the view is currently hovered.
     * <p>
     * Calling this method also changes the drawable state of the view.  This
     * enables the view to react to hover by using different drawable resources
     * to change its appearance.
     * </p><p>
     * The {@link #onHoverChanged} method is called when the hovered state changes.
     * </p>
     *
     * @param hovered True if the view is hovered.
     *
     * @see #isHovered
     * @see #onHoverChanged
     */
    public void setHovered(boolean hovered) {
        if (hovered) {
            if ((mPrivateFlags & PFLAG_HOVERED) == 0) {
                mPrivateFlags |= PFLAG_HOVERED;
                refreshDrawableState();
                onHoverChanged(true);
            }
        } else {
            if ((mPrivateFlags & PFLAG_HOVERED) != 0) {
                mPrivateFlags &= ~PFLAG_HOVERED;
                refreshDrawableState();
                onHoverChanged(false);
            }
        }
    }

    /**
     * Implement this method to handle hover state changes.
     * <p>
     * This method is called whenever the hover state changes as a result of a
     * call to {@link #setHovered}.
     * </p>
     *
     * @param hovered The current hover state, as returned by {@link #isHovered}.
     *
     * @see #isHovered
     * @see #setHovered
     */
    public void onHoverChanged(boolean hovered) {
    }

    /**
     * Handles scroll bar dragging by mouse input.
     *
     * @hide
     * @param event The motion event.
     *
     * @return true if the event was handled as a scroll bar dragging, false otherwise.
     */
    protected boolean handleScrollBarDragging(MotionEvent event) {
        if (mScrollCache == null) {
            return false;
        }
        final float x = event.getX();
        final float y = event.getY();
        final int action = event.getAction();
        if ((mScrollCache.mScrollBarDraggingState == ScrollabilityCache.NOT_DRAGGING
                && action != MotionEvent.ACTION_DOWN)
                    || !event.isFromSource(InputDevice.SOURCE_MOUSE)
                    || !event.isButtonPressed(MotionEvent.BUTTON_PRIMARY)) {
            mScrollCache.mScrollBarDraggingState = ScrollabilityCache.NOT_DRAGGING;
            return false;
        }

        switch (action) {
```

```
        case MotionEvent.ACTION_MOVE:
            if (mScrollCache.mScrollBarDraggingState == ScrollabilityCache.NOT_DRAGGING) {
                return false;
            }
            if (mScrollCache.mScrollBarDraggingState
                    == ScrollabilityCache.DRAGGING_VERTICAL_SCROLL_BAR) {
                final Rect bounds = mScrollCache.mScrollBarBounds;
                getVerticalScrollBarBounds(bounds, null);
                final int range = computeVerticalScrollRange();
                final int offset = computeVerticalScrollOffset();
                final int extent = computeVerticalScrollExtent();

                final int thumbLength = ScrollBarUtils.getThumbLength(
                        bounds.height(), bounds.width(), extent, range);
                final int thumbOffset = ScrollBarUtils.getThumbOffset(
                        bounds.height(), thumbLength, extent, range, offset);

                final float diff = y - mScrollCache.mScrollBarDraggingPos;
                final float maxThumbOffset = bounds.height() - thumbLength;
                final float newThumbOffset =
                        Math.min(Math.max(thumbOffset + diff, 0.0f), maxThumbOffset);
                final int height = getHeight();
                if (Math.round(newThumbOffset) != thumbOffset && maxThumbOffset > 0
                        && height > 0 && extent > 0) {
                    final int newY = Math.round((range - extent)
                            / ((float)extent / height) * (newThumbOffset / maxThumbOffset));
                    if (newY != getScrollY()) {
                        mScrollCache.mScrollBarDraggingPos = y;
                        setScrollY(newY);
                    }
                }
                return true;
            }
            if (mScrollCache.mScrollBarDraggingState
                    == ScrollabilityCache.DRAGGING_HORIZONTAL_SCROLL_BAR) {
                final Rect bounds = mScrollCache.mScrollBarBounds;
                getHorizontalScrollBarBounds(bounds, null);
                final int range = computeHorizontalScrollRange();
                final int offset = computeHorizontalScrollOffset();
                final int extent = computeHorizontalScrollExtent();

                final int thumbLength = ScrollBarUtils.getThumbLength(
                        bounds.width(), bounds.height(), extent, range);
                final int thumbOffset = ScrollBarUtils.getThumbOffset(
                        bounds.width(), thumbLength, extent, range, offset);

                final float diff = x - mScrollCache.mScrollBarDraggingPos;
                final float maxThumbOffset = bounds.width() - thumbLength;
                final float newThumbOffset =
                        Math.min(Math.max(thumbOffset + diff, 0.0f), maxThumbOffset);
                final int width = getWidth();
                if (Math.round(newThumbOffset) != thumbOffset && maxThumbOffset > 0
                        && width > 0 && extent > 0) {
                    final int newX = Math.round((range - extent)
                            / ((float)extent / width) * (newThumbOffset / maxThumbOffset));
                    if (newX != getScrollX()) {
                        mScrollCache.mScrollBarDraggingPos = x;
                        setScrollX(newX);
                    }
                }
                return true;
            }
        case MotionEvent.ACTION_DOWN:
            if (mScrollCache.state == ScrollabilityCache.OFF) {
                return false;
            }
            if (isOnVerticalScrollbarThumb(x, y)) {
                mScrollCache.mScrollBarDraggingState =
                        ScrollabilityCache.DRAGGING_VERTICAL_SCROLL_BAR;
                mScrollCache.mScrollBarDraggingPos = y;
                return true;
            }
            if (isOnHorizontalScrollbarThumb(x, y)) {
                mScrollCache.mScrollBarDraggingState =
                        ScrollabilityCache.DRAGGING_HORIZONTAL_SCROLL_BAR;
                mScrollCache.mScrollBarDraggingPos = x;
                return true;
            }
        }
        mScrollCache.mScrollBarDraggingState = ScrollabilityCache.NOT_DRAGGING;
        return false;
    }
```

```java
/**
 * Implement this method to handle touch screen motion events.
 * <p>
 * If this method is used to detect click actions, it is recommended that
 * the actions be performed by implementing and calling
 * {@link #performClick()}. This will ensure consistent system behavior,
 * including:
 * <ul>
 * <li>obeying click sound preferences
 * <li>dispatching OnClickListener calls
 * <li>handling {@link AccessibilityNodeInfo#ACTION_CLICK ACTION_CLICK} when
 * accessibility features are enabled
 * </ul>
 *
 * @param event The motion event.
 * @return True if the event was handled, false otherwise.
 */
public boolean onTouchEvent(MotionEvent event) {
    final float x = event.getX();
    final float y = event.getY();
    final int viewFlags = mViewFlags;
    final int action = event.getAction();

    final boolean clickable = ((viewFlags & CLICKABLE) == CLICKABLE
            || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)
            || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;

    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (action == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        mPrivateFlags3 &= ~PFLAG3_FINGER_DOWN;
        // A disabled view that is clickable still consumes the touch
        // events, it just doesn't respond to them.
        return clickable;
    }
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }

    if (clickable || (viewFlags & TOOLTIP) == TOOLTIP) {
        switch (action) {
            case MotionEvent.ACTION_UP:
                mPrivateFlags3 &= ~PFLAG3_FINGER_DOWN;
                if ((viewFlags & TOOLTIP) == TOOLTIP) {
                    handleTooltipUp();
                }
                if (!clickable) {
                    removeTapCallback();
                    removeLongPressCallback();
                    mInContextButtonPress = false;
                    mHasPerformedLongPress = false;
                    mIgnoreNextUpEvent = false;
                    break;
                }
                boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
                if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                    // take focus if we don't have it already and we should in
                    // touch mode.
                    boolean focusTaken = false;
                    if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
                        focusTaken = requestFocus();
                    }

                    if (prepressed) {
                        // The button is being released before we actually
                        // showed it as pressed.  Make it show the pressed
                        // state now (before scheduling the click) to ensure
                        // the user sees it.
                        setPressed(true, x, y);
                    }

                    if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {
                        // This is a tap, so remove the longpress check
                        removeLongPressCallback();

                        // Only perform take click actions if we were in the pressed state
                        if (!focusTaken) {
                            // Use a Runnable and post this rather than calling
```

```
                // performClick directly. This lets other visual state
                // of the view update before click actions start.
                if (mPerformClick == null) {
                    mPerformClick = new PerformClick();
                }
                if (!post(mPerformClick)) {
                    performClick();
                }
            }
        }

        if (mUnsetPressedState == null) {
            mUnsetPressedState = new UnsetPressedState();
        }

        if (prepressed) {
            postDelayed(mUnsetPressedState,
                    ViewConfiguration.getPressedStateDuration());
        } else if (!post(mUnsetPressedState)) {
            // If the post failed, unpress right now
            mUnsetPressedState.run();
        }

        removeTapCallback();
    }
    mIgnoreNextUpEvent = false;
    break;

case MotionEvent.ACTION_DOWN:
    if (event.getSource() == InputDevice.SOURCE_TOUCHSCREEN) {
        mPrivateFlags3 |= PFLAG3_FINGER_DOWN;
    }
    mHasPerformedLongPress = false;

    if (!clickable) {
        checkForLongClick(0, x, y);
        break;
    }

    if (performButtonActionOnTouchDown(event)) {
        break;
    }

    // Walk up the hierarchy to determine if we're inside a scrolling container.
    boolean isInScrollingContainer = isInScrollingContainer();

    // For views inside a scrolling container, delay the pressed feedback for
    // a short period in case this is a scroll.
    if (isInScrollingContainer) {
        mPrivateFlags |= PFLAG_PREPRESSED;
        if (mPendingCheckForTap == null) {
            mPendingCheckForTap = new CheckForTap();
        }
        mPendingCheckForTap.x = event.getX();
        mPendingCheckForTap.y = event.getY();
        postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
    } else {
        // Not inside a scrolling container, so show the feedback right away
        setPressed(true, x, y);
        checkForLongClick(0, x, y);
    }
    break;

case MotionEvent.ACTION_CANCEL:
    if (clickable) {
        setPressed(false);
    }
    removeTapCallback();
    removeLongPressCallback();
    mInContextButtonPress = false;
    mHasPerformedLongPress = false;
    mIgnoreNextUpEvent = false;
    mPrivateFlags3 &= ~PFLAG3_FINGER_DOWN;
    break;

case MotionEvent.ACTION_MOVE:
    if (clickable) {
        drawableHotspotChanged(x, y);
    }

    // Be lenient about moving outside of buttons
    if (!pointInView(x, y, mTouchSlop)) {
```

```
                    // Outside button
                    // Remove any future long press/tap checks
                    removeTapCallback();
                    removeLongPressCallback();
                    if ((mPrivateFlags & PFLAG_PRESSED) != 0) {
                        setPressed(false);
                    }
                    mPrivateFlags3 &= ~PFLAG3_FINGER_DOWN;
                }
                break;
        }

        return true;
    }

    return false;
}

/**
 * @hide
 */
public boolean isInScrollingContainer() {
    ViewParent p = getParent();
    while (p != null && p instanceof ViewGroup) {
        if (((ViewGroup) p).shouldDelayChildPressedState()) {
            return true;
        }
        p = p.getParent();
    }
    return false;
}

/**
 * Remove the longpress detection timer.
 */
private void removeLongPressCallback() {
    if (mPendingCheckForLongPress != null) {
        removeCallbacks(mPendingCheckForLongPress);
    }
}

/**
 * Remove the pending click action
 */
private void removePerformClickCallback() {
    if (mPerformClick != null) {
        removeCallbacks(mPerformClick);
    }
}

/**
 * Remove the prepress detection timer.
 */
private void removeUnsetPressCallback() {
    if ((mPrivateFlags & PFLAG_PRESSED) != 0 && mUnsetPressedState != null) {
        setPressed(false);
        removeCallbacks(mUnsetPressedState);
    }
}

/**
 * Remove the tap detection timer.
 */
private void removeTapCallback() {
    if (mPendingCheckForTap != null) {
        mPrivateFlags &= ~PFLAG_PREPRESSED;
        removeCallbacks(mPendingCheckForTap);
    }
}

/**
 * Cancels a pending long press.  Your subclass can use this if you
 * want the context menu to come up if the user presses and holds
 * at the same place, but you don't want it to come up if they press
 * and then move around enough to cause scrolling.
 */
public void cancelLongPress() {
    removeLongPressCallback();

    /*
     * The prepressed state handled by the tap callback is a display
     * construct, but the tap callback will post a long press callback
```

```
     * less its own timeout. Remove it here.
     */
    removeTapCallback();
}

/**
 * Remove the pending callback for sending a
 * {@link AccessibilityEvent#TYPE_VIEW_SCROLLED} accessibility event.
 */
private void removeSendViewScrolledAccessibilityEventCallback() {
    if (mSendViewScrolledAccessibilityEvent != null) {
        removeCallbacks(mSendViewScrolledAccessibilityEvent);
        mSendViewScrolledAccessibilityEvent.mIsPending = false;
    }
}

/**
 * Sets the TouchDelegate for this View.
 */
public void setTouchDelegate(TouchDelegate delegate) {
    mTouchDelegate = delegate;
}

/**
 * Gets the TouchDelegate for this View.
 */
public TouchDelegate getTouchDelegate() {
    return mTouchDelegate;
}

/**
 * Request unbuffered dispatch of the given stream of MotionEvents to this View.
 *
 * Until this View receives a corresponding {@link MotionEvent#ACTION_UP}, ask that the input
 * system not batch {@link MotionEvent}s but instead deliver them as soon as they're
 * available. This method should only be called for touch events.
 *
 * <p class="note">This api is not intended for most applications. Buffered dispatch
 * provides many of benefits, and just requesting unbuffered dispatch on most MotionEvent
 * streams will not improve your input latency. Side effects include: increased latency,
 * jittery scrolls and inability to take advantage of system resampling. Talk to your input
 * professional to see if {@link #requestUnbufferedDispatch(MotionEvent)} is right for
 * you.</p>
 */
public final void requestUnbufferedDispatch(MotionEvent event) {
    final int action = event.getAction();
    if (mAttachInfo == null
            || action != MotionEvent.ACTION_DOWN && action != MotionEvent.ACTION_MOVE
            || !event.isTouchEvent()) {
        return;
    }
    mAttachInfo.mUnbufferedDispatchRequested = true;
}

/**
 * Set flags controlling behavior of this view.
 *
 * @param flags Constant indicating the value which should be set
 * @param mask Constant indicating the bit range that should be changed
 */
void setFlags(int flags, int mask) {
    final boolean accessibilityEnabled =
            AccessibilityManager.getInstance(mContext).isEnabled();
    final boolean oldIncludeForAccessibility = accessibilityEnabled && includeForAccessibility();

    int old = mViewFlags;
    mViewFlags = (mViewFlags & ~mask) | (flags & mask);

    int changed = mViewFlags ^ old;
    if (changed == 0) {
        return;
    }
    int privateFlags = mPrivateFlags;

    // If focusable is auto, update the FOCUSABLE bit.
    int focusableChangedByAuto = 0;
    if (((mViewFlags & FOCUSABLE_AUTO) != 0)
            && (changed & (FOCUSABLE_MASK | CLICKABLE)) != 0) {
        // Heuristic only takes into account whether view is clickable.
        final int newFocus;
        if ((mViewFlags & CLICKABLE) != 0) {
            newFocus = FOCUSABLE;
```

```
    } else {
        newFocus = NOT_FOCUSABLE;
    }
    mViewFlags = (mViewFlags & ~FOCUSABLE) | newFocus;
    focusableChangedByAuto = (old & FOCUSABLE) ^ (newFocus & FOCUSABLE);
    changed = (changed & ~FOCUSABLE) | focusableChangedByAuto;
}

/* Check if the FOCUSABLE bit has changed */
if (((changed & FOCUSABLE) != 0) && ((privateFlags & PFLAG_HAS_BOUNDS) != 0)) {
    if (((old & FOCUSABLE) == FOCUSABLE)
            && ((privateFlags & PFLAG_FOCUSED) != 0)) {
        /* Give up focus if we are no longer focusable */
        clearFocus();
        if (mParent instanceof ViewGroup) {
            ((ViewGroup) mParent).clearFocusedInCluster();
        }
    } else if (((old & FOCUSABLE) == NOT_FOCUSABLE)
            && ((privateFlags & PFLAG_FOCUSED) == 0)) {
        /*
         * Tell the view system that we are now available to take focus
         * if no one else already has it.
         */
        if (mParent != null) {
            ViewRootImpl viewRootImpl = getViewRootImpl();
            if (!sAutoFocusableOffUIThreadWontNotifyParents
                    || focusableChangedByAuto == 0
                    || viewRootImpl == null
                    || viewRootImpl.mThread == Thread.currentThread()) {
                mParent.focusableViewAvailable(this);
            }
        }
    }
}

final int newVisibility = flags & VISIBILITY_MASK;
if (newVisibility == VISIBLE) {
    if ((changed & VISIBILITY_MASK) != 0) {
        /*
         * If this view is becoming visible, invalidate it in case it changed while
         * it was not visible. Marking it drawn ensures that the invalidation will
         * go through.
         */
        mPrivateFlags |= PFLAG_DRAWN;
        invalidate(true);

        needGlobalAttributesUpdate(true);

        // a view becoming visible is worth notifying the parent
        // about in case nothing has focus.  even if this specific view
        // isn't focusable, it may contain something that is, so let
        // the root view try to give this focus if nothing else does.
        if ((mParent != null) && (mBottom > mTop) && (mRight > mLeft)) {
            mParent.focusableViewAvailable(this);
        }
    }
}

/* Check if the GONE bit has changed */
if ((changed & GONE) != 0) {
    needGlobalAttributesUpdate(false);
    requestLayout();

    if (((mViewFlags & VISIBILITY_MASK) == GONE)) {
        if (hasFocus()) {
            clearFocus();
            if (mParent instanceof ViewGroup) {
                ((ViewGroup) mParent).clearFocusedInCluster();
            }
        }
        clearAccessibilityFocus();
        destroyDrawingCache();
        if (mParent instanceof View) {
            // GONE views noop invalidation, so invalidate the parent
            ((View) mParent).invalidate(true);
        }
        // Mark the view drawn to ensure that it gets invalidated properly the next
        // time it is visible and gets invalidated
        mPrivateFlags |= PFLAG_DRAWN;
    }
    if (mAttachInfo != null) {
        mAttachInfo.mViewVisibilityChanged = true;
```

```
    }
}

/* Check if the VISIBLE bit has changed */
if ((changed & INVISIBLE) != 0) {
    needGlobalAttributesUpdate(false);
    /*
     * If this view is becoming invisible, set the DRAWN flag so that
     * the next invalidate() will not be skipped.
     */
    mPrivateFlags |= PFLAG_DRAWN;

    if (((mViewFlags & VISIBILITY_MASK) == INVISIBLE)) {
        // root view becoming invisible shouldn't clear focus and accessibility focus
        if (getRootView() != this) {
            if (hasFocus()) {
                clearFocus();
                if (mParent instanceof ViewGroup) {
                    ((ViewGroup) mParent).clearFocusedInCluster();
                }
            }
            clearAccessibilityFocus();
        }
    }
    if (mAttachInfo != null) {
        mAttachInfo.mViewVisibilityChanged = true;
    }
}

if ((changed & VISIBILITY_MASK) != 0) {
    // If the view is invisible, cleanup its display list to free up resources
    if (newVisibility != VISIBLE && mAttachInfo != null) {
        cleanupDraw();
    }

    if (mParent instanceof ViewGroup) {
        ((ViewGroup) mParent).onChildVisibilityChanged(this,
                (changed & VISIBILITY_MASK), newVisibility);
        ((View) mParent).invalidate(true);
    } else if (mParent != null) {
        mParent.invalidateChild(this, null);
    }

    if (mAttachInfo != null) {
        dispatchVisibilityChanged(this, newVisibility);

        // Aggregated visibility changes are dispatched to attached views
        // in visible windows where the parent is currently shown/drawn
        // or the parent is not a ViewGroup (and therefore assumed to be a ViewRoot),
        // discounting clipping or overlapping. This makes it a good place
        // to change animation states.
        if (mParent != null && getWindowVisibility() == VISIBLE &&
                ((!(mParent instanceof ViewGroup)) || ((ViewGroup) mParent).isShown())) {
            dispatchVisibilityAggregated(newVisibility == VISIBLE);
        }
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
}

if ((changed & WILL_NOT_CACHE_DRAWING) != 0) {
    destroyDrawingCache();
}

if ((changed & DRAWING_CACHE_ENABLED) != 0) {
    destroyDrawingCache();
    mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
    invalidateParentCaches();
}

if ((changed & DRAWING_CACHE_QUALITY_MASK) != 0) {
    destroyDrawingCache();
    mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
}

if ((changed & DRAW_MASK) != 0) {
    if ((mViewFlags & WILL_NOT_DRAW) != 0) {
        if (mBackground != null
                || mDefaultFocusHighlight != null
                || (mForegroundInfo != null && mForegroundInfo.mDrawable != null)) {
            mPrivateFlags &= ~PFLAG_SKIP_DRAW;
        } else {
            mPrivateFlags |= PFLAG_SKIP_DRAW;
```

```
            }
        } else {
            mPrivateFlags &= ~PFLAG_SKIP_DRAW;
        }
        requestLayout();
        invalidate(true);
    }

    if ((changed & KEEP_SCREEN_ON) != 0) {
        if (mParent != null && mAttachInfo != null && !mAttachInfo.mRecomputeGlobalAttributes) {
            mParent.recomputeViewAttributes(this);
        }
    }

    if (accessibilityEnabled) {
        if ((changed & FOCUSABLE) != 0 || (changed & VISIBILITY_MASK) != 0
                || (changed & CLICKABLE) != 0 || (changed & LONG_CLICKABLE) != 0
                || (changed & CONTEXT_CLICKABLE) != 0) {
            if (oldIncludeForAccessibility != includeForAccessibility()) {
                notifySubtreeAccessibilityStateChangedIfNeeded();
            } else {
                notifyViewAccessibilityStateChangedIfNeeded(
                        AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
            }
        } else if ((changed & ENABLED_MASK) != 0) {
            notifyViewAccessibilityStateChangedIfNeeded(
                    AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
        }
    }
}

/**
 * Change the view's z order in the tree, so it's on top of other sibling
 * views. This ordering change may affect layout, if the parent container
 * uses an order-dependent layout scheme (e.g., LinearLayout). Prior
 * to {@link android.os.Build.VERSION_CODES#KITKAT} this
 * method should be followed by calls to {@link #requestLayout()} and
 * {@link View#invalidate()} on the view's parent to force the parent to redraw
 * with the new child ordering.
 *
 * @see ViewGroup#bringChildToFront(View)
 */
public void bringToFront() {
    if (mParent != null) {
        mParent.bringChildToFront(this);
    }
}

/**
 * This is called in response to an internal scroll in this view (i.e., the
 * view scrolled its own contents). This is typically as a result of
 * {@link #scrollBy(int, int)} or {@link #scrollTo(int, int)} having been
 * called.
 *
 * @param l Current horizontal scroll origin.
 * @param t Current vertical scroll origin.
 * @param oldl Previous horizontal scroll origin.
 * @param oldt Previous vertical scroll origin.
 */
protected void onScrollChanged(int l, int t, int oldl, int oldt) {
    notifySubtreeAccessibilityStateChangedIfNeeded();

    if (AccessibilityManager.getInstance(mContext).isEnabled()) {
        postSendViewScrolledAccessibilityEventCallback();
    }

    mBackgroundSizeChanged = true;
    mDefaultFocusHighlightSizeChanged = true;
    if (mForegroundInfo != null) {
        mForegroundInfo.mBoundsChanged = true;
    }

    final AttachInfo ai = mAttachInfo;
    if (ai != null) {
        ai.mViewScrollChanged = true;
    }

    if (mListenerInfo != null && mListenerInfo.mOnScrollChangeListener != null) {
        mListenerInfo.mOnScrollChangeListener.onScrollChange(this, l, t, oldl, oldt);
    }
}
```

```java
/**
 * Interface definition for a callback to be invoked when the scroll
 * X or Y positions of a view change.
 * <p>
 * <b>Note:</b> Some views handle scrolling independently from View and may
 * have their own separate listeners for scroll-type events. For example,
 * {@link android.widget.ListView ListView} allows clients to register an
 * {@link android.widget.ListView#setOnScrollListener(android.widget.AbsListView.OnScrollListener) AbsListView.OnScrol
 * to listen for changes in list scroll position.
 *
 * @see #setOnScrollChangeListener(View.OnScrollChangeListener)
 */
public interface OnScrollChangeListener {
    /**
     * Called when the scroll position of a view changes.
     *
     * @param v The view whose scroll position has changed.
     * @param scrollX Current horizontal scroll origin.
     * @param scrollY Current vertical scroll origin.
     * @param oldScrollX Previous horizontal scroll origin.
     * @param oldScrollY Previous vertical scroll origin.
     */
    void onScrollChange(View v, int scrollX, int scrollY, int oldScrollX, int oldScrollY);
}

/**
 * Interface definition for a callback to be invoked when the layout bounds of a view
 * changes due to layout processing.
 */
public interface OnLayoutChangeListener {
    /**
     * Called when the layout bounds of a view changes due to layout processing.
     *
     * @param v The view whose bounds have changed.
     * @param left The new value of the view's left property.
     * @param top The new value of the view's top property.
     * @param right The new value of the view's right property.
     * @param bottom The new value of the view's bottom property.
     * @param oldLeft The previous value of the view's left property.
     * @param oldTop The previous value of the view's top property.
     * @param oldRight The previous value of the view's right property.
     * @param oldBottom The previous value of the view's bottom property.
     */
    void onLayoutChange(View v, int left, int top, int right, int bottom,
        int oldLeft, int oldTop, int oldRight, int oldBottom);
}

/**
 * This is called during layout when the size of this view has changed. If
 * you were just added to the view hierarchy, you're called with the old
 * values of 0.
 *
 * @param w Current width of this view.
 * @param h Current height of this view.
 * @param oldw Old width of this view.
 * @param oldh Old height of this view.
 */
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
}

/**
 * Called by draw to draw the child views. This may be overridden
 * by derived classes to gain control just before its children are drawn
 * (but after its own view has been drawn).
 * @param canvas the canvas on which to draw the view
 */
protected void dispatchDraw(Canvas canvas) {

}

/**
 * Gets the parent of this view. Note that the parent is a
 * ViewParent and not necessarily a View.
 *
 * @return Parent of this view.
 */
public final ViewParent getParent() {
    return mParent;
}

/**
 * Set the horizontal scrolled position of your view. This will cause a call to
```

```
     * {@link #onScrollChanged(int, int, int, int)} and the view will be
     * invalidated.
     * @param value the x position to scroll to
     */
    public void setScrollX(int value) {
        scrollTo(value, mScrollY);
    }

    /**
     * Set the vertical scrolled position of your view. This will cause a call to
     * {@link #onScrollChanged(int, int, int, int)} and the view will be
     * invalidated.
     * @param value the y position to scroll to
     */
    public void setScrollY(int value) {
        scrollTo(mScrollX, value);
    }

    /**
     * Return the scrolled left position of this view. This is the left edge of
     * the displayed part of your view. You do not need to draw any pixels
     * farther left, since those are outside of the frame of your view on
     * screen.
     *
     * @return The left edge of the displayed part of your view, in pixels.
     */
    public final int getScrollX() {
        return mScrollX;
    }

    /**
     * Return the scrolled top position of this view. This is the top edge of
     * the displayed part of your view. You do not need to draw any pixels above
     * it, since those are outside of the frame of your view on screen.
     *
     * @return The top edge of the displayed part of your view, in pixels.
     */
    public final int getScrollY() {
        return mScrollY;
    }

    /**
     * Return the width of the your view.
     *
     * @return The width of your view, in pixels.
     */
    @ViewDebug.ExportedProperty(category = "layout")
    public final int getWidth() {
        return mRight - mLeft;
    }

    /**
     * Return the height of your view.
     *
     * @return The height of your view, in pixels.
     */
    @ViewDebug.ExportedProperty(category = "layout")
    public final int getHeight() {
        return mBottom - mTop;
    }

    /**
     * Return the visible drawing bounds of your view. Fills in the output
     * rectangle with the values from getScrollX(), getScrollY(),
     * getWidth(), and getHeight(). These bounds do not account for any
     * transformation properties currently set on the view, such as
     * {@link #setScaleX(float)} or {@link #setRotation(float)}.
     *
     * @param outRect The (scrolled) drawing bounds of the view.
     */
    public void getDrawingRect(Rect outRect) {
        outRect.left = mScrollX;
        outRect.top = mScrollY;
        outRect.right = mScrollX + (mRight - mLeft);
        outRect.bottom = mScrollY + (mBottom - mTop);
    }

    /**
     * Like {@link #getMeasuredWidthAndState()}, but only returns the
     * raw width component (that is the result is masked by
     * {@link #MEASURED_SIZE_MASK}).
     *
```

```
     * @return The raw measured width of this view.
     */
    public final int getMeasuredWidth() {
        return mMeasuredWidth & MEASURED_SIZE_MASK;
    }

    /**
     * Return the full width measurement information for this view as computed
     * by the most recent call to {@link #measure(int, int)}.  This result is a bit mask
     * as defined by {@link #MEASURED_SIZE_MASK} and {@link #MEASURED_STATE_TOO_SMALL}.
     * This should be used during measurement and layout calculations only. Use
     * {@link #getWidth()} to see how wide a view is after layout.
     *
     * @return The measured width of this view as a bit mask.
     */
    @ViewDebug.ExportedProperty(category = "measurement", flagMapping = {
            @ViewDebug.FlagToString(mask = MEASURED_STATE_MASK, equals = MEASURED_STATE_TOO_SMALL,
                    name = "MEASURED_STATE_TOO_SMALL"),
    })
    public final int getMeasuredWidthAndState() {
        return mMeasuredWidth;
    }

    /**
     * Like {@link #getMeasuredHeightAndState()}, but only returns the
     * raw height component (that is the result is masked by
     * {@link #MEASURED_SIZE_MASK}).
     *
     * @return The raw measured height of this view.
     */
    public final int getMeasuredHeight() {
        return mMeasuredHeight & MEASURED_SIZE_MASK;
    }

    /**
     * Return the full height measurement information for this view as computed
     * by the most recent call to {@link #measure(int, int)}.  This result is a bit mask
     * as defined by {@link #MEASURED_SIZE_MASK} and {@link #MEASURED_STATE_TOO_SMALL}.
     * This should be used during measurement and layout calculations only. Use
     * {@link #getHeight()} to see how wide a view is after layout.
     *
     * @return The measured height of this view as a bit mask.
     */
    @ViewDebug.ExportedProperty(category = "measurement", flagMapping = {
            @ViewDebug.FlagToString(mask = MEASURED_STATE_MASK, equals = MEASURED_STATE_TOO_SMALL,
                    name = "MEASURED_STATE_TOO_SMALL"),
    })
    public final int getMeasuredHeightAndState() {
        return mMeasuredHeight;
    }

    /**
     * Return only the state bits of {@link #getMeasuredWidthAndState()}
     * and {@link #getMeasuredHeightAndState()}, combined into one integer.
     * The width component is in the regular bits {@link #MEASURED_STATE_MASK}
     * and the height component is at the shifted bits
     * {@link #MEASURED_HEIGHT_STATE_SHIFT}>>{@link #MEASURED_STATE_MASK}.
     */
    public final int getMeasuredState() {
        return (mMeasuredWidth&MEASURED_STATE_MASK)
                | ((mMeasuredHeight>>MEASURED_HEIGHT_STATE_SHIFT)
                        & (MEASURED_STATE_MASK>>MEASURED_HEIGHT_STATE_SHIFT));
    }

    /**
     * The transform matrix of this view, which is calculated based on the current
     * rotation, scale, and pivot properties.
     *
     * @see #getRotation()
     * @see #getScaleX()
     * @see #getScaleY()
     * @see #getPivotX()
     * @see #getPivotY()
     * @return The current transform matrix for the view
     */
    public Matrix getMatrix() {
        ensureTransformationInfo();
        final Matrix matrix = mTransformationInfo.mMatrix;
        mRenderNode.getMatrix(matrix);
        return matrix;
    }
```

```
/**
 * Returns true if the transform matrix is the identity matrix.
 * Recomputes the matrix if necessary.
 *
 * @return True if the transform matrix is the identity matrix, false otherwise.
 */
final boolean hasIdentityMatrix() {
    return mRenderNode.hasIdentityMatrix();
}

void ensureTransformationInfo() {
    if (mTransformationInfo == null) {
        mTransformationInfo = new TransformationInfo();
    }
}

/**
 * Utility method to retrieve the inverse of the current mMatrix property.
 * We cache the matrix to avoid recalculating it when transform properties
 * have not changed.
 *
 * @return The inverse of the current matrix of this view.
 * @hide
 */
public final Matrix getInverseMatrix() {
    ensureTransformationInfo();
    if (mTransformationInfo.mInverseMatrix == null) {
        mTransformationInfo.mInverseMatrix = new Matrix();
    }
    final Matrix matrix = mTransformationInfo.mInverseMatrix;
    mRenderNode.getInverseMatrix(matrix);
    return matrix;
}

/**
 * Gets the distance along the Z axis from the camera to this view.
 *
 * @see #setCameraDistance(float)
 *
 * @return The distance along the Z axis.
 */
public float getCameraDistance() {
    final float dpi = mResources.getDisplayMetrics().densityDpi;
    return -(mRenderNode.getCameraDistance() * dpi);
}

/**
 * <p>Sets the distance along the Z axis (orthogonal to the X/Y plane on which
 * views are drawn) from the camera to this view. The camera's distance
 * affects 3D transformations, for instance rotations around the X and Y
 * axis. If the rotationX or rotationY properties are changed and this view is
 * large (more than half the size of the screen), it is recommended to always
 * use a camera distance that's greater than the height (X axis rotation) or
 * the width (Y axis rotation) of this view.</p>
 *
 * <p>The distance of the camera from the view plane can have an affect on the
 * perspective distortion of the view when it is rotated around the x or y axis.
 * For example, a large distance will result in a large viewing angle, and there
 * will not be much perspective distortion of the view as it rotates. A short
 * distance may cause much more perspective distortion upon rotation, and can
 * also result in some drawing artifacts if the rotated view ends up partially
 * behind the camera (which is why the recommendation is to use a distance at
 * least as far as the size of the view, if the view is to be rotated.)</p>
 *
 * <p>The distance is expressed in "depth pixels." The default distance depends
 * on the screen density. For instance, on a medium density display, the
 * default distance is 1280. On a high density display, the default distance
 * is 1920.</p>
 *
 * <p>If you want to specify a distance that leads to visually consistent
 * results across various densities, use the following formula:</p>
 * <pre>
 * float scale = context.getResources().getDisplayMetrics().density;
 * view.setCameraDistance(distance * scale);
 * </pre>
 *
 * <p>The density scale factor of a high density display is 1.5,
 * and 1920 = 1280 * 1.5.</p>
 *
 * @param distance The distance in "depth pixels", if negative the opposite
 *        value is used
 *
```

```
     * @see #setRotationX(float)
     * @see #setRotationY(float)
     */
    public void setCameraDistance(float distance) {
        final float dpi = mResources.getDisplayMetrics().densityDpi;

        invalidateViewProperty(true, false);
        mRenderNode.setCameraDistance(-Math.abs(distance) / dpi);
        invalidateViewProperty(false, false);

        invalidateParentIfNeededAndWasQuickRejected();
    }

    /**
     * The degrees that the view is rotated around the pivot point.
     *
     * @see #setRotation(float)
     * @see #getPivotX()
     * @see #getPivotY()
     *
     * @return The degrees of rotation.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getRotation() {
        return mRenderNode.getRotation();
    }

    /**
     * Sets the degrees that the view is rotated around the pivot point. Increasing values
     * result in clockwise rotation.
     *
     * @param rotation The degrees of rotation.
     *
     * @see #getRotation()
     * @see #getPivotX()
     * @see #getPivotY()
     * @see #setRotationX(float)
     * @see #setRotationY(float)
     *
     * @attr ref android.R.styleable#View_rotation
     */
    public void setRotation(float rotation) {
        if (rotation != getRotation()) {
            // Double-invalidation is necessary to capture view's old and new areas
            invalidateViewProperty(true, false);
            mRenderNode.setRotation(rotation);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The degrees that the view is rotated around the vertical axis through the pivot point.
     *
     * @see #getPivotX()
     * @see #getPivotY()
     * @see #setRotationY(float)
     *
     * @return The degrees of Y rotation.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getRotationY() {
        return mRenderNode.getRotationY();
    }

    /**
     * Sets the degrees that the view is rotated around the vertical axis through the pivot point.
     * Increasing values result in counter-clockwise rotation from the viewpoint of looking
     * down the y axis.
     *
     * When rotating large views, it is recommended to adjust the camera distance
     * accordingly. Refer to {@link #setCameraDistance(float)} for more information.
     *
     * @param rotationY The degrees of Y rotation.
     *
     * @see #getRotationY()
     * @see #getPivotX()
     * @see #getPivotY()
     * @see #setRotation(float)
     * @see #setRotationX(float)
```

```
     * @see #setCameraDistance(float)
     *
     * @attr ref android.R.styleable#View_rotationY
     */
    public void setRotationY(float rotationY) {
        if (rotationY != getRotationY()) {
            invalidateViewProperty(true, false);
            mRenderNode.setRotationY(rotationY);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The degrees that the view is rotated around the horizontal axis through the pivot point.
     *
     * @see #getPivotX()
     * @see #getPivotY()
     * @see #setRotationX(float)
     *
     * @return The degrees of X rotation.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getRotationX() {
        return mRenderNode.getRotationX();
    }

    /**
     * Sets the degrees that the view is rotated around the horizontal axis through the pivot point.
     * Increasing values result in clockwise rotation from the viewpoint of looking down the
     * x axis.
     *
     * When rotating large views, it is recommended to adjust the camera distance
     * accordingly. Refer to {@link #setCameraDistance(float)} for more information.
     *
     * @param rotationX The degrees of X rotation.
     *
     * @see #getRotationX()
     * @see #getPivotX()
     * @see #getPivotY()
     * @see #setRotation(float)
     * @see #setRotationY(float)
     * @see #setCameraDistance(float)
     *
     * @attr ref android.R.styleable#View_rotationX
     */
    public void setRotationX(float rotationX) {
        if (rotationX != getRotationX()) {
            invalidateViewProperty(true, false);
            mRenderNode.setRotationX(rotationX);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The amount that the view is scaled in x around the pivot point, as a proportion of
     * the view's unscaled width. A value of 1, the default, means that no scaling is applied.
     *
     * <p>By default, this is 1.0f.
     *
     * @see #getPivotX()
     * @see #getPivotY()
     * @return The scaling factor.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getScaleX() {
        return mRenderNode.getScaleX();
    }

    /**
     * Sets the amount that the view is scaled in x around the pivot point, as a proportion of
     * the view's unscaled width. A value of 1 means that no scaling is applied.
     *
     * @param scaleX The scaling factor.
     * @see #getPivotX()
     * @see #getPivotY()
     *
```

```
     * @attr ref android.R.styleable#View_scaleX
     */
    public void setScaleX(float scaleX) {
        if (scaleX != getScaleX()) {
            invalidateViewProperty(true, false);
            mRenderNode.setScaleX(scaleX);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The amount that the view is scaled in y around the pivot point, as a proportion of
     * the view's unscaled height. A value of 1, the default, means that no scaling is applied.
     *
     * <p>By default, this is 1.0f.
     *
     * @see #getPivotX()
     * @see #getPivotY()
     * @return The scaling factor.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getScaleY() {
        return mRenderNode.getScaleY();
    }

    /**
     * Sets the amount that the view is scaled in Y around the pivot point, as a proportion of
     * the view's unscaled width. A value of 1 means that no scaling is applied.
     *
     * @param scaleY The scaling factor.
     * @see #getPivotX()
     * @see #getPivotY()
     *
     * @attr ref android.R.styleable#View_scaleY
     */
    public void setScaleY(float scaleY) {
        if (scaleY != getScaleY()) {
            invalidateViewProperty(true, false);
            mRenderNode.setScaleY(scaleY);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The x location of the point around which the view is {@link #setRotation(float) rotated}
     * and {@link #setScaleX(float) scaled}.
     *
     * @see #getRotation()
     * @see #getScaleX()
     * @see #getScaleY()
     * @see #getPivotY()
     * @return The x location of the pivot point.
     *
     * @attr ref android.R.styleable#View_transformPivotX
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getPivotX() {
        return mRenderNode.getPivotX();
    }

    /**
     * Sets the x location of the point around which the view is
     * {@link #setRotation(float) rotated} and {@link #setScaleX(float) scaled}.
     * By default, the pivot point is centered on the object.
     * Setting this property disables this behavior and causes the view to use only the
     * explicitly set pivotX and pivotY values.
     *
     * @param pivotX The x location of the pivot point.
     * @see #getRotation()
     * @see #getScaleX()
     * @see #getScaleY()
     * @see #getPivotY()
     *
     * @attr ref android.R.styleable#View_transformPivotX
     */
    public void setPivotX(float pivotX) {
```

```
    if (!mRenderNode.isPivotExplicitlySet() || pivotX != getPivotX()) {
        invalidateViewProperty(true, false);
        mRenderNode.setPivotX(pivotX);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
    }
}

/**
 * The y location of the point around which the view is {@link #setRotation(float) rotated}
 * and {@link #setScaleY(float) scaled}.
 *
 * @see #getRotation()
 * @see #getScaleX()
 * @see #getScaleY()
 * @see #getPivotY()
 * @return The y location of the pivot point.
 *
 * @attr ref android.R.styleable#View_transformPivotY
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getPivotY() {
    return mRenderNode.getPivotY();
}

/**
 * Sets the y location of the point around which the view is {@link #setRotation(float) rotated}
 * and {@link #setScaleY(float) scaled}. By default, the pivot point is centered on the object.
 * Setting this property disables this behavior and causes the view to use only the
 * explicitly set pivotX and pivotY values.
 *
 * @param pivotY The y location of the pivot point.
 * @see #getRotation()
 * @see #getScaleX()
 * @see #getScaleY()
 * @see #getPivotY()
 *
 * @attr ref android.R.styleable#View_transformPivotY
 */
public void setPivotY(float pivotY) {
    if (!mRenderNode.isPivotExplicitlySet() || pivotY != getPivotY()) {
        invalidateViewProperty(true, false);
        mRenderNode.setPivotY(pivotY);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
    }
}

/**
 * The opacity of the view. This is a value from 0 to 1, where 0 means the view is
 * completely transparent and 1 means the view is completely opaque.
 *
 * <p>By default this is 1.0f.
 * @return The opacity of the view.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getAlpha() {
    return mTransformationInfo != null ? mTransformationInfo.mAlpha : 1;
}

/**
 * Sets the behavior for overlapping rendering for this view (see {@link
 * #hasOverlappingRendering()} for more details on this behavior). Calling this method
 * is an alternative to overriding {@link #hasOverlappingRendering()} in a subclass,
 * providing the value which is then used internally. That is, when {@link
 * #forceHasOverlappingRendering(boolean)} is called, the value of {@link
 * #hasOverlappingRendering()} is ignored and the value passed into this method is used
 * instead.
 *
 * @param hasOverlappingRendering The value for overlapping rendering to be used internally
 * instead of that returned by {@link #hasOverlappingRendering()}.
 *
 * @attr ref android.R.styleable#View_forceHasOverlappingRendering
 */
public void forceHasOverlappingRendering(boolean hasOverlappingRendering) {
    mPrivateFlags3 |= PFLAG3_HAS_OVERLAPPING_RENDERING_FORCED;
    if (hasOverlappingRendering) {
        mPrivateFlags3 |= PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE;
    } else {
        mPrivateFlags3 &= ~PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE;
```

```
    }
}

/**
 * Returns the value for overlapping rendering that is used internally. This is either
 * the value passed into {@link #forceHasOverlappingRendering(boolean)}, if called, or
 * the return value of {@link #hasOverlappingRendering()}, otherwise.
 *
 * @return The value for overlapping rendering being used internally.
 */
public final boolean getHasOverlappingRendering() {
    return (mPrivateFlags3 & PFLAG3_HAS_OVERLAPPING_RENDERING_FORCED) != 0 ?
            (mPrivateFlags3 & PFLAG3_OVERLAPPING_RENDERING_FORCED_VALUE) != 0 :
            hasOverlappingRendering();
}

/**
 * Returns whether this View has content which overlaps.
 *
 * <p>This function, intended to be overridden by specific View types, is an optimization when
 * alpha is set on a view. If rendering overlaps in a view with alpha < 1, that view is drawn to
 * an offscreen buffer and then composited into place, which can be expensive. If the view has
 * no overlapping rendering, the view can draw each primitive with the appropriate alpha value
 * directly. An example of overlapping rendering is a TextView with a background image, such as
 * a Button. An example of non-overlapping rendering is a TextView with no background, or an
 * ImageView with only the foreground image. The default implementation returns true; subclasses
 * should override if they have cases which can be optimized.</p>
 *
 * <p>The current implementation of the saveLayer and saveLayerAlpha methods in {@link Canvas}
 * necessitates that a View return true if it uses the methods internally without passing the
 * {@link Canvas#CLIP_TO_LAYER_SAVE_FLAG}.</p>
 *
 * <p><strong>Note:</strong> The return value of this method is ignored if {@link
 * #forceHasOverlappingRendering(boolean)} has been called on this view.</p>
 *
 * @return true if the content in this view might overlap, false otherwise.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public boolean hasOverlappingRendering() {
    return true;
}

/**
 * Sets the opacity of the view to a value from 0 to 1, where 0 means the view is
 * completely transparent and 1 means the view is completely opaque.
 *
 * <p class="note"><strong>Note:</strong> setting alpha to a translucent value (0 < alpha < 1)
 * can have significant performance implications, especially for large views. It is best to use
 * the alpha property sparingly and transiently, as in the case of fading animations.</p>
 *
 * <p>For a view with a frequently changing alpha, such as during a fading animation, it is
 * strongly recommended for performance reasons to either override
 * {@link #hasOverlappingRendering()} to return <code>false</code> if appropriate, or setting a
 * {@link #setLayerType(int, android.graphics.Paint) layer type} on the view for the duration
 * of the animation. On versions {@link android.os.Build.VERSION_CODES#M} and below,
 * the default path for rendering an unlayered View with alpha could add multiple milliseconds
 * of rendering cost, even for simple or small views. Starting with
 * {@link android.os.Build.VERSION_CODES#M}, {@link #LAYER_TYPE_HARDWARE} is automatically
 * applied to the view at the rendering level.</p>
 *
 * <p>If this view overrides {@link #onSetAlpha(int)} to return true, then this view is
 * responsible for applying the opacity itself.</p>
 *
 * <p>On versions {@link android.os.Build.VERSION_CODES#LOLLIPOP_MR1} and below, note that if
 * the view is backed by a {@link #setLayerType(int, android.graphics.Paint) layer} and is
 * associated with a {@link #setLayerPaint(android.graphics.Paint) layer paint}, setting an
 * alpha value less than 1.0 will supersede the alpha of the layer paint.</p>
 *
 * <p>Starting with {@link android.os.Build.VERSION_CODES#M}, setting a translucent alpha
 * value will clip a View to its bounds, unless the View returns <code>false</code> from
 * {@link #hasOverlappingRendering}.</p>
 *
 * @param alpha The opacity of the view.
 *
 * @see #hasOverlappingRendering()
 * @see #setLayerType(int, android.graphics.Paint)
 *
 * @attr ref android.R.styleable#View_alpha
 */
public void setAlpha(@FloatRange(from=0.0, to=1.0) float alpha) {
    ensureTransformationInfo();
    if (mTransformationInfo.mAlpha != alpha) {
```

```
        // Report visibility changes, which can affect children, to accessibility
        if ((alpha == 0) ^ (mTransformationInfo.mAlpha == 0)) {
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
        mTransformationInfo.mAlpha = alpha;
        if (onSetAlpha((int) (alpha * 255))) {
            mPrivateFlags |= PFLAG_ALPHA_SET;
            // subclass is handling alpha - don't optimize rendering cache invalidation
            invalidateParentCaches();
            invalidate(true);
        } else {
            mPrivateFlags &= ~PFLAG_ALPHA_SET;
            invalidateViewProperty(true, false);
            mRenderNode.setAlpha(getFinalAlpha());
        }
    }
}

/**
 * Faster version of setAlpha() which performs the same steps except there are
 * no calls to invalidate(). The caller of this function should perform proper invalidation
 * on the parent and this object. The return value indicates whether the subclass handles
 * alpha (the return value for onSetAlpha()).
 *
 * @param alpha The new value for the alpha property
 * @return true if the View subclass handles alpha (the return value for onSetAlpha()) and
 *         the new value for the alpha property is different from the old value
 */
boolean setAlphaNoInvalidation(float alpha) {
    ensureTransformationInfo();
    if (mTransformationInfo.mAlpha != alpha) {
        mTransformationInfo.mAlpha = alpha;
        boolean subclassHandlesAlpha = onSetAlpha((int) (alpha * 255));
        if (subclassHandlesAlpha) {
            mPrivateFlags |= PFLAG_ALPHA_SET;
            return true;
        } else {
            mPrivateFlags &= ~PFLAG_ALPHA_SET;
            mRenderNode.setAlpha(getFinalAlpha());
        }
    }
    return false;
}

/**
 * This property is hidden and intended only for use by the Fade transition, which
 * animates it to produce a visual translucency that does not side-effect (or get
 * affected by) the real alpha property. This value is composited with the other
 * alpha value (and the AlphaAnimation value, when that is present) to produce
 * a final visual translucency result, which is what is passed into the DisplayList.
 *
 * @hide
 */
public void setTransitionAlpha(float alpha) {
    ensureTransformationInfo();
    if (mTransformationInfo.mTransitionAlpha != alpha) {
        mTransformationInfo.mTransitionAlpha = alpha;
        mPrivateFlags &= ~PFLAG_ALPHA_SET;
        invalidateViewProperty(true, false);
        mRenderNode.setAlpha(getFinalAlpha());
    }
}

/**
 * Calculates the visual alpha of this view, which is a combination of the actual
 * alpha value and the transitionAlpha value (if set).
 */
private float getFinalAlpha() {
    if (mTransformationInfo != null) {
        return mTransformationInfo.mAlpha * mTransformationInfo.mTransitionAlpha;
    }
    return 1;
}

/**
 * This property is hidden and intended only for use by the Fade transition, which
 * animates it to produce a visual translucency that does not side-effect (or get
 * affected by) the real alpha property. This value is composited with the other
 * alpha value (and the AlphaAnimation value, when that is present) to produce
 * a final visual translucency result, which is what is passed into the DisplayList.
 *
 * @hide
```

```
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getTransitionAlpha() {
        return mTransformationInfo != null ? mTransformationInfo.mTransitionAlpha : 1;
    }

    /**
     * Top position of this view relative to its parent.
     *
     * @return The top of this view, in pixels.
     */
    @ViewDebug.CapturedViewProperty
    public final int getTop() {
        return mTop;
    }

    /**
     * Sets the top position of this view relative to its parent. This method is meant to be called
     * by the layout system and should not generally be called otherwise, because the property
     * may be changed at any time by the layout.
     *
     * @param top The top of this view, in pixels.
     */
    public final void setTop(int top) {
        if (top != mTop) {
            final boolean matrixIsIdentity = hasIdentityMatrix();
            if (matrixIsIdentity) {
                if (mAttachInfo != null) {
                    int minTop;
                    int yLoc;
                    if (top < mTop) {
                        minTop = top;
                        yLoc = top - mTop;
                    } else {
                        minTop = mTop;
                        yLoc = 0;
                    }
                    invalidate(0, yLoc, mRight - mLeft, mBottom - minTop);
                }
            } else {
                // Double-invalidation is necessary to capture view's old and new areas
                invalidate(true);
            }

            int width = mRight - mLeft;
            int oldHeight = mBottom - mTop;

            mTop = top;
            mRenderNode.setTop(mTop);

            sizeChange(width, mBottom - mTop, width, oldHeight);

            if (!matrixIsIdentity) {
                mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
                invalidate(true);
            }
            mBackgroundSizeChanged = true;
            mDefaultFocusHighlightSizeChanged = true;
            if (mForegroundInfo != null) {
                mForegroundInfo.mBoundsChanged = true;
            }
            invalidateParentIfNeeded();
            if ((mPrivateFlags2 & PFLAG2_VIEW_QUICK_REJECTED) == PFLAG2_VIEW_QUICK_REJECTED) {
                // View was rejected last time it was drawn by its parent; this may have changed
                invalidateParentIfNeeded();
            }
        }
    }

    /**
     * Bottom position of this view relative to its parent.
     *
     * @return The bottom of this view, in pixels.
     */
    @ViewDebug.CapturedViewProperty
    public final int getBottom() {
        return mBottom;
    }

    /**
     * True if this view has changed since the last time being drawn.
     *
```

```
     * @return The dirty state of this view.
     */
    public boolean isDirty() {
        return (mPrivateFlags & PFLAG_DIRTY_MASK) != 0;
    }

    /**
     * Sets the bottom position of this view relative to its parent. This method is meant to be
     * called by the layout system and should not generally be called otherwise, because the
     * property may be changed at any time by the layout.
     *
     * @param bottom The bottom of this view, in pixels.
     */
    public final void setBottom(int bottom) {
        if (bottom != mBottom) {
            final boolean matrixIsIdentity = hasIdentityMatrix();
            if (matrixIsIdentity) {
                if (mAttachInfo != null) {
                    int maxBottom;
                    if (bottom < mBottom) {
                        maxBottom = mBottom;
                    } else {
                        maxBottom = bottom;
                    }
                    invalidate(0, 0, mRight - mLeft, maxBottom - mTop);
                }
            } else {
                // Double-invalidation is necessary to capture view's old and new areas
                invalidate(true);
            }

            int width = mRight - mLeft;
            int oldHeight = mBottom - mTop;

            mBottom = bottom;
            mRenderNode.setBottom(mBottom);

            sizeChange(width, mBottom - mTop, width, oldHeight);

            if (!matrixIsIdentity) {
                mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
                invalidate(true);
            }
            mBackgroundSizeChanged = true;
            mDefaultFocusHighlightSizeChanged = true;
            if (mForegroundInfo != null) {
                mForegroundInfo.mBoundsChanged = true;
            }
            invalidateParentIfNeeded();
            if ((mPrivateFlags2 & PFLAG2_VIEW_QUICK_REJECTED) == PFLAG2_VIEW_QUICK_REJECTED) {
                // View was rejected last time it was drawn by its parent; this may have changed
                invalidateParentIfNeeded();
            }
        }
    }

    /**
     * Left position of this view relative to its parent.
     *
     * @return The left edge of this view, in pixels.
     */
    @ViewDebug.CapturedViewProperty
    public final int getLeft() {
        return mLeft;
    }

    /**
     * Sets the left position of this view relative to its parent. This method is meant to be called
     * by the layout system and should not generally be called otherwise, because the property
     * may be changed at any time by the layout.
     *
     * @param left The left of this view, in pixels.
     */
    public final void setLeft(int left) {
        if (left != mLeft) {
            final boolean matrixIsIdentity = hasIdentityMatrix();
            if (matrixIsIdentity) {
                if (mAttachInfo != null) {
                    int minLeft;
                    int xLoc;
                    if (left < mLeft) {
                        minLeft = left;
```

```
                    xLoc = left - mLeft;
                } else {
                    minLeft = mLeft;
                    xLoc = 0;
                }
                invalidate(xLoc, 0, mRight - minLeft, mBottom - mTop);
            }
        } else {
            // Double-invalidation is necessary to capture view's old and new areas
            invalidate(true);
        }

        int oldWidth = mRight - mLeft;
        int height = mBottom - mTop;

        mLeft = left;
        mRenderNode.setLeft(left);

        sizeChange(mRight - mLeft, height, oldWidth, height);

        if (!matrixIsIdentity) {
            mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
            invalidate(true);
        }
        mBackgroundSizeChanged = true;
        mDefaultFocusHighlightSizeChanged = true;
        if (mForegroundInfo != null) {
            mForegroundInfo.mBoundsChanged = true;
        }
        invalidateParentIfNeeded();
        if ((mPrivateFlags2 & PFLAG2_VIEW_QUICK_REJECTED) == PFLAG2_VIEW_QUICK_REJECTED) {
            // View was rejected last time it was drawn by its parent; this may have changed
            invalidateParentIfNeeded();
        }
    }
}

/**
 * Right position of this view relative to its parent.
 *
 * @return The right edge of this view, in pixels.
 */
@ViewDebug.CapturedViewProperty
public final int getRight() {
    return mRight;
}

/**
 * Sets the right position of this view relative to its parent. This method is meant to be called
 * by the layout system and should not generally be called otherwise, because the property
 * may be changed at any time by the layout.
 *
 * @param right The right of this view, in pixels.
 */
public final void setRight(int right) {
    if (right != mRight) {
        final boolean matrixIsIdentity = hasIdentityMatrix();
        if (matrixIsIdentity) {
            if (mAttachInfo != null) {
                int maxRight;
                if (right < mRight) {
                    maxRight = mRight;
                } else {
                    maxRight = right;
                }
                invalidate(0, 0, maxRight - mLeft, mBottom - mTop);
            }
        } else {
            // Double-invalidation is necessary to capture view's old and new areas
            invalidate(true);
        }

        int oldWidth = mRight - mLeft;
        int height = mBottom - mTop;

        mRight = right;
        mRenderNode.setRight(mRight);

        sizeChange(mRight - mLeft, height, oldWidth, height);

        if (!matrixIsIdentity) {
            mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
```

```
            invalidate(true);
        }
        mBackgroundSizeChanged = true;
        mDefaultFocusHighlightSizeChanged = true;
        if (mForegroundInfo != null) {
            mForegroundInfo.mBoundsChanged = true;
        }
        invalidateParentIfNeeded();
        if ((mPrivateFlags2 & PFLAG2_VIEW_QUICK_REJECTED) == PFLAG2_VIEW_QUICK_REJECTED) {
            // View was rejected last time it was drawn by its parent; this may have changed
            invalidateParentIfNeeded();
        }
    }
}

/**
 * The visual x position of this view, in pixels. This is equivalent to the
 * {@link #setTranslationX(float) translationX} property plus the current
 * {@link #getLeft() left} property.
 *
 * @return The visual x position of this view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getX() {
    return mLeft + getTranslationX();
}

/**
 * Sets the visual x position of this view, in pixels. This is equivalent to setting the
 * {@link #setTranslationX(float) translationX} property to be the difference between
 * the x value passed in and the current {@link #getLeft() left} property.
 *
 * @param x The visual x position of this view, in pixels.
 */
public void setX(float x) {
    setTranslationX(x - mLeft);
}

/**
 * The visual y position of this view, in pixels. This is equivalent to the
 * {@link #setTranslationY(float) translationY} property plus the current
 * {@link #getTop() top} property.
 *
 * @return The visual y position of this view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getY() {
    return mTop + getTranslationY();
}

/**
 * Sets the visual y position of this view, in pixels. This is equivalent to setting the
 * {@link #setTranslationY(float) translationY} property to be the difference between
 * the y value passed in and the current {@link #getTop() top} property.
 *
 * @param y The visual y position of this view, in pixels.
 */
public void setY(float y) {
    setTranslationY(y - mTop);
}

/**
 * The visual z position of this view, in pixels. This is equivalent to the
 * {@link #setTranslationZ(float) translationZ} property plus the current
 * {@link #getElevation() elevation} property.
 *
 * @return The visual z position of this view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getZ() {
    return getElevation() + getTranslationZ();
}

/**
 * Sets the visual z position of this view, in pixels. This is equivalent to setting the
 * {@link #setTranslationZ(float) translationZ} property to be the difference between
 * the x value passed in and the current {@link #getElevation() elevation} property.
 *
 * @param z The visual z position of this view, in pixels.
 */
public void setZ(float z) {
    setTranslationZ(z - getElevation());
```

```
}

/**
 * The base elevation of this view relative to its parent, in pixels.
 *
 * @return The base depth position of the view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getElevation() {
    return mRenderNode.getElevation();
}

/**
 * Sets the base elevation of this view, in pixels.
 *
 * @attr ref android.R.styleable#View_elevation
 */
public void setElevation(float elevation) {
    if (elevation != getElevation()) {
        invalidateViewProperty(true, false);
        mRenderNode.setElevation(elevation);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
    }
}

/**
 * The horizontal location of this view relative to its {@link #getLeft() left} position.
 * This position is post-layout, in addition to wherever the object's
 * layout placed it.
 *
 * @return The horizontal position of this view relative to its left position, in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getTranslationX() {
    return mRenderNode.getTranslationX();
}

/**
 * Sets the horizontal location of this view relative to its {@link #getLeft() left} position.
 * This effectively positions the object post-layout, in addition to wherever the object's
 * layout placed it.
 *
 * @param translationX The horizontal position of this view relative to its left position,
 * in pixels.
 *
 * @attr ref android.R.styleable#View_translationX
 */
public void setTranslationX(float translationX) {
    if (translationX != getTranslationX()) {
        invalidateViewProperty(true, false);
        mRenderNode.setTranslationX(translationX);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
}

/**
 * The vertical location of this view relative to its {@link #getTop() top} position.
 * This position is post-layout, in addition to wherever the object's
 * layout placed it.
 *
 * @return The vertical position of this view relative to its top position,
 * in pixels.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public float getTranslationY() {
    return mRenderNode.getTranslationY();
}

/**
 * Sets the vertical location of this view relative to its {@link #getTop() top} position.
 * This effectively positions the object post-layout, in addition to wherever the object's
 * layout placed it.
 *
 * @param translationY The vertical position of this view relative to its top position,
 * in pixels.
 *
 * @attr ref android.R.styleable#View_translationY
```

```
     */
    public void setTranslationY(float translationY) {
        if (translationY != getTranslationY()) {
            invalidateViewProperty(true, false);
            mRenderNode.setTranslationY(translationY);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
            notifySubtreeAccessibilityStateChangedIfNeeded();
        }
    }

    /**
     * The depth location of this view relative to its {@link #getElevation() elevation}.
     *
     * @return The depth of this view relative to its elevation.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public float getTranslationZ() {
        return mRenderNode.getTranslationZ();
    }

    /**
     * Sets the depth location of this view relative to its {@link #getElevation() elevation}.
     *
     * @attr ref android.R.styleable#View_translationZ
     */
    public void setTranslationZ(float translationZ) {
        if (translationZ != getTranslationZ()) {
            invalidateViewProperty(true, false);
            mRenderNode.setTranslationZ(translationZ);
            invalidateViewProperty(false, true);

            invalidateParentIfNeededAndWasQuickRejected();
        }
    }

    /** @hide */
    public void setAnimationMatrix(Matrix matrix) {
        invalidateViewProperty(true, false);
        mRenderNode.setAnimationMatrix(matrix);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
    }

    /**
     * Returns the current StateListAnimator if exists.
     *
     * @return StateListAnimator or null if it does not exists
     * @see    #setStateListAnimator(android.animation.StateListAnimator)
     */
    public StateListAnimator getStateListAnimator() {
        return mStateListAnimator;
    }

    /**
     * Attaches the provided StateListAnimator to this View.
     * <p>
     * Any previously attached StateListAnimator will be detached.
     *
     * @param stateListAnimator The StateListAnimator to update the view
     * @see android.animation.StateListAnimator
     */
    public void setStateListAnimator(StateListAnimator stateListAnimator) {
        if (mStateListAnimator == stateListAnimator) {
            return;
        }
        if (mStateListAnimator != null) {
            mStateListAnimator.setTarget(null);
        }
        mStateListAnimator = stateListAnimator;
        if (stateListAnimator != null) {
            stateListAnimator.setTarget(this);
            if (isAttachedToWindow()) {
                stateListAnimator.setState(getDrawableState());
            }
        }
    }

    /**
     * Returns whether the Outline should be used to clip the contents of the View.
```

```
     * <p>
     * Note that this flag will only be respected if the View's Outline returns true from
     * {@link Outline#canClip()}.
     *
     * @see #setOutlineProvider(ViewOutlineProvider)
     * @see #setClipToOutline(boolean)
     */
    public final boolean getClipToOutline() {
        return mRenderNode.getClipToOutline();
    }

    /**
     * Sets whether the View's Outline should be used to clip the contents of the View.
     * <p>
     * Only a single non-rectangular clip can be applied on a View at any time.
     * Circular clips from a {@link ViewAnimationUtils#createCircularReveal(View, int, int, float, float)
     * circular reveal} animation take priority over Outline clipping, and
     * child Outline clipping takes priority over Outline clipping done by a
     * parent.
     * <p>
     * Note that this flag will only be respected if the View's Outline returns true from
     * {@link Outline#canClip()}.
     *
     * @see #setOutlineProvider(ViewOutlineProvider)
     * @see #getClipToOutline()
     */
    public void setClipToOutline(boolean clipToOutline) {
        damageInParent();
        if (getClipToOutline() != clipToOutline) {
            mRenderNode.setClipToOutline(clipToOutline);
        }
    }

    // correspond to the enum values of View_outlineProvider
    private static final int PROVIDER_BACKGROUND = 0;
    private static final int PROVIDER_NONE = 1;
    private static final int PROVIDER_BOUNDS = 2;
    private static final int PROVIDER_PADDED_BOUNDS = 3;
    private void setOutlineProviderFromAttribute(int providerInt) {
        switch (providerInt) {
            case PROVIDER_BACKGROUND:
                setOutlineProvider(ViewOutlineProvider.BACKGROUND);
                break;
            case PROVIDER_NONE:
                setOutlineProvider(null);
                break;
            case PROVIDER_BOUNDS:
                setOutlineProvider(ViewOutlineProvider.BOUNDS);
                break;
            case PROVIDER_PADDED_BOUNDS:
                setOutlineProvider(ViewOutlineProvider.PADDED_BOUNDS);
                break;
        }
    }

    /**
     * Sets the {@link ViewOutlineProvider} of the view, which generates the Outline that defines
     * the shape of the shadow it casts, and enables outline clipping.
     * <p>
     * The default ViewOutlineProvider, {@link ViewOutlineProvider#BACKGROUND}, queries the Outline
     * from the View's background drawable, via {@link Drawable#getOutline(Outline)}. Changing the
     * outline provider with this method allows this behavior to be overridden.
     * <p>
     * If the ViewOutlineProvider is null, if querying it for an outline returns false,
     * or if the produced Outline is {@link Outline#isEmpty()}, shadows will not be cast.
     * <p>
     * Only outlines that return true from {@link Outline#canClip()} may be used for clipping.
     *
     * @see #setClipToOutline(boolean)
     * @see #getClipToOutline()
     * @see #getOutlineProvider()
     */
    public void setOutlineProvider(ViewOutlineProvider provider) {
        mOutlineProvider = provider;
        invalidateOutline();
    }

    /**
     * Returns the current {@link ViewOutlineProvider} of the view, which generates the Outline
     * that defines the shape of the shadow it casts, and enables outline clipping.
     *
     * @see #setOutlineProvider(ViewOutlineProvider)
```

```
     */
    public ViewOutlineProvider getOutlineProvider() {
        return mOutlineProvider;
    }

    /**
     * Called to rebuild this View's Outline from its {@link ViewOutlineProvider outline provider}
     *
     * @see #setOutlineProvider(ViewOutlineProvider)
     */
    public void invalidateOutline() {
        rebuildOutline();

        notifySubtreeAccessibilityStateChangedIfNeeded();
        invalidateViewProperty(false, false);
    }

    /**
     * Internal version of {@link #invalidateOutline()} which invalidates the
     * outline without invalidating the view itself. This is intended to be called from
     * within methods in the View class itself which are the result of the view being
     * invalidated already. For example, when we are drawing the background of a View,
     * we invalidate the outline in case it changed in the meantime, but we do not
     * need to invalidate the view because we're already drawing the background as part
     * of drawing the view in response to an earlier invalidation of the view.
     */
    private void rebuildOutline() {
        // Unattached views ignore this signal, and outline is recomputed in onAttachedToWindow()
        if (mAttachInfo == null) return;

        if (mOutlineProvider == null) {
            // no provider, remove outline
            mRenderNode.setOutline(null);
        } else {
            final Outline outline = mAttachInfo.mTmpOutline;
            outline.setEmpty();
            outline.setAlpha(1.0f);

            mOutlineProvider.getOutline(this, outline);
            mRenderNode.setOutline(outline);
        }
    }

    /**
     * HierarchyViewer only
     *
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public boolean hasShadow() {
        return mRenderNode.hasShadow();
    }


    /** @hide */
    public void setRevealClip(boolean shouldClip, float x, float y, float radius) {
        mRenderNode.setRevealClip(shouldClip, x, y, radius);
        invalidateViewProperty(false, false);
    }

    /**
     * Hit rectangle in parent's coordinates
     *
     * @param outRect The hit rectangle of the view.
     */
    public void getHitRect(Rect outRect) {
        if (hasIdentityMatrix() || mAttachInfo == null) {
            outRect.set(mLeft, mTop, mRight, mBottom);
        } else {
            final RectF tmpRect = mAttachInfo.mTmpTransformRect;
            tmpRect.set(0, 0, getWidth(), getHeight());
            getMatrix().mapRect(tmpRect); // TODO: mRenderNode.mapRect(tmpRect)
            outRect.set((int) tmpRect.left + mLeft, (int) tmpRect.top + mTop,
                    (int) tmpRect.right + mLeft, (int) tmpRect.bottom + mTop);
        }
    }

    /**
     * Determines whether the given point, in local coordinates is inside the view.
     */
    /*package*/ final boolean pointInView(float localX, float localY) {
        return pointInView(localX, localY, 0);
```

```
}

/**
 * Utility method to determine whether the given point, in local coordinates,
 * is inside the view, where the area of the view is expanded by the slop factor.
 * This method is called while processing touch-move events to determine if the event
 * is still within the view.
 *
 * @hide
 */
public boolean pointInView(float localX, float localY, float slop) {
    return localX >= -slop && localY >= -slop && localX < ((mRight - mLeft) + slop) &&
            localY < ((mBottom - mTop) + slop);
}

/**
 * When a view has focus and the user navigates away from it, the next view is searched for
 * starting from the rectangle filled in by this method.
 *
 * By default, the rectangle is the {@link #getDrawingRect(android.graphics.Rect)}
 * of the view.  However, if your view maintains some idea of internal selection,
 * such as a cursor, or a selected row or column, you should override this method and
 * fill in a more specific rectangle.
 *
 * @param r The rectangle to fill in, in this view's coordinates.
 */
public void getFocusedRect(Rect r) {
    getDrawingRect(r);
}

/**
 * If some part of this view is not clipped by any of its parents, then
 * return that area in r in global (root) coordinates. To convert r to local
 * coordinates (without taking possible View rotations into account), offset
 * it by -globalOffset (e.g. r.offset(-globalOffset.x, -globalOffset.y)).
 * If the view is completely clipped or translated out, return false.
 *
 * @param r If true is returned, r holds the global coordinates of the
 *        visible portion of this view.
 * @param globalOffset If true is returned, globalOffset holds the dx,dy
 *        between this view and its root. globalOffet may be null.
 * @return true if r is non-empty (i.e. part of the view is visible at the
 *         root level.
 */
public boolean getGlobalVisibleRect(Rect r, Point globalOffset) {
    int width = mRight - mLeft;
    int height = mBottom - mTop;
    if (width > 0 && height > 0) {
        r.set(0, 0, width, height);
        if (globalOffset != null) {
            globalOffset.set(-mScrollX, -mScrollY);
        }
        return mParent == null || mParent.getChildVisibleRect(this, r, globalOffset);
    }
    return false;
}

public final boolean getGlobalVisibleRect(Rect r) {
    return getGlobalVisibleRect(r, null);
}

public final boolean getLocalVisibleRect(Rect r) {
    final Point offset = mAttachInfo != null ? mAttachInfo.mPoint : new Point();
    if (getGlobalVisibleRect(r, offset)) {
        r.offset(-offset.x, -offset.y); // make r local
        return true;
    }
    return false;
}

/**
 * Offset this view's vertical location by the specified number of pixels.
 *
 * @param offset the number of pixels to offset the view by
 */
public void offsetTopAndBottom(int offset) {
    if (offset != 0) {
        final boolean matrixIsIdentity = hasIdentityMatrix();
        if (matrixIsIdentity) {
            if (isHardwareAccelerated()) {
                invalidateViewProperty(false, false);
            } else {
```

```
                final ViewParent p = mParent;
                if (p != null && mAttachInfo != null) {
                    final Rect r = mAttachInfo.mTmpInvalRect;
                    int minTop;
                    int maxBottom;
                    int yLoc;
                    if (offset < 0) {
                        minTop = mTop + offset;
                        maxBottom = mBottom;
                        yLoc = offset;
                    } else {
                        minTop = mTop;
                        maxBottom = mBottom + offset;
                        yLoc = 0;
                    }
                    r.set(0, yLoc, mRight - mLeft, maxBottom - minTop);
                    p.invalidateChild(this, r);
                }
            }
        } else {
            invalidateViewProperty(false, false);
        }

        mTop += offset;
        mBottom += offset;
        mRenderNode.offsetTopAndBottom(offset);
        if (isHardwareAccelerated()) {
            invalidateViewProperty(false, false);
            invalidateParentIfNeededAndWasQuickRejected();
        } else {
            if (!matrixIsIdentity) {
                invalidateViewProperty(false, true);
            }
            invalidateParentIfNeeded();
        }
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
}

/**
 * Offset this view's horizontal location by the specified amount of pixels.
 *
 * @param offset the number of pixels to offset the view by
 */
public void offsetLeftAndRight(int offset) {
    if (offset != 0) {
        final boolean matrixIsIdentity = hasIdentityMatrix();
        if (matrixIsIdentity) {
            if (isHardwareAccelerated()) {
                invalidateViewProperty(false, false);
            } else {
                final ViewParent p = mParent;
                if (p != null && mAttachInfo != null) {
                    final Rect r = mAttachInfo.mTmpInvalRect;
                    int minLeft;
                    int maxRight;
                    if (offset < 0) {
                        minLeft = mLeft + offset;
                        maxRight = mRight;
                    } else {
                        minLeft = mLeft;
                        maxRight = mRight + offset;
                    }
                    r.set(0, 0, maxRight - minLeft, mBottom - mTop);
                    p.invalidateChild(this, r);
                }
            }
        } else {
            invalidateViewProperty(false, false);
        }

        mLeft += offset;
        mRight += offset;
        mRenderNode.offsetLeftAndRight(offset);
        if (isHardwareAccelerated()) {
            invalidateViewProperty(false, false);
            invalidateParentIfNeededAndWasQuickRejected();
        } else {
            if (!matrixIsIdentity) {
                invalidateViewProperty(false, true);
            }
            invalidateParentIfNeeded();
```

```
        }
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
}

/**
 * Get the LayoutParams associated with this view. All views should have
 * layout parameters. These supply parameters to the <i>parent</i> of this
 * view specifying how it should be arranged. There are many subclasses of
 * ViewGroup.LayoutParams, and these correspond to the different subclasses
 * of ViewGroup that are responsible for arranging their children.
 *
 * This method may return null if this View is not attached to a parent
 * ViewGroup or {@link #setLayoutParams(android.view.ViewGroup.LayoutParams)}
 * was not invoked successfully. When a View is attached to a parent
 * ViewGroup, this method must not return null.
 *
 * @return The LayoutParams associated with this view, or null if no
 *         parameters have been set yet
 */
@ViewDebug.ExportedProperty(deepExport = true, prefix = "layout_")
public ViewGroup.LayoutParams getLayoutParams() {
    return mLayoutParams;
}

/**
 * Set the layout parameters associated with this view. These supply
 * parameters to the <i>parent</i> of this view specifying how it should be
 * arranged. There are many subclasses of ViewGroup.LayoutParams, and these
 * correspond to the different subclasses of ViewGroup that are responsible
 * for arranging their children.
 *
 * @param params The layout parameters for this view, cannot be null
 */
public void setLayoutParams(ViewGroup.LayoutParams params) {
    if (params == null) {
        throw new NullPointerException("Layout parameters cannot be null");
    }
    mLayoutParams = params;
    resolveLayoutParams();
    if (mParent instanceof ViewGroup) {
        ((ViewGroup) mParent).onSetLayoutParams(this, params);
    }
    requestLayout();
}

/**
 * Resolve the layout parameters depending on the resolved layout direction
 *
 * @hide
 */
public void resolveLayoutParams() {
    if (mLayoutParams != null) {
        mLayoutParams.resolveLayoutDirection(getLayoutDirection());
    }
}

/**
 * Set the scrolled position of your view. This will cause a call to
 * {@link #onScrollChanged(int, int, int, int)} and the view will be
 * invalidated.
 * @param x the x position to scroll to
 * @param y the y position to scroll to
 */
public void scrollTo(int x, int y) {
    if (mScrollX != x || mScrollY != y) {
        int oldX = mScrollX;
        int oldY = mScrollY;
        mScrollX = x;
        mScrollY = y;
        invalidateParentCaches();
        onScrollChanged(mScrollX, mScrollY, oldX, oldY);
        if (!awakenScrollBars()) {
            postInvalidateOnAnimation();
        }
    }
}

/**
 * Move the scrolled position of your view. This will cause a call to
 * {@link #onScrollChanged(int, int, int, int)} and the view will be
 * invalidated.
```

```
     * @param x the amount of pixels to scroll by horizontally
     * @param y the amount of pixels to scroll by vertically
     */
    public void scrollBy(int x, int y) {
        scrollTo(mScrollX + x, mScrollY + y);
    }

    /**
     * <p>Trigger the scrollbars to draw. When invoked this method starts an
     * animation to fade the scrollbars out after a default delay. If a subclass
     * provides animated scrolling, the start delay should equal the duration
     * of the scrolling animation.</p>
     *
     * <p>The animation starts only if at least one of the scrollbars is
     * enabled, as specified by {@link #isHorizontalScrollBarEnabled()} and
     * {@link #isVerticalScrollBarEnabled()}. When the animation is started,
     * this method returns true, and false otherwise. If the animation is
     * started, this method calls {@link #invalidate()}; in that case the
     * caller should not call {@link #invalidate()}.</p>
     *
     * <p>This method should be invoked every time a subclass directly updates
     * the scroll parameters.</p>
     *
     * <p>This method is automatically invoked by {@link #scrollBy(int, int)}
     * and {@link #scrollTo(int, int)}.</p>
     *
     * @return true if the animation is played, false otherwise
     *
     * @see #awakenScrollBars(int)
     * @see #scrollBy(int, int)
     * @see #scrollTo(int, int)
     * @see #isHorizontalScrollBarEnabled()
     * @see #isVerticalScrollBarEnabled()
     * @see #setHorizontalScrollBarEnabled(boolean)
     * @see #setVerticalScrollBarEnabled(boolean)
     */
    protected boolean awakenScrollBars() {
        return mScrollCache != null &&
                awakenScrollBars(mScrollCache.scrollBarDefaultDelayBeforeFade, true);
    }

    /**
     * Trigger the scrollbars to draw.
     * This method differs from awakenScrollBars() only in its default duration.
     * initialAwakenScrollBars() will show the scroll bars for longer than
     * usual to give the user more of a chance to notice them.
     *
     * @return true if the animation is played, false otherwise.
     */
    private boolean initialAwakenScrollBars() {
        return mScrollCache != null &&
                awakenScrollBars(mScrollCache.scrollBarDefaultDelayBeforeFade * 4, true);
    }

    /**
     * <p>
     * Trigger the scrollbars to draw. When invoked this method starts an
     * animation to fade the scrollbars out after a fixed delay. If a subclass
     * provides animated scrolling, the start delay should equal the duration of
     * the scrolling animation.
     * </p>
     *
     * <p>
     * The animation starts only if at least one of the scrollbars is enabled,
     * as specified by {@link #isHorizontalScrollBarEnabled()} and
     * {@link #isVerticalScrollBarEnabled()}. When the animation is started,
     * this method returns true, and false otherwise. If the animation is
     * started, this method calls {@link #invalidate()}; in that case the caller
     * should not call {@link #invalidate()}.
     * </p>
     *
     * <p>
     * This method should be invoked every time a subclass directly updates the
     * scroll parameters.
     * </p>
     *
     * @param startDelay the delay, in milliseconds, after which the animation
     *        should start; when the delay is 0, the animation starts
     *        immediately
     * @return true if the animation is played, false otherwise
     *
     * @see #scrollBy(int, int)
```

```
     * @see #scrollTo(int, int)
     * @see #isHorizontalScrollBarEnabled()
     * @see #isVerticalScrollBarEnabled()
     * @see #setHorizontalScrollBarEnabled(boolean)
     * @see #setVerticalScrollBarEnabled(boolean)
     */
    protected boolean awakenScrollBars(int startDelay) {
        return awakenScrollBars(startDelay, true);
    }

    /**
     * <p>
     * Trigger the scrollbars to draw. When invoked this method starts an
     * animation to fade the scrollbars out after a fixed delay. If a subclass
     * provides animated scrolling, the start delay should equal the duration of
     * the scrolling animation.
     * </p>
     *
     * <p>
     * The animation starts only if at least one of the scrollbars is enabled,
     * as specified by {@link #isHorizontalScrollBarEnabled()} and
     * {@link #isVerticalScrollBarEnabled()}. When the animation is started,
     * this method returns true, and false otherwise. If the animation is
     * started, this method calls {@link #invalidate()} if the invalidate parameter
     * is set to true; in that case the caller
     * should not call {@link #invalidate()}.
     * </p>
     *
     * <p>
     * This method should be invoked every time a subclass directly updates the
     * scroll parameters.
     * </p>
     *
     * @param startDelay the delay, in milliseconds, after which the animation
     *        should start; when the delay is 0, the animation starts
     *        immediately
     *
     * @param invalidate Whether this method should call invalidate
     *
     * @return true if the animation is played, false otherwise
     *
     * @see #scrollBy(int, int)
     * @see #scrollTo(int, int)
     * @see #isHorizontalScrollBarEnabled()
     * @see #isVerticalScrollBarEnabled()
     * @see #setHorizontalScrollBarEnabled(boolean)
     * @see #setVerticalScrollBarEnabled(boolean)
     */
    protected boolean awakenScrollBars(int startDelay, boolean invalidate) {
        final ScrollabilityCache scrollCache = mScrollCache;

        if (scrollCache == null || !scrollCache.fadeScrollBars) {
            return false;
        }

        if (scrollCache.scrollBar == null) {
            scrollCache.scrollBar = new ScrollBarDrawable();
            scrollCache.scrollBar.setState(getDrawableState());
            scrollCache.scrollBar.setCallback(this);
        }

        if (isHorizontalScrollBarEnabled() || isVerticalScrollBarEnabled()) {

            if (invalidate) {
                // Invalidate to show the scrollbars
                postInvalidateOnAnimation();
            }

            if (scrollCache.state == ScrollabilityCache.OFF) {
                // FIXME: this is copied from WindowManagerService.
                // We should get this value from the system when it
                // is possible to do so.
                final int KEY_REPEAT_FIRST_DELAY = 750;
                startDelay = Math.max(KEY_REPEAT_FIRST_DELAY, startDelay);
            }

            // Tell mScrollCache when we should start fading. This may
            // extend the fade start time if one was already scheduled
            long fadeStartTime = AnimationUtils.currentAnimationTimeMillis() + startDelay;
            scrollCache.fadeStartTime = fadeStartTime;
            scrollCache.state = ScrollabilityCache.ON;
```

```
        // Schedule our fader to run, unscheduling any old ones first
        if (mAttachInfo != null) {
            mAttachInfo.mHandler.removeCallbacks(scrollCache);
            mAttachInfo.mHandler.postAtTime(scrollCache, fadeStartTime);
        }

        return true;
    }

    return false;
}

/**
 * Do not invalidate views which are not visible and which are not running an animation. They
 * will not get drawn and they should not set dirty flags as if they will be drawn
 */
private boolean skipInvalidate() {
    return (mViewFlags & VISIBILITY_MASK) != VISIBLE && mCurrentAnimation == null &&
            (!(mParent instanceof ViewGroup) ||
                    !((ViewGroup) mParent).isViewTransitioning(this));
}

/**
 * Mark the area defined by dirty as needing to be drawn. If the view is
 * visible, {@link #onDraw(android.graphics.Canvas)} will be called at some
 * point in the future.
 * <p>
 * This must be called from a UI thread. To call from a non-UI thread, call
 * {@link #postInvalidate()}.
 * <p>
 * <b>WARNING:</b> In API 19 and below, this method may be destructive to
 * {@code dirty}.
 *
 * @param dirty the rectangle representing the bounds of the dirty region
 */
public void invalidate(Rect dirty) {
    final int scrollX = mScrollX;
    final int scrollY = mScrollY;
    invalidateInternal(dirty.left - scrollX, dirty.top - scrollY,
            dirty.right - scrollX, dirty.bottom - scrollY, true, false);
}

/**
 * Mark the area defined by the rect (l,t,r,b) as needing to be drawn. The
 * coordinates of the dirty rect are relative to the view. If the view is
 * visible, {@link #onDraw(android.graphics.Canvas)} will be called at some
 * point in the future.
 * <p>
 * This must be called from a UI thread. To call from a non-UI thread, call
 * {@link #postInvalidate()}.
 *
 * @param l the left position of the dirty region
 * @param t the top position of the dirty region
 * @param r the right position of the dirty region
 * @param b the bottom position of the dirty region
 */
public void invalidate(int l, int t, int r, int b) {
    final int scrollX = mScrollX;
    final int scrollY = mScrollY;
    invalidateInternal(l - scrollX, t - scrollY, r - scrollX, b - scrollY, true, false);
}

/**
 * Invalidate the whole view. If the view is visible,
 * {@link #onDraw(android.graphics.Canvas)} will be called at some point in
 * the future.
 * <p>
 * This must be called from a UI thread. To call from a non-UI thread, call
 * {@link #postInvalidate()}.
 */
public void invalidate() {
    invalidate(true);
}

/**
 * This is where the invalidate() work actually happens. A full invalidate()
 * causes the drawing cache to be invalidated, but this function can be
 * called with invalidateCache set to false to skip that invalidation step
 * for cases that do not need it (for example, a component that remains at
 * the same dimensions with the same content).
 *
 * @param invalidateCache Whether the drawing cache for this view should be
```

```
     *            invalidated as well. This is usually true for a full
     *            invalidate, but may be set to false if the View's contents or
     *            dimensions have not changed.
     * @hide
     */
    public void invalidate(boolean invalidateCache) {
        invalidateInternal(0, 0, mRight - mLeft, mBottom - mTop, invalidateCache, true);
    }

    void invalidateInternal(int l, int t, int r, int b, boolean invalidateCache,
            boolean fullInvalidate) {
        if (mGhostView != null) {
            mGhostView.invalidate(true);
            return;
        }

        if (skipInvalidate()) {
            return;
        }

        if ((mPrivateFlags & (PFLAG_DRAWN | PFLAG_HAS_BOUNDS)) == (PFLAG_DRAWN | PFLAG_HAS_BOUNDS)
                || (invalidateCache && (mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == PFLAG_DRAWING_CACHE_VALID)
                || (mPrivateFlags & PFLAG_INVALIDATED) != PFLAG_INVALIDATED
                || (fullInvalidate && isOpaque() != mLastIsOpaque)) {
            if (fullInvalidate) {
                mLastIsOpaque = isOpaque();
                mPrivateFlags &= ~PFLAG_DRAWN;
            }

            mPrivateFlags |= PFLAG_DIRTY;

            if (invalidateCache) {
                mPrivateFlags |= PFLAG_INVALIDATED;
                mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
            }

            // Propagate the damage rectangle to the parent view.
            final AttachInfo ai = mAttachInfo;
            final ViewParent p = mParent;
            if (p != null && ai != null && l < r && t < b) {
                final Rect damage = ai.mTmpInvalRect;
                damage.set(l, t, r, b);
                p.invalidateChild(this, damage);
            }

            // Damage the entire projection receiver, if necessary.
            if (mBackground != null && mBackground.isProjected()) {
                final View receiver = getProjectionReceiver();
                if (receiver != null) {
                    receiver.damageInParent();
                }
            }
        }
    }

    /**
     * @return this view's projection receiver, or {@code null} if none exists
     */
    private View getProjectionReceiver() {
        ViewParent p = getParent();
        while (p != null && p instanceof View) {
            final View v = (View) p;
            if (v.isProjectionReceiver()) {
                return v;
            }
            p = p.getParent();
        }

        return null;
    }

    /**
     * @return whether the view is a projection receiver
     */
    private boolean isProjectionReceiver() {
        return mBackground != null;
    }

    /**
     * Quick invalidation for View property changes (alpha, translationXY, etc.). We don't want to
     * set any flags or handle all of the cases handled by the default invalidation methods.
     * Instead, we just want to schedule a traversal in ViewRootImpl with the appropriate
```

```
     * dirty rect. This method calls into fast invalidation methods in ViewGroup that
     * walk up the hierarchy, transforming the dirty rect as necessary.
     *
     * The method also handles normal invalidation logic if display list properties are not
     * being used in this view. The invalidateParent and forceRedraw flags are used by that
     * backup approach, to handle these cases used in the various property-setting methods.
     *
     * @param invalidateParent Force a call to invalidateParentCaches() if display list properties
     * are not being used in this view
     * @param forceRedraw Mark the view as DRAWN to force the invalidation to propagate, if display
     * list properties are not being used in this view
     */
    void invalidateViewProperty(boolean invalidateParent, boolean forceRedraw) {
        if (!isHardwareAccelerated()
                || !mRenderNode.isValid()
                || (mPrivateFlags & PFLAG_DRAW_ANIMATION) != 0) {
            if (invalidateParent) {
                invalidateParentCaches();
            }
            if (forceRedraw) {
                mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
            }
            invalidate(false);
        } else {
            damageInParent();
        }
    }

    /**
     * Tells the parent view to damage this view's bounds.
     *
     * @hide
     */
    protected void damageInParent() {
        if (mParent != null && mAttachInfo != null) {
            mParent.onDescendantInvalidated(this, this);
        }
    }

    /**
     * Utility method to transform a given Rect by the current matrix of this view.
     */
    void transformRect(final Rect rect) {
        if (!getMatrix().isIdentity()) {
            RectF boundingRect = mAttachInfo.mTmpTransformRect;
            boundingRect.set(rect);
            getMatrix().mapRect(boundingRect);
            rect.set((int) Math.floor(boundingRect.left),
                    (int) Math.floor(boundingRect.top),
                    (int) Math.ceil(boundingRect.right),
                    (int) Math.ceil(boundingRect.bottom));
        }
    }

    /**
     * Used to indicate that the parent of this view should clear its caches. This functionality
     * is used to force the parent to rebuild its display list (when hardware-accelerated),
     * which is necessary when various parent-managed properties of the view change, such as
     * alpha, translationX/Y, scrollX/Y, scaleX/Y, and rotation/X/Y. This method only
     * clears the parent caches and does not causes an invalidate event.
     *
     * @hide
     */
    protected void invalidateParentCaches() {
        if (mParent instanceof View) {
            ((View) mParent).mPrivateFlags |= PFLAG_INVALIDATED;
        }
    }

    /**
     * Used to indicate that the parent of this view should be invalidated. This functionality
     * is used to force the parent to rebuild its display list (when hardware-accelerated),
     * which is necessary when various parent-managed properties of the view change, such as
     * alpha, translationX/Y, scrollX/Y, scaleX/Y, and rotation/X/Y. This method will propagate
     * an invalidation event to the parent.
     *
     * @hide
     */
    protected void invalidateParentIfNeeded() {
        if (isHardwareAccelerated() && mParent instanceof View) {
            ((View) mParent).invalidate(true);
        }
```

```
}

/**
 * @hide
 */
protected void invalidateParentIfNeededAndWasQuickRejected() {
    if ((mPrivateFlags2 & PFLAG2_VIEW_QUICK_REJECTED) != 0) {
        // View was rejected last time it was drawn by its parent; this may have changed
        invalidateParentIfNeeded();
    }
}

/**
 * Indicates whether this View is opaque. An opaque View guarantees that it will
 * draw all the pixels overlapping its bounds using a fully opaque color.
 *
 * Subclasses of View should override this method whenever possible to indicate
 * whether an instance is opaque. Opaque Views are treated in a special way by
 * the View hierarchy, possibly allowing it to perform optimizations during
 * invalidate/draw passes.
 *
 * @return True if this View is guaranteed to be fully opaque, false otherwise.
 */
@ViewDebug.ExportedProperty(category = "drawing")
public boolean isOpaque() {
    return (mPrivateFlags & PFLAG_OPAQUE_MASK) == PFLAG_OPAQUE_MASK &&
            getFinalAlpha() >= 1.0f;
}

/**
 * @hide
 */
protected void computeOpaqueFlags() {
    // Opaque if:
    //   - Has a background
    //   - Background is opaque
    //   - Doesn't have scrollbars or scrollbars overlay

    if (mBackground != null && mBackground.getOpacity() == PixelFormat.OPAQUE) {
        mPrivateFlags |= PFLAG_OPAQUE_BACKGROUND;
    } else {
        mPrivateFlags &= ~PFLAG_OPAQUE_BACKGROUND;
    }

    final int flags = mViewFlags;
    if (((flags & SCROLLBARS_VERTICAL) == 0 && (flags & SCROLLBARS_HORIZONTAL) == 0) ||
            (flags & SCROLLBARS_STYLE_MASK) == SCROLLBARS_INSIDE_OVERLAY ||
            (flags & SCROLLBARS_STYLE_MASK) == SCROLLBARS_OUTSIDE_OVERLAY) {
        mPrivateFlags |= PFLAG_OPAQUE_SCROLLBARS;
    } else {
        mPrivateFlags &= ~PFLAG_OPAQUE_SCROLLBARS;
    }
}

/**
 * @hide
 */
protected boolean hasOpaqueScrollbars() {
    return (mPrivateFlags & PFLAG_OPAQUE_SCROLLBARS) == PFLAG_OPAQUE_SCROLLBARS;
}

/**
 * @return A handler associated with the thread running the View. This
 * handler can be used to pump events in the UI events queue.
 */
public Handler getHandler() {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler;
    }
    return null;
}

/**
 * Returns the queue of runnable for this view.
 *
 * @return the queue of runnables for this view
 */
private HandlerActionQueue getRunQueue() {
    if (mRunQueue == null) {
        mRunQueue = new HandlerActionQueue();
    }
```

```
    return mRunQueue;
}

/**
 * Gets the view root associated with the View.
 * @return The view root, or null if none.
 * @hide
 */
public ViewRootImpl getViewRootImpl() {
    if (mAttachInfo != null) {
        return mAttachInfo.mViewRootImpl;
    }
    return null;
}

/**
 * @hide
 */
public ThreadedRenderer getThreadedRenderer() {
    return mAttachInfo != null ? mAttachInfo.mThreadedRenderer : null;
}

/**
 * <p>Causes the Runnable to be added to the message queue.
 * The runnable will be run on the user interface thread.</p>
 *
 * @param action The Runnable that will be executed.
 *
 * @return Returns true if the Runnable was successfully placed in to the
 *         message queue.  Returns false on failure, usually because the
 *         looper processing the message queue is exiting.
 *
 * @see #postDelayed
 * @see #removeCallbacks
 */
public boolean post(Runnable action) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action);
    }

    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    getRunQueue().post(action);
    return true;
}

/**
 * <p>Causes the Runnable to be added to the message queue, to be run
 * after the specified amount of time elapses.
 * The runnable will be run on the user interface thread.</p>
 *
 * @param action The Runnable that will be executed.
 * @param delayMillis The delay (in milliseconds) until the Runnable
 *        will be executed.
 *
 * @return true if the Runnable was successfully placed in to the
 *         message queue.  Returns false on failure, usually because the
 *         looper processing the message queue is exiting.  Note that a
 *         result of true does not mean the Runnable will be processed --
 *         if the looper is quit before the delivery time of the message
 *         occurs then the message will be dropped.
 *
 * @see #post
 * @see #removeCallbacks
 */
public boolean postDelayed(Runnable action, long delayMillis) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.postDelayed(action, delayMillis);
    }

    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    getRunQueue().postDelayed(action, delayMillis);
    return true;
}

/**
 * <p>Causes the Runnable to execute on the next animation time step.
 * The runnable will be run on the user interface thread.</p>
 *
```

```
     * @param action The Runnable that will be executed.
     *
     * @see #postOnAnimationDelayed
     * @see #removeCallbacks
     */
    public void postOnAnimation(Runnable action) {
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null) {
            attachInfo.mViewRootImpl.mChoreographer.postCallback(
                    Choreographer.CALLBACK_ANIMATION, action, null);
        } else {
            // Postpone the runnable until we know
            // on which thread it needs to run.
            getRunQueue().post(action);
        }
    }

    /**
     * <p>Causes the Runnable to execute on the next animation time step,
     * after the specified amount of time elapses.
     * The runnable will be run on the user interface thread.</p>
     *
     * @param action The Runnable that will be executed.
     * @param delayMillis The delay (in milliseconds) until the Runnable
     *        will be executed.
     *
     * @see #postOnAnimation
     * @see #removeCallbacks
     */
    public void postOnAnimationDelayed(Runnable action, long delayMillis) {
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null) {
            attachInfo.mViewRootImpl.mChoreographer.postCallbackDelayed(
                    Choreographer.CALLBACK_ANIMATION, action, null, delayMillis);
        } else {
            // Postpone the runnable until we know
            // on which thread it needs to run.
            getRunQueue().postDelayed(action, delayMillis);
        }
    }

    /**
     * <p>Removes the specified Runnable from the message queue.</p>
     *
     * @param action The Runnable to remove from the message handling queue
     *
     * @return true if this view could ask the Handler to remove the Runnable,
     *         false otherwise. When the returned value is true, the Runnable
     *         may or may not have been actually removed from the message queue
     *         (for instance, if the Runnable was not in the queue already.)
     *
     * @see #post
     * @see #postDelayed
     * @see #postOnAnimation
     * @see #postOnAnimationDelayed
     */
    public boolean removeCallbacks(Runnable action) {
        if (action != null) {
            final AttachInfo attachInfo = mAttachInfo;
            if (attachInfo != null) {
                attachInfo.mHandler.removeCallbacks(action);
                attachInfo.mViewRootImpl.mChoreographer.removeCallbacks(
                        Choreographer.CALLBACK_ANIMATION, action, null);
            }
            getRunQueue().removeCallbacks(action);
        }
        return true;
    }

    /**
     * <p>Cause an invalidate to happen on a subsequent cycle through the event loop.
     * Use this to invalidate the View from a non-UI thread.</p>
     *
     * <p>This method can be invoked from outside of the UI thread
     * only when this View is attached to a window.</p>
     *
     * @see #invalidate()
     * @see #postInvalidateDelayed(long)
     */
    public void postInvalidate() {
        postInvalidateDelayed(0);
    }
```

```java
/**
 * <p>Cause an invalidate of the specified area to happen on a subsequent cycle
 * through the event loop. Use this to invalidate the View from a non-UI thread.</p>
 *
 * <p>This method can be invoked from outside of the UI thread
 * only when this View is attached to a window.</p>
 *
 * @param left The left coordinate of the rectangle to invalidate.
 * @param top The top coordinate of the rectangle to invalidate.
 * @param right The right coordinate of the rectangle to invalidate.
 * @param bottom The bottom coordinate of the rectangle to invalidate.
 *
 * @see #invalidate(int, int, int, int)
 * @see #invalidate(Rect)
 * @see #postInvalidateDelayed(long, int, int, int, int)
 */
public void postInvalidate(int left, int top, int right, int bottom) {
    postInvalidateDelayed(0, left, top, right, bottom);
}

/**
 * <p>Cause an invalidate to happen on a subsequent cycle through the event
 * loop. Waits for the specified amount of time.</p>
 *
 * <p>This method can be invoked from outside of the UI thread
 * only when this View is attached to a window.</p>
 *
 * @param delayMilliseconds the duration in milliseconds to delay the
 *         invalidation by
 *
 * @see #invalidate()
 * @see #postInvalidate()
 */
public void postInvalidateDelayed(long delayMilliseconds) {
    // We try only with the AttachInfo because there's no point in invalidating
    // if we are not attached to our window
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        attachInfo.mViewRootImpl.dispatchInvalidateDelayed(this, delayMilliseconds);
    }
}

/**
 * <p>Cause an invalidate of the specified area to happen on a subsequent cycle
 * through the event loop. Waits for the specified amount of time.</p>
 *
 * <p>This method can be invoked from outside of the UI thread
 * only when this View is attached to a window.</p>
 *
 * @param delayMilliseconds the duration in milliseconds to delay the
 *         invalidation by
 * @param left The left coordinate of the rectangle to invalidate.
 * @param top The top coordinate of the rectangle to invalidate.
 * @param right The right coordinate of the rectangle to invalidate.
 * @param bottom The bottom coordinate of the rectangle to invalidate.
 *
 * @see #invalidate(int, int, int, int)
 * @see #invalidate(Rect)
 * @see #postInvalidate(int, int, int, int)
 */
public void postInvalidateDelayed(long delayMilliseconds, int left, int top,
        int right, int bottom) {

    // We try only with the AttachInfo because there's no point in invalidating
    // if we are not attached to our window
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        final AttachInfo.InvalidateInfo info = AttachInfo.InvalidateInfo.obtain();
        info.target = this;
        info.left = left;
        info.top = top;
        info.right = right;
        info.bottom = bottom;

        attachInfo.mViewRootImpl.dispatchInvalidateRectDelayed(info, delayMilliseconds);
    }
}

/**
 * <p>Cause an invalidate to happen on the next animation time step, typically the
 * next display frame.</p>
```

```
     *
     * <p>This method can be invoked from outside of the UI thread
     * only when this View is attached to a window.</p>
     *
     * @see #invalidate()
     */
    public void postInvalidateOnAnimation() {
        // We try only with the AttachInfo because there's no point in invalidating
        // if we are not attached to our window
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null) {
            attachInfo.mViewRootImpl.dispatchInvalidateOnAnimation(this);
        }
    }

    /**
     * <p>Cause an invalidate of the specified area to happen on the next animation
     * time step, typically the next display frame.</p>
     *
     * <p>This method can be invoked from outside of the UI thread
     * only when this View is attached to a window.</p>
     *
     * @param left The left coordinate of the rectangle to invalidate.
     * @param top The top coordinate of the rectangle to invalidate.
     * @param right The right coordinate of the rectangle to invalidate.
     * @param bottom The bottom coordinate of the rectangle to invalidate.
     *
     * @see #invalidate(int, int, int, int)
     * @see #invalidate(Rect)
     */
    public void postInvalidateOnAnimation(int left, int top, int right, int bottom) {
        // We try only with the AttachInfo because there's no point in invalidating
        // if we are not attached to our window
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null) {
            final AttachInfo.InvalidateInfo info = AttachInfo.InvalidateInfo.obtain();
            info.target = this;
            info.left = left;
            info.top = top;
            info.right = right;
            info.bottom = bottom;

            attachInfo.mViewRootImpl.dispatchInvalidateRectOnAnimation(info);
        }
    }

    /**
     * Post a callback to send a {@link AccessibilityEvent#TYPE_VIEW_SCROLLED} event.
     * This event is sent at most once every
     * {@link ViewConfiguration#getSendRecurringAccessibilityEventsInterval()}.
     */
    private void postSendViewScrolledAccessibilityEventCallback() {
        if (mSendViewScrolledAccessibilityEvent == null) {
            mSendViewScrolledAccessibilityEvent = new SendViewScrolledAccessibilityEvent();
        }
        if (!mSendViewScrolledAccessibilityEvent.mIsPending) {
            mSendViewScrolledAccessibilityEvent.mIsPending = true;
            postDelayed(mSendViewScrolledAccessibilityEvent,
                    ViewConfiguration.getSendRecurringAccessibilityEventsInterval());
        }
    }

    /**
     * Called by a parent to request that a child update its values for mScrollX
     * and mScrollY if necessary. This will typically be done if the child is
     * animating a scroll using a {@link android.widget.Scroller Scroller}
     * object.
     */
    public void computeScroll() {
    }

    /**
     * <p>Indicate whether the horizontal edges are faded when the view is
     * scrolled horizontally.</p>
     *
     * @return true if the horizontal edges should are faded on scroll, false
     *         otherwise
     *
     * @see #setHorizontalFadingEdgeEnabled(boolean)
     *
     * @attr ref android.R.styleable#View_requiresFadingEdge
     */
```

```
public boolean isHorizontalFadingEdgeEnabled() {
    return (mViewFlags & FADING_EDGE_HORIZONTAL) == FADING_EDGE_HORIZONTAL;
}

/**
 * <p>Define whether the horizontal edges should be faded when this view
 * is scrolled horizontally.</p>
 *
 * @param horizontalFadingEdgeEnabled true if the horizontal edges should
 *                                    be faded when the view is scrolled
 *                                    horizontally
 *
 * @see #isHorizontalFadingEdgeEnabled()
 *
 * @attr ref android.R.styleable#View_requiresFadingEdge
 */
public void setHorizontalFadingEdgeEnabled(boolean horizontalFadingEdgeEnabled) {
    if (isHorizontalFadingEdgeEnabled() != horizontalFadingEdgeEnabled) {
        if (horizontalFadingEdgeEnabled) {
            initScrollCache();
        }

        mViewFlags ^= FADING_EDGE_HORIZONTAL;
    }
}

/**
 * <p>Indicate whether the vertical edges are faded when the view is
 * scrolled horizontally.</p>
 *
 * @return true if the vertical edges should are faded on scroll, false
 *         otherwise
 *
 * @see #setVerticalFadingEdgeEnabled(boolean)
 *
 * @attr ref android.R.styleable#View_requiresFadingEdge
 */
public boolean isVerticalFadingEdgeEnabled() {
    return (mViewFlags & FADING_EDGE_VERTICAL) == FADING_EDGE_VERTICAL;
}

/**
 * <p>Define whether the vertical edges should be faded when this view
 * is scrolled vertically.</p>
 *
 * @param verticalFadingEdgeEnabled true if the vertical edges should
 *                                  be faded when the view is scrolled
 *                                  vertically
 *
 * @see #isVerticalFadingEdgeEnabled()
 *
 * @attr ref android.R.styleable#View_requiresFadingEdge
 */
public void setVerticalFadingEdgeEnabled(boolean verticalFadingEdgeEnabled) {
    if (isVerticalFadingEdgeEnabled() != verticalFadingEdgeEnabled) {
        if (verticalFadingEdgeEnabled) {
            initScrollCache();
        }

        mViewFlags ^= FADING_EDGE_VERTICAL;
    }
}

/**
 * Returns the strength, or intensity, of the top faded edge. The strength is
 * a value between 0.0 (no fade) and 1.0 (full fade). The default implementation
 * returns 0.0 or 1.0 but no value in between.
 *
 * Subclasses should override this method to provide a smoother fade transition
 * when scrolling occurs.
 *
 * @return the intensity of the top fade as a float between 0.0f and 1.0f
 */
protected float getTopFadingEdgeStrength() {
    return computeVerticalScrollOffset() > 0 ? 1.0f : 0.0f;
}

/**
 * Returns the strength, or intensity, of the bottom faded edge. The strength is
 * a value between 0.0 (no fade) and 1.0 (full fade). The default implementation
 * returns 0.0 or 1.0 but no value in between.
 *
```

```
     * Subclasses should override this method to provide a smoother fade transition
     * when scrolling occurs.
     *
     * @return the intensity of the bottom fade as a float between 0.0f and 1.0f
     */
    protected float getBottomFadingEdgeStrength() {
        return computeVerticalScrollOffset() + computeVerticalScrollExtent() <
                computeVerticalScrollRange() ? 1.0f : 0.0f;
    }

    /**
     * Returns the strength, or intensity, of the left faded edge. The strength is
     * a value between 0.0 (no fade) and 1.0 (full fade). The default implementation
     * returns 0.0 or 1.0 but no value in between.
     *
     * Subclasses should override this method to provide a smoother fade transition
     * when scrolling occurs.
     *
     * @return the intensity of the left fade as a float between 0.0f and 1.0f
     */
    protected float getLeftFadingEdgeStrength() {
        return computeHorizontalScrollOffset() > 0 ? 1.0f : 0.0f;
    }

    /**
     * Returns the strength, or intensity, of the right faded edge. The strength is
     * a value between 0.0 (no fade) and 1.0 (full fade). The default implementation
     * returns 0.0 or 1.0 but no value in between.
     *
     * Subclasses should override this method to provide a smoother fade transition
     * when scrolling occurs.
     *
     * @return the intensity of the right fade as a float between 0.0f and 1.0f
     */
    protected float getRightFadingEdgeStrength() {
        return computeHorizontalScrollOffset() + computeHorizontalScrollExtent() <
                computeHorizontalScrollRange() ? 1.0f : 0.0f;
    }

    /**
     * <p>Indicate whether the horizontal scrollbar should be drawn or not. The
     * scrollbar is not drawn by default.</p>
     *
     * @return true if the horizontal scrollbar should be painted, false
     *         otherwise
     *
     * @see #setHorizontalScrollBarEnabled(boolean)
     */
    public boolean isHorizontalScrollBarEnabled() {
        return (mViewFlags & SCROLLBARS_HORIZONTAL) == SCROLLBARS_HORIZONTAL;
    }

    /**
     * <p>Define whether the horizontal scrollbar should be drawn or not. The
     * scrollbar is not drawn by default.</p>
     *
     * @param horizontalScrollBarEnabled true if the horizontal scrollbar should
     *                                   be painted
     *
     * @see #isHorizontalScrollBarEnabled()
     */
    public void setHorizontalScrollBarEnabled(boolean horizontalScrollBarEnabled) {
        if (isHorizontalScrollBarEnabled() != horizontalScrollBarEnabled) {
            mViewFlags ^= SCROLLBARS_HORIZONTAL;
            computeOpaqueFlags();
            resolvePadding();
        }
    }

    /**
     * <p>Indicate whether the vertical scrollbar should be drawn or not. The
     * scrollbar is not drawn by default.</p>
     *
     * @return true if the vertical scrollbar should be painted, false
     *         otherwise
     *
     * @see #setVerticalScrollBarEnabled(boolean)
     */
    public boolean isVerticalScrollBarEnabled() {
        return (mViewFlags & SCROLLBARS_VERTICAL) == SCROLLBARS_VERTICAL;
    }
```

```java
/**
 * <p>Define whether the vertical scrollbar should be drawn or not. The
 * scrollbar is not drawn by default.</p>
 *
 * @param verticalScrollBarEnabled true if the vertical scrollbar should
 *                                 be painted
 *
 * @see #isVerticalScrollBarEnabled()
 */
public void setVerticalScrollBarEnabled(boolean verticalScrollBarEnabled) {
    if (isVerticalScrollBarEnabled() != verticalScrollBarEnabled) {
        mViewFlags ^= SCROLLBARS_VERTICAL;
        computeOpaqueFlags();
        resolvePadding();
    }
}

/**
 * @hide
 */
protected void recomputePadding() {
    internalSetPadding(mUserPaddingLeft, mPaddingTop, mUserPaddingRight, mUserPaddingBottom);
}

/**
 * Define whether scrollbars will fade when the view is not scrolling.
 *
 * @param fadeScrollbars whether to enable fading
 *
 * @attr ref android.R.styleable#View_fadeScrollbars
 */
public void setScrollbarFadingEnabled(boolean fadeScrollbars) {
    initScrollCache();
    final ScrollabilityCache scrollabilityCache = mScrollCache;
    scrollabilityCache.fadeScrollBars = fadeScrollbars;
    if (fadeScrollbars) {
        scrollabilityCache.state = ScrollabilityCache.OFF;
    } else {
        scrollabilityCache.state = ScrollabilityCache.ON;
    }
}

/**
 *
 * Returns true if scrollbars will fade when this view is not scrolling
 *
 * @return true if scrollbar fading is enabled
 *
 * @attr ref android.R.styleable#View_fadeScrollbars
 */
public boolean isScrollbarFadingEnabled() {
    return mScrollCache != null && mScrollCache.fadeScrollBars;
}

/**
 *
 * Returns the delay before scrollbars fade.
 *
 * @return the delay before scrollbars fade
 *
 * @attr ref android.R.styleable#View_scrollbarDefaultDelayBeforeFade
 */
public int getScrollBarDefaultDelayBeforeFade() {
    return mScrollCache == null ? ViewConfiguration.getScrollDefaultDelay() :
            mScrollCache.scrollBarDefaultDelayBeforeFade;
}

/**
 * Define the delay before scrollbars fade.
 *
 * @param scrollBarDefaultDelayBeforeFade - the delay before scrollbars fade
 *
 * @attr ref android.R.styleable#View_scrollbarDefaultDelayBeforeFade
 */
public void setScrollBarDefaultDelayBeforeFade(int scrollBarDefaultDelayBeforeFade) {
    getScrollCache().scrollBarDefaultDelayBeforeFade = scrollBarDefaultDelayBeforeFade;
}

/**
 *
 * Returns the scrollbar fade duration.
 *
```

```
     * @return the scrollbar fade duration, in milliseconds
     *
     * @attr ref android.R.styleable#View_scrollbarFadeDuration
     */
    public int getScrollBarFadeDuration() {
        return mScrollCache == null ? ViewConfiguration.getScrollBarFadeDuration() :
                mScrollCache.scrollBarFadeDuration;
    }

    /**
     * Define the scrollbar fade duration.
     *
     * @param scrollBarFadeDuration - the scrollbar fade duration, in milliseconds
     *
     * @attr ref android.R.styleable#View_scrollbarFadeDuration
     */
    public void setScrollBarFadeDuration(int scrollBarFadeDuration) {
        getScrollCache().scrollBarFadeDuration = scrollBarFadeDuration;
    }

    /**
     *
     * Returns the scrollbar size.
     *
     * @return the scrollbar size
     *
     * @attr ref android.R.styleable#View_scrollbarSize
     */
    public int getScrollBarSize() {
        return mScrollCache == null ? ViewConfiguration.get(mContext).getScaledScrollBarSize() :
                mScrollCache.scrollBarSize;
    }

    /**
     * Define the scrollbar size.
     *
     * @param scrollBarSize - the scrollbar size
     *
     * @attr ref android.R.styleable#View_scrollbarSize
     */
    public void setScrollBarSize(int scrollBarSize) {
        getScrollCache().scrollBarSize = scrollBarSize;
    }

    /**
     * <p>Specify the style of the scrollbars. The scrollbars can be overlaid or
     * inset. When inset, they add to the padding of the view. And the scrollbars
     * can be drawn inside the padding area or on the edge of the view. For example,
     * if a view has a background drawable and you want to draw the scrollbars
     * inside the padding specified by the drawable, you can use
     * SCROLLBARS_INSIDE_OVERLAY or SCROLLBARS_INSIDE_INSET. If you want them to
     * appear at the edge of the view, ignoring the padding, then you can use
     * SCROLLBARS_OUTSIDE_OVERLAY or SCROLLBARS_OUTSIDE_INSET.</p>
     * @param style the style of the scrollbars. Should be one of
     * SCROLLBARS_INSIDE_OVERLAY, SCROLLBARS_INSIDE_INSET,
     * SCROLLBARS_OUTSIDE_OVERLAY or SCROLLBARS_OUTSIDE_INSET.
     * @see #SCROLLBARS_INSIDE_OVERLAY
     * @see #SCROLLBARS_INSIDE_INSET
     * @see #SCROLLBARS_OUTSIDE_OVERLAY
     * @see #SCROLLBARS_OUTSIDE_INSET
     *
     * @attr ref android.R.styleable#View_scrollbarStyle
     */
    public void setScrollBarStyle(@ScrollBarStyle int style) {
        if (style != (mViewFlags & SCROLLBARS_STYLE_MASK)) {
            mViewFlags = (mViewFlags & ~SCROLLBARS_STYLE_MASK) | (style & SCROLLBARS_STYLE_MASK);
            computeOpaqueFlags();
            resolvePadding();
        }
    }

    /**
     * <p>Returns the current scrollbar style.</p>
     * @return the current scrollbar style
     * @see #SCROLLBARS_INSIDE_OVERLAY
     * @see #SCROLLBARS_INSIDE_INSET
     * @see #SCROLLBARS_OUTSIDE_OVERLAY
     * @see #SCROLLBARS_OUTSIDE_INSET
     *
     * @attr ref android.R.styleable#View_scrollbarStyle
     */
    @ViewDebug.ExportedProperty(mapping = {
```

```
        @ViewDebug.IntToString(from = SCROLLBARS_INSIDE_OVERLAY, to = "INSIDE_OVERLAY"),
        @ViewDebug.IntToString(from = SCROLLBARS_INSIDE_INSET, to = "INSIDE_INSET"),
        @ViewDebug.IntToString(from = SCROLLBARS_OUTSIDE_OVERLAY, to = "OUTSIDE_OVERLAY"),
        @ViewDebug.IntToString(from = SCROLLBARS_OUTSIDE_INSET, to = "OUTSIDE_INSET")
})
@ScrollBarStyle
public int getScrollBarStyle() {
    return mViewFlags & SCROLLBARS_STYLE_MASK;
}

/**
 * <p>Compute the horizontal range that the horizontal scrollbar
 * represents.</p>
 *
 * <p>The range is expressed in arbitrary units that must be the same as the
 * units used by {@link #computeHorizontalScrollExtent()} and
 * {@link #computeHorizontalScrollOffset()}.</p>
 *
 * <p>The default range is the drawing width of this view.</p>
 *
 * @return the total horizontal range represented by the horizontal
 *         scrollbar
 *
 * @see #computeHorizontalScrollExtent()
 * @see #computeHorizontalScrollOffset()
 * @see android.widget.ScrollBarDrawable
 */
protected int computeHorizontalScrollRange() {
    return getWidth();
}

/**
 * <p>Compute the horizontal offset of the horizontal scrollbar's thumb
 * within the horizontal range. This value is used to compute the position
 * of the thumb within the scrollbar's track.</p>
 *
 * <p>The range is expressed in arbitrary units that must be the same as the
 * units used by {@link #computeHorizontalScrollRange()} and
 * {@link #computeHorizontalScrollExtent()}.</p>
 *
 * <p>The default offset is the scroll offset of this view.</p>
 *
 * @return the horizontal offset of the scrollbar's thumb
 *
 * @see #computeHorizontalScrollRange()
 * @see #computeHorizontalScrollExtent()
 * @see android.widget.ScrollBarDrawable
 */
protected int computeHorizontalScrollOffset() {
    return mScrollX;
}

/**
 * <p>Compute the horizontal extent of the horizontal scrollbar's thumb
 * within the horizontal range. This value is used to compute the length
 * of the thumb within the scrollbar's track.</p>
 *
 * <p>The range is expressed in arbitrary units that must be the same as the
 * units used by {@link #computeHorizontalScrollRange()} and
 * {@link #computeHorizontalScrollOffset()}.</p>
 *
 * <p>The default extent is the drawing width of this view.</p>
 *
 * @return the horizontal extent of the scrollbar's thumb
 *
 * @see #computeHorizontalScrollRange()
 * @see #computeHorizontalScrollOffset()
 * @see android.widget.ScrollBarDrawable
 */
protected int computeHorizontalScrollExtent() {
    return getWidth();
}

/**
 * <p>Compute the vertical range that the vertical scrollbar represents.</p>
 *
 * <p>The range is expressed in arbitrary units that must be the same as the
 * units used by {@link #computeVerticalScrollExtent()} and
 * {@link #computeVerticalScrollOffset()}.</p>
 *
 * @return the total vertical range represented by the vertical scrollbar
 *
```

```
     * <p>The default range is the drawing height of this view.</p>
     *
     * @see #computeVerticalScrollExtent()
     * @see #computeVerticalScrollOffset()
     * @see android.widget.ScrollBarDrawable
     */
    protected int computeVerticalScrollRange() {
        return getHeight();
    }

    /**
     * <p>Compute the vertical offset of the vertical scrollbar's thumb
     * within the horizontal range. This value is used to compute the position
     * of the thumb within the scrollbar's track.</p>
     *
     * <p>The range is expressed in arbitrary units that must be the same as the
     * units used by {@link #computeVerticalScrollRange()} and
     * {@link #computeVerticalScrollExtent()}.</p>
     *
     * <p>The default offset is the scroll offset of this view.</p>
     *
     * @return the vertical offset of the scrollbar's thumb
     *
     * @see #computeVerticalScrollRange()
     * @see #computeVerticalScrollExtent()
     * @see android.widget.ScrollBarDrawable
     */
    protected int computeVerticalScrollOffset() {
        return mScrollY;
    }

    /**
     * <p>Compute the vertical extent of the vertical scrollbar's thumb
     * within the vertical range. This value is used to compute the length
     * of the thumb within the scrollbar's track.</p>
     *
     * <p>The range is expressed in arbitrary units that must be the same as the
     * units used by {@link #computeVerticalScrollRange()} and
     * {@link #computeVerticalScrollOffset()}.</p>
     *
     * <p>The default extent is the drawing height of this view.</p>
     *
     * @return the vertical extent of the scrollbar's thumb
     *
     * @see #computeVerticalScrollRange()
     * @see #computeVerticalScrollOffset()
     * @see android.widget.ScrollBarDrawable
     */
    protected int computeVerticalScrollExtent() {
        return getHeight();
    }

    /**
     * Check if this view can be scrolled horizontally in a certain direction.
     *
     * @param direction Negative to check scrolling left, positive to check scrolling right.
     * @return true if this view can be scrolled in the specified direction, false otherwise.
     */
    public boolean canScrollHorizontally(int direction) {
        final int offset = computeHorizontalScrollOffset();
        final int range = computeHorizontalScrollRange() - computeHorizontalScrollExtent();
        if (range == 0) return false;
        if (direction < 0) {
            return offset > 0;
        } else {
            return offset < range - 1;
        }
    }

    /**
     * Check if this view can be scrolled vertically in a certain direction.
     *
     * @param direction Negative to check scrolling up, positive to check scrolling down.
     * @return true if this view can be scrolled in the specified direction, false otherwise.
     */
    public boolean canScrollVertically(int direction) {
        final int offset = computeVerticalScrollOffset();
        final int range = computeVerticalScrollRange() - computeVerticalScrollExtent();
        if (range == 0) return false;
        if (direction < 0) {
            return offset > 0;
        } else {
```

```
            return offset < range - 1;
        }
    }

    void getScrollIndicatorBounds(@NonNull Rect out) {
        out.left = mScrollX;
        out.right = mScrollX + mRight - mLeft;
        out.top = mScrollY;
        out.bottom = mScrollY + mBottom - mTop;
    }

    private void onDrawScrollIndicators(Canvas c) {
        if ((mPrivateFlags3 & SCROLL_INDICATORS_PFLAG3_MASK) == 0) {
            // No scroll indicators enabled.
            return;
        }

        final Drawable dr = mScrollIndicatorDrawable;
        if (dr == null) {
            // Scroll indicators aren't supported here.
            return;
        }

        final int h = dr.getIntrinsicHeight();
        final int w = dr.getIntrinsicWidth();
        final Rect rect = mAttachInfo.mTmpInvalRect;
        getScrollIndicatorBounds(rect);

        if ((mPrivateFlags3 & PFLAG3_SCROLL_INDICATOR_TOP) != 0) {
            final boolean canScrollUp = canScrollVertically(-1);
            if (canScrollUp) {
                dr.setBounds(rect.left, rect.top, rect.right, rect.top + h);
                dr.draw(c);
            }
        }

        if ((mPrivateFlags3 & PFLAG3_SCROLL_INDICATOR_BOTTOM) != 0) {
            final boolean canScrollDown = canScrollVertically(1);
            if (canScrollDown) {
                dr.setBounds(rect.left, rect.bottom - h, rect.right, rect.bottom);
                dr.draw(c);
            }
        }

        final int leftRtl;
        final int rightRtl;
        if (getLayoutDirection() == LAYOUT_DIRECTION_RTL) {
            leftRtl = PFLAG3_SCROLL_INDICATOR_END;
            rightRtl = PFLAG3_SCROLL_INDICATOR_START;
        } else {
            leftRtl = PFLAG3_SCROLL_INDICATOR_START;
            rightRtl = PFLAG3_SCROLL_INDICATOR_END;
        }

        final int leftMask = PFLAG3_SCROLL_INDICATOR_LEFT | leftRtl;
        if ((mPrivateFlags3 & leftMask) != 0) {
            final boolean canScrollLeft = canScrollHorizontally(-1);
            if (canScrollLeft) {
                dr.setBounds(rect.left, rect.top, rect.left + w, rect.bottom);
                dr.draw(c);
            }
        }

        final int rightMask = PFLAG3_SCROLL_INDICATOR_RIGHT | rightRtl;
        if ((mPrivateFlags3 & rightMask) != 0) {
            final boolean canScrollRight = canScrollHorizontally(1);
            if (canScrollRight) {
                dr.setBounds(rect.right - w, rect.top, rect.right, rect.bottom);
                dr.draw(c);
            }
        }
    }

    private void getHorizontalScrollBarBounds(@Nullable Rect drawBounds,
            @Nullable Rect touchBounds) {
        final Rect bounds = drawBounds != null ? drawBounds : touchBounds;
        if (bounds == null) {
            return;
        }
        final int inside = (mViewFlags & SCROLLBARS_OUTSIDE_MASK) == 0 ? ~0 : 0;
        final boolean drawVerticalScrollBar = isVerticalScrollBarEnabled()
                && !isVerticalScrollBarHidden();
```

```
        final int size = getHorizontalScrollbarHeight();
        final int verticalScrollBarGap = drawVerticalScrollBar ?
                getVerticalScrollbarWidth() : 0;
        final int width = mRight - mLeft;
        final int height = mBottom - mTop;
        bounds.top = mScrollY + height - size - (mUserPaddingBottom & inside);
        bounds.left = mScrollX + (mPaddingLeft & inside);
        bounds.right = mScrollX + width - (mUserPaddingRight & inside) - verticalScrollBarGap;
        bounds.bottom = bounds.top + size;

        if (touchBounds == null) {
            return;
        }
        if (touchBounds != bounds) {
            touchBounds.set(bounds);
        }
        final int minTouchTarget = mScrollCache.scrollBarMinTouchTarget;
        if (touchBounds.height() < minTouchTarget) {
            final int adjust = (minTouchTarget - touchBounds.height()) / 2;
            touchBounds.bottom = Math.min(touchBounds.bottom + adjust, mScrollY + height);
            touchBounds.top = touchBounds.bottom - minTouchTarget;
        }
        if (touchBounds.width() < minTouchTarget) {
            final int adjust = (minTouchTarget - touchBounds.width()) / 2;
            touchBounds.left -= adjust;
            touchBounds.right = touchBounds.left + minTouchTarget;
        }
    }

    private void getVerticalScrollBarBounds(@Nullable Rect bounds, @Nullable Rect touchBounds) {
        if (mRoundScrollbarRenderer == null) {
            getStraightVerticalScrollBarBounds(bounds, touchBounds);
        } else {
            getRoundVerticalScrollBarBounds(bounds != null ? bounds : touchBounds);
        }
    }

    private void getRoundVerticalScrollBarBounds(Rect bounds) {
        final int width = mRight - mLeft;
        final int height = mBottom - mTop;
        // Do not take padding into account as we always want the scrollbars
        // to hug the screen for round wearable devices.
        bounds.left = mScrollX;
        bounds.top = mScrollY;
        bounds.right = bounds.left + width;
        bounds.bottom = mScrollY + height;
    }

    private void getStraightVerticalScrollBarBounds(@Nullable Rect drawBounds,
            @Nullable Rect touchBounds) {
        final Rect bounds = drawBounds != null ? drawBounds : touchBounds;
        if (bounds == null) {
            return;
        }
        final int inside = (mViewFlags & SCROLLBARS_OUTSIDE_MASK) == 0 ? ~0 : 0;
        final int size = getVerticalScrollbarWidth();
        int verticalScrollbarPosition = mVerticalScrollbarPosition;
        if (verticalScrollbarPosition == SCROLLBAR_POSITION_DEFAULT) {
            verticalScrollbarPosition = isLayoutRtl() ?
                    SCROLLBAR_POSITION_LEFT : SCROLLBAR_POSITION_RIGHT;
        }
        final int width = mRight - mLeft;
        final int height = mBottom - mTop;
        switch (verticalScrollbarPosition) {
            default:
            case SCROLLBAR_POSITION_RIGHT:
                bounds.left = mScrollX + width - size - (mUserPaddingRight & inside);
                break;
            case SCROLLBAR_POSITION_LEFT:
                bounds.left = mScrollX + (mUserPaddingLeft & inside);
                break;
        }
        bounds.top = mScrollY + (mPaddingTop & inside);
        bounds.right = bounds.left + size;
        bounds.bottom = mScrollY + height - (mUserPaddingBottom & inside);

        if (touchBounds == null) {
            return;
        }
        if (touchBounds != bounds) {
            touchBounds.set(bounds);
        }
```

```
    final int minTouchTarget = mScrollCache.scrollBarMinTouchTarget;
    if (touchBounds.width() < minTouchTarget) {
        final int adjust = (minTouchTarget - touchBounds.width()) / 2;
        if (verticalScrollbarPosition == SCROLLBAR_POSITION_RIGHT) {
            touchBounds.right = Math.min(touchBounds.right + adjust, mScrollX + width);
            touchBounds.left = touchBounds.right - minTouchTarget;
        } else {
            touchBounds.left = Math.max(touchBounds.left + adjust, mScrollX);
            touchBounds.right = touchBounds.left + minTouchTarget;
        }
    }
    if (touchBounds.height() < minTouchTarget) {
        final int adjust = (minTouchTarget - touchBounds.height()) / 2;
        touchBounds.top -= adjust;
        touchBounds.bottom = touchBounds.top + minTouchTarget;
    }
}

/**
 * <p>Request the drawing of the horizontal and the vertical scrollbar. The
 * scrollbars are painted only if they have been awakened first.</p>
 *
 * @param canvas the canvas on which to draw the scrollbars
 *
 * @see #awakenScrollBars(int)
 */
protected final void onDrawScrollBars(Canvas canvas) {
    // scrollbars are drawn only when the animation is running
    final ScrollabilityCache cache = mScrollCache;

    if (cache != null) {

        int state = cache.state;

        if (state == ScrollabilityCache.OFF) {
            return;
        }

        boolean invalidate = false;

        if (state == ScrollabilityCache.FADING) {
            // We're fading -- get our fade interpolation
            if (cache.interpolatorValues == null) {
                cache.interpolatorValues = new float[1];
            }

            float[] values = cache.interpolatorValues;

            // Stops the animation if we're done
            if (cache.scrollBarInterpolator.timeToValues(values) ==
                    Interpolator.Result.FREEZE_END) {
                cache.state = ScrollabilityCache.OFF;
            } else {
                cache.scrollBar.mutate().setAlpha(Math.round(values[0]));
            }

            // This will make the scroll bars inval themselves after
            // drawing. We only want this when we're fading so that
            // we prevent excessive redraws
            invalidate = true;
        } else {
            // We're just on -- but we may have been fading before so
            // reset alpha
            cache.scrollBar.mutate().setAlpha(255);
        }

        final boolean drawHorizontalScrollBar = isHorizontalScrollBarEnabled();
        final boolean drawVerticalScrollBar = isVerticalScrollBarEnabled()
                && !isVerticalScrollBarHidden();

        // Fork out the scroll bar drawing for round wearable devices.
        if (mRoundScrollbarRenderer != null) {
            if (drawVerticalScrollBar) {
                final Rect bounds = cache.mScrollBarBounds;
                getVerticalScrollBarBounds(bounds, null);
                mRoundScrollbarRenderer.drawRoundScrollbars(
                        canvas, (float) cache.scrollBar.getAlpha() / 255f, bounds);
                if (invalidate) {
                    invalidate();
                }
            }
            // Do not draw horizontal scroll bars for round wearable devices.
```

```
        } else if (drawVerticalScrollBar || drawHorizontalScrollBar) {
            final ScrollBarDrawable scrollBar = cache.scrollBar;

            if (drawHorizontalScrollBar) {
                scrollBar.setParameters(computeHorizontalScrollRange(),
                        computeHorizontalScrollOffset(),
                        computeHorizontalScrollExtent(), false);
                final Rect bounds = cache.mScrollBarBounds;
                getHorizontalScrollBarBounds(bounds, null);
                onDrawHorizontalScrollBar(canvas, scrollBar, bounds.left, bounds.top,
                        bounds.right, bounds.bottom);
                if (invalidate) {
                    invalidate(bounds);
                }
            }

            if (drawVerticalScrollBar) {
                scrollBar.setParameters(computeVerticalScrollRange(),
                        computeVerticalScrollOffset(),
                        computeVerticalScrollExtent(), true);
                final Rect bounds = cache.mScrollBarBounds;
                getVerticalScrollBarBounds(bounds, null);
                onDrawVerticalScrollBar(canvas, scrollBar, bounds.left, bounds.top,
                        bounds.right, bounds.bottom);
                if (invalidate) {
                    invalidate(bounds);
                }
            }
        }
    }
}

/**
 * Override this if the vertical scrollbar needs to be hidden in a subclass, like when
 * FastScroller is visible.
 * @return whether to temporarily hide the vertical scrollbar
 * @hide
 */
protected boolean isVerticalScrollBarHidden() {
    return false;
}

/**
 * <p>Draw the horizontal scrollbar if
 * {@link #isHorizontalScrollBarEnabled()} returns true.</p>
 *
 * @param canvas the canvas on which to draw the scrollbar
 * @param scrollBar the scrollbar's drawable
 *
 * @see #isHorizontalScrollBarEnabled()
 * @see #computeHorizontalScrollRange()
 * @see #computeHorizontalScrollExtent()
 * @see #computeHorizontalScrollOffset()
 * @see android.widget.ScrollBarDrawable
 * @hide
 */
protected void onDrawHorizontalScrollBar(Canvas canvas, Drawable scrollBar,
        int l, int t, int r, int b) {
    scrollBar.setBounds(l, t, r, b);
    scrollBar.draw(canvas);
}

/**
 * <p>Draw the vertical scrollbar if {@link #isVerticalScrollBarEnabled()}
 * returns true.</p>
 *
 * @param canvas the canvas on which to draw the scrollbar
 * @param scrollBar the scrollbar's drawable
 *
 * @see #isVerticalScrollBarEnabled()
 * @see #computeVerticalScrollRange()
 * @see #computeVerticalScrollExtent()
 * @see #computeVerticalScrollOffset()
 * @see android.widget.ScrollBarDrawable
 * @hide
 */
protected void onDrawVerticalScrollBar(Canvas canvas, Drawable scrollBar,
        int l, int t, int r, int b) {
    scrollBar.setBounds(l, t, r, b);
    scrollBar.draw(canvas);
}
```

```
/**
 * Implement this to do your drawing.
 *
 * @param canvas the canvas on which the background will be drawn
 */
protected void onDraw(Canvas canvas) {
}

/*
 * Caller is responsible for calling requestLayout if necessary.
 * (This allows addViewInLayout to not request a new layout.)
 */
void assignParent(ViewParent parent) {
    if (mParent == null) {
        mParent = parent;
    } else if (parent == null) {
        mParent = null;
    } else {
        throw new RuntimeException("view " + this + " being added, but"
                + " it already has a parent");
    }
}

/**
 * This is called when the view is attached to a window.  At this point it
 * has a Surface and will start drawing.  Note that this function is
 * guaranteed to be called before {@link #onDraw(android.graphics.Canvas)},
 * however it may be called any time before the first onDraw -- including
 * before or after {@link #onMeasure(int, int)}.
 *
 * @see #onDetachedFromWindow()
 */
@CallSuper
protected void onAttachedToWindow() {
    if ((mPrivateFlags & PFLAG_REQUEST_TRANSPARENT_REGIONS) != 0) {
        mParent.requestTransparentRegion(this);
    }

    mPrivateFlags3 &= ~PFLAG3_IS_LAID_OUT;

    jumpDrawablesToCurrentState();

    resetSubtreeAccessibilityStateChanged();

    // rebuild, since Outline not maintained while View is detached
    rebuildOutline();

    if (isFocused()) {
        InputMethodManager imm = InputMethodManager.peekInstance();
        if (imm != null) {
            imm.focusIn(this);
        }
    }
}

/**
 * Resolve all RTL related properties.
 *
 * @return true if resolution of RTL properties has been done
 *
 * @hide
 */
public boolean resolveRtlPropertiesIfNeeded() {
    if (!needRtlPropertiesResolution()) return false;

    // Order is important here: LayoutDirection MUST be resolved first
    if (!isLayoutDirectionResolved()) {
        resolveLayoutDirection();
        resolveLayoutParams();
    }
    // ... then we can resolve the others properties depending on the resolved LayoutDirection.
    if (!isTextDirectionResolved()) {
        resolveTextDirection();
    }
    if (!isTextAlignmentResolved()) {
        resolveTextAlignment();
    }
    // Should resolve Drawables before Padding because we need the layout direction of the
    // Drawable to correctly resolve Padding.
    if (!areDrawablesResolved()) {
        resolveDrawables();
    }
```

```
    if (!isPaddingResolved()) {
        resolvePadding();
    }
    onRtlPropertiesChanged(getLayoutDirection());
    return true;
}

/**
 * Reset resolution of all RTL related properties.
 *
 * @hide
 */
public void resetRtlProperties() {
    resetResolvedLayoutDirection();
    resetResolvedTextDirection();
    resetResolvedTextAlignment();
    resetResolvedPadding();
    resetResolvedDrawables();
}

/**
 * @see #onScreenStateChanged(int)
 */
void dispatchScreenStateChanged(int screenState) {
    onScreenStateChanged(screenState);
}

/**
 * This method is called whenever the state of the screen this view is
 * attached to changes. A state change will usually occurs when the screen
 * turns on or off (whether it happens automatically or the user does it
 * manually.)
 *
 * @param screenState The new state of the screen. Can be either
 *                    {@link #SCREEN_STATE_ON} or {@link #SCREEN_STATE_OFF}
 */
public void onScreenStateChanged(int screenState) {
}

/**
 * @see #onMovedToDisplay(int, Configuration)
 */
void dispatchMovedToDisplay(Display display, Configuration config) {
    mAttachInfo.mDisplay = display;
    mAttachInfo.mDisplayState = display.getState();
    onMovedToDisplay(display.getDisplayId(), config);
}

/**
 * Called by the system when the hosting activity is moved from one display to another without
 * recreation. This means that the activity is declared to handle all changes to configuration
 * that happened when it was switched to another display, so it wasn't destroyed and created
 * again.
 *
 * <p>This call will be followed by {@link #onConfigurationChanged(Configuration)} if the
 * applied configuration actually changed. It is up to app developer to choose whether to handle
 * the change in this method or in the following {@link #onConfigurationChanged(Configuration)}
 * call.
 *
 * <p>Use this callback to track changes to the displays if some functionality relies on an
 * association with some display properties.
 *
 * @param displayId The id of the display to which the view was moved.
 * @param config Configuration of the resources on new display after move.
 *
 * @see #onConfigurationChanged(Configuration)
 * @hide
 */
public void onMovedToDisplay(int displayId, Configuration config) {
}

/**
 * Return true if the application tag in the AndroidManifest has set "supportRtl" to true
 */
private boolean hasRtlSupport() {
    return mContext.getApplicationInfo().hasRtlSupport();
}

/**
 * Return true if we are in RTL compatibility mode (either before Jelly Bean MR1 or
 * RTL not supported)
 */
```

```java
private boolean isRtlCompatibilityMode() {
    final int targetSdkVersion = getContext().getApplicationInfo().targetSdkVersion;
    return targetSdkVersion < Build.VERSION_CODES.JELLY_BEAN_MR1 || !hasRtlSupport();
}

/**
 * @return true if RTL properties need resolution.
 *
 */
private boolean needRtlPropertiesResolution() {
    return (mPrivateFlags2 & ALL_RTL_PROPERTIES_RESOLVED) != ALL_RTL_PROPERTIES_RESOLVED;
}

/**
 * Called when any RTL property (layout direction or text direction or text alignment) has
 * been changed.
 *
 * Subclasses need to override this method to take care of cached information that depends on the
 * resolved layout direction, or to inform child views that inherit their layout direction.
 *
 * The default implementation does nothing.
 *
 * @param layoutDirection the direction of the layout
 *
 * @see #LAYOUT_DIRECTION_LTR
 * @see #LAYOUT_DIRECTION_RTL
 */
public void onRtlPropertiesChanged(@ResolvedLayoutDir int layoutDirection) {
}

/**
 * Resolve and cache the layout direction. LTR is set initially. This is implicitly supposing
 * that the parent directionality can and will be resolved before its children.
 *
 * @return true if resolution has been done, false otherwise.
 *
 * @hide
 */
public boolean resolveLayoutDirection() {
    // Clear any previous layout direction resolution
    mPrivateFlags2 &= ~PFLAG2_LAYOUT_DIRECTION_RESOLVED_MASK;

    if (hasRtlSupport()) {
        // Set resolved depending on layout direction
        switch ((mPrivateFlags2 & PFLAG2_LAYOUT_DIRECTION_MASK) >>
                PFLAG2_LAYOUT_DIRECTION_MASK_SHIFT) {
            case LAYOUT_DIRECTION_INHERIT:
                // We cannot resolve yet. LTR is by default and let the resolution happen again
                // later to get the correct resolved value
                if (!canResolveLayoutDirection()) return false;

                // Parent has not yet resolved, LTR is still the default
                try {
                    if (!mParent.isLayoutDirectionResolved()) return false;

                    if (mParent.getLayoutDirection() == LAYOUT_DIRECTION_RTL) {
                        mPrivateFlags2 |= PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL;
                    }
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
                break;
            case LAYOUT_DIRECTION_RTL:
                mPrivateFlags2 |= PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL;
                break;
            case LAYOUT_DIRECTION_LOCALE:
                if((LAYOUT_DIRECTION_RTL ==
                        TextUtils.getLayoutDirectionFromLocale(Locale.getDefault()))) {
                    mPrivateFlags2 |= PFLAG2_LAYOUT_DIRECTION_RESOLVED_RTL;
                }
                break;
            default:
                // Nothing to do, LTR by default
        }
    }

    // Set to resolved
    mPrivateFlags2 |= PFLAG2_LAYOUT_DIRECTION_RESOLVED;
    return true;
}
```

```
/**
 * Check if layout direction resolution can be done.
 *
 * @return true if layout direction resolution can be done otherwise return false.
 */
public boolean canResolveLayoutDirection() {
    switch (getRawLayoutDirection()) {
        case LAYOUT_DIRECTION_INHERIT:
            if (mParent != null) {
                try {
                    return mParent.canResolveLayoutDirection();
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
            }
            return false;

        default:
            return true;
    }
}

/**
 * Reset the resolved layout direction. Layout direction will be resolved during a call to
 * {@link #onMeasure(int, int)}.
 *
 * @hide
 */
public void resetResolvedLayoutDirection() {
    // Reset the current resolved bits
    mPrivateFlags2 &= ~PFLAG2_LAYOUT_DIRECTION_RESOLVED_MASK;
}

/**
 * @return true if the layout direction is inherited.
 *
 * @hide
 */
public boolean isLayoutDirectionInherited() {
    return (getRawLayoutDirection() == LAYOUT_DIRECTION_INHERIT);
}

/**
 * @return true if layout direction has been resolved.
 */
public boolean isLayoutDirectionResolved() {
    return (mPrivateFlags2 & PFLAG2_LAYOUT_DIRECTION_RESOLVED) == PFLAG2_LAYOUT_DIRECTION_RESOLVED;
}

/**
 * Return if padding has been resolved
 *
 * @hide
 */
boolean isPaddingResolved() {
    return (mPrivateFlags2 & PFLAG2_PADDING_RESOLVED) == PFLAG2_PADDING_RESOLVED;
}

/**
 * Resolves padding depending on layout direction, if applicable, and
 * recomputes internal padding values to adjust for scroll bars.
 *
 * @hide
 */
public void resolvePadding() {
    final int resolvedLayoutDirection = getLayoutDirection();

    if (!isRtlCompatibilityMode()) {
        // Post Jelly Bean MR1 case: we need to take the resolved layout direction into account.
        // If start / end padding are defined, they will be resolved (hence overriding) to
        // left / right or right / left depending on the resolved layout direction.
        // If start / end padding are not defined, use the left / right ones.
        if (mBackground != null && (!mLeftPaddingDefined || !mRightPaddingDefined)) {
            Rect padding = sThreadLocal.get();
            if (padding == null) {
                padding = new Rect();
                sThreadLocal.set(padding);
            }
            mBackground.getPadding(padding);
            if (!mLeftPaddingDefined) {
                mUserPaddingLeftInitial = padding.left;
```

```
            }
            if (!mRightPaddingDefined) {
                mUserPaddingRightInitial = padding.right;
            }
        }
        switch (resolvedLayoutDirection) {
            case LAYOUT_DIRECTION_RTL:
                if (mUserPaddingStart != UNDEFINED_PADDING) {
                    mUserPaddingRight = mUserPaddingStart;
                } else {
                    mUserPaddingRight = mUserPaddingRightInitial;
                }
                if (mUserPaddingEnd != UNDEFINED_PADDING) {
                    mUserPaddingLeft = mUserPaddingEnd;
                } else {
                    mUserPaddingLeft = mUserPaddingLeftInitial;
                }
                break;
            case LAYOUT_DIRECTION_LTR:
            default:
                if (mUserPaddingStart != UNDEFINED_PADDING) {
                    mUserPaddingLeft = mUserPaddingStart;
                } else {
                    mUserPaddingLeft = mUserPaddingLeftInitial;
                }
                if (mUserPaddingEnd != UNDEFINED_PADDING) {
                    mUserPaddingRight = mUserPaddingEnd;
                } else {
                    mUserPaddingRight = mUserPaddingRightInitial;
                }
        }

        mUserPaddingBottom = (mUserPaddingBottom >= 0) ? mUserPaddingBottom : mPaddingBottom;
    }

    internalSetPadding(mUserPaddingLeft, mPaddingTop, mUserPaddingRight, mUserPaddingBottom);
    onRtlPropertiesChanged(resolvedLayoutDirection);

    mPrivateFlags2 |= PFLAG2_PADDING_RESOLVED;
}

/**
 * Reset the resolved layout direction.
 *
 * @hide
 */
public void resetResolvedPadding() {
    resetResolvedPaddingInternal();
}

/**
 * Used when we only want to reset *this* view's padding and not trigger overrides
 * in ViewGroup that reset children too.
 */
void resetResolvedPaddingInternal() {
    mPrivateFlags2 &= ~PFLAG2_PADDING_RESOLVED;
}

/**
 * This is called when the view is detached from a window.  At this point it
 * no longer has a surface for drawing.
 *
 * @see #onAttachedToWindow()
 */
@CallSuper
protected void onDetachedFromWindow() {
}

/**
 * This is a framework-internal mirror of onDetachedFromWindow() that's called
 * after onDetachedFromWindow().
 *
 * If you override this you *MUST* call super.onDetachedFromWindowInternal()!
 * The super method should be called at the end of the overridden method to ensure
 * subclasses are destroyed first
 *
 * @hide
 */
@CallSuper
protected void onDetachedFromWindowInternal() {
    mPrivateFlags &= ~PFLAG_CANCEL_NEXT_UP_EVENT;
    mPrivateFlags3 &= ~PFLAG3_IS_LAID_OUT;
```

```
    mPrivateFlags3 &= ~PFLAG3_TEMPORARY_DETACH;

    removeUnsetPressCallback();
    removeLongPressCallback();
    removePerformClickCallback();
    removeSendViewScrolledAccessibilityEventCallback();
    stopNestedScroll();

    // Anything that started animating right before detach should already
    // be in its final state when re-attached.
    jumpDrawablesToCurrentState();

    destroyDrawingCache();

    cleanupDraw();
    mCurrentAnimation = null;

    if ((mViewFlags & TOOLTIP) == TOOLTIP) {
        hideTooltip();
    }
}

private void cleanupDraw() {
    resetDisplayList();
    if (mAttachInfo != null) {
        mAttachInfo.mViewRootImpl.cancelInvalidate(this);
    }
}

void invalidateInheritedLayoutMode(int layoutModeOfRoot) {
}

/**
 * @return The number of times this view has been attached to a window
 */
protected int getWindowAttachCount() {
    return mWindowAttachCount;
}

/**
 * Retrieve a unique token identifying the window this view is attached to.
 * @return Return the window's token for use in
 * {@link WindowManager.LayoutParams#token WindowManager.LayoutParams.token}.
 */
public IBinder getWindowToken() {
    return mAttachInfo != null ? mAttachInfo.mWindowToken : null;
}

/**
 * Retrieve the {@link WindowId} for the window this view is
 * currently attached to.
 */
public WindowId getWindowId() {
    if (mAttachInfo == null) {
        return null;
    }
    if (mAttachInfo.mWindowId == null) {
        try {
            mAttachInfo.mIWindowId = mAttachInfo.mSession.getWindowId(
                    mAttachInfo.mWindowToken);
            mAttachInfo.mWindowId = new WindowId(
                    mAttachInfo.mIWindowId);
        } catch (RemoteException e) {
        }
    }
    return mAttachInfo.mWindowId;
}

/**
 * Retrieve a unique token identifying the top-level "real" window of
 * the window that this view is attached to.  That is, this is like
 * {@link #getWindowToken}, except if the window this view in is a panel
 * window (attached to another containing window), then the token of
 * the containing window is returned instead.
 *
 * @return Returns the associated window token, either
 * {@link #getWindowToken()} or the containing window's token.
 */
public IBinder getApplicationWindowToken() {
    AttachInfo ai = mAttachInfo;
    if (ai != null) {
        IBinder appWindowToken = ai.mPanelParentWindowToken;
```

```
        if (appWindowToken == null) {
            appWindowToken = ai.mWindowToken;
        }
        return appWindowToken;
    }
    return null;
}

/**
 * Gets the logical display to which the view's window has been attached.
 *
 * @return The logical display, or null if the view is not currently attached to a window.
 */
public Display getDisplay() {
    return mAttachInfo != null ? mAttachInfo.mDisplay : null;
}

/**
 * Retrieve private session object this view hierarchy is using to
 * communicate with the window manager.
 * @return the session object to communicate with the window manager
 */
/*package*/ IWindowSession getWindowSession() {
    return mAttachInfo != null ? mAttachInfo.mSession : null;
}

/**
 * Return the visibility value of the least visible component passed.
 */
int combineVisibility(int vis1, int vis2) {
    // This works because VISIBLE < INVISIBLE < GONE.
    return Math.max(vis1, vis2);
}

/**
 * @param info the {@link android.view.View.AttachInfo} to associated with
 *        this view
 */
void dispatchAttachedToWindow(AttachInfo info, int visibility) {
    mAttachInfo = info;
    if (mOverlay != null) {
        mOverlay.getOverlayView().dispatchAttachedToWindow(info, visibility);
    }
    mWindowAttachCount++;
    // We will need to evaluate the drawable state at least once.
    mPrivateFlags |= PFLAG_DRAWABLE_STATE_DIRTY;
    if (mFloatingTreeObserver != null) {
        info.mTreeObserver.merge(mFloatingTreeObserver);
        mFloatingTreeObserver = null;
    }

    registerPendingFrameMetricsObservers();

    if ((mPrivateFlags&PFLAG_SCROLL_CONTAINER) != 0) {
        mAttachInfo.mScrollContainers.add(this);
        mPrivateFlags |= PFLAG_SCROLL_CONTAINER_ADDED;
    }
    // Transfer all pending runnables.
    if (mRunQueue != null) {
        mRunQueue.executeActions(info.mHandler);
        mRunQueue = null;
    }
    performCollectViewAttributes(mAttachInfo, visibility);
    onAttachedToWindow();

    ListenerInfo li = mListenerInfo;
    final CopyOnWriteArrayList<OnAttachStateChangeListener> listeners =
            li != null ? li.mOnAttachStateChangeListeners : null;
    if (listeners != null && listeners.size() > 0) {
        // NOTE: because of the use of CopyOnWriteArrayList, we *must* use an iterator to
        // perform the dispatching. The iterator is a safe guard against listeners that
        // could mutate the list by calling the various add/remove methods. This prevents
        // the array from being modified while we iterate it.
        for (OnAttachStateChangeListener listener : listeners) {
            listener.onViewAttachedToWindow(this);
        }
    }

    int vis = info.mWindowVisibility;
    if (vis != GONE) {
        onWindowVisibilityChanged(vis);
        if (isShown()) {
```

```
            // Calling onVisibilityAggregated directly here since the subtree will also
            // receive dispatchAttachedToWindow and this same call
            onVisibilityAggregated(vis == VISIBLE);
        }
    }

    // Send onVisibilityChanged directly instead of dispatchVisibilityChanged.
    // As all views in the subtree will already receive dispatchAttachedToWindow
    // traversing the subtree again here is not desired.
    onVisibilityChanged(this, visibility);

    if ((mPrivateFlags&PFLAG_DRAWABLE_STATE_DIRTY) != 0) {
        // If nobody has evaluated the drawable state yet, then do it now.
        refreshDrawableState();
    }
    needGlobalAttributesUpdate(false);

    notifyEnterOrExitForAutoFillIfNeeded(true);
}

void dispatchDetachedFromWindow() {
    AttachInfo info = mAttachInfo;
    if (info != null) {
        int vis = info.mWindowVisibility;
        if (vis != GONE) {
            onWindowVisibilityChanged(GONE);
            if (isShown()) {
                // Invoking onVisibilityAggregated directly here since the subtree
                // will also receive detached from window
                onVisibilityAggregated(false);
            }
        }
    }

    onDetachedFromWindow();
    onDetachedFromWindowInternal();

    InputMethodManager imm = InputMethodManager.peekInstance();
    if (imm != null) {
        imm.onViewDetachedFromWindow(this);
    }

    ListenerInfo li = mListenerInfo;
    final CopyOnWriteArrayList<OnAttachStateChangeListener> listeners =
            li != null ? li.mOnAttachStateChangeListeners : null;
    if (listeners != null && listeners.size() > 0) {
        // NOTE: because of the use of CopyOnWriteArrayList, we *must* use an iterator to
        // perform the dispatching. The iterator is a safe guard against listeners that
        // could mutate the list by calling the various add/remove methods. This prevents
        // the array from being modified while we iterate it.
        for (OnAttachStateChangeListener listener : listeners) {
            listener.onViewDetachedFromWindow(this);
        }
    }

    if ((mPrivateFlags & PFLAG_SCROLL_CONTAINER_ADDED) != 0) {
        mAttachInfo.mScrollContainers.remove(this);
        mPrivateFlags &= ~PFLAG_SCROLL_CONTAINER_ADDED;
    }

    mAttachInfo = null;
    if (mOverlay != null) {
        mOverlay.getOverlayView().dispatchDetachedFromWindow();
    }

    notifyEnterOrExitForAutoFillIfNeeded(false);
}

/**
 * Cancel any deferred high-level input events that were previously posted to the event queue.
 *
 * <p>Many views post high-level events such as click handlers to the event queue
 * to run deferred in order to preserve a desired user experience - clearing visible
 * pressed states before executing, etc. This method will abort any events of this nature
 * that are currently in flight.</p>
 *
 * <p>Custom views that generate their own high-level deferred input events should override
 * {@Link #onCancelPendingInputEvents()} and remove those pending events from the queue.</p>
 *
 * <p>This will also cancel pending input events for any child views.</p>
 *
 * <p>Note that this may not be sufficient as a debouncing strategy for clicks in all cases.
```

```
     * This will not impact newer events posted after this call that may occur as a result of
     * lower-level input events still waiting in the queue. If you are trying to prevent
     * double-submitted  events for the duration of some sort of asynchronous transaction
     * you should also take other steps to protect against unexpected double inputs e.g. calling
     * {@link #setEnabled(boolean) setEnabled(false)} and re-enabling the view when
     * the transaction completes, tracking already submitted transaction IDs, etc.</p>
     */
    public final void cancelPendingInputEvents() {
        dispatchCancelPendingInputEvents();
    }

    /**
     * Called by {@link #cancelPendingInputEvents()} to cancel input events in flight.
     * Overridden by ViewGroup to dispatch. Package scoped to prevent app-side meddling.
     */
    void dispatchCancelPendingInputEvents() {
        mPrivateFlags3 &= ~PFLAG3_CALLED_SUPER;
        onCancelPendingInputEvents();
        if ((mPrivateFlags3 & PFLAG3_CALLED_SUPER) != PFLAG3_CALLED_SUPER) {
            throw new SuperNotCalledException("View " + getClass().getSimpleName() +
                    " did not call through to super.onCancelPendingInputEvents()");
        }
    }

    /**
     * Called as the result of a call to {@link #cancelPendingInputEvents()} on this view or
     * a parent view.
     *
     * <p>This method is responsible for removing any pending high-level input events that were
     * posted to the event queue to run later. Custom view classes that post their own deferred
     * high-level events via {@link #post(Runnable)}, {@link #postDelayed(Runnable, long)} or
     * {@link android.os.Handler} should override this method, call
     * <code>super.onCancelPendingInputEvents()</code> and remove those callbacks as appropriate.
     * </p>
     */
    public void onCancelPendingInputEvents() {
        removePerformClickCallback();
        cancelLongPress();
        mPrivateFlags3 |= PFLAG3_CALLED_SUPER;
    }

    /**
     * Store this view hierarchy's frozen state into the given container.
     *
     * @param container The SparseArray in which to save the view's state.
     *
     * @see #restoreHierarchyState(android.util.SparseArray)
     * @see #dispatchSaveInstanceState(android.util.SparseArray)
     * @see #onSaveInstanceState()
     */
    public void saveHierarchyState(SparseArray<Parcelable> container) {
        dispatchSaveInstanceState(container);
    }

    /**
     * Called by {@link #saveHierarchyState(android.util.SparseArray)} to store the state for
     * this view and its children. May be overridden to modify how freezing happens to a
     * view's children; for example, some views may want to not store state for their children.
     *
     * @param container The SparseArray in which to save the view's state.
     *
     * @see #dispatchRestoreInstanceState(android.util.SparseArray)
     * @see #saveHierarchyState(android.util.SparseArray)
     * @see #onSaveInstanceState()
     */
    protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
        if (mID != NO_ID && (mViewFlags & SAVE_DISABLED_MASK) == 0) {
            mPrivateFlags &= ~PFLAG_SAVE_STATE_CALLED;
            Parcelable state = onSaveInstanceState();
            if ((mPrivateFlags & PFLAG_SAVE_STATE_CALLED) == 0) {
                throw new IllegalStateException(
                        "Derived class did not call super.onSaveInstanceState()");
            }
            if (state != null) {
                // Log.i("View", "Freezing #" + Integer.toHexString(mID)
                // + ": " + state);
                container.put(mID, state);
            }
        }
    }

    /**
```

```
     * Hook allowing a view to generate a representation of its internal state
     * that can later be used to create a new instance with that same state.
     * This state should only contain information that is not persistent or can
     * not be reconstructed later. For example, you will never store your
     * current position on screen because that will be computed again when a
     * new instance of the view is placed in its view hierarchy.
     * <p>
     * Some examples of things you may store here: the current cursor position
     * in a text view (but usually not the text itself since that is stored in a
     * content provider or other persistent storage), the currently selected
     * item in a list view.
     *
     * @return Returns a Parcelable object containing the view's current dynamic
     *         state, or null if there is nothing interesting to save.
     * @see #onRestoreInstanceState(Parcelable)
     * @see #saveHierarchyState(SparseArray)
     * @see #dispatchSaveInstanceState(SparseArray)
     * @see #setSaveEnabled(boolean)
     */
    @CallSuper
    @Nullable protected Parcelable onSaveInstanceState() {
        mPrivateFlags |= PFLAG_SAVE_STATE_CALLED;
        if (mStartActivityRequestWho != null || isAutofilled()
                || mAutofillViewId > LAST_APP_AUTOFILL_ID) {
            BaseSavedState state = new BaseSavedState(AbsSavedState.EMPTY_STATE);

            if (mStartActivityRequestWho != null) {
                state.mSavedData |= BaseSavedState.START_ACTIVITY_REQUESTED_WHO_SAVED;
            }

            if (isAutofilled()) {
                state.mSavedData |= BaseSavedState.IS_AUTOFILLED;
            }

            if (mAutofillViewId > LAST_APP_AUTOFILL_ID) {
                state.mSavedData |= BaseSavedState.AUTOFILL_ID;
            }

            state.mStartActivityRequestWhoSaved = mStartActivityRequestWho;
            state.mIsAutofilled = isAutofilled();
            state.mAutofillViewId = mAutofillViewId;
            return state;
        }
        return BaseSavedState.EMPTY_STATE;
    }

    /**
     * Restore this view hierarchy's frozen state from the given container.
     *
     * @param container The SparseArray which holds previously frozen states.
     *
     * @see #saveHierarchyState(android.util.SparseArray)
     * @see #dispatchRestoreInstanceState(android.util.SparseArray)
     * @see #onRestoreInstanceState(android.os.Parcelable)
     */
    public void restoreHierarchyState(SparseArray<Parcelable> container) {
        dispatchRestoreInstanceState(container);
    }

    /**
     * Called by {@link #restoreHierarchyState(android.util.SparseArray)} to retrieve the
     * state for this view and its children. May be overridden to modify how restoring
     * happens to a view's children; for example, some views may want to not store state
     * for their children.
     *
     * @param container The SparseArray which holds previously saved state.
     *
     * @see #dispatchSaveInstanceState(android.util.SparseArray)
     * @see #restoreHierarchyState(android.util.SparseArray)
     * @see #onRestoreInstanceState(android.os.Parcelable)
     */
    protected void dispatchRestoreInstanceState(SparseArray<Parcelable> container) {
        if (mID != NO_ID) {
            Parcelable state = container.get(mID);
            if (state != null) {
                // Log.i("View", "Restoreing #" + Integer.toHexString(mID)
                // + ": " + state);
                mPrivateFlags &= ~PFLAG_SAVE_STATE_CALLED;
                onRestoreInstanceState(state);
                if ((mPrivateFlags & PFLAG_SAVE_STATE_CALLED) == 0) {
                    throw new IllegalStateException(
                            "Derived class did not call super.onRestoreInstanceState()");
```

```
            }
        }
    }
}

/**
 * Hook allowing a view to re-apply a representation of its internal state that had previously
 * been generated by {@link #onSaveInstanceState}. This function will never be called with a
 * null state.
 *
 * @param state The frozen state that had previously been returned by
 *        {@link #onSaveInstanceState}.
 *
 * @see #onSaveInstanceState()
 * @see #restoreHierarchyState(android.util.SparseArray)
 * @see #dispatchRestoreInstanceState(android.util.SparseArray)
 */
@CallSuper
protected void onRestoreInstanceState(Parcelable state) {
    mPrivateFlags |= PFLAG_SAVE_STATE_CALLED;
    if (state != null && !(state instanceof AbsSavedState)) {
        throw new IllegalArgumentException("Wrong state class, expecting View State but "
                + "received " + state.getClass().toString() + " instead. This usually happens "
                + "when two views of different type have the same id in the same hierarchy. "
                + "This view's id is " + ViewDebug.resolveId(mContext, getId()) + ". Make sure "
                + "other views do not use the same id.");
    }
    if (state != null && state instanceof BaseSavedState) {
        BaseSavedState baseState = (BaseSavedState) state;

        if ((baseState.mSavedData & BaseSavedState.START_ACTIVITY_REQUESTED_WHO_SAVED) != 0) {
            mStartActivityRequestWho = baseState.mStartActivityRequestWhoSaved;
        }
        if ((baseState.mSavedData & BaseSavedState.IS_AUTOFILLED) != 0) {
            setAutofilled(baseState.mIsAutofilled);
        }
        if ((baseState.mSavedData & BaseSavedState.AUTOFILL_ID) != 0) {
            // It can happen that views have the same view id and the restoration path will not
            // be able to distinguish between them. The autofill id needs to be unique though.
            // Hence prevent the same autofill view id from being restored multiple times.
            ((BaseSavedState) state).mSavedData &= ~BaseSavedState.AUTOFILL_ID;

            mAutofillViewId = baseState.mAutofillViewId;
        }
    }
}

/**
 * <p>Return the time at which the drawing of the view hierarchy started.</p>
 *
 * @return the drawing start time in milliseconds
 */
public long getDrawingTime() {
    return mAttachInfo != null ? mAttachInfo.mDrawingTime : 0;
}

/**
 * <p>Enables or disables the duplication of the parent's state into this view. When
 * duplication is enabled, this view gets its drawable state from its parent rather
 * than from its own internal properties.</p>
 *
 * <p>Note: in the current implementation, setting this property to true after the
 * view was added to a ViewGroup might have no effect at all. This property should
 * always be used from XML or set to true before adding this view to a ViewGroup.</p>
 *
 * <p>Note: if this view's parent addStateFromChildren property is enabled and this
 * property is enabled, an exception will be thrown.</p>
 *
 * <p>Note: if the child view uses and updates additional states which are unknown to the
 * parent, these states should not be affected by this method.</p>
 *
 * @param enabled True to enable duplication of the parent's drawable state, false
 *                to disable it.
 *
 * @see #getDrawableState()
 * @see #isDuplicateParentStateEnabled()
 */
public void setDuplicateParentStateEnabled(boolean enabled) {
    setFlags(enabled ? DUPLICATE_PARENT_STATE : 0, DUPLICATE_PARENT_STATE);
}

/**
```

```
     * <p>Indicates whether this duplicates its drawable state from its parent.</p>
     *
     * @return True if this view's drawable state is duplicated from the parent,
     *         false otherwise
     *
     * @see #getDrawableState()
     * @see #setDuplicateParentStateEnabled(boolean)
     */
    public boolean isDuplicateParentStateEnabled() {
        return (mViewFlags & DUPLICATE_PARENT_STATE) == DUPLICATE_PARENT_STATE;
    }

    /**
     * <p>Specifies the type of layer backing this view. The layer can be
     * {@link #LAYER_TYPE_NONE}, {@link #LAYER_TYPE_SOFTWARE} or
     * {@link #LAYER_TYPE_HARDWARE}.</p>
     *
     * <p>A layer is associated with an optional {@link android.graphics.Paint}
     * instance that controls how the layer is composed on screen. The following
     * properties of the paint are taken into account when composing the layer:</p>
     * <ul>
     * <li>{@link android.graphics.Paint#getAlpha() Translucency (alpha)}</li>
     * <li>{@link android.graphics.Paint#getXfermode() Blending mode}</li>
     * <li>{@link android.graphics.Paint#getColorFilter() Color filter}</li>
     * </ul>
     *
     * <p>If this view has an alpha value set to < 1.0 by calling
     * {@link #setAlpha(float)}, the alpha value of the layer's paint is superseded
     * by this view's alpha value.</p>
     *
     * <p>Refer to the documentation of {@link #LAYER_TYPE_NONE},
     * {@link #LAYER_TYPE_SOFTWARE} and {@link #LAYER_TYPE_HARDWARE}
     * for more information on when and how to use layers.</p>
     *
     * @param layerType The type of layer to use with this view, must be one of
     *        {@link #LAYER_TYPE_NONE}, {@link #LAYER_TYPE_SOFTWARE} or
     *        {@link #LAYER_TYPE_HARDWARE}
     * @param paint The paint used to compose the layer. This argument is optional
     *        and can be null. It is ignored when the layer type is
     *        {@link #LAYER_TYPE_NONE}
     *
     * @see #getLayerType()
     * @see #LAYER_TYPE_NONE
     * @see #LAYER_TYPE_SOFTWARE
     * @see #LAYER_TYPE_HARDWARE
     * @see #setAlpha(float)
     *
     * @attr ref android.R.styleable#View_layerType
     */
    public void setLayerType(int layerType, @Nullable Paint paint) {
        if (layerType < LAYER_TYPE_NONE || layerType > LAYER_TYPE_HARDWARE) {
            throw new IllegalArgumentException("Layer type can only be one of: LAYER_TYPE_NONE, "
                    + "LAYER_TYPE_SOFTWARE or LAYER_TYPE_HARDWARE");
        }

        boolean typeChanged = mRenderNode.setLayerType(layerType);

        if (!typeChanged) {
            setLayerPaint(paint);
            return;
        }

        if (layerType != LAYER_TYPE_SOFTWARE) {
            // Destroy any previous software drawing cache if present
            // NOTE: even if previous layer type is HW, we do this to ensure we've cleaned up
            // drawing cache created in View#draw when drawing to a SW canvas.
            destroyDrawingCache();
        }

        mLayerType = layerType;
        mLayerPaint = mLayerType == LAYER_TYPE_NONE ? null : paint;
        mRenderNode.setLayerPaint(mLayerPaint);

        // draw() behaves differently if we are on a layer, so we need to
        // invalidate() here
        invalidateParentCaches();
        invalidate(true);
    }

    /**
     * Updates the {@link Paint} object used with the current layer (used only if the current
     * layer type is not set to {@link #LAYER_TYPE_NONE}). Changed properties of the Paint
```

```
     * provided to {@link #setLayerType(int, android.graphics.Paint)} will be used the next time
     * the View is redrawn, but {@link #setLayerPaint(android.graphics.Paint)} must be called to
     * ensure that the view gets redrawn immediately.
     *
     * <p>A layer is associated with an optional {@link android.graphics.Paint}
     * instance that controls how the layer is composed on screen. The following
     * properties of the paint are taken into account when composing the layer:</p>
     * <ul>
     * <li>{@link android.graphics.Paint#getAlpha() Translucency (alpha)}</li>
     * <li>{@link android.graphics.Paint#getXfermode() Blending mode}</li>
     * <li>{@link android.graphics.Paint#getColorFilter() Color filter}</li>
     * </ul>
     *
     * <p>If this view has an alpha value set to < 1.0 by calling {@link #setAlpha(float)}, the
     * alpha value of the layer's paint is superseded by this view's alpha value.</p>
     *
     * @param paint The paint used to compose the layer. This argument is optional
     *        and can be null. It is ignored when the layer type is
     *        {@link #LAYER_TYPE_NONE}
     *
     * @see #setLayerType(int, android.graphics.Paint)
     */
    public void setLayerPaint(@Nullable Paint paint) {
        int layerType = getLayerType();
        if (layerType != LAYER_TYPE_NONE) {
            mLayerPaint = paint;
            if (layerType == LAYER_TYPE_HARDWARE) {
                if (mRenderNode.setLayerPaint(paint)) {
                    invalidateViewProperty(false, false);
                }
            } else {
                invalidate();
            }
        }
    }

    /**
     * Indicates what type of layer is currently associated with this view. By default
     * a view does not have a layer, and the layer type is {@link #LAYER_TYPE_NONE}.
     * Refer to the documentation of {@link #setLayerType(int, android.graphics.Paint)}
     * for more information on the different types of layers.
     *
     * @return {@link #LAYER_TYPE_NONE}, {@link #LAYER_TYPE_SOFTWARE} or
     *         {@link #LAYER_TYPE_HARDWARE}
     *
     * @see #setLayerType(int, android.graphics.Paint)
     * @see #buildLayer()
     * @see #LAYER_TYPE_NONE
     * @see #LAYER_TYPE_SOFTWARE
     * @see #LAYER_TYPE_HARDWARE
     */
    public int getLayerType() {
        return mLayerType;
    }

    /**
     * Forces this view's layer to be created and this view to be rendered
     * into its layer. If this view's layer type is set to {@link #LAYER_TYPE_NONE},
     * invoking this method will have no effect.
     *
     * This method can for instance be used to render a view into its layer before
     * starting an animation. If this view is complex, rendering into the layer
     * before starting the animation will avoid skipping frames.
     *
     * @throws IllegalStateException If this view is not attached to a window
     *
     * @see #setLayerType(int, android.graphics.Paint)
     */
    public void buildLayer() {
        if (mLayerType == LAYER_TYPE_NONE) return;

        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo == null) {
            throw new IllegalStateException("This view must be attached to a window first");
        }

        if (getWidth() == 0 || getHeight() == 0) {
            return;
        }

        switch (mLayerType) {
            case LAYER_TYPE_HARDWARE:
```

```
            updateDisplayListIfDirty();
            if (attachInfo.mThreadedRenderer != null && mRenderNode.isValid()) {
                attachInfo.mThreadedRenderer.buildLayer(mRenderNode);
            }
            break;
        case LAYER_TYPE_SOFTWARE:
            buildDrawingCache(true);
            break;
    }
}

/**
 * Destroys all hardware rendering resources. This method is invoked
 * when the system needs to reclaim resources. Upon execution of this
 * method, you should free any OpenGL resources created by the view.
 *
 * Note: you <strong>must</strong> call
 * <code>super.destroyHardwareResources()</code> when overriding
 * this method.
 *
 * @hide
 */
@CallSuper
protected void destroyHardwareResources() {
    if (mOverlay != null) {
        mOverlay.getOverlayView().destroyHardwareResources();
    }
    if (mGhostView != null) {
        mGhostView.destroyHardwareResources();
    }
}

/**
 * <p>Enables or disables the drawing cache. When the drawing cache is enabled, the next call
 * to {@link #getDrawingCache()} or {@link #buildDrawingCache()} will draw the view in a
 * bitmap. Calling {@link #draw(android.graphics.Canvas)} will not draw from the cache when
 * the cache is enabled. To benefit from the cache, you must request the drawing cache by
 * calling {@link #getDrawingCache()} and draw it on screen if the returned bitmap is not
 * null.</p>
 *
 * <p>Enabling the drawing cache is similar to
 * {@link #setLayerType(int, android.graphics.Paint) setting a layer} when hardware
 * acceleration is turned off. When hardware acceleration is turned on, enabling the
 * drawing cache has no effect on rendering because the system uses a different mechanism
 * for acceleration which ignores the flag. If you want to use a Bitmap for the view, even
 * when hardware acceleration is enabled, see {@link #setLayerType(int, android.graphics.Paint)}
 * for information on how to enable software and hardware layers.</p>
 *
 * <p>This API can be used to manually generate
 * a bitmap copy of this view, by setting the flag to <code>true</code> and calling
 * {@link #getDrawingCache()}.</p>
 *
 * @param enabled true to enable the drawing cache, false otherwise
 *
 * @see #isDrawingCacheEnabled()
 * @see #getDrawingCache()
 * @see #buildDrawingCache()
 * @see #setLayerType(int, android.graphics.Paint)
 */
public void setDrawingCacheEnabled(boolean enabled) {
    mCachingFailed = false;
    setFlags(enabled ? DRAWING_CACHE_ENABLED : 0, DRAWING_CACHE_ENABLED);
}

/**
 * <p>Indicates whether the drawing cache is enabled for this view.</p>
 *
 * @return true if the drawing cache is enabled
 *
 * @see #setDrawingCacheEnabled(boolean)
 * @see #getDrawingCache()
 */
@ViewDebug.ExportedProperty(category = "drawing")
public boolean isDrawingCacheEnabled() {
    return (mViewFlags & DRAWING_CACHE_ENABLED) == DRAWING_CACHE_ENABLED;
}

/**
 * Debugging utility which recursively outputs the dirty state of a view and its
 * descendants.
 *
 * @hide
```

```
 */
@SuppressWarnings({"UnusedDeclaration"})
public void outputDirtyFlags(String indent, boolean clear, int clearMask) {
    Log.d("View", indent + this + "             DIRTY(" + (mPrivateFlags & View.PFLAG_DIRTY_MASK) +
            ") DRAWN(" + (mPrivateFlags & PFLAG_DRAWN) + ")" + " CACHE_VALID(" +
            (mPrivateFlags & View.PFLAG_DRAWING_CACHE_VALID) +
            ") INVALIDATED(" + (mPrivateFlags & PFLAG_INVALIDATED) + ")");
    if (clear) {
        mPrivateFlags &= clearMask;
    }
    if (this instanceof ViewGroup) {
        ViewGroup parent = (ViewGroup) this;
        final int count = parent.getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = parent.getChildAt(i);
            child.outputDirtyFlags(indent + "  ", clear, clearMask);
        }
    }
}

/**
 * This method is used by ViewGroup to cause its children to restore or recreate their
 * display lists. It is called by getDisplayList() when the parent ViewGroup does not need
 * to recreate its own display list, which would happen if it went through the normal
 * draw/dispatchDraw mechanisms.
 *
 * @hide
 */
protected void dispatchGetDisplayList() {}

/**
 * A view that is not attached or hardware accelerated cannot create a display list.
 * This method checks these conditions and returns the appropriate result.
 *
 * @return true if view has the ability to create a display list, false otherwise.
 *
 * @hide
 */
public boolean canHaveDisplayList() {
    return !(mAttachInfo == null || mAttachInfo.mThreadedRenderer == null);
}

/**
 * Gets the RenderNode for the view, and updates its DisplayList (if needed and supported)
 * @hide
 */
@NonNull
public RenderNode updateDisplayListIfDirty() {
    final RenderNode renderNode = mRenderNode;
    if (!canHaveDisplayList()) {
        // can't populate RenderNode, don't try
        return renderNode;
    }

    if ((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0
            || !renderNode.isValid()
            || (mRecreateDisplayList)) {
        // Don't need to recreate the display list, just need to tell our
        // children to restore/recreate theirs
        if (renderNode.isValid()
                && !mRecreateDisplayList) {
            mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            dispatchGetDisplayList();

            return renderNode; // no work needed
        }

        // If we got here, we're recreating it. Mark it as such to ensure that
        // we copy in child display lists into ours in drawChild()
        mRecreateDisplayList = true;

        int width = mRight - mLeft;
        int height = mBottom - mTop;
        int layerType = getLayerType();

        final DisplayListCanvas canvas = renderNode.start(width, height);
        canvas.setHighContrastText(mAttachInfo.mHighContrastText);

        try {
            if (layerType == LAYER_TYPE_SOFTWARE) {
                buildDrawingCache(true);
```

```
                Bitmap cache = getDrawingCache(true);
                if (cache != null) {
                    canvas.drawBitmap(cache, 0, 0, mLayerPaint);
                }
            } else {
                computeScroll();

                canvas.translate(-mScrollX, -mScrollY);
                mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
                mPrivateFlags &= ~PFLAG_DIRTY_MASK;

                // Fast path for layouts with no backgrounds
                if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
                    dispatchDraw(canvas);
                    drawAutofilledHighlight(canvas);
                    if (mOverlay != null && !mOverlay.isEmpty()) {
                        mOverlay.getOverlayView().draw(canvas);
                    }
                    if (debugDraw()) {
                        debugDrawFocus(canvas);
                    }
                } else {
                    draw(canvas);
                }
            }
        } finally {
            renderNode.end(canvas);
            setDisplayListProperties(renderNode);
        }
    } else {
        mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
    }
    return renderNode;
}

private void resetDisplayList() {
    mRenderNode.discardDisplayList();
    if (mBackgroundRenderNode != null) {
        mBackgroundRenderNode.discardDisplayList();
    }
}

/**
 * <p>Calling this method is equivalent to calling <code>getDrawingCache(false)</code>.</p>
 *
 * @return A non-scaled bitmap representing this view or null if cache is disabled.
 *
 * @see #getDrawingCache(boolean)
 */
public Bitmap getDrawingCache() {
    return getDrawingCache(false);
}

/**
 * <p>Returns the bitmap in which this view drawing is cached. The returned bitmap
 * is null when caching is disabled. If caching is enabled and the cache is not ready,
 * this method will create it. Calling {@link #draw(android.graphics.Canvas)} will not
 * draw from the cache when the cache is enabled. To benefit from the cache, you must
 * request the drawing cache by calling this method and draw it on screen if the
 * returned bitmap is not null.</p>
 *
 * <p>Note about auto scaling in compatibility mode: When auto scaling is not enabled,
 * this method will create a bitmap of the same size as this view. Because this bitmap
 * will be drawn scaled by the parent ViewGroup, the result on screen might show
 * scaling artifacts. To avoid such artifacts, you should call this method by setting
 * the auto scaling to true. Doing so, however, will generate a bitmap of a different
 * size than the view. This implies that your application must be able to handle this
 * size.</p>
 *
 * @param autoScale Indicates whether the generated bitmap should be scaled based on
 *        the current density of the screen when the application is in compatibility
 *        mode.
 *
 * @return A bitmap representing this view or null if cache is disabled.
 *
 * @see #setDrawingCacheEnabled(boolean)
 * @see #isDrawingCacheEnabled()
 * @see #buildDrawingCache(boolean)
 * @see #destroyDrawingCache()
 */
public Bitmap getDrawingCache(boolean autoScale) {
```

```
    if ((mViewFlags & WILL_NOT_CACHE_DRAWING) == WILL_NOT_CACHE_DRAWING) {
        return null;
    }
    if ((mViewFlags & DRAWING_CACHE_ENABLED) == DRAWING_CACHE_ENABLED) {
        buildDrawingCache(autoScale);
    }
    return autoScale ? mDrawingCache : mUnscaledDrawingCache;
}

/**
 * <p>Frees the resources used by the drawing cache. If you call
 * {@link #buildDrawingCache()} manually without calling
 * {@link #setDrawingCacheEnabled(boolean) setDrawingCacheEnabled(true)}, you
 * should cleanup the cache with this method afterwards.</p>
 *
 * @see #setDrawingCacheEnabled(boolean)
 * @see #buildDrawingCache()
 * @see #getDrawingCache()
 */
public void destroyDrawingCache() {
    if (mDrawingCache != null) {
        mDrawingCache.recycle();
        mDrawingCache = null;
    }
    if (mUnscaledDrawingCache != null) {
        mUnscaledDrawingCache.recycle();
        mUnscaledDrawingCache = null;
    }
}

/**
 * Setting a solid background color for the drawing cache's bitmaps will improve
 * performance and memory usage. Note, though that this should only be used if this
 * view will always be drawn on top of a solid color.
 *
 * @param color The background color to use for the drawing cache's bitmap
 *
 * @see #setDrawingCacheEnabled(boolean)
 * @see #buildDrawingCache()
 * @see #getDrawingCache()
 */
public void setDrawingCacheBackgroundColor(@ColorInt int color) {
    if (color != mDrawingCacheBackgroundColor) {
        mDrawingCacheBackgroundColor = color;
        mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
    }
}

/**
 * @see #setDrawingCacheBackgroundColor(int)
 *
 * @return The background color to used for the drawing cache's bitmap
 */
@ColorInt
public int getDrawingCacheBackgroundColor() {
    return mDrawingCacheBackgroundColor;
}

/**
 * <p>Calling this method is equivalent to calling <code>buildDrawingCache(false)</code>.</p>
 *
 * @see #buildDrawingCache(boolean)
 */
public void buildDrawingCache() {
    buildDrawingCache(false);
}

/**
 * <p>Forces the drawing cache to be built if the drawing cache is invalid.</p>
 *
 * <p>If you call {@link #buildDrawingCache()} manually without calling
 * {@link #setDrawingCacheEnabled(boolean) setDrawingCacheEnabled(true)}, you
 * should cleanup the cache by calling {@link #destroyDrawingCache()} afterwards.</p>
 *
 * <p>Note about auto scaling in compatibility mode: When auto scaling is not enabled,
 * this method will create a bitmap of the same size as this view. Because this bitmap
 * will be drawn scaled by the parent ViewGroup, the result on screen might show
 * scaling artifacts. To avoid such artifacts, you should call this method by setting
 * the auto scaling to true. Doing so, however, will generate a bitmap of a different
 * size than the view. This implies that your application must be able to handle this
 * size.</p>
 *
```

```
     * <p>You should avoid calling this method when hardware acceleration is enabled. If
     * you do not need the drawing cache bitmap, calling this method will increase memory
     * usage and cause the view to be rendered in software once, thus negatively impacting
     * performance.</p>
     *
     * @see #getDrawingCache()
     * @see #destroyDrawingCache()
     */
    public void buildDrawingCache(boolean autoScale) {
        if ((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 || (autoScale ?
                mDrawingCache == null : mUnscaledDrawingCache == null)) {
            if (Trace.isTagEnabled(Trace.TRACE_TAG_VIEW)) {
                Trace.traceBegin(Trace.TRACE_TAG_VIEW,
                        "buildDrawingCache/SW Layer for " + getClass().getSimpleName());
            }
            try {
                buildDrawingCacheImpl(autoScale);
            } finally {
                Trace.traceEnd(Trace.TRACE_TAG_VIEW);
            }
        }
    }

    /**
     * private, internal implementation of buildDrawingCache, used to enable tracing
     */
    private void buildDrawingCacheImpl(boolean autoScale) {
        mCachingFailed = false;

        int width = mRight - mLeft;
        int height = mBottom - mTop;

        final AttachInfo attachInfo = mAttachInfo;
        final boolean scalingRequired = attachInfo != null && attachInfo.mScalingRequired;

        if (autoScale && scalingRequired) {
            width = (int) ((width * attachInfo.mApplicationScale) + 0.5f);
            height = (int) ((height * attachInfo.mApplicationScale) + 0.5f);
        }

        final int drawingCacheBackgroundColor = mDrawingCacheBackgroundColor;
        final boolean opaque = drawingCacheBackgroundColor != 0 || isOpaque();
        final boolean use32BitCache = attachInfo != null && attachInfo.mUse32BitDrawingCache;

        final long projectedBitmapSize = width * height * (opaque && !use32BitCache ? 2 : 4);
        final long drawingCacheSize =
                ViewConfiguration.get(mContext).getScaledMaximumDrawingCacheSize();
        if (width <= 0 || height <= 0 || projectedBitmapSize > drawingCacheSize) {
            if (width > 0 && height > 0) {
                Log.w(VIEW_LOG_TAG, getClass().getSimpleName() + " not displayed because it is"
                        + " too large to fit into a software layer (or drawing cache), needs "
                        + projectedBitmapSize + " bytes, only "
                        + drawingCacheSize + " available");
            }
            destroyDrawingCache();
            mCachingFailed = true;
            return;
        }

        boolean clear = true;
        Bitmap bitmap = autoScale ? mDrawingCache : mUnscaledDrawingCache;

        if (bitmap == null || bitmap.getWidth() != width || bitmap.getHeight() != height) {
            Bitmap.Config quality;
            if (!opaque) {
                // Never pick ARGB_4444 because it looks awful
                // Keep the DRAWING_CACHE_QUALITY_LOW flag just in case
                switch (mViewFlags & DRAWING_CACHE_QUALITY_MASK) {
                    case DRAWING_CACHE_QUALITY_AUTO:
                    case DRAWING_CACHE_QUALITY_LOW:
                    case DRAWING_CACHE_QUALITY_HIGH:
                    default:
                        quality = Bitmap.Config.ARGB_8888;
                        break;
                }
            } else {
                // Optimization for translucent windows
                // If the window is translucent, use a 32 bits bitmap to benefit from memcpy()
                quality = use32BitCache ? Bitmap.Config.ARGB_8888 : Bitmap.Config.RGB_565;
            }

            // Try to cleanup memory
```

```
    if (bitmap != null) bitmap.recycle();

    try {
        bitmap = Bitmap.createBitmap(mResources.getDisplayMetrics(),
                width, height, quality);
        bitmap.setDensity(getResources().getDisplayMetrics().densityDpi);
        if (autoScale) {
            mDrawingCache = bitmap;
        } else {
            mUnscaledDrawingCache = bitmap;
        }
        if (opaque && use32BitCache) bitmap.setHasAlpha(false);
    } catch (OutOfMemoryError e) {
        // If there is not enough memory to create the bitmap cache, just
        // ignore the issue as bitmap caches are not required to draw the
        // view hierarchy
        if (autoScale) {
            mDrawingCache = null;
        } else {
            mUnscaledDrawingCache = null;
        }
        mCachingFailed = true;
        return;
    }

    clear = drawingCacheBackgroundColor != 0;
}

Canvas canvas;
if (attachInfo != null) {
    canvas = attachInfo.mCanvas;
    if (canvas == null) {
        canvas = new Canvas();
    }
    canvas.setBitmap(bitmap);
    // Temporarily clobber the cached Canvas in case one of our children
    // is also using a drawing cache. Without this, the children would
    // steal the canvas by attaching their own bitmap to it and bad, bad
    // thing would happen (invisible views, corrupted drawings, etc.)
    attachInfo.mCanvas = null;
} else {
    // This case should hopefully never or seldom happen
    canvas = new Canvas(bitmap);
}

if (clear) {
    bitmap.eraseColor(drawingCacheBackgroundColor);
}

computeScroll();
final int restoreCount = canvas.save();

if (autoScale && scalingRequired) {
    final float scale = attachInfo.mApplicationScale;
    canvas.scale(scale, scale);
}

canvas.translate(-mScrollX, -mScrollY);

mPrivateFlags |= PFLAG_DRAWN;
if (mAttachInfo == null || !mAttachInfo.mHardwareAccelerated ||
        mLayerType != LAYER_TYPE_NONE) {
    mPrivateFlags |= PFLAG_DRAWING_CACHE_VALID;
}

// Fast path for layouts with no backgrounds
if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;
    dispatchDraw(canvas);
    drawAutofilledHighlight(canvas);
    if (mOverlay != null && !mOverlay.isEmpty()) {
        mOverlay.getOverlayView().draw(canvas);
    }
} else {
    draw(canvas);
}

canvas.restoreToCount(restoreCount);
canvas.setBitmap(null);

if (attachInfo != null) {
    // Restore the cached Canvas for our siblings
```

```
        attachInfo.mCanvas = canvas;
    }
}

/**
 * Create a snapshot of the view into a bitmap.  We should probably make
 * some form of this public, but should think about the API.
 *
 * @hide
 */
public Bitmap createSnapshot(Bitmap.Config quality, int backgroundColor, boolean skipChildren) {
    int width = mRight - mLeft;
    int height = mBottom - mTop;

    final AttachInfo attachInfo = mAttachInfo;
    final float scale = attachInfo != null ? attachInfo.mApplicationScale : 1.0f;
    width = (int) ((width * scale) + 0.5f);
    height = (int) ((height * scale) + 0.5f);

    Bitmap bitmap = Bitmap.createBitmap(mResources.getDisplayMetrics(),
            width > 0 ? width : 1, height > 0 ? height : 1, quality);
    if (bitmap == null) {
        throw new OutOfMemoryError();
    }

    Resources resources = getResources();
    if (resources != null) {
        bitmap.setDensity(resources.getDisplayMetrics().densityDpi);
    }

    Canvas canvas;
    if (attachInfo != null) {
        canvas = attachInfo.mCanvas;
        if (canvas == null) {
            canvas = new Canvas();
        }
        canvas.setBitmap(bitmap);
        // Temporarily clobber the cached Canvas in case one of our children
        // is also using a drawing cache. Without this, the children would
        // steal the canvas by attaching their own bitmap to it and bad, bad
        // things would happen (invisible views, corrupted drawings, etc.)
        attachInfo.mCanvas = null;
    } else {
        // This case should hopefully never or seldom happen
        canvas = new Canvas(bitmap);
    }
    boolean enabledHwBitmapsInSwMode = canvas.isHwBitmapsInSwModeEnabled();
    canvas.setHwBitmapsInSwModeEnabled(true);
    if ((backgroundColor & 0xff000000) != 0) {
        bitmap.eraseColor(backgroundColor);
    }

    computeScroll();
    final int restoreCount = canvas.save();
    canvas.scale(scale, scale);
    canvas.translate(-mScrollX, -mScrollY);

    // Temporarily remove the dirty mask
    int flags = mPrivateFlags;
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;

    // Fast path for layouts with no backgrounds
    if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
        dispatchDraw(canvas);
        drawAutofilledHighlight(canvas);
        if (mOverlay != null && !mOverlay.isEmpty()) {
            mOverlay.getOverlayView().draw(canvas);
        }
    } else {
        draw(canvas);
    }

    mPrivateFlags = flags;

    canvas.restoreToCount(restoreCount);
    canvas.setBitmap(null);
    canvas.setHwBitmapsInSwModeEnabled(enabledHwBitmapsInSwMode);

    if (attachInfo != null) {
        // Restore the cached Canvas for our siblings
        attachInfo.mCanvas = canvas;
    }
```

```
    return bitmap;
}

/**
 * Indicates whether this View is currently in edit mode. A View is usually
 * in edit mode when displayed within a developer tool. For instance, if
 * this View is being drawn by a visual user interface builder, this method
 * should return true.
 *
 * Subclasses should check the return value of this method to provide
 * different behaviors if their normal behavior might interfere with the
 * host environment. For instance: the class spawns a thread in its
 * constructor, the drawing code relies on device-specific features, etc.
 *
 * This method is usually checked in the drawing code of custom widgets.
 *
 * @return True if this View is in edit mode, false otherwise.
 */
public boolean isInEditMode() {
    return false;
}

/**
 * If the View draws content inside its padding and enables fading edges,
 * it needs to support padding offsets. Padding offsets are added to the
 * fading edges to extend the length of the fade so that it covers pixels
 * drawn inside the padding.
 *
 * Subclasses of this class should override this method if they need
 * to draw content inside the padding.
 *
 * @return True if padding offset must be applied, false otherwise.
 *
 * @see #getLeftPaddingOffset()
 * @see #getRightPaddingOffset()
 * @see #getTopPaddingOffset()
 * @see #getBottomPaddingOffset()
 *
 * @since CURRENT
 */
protected boolean isPaddingOffsetRequired() {
    return false;
}

/**
 * Amount by which to extend the left fading region. Called only when
 * {@link #isPaddingOffsetRequired()} returns true.
 *
 * @return The left padding offset in pixels.
 *
 * @see #isPaddingOffsetRequired()
 *
 * @since CURRENT
 */
protected int getLeftPaddingOffset() {
    return 0;
}

/**
 * Amount by which to extend the right fading region. Called only when
 * {@link #isPaddingOffsetRequired()} returns true.
 *
 * @return The right padding offset in pixels.
 *
 * @see #isPaddingOffsetRequired()
 *
 * @since CURRENT
 */
protected int getRightPaddingOffset() {
    return 0;
}

/**
 * Amount by which to extend the top fading region. Called only when
 * {@link #isPaddingOffsetRequired()} returns true.
 *
 * @return The top padding offset in pixels.
 *
 * @see #isPaddingOffsetRequired()
 *
 * @since CURRENT
```

```
     */
    protected int getTopPaddingOffset() {
        return 0;
    }

    /**
     * Amount by which to extend the bottom fading region. Called only when
     * {@link #isPaddingOffsetRequired()} returns true.
     *
     * @return The bottom padding offset in pixels.
     *
     * @see #isPaddingOffsetRequired()
     *
     * @since CURRENT
     */
    protected int getBottomPaddingOffset() {
        return 0;
    }

    /**
     * @hide
     * @param offsetRequired
     */
    protected int getFadeTop(boolean offsetRequired) {
        int top = mPaddingTop;
        if (offsetRequired) top += getTopPaddingOffset();
        return top;
    }

    /**
     * @hide
     * @param offsetRequired
     */
    protected int getFadeHeight(boolean offsetRequired) {
        int padding = mPaddingTop;
        if (offsetRequired) padding += getTopPaddingOffset();
        return mBottom - mTop - mPaddingBottom - padding;
    }

    /**
     * <p>Indicates whether this view is attached to a hardware accelerated
     * window or not.</p>
     *
     * <p>Even if this method returns true, it does not mean that every call
     * to {@link #draw(android.graphics.Canvas)} will be made with an hardware
     * accelerated {@link android.graphics.Canvas}. For instance, if this view
     * is drawn onto an offscreen {@link android.graphics.Bitmap} and its
     * window is hardware accelerated,
     * {@link android.graphics.Canvas#isHardwareAccelerated()} will likely
     * return false, and this method will return true.</p>
     *
     * @return True if the view is attached to a window and the window is
     *         hardware accelerated; false in any other case.
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    public boolean isHardwareAccelerated() {
        return mAttachInfo != null && mAttachInfo.mHardwareAccelerated;
    }

    /**
     * Sets a rectangular area on this view to which the view will be clipped
     * when it is drawn. Setting the value to null will remove the clip bounds
     * and the view will draw normally, using its full bounds.
     *
     * @param clipBounds The rectangular area, in the local coordinates of
     * this view, to which future drawing operations will be clipped.
     */
    public void setClipBounds(Rect clipBounds) {
        if (clipBounds == mClipBounds
                || (clipBounds != null && clipBounds.equals(mClipBounds))) {
            return;
        }
        if (clipBounds != null) {
            if (mClipBounds == null) {
                mClipBounds = new Rect(clipBounds);
            } else {
                mClipBounds.set(clipBounds);
            }
        } else {
            mClipBounds = null;
        }
        mRenderNode.setClipBounds(mClipBounds);
```

```
    invalidateViewProperty(false, false);
}

/**
 * Returns a copy of the current {@link #setClipBounds(Rect) clipBounds}.
 *
 * @return A copy of the current clip bounds if clip bounds are set,
 * otherwise null.
 */
public Rect getClipBounds() {
    return (mClipBounds != null) ? new Rect(mClipBounds) : null;
}


/**
 * Populates an output rectangle with the clip bounds of the view,
 * returning {@code true} if successful or {@code false} if the view's
 * clip bounds are {@code null}.
 *
 * @param outRect rectangle in which to place the clip bounds of the view
 * @return {@code true} if successful or {@code false} if the view's
 *         clip bounds are {@code null}
 */
public boolean getClipBounds(Rect outRect) {
    if (mClipBounds != null) {
        outRect.set(mClipBounds);
        return true;
    }
    return false;
}

/**
 * Utility function, called by draw(canvas, parent, drawingTime) to handle the less common
 * case of an active Animation being run on the view.
 */
private boolean applyLegacyAnimation(ViewGroup parent, long drawingTime,
        Animation a, boolean scalingRequired) {
    Transformation invalidationTransform;
    final int flags = parent.mGroupFlags;
    final boolean initialized = a.isInitialized();
    if (!initialized) {
        a.initialize(mRight - mLeft, mBottom - mTop, parent.getWidth(), parent.getHeight());
        a.initializeInvalidateRegion(0, 0, mRight - mLeft, mBottom - mTop);
        if (mAttachInfo != null) a.setListenerHandler(mAttachInfo.mHandler);
        onAnimationStart();
    }

    final Transformation t = parent.getChildTransformation();
    boolean more = a.getTransformation(drawingTime, t, 1f);
    if (scalingRequired && mAttachInfo.mApplicationScale != 1f) {
        if (parent.mInvalidationTransformation == null) {
            parent.mInvalidationTransformation = new Transformation();
        }
        invalidationTransform = parent.mInvalidationTransformation;
        a.getTransformation(drawingTime, invalidationTransform, 1f);
    } else {
        invalidationTransform = t;
    }

    if (more) {
        if (!a.willChangeBounds()) {
            if ((flags & (ViewGroup.FLAG_OPTIMIZE_INVALIDATE | ViewGroup.FLAG_ANIMATION_DONE)) ==
                    ViewGroup.FLAG_OPTIMIZE_INVALIDATE) {
                parent.mGroupFlags |= ViewGroup.FLAG_INVALIDATE_REQUIRED;
            } else if ((flags & ViewGroup.FLAG_INVALIDATE_REQUIRED) == 0) {
                // The child need to draw an animation, potentially offscreen, so
                // make sure we do not cancel invalidate requests
                parent.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
                parent.invalidate(mLeft, mTop, mRight, mBottom);
            }
        } else {
            if (parent.mInvalidateRegion == null) {
                parent.mInvalidateRegion = new RectF();
            }
            final RectF region = parent.mInvalidateRegion;
            a.getInvalidateRegion(0, 0, mRight - mLeft, mBottom - mTop, region,
                    invalidationTransform);

            // The child need to draw an animation, potentially offscreen, so
            // make sure we do not cancel invalidate requests
            parent.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
```

```
            final int left = mLeft + (int) region.left;
            final int top = mTop + (int) region.top;
            parent.invalidate(left, top, left + (int) (region.width() + .5f),
                    top + (int) (region.height() + .5f));
        }
    }
    return more;
}

/**
 * This method is called by getDisplayList() when a display list is recorded for a View.
 * It pushes any properties to the RenderNode that aren't managed by the RenderNode.
 */
void setDisplayListProperties(RenderNode renderNode) {
    if (renderNode != null) {
        renderNode.setHasOverlappingRendering(getHasOverlappingRendering());
        renderNode.setClipToBounds(mParent instanceof ViewGroup
                && ((ViewGroup) mParent).getClipChildren());

        float alpha = 1;
        if (mParent instanceof ViewGroup && (((ViewGroup) mParent).mGroupFlags &
                ViewGroup.FLAG_SUPPORT_STATIC_TRANSFORMATIONS) != 0) {
            ViewGroup parentVG = (ViewGroup) mParent;
            final Transformation t = parentVG.getChildTransformation();
            if (parentVG.getChildStaticTransformation(this, t)) {
                final int transformType = t.getTransformationType();
                if (transformType != Transformation.TYPE_IDENTITY) {
                    if ((transformType & Transformation.TYPE_ALPHA) != 0) {
                        alpha = t.getAlpha();
                    }
                    if ((transformType & Transformation.TYPE_MATRIX) != 0) {
                        renderNode.setStaticMatrix(t.getMatrix());
                    }
                }
            }
        }
        if (mTransformationInfo != null) {
            alpha *= getFinalAlpha();
            if (alpha < 1) {
                final int multipliedAlpha = (int) (255 * alpha);
                if (onSetAlpha(multipliedAlpha)) {
                    alpha = 1;
                }
            }
            renderNode.setAlpha(alpha);
        } else if (alpha < 1) {
            renderNode.setAlpha(alpha);
        }
    }
}

/**
 * This method is called by ViewGroup.drawChild() to have each child view draw itself.
 *
 * This is where the View specializes rendering behavior based on layer type,
 * and hardware acceleration.
 */
boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {
    final boolean hardwareAcceleratedCanvas = canvas.isHardwareAccelerated();
    /* If an attached view draws to a HW canvas, it may use its RenderNode + DisplayList.
     *
     * If a view is dettached, its DisplayList shouldn't exist. If the canvas isn't
     * HW accelerated, it can't handle drawing RenderNodes.
     */
    boolean drawingWithRenderNode = mAttachInfo != null
            && mAttachInfo.mHardwareAccelerated
            && hardwareAcceleratedCanvas;

    boolean more = false;
    final boolean childHasIdentityMatrix = hasIdentityMatrix();
    final int parentFlags = parent.mGroupFlags;

    if ((parentFlags & ViewGroup.FLAG_CLEAR_TRANSFORMATION) != 0) {
        parent.getChildTransformation().clear();
        parent.mGroupFlags &= ~ViewGroup.FLAG_CLEAR_TRANSFORMATION;
    }

    Transformation transformToApply = null;
    boolean concatMatrix = false;
    final boolean scalingRequired = mAttachInfo != null && mAttachInfo.mScalingRequired;
    final Animation a = getAnimation();
    if (a != null) {
```

```
    more = applyLegacyAnimation(parent, drawingTime, a, scalingRequired);
    concatMatrix = a.willChangeTransformationMatrix();
    if (concatMatrix) {
        mPrivateFlags3 |= PFLAG3_VIEW_IS_ANIMATING_TRANSFORM;
    }
    transformToApply = parent.getChildTransformation();
} else {
    if ((mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_TRANSFORM) != 0) {
        // No longer animating: clear out old animation matrix
        mRenderNode.setAnimationMatrix(null);
        mPrivateFlags3 &= ~PFLAG3_VIEW_IS_ANIMATING_TRANSFORM;
    }
    if (!drawingWithRenderNode
            && (parentFlags & ViewGroup.FLAG_SUPPORT_STATIC_TRANSFORMATIONS) != 0) {
        final Transformation t = parent.getChildTransformation();
        final boolean hasTransform = parent.getChildStaticTransformation(this, t);
        if (hasTransform) {
            final int transformType = t.getTransformationType();
            transformToApply = transformType != Transformation.TYPE_IDENTITY ? t : null;
            concatMatrix = (transformType & Transformation.TYPE_MATRIX) != 0;
        }
    }
}

concatMatrix |= !childHasIdentityMatrix;

// Sets the flag as early as possible to allow draw() implementations
// to call invalidate() successfully when doing animations
mPrivateFlags |= PFLAG_DRAWN;

if (!concatMatrix &&
        (parentFlags & (ViewGroup.FLAG_SUPPORT_STATIC_TRANSFORMATIONS |
                ViewGroup.FLAG_CLIP_CHILDREN)) == ViewGroup.FLAG_CLIP_CHILDREN &&
        canvas.quickReject(mLeft, mTop, mRight, mBottom, Canvas.EdgeType.BW) &&
        (mPrivateFlags & PFLAG_DRAW_ANIMATION) == 0) {
    mPrivateFlags2 |= PFLAG2_VIEW_QUICK_REJECTED;
    return more;
}
mPrivateFlags2 &= ~PFLAG2_VIEW_QUICK_REJECTED;

if (hardwareAcceleratedCanvas) {
    // Clear INVALIDATED flag to allow invalidation to occur during rendering, but
    // retain the flag's value temporarily in the mRecreateDisplayList flag
    mRecreateDisplayList = (mPrivateFlags & PFLAG_INVALIDATED) != 0;
    mPrivateFlags &= ~PFLAG_INVALIDATED;
}

RenderNode renderNode = null;
Bitmap cache = null;
int layerType = getLayerType(); // TODO: signify cache state with just 'cache' local
if (layerType == LAYER_TYPE_SOFTWARE || !drawingWithRenderNode) {
     if (layerType != LAYER_TYPE_NONE) {
         // If not drawing with RenderNode, treat HW layers as SW
         layerType = LAYER_TYPE_SOFTWARE;
         buildDrawingCache(true);
    }
    cache = getDrawingCache(true);
}

if (drawingWithRenderNode) {
    // Delay getting the display list until animation-driven alpha values are
    // set up and possibly passed on to the view
    renderNode = updateDisplayListIfDirty();
    if (!renderNode.isValid()) {
        // Uncommon, but possible. If a view is removed from the hierarchy during the call
        // to getDisplayList(), the display list will be marked invalid and we should not
        // try to use it again.
        renderNode = null;
        drawingWithRenderNode = false;
    }
}

int sx = 0;
int sy = 0;
if (!drawingWithRenderNode) {
    computeScroll();
    sx = mScrollX;
    sy = mScrollY;
}

final boolean drawingWithDrawingCache = cache != null && !drawingWithRenderNode;
final boolean offsetForScroll = cache == null && !drawingWithRenderNode;
```

```
int restoreTo = -1;
if (!drawingWithRenderNode || transformToApply != null) {
    restoreTo = canvas.save();
}
if (offsetForScroll) {
    canvas.translate(mLeft - sx, mTop - sy);
} else {
    if (!drawingWithRenderNode) {
        canvas.translate(mLeft, mTop);
    }
    if (scalingRequired) {
        if (drawingWithRenderNode) {
            // TODO: Might not need this if we put everything inside the DL
            restoreTo = canvas.save();
        }
        // mAttachInfo cannot be null, otherwise scalingRequired == false
        final float scale = 1.0f / mAttachInfo.mApplicationScale;
        canvas.scale(scale, scale);
    }
}

float alpha = drawingWithRenderNode ? 1 : (getAlpha() * getTransitionAlpha());
if (transformToApply != null
        || alpha < 1
        || !hasIdentityMatrix()
        || (mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_ALPHA) != 0) {
    if (transformToApply != null || !childHasIdentityMatrix) {
        int transX = 0;
        int transY = 0;

        if (offsetForScroll) {
            transX = -sx;
            transY = -sy;
        }

        if (transformToApply != null) {
            if (concatMatrix) {
                if (drawingWithRenderNode) {
                    renderNode.setAnimationMatrix(transformToApply.getMatrix());
                } else {
                    // Undo the scroll translation, apply the transformation matrix,
                    // then redo the scroll translate to get the correct result.
                    canvas.translate(-transX, -transY);
                    canvas.concat(transformToApply.getMatrix());
                    canvas.translate(transX, transY);
                }
                parent.mGroupFlags |= ViewGroup.FLAG_CLEAR_TRANSFORMATION;
            }

            float transformAlpha = transformToApply.getAlpha();
            if (transformAlpha < 1) {
                alpha *= transformAlpha;
                parent.mGroupFlags |= ViewGroup.FLAG_CLEAR_TRANSFORMATION;
            }
        }

        if (!childHasIdentityMatrix && !drawingWithRenderNode) {
            canvas.translate(-transX, -transY);
            canvas.concat(getMatrix());
            canvas.translate(transX, transY);
        }
    }

    // Deal with alpha if it is or used to be <1
    if (alpha < 1 || (mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_ALPHA) != 0) {
        if (alpha < 1) {
            mPrivateFlags3 |= PFLAG3_VIEW_IS_ANIMATING_ALPHA;
        } else {
            mPrivateFlags3 &= ~PFLAG3_VIEW_IS_ANIMATING_ALPHA;
        }
        parent.mGroupFlags |= ViewGroup.FLAG_CLEAR_TRANSFORMATION;
        if (!drawingWithDrawingCache) {
            final int multipliedAlpha = (int) (255 * alpha);
            if (!onSetAlpha(multipliedAlpha)) {
                if (drawingWithRenderNode) {
                    renderNode.setAlpha(alpha * getAlpha() * getTransitionAlpha());
                } else if (layerType == LAYER_TYPE_NONE) {
                    canvas.saveLayerAlpha(sx, sy, sx + getWidth(), sy + getHeight(),
                            multipliedAlpha);
                }
            } else {
```

```
                // Alpha is handled by the child directly, clobber the layer's alpha
                mPrivateFlags |= PFLAG_ALPHA_SET;
            }
        }
    }
} else if ((mPrivateFlags & PFLAG_ALPHA_SET) == PFLAG_ALPHA_SET) {
    onSetAlpha(255);
    mPrivateFlags &= ~PFLAG_ALPHA_SET;
}

if (!drawingWithRenderNode) {
    // apply clips directly, since RenderNode won't do it for this draw
    if ((parentFlags & ViewGroup.FLAG_CLIP_CHILDREN) != 0 && cache == null) {
        if (offsetForScroll) {
            canvas.clipRect(sx, sy, sx + getWidth(), sy + getHeight());
        } else {
            if (!scalingRequired || cache == null) {
                canvas.clipRect(0, 0, getWidth(), getHeight());
            } else {
                canvas.clipRect(0, 0, cache.getWidth(), cache.getHeight());
            }
        }
    }

    if (mClipBounds != null) {
        // clip bounds ignore scroll
        canvas.clipRect(mClipBounds);
    }
}

if (!drawingWithDrawingCache) {
    if (drawingWithRenderNode) {
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
    } else {
        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            dispatchDraw(canvas);
        } else {
            draw(canvas);
        }
    }
} else if (cache != null) {
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;
    if (layerType == LAYER_TYPE_NONE || mLayerPaint == null) {
        // no layer paint, use temporary paint to draw bitmap
        Paint cachePaint = parent.mCachePaint;
        if (cachePaint == null) {
            cachePaint = new Paint();
            cachePaint.setDither(false);
            parent.mCachePaint = cachePaint;
        }
        cachePaint.setAlpha((int) (alpha * 255));
        canvas.drawBitmap(cache, 0.0f, 0.0f, cachePaint);
    } else {
        // use layer paint to draw the bitmap, merging the two alphas, but also restore
        int layerPaintAlpha = mLayerPaint.getAlpha();
        if (alpha < 1) {
            mLayerPaint.setAlpha((int) (alpha * layerPaintAlpha));
        }
        canvas.drawBitmap(cache, 0.0f, 0.0f, mLayerPaint);
        if (alpha < 1) {
            mLayerPaint.setAlpha(layerPaintAlpha);
        }
    }
}

if (restoreTo >= 0) {
    canvas.restoreToCount(restoreTo);
}

if (a != null && !more) {
    if (!hardwareAcceleratedCanvas && !a.getFillAfter()) {
        onSetAlpha(255);
    }
    parent.finishAnimatingView(this, a);
}

if (more && hardwareAcceleratedCanvas) {
    if (a.hasAlpha() && (mPrivateFlags & PFLAG_ALPHA_SET) == PFLAG_ALPHA_SET) {
        // alpha animations should cause the child to recreate its display list
```

```
            invalidate(true);
        }
    }

    mRecreateDisplayList = false;

    return more;
}

static Paint getDebugPaint() {
    if (sDebugPaint == null) {
        sDebugPaint = new Paint();
        sDebugPaint.setAntiAlias(false);
    }
    return sDebugPaint;
}

final int dipsToPixels(int dips) {
    float scale = getContext().getResources().getDisplayMetrics().density;
    return (int) (dips * scale + 0.5f);
}

final private void debugDrawFocus(Canvas canvas) {
    if (isFocused()) {
        final int cornerSquareSize = dipsToPixels(DEBUG_CORNERS_SIZE_DIP);
        final int l = mScrollX;
        final int r = l + mRight - mLeft;
        final int t = mScrollY;
        final int b = t + mBottom - mTop;

        final Paint paint = getDebugPaint();
        paint.setColor(DEBUG_CORNERS_COLOR);

        // Draw squares in corners.
        paint.setStyle(Paint.Style.FILL);
        canvas.drawRect(l, t, l + cornerSquareSize, t + cornerSquareSize, paint);
        canvas.drawRect(r - cornerSquareSize, t, r, t + cornerSquareSize, paint);
        canvas.drawRect(l, b - cornerSquareSize, l + cornerSquareSize, b, paint);
        canvas.drawRect(r - cornerSquareSize, b - cornerSquareSize, r, b, paint);

        // Draw big X across the view.
        paint.setStyle(Paint.Style.STROKE);
        canvas.drawLine(l, t, r, b, paint);
        canvas.drawLine(l, b, r, t, paint);
    }
}

/**
 * Manually render this view (and all of its children) to the given Canvas.
 * The view must have already done a full layout before this function is
 * called.  When implementing a view, implement
 * {@link #onDraw(android.graphics.Canvas)} instead of overriding this method.
 * If you do need to override this method, call the superclass version.
 *
 * @param canvas The Canvas to which the View is rendered.
 */
@CallSuper
public void draw(Canvas canvas) {
    final int privateFlags = mPrivateFlags;
    final boolean dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) == PFLAG_DIRTY_OPAQUE &&
            (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);
    mPrivateFlags = (privateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;

    /*
     * Draw traversal performs several drawing steps which must be executed
     * in the appropriate order:
     *
     *      1. Draw the background
     *      2. If necessary, save the canvas' layers to prepare for fading
     *      3. Draw view's content
     *      4. Draw children
     *      5. If necessary, draw the fading edges and restore layers
     *      6. Draw decorations (scrollbars for instance)
     */

    // Step 1, draw the background, if needed
    int saveCount;

    if (!dirtyOpaque) {
        drawBackground(canvas);
    }
```

```
// skip step 2 & 5 if possible (common case)
final int viewFlags = mViewFlags;
boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
if (!verticalEdges && !horizontalEdges) {
    // Step 3, draw the content
    if (!dirtyOpaque) onDraw(canvas);

    // Step 4, draw the children
    dispatchDraw(canvas);

    drawAutofilledHighlight(canvas);

    // Overlay is part of the content and draws beneath Foreground
    if (mOverlay != null && !mOverlay.isEmpty()) {
        mOverlay.getOverlayView().dispatchDraw(canvas);
    }

    // Step 6, draw decorations (foreground, scrollbars)
    onDrawForeground(canvas);

    // Step 7, draw the default focus highlight
    drawDefaultFocusHighlight(canvas);

    if (debugDraw()) {
        debugDrawFocus(canvas);
    }

    // we're done...
    return;
}

/*
 * Here we do the full fledged routine...
 * (this is an uncommon case where speed matters less,
 * this is why we repeat some of the tests that have been
 * done above)
 */

boolean drawTop = false;
boolean drawBottom = false;
boolean drawLeft = false;
boolean drawRight = false;

float topFadeStrength = 0.0f;
float bottomFadeStrength = 0.0f;
float leftFadeStrength = 0.0f;
float rightFadeStrength = 0.0f;

// Step 2, save the canvas' layers
int paddingLeft = mPaddingLeft;

final boolean offsetRequired = isPaddingOffsetRequired();
if (offsetRequired) {
    paddingLeft += getLeftPaddingOffset();
}

int left = mScrollX + paddingLeft;
int right = left + mRight - mLeft - mPaddingRight - paddingLeft;
int top = mScrollY + getFadeTop(offsetRequired);
int bottom = top + getFadeHeight(offsetRequired);

if (offsetRequired) {
    right += getRightPaddingOffset();
    bottom += getBottomPaddingOffset();
}

final ScrollabilityCache scrollabilityCache = mScrollCache;
final float fadeHeight = scrollabilityCache.fadingEdgeLength;
int length = (int) fadeHeight;

// clip the fade length if top and bottom fades overlap
// overlapping fades produce odd-looking artifacts
if (verticalEdges && (top + length > bottom - length)) {
    length = (bottom - top) / 2;
}

// also clip horizontal fades if necessary
if (horizontalEdges && (left + length > right - length)) {
    length = (right - left) / 2;
}
```

```
if (verticalEdges) {
    topFadeStrength = Math.max(0.0f, Math.min(1.0f, getTopFadingEdgeStrength()));
    drawTop = topFadeStrength * fadeHeight > 1.0f;
    bottomFadeStrength = Math.max(0.0f, Math.min(1.0f, getBottomFadingEdgeStrength()));
    drawBottom = bottomFadeStrength * fadeHeight > 1.0f;
}

if (horizontalEdges) {
    leftFadeStrength = Math.max(0.0f, Math.min(1.0f, getLeftFadingEdgeStrength()));
    drawLeft = leftFadeStrength * fadeHeight > 1.0f;
    rightFadeStrength = Math.max(0.0f, Math.min(1.0f, getRightFadingEdgeStrength()));
    drawRight = rightFadeStrength * fadeHeight > 1.0f;
}

saveCount = canvas.getSaveCount();

int solidColor = getSolidColor();
if (solidColor == 0) {
    final int flags = Canvas.HAS_ALPHA_LAYER_SAVE_FLAG;

    if (drawTop) {
        canvas.saveLayer(left, top, right, top + length, null, flags);
    }

    if (drawBottom) {
        canvas.saveLayer(left, bottom - length, right, bottom, null, flags);
    }

    if (drawLeft) {
        canvas.saveLayer(left, top, left + length, bottom, null, flags);
    }

    if (drawRight) {
        canvas.saveLayer(right - length, top, right, bottom, null, flags);
    }
} else {
    scrollabilityCache.setFadeColor(solidColor);
}

// Step 3, draw the content
if (!dirtyOpaque) onDraw(canvas);

// Step 4, draw the children
dispatchDraw(canvas);

// Step 5, draw the fade effect and restore layers
final Paint p = scrollabilityCache.paint;
final Matrix matrix = scrollabilityCache.matrix;
final Shader fade = scrollabilityCache.shader;

if (drawTop) {
    matrix.setScale(1, fadeHeight * topFadeStrength);
    matrix.postTranslate(left, top);
    fade.setLocalMatrix(matrix);
    p.setShader(fade);
    canvas.drawRect(left, top, right, top + length, p);
}

if (drawBottom) {
    matrix.setScale(1, fadeHeight * bottomFadeStrength);
    matrix.postRotate(180);
    matrix.postTranslate(left, bottom);
    fade.setLocalMatrix(matrix);
    p.setShader(fade);
    canvas.drawRect(left, bottom - length, right, bottom, p);
}

if (drawLeft) {
    matrix.setScale(1, fadeHeight * leftFadeStrength);
    matrix.postRotate(-90);
    matrix.postTranslate(left, top);
    fade.setLocalMatrix(matrix);
    p.setShader(fade);
    canvas.drawRect(left, top, left + length, bottom, p);
}

if (drawRight) {
    matrix.setScale(1, fadeHeight * rightFadeStrength);
    matrix.postRotate(90);
    matrix.postTranslate(right, top);
    fade.setLocalMatrix(matrix);
    p.setShader(fade);
```

```
            canvas.drawRect(right - length, top, right, bottom, p);
        }

        canvas.restoreToCount(saveCount);

        drawAutofilledHighlight(canvas);

        // Overlay is part of the content and draws beneath Foreground
        if (mOverlay != null && !mOverlay.isEmpty()) {
            mOverlay.getOverlayView().dispatchDraw(canvas);
        }

        // Step 6, draw decorations (foreground, scrollbars)
        onDrawForeground(canvas);

        if (debugDraw()) {
            debugDrawFocus(canvas);
        }
    }

    /**
     * Draws the background onto the specified canvas.
     *
     * @param canvas Canvas on which to draw the background
     */
    private void drawBackground(Canvas canvas) {
        final Drawable background = mBackground;
        if (background == null) {
            return;
        }

        setBackgroundBounds();

        // Attempt to use a display list if requested.
        if (canvas.isHardwareAccelerated() && mAttachInfo != null
                && mAttachInfo.mThreadedRenderer != null) {
            mBackgroundRenderNode = getDrawableRenderNode(background, mBackgroundRenderNode);

            final RenderNode renderNode = mBackgroundRenderNode;
            if (renderNode != null && renderNode.isValid()) {
                setBackgroundRenderNodeProperties(renderNode);
                ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
                return;
            }
        }

        final int scrollX = mScrollX;
        final int scrollY = mScrollY;
        if ((scrollX | scrollY) == 0) {
            background.draw(canvas);
        } else {
            canvas.translate(scrollX, scrollY);
            background.draw(canvas);
            canvas.translate(-scrollX, -scrollY);
        }
    }

    /**
     * Sets the correct background bounds and rebuilds the outline, if needed.
     * <p/>
     * This is called by LayoutLib.
     */
    void setBackgroundBounds() {
        if (mBackgroundSizeChanged && mBackground != null) {
            mBackground.setBounds(0, 0, mRight - mLeft, mBottom - mTop);
            mBackgroundSizeChanged = false;
            rebuildOutline();
        }
    }

    private void setBackgroundRenderNodeProperties(RenderNode renderNode) {
        renderNode.setTranslationX(mScrollX);
        renderNode.setTranslationY(mScrollY);
    }

    /**
     * Creates a new display list or updates the existing display list for the
     * specified Drawable.
     *
     * @param drawable Drawable for which to create a display list
     * @param renderNode Existing RenderNode, or {@code null}
     * @return A valid display list for the specified drawable
```

```
 */
private RenderNode getDrawableRenderNode(Drawable drawable, RenderNode renderNode) {
    if (renderNode == null) {
        renderNode = RenderNode.create(drawable.getClass().getName(), this);
    }

    final Rect bounds = drawable.getBounds();
    final int width = bounds.width();
    final int height = bounds.height();
    final DisplayListCanvas canvas = renderNode.start(width, height);

    // Reverse left/top translation done by drawable canvas, which will
    // instead be applied by rendernode's LTRB bounds below. This way, the
    // drawable's bounds match with its rendernode bounds and its content
    // will lie within those bounds in the rendernode tree.
    canvas.translate(-bounds.left, -bounds.top);

    try {
        drawable.draw(canvas);
    } finally {
        renderNode.end(canvas);
    }

    // Set up drawable properties that are view-independent.
    renderNode.setLeftTopRightBottom(bounds.left, bounds.top, bounds.right, bounds.bottom);
    renderNode.setProjectBackwards(drawable.isProjected());
    renderNode.setProjectionReceiver(true);
    renderNode.setClipToBounds(false);
    return renderNode;
}

/**
 * Returns the overlay for this view, creating it if it does not yet exist.
 * Adding drawables to the overlay will cause them to be displayed whenever
 * the view itself is redrawn. Objects in the overlay should be actively
 * managed: remove them when they should not be displayed anymore. The
 * overlay will always have the same size as its host view.
 *
 * <p>Note: Overlays do not currently work correctly with {@link
 * SurfaceView} or {@link TextureView}; contents in overlays for these
 * types of views may not display correctly.</p>
 *
 * @return The ViewOverlay object for this view.
 * @see ViewOverlay
 */
public ViewOverlay getOverlay() {
    if (mOverlay == null) {
        mOverlay = new ViewOverlay(mContext, this);
    }
    return mOverlay;
}

/**
 * Override this if your view is known to always be drawn on top of a solid color background,
 * and needs to draw fading edges. Returning a non-zero color enables the view system to
 * optimize the drawing of the fading edges. If you do return a non-zero color, the alpha
 * should be set to 0xFF.
 *
 * @see #setVerticalFadingEdgeEnabled(boolean)
 * @see #setHorizontalFadingEdgeEnabled(boolean)
 *
 * @return The known solid color background for this view, or 0 if the color may vary
 */
@ViewDebug.ExportedProperty(category = "drawing")
@ColorInt
public int getSolidColor() {
    return 0;
}

/**
 * Build a human readable string representation of the specified view flags.
 *
 * @param flags the view flags to convert to a string
 * @return a String representing the supplied flags
 */
private static String printFlags(int flags) {
    String output = "";
    int numFlags = 0;
    if ((flags & FOCUSABLE) == FOCUSABLE) {
        output += "TAKES_FOCUS";
        numFlags++;
    }
```

```
    switch (flags & VISIBILITY_MASK) {
    case INVISIBLE:
        if (numFlags > 0) {
            output += " ";
        }
        output += "INVISIBLE";
        // USELESS HERE numFlags++;
        break;
    case GONE:
        if (numFlags > 0) {
            output += " ";
        }
        output += "GONE";
        // USELESS HERE numFlags++;
        break;
    default:
        break;
    }
    return output;
}

/**
 * Build a human readable string representation of the specified private
 * view flags.
 *
 * @param privateFlags the private view flags to convert to a string
 * @return a String representing the supplied flags
 */
private static String printPrivateFlags(int privateFlags) {
    String output = "";
    int numFlags = 0;

    if ((privateFlags & PFLAG_WANTS_FOCUS) == PFLAG_WANTS_FOCUS) {
        output += "WANTS_FOCUS";
        numFlags++;
    }

    if ((privateFlags & PFLAG_FOCUSED) == PFLAG_FOCUSED) {
        if (numFlags > 0) {
            output += " ";
        }
        output += "FOCUSED";
        numFlags++;
    }

    if ((privateFlags & PFLAG_SELECTED) == PFLAG_SELECTED) {
        if (numFlags > 0) {
            output += " ";
        }
        output += "SELECTED";
        numFlags++;
    }

    if ((privateFlags & PFLAG_IS_ROOT_NAMESPACE) == PFLAG_IS_ROOT_NAMESPACE) {
        if (numFlags > 0) {
            output += " ";
        }
        output += "IS_ROOT_NAMESPACE";
        numFlags++;
    }

    if ((privateFlags & PFLAG_HAS_BOUNDS) == PFLAG_HAS_BOUNDS) {
        if (numFlags > 0) {
            output += " ";
        }
        output += "HAS_BOUNDS";
        numFlags++;
    }

    if ((privateFlags & PFLAG_DRAWN) == PFLAG_DRAWN) {
        if (numFlags > 0) {
            output += " ";
        }
        output += "DRAWN";
        // USELESS HERE numFlags++;
    }
    return output;
}

/**
 * <p>Indicates whether or not this view's layout will be requested during
```

```
     * the next hierarchy layout pass.</p>
     *
     * @return true if the layout will be forced during next layout pass
     */
    public boolean isLayoutRequested() {
        return (mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT;
    }

    /**
     * Return true if o is a ViewGroup that is laying out using optical bounds.
     * @hide
     */
    public static boolean isLayoutModeOptical(Object o) {
        return o instanceof ViewGroup && ((ViewGroup) o).isLayoutModeOptical();
    }

    private boolean setOpticalFrame(int left, int top, int right, int bottom) {
        Insets parentInsets = mParent instanceof View ?
                ((View) mParent).getOpticalInsets() : Insets.NONE;
        Insets childInsets = getOpticalInsets();
        return setFrame(
                left   + parentInsets.left - childInsets.left,
                top    + parentInsets.top  - childInsets.top,
                right  + parentInsets.left + childInsets.right,
                bottom + parentInsets.top  + childInsets.bottom);
    }

    /**
     * Assign a size and position to a view and all of its
     * descendants
     *
     * <p>This is the second phase of the layout mechanism.
     * (The first is measuring). In this phase, each parent calls
     * layout on all of its children to position them.
     * This is typically done using the child measurements
     * that were stored in the measure pass().</p>
     *
     * <p>Derived classes should not override this method.
     * Derived classes with children should override
     * onLayout. In that method, they should
     * call layout on each of their children.</p>
     *
     * @param l Left position, relative to parent
     * @param t Top position, relative to parent
     * @param r Right position, relative to parent
     * @param b Bottom position, relative to parent
     */
    @SuppressWarnings({"unchecked"})
    public void layout(int l, int t, int r, int b) {
        if ((mPrivateFlags3 & PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT) != 0) {
            onMeasure(mOldWidthMeasureSpec, mOldHeightMeasureSpec);
            mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
        }

        int oldL = mLeft;
        int oldT = mTop;
        int oldB = mBottom;
        int oldR = mRight;

        boolean changed = isLayoutModeOptical(mParent) ?
                setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

        if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_REQUIRED) {
            onLayout(changed, l, t, r, b);

            if (shouldDrawRoundScrollbar()) {
                if(mRoundScrollbarRenderer == null) {
                    mRoundScrollbarRenderer = new RoundScrollbarRenderer(this);
                }
            } else {
                mRoundScrollbarRenderer = null;
            }

            mPrivateFlags &= ~PFLAG_LAYOUT_REQUIRED;

            ListenerInfo li = mListenerInfo;
            if (li != null && li.mOnLayoutChangeListeners != null) {
                ArrayList<OnLayoutChangeListener> listenersCopy =
                        (ArrayList<OnLayoutChangeListener>)li.mOnLayoutChangeListeners.clone();
                int numListeners = listenersCopy.size();
                for (int i = 0; i < numListeners; ++i) {
                    listenersCopy.get(i).onLayoutChange(this, l, t, r, b, oldL, oldT, oldR, oldB);
```

```
            }
        }
    }

    mPrivateFlags &= ~PFLAG_FORCE_LAYOUT;
    mPrivateFlags3 |= PFLAG3_IS_LAID_OUT;

    if ((mPrivateFlags3 & PFLAG3_NOTIFY_AUTOFILL_ENTER_ON_LAYOUT) != 0) {
        mPrivateFlags3 &= ~PFLAG3_NOTIFY_AUTOFILL_ENTER_ON_LAYOUT;
        notifyEnterOrExitForAutoFillIfNeeded(true);
    }
}

/**
 * Called from layout when this view should
 * assign a size and position to each of its children.
 *
 * Derived classes with children should override
 * this method and call layout on each of
 * their children.
 * @param changed This is a new size or position for this view
 * @param left Left position, relative to parent
 * @param top Top position, relative to parent
 * @param right Right position, relative to parent
 * @param bottom Bottom position, relative to parent
 */
protected void onLayout(boolean changed, int left, int top, int right, int bottom) {
}

/**
 * Assign a size and position to this view.
 *
 * This is called from layout.
 *
 * @param left Left position, relative to parent
 * @param top Top position, relative to parent
 * @param right Right position, relative to parent
 * @param bottom Bottom position, relative to parent
 * @return true if the new size and position are different than the
 *         previous ones
 * {@hide}
 */
protected boolean setFrame(int left, int top, int right, int bottom) {
    boolean changed = false;

    if (DBG) {
        Log.d("View", this + " View.setFrame(" + left + "," + top + ","
                + right + "," + bottom + ")");
    }

    if (mLeft != left || mRight != right || mTop != top || mBottom != bottom) {
        changed = true;

        // Remember our drawn bit
        int drawn = mPrivateFlags & PFLAG_DRAWN;

        int oldWidth = mRight - mLeft;
        int oldHeight = mBottom - mTop;
        int newWidth = right - left;
        int newHeight = bottom - top;
        boolean sizeChanged = (newWidth != oldWidth) || (newHeight != oldHeight);

        // Invalidate our old position
        invalidate(sizeChanged);

        mLeft = left;
        mTop = top;
        mRight = right;
        mBottom = bottom;
        mRenderNode.setLeftTopRightBottom(mLeft, mTop, mRight, mBottom);

        mPrivateFlags |= PFLAG_HAS_BOUNDS;


        if (sizeChanged) {
            sizeChange(newWidth, newHeight, oldWidth, oldHeight);
        }

        if ((mViewFlags & VISIBILITY_MASK) == VISIBLE || mGhostView != null) {
            // If we are visible, force the DRAWN bit to on so that
            // this invalidate will go through (at least to our parent).
            // This is because someone may have invalidated this view
```

```
            // before this call to setFrame came in, thereby clearing
            // the DRAWN bit.
            mPrivateFlags |= PFLAG_DRAWN;
            invalidate(sizeChanged);
            // parent display list may need to be recreated based on a change in the bounds
            // of any child
            invalidateParentCaches();
        }

        // Reset drawn bit to original value (invalidate turns it off)
        mPrivateFlags |= drawn;

        mBackgroundSizeChanged = true;
        mDefaultFocusHighlightSizeChanged = true;
        if (mForegroundInfo != null) {
            mForegroundInfo.mBoundsChanged = true;
        }

        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
    return changed;
}

/**
 * Same as setFrame, but public and hidden. For use in {@link android.transition.ChangeBounds}.
 * @hide
 */
public void setLeftTopRightBottom(int left, int top, int right, int bottom) {
    setFrame(left, top, right, bottom);
}

private void sizeChange(int newWidth, int newHeight, int oldWidth, int oldHeight) {
    onSizeChanged(newWidth, newHeight, oldWidth, oldHeight);
    if (mOverlay != null) {
        mOverlay.getOverlayView().setRight(newWidth);
        mOverlay.getOverlayView().setBottom(newHeight);
    }
    rebuildOutline();
}

/**
 * Finalize inflating a view from XML.  This is called as the last phase
 * of inflation, after all child views have been added.
 *
 * <p>Even if the subclass overrides onFinishInflate, they should always be
 * sure to call the super method, so that we get called.
 */
@CallSuper
protected void onFinishInflate() {
}

/**
 * Returns the resources associated with this view.
 *
 * @return Resources object.
 */
public Resources getResources() {
    return mResources;
}

/**
 * Invalidates the specified Drawable.
 *
 * @param drawable the drawable to invalidate
 */
@Override
public void invalidateDrawable(@NonNull Drawable drawable) {
    if (verifyDrawable(drawable)) {
        final Rect dirty = drawable.getDirtyBounds();
        final int scrollX = mScrollX;
        final int scrollY = mScrollY;

        invalidate(dirty.left + scrollX, dirty.top + scrollY,
                dirty.right + scrollX, dirty.bottom + scrollY);
        rebuildOutline();
    }
}

/**
 * Schedules an action on a drawable to occur at a specified time.
 *
 * @param who the recipient of the action
```

```
     * @param what the action to run on the drawable
     * @param when the time at which the action must occur. Uses the
     *        {@link SystemClock#uptimeMillis} timebase.
     */
    @Override
    public void scheduleDrawable(@NonNull Drawable who, @NonNull Runnable what, long when) {
        if (verifyDrawable(who) && what != null) {
            final long delay = when - SystemClock.uptimeMillis();
            if (mAttachInfo != null) {
                mAttachInfo.mViewRootImpl.mChoreographer.postCallbackDelayed(
                        Choreographer.CALLBACK_ANIMATION, what, who,
                        Choreographer.subtractFrameDelay(delay));
            } else {
                // Postpone the runnable until we know
                // on which thread it needs to run.
                getRunQueue().postDelayed(what, delay);
            }
        }
    }

    /**
     * Cancels a scheduled action on a drawable.
     *
     * @param who the recipient of the action
     * @param what the action to cancel
     */
    @Override
    public void unscheduleDrawable(@NonNull Drawable who, @NonNull Runnable what) {
        if (verifyDrawable(who) && what != null) {
            if (mAttachInfo != null) {
                mAttachInfo.mViewRootImpl.mChoreographer.removeCallbacks(
                        Choreographer.CALLBACK_ANIMATION, what, who);
            }
            getRunQueue().removeCallbacks(what);
        }
    }

    /**
     * Unschedule any events associated with the given Drawable.  This can be
     * used when selecting a new Drawable into a view, so that the previous
     * one is completely unscheduled.
     *
     * @param who The Drawable to unschedule.
     *
     * @see #drawableStateChanged
     */
    public void unscheduleDrawable(Drawable who) {
        if (mAttachInfo != null && who != null) {
            mAttachInfo.mViewRootImpl.mChoreographer.removeCallbacks(
                    Choreographer.CALLBACK_ANIMATION, null, who);
        }
    }

    /**
     * Resolve the Drawables depending on the layout direction. This is implicitly supposing
     * that the View directionality can and will be resolved before its Drawables.
     *
     * Will call {@link View#onResolveDrawables} when resolution is done.
     *
     * @hide
     */
    protected void resolveDrawables() {
        // Drawables resolution may need to happen before resolving the layout direction (which is
        // done only during the measure() call).
        // If the layout direction is not resolved yet, we cannot resolve the Drawables except in
        // one case: when the raw layout direction has not been defined as LAYOUT_DIRECTION_INHERIT.
        // So, if the raw layout direction is LAYOUT_DIRECTION_LTR or LAYOUT_DIRECTION_RTL or
        // LAYOUT_DIRECTION_LOCALE, we can "cheat" and we don't need to wait for the layout
        // direction to be resolved as its resolved value will be the same as its raw value.
        if (!isLayoutDirectionResolved() &&
                getRawLayoutDirection() == View.LAYOUT_DIRECTION_INHERIT) {
            return;
        }

        final int layoutDirection = isLayoutDirectionResolved() ?
                getLayoutDirection() : getRawLayoutDirection();

        if (mBackground != null) {
            mBackground.setLayoutDirection(layoutDirection);
        }
        if (mForegroundInfo != null && mForegroundInfo.mDrawable != null) {
            mForegroundInfo.mDrawable.setLayoutDirection(layoutDirection);
```

```
    }
    if (mDefaultFocusHighlight != null) {
        mDefaultFocusHighlight.setLayoutDirection(layoutDirection);
    }
    mPrivateFlags2 |= PFLAG2_DRAWABLE_RESOLVED;
    onResolveDrawables(layoutDirection);
}

boolean areDrawablesResolved() {
    return (mPrivateFlags2 & PFLAG2_DRAWABLE_RESOLVED) == PFLAG2_DRAWABLE_RESOLVED;
}

/**
 * Called when layout direction has been resolved.
 *
 * The default implementation does nothing.
 *
 * @param layoutDirection The resolved layout direction.
 *
 * @see #LAYOUT_DIRECTION_LTR
 * @see #LAYOUT_DIRECTION_RTL
 *
 * @hide
 */
public void onResolveDrawables(@ResolvedLayoutDir int layoutDirection) {
}

/**
 * @hide
 */
protected void resetResolvedDrawables() {
    resetResolvedDrawablesInternal();
}

void resetResolvedDrawablesInternal() {
    mPrivateFlags2 &= ~PFLAG2_DRAWABLE_RESOLVED;
}

/**
 * If your view subclass is displaying its own Drawable objects, it should
 * override this function and return true for any Drawable it is
 * displaying.  This allows animations for those drawables to be
 * scheduled.
 *
 * <p>Be sure to call through to the super class when overriding this
 * function.
 *
 * @param who The Drawable to verify.  Return true if it is one you are
 *            displaying, else return the result of calling through to the
 *            super class.
 *
 * @return boolean If true than the Drawable is being displayed in the
 *         view; else false and it is not allowed to animate.
 *
 * @see #unscheduleDrawable(android.graphics.drawable.Drawable)
 * @see #drawableStateChanged()
 */
@CallSuper
protected boolean verifyDrawable(@NonNull Drawable who) {
    // Avoid verifying the scroll bar drawable so that we don't end up in
    // an invalidation loop. This effectively prevents the scroll bar
    // drawable from triggering invalidations and scheduling runnables.
    return who == mBackground || (mForegroundInfo != null && mForegroundInfo.mDrawable == who)
            || (mDefaultFocusHighlight == who);
}

/**
 * This function is called whenever the state of the view changes in such
 * a way that it impacts the state of drawables being shown.
 * <p>
 * If the View has a StateListAnimator, it will also be called to run necessary state
 * change animations.
 * <p>
 * Be sure to call through to the superclass when overriding this function.
 *
 * @see Drawable#setState(int[])
 */
@CallSuper
protected void drawableStateChanged() {
    final int[] state = getDrawableState();
    boolean changed = false;
```

```
    final Drawable bg = mBackground;
    if (bg != null && bg.isStateful()) {
        changed |= bg.setState(state);
    }

    final Drawable hl = mDefaultFocusHighlight;
    if (hl != null && hl.isStateful()) {
        changed |= hl.setState(state);
    }

    final Drawable fg = mForegroundInfo != null ? mForegroundInfo.mDrawable : null;
    if (fg != null && fg.isStateful()) {
        changed |= fg.setState(state);
    }

    if (mScrollCache != null) {
        final Drawable scrollBar = mScrollCache.scrollBar;
        if (scrollBar != null && scrollBar.isStateful()) {
            changed |= scrollBar.setState(state)
                    && mScrollCache.state != ScrollabilityCache.OFF;
        }
    }

    if (mStateListAnimator != null) {
        mStateListAnimator.setState(state);
    }

    if (changed) {
        invalidate();
    }
}

/**
 * This function is called whenever the view hotspot changes and needs to
 * be propagated to drawables or child views managed by the view.
 * <p>
 * Dispatching to child views is handled by
 * {@link #dispatchDrawableHotspotChanged(float, float)}.
 * <p>
 * Be sure to call through to the superclass when overriding this function.
 *
 * @param x hotspot x coordinate
 * @param y hotspot y coordinate
 */
@CallSuper
public void drawableHotspotChanged(float x, float y) {
    if (mBackground != null) {
        mBackground.setHotspot(x, y);
    }
    if (mDefaultFocusHighlight != null) {
        mDefaultFocusHighlight.setHotspot(x, y);
    }
    if (mForegroundInfo != null && mForegroundInfo.mDrawable != null) {
        mForegroundInfo.mDrawable.setHotspot(x, y);
    }

    dispatchDrawableHotspotChanged(x, y);
}

/**
 * Dispatches drawableHotspotChanged to all of this View's children.
 *
 * @param x hotspot x coordinate
 * @param y hotspot y coordinate
 * @see #drawableHotspotChanged(float, float)
 */
public void dispatchDrawableHotspotChanged(float x, float y) {
}

/**
 * Call this to force a view to update its drawable state. This will cause
 * drawableStateChanged to be called on this view. Views that are interested
 * in the new state should call getDrawableState.
 *
 * @see #drawableStateChanged
 * @see #getDrawableState
 */
public void refreshDrawableState() {
    mPrivateFlags |= PFLAG_DRAWABLE_STATE_DIRTY;
    drawableStateChanged();

    ViewParent parent = mParent;
```

```
    if (parent != null) {
        parent.childDrawableStateChanged(this);
    }
}

/**
 * Create a default focus highlight if it doesn't exist.
 * @return a default focus highlight.
 */
private Drawable getDefaultFocusHighlightDrawable() {
    if (mDefaultFocusHighlightCache == null) {
        if (mContext != null) {
            final int[] attrs = new int[] { android.R.attr.selectableItemBackground };
            final TypedArray ta = mContext.obtainStyledAttributes(attrs);
            mDefaultFocusHighlightCache = ta.getDrawable(0);
            ta.recycle();
        }
    }
    return mDefaultFocusHighlightCache;
}

/**
 * Set the current default focus highlight.
 * @param highlight the highlight drawable, or {@code null} if it's no longer needed.
 */
private void setDefaultFocusHighlight(Drawable highlight) {
    mDefaultFocusHighlight = highlight;
    mDefaultFocusHighlightSizeChanged = true;
    if (highlight != null) {
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) != 0) {
            mPrivateFlags &= ~PFLAG_SKIP_DRAW;
        }
        highlight.setLayoutDirection(getLayoutDirection());
        if (highlight.isStateful()) {
            highlight.setState(getDrawableState());
        }
        if (isAttachedToWindow()) {
            highlight.setVisible(getWindowVisibility() == VISIBLE && isShown(), false);
        }
        // Set callback last, since the view may still be initializing.
        highlight.setCallback(this);
    } else if ((mViewFlags & WILL_NOT_DRAW) != 0 && mBackground == null
            && (mForegroundInfo == null || mForegroundInfo.mDrawable == null)) {
        mPrivateFlags |= PFLAG_SKIP_DRAW;
    }
    invalidate();
}

/**
 * Check whether we need to draw a default focus highlight when this view gets focused,
 * which requires:
 * <ul>
 *     <li>In both background and foreground, {@link android.R.attr#state_focused}
 *         is not defined.</li>
 *     <li>This view is not in touch mode.</li>
 *     <li>This view doesn't opt out for a default focus highlight, via
 *         {@link #setDefaultFocusHighlightEnabled(boolean)}.</li>
 *     <li>This view is attached to window.</li>
 * </ul>
 * @return {@code true} if a default focus highlight is needed.
 * @hide
 */
@TestApi
public boolean isDefaultFocusHighlightNeeded(Drawable background, Drawable foreground) {
    final boolean lackFocusState = (background == null || !background.isStateful()
            || !background.hasFocusStateSpecified())
            && (foreground == null || !foreground.isStateful()
            || !foreground.hasFocusStateSpecified());
    return !isInTouchMode() && getDefaultFocusHighlightEnabled() && lackFocusState
            && isAttachedToWindow() && sUseDefaultFocusHighlight;
}

/**
 * When this view is focused, switches on/off the default focused highlight.
 * <p>
 * This always happens when this view is focused, and only at this moment the default focus
 * highlight can be visible.
 */
private void switchDefaultFocusHighlight() {
    if (isFocused()) {
        final boolean needed = isDefaultFocusHighlightNeeded(mBackground,
                mForegroundInfo == null ? null : mForegroundInfo.mDrawable);
```

```
        final boolean active = mDefaultFocusHighlight != null;
        if (needed && !active) {
            setDefaultFocusHighlight(getDefaultFocusHighlightDrawable());
        } else if (!needed && active) {
            // The highlight is no longer needed, so tear it down.
            setDefaultFocusHighlight(null);
        }
    }
}

/**
 * Draw the default focus highlight onto the canvas.
 * @param canvas the canvas where we're drawing the highlight.
 */
private void drawDefaultFocusHighlight(Canvas canvas) {
    if (mDefaultFocusHighlight != null) {
        if (mDefaultFocusHighlightSizeChanged) {
            mDefaultFocusHighlightSizeChanged = false;
            final int l = mScrollX;
            final int r = l + mRight - mLeft;
            final int t = mScrollY;
            final int b = t + mBottom - mTop;
            mDefaultFocusHighlight.setBounds(l, t, r, b);
        }
        mDefaultFocusHighlight.draw(canvas);
    }
}

/**
 * Return an array of resource IDs of the drawable states representing the
 * current state of the view.
 *
 * @return The current drawable state
 *
 * @see Drawable#setState(int[])
 * @see #drawableStateChanged()
 * @see #onCreateDrawableState(int)
 */
public final int[] getDrawableState() {
    if ((mDrawableState != null) && ((mPrivateFlags & PFLAG_DRAWABLE_STATE_DIRTY) == 0)) {
        return mDrawableState;
    } else {
        mDrawableState = onCreateDrawableState(0);
        mPrivateFlags &= ~PFLAG_DRAWABLE_STATE_DIRTY;
        return mDrawableState;
    }
}

/**
 * Generate the new {@link android.graphics.drawable.Drawable} state for
 * this view. This is called by the view
 * system when the cached Drawable state is determined to be invalid.  To
 * retrieve the current state, you should use {@link #getDrawableState}.
 *
 * @param extraSpace if non-zero, this is the number of extra entries you
 * would like in the returned array in which you can place your own
 * states.
 *
 * @return Returns an array holding the current {@link Drawable} state of
 * the view.
 *
 * @see #mergeDrawableStates(int[], int[])
 */
protected int[] onCreateDrawableState(int extraSpace) {
    if ((mViewFlags & DUPLICATE_PARENT_STATE) == DUPLICATE_PARENT_STATE &&
            mParent instanceof View) {
        return ((View) mParent).onCreateDrawableState(extraSpace);
    }

    int[] drawableState;

    int privateFlags = mPrivateFlags;

    int viewStateIndex = 0;
    if ((privateFlags & PFLAG_PRESSED) != 0) viewStateIndex |= StateSet.VIEW_STATE_PRESSED;
    if ((mViewFlags & ENABLED_MASK) == ENABLED) viewStateIndex |= StateSet.VIEW_STATE_ENABLED;
    if (isFocused()) viewStateIndex |= StateSet.VIEW_STATE_FOCUSED;
    if ((privateFlags & PFLAG_SELECTED) != 0) viewStateIndex |= StateSet.VIEW_STATE_SELECTED;
    if (hasWindowFocus()) viewStateIndex |= StateSet.VIEW_STATE_WINDOW_FOCUSED;
    if ((privateFlags & PFLAG_ACTIVATED) != 0) viewStateIndex |= StateSet.VIEW_STATE_ACTIVATED;
    if (mAttachInfo != null && mAttachInfo.mHardwareAccelerationRequested &&
            ThreadedRenderer.isAvailable()) {
```

```
            // This is set if HW acceleration is requested, even if the current
            // process doesn't allow it.  This is just to allow app preview
            // windows to better match their app.
            viewStateIndex |= StateSet.VIEW_STATE_ACCELERATED;
        }
        if ((privateFlags & PFLAG_HOVERED) != 0) viewStateIndex |= StateSet.VIEW_STATE_HOVERED;

        final int privateFlags2 = mPrivateFlags2;
        if ((privateFlags2 & PFLAG2_DRAG_CAN_ACCEPT) != 0) {
            viewStateIndex |= StateSet.VIEW_STATE_DRAG_CAN_ACCEPT;
        }
        if ((privateFlags2 & PFLAG2_DRAG_HOVERED) != 0) {
            viewStateIndex |= StateSet.VIEW_STATE_DRAG_HOVERED;
        }

        drawableState = StateSet.get(viewStateIndex);

        //noinspection ConstantIfStatement
        if (false) {
            Log.i("View", "drawableStateIndex=" + viewStateIndex);
            Log.i("View", toString()
                    + " pressed=" + ((privateFlags & PFLAG_PRESSED) != 0)
                    + " en=" + ((mViewFlags & ENABLED_MASK) == ENABLED)
                    + " fo=" + hasFocus()
                    + " sl=" + ((privateFlags & PFLAG_SELECTED) != 0)
                    + " wf=" + hasWindowFocus()
                    + ": " + Arrays.toString(drawableState));
        }

        if (extraSpace == 0) {
            return drawableState;
        }

        final int[] fullState;
        if (drawableState != null) {
            fullState = new int[drawableState.length + extraSpace];
            System.arraycopy(drawableState, 0, fullState, 0, drawableState.length);
        } else {
            fullState = new int[extraSpace];
        }

        return fullState;
    }

    /**
     * Merge your own state values in <var>additionalState</var> into the base
     * state values <var>baseState</var> that were returned by
     * {@link #onCreateDrawableState(int)}.
     *
     * @param baseState The base state values returned by
     * {@link #onCreateDrawableState(int)}, which will be modified to also hold your
     * own additional state values.
     *
     * @param additionalState The additional state values you would like
     * added to <var>baseState</var>; this array is not modified.
     *
     * @return As a convenience, the <var>baseState</var> array you originally
     * passed into the function is returned.
     *
     * @see #onCreateDrawableState(int)
     */
    protected static int[] mergeDrawableStates(int[] baseState, int[] additionalState) {
        final int N = baseState.length;
        int i = N - 1;
        while (i >= 0 && baseState[i] == 0) {
            i--;
        }
        System.arraycopy(additionalState, 0, baseState, i + 1, additionalState.length);
        return baseState;
    }

    /**
     * Call {@link Drawable#jumpToCurrentState() Drawable.jumpToCurrentState()}
     * on all Drawable objects associated with this view.
     * <p>
     * Also calls {@link StateListAnimator#jumpToCurrentState()} if there is a StateListAnimator
     * attached to this view.
     */
    @CallSuper
    public void jumpDrawablesToCurrentState() {
        if (mBackground != null) {
            mBackground.jumpToCurrentState();
```

```
    }
    if (mStateListAnimator != null) {
        mStateListAnimator.jumpToCurrentState();
    }
    if (mDefaultFocusHighlight != null) {
        mDefaultFocusHighlight.jumpToCurrentState();
    }
    if (mForegroundInfo != null && mForegroundInfo.mDrawable != null) {
        mForegroundInfo.mDrawable.jumpToCurrentState();
    }
}

/**
 * Sets the background color for this view.
 * @param color the color of the background
 */
@RemotableViewMethod
public void setBackgroundColor(@ColorInt int color) {
    if (mBackground instanceof ColorDrawable) {
        ((ColorDrawable) mBackground.mutate()).setColor(color);
        computeOpaqueFlags();
        mBackgroundResource = 0;
    } else {
        setBackground(new ColorDrawable(color));
    }
}

/**
 * Set the background to a given resource. The resource should refer to
 * a Drawable object or 0 to remove the background.
 * @param resid The identifier of the resource.
 *
 * @attr ref android.R.styleable#View_background
 */
@RemotableViewMethod
public void setBackgroundResource(@DrawableRes int resid) {
    if (resid != 0 && resid == mBackgroundResource) {
        return;
    }

    Drawable d = null;
    if (resid != 0) {
        d = mContext.getDrawable(resid);
    }
    setBackground(d);

    mBackgroundResource = resid;
}

/**
 * Set the background to a given Drawable, or remove the background. If the
 * background has padding, this View's padding is set to the background's
 * padding. However, when a background is removed, this View's padding isn't
 * touched. If setting the padding is desired, please use
 * {@link #setPadding(int, int, int, int)}.
 *
 * @param background The Drawable to use as the background, or null to remove the
 *        background
 */
public void setBackground(Drawable background) {
    //noinspection deprecation
    setBackgroundDrawable(background);
}

/**
 * @deprecated use {@link #setBackground(Drawable)} instead
 */
@Deprecated
public void setBackgroundDrawable(Drawable background) {
    computeOpaqueFlags();

    if (background == mBackground) {
        return;
    }

    boolean requestLayout = false;

    mBackgroundResource = 0;

    /*
     * Regardless of whether we're setting a new background or not, we want
     * to clear the previous drawable. setVisible first while we still have the callback set.
```

```
     */
    if (mBackground != null) {
        if (isAttachedToWindow()) {
            mBackground.setVisible(false, false);
        }
        mBackground.setCallback(null);
        unscheduleDrawable(mBackground);
    }

    if (background != null) {
        Rect padding = sThreadLocal.get();
        if (padding == null) {
            padding = new Rect();
            sThreadLocal.set(padding);
        }
        resetResolvedDrawablesInternal();
        background.setLayoutDirection(getLayoutDirection());
        if (background.getPadding(padding)) {
            resetResolvedPaddingInternal();
            switch (background.getLayoutDirection()) {
                case LAYOUT_DIRECTION_RTL:
                    mUserPaddingLeftInitial = padding.right;
                    mUserPaddingRightInitial = padding.left;
                    internalSetPadding(padding.right, padding.top, padding.left, padding.bottom);
                    break;
                case LAYOUT_DIRECTION_LTR:
                default:
                    mUserPaddingLeftInitial = padding.left;
                    mUserPaddingRightInitial = padding.right;
                    internalSetPadding(padding.left, padding.top, padding.right, padding.bottom);
            }
            mLeftPaddingDefined = false;
            mRightPaddingDefined = false;
        }

        // Compare the minimum sizes of the old Drawable and the new.  If there isn't an old or
        // if it has a different minimum size, we should layout again
        if (mBackground == null
                || mBackground.getMinimumHeight() != background.getMinimumHeight()
                || mBackground.getMinimumWidth() != background.getMinimumWidth()) {
            requestLayout = true;
        }

        // Set mBackground before we set this as the callback and start making other
        // background drawable state change calls. In particular, the setVisible call below
        // can result in drawables attempting to start animations or otherwise invalidate,
        // which requires the view set as the callback (us) to recognize the drawable as
        // belonging to it as per verifyDrawable.
        mBackground = background;
        if (background.isStateful()) {
            background.setState(getDrawableState());
        }
        if (isAttachedToWindow()) {
            background.setVisible(getWindowVisibility() == VISIBLE && isShown(), false);
        }

        applyBackgroundTint();

        // Set callback last, since the view may still be initializing.
        background.setCallback(this);

        if ((mPrivateFlags & PFLAG_SKIP_DRAW) != 0) {
            mPrivateFlags &= ~PFLAG_SKIP_DRAW;
            requestLayout = true;
        }
    } else {
        /* Remove the background */
        mBackground = null;
        if ((mViewFlags & WILL_NOT_DRAW) != 0
                && (mDefaultFocusHighlight == null)
                && (mForegroundInfo == null || mForegroundInfo.mDrawable == null)) {
            mPrivateFlags |= PFLAG_SKIP_DRAW;
        }

        /*
         * When the background is set, we try to apply its padding to this
         * View. When the background is removed, we don't touch this View's
         * padding. This is noted in the Javadocs. Hence, we don't need to
         * requestLayout(), the invalidate() below is sufficient.
         */

        // The old background's minimum size could have affected this
```

```
            // View's layout, so let's requestLayout
            requestLayout = true;
        }

        computeOpaqueFlags();

        if (requestLayout) {
            requestLayout();
        }

        mBackgroundSizeChanged = true;
        invalidate(true);
        invalidateOutline();
    }

    /**
     * Gets the background drawable
     *
     * @return The drawable used as the background for this view, if any.
     *
     * @see #setBackground(Drawable)
     *
     * @attr ref android.R.styleable#View_background
     */
    public Drawable getBackground() {
        return mBackground;
    }

    /**
     * Applies a tint to the background drawable. Does not modify the current tint
     * mode, which is {@link PorterDuff.Mode#SRC_IN} by default.
     * <p>
     * Subsequent calls to {@link #setBackground(Drawable)} will automatically
     * mutate the drawable and apply the specified tint and tint mode using
     * {@link Drawable#setTintList(ColorStateList)}.
     *
     * @param tint the tint to apply, may be {@code null} to clear tint
     *
     * @attr ref android.R.styleable#View_backgroundTint
     * @see #getBackgroundTintList()
     * @see Drawable#setTintList(ColorStateList)
     */
    public void setBackgroundTintList(@Nullable ColorStateList tint) {
        if (mBackgroundTint == null) {
            mBackgroundTint = new TintInfo();
        }
        mBackgroundTint.mTintList = tint;
        mBackgroundTint.mHasTintList = true;

        applyBackgroundTint();
    }

    /**
     * Return the tint applied to the background drawable, if specified.
     *
     * @return the tint applied to the background drawable
     * @attr ref android.R.styleable#View_backgroundTint
     * @see #setBackgroundTintList(ColorStateList)
     */
    @Nullable
    public ColorStateList getBackgroundTintList() {
        return mBackgroundTint != null ? mBackgroundTint.mTintList : null;
    }

    /**
     * Specifies the blending mode used to apply the tint specified by
     * {@link #setBackgroundTintList(ColorStateList)}} to the background
     * drawable. The default mode is {@link PorterDuff.Mode#SRC_IN}.
     *
     * @param tintMode the blending mode used to apply the tint, may be
     *                 {@code null} to clear tint
     * @attr ref android.R.styleable#View_backgroundTintMode
     * @see #getBackgroundTintMode()
     * @see Drawable#setTintMode(PorterDuff.Mode)
     */
    public void setBackgroundTintMode(@Nullable PorterDuff.Mode tintMode) {
        if (mBackgroundTint == null) {
            mBackgroundTint = new TintInfo();
        }
        mBackgroundTint.mTintMode = tintMode;
        mBackgroundTint.mHasTintMode = true;
```

```
    applyBackgroundTint();
}

/**
 * Return the blending mode used to apply the tint to the background
 * drawable, if specified.
 *
 * @return the blending mode used to apply the tint to the background
 *         drawable
 * @attr ref android.R.styleable#View_backgroundTintMode
 * @see #setBackgroundTintMode(PorterDuff.Mode)
 */
@Nullable
public PorterDuff.Mode getBackgroundTintMode() {
    return mBackgroundTint != null ? mBackgroundTint.mTintMode : null;
}

private void applyBackgroundTint() {
    if (mBackground != null && mBackgroundTint != null) {
        final TintInfo tintInfo = mBackgroundTint;
        if (tintInfo.mHasTintList || tintInfo.mHasTintMode) {
            mBackground = mBackground.mutate();

            if (tintInfo.mHasTintList) {
                mBackground.setTintList(tintInfo.mTintList);
            }

            if (tintInfo.mHasTintMode) {
                mBackground.setTintMode(tintInfo.mTintMode);
            }

            // The drawable (or one of its children) may not have been
            // stateful before applying the tint, so let's try again.
            if (mBackground.isStateful()) {
                mBackground.setState(getDrawableState());
            }
        }
    }
}

/**
 * Returns the drawable used as the foreground of this View. The
 * foreground drawable, if non-null, is always drawn on top of the view's content.
 *
 * @return a Drawable or null if no foreground was set
 *
 * @see #onDrawForeground(Canvas)
 */
public Drawable getForeground() {
    return mForegroundInfo != null ? mForegroundInfo.mDrawable : null;
}

/**
 * Supply a Drawable that is to be rendered on top of all of the content in the view.
 *
 * @param foreground the Drawable to be drawn on top of the children
 *
 * @attr ref android.R.styleable#View_foreground
 */
public void setForeground(Drawable foreground) {
    if (mForegroundInfo == null) {
        if (foreground == null) {
            // Nothing to do.
            return;
        }
        mForegroundInfo = new ForegroundInfo();
    }

    if (foreground == mForegroundInfo.mDrawable) {
        // Nothing to do
        return;
    }

    if (mForegroundInfo.mDrawable != null) {
        if (isAttachedToWindow()) {
            mForegroundInfo.mDrawable.setVisible(false, false);
        }
        mForegroundInfo.mDrawable.setCallback(null);
        unscheduleDrawable(mForegroundInfo.mDrawable);
    }

    mForegroundInfo.mDrawable = foreground;
```

```
    mForegroundInfo.mBoundsChanged = true;
    if (foreground != null) {
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) != 0) {
            mPrivateFlags &= ~PFLAG_SKIP_DRAW;
        }
        foreground.setLayoutDirection(getLayoutDirection());
        if (foreground.isStateful()) {
            foreground.setState(getDrawableState());
        }
        applyForegroundTint();
        if (isAttachedToWindow()) {
            foreground.setVisible(getWindowVisibility() == VISIBLE && isShown(), false);
        }
        // Set callback last, since the view may still be initializing.
        foreground.setCallback(this);
    } else if ((mViewFlags & WILL_NOT_DRAW) != 0 && mBackground == null
            && (mDefaultFocusHighlight == null)) {
        mPrivateFlags |= PFLAG_SKIP_DRAW;
    }
    requestLayout();
    invalidate();
}

/**
 * Magic bit used to support features of framework-internal window decor implementation details.
 * This used to live exclusively in FrameLayout.
 *
 * @return true if the foreground should draw inside the padding region or false
 *         if it should draw inset by the view's padding
 * @hide internal use only; only used by FrameLayout and internal screen layouts.
 */
public boolean isForegroundInsidePadding() {
    return mForegroundInfo != null ? mForegroundInfo.mInsidePadding : true;
}

/**
 * Describes how the foreground is positioned.
 *
 * @return foreground gravity.
 *
 * @see #setForegroundGravity(int)
 *
 * @attr ref android.R.styleable#View_foregroundGravity
 */
public int getForegroundGravity() {
    return mForegroundInfo != null ? mForegroundInfo.mGravity
            : Gravity.START | Gravity.TOP;
}

/**
 * Describes how the foreground is positioned. Defaults to START and TOP.
 *
 * @param gravity see {@link android.view.Gravity}
 *
 * @see #getForegroundGravity()
 *
 * @attr ref android.R.styleable#View_foregroundGravity
 */
public void setForegroundGravity(int gravity) {
    if (mForegroundInfo == null) {
        mForegroundInfo = new ForegroundInfo();
    }

    if (mForegroundInfo.mGravity != gravity) {
        if ((gravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK) == 0) {
            gravity |= Gravity.START;
        }

        if ((gravity & Gravity.VERTICAL_GRAVITY_MASK) == 0) {
            gravity |= Gravity.TOP;
        }

        mForegroundInfo.mGravity = gravity;
        requestLayout();
    }
}

/**
 * Applies a tint to the foreground drawable. Does not modify the current tint
 * mode, which is {@link PorterDuff.Mode#SRC_IN} by default.
 * <p>
 * Subsequent calls to {@link #setForeground(Drawable)} will automatically
```

```
     * mutate the drawable and apply the specified tint and tint mode using
     * {@link Drawable#setTintList(ColorStateList)}.
     *
     * @param tint the tint to apply, may be {@code null} to clear tint
     *
     * @attr ref android.R.styleable#View_foregroundTint
     * @see #getForegroundTintList()
     * @see Drawable#setTintList(ColorStateList)
     */
    public void setForegroundTintList(@Nullable ColorStateList tint) {
        if (mForegroundInfo == null) {
            mForegroundInfo = new ForegroundInfo();
        }
        if (mForegroundInfo.mTintInfo == null) {
            mForegroundInfo.mTintInfo = new TintInfo();
        }
        mForegroundInfo.mTintInfo.mTintList = tint;
        mForegroundInfo.mTintInfo.mHasTintList = true;

        applyForegroundTint();
    }

    /**
     * Return the tint applied to the foreground drawable, if specified.
     *
     * @return the tint applied to the foreground drawable
     * @attr ref android.R.styleable#View_foregroundTint
     * @see #setForegroundTintList(ColorStateList)
     */
    @Nullable
    public ColorStateList getForegroundTintList() {
        return mForegroundInfo != null && mForegroundInfo.mTintInfo != null
                ? mForegroundInfo.mTintInfo.mTintList : null;
    }

    /**
     * Specifies the blending mode used to apply the tint specified by
     * {@link #setForegroundTintList(ColorStateList)}} to the background
     * drawable. The default mode is {@link PorterDuff.Mode#SRC_IN}.
     *
     * @param tintMode the blending mode used to apply the tint, may be
     *                 {@code null} to clear tint
     * @attr ref android.R.styleable#View_foregroundTintMode
     * @see #getForegroundTintMode()
     * @see Drawable#setTintMode(PorterDuff.Mode)
     */
    public void setForegroundTintMode(@Nullable PorterDuff.Mode tintMode) {
        if (mForegroundInfo == null) {
            mForegroundInfo = new ForegroundInfo();
        }
        if (mForegroundInfo.mTintInfo == null) {
            mForegroundInfo.mTintInfo = new TintInfo();
        }
        mForegroundInfo.mTintInfo.mTintMode = tintMode;
        mForegroundInfo.mTintInfo.mHasTintMode = true;

        applyForegroundTint();
    }

    /**
     * Return the blending mode used to apply the tint to the foreground
     * drawable, if specified.
     *
     * @return the blending mode used to apply the tint to the foreground
     *         drawable
     * @attr ref android.R.styleable#View_foregroundTintMode
     * @see #setForegroundTintMode(PorterDuff.Mode)
     */
    @Nullable
    public PorterDuff.Mode getForegroundTintMode() {
        return mForegroundInfo != null && mForegroundInfo.mTintInfo != null
                ? mForegroundInfo.mTintInfo.mTintMode : null;
    }

    private void applyForegroundTint() {
        if (mForegroundInfo != null && mForegroundInfo.mDrawable != null
                && mForegroundInfo.mTintInfo != null) {
            final TintInfo tintInfo = mForegroundInfo.mTintInfo;
            if (tintInfo.mHasTintList || tintInfo.mHasTintMode) {
                mForegroundInfo.mDrawable = mForegroundInfo.mDrawable.mutate();

                if (tintInfo.mHasTintList) {
```

```
                mForegroundInfo.mDrawable.setTintList(tintInfo.mTintList);
            }

            if (tintInfo.mHasTintMode) {
                mForegroundInfo.mDrawable.setTintMode(tintInfo.mTintMode);
            }

            // The drawable (or one of its children) may not have been
            // stateful before applying the tint, so let's try again.
            if (mForegroundInfo.mDrawable.isStateful()) {
                mForegroundInfo.mDrawable.setState(getDrawableState());
            }
        }
    }
}

/**
 * Get the drawable to be overlayed when a view is autofilled
 *
 * @return The drawable
 *
 * @throws IllegalStateException if the drawable could not be found.
 */
@Nullable private Drawable getAutofilledDrawable() {
    if (mAttachInfo == null) {
        return null;
    }
    // Lazily load the isAutofilled drawable.
    if (mAttachInfo.mAutofilledDrawable == null) {
        Context rootContext = getRootView().getContext();
        TypedArray a = rootContext.getTheme().obtainStyledAttributes(AUTOFILL_HIGHLIGHT_ATTR);
        int attributeResourceId = a.getResourceId(0, 0);
        mAttachInfo.mAutofilledDrawable = rootContext.getDrawable(attributeResourceId);
        a.recycle();
    }

    return mAttachInfo.mAutofilledDrawable;
}

/**
 * Draw {@link View#isAutofilled()} highlight over view if the view is autofilled.
 *
 * @param canvas The canvas to draw on
 */
private void drawAutofilledHighlight(@NonNull Canvas canvas) {
    if (isAutofilled()) {
        Drawable autofilledHighlight = getAutofilledDrawable();

        if (autofilledHighlight != null) {
            autofilledHighlight.setBounds(0, 0, getWidth(), getHeight());
            autofilledHighlight.draw(canvas);
        }
    }
}

/**
 * Draw any foreground content for this view.
 *
 * <p>Foreground content may consist of scroll bars, a {@link #setForeground foreground}
 * drawable or other view-specific decorations. The foreground is drawn on top of the
 * primary view content.</p>
 *
 * @param canvas canvas to draw into
 */
public void onDrawForeground(Canvas canvas) {
    onDrawScrollIndicators(canvas);
    onDrawScrollBars(canvas);

    final Drawable foreground = mForegroundInfo != null ? mForegroundInfo.mDrawable : null;
    if (foreground != null) {
        if (mForegroundInfo.mBoundsChanged) {
            mForegroundInfo.mBoundsChanged = false;
            final Rect selfBounds = mForegroundInfo.mSelfBounds;
            final Rect overlayBounds = mForegroundInfo.mOverlayBounds;

            if (mForegroundInfo.mInsidePadding) {
                selfBounds.set(0, 0, getWidth(), getHeight());
            } else {
                selfBounds.set(getPaddingLeft(), getPaddingTop(),
                        getWidth() - getPaddingRight(), getHeight() - getPaddingBottom());
            }
```

```
            final int ld = getLayoutDirection();
            Gravity.apply(mForegroundInfo.mGravity, foreground.getIntrinsicWidth(),
                    foreground.getIntrinsicHeight(), selfBounds, overlayBounds, ld);
            foreground.setBounds(overlayBounds);
        }

        foreground.draw(canvas);
    }
}

/**
 * Sets the padding. The view may add on the space required to display
 * the scrollbars, depending on the style and visibility of the scrollbars.
 * So the values returned from {@link #getPaddingLeft}, {@link #getPaddingTop},
 * {@link #getPaddingRight} and {@link #getPaddingBottom} may be different
 * from the values set in this call.
 *
 * @attr ref android.R.styleable#View_padding
 * @attr ref android.R.styleable#View_paddingBottom
 * @attr ref android.R.styleable#View_paddingLeft
 * @attr ref android.R.styleable#View_paddingRight
 * @attr ref android.R.styleable#View_paddingTop
 * @param left the left padding in pixels
 * @param top the top padding in pixels
 * @param right the right padding in pixels
 * @param bottom the bottom padding in pixels
 */
public void setPadding(int left, int top, int right, int bottom) {
    resetResolvedPaddingInternal();

    mUserPaddingStart = UNDEFINED_PADDING;
    mUserPaddingEnd = UNDEFINED_PADDING;

    mUserPaddingLeftInitial = left;
    mUserPaddingRightInitial = right;

    mLeftPaddingDefined = true;
    mRightPaddingDefined = true;

    internalSetPadding(left, top, right, bottom);
}

/**
 * @hide
 */
protected void internalSetPadding(int left, int top, int right, int bottom) {
    mUserPaddingLeft = left;
    mUserPaddingRight = right;
    mUserPaddingBottom = bottom;

    final int viewFlags = mViewFlags;
    boolean changed = false;

    // Common case is there are no scroll bars.
    if ((viewFlags & (SCROLLBARS_VERTICAL|SCROLLBARS_HORIZONTAL)) != 0) {
        if ((viewFlags & SCROLLBARS_VERTICAL) != 0) {
            final int offset = (viewFlags & SCROLLBARS_INSET_MASK) == 0
                    ? 0 : getVerticalScrollbarWidth();
            switch (mVerticalScrollbarPosition) {
                case SCROLLBAR_POSITION_DEFAULT:
                    if (isLayoutRtl()) {
                        left += offset;
                    } else {
                        right += offset;
                    }
                    break;
                case SCROLLBAR_POSITION_RIGHT:
                    right += offset;
                    break;
                case SCROLLBAR_POSITION_LEFT:
                    left += offset;
                    break;
            }
        }
        if ((viewFlags & SCROLLBARS_HORIZONTAL) != 0) {
            bottom += (viewFlags & SCROLLBARS_INSET_MASK) == 0
                    ? 0 : getHorizontalScrollbarHeight();
        }
    }

    if (mPaddingLeft != left) {
        changed = true;
```

```
            mPaddingLeft = left;
        }
        if (mPaddingTop != top) {
            changed = true;
            mPaddingTop = top;
        }
        if (mPaddingRight != right) {
            changed = true;
            mPaddingRight = right;
        }
        if (mPaddingBottom != bottom) {
            changed = true;
            mPaddingBottom = bottom;
        }

        if (changed) {
            requestLayout();
            invalidateOutline();
        }
    }

    /**
     * Sets the relative padding. The view may add on the space required to display
     * the scrollbars, depending on the style and visibility of the scrollbars.
     * So the values returned from {@link #getPaddingStart}, {@link #getPaddingTop},
     * {@link #getPaddingEnd} and {@link #getPaddingBottom} may be different
     * from the values set in this call.
     *
     * @attr ref android.R.styleable#View_padding
     * @attr ref android.R.styleable#View_paddingBottom
     * @attr ref android.R.styleable#View_paddingStart
     * @attr ref android.R.styleable#View_paddingEnd
     * @attr ref android.R.styleable#View_paddingTop
     * @param start the start padding in pixels
     * @param top the top padding in pixels
     * @param end the end padding in pixels
     * @param bottom the bottom padding in pixels
     */
    public void setPaddingRelative(int start, int top, int end, int bottom) {
        resetResolvedPaddingInternal();

        mUserPaddingStart = start;
        mUserPaddingEnd = end;
        mLeftPaddingDefined = true;
        mRightPaddingDefined = true;

        switch(getLayoutDirection()) {
            case LAYOUT_DIRECTION_RTL:
                mUserPaddingLeftInitial = end;
                mUserPaddingRightInitial = start;
                internalSetPadding(end, top, start, bottom);
                break;
            case LAYOUT_DIRECTION_LTR:
            default:
                mUserPaddingLeftInitial = start;
                mUserPaddingRightInitial = end;
                internalSetPadding(start, top, end, bottom);
        }
    }

    /**
     * Returns the top padding of this view.
     *
     * @return the top padding in pixels
     */
    public int getPaddingTop() {
        return mPaddingTop;
    }

    /**
     * Returns the bottom padding of this view. If there are inset and enabled
     * scrollbars, this value may include the space required to display the
     * scrollbars as well.
     *
     * @return the bottom padding in pixels
     */
    public int getPaddingBottom() {
        return mPaddingBottom;
    }

    /**
     * Returns the left padding of this view. If there are inset and enabled
```

```
     * scrollbars, this value may include the space required to display the
     * scrollbars as well.
     *
     * @return the left padding in pixels
     */
    public int getPaddingLeft() {
        if (!isPaddingResolved()) {
            resolvePadding();
        }
        return mPaddingLeft;
    }

    /**
     * Returns the start padding of this view depending on its resolved layout direction.
     * If there are inset and enabled scrollbars, this value may include the space
     * required to display the scrollbars as well.
     *
     * @return the start padding in pixels
     */
    public int getPaddingStart() {
        if (!isPaddingResolved()) {
            resolvePadding();
        }
        return (getLayoutDirection() == LAYOUT_DIRECTION_RTL) ?
                mPaddingRight : mPaddingLeft;
    }

    /**
     * Returns the right padding of this view. If there are inset and enabled
     * scrollbars, this value may include the space required to display the
     * scrollbars as well.
     *
     * @return the right padding in pixels
     */
    public int getPaddingRight() {
        if (!isPaddingResolved()) {
            resolvePadding();
        }
        return mPaddingRight;
    }

    /**
     * Returns the end padding of this view depending on its resolved layout direction.
     * If there are inset and enabled scrollbars, this value may include the space
     * required to display the scrollbars as well.
     *
     * @return the end padding in pixels
     */
    public int getPaddingEnd() {
        if (!isPaddingResolved()) {
            resolvePadding();
        }
        return (getLayoutDirection() == LAYOUT_DIRECTION_RTL) ?
                mPaddingLeft : mPaddingRight;
    }

    /**
     * Return if the padding has been set through relative values
     * {@link #setPaddingRelative(int, int, int, int)} or through
     * @attr ref android.R.styleable#View_paddingStart or
     * @attr ref android.R.styleable#View_paddingEnd
     *
     * @return true if the padding is relative or false if it is not.
     */
    public boolean isPaddingRelative() {
        return (mUserPaddingStart != UNDEFINED_PADDING || mUserPaddingEnd != UNDEFINED_PADDING);
    }

    Insets computeOpticalInsets() {
        return (mBackground == null) ? Insets.NONE : mBackground.getOpticalInsets();
    }

    /**
     * @hide
     */
    public void resetPaddingToInitialValues() {
        if (isRtlCompatibilityMode()) {
            mPaddingLeft = mUserPaddingLeftInitial;
            mPaddingRight = mUserPaddingRightInitial;
            return;
        }
        if (isLayoutRtl()) {
```

```
        mPaddingLeft = (mUserPaddingEnd >= 0) ? mUserPaddingEnd : mUserPaddingLeftInitial;
        mPaddingRight = (mUserPaddingStart >= 0) ? mUserPaddingStart : mUserPaddingRightInitial;
    } else {
        mPaddingLeft = (mUserPaddingStart >= 0) ? mUserPaddingStart : mUserPaddingLeftInitial;
        mPaddingRight = (mUserPaddingEnd >= 0) ? mUserPaddingEnd : mUserPaddingRightInitial;
    }
}

/**
 * @hide
 */
public Insets getOpticalInsets() {
    if (mLayoutInsets == null) {
        mLayoutInsets = computeOpticalInsets();
    }
    return mLayoutInsets;
}

/**
 * Set this view's optical insets.
 *
 * <p>This method should be treated similarly to setMeasuredDimension and not as a general
 * property. Views that compute their own optical insets should call it as part of measurement.
 * This method does not request layout. If you are setting optical insets outside of
 * measure/layout itself you will want to call requestLayout() yourself.
 * </p>
 * @hide
 */
public void setOpticalInsets(Insets insets) {
    mLayoutInsets = insets;
}

/**
 * Changes the selection state of this view. A view can be selected or not.
 * Note that selection is not the same as focus. Views are typically
 * selected in the context of an AdapterView like ListView or GridView;
 * the selected view is the view that is highlighted.
 *
 * @param selected true if the view must be selected, false otherwise
 */
public void setSelected(boolean selected) {
    //noinspection DoubleNegation
    if (((mPrivateFlags & PFLAG_SELECTED) != 0) != selected) {
        mPrivateFlags = (mPrivateFlags & ~PFLAG_SELECTED) | (selected ? PFLAG_SELECTED : 0);
        if (!selected) resetPressedState();
        invalidate(true);
        refreshDrawableState();
        dispatchSetSelected(selected);
        if (selected) {
            sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_SELECTED);
        } else {
            notifyViewAccessibilityStateChangedIfNeeded(
                    AccessibilityEvent.CONTENT_CHANGE_TYPE_UNDEFINED);
        }
    }
}

/**
 * Dispatch setSelected to all of this View's children.
 *
 * @see #setSelected(boolean)
 *
 * @param selected The new selected state
 */
protected void dispatchSetSelected(boolean selected) {
}

/**
 * Indicates the selection state of this view.
 *
 * @return true if the view is selected, false otherwise
 */
@ViewDebug.ExportedProperty
public boolean isSelected() {
    return (mPrivateFlags & PFLAG_SELECTED) != 0;
}

/**
 * Changes the activated state of this view. A view can be activated or not.
 * Note that activation is not the same as selection.  Selection is
 * a transient property, representing the view (hierarchy) the user is
 * currently interacting with.  Activation is a longer-term state that the
```

```
     * user can move views in and out of.  For example, in a list view with
     * single or multiple selection enabled, the views in the current selection
     * set are activated.  (Um, yeah, we are deeply sorry about the terminology
     * here.)  The activated state is propagated down to children of the view it
     * is set on.
     *
     * @param activated true if the view must be activated, false otherwise
     */
    public void setActivated(boolean activated) {
        //noinspection DoubleNegation
        if (((mPrivateFlags & PFLAG_ACTIVATED) != 0) != activated) {
            mPrivateFlags = (mPrivateFlags & ~PFLAG_ACTIVATED) | (activated ? PFLAG_ACTIVATED : 0);
            invalidate(true);
            refreshDrawableState();
            dispatchSetActivated(activated);
        }
    }

    /**
     * Dispatch setActivated to all of this View's children.
     *
     * @see #setActivated(boolean)
     *
     * @param activated The new activated state
     */
    protected void dispatchSetActivated(boolean activated) {
    }

    /**
     * Indicates the activation state of this view.
     *
     * @return true if the view is activated, false otherwise
     */
    @ViewDebug.ExportedProperty
    public boolean isActivated() {
        return (mPrivateFlags & PFLAG_ACTIVATED) != 0;
    }

    /**
     * Returns the ViewTreeObserver for this view's hierarchy. The view tree
     * observer can be used to get notifications when global events, like
     * layout, happen.
     *
     * The returned ViewTreeObserver observer is not guaranteed to remain
     * valid for the lifetime of this View. If the caller of this method keeps
     * a long-lived reference to ViewTreeObserver, it should always check for
     * the return value of {@link ViewTreeObserver#isAlive()}.
     *
     * @return The ViewTreeObserver for this view's hierarchy.
     */
    public ViewTreeObserver getViewTreeObserver() {
        if (mAttachInfo != null) {
            return mAttachInfo.mTreeObserver;
        }
        if (mFloatingTreeObserver == null) {
            mFloatingTreeObserver = new ViewTreeObserver(mContext);
        }
        return mFloatingTreeObserver;
    }

    /**
     * <p>Finds the topmost view in the current view hierarchy.</p>
     *
     * @return the topmost view containing this view
     */
    public View getRootView() {
        if (mAttachInfo != null) {
            final View v = mAttachInfo.mRootView;
            if (v != null) {
                return v;
            }
        }

        View parent = this;

        while (parent.mParent != null && parent.mParent instanceof View) {
            parent = (View) parent.mParent;
        }

        return parent;
    }
```

```
/**
 * Transforms a motion event from view-local coordinates to on-screen
 * coordinates.
 *
 * @param ev the view-local motion event
 * @return false if the transformation could not be applied
 * @hide
 */
public boolean toGlobalMotionEvent(MotionEvent ev) {
    final AttachInfo info = mAttachInfo;
    if (info == null) {
        return false;
    }

    final Matrix m = info.mTmpMatrix;
    m.set(Matrix.IDENTITY_MATRIX);
    transformMatrixToGlobal(m);
    ev.transform(m);
    return true;
}

/**
 * Transforms a motion event from on-screen coordinates to view-local
 * coordinates.
 *
 * @param ev the on-screen motion event
 * @return false if the transformation could not be applied
 * @hide
 */
public boolean toLocalMotionEvent(MotionEvent ev) {
    final AttachInfo info = mAttachInfo;
    if (info == null) {
        return false;
    }

    final Matrix m = info.mTmpMatrix;
    m.set(Matrix.IDENTITY_MATRIX);
    transformMatrixToLocal(m);
    ev.transform(m);
    return true;
}

/**
 * Modifies the input matrix such that it maps view-local coordinates to
 * on-screen coordinates.
 *
 * @param m input matrix to modify
 * @hide
 */
public void transformMatrixToGlobal(Matrix m) {
    final ViewParent parent = mParent;
    if (parent instanceof View) {
        final View vp = (View) parent;
        vp.transformMatrixToGlobal(m);
        m.preTranslate(-vp.mScrollX, -vp.mScrollY);
    } else if (parent instanceof ViewRootImpl) {
        final ViewRootImpl vr = (ViewRootImpl) parent;
        vr.transformMatrixToGlobal(m);
        m.preTranslate(0, -vr.mCurScrollY);
    }

    m.preTranslate(mLeft, mTop);

    if (!hasIdentityMatrix()) {
        m.preConcat(getMatrix());
    }
}

/**
 * Modifies the input matrix such that it maps on-screen coordinates to
 * view-local coordinates.
 *
 * @param m input matrix to modify
 * @hide
 */
public void transformMatrixToLocal(Matrix m) {
    final ViewParent parent = mParent;
    if (parent instanceof View) {
        final View vp = (View) parent;
        vp.transformMatrixToLocal(m);
        m.postTranslate(vp.mScrollX, vp.mScrollY);
    } else if (parent instanceof ViewRootImpl) {
```

```
            final ViewRootImpl vr = (ViewRootImpl) parent;
            vr.transformMatrixToLocal(m);
            m.postTranslate(0, vr.mCurScrollY);
        }

        m.postTranslate(-mLeft, -mTop);

        if (!hasIdentityMatrix()) {
            m.postConcat(getInverseMatrix());
        }
    }

    /**
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "layout", indexMapping = {
            @ViewDebug.IntToString(from = 0, to = "x"),
            @ViewDebug.IntToString(from = 1, to = "y")
    })
    public int[] getLocationOnScreen() {
        int[] location = new int[2];
        getLocationOnScreen(location);
        return location;
    }

    /**
     * <p>Computes the coordinates of this view on the screen. The argument
     * must be an array of two integers. After the method returns, the array
     * contains the x and y location in that order.</p>
     *
     * @param outLocation an array of two integers in which to hold the coordinates
     */
    public void getLocationOnScreen(@Size(2) int[] outLocation) {
        getLocationInWindow(outLocation);

        final AttachInfo info = mAttachInfo;
        if (info != null) {
            outLocation[0] += info.mWindowLeft;
            outLocation[1] += info.mWindowTop;
        }
    }

    /**
     * <p>Computes the coordinates of this view in its window. The argument
     * must be an array of two integers. After the method returns, the array
     * contains the x and y location in that order.</p>
     *
     * @param outLocation an array of two integers in which to hold the coordinates
     */
    public void getLocationInWindow(@Size(2) int[] outLocation) {
        if (outLocation == null || outLocation.length < 2) {
            throw new IllegalArgumentException("outLocation must be an array of two integers");
        }

        outLocation[0] = 0;
        outLocation[1] = 0;

        transformFromViewToWindowSpace(outLocation);
    }

    /** @hide */
    public void transformFromViewToWindowSpace(@Size(2) int[] inOutLocation) {
        if (inOutLocation == null || inOutLocation.length < 2) {
            throw new IllegalArgumentException("inOutLocation must be an array of two integers");
        }

        if (mAttachInfo == null) {
            // When the view is not attached to a window, this method does not make sense
            inOutLocation[0] = inOutLocation[1] = 0;
            return;
        }

        float position[] = mAttachInfo.mTmpTransformLocation;
        position[0] = inOutLocation[0];
        position[1] = inOutLocation[1];

        if (!hasIdentityMatrix()) {
            getMatrix().mapPoints(position);
        }

        position[0] += mLeft;
        position[1] += mTop;
```

```
    ViewParent viewParent = mParent;
    while (viewParent instanceof View) {
        final View view = (View) viewParent;

        position[0] -= view.mScrollX;
        position[1] -= view.mScrollY;

        if (!view.hasIdentityMatrix()) {
            view.getMatrix().mapPoints(position);
        }

        position[0] += view.mLeft;
        position[1] += view.mTop;

        viewParent = view.mParent;
     }

    if (viewParent instanceof ViewRootImpl) {
        // *cough*
        final ViewRootImpl vr = (ViewRootImpl) viewParent;
        position[1] -= vr.mCurScrollY;
    }

    inOutLocation[0] = Math.round(position[0]);
    inOutLocation[1] = Math.round(position[1]);
}

/**
 * @param id the id of the view to be found
 * @return the view of the specified id, null if cannot be found
 * @hide
 */
protected <T extends View> T findViewTraversal(@IdRes int id) {
    if (id == mID) {
        return (T) this;
    }
    return null;
}

/**
 * @param tag the tag of the view to be found
 * @return the view of specified tag, null if cannot be found
 * @hide
 */
protected <T extends View> T findViewWithTagTraversal(Object tag) {
    if (tag != null && tag.equals(mTag)) {
        return (T) this;
    }
    return null;
}

/**
 * @param predicate The predicate to evaluate.
 * @param childToSkip If not null, ignores this child during the recursive traversal.
 * @return The first view that matches the predicate or null.
 * @hide
 */
protected <T extends View> T findViewByPredicateTraversal(Predicate<View> predicate,
        View childToSkip) {
    if (predicate.test(this)) {
        return (T) this;
    }
    return null;
}

/**
 * Finds the first descendant view with the given ID, the view itself if
 * the ID matches {@link #getId()}, or {@code null} if the ID is invalid
 * (< 0) or there is no matching view in the hierarchy.
 * <p>
 * <strong>Note:</strong> In most cases -- depending on compiler support --
 * the resulting view is automatically cast to the target class type. If
 * the target class type is unconstrained, an explicit cast may be
 * necessary.
 *
 * @param id the ID to search for
 * @return a view with given ID if found, or {@code null} otherwise
 * @see View#findViewById(int)
 */
@Nullable
public final <T extends View> T findViewById(@IdRes int id) {
```

```
    if (id == NO_ID) {
        return null;
    }
    return findViewTraversal(id);
}

/**
 * Finds a view by its unuque and stable accessibility id.
 *
 * @param accessibilityId The searched accessibility id.
 * @return The found view.
 */
final <T extends View> T findViewByAccessibilityId(int accessibilityId) {
    if (accessibilityId < 0) {
        return null;
    }
    T view = findViewByAccessibilityIdTraversal(accessibilityId);
    if (view != null) {
        return view.includeForAccessibility() ? view : null;
    }
    return null;
}

/**
 * Performs the traversal to find a view by its unique and stable accessibility id.
 *
 * <strong>Note:</strong>This method does not stop at the root namespace
 * boundary since the user can touch the screen at an arbitrary location
 * potentially crossing the root namespace boundary which will send an
 * accessibility event to accessibility services and they should be able
 * to obtain the event source. Also accessibility ids are guaranteed to be
 * unique in the window.
 *
 * @param accessibilityId The accessibility id.
 * @return The found view.
 * @hide
 */
public <T extends View> T findViewByAccessibilityIdTraversal(int accessibilityId) {
    if (getAccessibilityViewId() == accessibilityId) {
        return (T) this;
    }
    return null;
}

/**
 * Performs the traversal to find a view by its autofill id.
 *
 * <strong>Note:</strong>This method does not stop at the root namespace
 * boundary.
 *
 * @param autofillId The autofill id.
 * @return The found view.
 * @hide
 */
public <T extends View> T findViewByAutofillIdTraversal(int autofillId) {
    if (getAutofillViewId() == autofillId) {
        return (T) this;
    }
    return null;
}

/**
 * Look for a child view with the given tag.  If this view has the given
 * tag, return this view.
 *
 * @param tag The tag to search for, using "tag.equals(getTag())".
 * @return The View that has the given tag in the hierarchy or null
 */
public final <T extends View> T findViewWithTag(Object tag) {
    if (tag == null) {
        return null;
    }
    return findViewWithTagTraversal(tag);
}

/**
 * Look for a child view that matches the specified predicate.
 * If this view matches the predicate, return this view.
 *
 * @param predicate The predicate to evaluate.
 * @return The first view that matches the predicate or null.
 * @hide
```

```
 */
public final <T extends View> T findViewByPredicate(Predicate<View> predicate) {
    return findViewByPredicateTraversal(predicate, null);
}

/**
 * Look for a child view that matches the specified predicate,
 * starting with the specified view and its descendents and then
 * recusively searching the ancestors and siblings of that view
 * until this view is reached.
 *
 * This method is useful in cases where the predicate does not match
 * a single unique view (perhaps multiple views use the same id)
 * and we are trying to find the view that is "closest" in scope to the
 * starting view.
 *
 * @param start The view to start from.
 * @param predicate The predicate to evaluate.
 * @return The first view that matches the predicate or null.
 * @hide
 */
public final <T extends View> T findViewByPredicateInsideOut(
        View start, Predicate<View> predicate) {
    View childToSkip = null;
    for (;;) {
        T view = start.findViewByPredicateTraversal(predicate, childToSkip);
        if (view != null || start == this) {
            return view;
        }

        ViewParent parent = start.getParent();
        if (parent == null || !(parent instanceof View)) {
            return null;
        }

        childToSkip = start;
        start = (View) parent;
    }
}

/**
 * Sets the identifier for this view. The identifier does not have to be
 * unique in this view's hierarchy. The identifier should be a positive
 * number.
 *
 * @see #NO_ID
 * @see #getId()
 * @see #findViewById(int)
 *
 * @param id a number used to identify the view
 *
 * @attr ref android.R.styleable#View_id
 */
public void setId(@IdRes int id) {
    mID = id;
    if (mID == View.NO_ID && mLabelForId != View.NO_ID) {
        mID = generateViewId();
    }
}

/**
 * {@hide}
 *
 * @param isRoot true if the view belongs to the root namespace, false
 *        otherwise
 */
public void setIsRootNamespace(boolean isRoot) {
    if (isRoot) {
        mPrivateFlags |= PFLAG_IS_ROOT_NAMESPACE;
    } else {
        mPrivateFlags &= ~PFLAG_IS_ROOT_NAMESPACE;
    }
}

/**
 * {@hide}
 *
 * @return true if the view belongs to the root namespace, false otherwise
 */
public boolean isRootNamespace() {
    return (mPrivateFlags&PFLAG_IS_ROOT_NAMESPACE) != 0;
}
```

```
/**
 * Returns this view's identifier.
 *
 * @return a positive integer used to identify the view or {@link #NO_ID}
 *         if the view has no ID
 *
 * @see #setId(int)
 * @see #findViewById(int)
 * @attr ref android.R.styleable#View_id
 */
@IdRes
@ViewDebug.CapturedViewProperty
public int getId() {
    return mID;
}

/**
 * Returns this view's tag.
 *
 * @return the Object stored in this view as a tag, or {@code null} if not
 *         set
 *
 * @see #setTag(Object)
 * @see #getTag(int)
 */
@ViewDebug.ExportedProperty
public Object getTag() {
    return mTag;
}

/**
 * Sets the tag associated with this view. A tag can be used to mark
 * a view in its hierarchy and does not have to be unique within the
 * hierarchy. Tags can also be used to store data within a view without
 * resorting to another data structure.
 *
 * @param tag an Object to tag the view with
 *
 * @see #getTag()
 * @see #setTag(int, Object)
 */
public void setTag(final Object tag) {
    mTag = tag;
}

/**
 * Returns the tag associated with this view and the specified key.
 *
 * @param key The key identifying the tag
 *
 * @return the Object stored in this view as a tag, or {@code null} if not
 *         set
 *
 * @see #setTag(int, Object)
 * @see #getTag()
 */
public Object getTag(int key) {
    if (mKeyedTags != null) return mKeyedTags.get(key);
    return null;
}

/**
 * Sets a tag associated with this view and a key. A tag can be used
 * to mark a view in its hierarchy and does not have to be unique within
 * the hierarchy. Tags can also be used to store data within a view
 * without resorting to another data structure.
 *
 * The specified key should be an id declared in the resources of the
 * application to ensure it is unique (see the <a
 * href="{@docRoot}guide/topics/resources/more-resources.html#Id">ID resource type</a>).
 * Keys identified as belonging to
 * the Android framework or not associated with any package will cause
 * an {@link IllegalArgumentException} to be thrown.
 *
 * @param key The key identifying the tag
 * @param tag An Object to tag the view with
 *
 * @throws IllegalArgumentException If they specified key is not valid
 *
 * @see #setTag(Object)
 * @see #getTag(int)
```

```
     */
    public void setTag(int key, final Object tag) {
        // If the package id is 0x00 or 0x01, it's either an undefined package
        // or a framework id
        if ((key >>> 24) < 2) {
            throw new IllegalArgumentException("The key must be an application-specific "
                    + "resource id.");
        }

        setKeyedTag(key, tag);
    }

    /**
     * Variation of {@link #setTag(int, Object)} that enforces the key to be a
     * framework id.
     *
     * @hide
     */
    public void setTagInternal(int key, Object tag) {
        if ((key >>> 24) != 0x1) {
            throw new IllegalArgumentException("The key must be a framework-specific "
                    + "resource id.");
        }

        setKeyedTag(key, tag);
    }

    private void setKeyedTag(int key, Object tag) {
        if (mKeyedTags == null) {
            mKeyedTags = new SparseArray<Object>(2);
        }

        mKeyedTags.put(key, tag);
    }

    /**
     * Prints information about this view in the log output, with the tag
     * {@link #VIEW_LOG_TAG}.
     *
     * @hide
     */
    public void debug() {
        debug(0);
    }

    /**
     * Prints information about this view in the log output, with the tag
     * {@link #VIEW_LOG_TAG}. Each line in the output is preceded with an
     * indentation defined by the <code>depth</code>.
     *
     * @param depth the indentation level
     *
     * @hide
     */
    protected void debug(int depth) {
        String output = debugIndent(depth - 1);

        output += "+ " + this;
        int id = getId();
        if (id != -1) {
            output += " (id=" + id + ")";
        }
        Object tag = getTag();
        if (tag != null) {
            output += " (tag=" + tag + ")";
        }
        Log.d(VIEW_LOG_TAG, output);

        if ((mPrivateFlags & PFLAG_FOCUSED) != 0) {
            output = debugIndent(depth) + " FOCUSED";
            Log.d(VIEW_LOG_TAG, output);
        }

        output = debugIndent(depth);
        output += "frame={" + mLeft + ", " + mTop + ", " + mRight
                + ", " + mBottom + "} scroll={" + mScrollX + ", " + mScrollY
                + "} ";
        Log.d(VIEW_LOG_TAG, output);

        if (mPaddingLeft != 0 || mPaddingTop != 0 || mPaddingRight != 0
                || mPaddingBottom != 0) {
            output = debugIndent(depth);
```

```
        output += "padding={" + mPaddingLeft + ", " + mPaddingTop
                + ", " + mPaddingRight + ", " + mPaddingBottom + "}";
        Log.d(VIEW_LOG_TAG, output);
    }

    output = debugIndent(depth);
    output += "mMeasureWidth=" + mMeasuredWidth +
            " mMeasureHeight=" + mMeasuredHeight;
    Log.d(VIEW_LOG_TAG, output);

    output = debugIndent(depth);
    if (mLayoutParams == null) {
        output += "BAD! no layout params";
    } else {
        output = mLayoutParams.debug(output);
    }
    Log.d(VIEW_LOG_TAG, output);

    output = debugIndent(depth);
    output += "flags={";
    output += View.printFlags(mViewFlags);
    output += "}";
    Log.d(VIEW_LOG_TAG, output);

    output = debugIndent(depth);
    output += "privateFlags={";
    output += View.printPrivateFlags(mPrivateFlags);
    output += "}";
    Log.d(VIEW_LOG_TAG, output);
}

/**
 * Creates a string of whitespaces used for indentation.
 *
 * @param depth the indentation level
 * @return a String containing (depth * 2 + 3) * 2 white spaces
 *
 * @hide
 */
protected static String debugIndent(int depth) {
    StringBuilder spaces = new StringBuilder((depth * 2 + 3) * 2);
    for (int i = 0; i < (depth * 2) + 3; i++) {
        spaces.append(' ').append(' ');
    }
    return spaces.toString();
}

/**
 * <p>Return the offset of the widget's text baseline from the widget's top
 * boundary. If this widget does not support baseline alignment, this
 * method returns -1. </p>
 *
 * @return the offset of the baseline within the widget's bounds or -1
 *         if baseline alignment is not supported
 */
@ViewDebug.ExportedProperty(category = "layout")
public int getBaseline() {
    return -1;
}

/**
 * Returns whether the view hierarchy is currently undergoing a layout pass. This
 * information is useful to avoid situations such as calling {@link #requestLayout()} during
 * a layout pass.
 *
 * @return whether the view hierarchy is currently undergoing a layout pass
 */
public boolean isInLayout() {
    ViewRootImpl viewRoot = getViewRootImpl();
    return (viewRoot != null && viewRoot.isInLayout());
}

/**
 * Call this when something has changed which has invalidated the
 * layout of this view. This will schedule a layout pass of the view
 * tree. This should not be called while the view hierarchy is currently in a layout
 * pass ({@link #isInLayout()}. If layout is happening, the request may be honored at the
 * end of the current layout pass (and then layout will run again) or after the current
 * frame is drawn and the next layout occurs.
 *
 * <p>Subclasses which override this method should call the superclass method to
 * handle possible request-during-layout errors correctly.</p>
```

```
     */
    @CallSuper
    public void requestLayout() {
        if (mMeasureCache != null) mMeasureCache.clear();

        if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == null) {
            // Only trigger request-during-layout logic if this is the view requesting it,
            // not the views in its parent hierarchy
            ViewRootImpl viewRoot = getViewRootImpl();
            if (viewRoot != null && viewRoot.isInLayout()) {
                if (!viewRoot.requestLayoutDuringLayout(this)) {
                    return;
                }
            }
            mAttachInfo.mViewRequestingLayout = this;
        }

        mPrivateFlags |= PFLAG_FORCE_LAYOUT;
        mPrivateFlags |= PFLAG_INVALIDATED;

        if (mParent != null && !mParent.isLayoutRequested()) {
            mParent.requestLayout();
        }
        if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == this) {
            mAttachInfo.mViewRequestingLayout = null;
        }
    }

    /**
     * Forces this view to be laid out during the next layout pass.
     * This method does not call requestLayout() or forceLayout()
     * on the parent.
     */
    public void forceLayout() {
        if (mMeasureCache != null) mMeasureCache.clear();

        mPrivateFlags |= PFLAG_FORCE_LAYOUT;
        mPrivateFlags |= PFLAG_INVALIDATED;
    }

    /**
     * <p>
     * This is called to find out how big a view should be. The parent
     * supplies constraint information in the width and height parameters.
     * </p>
     *
     * <p>
     * The actual measurement work of a view is performed in
     * {@link #onMeasure(int, int)}, called by this method. Therefore, only
     * {@link #onMeasure(int, int)} can and must be overridden by subclasses.
     * </p>
     *
     *
     * @param widthMeasureSpec Horizontal space requirements as imposed by the
     *        parent
     * @param heightMeasureSpec Vertical space requirements as imposed by the
     *        parent
     *
     * @see #onMeasure(int, int)
     */
    public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
        boolean optical = isLayoutModeOptical(this);
        if (optical != isLayoutModeOptical(mParent)) {
            Insets insets = getOpticalInsets();
            int oWidth  = insets.left + insets.right;
            int oHeight = insets.top  + insets.bottom;
            widthMeasureSpec  = MeasureSpec.adjust(widthMeasureSpec,  optical ? -oWidth  : oWidth);
            heightMeasureSpec = MeasureSpec.adjust(heightMeasureSpec, optical ? -oHeight : oHeight);
        }

        // Suppress sign extension for the low bytes
        long key = (long) widthMeasureSpec << 32 | (long) heightMeasureSpec & 0xffffffffL;
        if (mMeasureCache == null) mMeasureCache = new LongSparseLongArray(2);

        final boolean forceLayout = (mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT;

        // Optimize layout by avoiding an extra EXACTLY pass when the view is
        // already measured as the correct size. In API 23 and below, this
        // extra pass is required to make LinearLayout re-distribute weight.
        final boolean specChanged = widthMeasureSpec != mOldWidthMeasureSpec
                || heightMeasureSpec != mOldHeightMeasureSpec;
        final boolean isSpecExactly = MeasureSpec.getMode(widthMeasureSpec) == MeasureSpec.EXACTLY
```

```
                && MeasureSpec.getMode(heightMeasureSpec) == MeasureSpec.EXACTLY;
        final boolean matchesSpecSize = getMeasuredWidth() == MeasureSpec.getSize(widthMeasureSpec)
                && getMeasuredHeight() == MeasureSpec.getSize(heightMeasureSpec);
        final boolean needsLayout = specChanged
                && (sAlwaysRemeasureExactly || !isSpecExactly || !matchesSpecSize);

        if (forceLayout || needsLayout) {
            // first clears the measured dimension flag
            mPrivateFlags &= ~PFLAG_MEASURED_DIMENSION_SET;

            resolveRtlPropertiesIfNeeded();

            int cacheIndex = forceLayout ? -1 : mMeasureCache.indexOfKey(key);
            if (cacheIndex < 0 || sIgnoreMeasureCache) {
                // measure ourselves, this should set the measured dimension flag back
                onMeasure(widthMeasureSpec, heightMeasureSpec);
                mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
            } else {
                long value = mMeasureCache.valueAt(cacheIndex);
                // Casting a long to int drops the high 32 bits, no mask needed
                setMeasuredDimensionRaw((int) (value >> 32), (int) value);
                mPrivateFlags3 |= PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
            }

            // flag not set, setMeasuredDimension() was not invoked, we raise
            // an exception to warn the developer
            if ((mPrivateFlags & PFLAG_MEASURED_DIMENSION_SET) != PFLAG_MEASURED_DIMENSION_SET) {
                throw new IllegalStateException("View with id " + getId() + ": "
                        + getClass().getName() + "#onMeasure() did not set the"
                        + " measured dimension by calling"
                        + " setMeasuredDimension()");
            }

            mPrivateFlags |= PFLAG_LAYOUT_REQUIRED;
        }

        mOldWidthMeasureSpec = widthMeasureSpec;
        mOldHeightMeasureSpec = heightMeasureSpec;

        mMeasureCache.put(key, ((long) mMeasuredWidth) << 32 |
                (long) mMeasuredHeight & 0xffffffffL); // suppress sign extension
    }

    /**
     * <p>
     * Measure the view and its content to determine the measured width and the
     * measured height. This method is invoked by {@link #measure(int, int)} and
     * should be overridden by subclasses to provide accurate and efficient
     * measurement of their contents.
     * </p>
     *
     * <p>
     * <strong>CONTRACT:</strong> When overriding this method, you
     * <em>must</em> call {@link #setMeasuredDimension(int, int)} to store the
     * measured width and height of this view. Failure to do so will trigger an
     * <code>IllegalStateException</code>, thrown by
     * {@link #measure(int, int)}. Calling the superclass'
     * {@link #onMeasure(int, int)} is a valid use.
     * </p>
     *
     * <p>
     * The base class implementation of measure defaults to the background size,
     * unless a larger size is allowed by the MeasureSpec. Subclasses should
     * override {@link #onMeasure(int, int)} to provide better measurements of
     * their content.
     * </p>
     *
     * <p>
     * If this method is overridden, it is the subclass's responsibility to make
     * sure the measured height and width are at least the view's minimum height
     * and width ({@link #getSuggestedMinimumHeight()} and
     * {@link #getSuggestedMinimumWidth()}).
     * </p>
     *
     * @param widthMeasureSpec horizontal space requirements as imposed by the parent.
     *                         The requirements are encoded with
     *                         {@link android.view.View.MeasureSpec}.
     * @param heightMeasureSpec vertical space requirements as imposed by the parent.
     *                         The requirements are encoded with
     *                         {@link android.view.View.MeasureSpec}.
     *
     * @see #getMeasuredWidth()
```

```
     * @see #getMeasuredHeight()
     * @see #setMeasuredDimension(int, int)
     * @see #getSuggestedMinimumHeight()
     * @see #getSuggestedMinimumWidth()
     * @see android.view.View.MeasureSpec#getMode(int)
     * @see android.view.View.MeasureSpec#getSize(int)
     */
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
                getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
    }

    /**
     * <p>This method must be called by {@link #onMeasure(int, int)} to store the
     * measured width and measured height. Failing to do so will trigger an
     * exception at measurement time.</p>
     *
     * @param measuredWidth The measured width of this view.  May be a complex
     * bit mask as defined by {@link #MEASURED_SIZE_MASK} and
     * {@link #MEASURED_STATE_TOO_SMALL}.
     * @param measuredHeight The measured height of this view.  May be a complex
     * bit mask as defined by {@link #MEASURED_SIZE_MASK} and
     * {@link #MEASURED_STATE_TOO_SMALL}.
     */
    protected final void setMeasuredDimension(int measuredWidth, int measuredHeight) {
        boolean optical = isLayoutModeOptical(this);
        if (optical != isLayoutModeOptical(mParent)) {
            Insets insets = getOpticalInsets();
            int opticalWidth  = insets.left + insets.right;
            int opticalHeight = insets.top  + insets.bottom;

            measuredWidth  += optical ? opticalWidth  : -opticalWidth;
            measuredHeight += optical ? opticalHeight : -opticalHeight;
        }
        setMeasuredDimensionRaw(measuredWidth, measuredHeight);
    }

    /**
     * Sets the measured dimension without extra processing for things like optical bounds.
     * Useful for reapplying consistent values that have already been cooked with adjustments
     * for optical bounds, etc. such as those from the measurement cache.
     *
     * @param measuredWidth The measured width of this view.  May be a complex
     * bit mask as defined by {@link #MEASURED_SIZE_MASK} and
     * {@link #MEASURED_STATE_TOO_SMALL}.
     * @param measuredHeight The measured height of this view.  May be a complex
     * bit mask as defined by {@link #MEASURED_SIZE_MASK} and
     * {@link #MEASURED_STATE_TOO_SMALL}.
     */
    private void setMeasuredDimensionRaw(int measuredWidth, int measuredHeight) {
        mMeasuredWidth = measuredWidth;
        mMeasuredHeight = measuredHeight;

        mPrivateFlags |= PFLAG_MEASURED_DIMENSION_SET;
    }

    /**
     * Merge two states as returned by {@link #getMeasuredState()}.
     * @param curState The current state as returned from a view or the result
     * of combining multiple views.
     * @param newState The new view state to combine.
     * @return Returns a new integer reflecting the combination of the two
     * states.
     */
    public static int combineMeasuredStates(int curState, int newState) {
        return curState | newState;
    }

    /**
     * Version of {@link #resolveSizeAndState(int, int, int)}
     * returning only the {@link #MEASURED_SIZE_MASK} bits of the result.
     */
    public static int resolveSize(int size, int measureSpec) {
        return resolveSizeAndState(size, measureSpec, 0) & MEASURED_SIZE_MASK;
    }

    /**
     * Utility to reconcile a desired size and state, with constraints imposed
     * by a MeasureSpec. Will take the desired size, unless a different size
     * is imposed by the constraints. The returned value is a compound integer,
     * with the resolved size in the {@link #MEASURED_SIZE_MASK} bits and
     * optionally the bit {@link #MEASURED_STATE_TOO_SMALL} set if the
```

```
     * resulting size is smaller than the size the view wants to be.
     *
     * @param size How big the view wants to be.
     * @param measureSpec Constraints imposed by the parent.
     * @param childMeasuredState Size information bit mask for the view's
     *                           children.
     * @return Size information bit mask as defined by
     *         {@link #MEASURED_SIZE_MASK} and
     *         {@link #MEASURED_STATE_TOO_SMALL}.
     */
    public static int resolveSizeAndState(int size, int measureSpec, int childMeasuredState) {
        final int specMode = MeasureSpec.getMode(measureSpec);
        final int specSize = MeasureSpec.getSize(measureSpec);
        final int result;
        switch (specMode) {
            case MeasureSpec.AT_MOST:
                if (specSize < size) {
                    result = specSize | MEASURED_STATE_TOO_SMALL;
                } else {
                    result = size;
                }
                break;
            case MeasureSpec.EXACTLY:
                result = specSize;
                break;
            case MeasureSpec.UNSPECIFIED:
            default:
                result = size;
        }
        return result | (childMeasuredState & MEASURED_STATE_MASK);
    }

    /**
     * Utility to return a default size. Uses the supplied size if the
     * MeasureSpec imposed no constraints. Will get larger if allowed
     * by the MeasureSpec.
     *
     * @param size Default size for this view
     * @param measureSpec Constraints imposed by the parent
     * @return The size this view should be.
     */
    public static int getDefaultSize(int size, int measureSpec) {
        int result = size;
        int specMode = MeasureSpec.getMode(measureSpec);
        int specSize = MeasureSpec.getSize(measureSpec);

        switch (specMode) {
        case MeasureSpec.UNSPECIFIED:
            result = size;
            break;
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
            result = specSize;
            break;
        }
        return result;
    }

    /**
     * Returns the suggested minimum height that the view should use. This
     * returns the maximum of the view's minimum height
     * and the background's minimum height
     * ({@link android.graphics.drawable.Drawable#getMinimumHeight()}).
     * <p>
     * When being used in {@link #onMeasure(int, int)}, the caller should still
     * ensure the returned height is within the requirements of the parent.
     *
     * @return The suggested minimum height of the view.
     */
    protected int getSuggestedMinimumHeight() {
        return (mBackground == null) ? mMinHeight : max(mMinHeight, mBackground.getMinimumHeight());

    }

    /**
     * Returns the suggested minimum width that the view should use. This
     * returns the maximum of the view's minimum width
     * and the background's minimum width
     *  ({@link android.graphics.drawable.Drawable#getMinimumWidth()}).
     * <p>
     * When being used in {@link #onMeasure(int, int)}, the caller should still
     * ensure the returned width is within the requirements of the parent.
```

```
     *
     * @return The suggested minimum width of the view.
     */
    protected int getSuggestedMinimumWidth() {
        return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());
    }

    /**
     * Returns the minimum height of the view.
     *
     * @return the minimum height the view will try to be, in pixels
     *
     * @see #setMinimumHeight(int)
     *
     * @attr ref android.R.styleable#View_minHeight
     */
    public int getMinimumHeight() {
        return mMinHeight;
    }

    /**
     * Sets the minimum height of the view. It is not guaranteed the view will
     * be able to achieve this minimum height (for example, if its parent layout
     * constrains it with less available height).
     *
     * @param minHeight The minimum height the view will try to be, in pixels
     *
     * @see #getMinimumHeight()
     *
     * @attr ref android.R.styleable#View_minHeight
     */
    @RemotableViewMethod
    public void setMinimumHeight(int minHeight) {
        mMinHeight = minHeight;
        requestLayout();
    }

    /**
     * Returns the minimum width of the view.
     *
     * @return the minimum width the view will try to be, in pixels
     *
     * @see #setMinimumWidth(int)
     *
     * @attr ref android.R.styleable#View_minWidth
     */
    public int getMinimumWidth() {
        return mMinWidth;
    }

    /**
     * Sets the minimum width of the view. It is not guaranteed the view will
     * be able to achieve this minimum width (for example, if its parent layout
     * constrains it with less available width).
     *
     * @param minWidth The minimum width the view will try to be, in pixels
     *
     * @see #getMinimumWidth()
     *
     * @attr ref android.R.styleable#View_minWidth
     */
    public void setMinimumWidth(int minWidth) {
        mMinWidth = minWidth;
        requestLayout();

    }

    /**
     * Get the animation currently associated with this view.
     *
     * @return The animation that is currently playing or
     *         scheduled to play for this view.
     */
    public Animation getAnimation() {
        return mCurrentAnimation;
    }

    /**
     * Start the specified animation now.
     *
     * @param animation the animation to start now
     */
```

```
public void startAnimation(Animation animation) {
    animation.setStartTime(Animation.START_ON_FIRST_FRAME);
    setAnimation(animation);
    invalidateParentCaches();
    invalidate(true);
}

/**
 * Cancels any animations for this view.
 */
public void clearAnimation() {
    if (mCurrentAnimation != null) {
        mCurrentAnimation.detach();
    }
    mCurrentAnimation = null;
    invalidateParentIfNeeded();
}

/**
 * Sets the next animation to play for this view.
 * If you want the animation to play immediately, use
 * {@link #startAnimation(android.view.animation.Animation)} instead.
 * This method provides allows fine-grained
 * control over the start time and invalidation, but you
 * must make sure that 1) the animation has a start time set, and
 * 2) the view's parent (which controls animations on its children)
 * will be invalidated when the animation is supposed to
 * start.
 *
 * @param animation The next animation, or null.
 */
public void setAnimation(Animation animation) {
    mCurrentAnimation = animation;

    if (animation != null) {
        // If the screen is off assume the animation start time is now instead of
        // the next frame we draw. Keeping the START_ON_FIRST_FRAME start time
        // would cause the animation to start when the screen turns back on
        if (mAttachInfo != null && mAttachInfo.mDisplayState == Display.STATE_OFF
                && animation.getStartTime() == Animation.START_ON_FIRST_FRAME) {
            animation.setStartTime(AnimationUtils.currentAnimationTimeMillis());
        }
        animation.reset();
    }
}

/**
 * Invoked by a parent ViewGroup to notify the start of the animation
 * currently associated with this view. If you override this method,
 * always call super.onAnimationStart();
 *
 * @see #setAnimation(android.view.animation.Animation)
 * @see #getAnimation()
 */
@CallSuper
protected void onAnimationStart() {
    mPrivateFlags |= PFLAG_ANIMATION_STARTED;
}

/**
 * Invoked by a parent ViewGroup to notify the end of the animation
 * currently associated with this view. If you override this method,
 * always call super.onAnimationEnd();
 *
 * @see #setAnimation(android.view.animation.Animation)
 * @see #getAnimation()
 */
@CallSuper
protected void onAnimationEnd() {
    mPrivateFlags &= ~PFLAG_ANIMATION_STARTED;
}

/**
 * Invoked if there is a Transform that involves alpha. Subclass that can
 * draw themselves with the specified alpha should return true, and then
 * respect that alpha when their onDraw() is called. If this returns false
 * then the view may be redirected to draw into an offscreen buffer to
 * fulfill the request, which will look fine, but may be slower than if the
 * subclass handles it internally. The default implementation returns false.
 *
 * @param alpha The alpha (0..255) to apply to the view's drawing
 * @return true if the view can draw with the specified alpha.
```

```
     */
    protected boolean onSetAlpha(int alpha) {
        return false;
    }

    /**
     * This is used by the RootView to perform an optimization when
     * the view hierarchy contains one or several SurfaceView.
     * SurfaceView is always considered transparent, but its children are not,
     * therefore all View objects remove themselves from the global transparent
     * region (passed as a parameter to this function).
     *
     * @param region The transparent region for this ViewAncestor (window).
     *
     * @return Returns true if the effective visibility of the view at this
     * point is opaque, regardless of the transparent region; returns false
     * if it is possible for underlying windows to be seen behind the view.
     *
     * {@hide}
     */
    public boolean gatherTransparentRegion(Region region) {
        final AttachInfo attachInfo = mAttachInfo;
        if (region != null && attachInfo != null) {
            final int pflags = mPrivateFlags;
            if ((pflags & PFLAG_SKIP_DRAW) == 0) {
                // The SKIP_DRAW flag IS NOT set, so this view draws. We need to
                // remove it from the transparent region.
                final int[] location = attachInfo.mTransparentLocation;
                getLocationInWindow(location);
                // When a view has Z value, then it will be better to leave some area below the view
                // for drawing shadow. The shadow outset is proportional to the Z value. Note that
                // the bottom part needs more offset than the left, top and right parts due to the
                // spot light effects.
                int shadowOffset = getZ() > 0 ? (int) getZ() : 0;
                region.op(location[0] - shadowOffset, location[1] - shadowOffset,
                        location[0] + mRight - mLeft + shadowOffset,
                        location[1] + mBottom - mTop + (shadowOffset * 3), Region.Op.DIFFERENCE);
            } else {
                if (mBackground != null && mBackground.getOpacity() != PixelFormat.TRANSPARENT) {
                    // The SKIP_DRAW flag IS set and the background drawable exists, we remove
                    // the background drawable's non-transparent parts from this transparent region.
                    applyDrawableToTransparentRegion(mBackground, region);
                }
                if (mForegroundInfo != null && mForegroundInfo.mDrawable != null
                        && mForegroundInfo.mDrawable.getOpacity() != PixelFormat.TRANSPARENT) {
                    // Similarly, we remove the foreground drawable's non-transparent parts.
                    applyDrawableToTransparentRegion(mForegroundInfo.mDrawable, region);
                }
                if (mDefaultFocusHighlight != null
                        && mDefaultFocusHighlight.getOpacity() != PixelFormat.TRANSPARENT) {
                    // Similarly, we remove the default focus highlight's non-transparent parts.
                    applyDrawableToTransparentRegion(mDefaultFocusHighlight, region);
                }
            }
        }
        return true;
    }

    /**
     * Play a sound effect for this view.
     *
     * <p>The framework will play sound effects for some built in actions, such as
     * clicking, but you may wish to play these effects in your widget,
     * for instance, for internal navigation.
     *
     * <p>The sound effect will only be played if sound effects are enabled by the user, and
     * {@link #isSoundEffectsEnabled()} is true.
     *
     * @param soundConstant One of the constants defined in {@link SoundEffectConstants}
     */
    public void playSoundEffect(int soundConstant) {
        if (mAttachInfo == null || mAttachInfo.mRootCallbacks == null || !isSoundEffectsEnabled()) {
            return;
        }
        mAttachInfo.mRootCallbacks.playSoundEffect(soundConstant);
    }

    /**
     * BZZZTT!!1!
     *
     * <p>Provide haptic feedback to the user for this view.
     *
```

```
     * <p>The framework will provide haptic feedback for some built in actions,
     * such as long presses, but you may wish to provide feedback for your
     * own widget.
     *
     * <p>The feedback will only be performed if
     * {@link #isHapticFeedbackEnabled()} is true.
     *
     * @param feedbackConstant One of the constants defined in
     * {@link HapticFeedbackConstants}
     */
    public boolean performHapticFeedback(int feedbackConstant) {
        return performHapticFeedback(feedbackConstant, 0);
    }

    /**
     * BZZZTT!!1!
     *
     * <p>Like {@link #performHapticFeedback(int)}, with additional options.
     *
     * @param feedbackConstant One of the constants defined in
     * {@link HapticFeedbackConstants}
     * @param flags Additional flags as per {@link HapticFeedbackConstants}.
     */
    public boolean performHapticFeedback(int feedbackConstant, int flags) {
        if (mAttachInfo == null) {
            return false;
        }
        //noinspection SimplifiableIfStatement
        if ((flags & HapticFeedbackConstants.FLAG_IGNORE_VIEW_SETTING) == 0
                && !isHapticFeedbackEnabled()) {
            return false;
        }
        return mAttachInfo.mRootCallbacks.performHapticFeedback(feedbackConstant,
                (flags & HapticFeedbackConstants.FLAG_IGNORE_GLOBAL_SETTING) != 0);
    }

    /**
     * Request that the visibility of the status bar or other screen/window
     * decorations be changed.
     *
     * <p>This method is used to put the over device UI into temporary modes
     * where the user's attention is focused more on the application content,
     * by dimming or hiding surrounding system affordances.  This is typically
     * used in conjunction with {@link Window#FEATURE_ACTION_BAR_OVERLAY
     * Window.FEATURE_ACTION_BAR_OVERLAY}, allowing the applications content
     * to be placed behind the action bar (and with these flags other system
     * affordances) so that smooth transitions between hiding and showing them
     * can be done.
     *
     * <p>Two representative examples of the use of system UI visibility is
     * implementing a content browsing application (like a magazine reader)
     * and a video playing application.
     *
     * <p>The first code shows a typical implementation of a View in a content
     * browsing application.  In this implementation, the application goes
     * into a content-oriented mode by hiding the status bar and action bar,
     * and putting the navigation elements into lights out mode.  The user can
     * then interact with content while in this mode.  Such an application should
     * provide an easy way for the user to toggle out of the mode (such as to
     * check information in the status bar or access notifications).  In the
     * implementation here, this is done simply by tapping on the content.
     *
     * {@sample development/samples/ApiDemos/src/com/example/android/apis/view/ContentBrowserActivity.java
     *      content}
     *
     * <p>This second code sample shows a typical implementation of a View
     * in a video playing application.  In this situation, while the video is
     * playing the application would like to go into a complete full-screen mode,
     * to use as much of the display as possible for the video.  When in this state
     * the user can not interact with the application; the system intercepts
     * touching on the screen to pop the UI out of full screen mode.  See
     * {@link #fitSystemWindows(Rect)} for a sample layout that goes with this code.
     *
     * {@sample development/samples/ApiDemos/src/com/example/android/apis/view/VideoPlayerActivity.java
     *      content}
     *
     * @param visibility  Bitwise-or of flags {@link #SYSTEM_UI_FLAG_LOW_PROFILE},
     * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, {@link #SYSTEM_UI_FLAG_FULLSCREEN},
     * {@link #SYSTEM_UI_FLAG_LAYOUT_STABLE}, {@link #SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION},
     * {@link #SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN}, {@link #SYSTEM_UI_FLAG_IMMERSIVE},
     * and {@link #SYSTEM_UI_FLAG_IMMERSIVE_STICKY}.
     */
```

```
public void setSystemUiVisibility(int visibility) {
    if (visibility != mSystemUiVisibility) {
        mSystemUiVisibility = visibility;
        if (mParent != null && mAttachInfo != null && !mAttachInfo.mRecomputeGlobalAttributes) {
            mParent.recomputeViewAttributes(this);
        }
    }
}

/**
 * Returns the last {@link #setSystemUiVisibility(int)} that this view has requested.
 * @return  Bitwise-or of flags {@link #SYSTEM_UI_FLAG_LOW_PROFILE},
 * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, {@link #SYSTEM_UI_FLAG_FULLSCREEN},
 * {@link #SYSTEM_UI_FLAG_LAYOUT_STABLE}, {@link #SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION},
 * {@link #SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN}, {@link #SYSTEM_UI_FLAG_IMMERSIVE},
 * and {@link #SYSTEM_UI_FLAG_IMMERSIVE_STICKY}.
 */
public int getSystemUiVisibility() {
    return mSystemUiVisibility;
}

/**
 * Returns the current system UI visibility that is currently set for
 * the entire window.  This is the combination of the
 * {@link #setSystemUiVisibility(int)} values supplied by all of the
 * views in the window.
 */
public int getWindowSystemUiVisibility() {
    return mAttachInfo != null ? mAttachInfo.mSystemUiVisibility : 0;
}

/**
 * Override to find out when the window's requested system UI visibility
 * has changed, that is the value returned by {@link #getWindowSystemUiVisibility()}.
 * This is different from the callbacks received through
 * {@link #setOnSystemUiVisibilityChangeListener(OnSystemUiVisibilityChangeListener)}
 * in that this is only telling you about the local request of the window,
 * not the actual values applied by the system.
 */
public void onWindowSystemUiVisibilityChanged(int visible) {
}

/**
 * Dispatch callbacks to {@link #onWindowSystemUiVisibilityChanged(int)} down
 * the view hierarchy.
 */
public void dispatchWindowSystemUiVisiblityChanged(int visible) {
    onWindowSystemUiVisibilityChanged(visible);
}

/**
 * Set a listener to receive callbacks when the visibility of the system bar changes.
 * @param l  The {@link OnSystemUiVisibilityChangeListener} to receive callbacks.
 */
public void setOnSystemUiVisibilityChangeListener(OnSystemUiVisibilityChangeListener l) {
    getListenerInfo().mOnSystemUiVisibilityChangeListener = l;
    if (mParent != null && mAttachInfo != null && !mAttachInfo.mRecomputeGlobalAttributes) {
        mParent.recomputeViewAttributes(this);
    }
}

/**
 * Dispatch callbacks to {@link #setOnSystemUiVisibilityChangeListener} down
 * the view hierarchy.
 */
public void dispatchSystemUiVisibilityChanged(int visibility) {
    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnSystemUiVisibilityChangeListener != null) {
        li.mOnSystemUiVisibilityChangeListener.onSystemUiVisibilityChange(
                visibility & PUBLIC_STATUS_BAR_VISIBILITY_MASK);
    }
}

boolean updateLocalSystemUiVisibility(int localValue, int localChanges) {
    int val = (mSystemUiVisibility&~localChanges) | (localValue&localChanges);
    if (val != mSystemUiVisibility) {
        setSystemUiVisibility(val);
        return true;
    }
    return false;
}
```

```java
/** @hide */
public void setDisabledSystemUiVisibility(int flags) {
    if (mAttachInfo != null) {
        if (mAttachInfo.mDisabledSystemUiVisibility != flags) {
            mAttachInfo.mDisabledSystemUiVisibility = flags;
            if (mParent != null) {
                mParent.recomputeViewAttributes(this);
            }
        }
    }
}

/**
 * Creates an image that the system displays during the drag and drop
 * operation. This is called a &amp;quot;drag shadow&amp;quot;. The default implementation
 * for a DragShadowBuilder based on a View returns an image that has exactly the same
 * appearance as the given View. The default also positions the center of the drag shadow
 * directly under the touch point. If no View is provided (the constructor with no parameters
 * is used), and {@link #onProvideShadowMetrics(Point,Point) onProvideShadowMetrics()} and
 * {@link #onDrawShadow(Canvas) onDrawShadow()} are not overridden, then the
 * default is an invisible drag shadow.
 * <p>
 * You are not required to use the View you provide to the constructor as the basis of the
 * drag shadow. The {@link #onDrawShadow(Canvas) onDrawShadow()} method allows you to draw
 * anything you want as the drag shadow.
 * </p>
 * <p>
 *  You pass a DragShadowBuilder object to the system when you start the drag. The system
 *  calls {@link #onProvideShadowMetrics(Point,Point) onProvideShadowMetrics()} to get the
 *  size and position of the drag shadow. It uses this data to construct a
 *  {@link android.graphics.Canvas} object, then it calls {@link #onDrawShadow(Canvas) onDrawShadow()}
 *  so that your application can draw the shadow image in the Canvas.
 * </p>
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For a guide to implementing drag and drop features, read the
 * <a href="{@docRoot}guide/topics/ui/drag-drop.html">Drag and Drop</a> developer guide.</p>
 * </div>
 */
public static class DragShadowBuilder {
    private final WeakReference<View> mView;

    /**
     * Constructs a shadow image builder based on a View. By default, the resulting drag
     * shadow will have the same appearance and dimensions as the View, with the touch point
     * over the center of the View.
     * @param view A View. Any View in scope can be used.
     */
    public DragShadowBuilder(View view) {
        mView = new WeakReference<View>(view);
    }

    /**
     * Construct a shadow builder object with no associated View.  This
     * constructor variant is only useful when the {@link #onProvideShadowMetrics(Point, Point)}
     * and {@link #onDrawShadow(Canvas)} methods are also overridden in order
     * to supply the drag shadow's dimensions and appearance without
     * reference to any View object. If they are not overridden, then the result is an
     * invisible drag shadow.
     */
    public DragShadowBuilder() {
        mView = new WeakReference<View>(null);
    }

    /**
     * Returns the View object that had been passed to the
     * {@link #View.DragShadowBuilder(View)}
     * constructor.  If that View parameter was {@code null} or if the
     * {@link #View.DragShadowBuilder()}
     * constructor was used to instantiate the builder object, this method will return
     * null.
     *
     * @return The View object associate with this builder object.
     */
    @SuppressWarnings({"JavadocReference"})
    final public View getView() {
        return mView.get();
    }

    /**
     * Provides the metrics for the shadow image. These include the dimensions of
```

```
     * the shadow image, and the point within that shadow that should
     * be centered under the touch location while dragging.
     * <p>
     * The default implementation sets the dimensions of the shadow to be the
     * same as the dimensions of the View itself and centers the shadow under
     * the touch point.
     * </p>
     *
     * @param outShadowSize A {@link android.graphics.Point} containing the width and height
     * of the shadow image. Your application must set {@link android.graphics.Point#x} to the
     * desired width and must set {@link android.graphics.Point#y} to the desired height of the
     * image.
     *
     * @param outShadowTouchPoint A {@link android.graphics.Point} for the position within the
     * shadow image that should be underneath the touch point during the drag and drop
     * operation. Your application must set {@link android.graphics.Point#x} to the
     * X coordinate and {@link android.graphics.Point#y} to the Y coordinate of this position.
     */
    public void onProvideShadowMetrics(Point outShadowSize, Point outShadowTouchPoint) {
        final View view = mView.get();
        if (view != null) {
            outShadowSize.set(view.getWidth(), view.getHeight());
            outShadowTouchPoint.set(outShadowSize.x / 2, outShadowSize.y / 2);
        } else {
            Log.e(View.VIEW_LOG_TAG, "Asked for drag thumb metrics but no view");
        }
    }

    /**
     * Draws the shadow image. The system creates the {@link android.graphics.Canvas} object
     * based on the dimensions it received from the
     * {@link #onProvideShadowMetrics(Point, Point)} callback.
     *
     * @param canvas A {@link android.graphics.Canvas} object in which to draw the shadow image.
     */
    public void onDrawShadow(Canvas canvas) {
        final View view = mView.get();
        if (view != null) {
            view.draw(canvas);
        } else {
            Log.e(View.VIEW_LOG_TAG, "Asked to draw drag shadow but no view");
        }
    }
}

/**
 * @deprecated Use {@link #startDragAndDrop(ClipData, DragShadowBuilder, Object, int)
 * startDragAndDrop()} for newer platform versions.
 */
@Deprecated
public final boolean startDrag(ClipData data, DragShadowBuilder shadowBuilder,
                               Object myLocalState, int flags) {
    return startDragAndDrop(data, shadowBuilder, myLocalState, flags);
}

/**
 * Starts a drag and drop operation. When your application calls this method, it passes a
 * {@link android.view.View.DragShadowBuilder} object to the system. The
 * system calls this object's {@link DragShadowBuilder#onProvideShadowMetrics(Point, Point)}
 * to get metrics for the drag shadow, and then calls the object's
 * {@link DragShadowBuilder#onDrawShadow(Canvas)} to draw the drag shadow itself.
 * <p>
 *  Once the system has the drag shadow, it begins the drag and drop operation by sending
 *  drag events to all the View objects in your application that are currently visible. It does
 *  this either by calling the View object's drag listener (an implementation of
 *  {@link android.view.View.OnDragListener#onDrag(View,DragEvent) onDrag()} or by calling the
 *  View object's {@link android.view.View#onDragEvent(DragEvent) onDragEvent()} method.
 *  Both are passed a {@link android.view.DragEvent} object that has a
 *  {@link android.view.DragEvent#getAction()} value of
 *  {@link android.view.DragEvent#ACTION_DRAG_STARTED}.
 * </p>
 * <p>
 * Your application can invoke {@link #startDragAndDrop(ClipData, DragShadowBuilder, Object,
 * int) startDragAndDrop()} on any attached View object. The View object does not need to be
 * the one used in {@link android.view.View.DragShadowBuilder}, nor does it need to be related
 * to the View the user selected for dragging.
 * </p>
 * @param data A {@link android.content.ClipData} object pointing to the data to be
 * transferred by the drag and drop operation.
 * @param shadowBuilder A {@link android.view.View.DragShadowBuilder} object for building the
 * drag shadow.
 * @param myLocalState An {@link java.lang.Object} containing local data about the drag and
```

```
     * drop operation. When dispatching drag events to views in the same activity this object
     * will be available through {@link android.view.DragEvent#getLocalState()}. Views in other
     * activities will not have access to this data ({@link android.view.DragEvent#getLocalState()}
     * will return null).
     * <p>
     * myLocalState is a lightweight mechanism for the sending information from the dragged View
     * to the target Views. For example, it can contain flags that differentiate between a
     * a copy operation and a move operation.
     * </p>
     * @param flags Flags that control the drag and drop operation. This can be set to 0 for no
     * flags, or any combination of the following:
     *     <ul>
     *         <li>{@link #DRAG_FLAG_GLOBAL}</li>
     *         <li>{@link #DRAG_FLAG_GLOBAL_PERSISTABLE_URI_PERMISSION}</li>
     *         <li>{@link #DRAG_FLAG_GLOBAL_PREFIX_URI_PERMISSION}</li>
     *         <li>{@link #DRAG_FLAG_GLOBAL_URI_READ}</li>
     *         <li>{@link #DRAG_FLAG_GLOBAL_URI_WRITE}</li>
     *         <li>{@link #DRAG_FLAG_OPAQUE}</li>
     *     </ul>
     * @return {@code true} if the method completes successfully, or
     * {@code false} if it fails anywhere. Returning {@code false} means the system was unable to
     * do a drag, and so no drag operation is in progress.
     */
    public final boolean startDragAndDrop(ClipData data, DragShadowBuilder shadowBuilder,
            Object myLocalState, int flags) {
        if (ViewDebug.DEBUG_DRAG) {
            Log.d(VIEW_LOG_TAG, "startDragAndDrop: data=" + data + " flags=" + flags);
        }
        if (mAttachInfo == null) {
            Log.w(VIEW_LOG_TAG, "startDragAndDrop called on a detached view.");
            return false;
        }

        if (data != null) {
            data.prepareToLeaveProcess((flags & View.DRAG_FLAG_GLOBAL) != 0);
        }

        boolean okay = false;

        Point shadowSize = new Point();
        Point shadowTouchPoint = new Point();
        shadowBuilder.onProvideShadowMetrics(shadowSize, shadowTouchPoint);

        if ((shadowSize.x < 0) || (shadowSize.y < 0) ||
                (shadowTouchPoint.x < 0) || (shadowTouchPoint.y < 0)) {
            throw new IllegalStateException("Drag shadow dimensions must not be negative");
        }

        if (ViewDebug.DEBUG_DRAG) {
            Log.d(VIEW_LOG_TAG, "drag shadow: width=" + shadowSize.x + " height=" + shadowSize.y
                    + " shadowX=" + shadowTouchPoint.x + " shadowY=" + shadowTouchPoint.y);
        }
        if (mAttachInfo.mDragSurface != null) {
            mAttachInfo.mDragSurface.release();
        }
        mAttachInfo.mDragSurface = new Surface();
        try {
            mAttachInfo.mDragToken = mAttachInfo.mSession.prepareDrag(mAttachInfo.mWindow,
                    flags, shadowSize.x, shadowSize.y, mAttachInfo.mDragSurface);
            if (ViewDebug.DEBUG_DRAG) Log.d(VIEW_LOG_TAG, "prepareDrag returned token="
                    + mAttachInfo.mDragToken + " surface=" + mAttachInfo.mDragSurface);
            if (mAttachInfo.mDragToken != null) {
                Canvas canvas = mAttachInfo.mDragSurface.lockCanvas(null);
                try {
                    canvas.drawColor(0, PorterDuff.Mode.CLEAR);
                    shadowBuilder.onDrawShadow(canvas);
                } finally {
                    mAttachInfo.mDragSurface.unlockCanvasAndPost(canvas);
                }

                final ViewRootImpl root = getViewRootImpl();

                // Cache the local state object for delivery with DragEvents
                root.setLocalDragState(myLocalState);

                // repurpose 'shadowSize' for the last touch point
                root.getLastTouchPoint(shadowSize);

                okay = mAttachInfo.mSession.performDrag(mAttachInfo.mWindow, mAttachInfo.mDragToken,
                        root.getLastTouchSource(), shadowSize.x, shadowSize.y,
                        shadowTouchPoint.x, shadowTouchPoint.y, data);
                if (ViewDebug.DEBUG_DRAG) Log.d(VIEW_LOG_TAG, "performDrag returned " + okay);
```

```
            }
        } catch (Exception e) {
            Log.e(VIEW_LOG_TAG, "Unable to initiate drag", e);
            mAttachInfo.mDragSurface.destroy();
            mAttachInfo.mDragSurface = null;
        }

        return okay;
    }

    /**
     * Cancels an ongoing drag and drop operation.
     * <p>
     * A {@link android.view.DragEvent} object with
     * {@link android.view.DragEvent#getAction()} value of
     * {@link android.view.DragEvent#ACTION_DRAG_ENDED} and
     * {@link android.view.DragEvent#getResult()} value of {@code false}
     * will be sent to every
     * View that received {@link android.view.DragEvent#ACTION_DRAG_STARTED}
     * even if they are not currently visible.
     * </p>
     * <p>
     * This method can be called on any View in the same window as the View on which
     * {@link #startDragAndDrop(ClipData, DragShadowBuilder, Object, int) startDragAndDrop}
     * was called.
     * </p>
     */
    public final void cancelDragAndDrop() {
        if (ViewDebug.DEBUG_DRAG) {
            Log.d(VIEW_LOG_TAG, "cancelDragAndDrop");
        }
        if (mAttachInfo == null) {
            Log.w(VIEW_LOG_TAG, "cancelDragAndDrop called on a detached view.");
            return;
        }
        if (mAttachInfo.mDragToken != null) {
            try {
                mAttachInfo.mSession.cancelDragAndDrop(mAttachInfo.mDragToken);
            } catch (Exception e) {
                Log.e(VIEW_LOG_TAG, "Unable to cancel drag", e);
            }
            mAttachInfo.mDragToken = null;
        } else {
            Log.e(VIEW_LOG_TAG, "No active drag to cancel");
        }
    }

    /**
     * Updates the drag shadow for the ongoing drag and drop operation.
     *
     * @param shadowBuilder A {@link android.view.View.DragShadowBuilder} object for building the
     * new drag shadow.
     */
    public final void updateDragShadow(DragShadowBuilder shadowBuilder) {
        if (ViewDebug.DEBUG_DRAG) {
            Log.d(VIEW_LOG_TAG, "updateDragShadow");
        }
        if (mAttachInfo == null) {
            Log.w(VIEW_LOG_TAG, "updateDragShadow called on a detached view.");
            return;
        }
        if (mAttachInfo.mDragToken != null) {
            try {
                Canvas canvas = mAttachInfo.mDragSurface.lockCanvas(null);
                try {
                    canvas.drawColor(0, PorterDuff.Mode.CLEAR);
                    shadowBuilder.onDrawShadow(canvas);
                } finally {
                    mAttachInfo.mDragSurface.unlockCanvasAndPost(canvas);
                }
            } catch (Exception e) {
                Log.e(VIEW_LOG_TAG, "Unable to update drag shadow", e);
            }
        } else {
            Log.e(VIEW_LOG_TAG, "No active drag");
        }
    }

    /**
     * Starts a move from {startX, startY}, the amount of the movement will be the offset
     * between {startX, startY} and the new cursor positon.
     * @param startX horizontal coordinate where the move started.
```

```
     * @param startY vertical coordinate where the move started.
     * @return whether moving was started successfully.
     * @hide
     */
    public final boolean startMovingTask(float startX, float startY) {
        if (ViewDebug.DEBUG_POSITIONING) {
            Log.d(VIEW_LOG_TAG, "startMovingTask: {" + startX + "," + startY + "}");
        }
        try {
            return mAttachInfo.mSession.startMovingTask(mAttachInfo.mWindow, startX, startY);
        } catch (RemoteException e) {
            Log.e(VIEW_LOG_TAG, "Unable to start moving", e);
        }
        return false;
    }

    /**
     * Handles drag events sent by the system following a call to
     * {@link android.view.View#startDragAndDrop(ClipData,DragShadowBuilder,Object,int)
     * startDragAndDrop()}.
     *<p>
     * When the system calls this method, it passes a
     * {@link android.view.DragEvent} object. A call to
     * {@link android.view.DragEvent#getAction()} returns one of the action type constants defined
     * in DragEvent. The method uses these to determine what is happening in the drag and drop
     * operation.
     * @param event The {@link android.view.DragEvent} sent by the system.
     * The {@link android.view.DragEvent#getAction()} method returns an action type constant defined
     * in DragEvent, indicating the type of drag event represented by this object.
     * @return {@code true} if the method was successful, otherwise {@code false}.
     * <p>
     *  The method should return {@code true} in response to an action type of
     *  {@link android.view.DragEvent#ACTION_DRAG_STARTED} to receive drag events for the current
     *  operation.
     * </p>
     * <p>
     *  The method should also return {@code true} in response to an action type of
     *  {@link android.view.DragEvent#ACTION_DROP} if it consumed the drop, or
     *  {@code false} if it didn't.
     * </p>
     * <p>
     *  For all other events, the return value is ignored.
     * </p>
     */
    public boolean onDragEvent(DragEvent event) {
        return false;
    }

    // Dispatches ACTION_DRAG_ENTERED and ACTION_DRAG_EXITED events for pre-Nougat apps.
    boolean dispatchDragEnterExitInPreN(DragEvent event) {
        return callDragEventHandler(event);
    }

    /**
     * Detects if this View is enabled and has a drag event listener.
     * If both are true, then it calls the drag event listener with the
     * {@link android.view.DragEvent} it received. If the drag event listener returns
     * {@code true}, then dispatchDragEvent() returns {@code true}.
     * <p>
     * For all other cases, the method calls the
     * {@link android.view.View#onDragEvent(DragEvent) onDragEvent()} drag event handler
     * method and returns its result.
     * </p>
     * <p>
     * This ensures that a drag event is always consumed, even if the View does not have a drag
     * event listener. However, if the View has a listener and the listener returns true, then
     * onDragEvent() is not called.
     * </p>
     */
    public boolean dispatchDragEvent(DragEvent event) {
        event.mEventHandlerWasCalled = true;
        if (event.mAction == DragEvent.ACTION_DRAG_LOCATION ||
            event.mAction == DragEvent.ACTION_DROP) {
            // About to deliver an event with coordinates to this view. Notify that now this view
            // has drag focus. This will send exit/enter events as needed.
            getViewRootImpl().setDragFocus(this, event);
        }
        return callDragEventHandler(event);
    }

    final boolean callDragEventHandler(DragEvent event) {
        final boolean result;
```

```
    ListenerInfo li = mListenerInfo;
    //noinspection SimplifiableIfStatement
    if (li != null && li.mOnDragListener != null && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnDragListener.onDrag(this, event)) {
        result = true;
    } else {
        result = onDragEvent(event);
    }

    switch (event.mAction) {
        case DragEvent.ACTION_DRAG_ENTERED: {
            mPrivateFlags2 |= View.PFLAG2_DRAG_HOVERED;
            refreshDrawableState();
        } break;
        case DragEvent.ACTION_DRAG_EXITED: {
            mPrivateFlags2 &= ~View.PFLAG2_DRAG_HOVERED;
            refreshDrawableState();
        } break;
        case DragEvent.ACTION_DRAG_ENDED: {
            mPrivateFlags2 &= ~View.DRAG_MASK;
            refreshDrawableState();
        } break;
    }

    return result;
}

boolean canAcceptDrag() {
    return (mPrivateFlags2 & PFLAG2_DRAG_CAN_ACCEPT) != 0;
}

/**
 * This needs to be a better API (NOT ON VIEW) before it is exposed.  If
 * it is ever exposed at all.
 * @hide
 */
public void onCloseSystemDialogs(String reason) {
}

/**
 * Given a Drawable whose bounds have been set to draw into this view,
 * update a Region being computed for
 * {@link #gatherTransparentRegion(android.graphics.Region)} so
 * that any non-transparent parts of the Drawable are removed from the
 * given transparent region.
 *
 * @param dr The Drawable whose transparency is to be applied to the region.
 * @param region A Region holding the current transparency information,
 * where any parts of the region that are set are considered to be
 * transparent.  On return, this region will be modified to have the
 * transparency information reduced by the corresponding parts of the
 * Drawable that are not transparent.
 * {@hide}
 */
public void applyDrawableToTransparentRegion(Drawable dr, Region region) {
    if (DBG) {
        Log.i("View", "Getting transparent region for: " + this);
    }
    final Region r = dr.getTransparentRegion();
    final Rect db = dr.getBounds();
    final AttachInfo attachInfo = mAttachInfo;
    if (r != null && attachInfo != null) {
        final int w = getRight()-getLeft();
        final int h = getBottom()-getTop();
        if (db.left > 0) {
            //Log.i("VIEW", "Drawable left " + db.left + " > view 0");
            r.op(0, 0, db.left, h, Region.Op.UNION);
        }
        if (db.right < w) {
            //Log.i("VIEW", "Drawable right " + db.right + " < view " + w);
            r.op(db.right, 0, w, h, Region.Op.UNION);
        }
        if (db.top > 0) {
            //Log.i("VIEW", "Drawable top " + db.top + " > view 0");
            r.op(0, 0, w, db.top, Region.Op.UNION);
        }
        if (db.bottom < h) {
            //Log.i("VIEW", "Drawable bottom " + db.bottom + " < view " + h);
            r.op(0, db.bottom, w, h, Region.Op.UNION);
        }
        final int[] location = attachInfo.mTransparentLocation;
```

```
        getLocationInWindow(location);
        r.translate(location[0], location[1]);
        region.op(r, Region.Op.INTERSECT);
    } else {
        region.op(db, Region.Op.DIFFERENCE);
    }
}

private void checkForLongClick(int delayOffset, float x, float y) {
    if ((mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE || (mViewFlags & TOOLTIP) == TOOLTIP) {
        mHasPerformedLongPress = false;

        if (mPendingCheckForLongPress == null) {
            mPendingCheckForLongPress = new CheckForLongPress();
        }
        mPendingCheckForLongPress.setAnchor(x, y);
        mPendingCheckForLongPress.rememberWindowAttachCount();
        mPendingCheckForLongPress.rememberPressedState();
        postDelayed(mPendingCheckForLongPress,
                ViewConfiguration.getLongPressTimeout() - delayOffset);
    }
}

/**
 * Inflate a view from an XML resource.  This convenience method wraps the {@link
 * LayoutInflater} class, which provides a full range of options for view inflation.
 *
 * @param context The Context object for your activity or application.
 * @param resource The resource ID to inflate
 * @param root A view group that will be the parent.  Used to properly inflate the
 * layout_* parameters.
 * @see LayoutInflater
 */
public static View inflate(Context context, @LayoutRes int resource, ViewGroup root) {
    LayoutInflater factory = LayoutInflater.from(context);
    return factory.inflate(resource, root);
}

/**
 * Scroll the view with standard behavior for scrolling beyond the normal
 * content boundaries. Views that call this method should override
 * {@link #onOverScrolled(int, int, boolean, boolean)} to respond to the
 * results of an over-scroll operation.
 *
 * Views can use this method to handle any touch or fling-based scrolling.
 *
 * @param deltaX Change in X in pixels
 * @param deltaY Change in Y in pixels
 * @param scrollX Current X scroll value in pixels before applying deltaX
 * @param scrollY Current Y scroll value in pixels before applying deltaY
 * @param scrollRangeX Maximum content scroll range along the X axis
 * @param scrollRangeY Maximum content scroll range along the Y axis
 * @param maxOverScrollX Number of pixels to overscroll by in either direction
 *          along the X axis.
 * @param maxOverScrollY Number of pixels to overscroll by in either direction
 *          along the Y axis.
 * @param isTouchEvent true if this scroll operation is the result of a touch event.
 * @return true if scrolling was clamped to an over-scroll boundary along either
 *          axis, false otherwise.
 */
@SuppressWarnings({"UnusedParameters"})
protected boolean overScrollBy(int deltaX, int deltaY,
        int scrollX, int scrollY,
        int scrollRangeX, int scrollRangeY,
        int maxOverScrollX, int maxOverScrollY,
        boolean isTouchEvent) {
    final int overScrollMode = mOverScrollMode;
    final boolean canScrollHorizontal =
            computeHorizontalScrollRange() > computeHorizontalScrollExtent();
    final boolean canScrollVertical =
            computeVerticalScrollRange() > computeVerticalScrollExtent();
    final boolean overScrollHorizontal = overScrollMode == OVER_SCROLL_ALWAYS ||
            (overScrollMode == OVER_SCROLL_IF_CONTENT_SCROLLS && canScrollHorizontal);
    final boolean overScrollVertical = overScrollMode == OVER_SCROLL_ALWAYS ||
            (overScrollMode == OVER_SCROLL_IF_CONTENT_SCROLLS && canScrollVertical);

    int newScrollX = scrollX + deltaX;
    if (!overScrollHorizontal) {
        maxOverScrollX = 0;
    }

    int newScrollY = scrollY + deltaY;
```

```
    if (!overScrollVertical) {
        maxOverScrollY = 0;
    }

    // Clamp values if at the limits and record
    final int left = -maxOverScrollX;
    final int right = maxOverScrollX + scrollRangeX;
    final int top = -maxOverScrollY;
    final int bottom = maxOverScrollY + scrollRangeY;

    boolean clampedX = false;
    if (newScrollX > right) {
        newScrollX = right;
        clampedX = true;
    } else if (newScrollX < left) {
        newScrollX = left;
        clampedX = true;
    }

    boolean clampedY = false;
    if (newScrollY > bottom) {
        newScrollY = bottom;
        clampedY = true;
    } else if (newScrollY < top) {
        newScrollY = top;
        clampedY = true;
    }

    onOverScrolled(newScrollX, newScrollY, clampedX, clampedY);

    return clampedX || clampedY;
}

/**
 * Called by {@link #overScrollBy(int, int, int, int, int, int, int, int, boolean)} to
 * respond to the results of an over-scroll operation.
 *
 * @param scrollX New X scroll value in pixels
 * @param scrollY New Y scroll value in pixels
 * @param clampedX True if scrollX was clamped to an over-scroll boundary
 * @param clampedY True if scrollY was clamped to an over-scroll boundary
 */
protected void onOverScrolled(int scrollX, int scrollY,
        boolean clampedX, boolean clampedY) {
    // Intentionally empty.
}

/**
 * Returns the over-scroll mode for this view. The result will be
 * one of {@link #OVER_SCROLL_ALWAYS} (default), {@link #OVER_SCROLL_IF_CONTENT_SCROLLS}
 * (allow over-scrolling only if the view content is larger than the container),
 * or {@link #OVER_SCROLL_NEVER}.
 *
 * @return This view's over-scroll mode.
 */
public int getOverScrollMode() {
    return mOverScrollMode;
}

/**
 * Set the over-scroll mode for this view. Valid over-scroll modes are
 * {@link #OVER_SCROLL_ALWAYS} (default), {@link #OVER_SCROLL_IF_CONTENT_SCROLLS}
 * (allow over-scrolling only if the view content is larger than the container),
 * or {@link #OVER_SCROLL_NEVER}.
 *
 * Setting the over-scroll mode of a view will have an effect only if the
 * view is capable of scrolling.
 *
 * @param overScrollMode The new over-scroll mode for this view.
 */
public void setOverScrollMode(int overScrollMode) {
    if (overScrollMode != OVER_SCROLL_ALWAYS &&
            overScrollMode != OVER_SCROLL_IF_CONTENT_SCROLLS &&
            overScrollMode != OVER_SCROLL_NEVER) {
        throw new IllegalArgumentException("Invalid overscroll mode " + overScrollMode);
    }
    mOverScrollMode = overScrollMode;
}

/**
 * Enable or disable nested scrolling for this view.
 *
```

```
     * <p>If this property is set to true the view will be permitted to initiate nested
     * scrolling operations with a compatible parent view in the current hierarchy. If this
     * view does not implement nested scrolling this will have no effect. Disabling nested scrolling
     * while a nested scroll is in progress has the effect of {@link #stopNestedScroll() stopping}
     * the nested scroll.</p>
     *
     * @param enabled true to enable nested scrolling, false to disable
     *
     * @see #isNestedScrollingEnabled()
     */
    public void setNestedScrollingEnabled(boolean enabled) {
        if (enabled) {
            mPrivateFlags3 |= PFLAG3_NESTED_SCROLLING_ENABLED;
        } else {
            stopNestedScroll();
            mPrivateFlags3 &= ~PFLAG3_NESTED_SCROLLING_ENABLED;
        }
    }

    /**
     * Returns true if nested scrolling is enabled for this view.
     *
     * <p>If nested scrolling is enabled and this View class implementation supports it,
     * this view will act as a nested scrolling child view when applicable, forwarding data
     * about the scroll operation in progress to a compatible and cooperating nested scrolling
     * parent.</p>
     *
     * @return true if nested scrolling is enabled
     *
     * @see #setNestedScrollingEnabled(boolean)
     */
    public boolean isNestedScrollingEnabled() {
        return (mPrivateFlags3 & PFLAG3_NESTED_SCROLLING_ENABLED) ==
                PFLAG3_NESTED_SCROLLING_ENABLED;
    }

    /**
     * Begin a nestable scroll operation along the given axes.
     *
     * <p>A view starting a nested scroll promises to abide by the following contract:</p>
     *
     * <p>The view will call startNestedScroll upon initiating a scroll operation. In the case
     * of a touch scroll this corresponds to the initial {@link MotionEvent#ACTION_DOWN}.
     * In the case of touch scrolling the nested scroll will be terminated automatically in
     * the same manner as {@link ViewParent#requestDisallowInterceptTouchEvent(boolean)}.
     * In the event of programmatic scrolling the caller must explicitly call
     * {@link #stopNestedScroll()} to indicate the end of the nested scroll.</p>
     *
     * <p>If <code>startNestedScroll</code> returns true, a cooperative parent was found.
     * If it returns false the caller may ignore the rest of this contract until the next scroll.
     * Calling startNestedScroll while a nested scroll is already in progress will return true.</p>
     *
     * <p>At each incremental step of the scroll the caller should invoke
     * {@link #dispatchNestedPreScroll(int, int, int[], int[]) dispatchNestedPreScroll}
     * once it has calculated the requested scrolling delta. If it returns true the nested scrolling
     * parent at least partially consumed the scroll and the caller should adjust the amount it
     * scrolls by.</p>
     *
     * <p>After applying the remainder of the scroll delta the caller should invoke
     * {@link #dispatchNestedScroll(int, int, int, int, int[]) dispatchNestedScroll}, passing
     * both the delta consumed and the delta unconsumed. A nested scrolling parent may treat
     * these values differently. See {@link ViewParent#onNestedScroll(View, int, int, int, int)}.
     * </p>
     *
     * @param axes Flags consisting of a combination of {@link #SCROLL_AXIS_HORIZONTAL} and/or
     *             {@link #SCROLL_AXIS_VERTICAL}.
     * @return true if a cooperative parent was found and nested scrolling has been enabled for
     *         the current gesture.
     *
     * @see #stopNestedScroll()
     * @see #dispatchNestedPreScroll(int, int, int[], int[])
     * @see #dispatchNestedScroll(int, int, int, int, int[])
     */
    public boolean startNestedScroll(int axes) {
        if (hasNestedScrollingParent()) {
            // Already in progress
            return true;
        }
        if (isNestedScrollingEnabled()) {
            ViewParent p = getParent();
            View child = this;
            while (p != null) {
```

```
            try {
                if (p.onStartNestedScroll(child, this, axes)) {
                    mNestedScrollingParent = p;
                    p.onNestedScrollAccepted(child, this, axes);
                    return true;
                }
            } catch (AbstractMethodError e) {
                Log.e(VIEW_LOG_TAG, "ViewParent " + p + " does not implement interface " +
                        "method onStartNestedScroll", e);
                // Allow the search upward to continue
            }
            if (p instanceof View) {
                child = (View) p;
            }
            p = p.getParent();
        }
    }
    return false;
}

/**
 * Stop a nested scroll in progress.
 *
 * <p>Calling this method when a nested scroll is not currently in progress is harmless.</p>
 *
 * @see #startNestedScroll(int)
 */
public void stopNestedScroll() {
    if (mNestedScrollingParent != null) {
        mNestedScrollingParent.onStopNestedScroll(this);
        mNestedScrollingParent = null;
    }
}

/**
 * Returns true if this view has a nested scrolling parent.
 *
 * <p>The presence of a nested scrolling parent indicates that this view has initiated
 * a nested scroll and it was accepted by an ancestor view further up the view hierarchy.</p>
 *
 * @return whether this view has a nested scrolling parent
 */
public boolean hasNestedScrollingParent() {
    return mNestedScrollingParent != null;
}

/**
 * Dispatch one step of a nested scroll in progress.
 *
 * <p>Implementations of views that support nested scrolling should call this to report
 * info about a scroll in progress to the current nested scrolling parent. If a nested scroll
 * is not currently in progress or nested scrolling is not
 * {@link #isNestedScrollingEnabled() enabled} for this view this method does nothing.</p>
 *
 * <p>Compatible View implementations should also call
 * {@link #dispatchNestedPreScroll(int, int, int[], int[]) dispatchNestedPreScroll} before
 * consuming a component of the scroll event themselves.</p>
 *
 * @param dxConsumed Horizontal distance in pixels consumed by this view during this scroll step
 * @param dyConsumed Vertical distance in pixels consumed by this view during this scroll step
 * @param dxUnconsumed Horizontal scroll distance in pixels not consumed by this view
 * @param dyUnconsumed Horizontal scroll distance in pixels not consumed by this view
 * @param offsetInWindow Optional. If not null, on return this will contain the offset
 *                       in local view coordinates of this view from before this operation
 *                       to after it completes. View implementations may use this to adjust
 *                       expected input coordinate tracking.
 * @return true if the event was dispatched, false if it could not be dispatched.
 * @see #dispatchNestedPreScroll(int, int, int[], int[])
 */
public boolean dispatchNestedScroll(int dxConsumed, int dyConsumed,
        int dxUnconsumed, int dyUnconsumed, @Nullable @Size(2) int[] offsetInWindow) {
    if (isNestedScrollingEnabled() && mNestedScrollingParent != null) {
        if (dxConsumed != 0 || dyConsumed != 0 || dxUnconsumed != 0 || dyUnconsumed != 0) {
            int startX = 0;
            int startY = 0;
            if (offsetInWindow != null) {
                getLocationInWindow(offsetInWindow);
                startX = offsetInWindow[0];
                startY = offsetInWindow[1];
            }

            mNestedScrollingParent.onNestedScroll(this, dxConsumed, dyConsumed,
```

```
                    dxUnconsumed, dyUnconsumed);

            if (offsetInWindow != null) {
                getLocationInWindow(offsetInWindow);
                offsetInWindow[0] -= startX;
                offsetInWindow[1] -= startY;
            }
            return true;
        } else if (offsetInWindow != null) {
            // No motion, no dispatch. Keep offsetInWindow up to date.
            offsetInWindow[0] = 0;
            offsetInWindow[1] = 0;
        }
    }
    return false;
}

/**
 * Dispatch one step of a nested scroll in progress before this view consumes any portion of it.
 *
 * <p>Nested pre-scroll events are to nested scroll events what touch intercept is to touch.
 * <code>dispatchNestedPreScroll</code> offers an opportunity for the parent view in a nested
 * scrolling operation to consume some or all of the scroll operation before the child view
 * consumes it.</p>
 *
 * @param dx Horizontal scroll distance in pixels
 * @param dy Vertical scroll distance in pixels
 * @param consumed Output. If not null, consumed[0] will contain the consumed component of dx
 *                 and consumed[1] the consumed dy.
 * @param offsetInWindow Optional. If not null, on return this will contain the offset
 *                       in local view coordinates of this view from before this operation
 *                       to after it completes. View implementations may use this to adjust
 *                       expected input coordinate tracking.
 * @return true if the parent consumed some or all of the scroll delta
 * @see #dispatchNestedScroll(int, int, int, int, int[])
 */
public boolean dispatchNestedPreScroll(int dx, int dy,
        @Nullable @Size(2) int[] consumed, @Nullable @Size(2) int[] offsetInWindow) {
    if (isNestedScrollingEnabled() && mNestedScrollingParent != null) {
        if (dx != 0 || dy != 0) {
            int startX = 0;
            int startY = 0;
            if (offsetInWindow != null) {
                getLocationInWindow(offsetInWindow);
                startX = offsetInWindow[0];
                startY = offsetInWindow[1];
            }

            if (consumed == null) {
                if (mTempNestedScrollConsumed == null) {
                    mTempNestedScrollConsumed = new int[2];
                }
                consumed = mTempNestedScrollConsumed;
            }
            consumed[0] = 0;
            consumed[1] = 0;
            mNestedScrollingParent.onNestedPreScroll(this, dx, dy, consumed);

            if (offsetInWindow != null) {
                getLocationInWindow(offsetInWindow);
                offsetInWindow[0] -= startX;
                offsetInWindow[1] -= startY;
            }
            return consumed[0] != 0 || consumed[1] != 0;
        } else if (offsetInWindow != null) {
            offsetInWindow[0] = 0;
            offsetInWindow[1] = 0;
        }
    }
    return false;
}

/**
 * Dispatch a fling to a nested scrolling parent.
 *
 * <p>This method should be used to indicate that a nested scrolling child has detected
 * suitable conditions for a fling. Generally this means that a touch scroll has ended with a
 * {@link VelocityTracker velocity} in the direction of scrolling that meets or exceeds
 * the {@link ViewConfiguration#getScaledMinimumFlingVelocity() minimum fling velocity}
 * along a scrollable axis.</p>
 *
 * <p>If a nested scrolling child view would normally fling but it is at the edge of
```

```
     * its own content, it can use this method to delegate the fling to its nested scrolling
     * parent instead. The parent may optionally consume the fling or observe a child fling.</p>
     *
     * @param velocityX Horizontal fling velocity in pixels per second
     * @param velocityY Vertical fling velocity in pixels per second
     * @param consumed true if the child consumed the fling, false otherwise
     * @return true if the nested scrolling parent consumed or otherwise reacted to the fling
     */
    public boolean dispatchNestedFling(float velocityX, float velocityY, boolean consumed) {
        if (isNestedScrollingEnabled() && mNestedScrollingParent != null) {
            return mNestedScrollingParent.onNestedFling(this, velocityX, velocityY, consumed);
        }
        return false;
    }

    /**
     * Dispatch a fling to a nested scrolling parent before it is processed by this view.
     *
     * <p>Nested pre-fling events are to nested fling events what touch intercept is to touch
     * and what nested pre-scroll is to nested scroll. <code>dispatchNestedPreFling</code>
     * offsets an opportunity for the parent view in a nested fling to fully consume the fling
     * before the child view consumes it. If this method returns <code>true</code>, a nested
     * parent view consumed the fling and this view should not scroll as a result.</p>
     *
     * <p>For a better user experience, only one view in a nested scrolling chain should consume
     * the fling at a time. If a parent view consumed the fling this method will return false.
     * Custom view implementations should account for this in two ways:</p>
     *
     * <ul>
     *     <li>If a custom view is paged and needs to settle to a fixed page-point, do not
     *     call <code>dispatchNestedPreFling</code>; consume the fling and settle to a valid
     *     position regardless.</li>
     *     <li>If a nested parent does consume the fling, this view should not scroll at all,
     *     even to settle back to a valid idle position.</li>
     * </ul>
     *
     * <p>Views should also not offer fling velocities to nested parent views along an axis
     * where scrolling is not currently supported; a {@link android.widget.ScrollView ScrollView}
     * should not offer a horizontal fling velocity to its parents since scrolling along that
     * axis is not permitted and carrying velocity along that motion does not make sense.</p>
     *
     * @param velocityX Horizontal fling velocity in pixels per second
     * @param velocityY Vertical fling velocity in pixels per second
     * @return true if a nested scrolling parent consumed the fling
     */
    public boolean dispatchNestedPreFling(float velocityX, float velocityY) {
        if (isNestedScrollingEnabled() && mNestedScrollingParent != null) {
            return mNestedScrollingParent.onNestedPreFling(this, velocityX, velocityY);
        }
        return false;
    }

    /**
     * Gets a scale factor that determines the distance the view should scroll
     * vertically in response to {@link MotionEvent#ACTION_SCROLL}.
     * @return The vertical scroll scale factor.
     * @hide
     */
    protected float getVerticalScrollFactor() {
        if (mVerticalScrollFactor == 0) {
            TypedValue outValue = new TypedValue();
            if (!mContext.getTheme().resolveAttribute(
                    com.android.internal.R.attr.listPreferredItemHeight, outValue, true)) {
                throw new IllegalStateException(
                        "Expected theme to define listPreferredItemHeight.");
            }
            mVerticalScrollFactor = outValue.getDimension(
                    mContext.getResources().getDisplayMetrics());
        }
        return mVerticalScrollFactor;
    }

    /**
     * Gets a scale factor that determines the distance the view should scroll
     * horizontally in response to {@link MotionEvent#ACTION_SCROLL}.
     * @return The horizontal scroll scale factor.
     * @hide
     */
    protected float getHorizontalScrollFactor() {
        // TODO: Should use something else.
        return getVerticalScrollFactor();
    }
```

```
/**
 * Return the value specifying the text direction or policy that was set with
 * {@link #setTextDirection(int)}.
 *
 * @return the defined text direction. It can be one of:
 *
 * {@link #TEXT_DIRECTION_INHERIT},
 * {@link #TEXT_DIRECTION_FIRST_STRONG},
 * {@link #TEXT_DIRECTION_ANY_RTL},
 * {@link #TEXT_DIRECTION_LTR},
 * {@link #TEXT_DIRECTION_RTL},
 * {@link #TEXT_DIRECTION_LOCALE},
 * {@link #TEXT_DIRECTION_FIRST_STRONG_LTR},
 * {@link #TEXT_DIRECTION_FIRST_STRONG_RTL}
 *
 * @attr ref android.R.styleable#View_textDirection
 *
 * @hide
 */
@ViewDebug.ExportedProperty(category = "text", mapping = {
        @ViewDebug.IntToString(from = TEXT_DIRECTION_INHERIT, to = "INHERIT"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG, to = "FIRST_STRONG"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_ANY_RTL, to = "ANY_RTL"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_LTR, to = "LTR"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_RTL, to = "RTL"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_LOCALE, to = "LOCALE"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG_LTR, to = "FIRST_STRONG_LTR"),
        @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG_RTL, to = "FIRST_STRONG_RTL")
})
public int getRawTextDirection() {
    return (mPrivateFlags2 & PFLAG2_TEXT_DIRECTION_MASK) >> PFLAG2_TEXT_DIRECTION_MASK_SHIFT;
}

/**
 * Set the text direction.
 *
 * @param textDirection the direction to set. Should be one of:
 *
 * {@link #TEXT_DIRECTION_INHERIT},
 * {@link #TEXT_DIRECTION_FIRST_STRONG},
 * {@link #TEXT_DIRECTION_ANY_RTL},
 * {@link #TEXT_DIRECTION_LTR},
 * {@link #TEXT_DIRECTION_RTL},
 * {@link #TEXT_DIRECTION_LOCALE}
 * {@link #TEXT_DIRECTION_FIRST_STRONG_LTR},
 * {@link #TEXT_DIRECTION_FIRST_STRONG_RTL},
 *
 * Resolution will be done if the value is set to TEXT_DIRECTION_INHERIT. The resolution
 * proceeds up the parent chain of the view to get the value. If there is no parent, then it will
 * return the default {@link #TEXT_DIRECTION_FIRST_STRONG}.
 *
 * @attr ref android.R.styleable#View_textDirection
 */
public void setTextDirection(int textDirection) {
    if (getRawTextDirection() != textDirection) {
        // Reset the current text direction and the resolved one
        mPrivateFlags2 &= ~PFLAG2_TEXT_DIRECTION_MASK;
        resetResolvedTextDirection();
        // Set the new text direction
        mPrivateFlags2 |= ((textDirection << PFLAG2_TEXT_DIRECTION_MASK_SHIFT) & PFLAG2_TEXT_DIRECTION_MASK);
        // Do resolution
        resolveTextDirection();
        // Notify change
        onRtlPropertiesChanged(getLayoutDirection());
        // Refresh
        requestLayout();
        invalidate(true);
    }
}

/**
 * Return the resolved text direction.
 *
 * @return the resolved text direction. Returns one of:
 *
 * {@link #TEXT_DIRECTION_FIRST_STRONG},
 * {@link #TEXT_DIRECTION_ANY_RTL},
 * {@link #TEXT_DIRECTION_LTR},
 * {@link #TEXT_DIRECTION_RTL},
 * {@link #TEXT_DIRECTION_LOCALE},
 * {@link #TEXT_DIRECTION_FIRST_STRONG_LTR},
```

```
     * {@link #TEXT_DIRECTION_FIRST_STRONG_RTL}
     *
     * @attr ref android.R.styleable#View_textDirection
     */
    @ViewDebug.ExportedProperty(category = "text", mapping = {
            @ViewDebug.IntToString(from = TEXT_DIRECTION_INHERIT, to = "INHERIT"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG, to = "FIRST_STRONG"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_ANY_RTL, to = "ANY_RTL"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_LTR, to = "LTR"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_RTL, to = "RTL"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_LOCALE, to = "LOCALE"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG_LTR, to = "FIRST_STRONG_LTR"),
            @ViewDebug.IntToString(from = TEXT_DIRECTION_FIRST_STRONG_RTL, to = "FIRST_STRONG_RTL")
    })
    public int getTextDirection() {
        return (mPrivateFlags2 & PFLAG2_TEXT_DIRECTION_RESOLVED_MASK) >> PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT;
    }

    /**
     * Resolve the text direction.
     *
     * @return true if resolution has been done, false otherwise.
     *
     * @hide
     */
    public boolean resolveTextDirection() {
        // Reset any previous text direction resolution
        mPrivateFlags2 &= ~(PFLAG2_TEXT_DIRECTION_RESOLVED | PFLAG2_TEXT_DIRECTION_RESOLVED_MASK);

        if (hasRtlSupport()) {
            // Set resolved text direction flag depending on text direction flag
            final int textDirection = getRawTextDirection();
            switch(textDirection) {
                case TEXT_DIRECTION_INHERIT:
                    if (!canResolveTextDirection()) {
                        // We cannot do the resolution if there is no parent, so use the default one
                        mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
                        // Resolution will need to happen again later
                        return false;
                    }

                    // Parent has not yet resolved, so we still return the default
                    try {
                        if (!mParent.isTextDirectionResolved()) {
                            mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
                            // Resolution will need to happen again later
                            return false;
                        }
                    } catch (AbstractMethodError e) {
                        Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                                " does not fully implement ViewParent", e);
                        mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED |
                                PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
                        return true;
                    }

                    // Set current resolved direction to the same value as the parent's one
                    int parentResolvedDirection;
                    try {
                        parentResolvedDirection = mParent.getTextDirection();
                    } catch (AbstractMethodError e) {
                        Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                                " does not fully implement ViewParent", e);
                        parentResolvedDirection = TEXT_DIRECTION_LTR;
                    }
                    switch (parentResolvedDirection) {
                        case TEXT_DIRECTION_FIRST_STRONG:
                        case TEXT_DIRECTION_ANY_RTL:
                        case TEXT_DIRECTION_LTR:
                        case TEXT_DIRECTION_RTL:
                        case TEXT_DIRECTION_LOCALE:
                        case TEXT_DIRECTION_FIRST_STRONG_LTR:
                        case TEXT_DIRECTION_FIRST_STRONG_RTL:
                            mPrivateFlags2 |=
                                    (parentResolvedDirection << PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT);
                            break;
                        default:
                            // Default resolved direction is "first strong" heuristic
                            mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
                    }
                    break;
                case TEXT_DIRECTION_FIRST_STRONG:
```

```
            case TEXT_DIRECTION_ANY_RTL:
            case TEXT_DIRECTION_LTR:
            case TEXT_DIRECTION_RTL:
            case TEXT_DIRECTION_LOCALE:
            case TEXT_DIRECTION_FIRST_STRONG_LTR:
            case TEXT_DIRECTION_FIRST_STRONG_RTL:
                // Resolved direction is the same as text direction
                mPrivateFlags2 |= (textDirection << PFLAG2_TEXT_DIRECTION_RESOLVED_MASK_SHIFT);
                break;
            default:
                // Default resolved direction is "first strong" heuristic
                mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
        }
    } else {
        // Default resolved direction is "first strong" heuristic
        mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
    }

    // Set to resolved
    mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED;
    return true;
}

/**
 * Check if text direction resolution can be done.
 *
 * @return true if text direction resolution can be done otherwise return false.
 */
public boolean canResolveTextDirection() {
    switch (getRawTextDirection()) {
        case TEXT_DIRECTION_INHERIT:
            if (mParent != null) {
                try {
                    return mParent.canResolveTextDirection();
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
            }
            return false;

        default:
            return true;
    }
}

/**
 * Reset resolved text direction. Text direction will be resolved during a call to
 * {@link #onMeasure(int, int)}.
 *
 * @hide
 */
public void resetResolvedTextDirection() {
    // Reset any previous text direction resolution
    mPrivateFlags2 &= ~(PFLAG2_TEXT_DIRECTION_RESOLVED | PFLAG2_TEXT_DIRECTION_RESOLVED_MASK);
    // Set to default value
    mPrivateFlags2 |= PFLAG2_TEXT_DIRECTION_RESOLVED_DEFAULT;
}

/**
 * @return true if text direction is inherited.
 *
 * @hide
 */
public boolean isTextDirectionInherited() {
    return (getRawTextDirection() == TEXT_DIRECTION_INHERIT);
}

/**
 * @return true if text direction is resolved.
 */
public boolean isTextDirectionResolved() {
    return (mPrivateFlags2 & PFLAG2_TEXT_DIRECTION_RESOLVED) == PFLAG2_TEXT_DIRECTION_RESOLVED;
}

/**
 * Return the value specifying the text alignment or policy that was set with
 * {@link #setTextAlignment(int)}.
 *
 * @return the defined text alignment. It can be one of:
 *
 * {@link #TEXT_ALIGNMENT_INHERIT},
```

```
     * {@link #TEXT_ALIGNMENT_GRAVITY},
     * {@link #TEXT_ALIGNMENT_CENTER},
     * {@link #TEXT_ALIGNMENT_TEXT_START},
     * {@link #TEXT_ALIGNMENT_TEXT_END},
     * {@link #TEXT_ALIGNMENT_VIEW_START},
     * {@link #TEXT_ALIGNMENT_VIEW_END}
     *
     * @attr ref android.R.styleable#View_textAlignment
     *
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "text", mapping = {
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_INHERIT, to = "INHERIT"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_GRAVITY, to = "GRAVITY"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_TEXT_START, to = "TEXT_START"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_TEXT_END, to = "TEXT_END"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_CENTER, to = "CENTER"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_VIEW_START, to = "VIEW_START"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_VIEW_END, to = "VIEW_END")
    })
    @TextAlignment
    public int getRawTextAlignment() {
        return (mPrivateFlags2 & PFLAG2_TEXT_ALIGNMENT_MASK) >> PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT;
    }

    /**
     * Set the text alignment.
     *
     * @param textAlignment The text alignment to set. Should be one of
     *
     * {@link #TEXT_ALIGNMENT_INHERIT},
     * {@link #TEXT_ALIGNMENT_GRAVITY},
     * {@link #TEXT_ALIGNMENT_CENTER},
     * {@link #TEXT_ALIGNMENT_TEXT_START},
     * {@link #TEXT_ALIGNMENT_TEXT_END},
     * {@link #TEXT_ALIGNMENT_VIEW_START},
     * {@link #TEXT_ALIGNMENT_VIEW_END}
     *
     * Resolution will be done if the value is set to TEXT_ALIGNMENT_INHERIT. The resolution
     * proceeds up the parent chain of the view to get the value. If there is no parent, then it
     * will return the default {@link #TEXT_ALIGNMENT_GRAVITY}.
     *
     * @attr ref android.R.styleable#View_textAlignment
     */
    public void setTextAlignment(@TextAlignment int textAlignment) {
        if (textAlignment != getRawTextAlignment()) {
            // Reset the current and resolved text alignment
            mPrivateFlags2 &= ~PFLAG2_TEXT_ALIGNMENT_MASK;
            resetResolvedTextAlignment();
            // Set the new text alignment
            mPrivateFlags2 |=
                    ((textAlignment << PFLAG2_TEXT_ALIGNMENT_MASK_SHIFT) & PFLAG2_TEXT_ALIGNMENT_MASK);
            // Do resolution
            resolveTextAlignment();
            // Notify change
            onRtlPropertiesChanged(getLayoutDirection());
            // Refresh
            requestLayout();
            invalidate(true);
        }
    }

    /**
     * Return the resolved text alignment.
     *
     * @return the resolved text alignment. Returns one of:
     *
     * {@link #TEXT_ALIGNMENT_GRAVITY},
     * {@link #TEXT_ALIGNMENT_CENTER},
     * {@link #TEXT_ALIGNMENT_TEXT_START},
     * {@link #TEXT_ALIGNMENT_TEXT_END},
     * {@link #TEXT_ALIGNMENT_VIEW_START},
     * {@link #TEXT_ALIGNMENT_VIEW_END}
     *
     * @attr ref android.R.styleable#View_textAlignment
     */
    @ViewDebug.ExportedProperty(category = "text", mapping = {
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_INHERIT, to = "INHERIT"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_GRAVITY, to = "GRAVITY"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_TEXT_START, to = "TEXT_START"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_TEXT_END, to = "TEXT_END"),
            @ViewDebug.IntToString(from = TEXT_ALIGNMENT_CENTER, to = "CENTER"),
```

```
        @ViewDebug.IntToString(from = TEXT_ALIGNMENT_VIEW_START, to = "VIEW_START"),
        @ViewDebug.IntToString(from = TEXT_ALIGNMENT_VIEW_END, to = "VIEW_END")
})
@TextAlignment
public int getTextAlignment() {
    return (mPrivateFlags2 & PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK) >>
            PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT;
}

/**
 * Resolve the text alignment.
 *
 * @return true if resolution has been done, false otherwise.
 *
 * @hide
 */
public boolean resolveTextAlignment() {
    // Reset any previous text alignment resolution
    mPrivateFlags2 &= ~(PFLAG2_TEXT_ALIGNMENT_RESOLVED | PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK);

    if (hasRtlSupport()) {
        // Set resolved text alignment flag depending on text alignment flag
        final int textAlignment = getRawTextAlignment();
        switch (textAlignment) {
            case TEXT_ALIGNMENT_INHERIT:
                // Check if we can resolve the text alignment
                if (!canResolveTextAlignment()) {
                    // We cannot do the resolution if there is no parent so use the default
                    mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
                    // Resolution will need to happen again later
                    return false;
                }

                // Parent has not yet resolved, so we still return the default
                try {
                    if (!mParent.isTextAlignmentResolved()) {
                        mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
                        // Resolution will need to happen again later
                        return false;
                    }
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                    mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED |
                            PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
                    return true;
                }

                int parentResolvedTextAlignment;
                try {
                    parentResolvedTextAlignment = mParent.getTextAlignment();
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                    parentResolvedTextAlignment = TEXT_ALIGNMENT_GRAVITY;
                }
                switch (parentResolvedTextAlignment) {
                    case TEXT_ALIGNMENT_GRAVITY:
                    case TEXT_ALIGNMENT_TEXT_START:
                    case TEXT_ALIGNMENT_TEXT_END:
                    case TEXT_ALIGNMENT_CENTER:
                    case TEXT_ALIGNMENT_VIEW_START:
                    case TEXT_ALIGNMENT_VIEW_END:
                        // Resolved text alignment is the same as the parent resolved
                        // text alignment
                        mPrivateFlags2 |=
                                (parentResolvedTextAlignment << PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT);
                        break;
                    default:
                        // Use default resolved text alignment
                        mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
                }
                break;
            case TEXT_ALIGNMENT_GRAVITY:
            case TEXT_ALIGNMENT_TEXT_START:
            case TEXT_ALIGNMENT_TEXT_END:
            case TEXT_ALIGNMENT_CENTER:
            case TEXT_ALIGNMENT_VIEW_START:
            case TEXT_ALIGNMENT_VIEW_END:
                // Resolved text alignment is the same as text alignment
                mPrivateFlags2 |= (textAlignment << PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK_SHIFT);
                break;
```

```
            default:
                // Use default resolved text alignment
                mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
        }
    } else {
        // Use default resolved text alignment
        mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
    }

    // Set the resolved
    mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED;
    return true;
}

/**
 * Check if text alignment resolution can be done.
 *
 * @return true if text alignment resolution can be done otherwise return false.
 */
public boolean canResolveTextAlignment() {
    switch (getRawTextAlignment()) {
        case TEXT_DIRECTION_INHERIT:
            if (mParent != null) {
                try {
                    return mParent.canResolveTextAlignment();
                } catch (AbstractMethodError e) {
                    Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                            " does not fully implement ViewParent", e);
                }
            }
            return false;

        default:
            return true;
    }
}

/**
 * Reset resolved text alignment. Text alignment will be resolved during a call to
 * {@link #onMeasure(int, int)}.
 *
 * @hide
 */
public void resetResolvedTextAlignment() {
    // Reset any previous text alignment resolution
    mPrivateFlags2 &= ~(PFLAG2_TEXT_ALIGNMENT_RESOLVED | PFLAG2_TEXT_ALIGNMENT_RESOLVED_MASK);
    // Set to default
    mPrivateFlags2 |= PFLAG2_TEXT_ALIGNMENT_RESOLVED_DEFAULT;
}

/**
 * @return true if text alignment is inherited.
 *
 * @hide
 */
public boolean isTextAlignmentInherited() {
    return (getRawTextAlignment() == TEXT_ALIGNMENT_INHERIT);
}

/**
 * @return true if text alignment is resolved.
 */
public boolean isTextAlignmentResolved() {
    return (mPrivateFlags2 & PFLAG2_TEXT_ALIGNMENT_RESOLVED) == PFLAG2_TEXT_ALIGNMENT_RESOLVED;
}

/**
 * Generate a value suitable for use in {@link #setId(int)}.
 * This value will not collide with ID values generated at build time by aapt for R.id.
 *
 * @return a generated ID value
 */
public static int generateViewId() {
    for (;;) {
        final int result = sNextGeneratedId.get();
        // aapt-generated IDs have the high byte nonzero; clamp to the range under that.
        int newValue = result + 1;
        if (newValue > 0x00FFFFFF) newValue = 1; // Roll over to 1, not 0.
        if (sNextGeneratedId.compareAndSet(result, newValue)) {
            return result;
        }
    }
```

```
}

private static boolean isViewIdGenerated(int id) {
    return (id & 0xFF000000) == 0 && (id & 0x00FFFFFF) != 0;
}

/**
 * Gets the Views in the hierarchy affected by entering and exiting Activity Scene transitions.
 * @param transitioningViews This View will be added to transitioningViews if it is VISIBLE and
 *                           a normal View or a ViewGroup with
 *                           {@link android.view.ViewGroup#isTransitionGroup()} true.
 * @hide
 */
public void captureTransitioningViews(List<View> transitioningViews) {
    if (getVisibility() == View.VISIBLE) {
        transitioningViews.add(this);
    }
}

/**
 * Adds all Views that have {@link #getTransitionName()} non-null to namedElements.
 * @param namedElements Will contain all Views in the hierarchy having a transitionName.
 * @hide
 */
public void findNamedViews(Map<String, View> namedElements) {
    if (getVisibility() == VISIBLE || mGhostView != null) {
        String transitionName = getTransitionName();
        if (transitionName != null) {
            namedElements.put(transitionName, this);
        }
    }
}

/**
 * Returns the pointer icon for the motion event, or null if it doesn't specify the icon.
 * The default implementation does not care the location or event types, but some subclasses
 * may use it (such as WebViews).
 * @param event The MotionEvent from a mouse
 * @param pointerIndex The index of the pointer for which to retrieve the {@link PointerIcon}.
 *                     This will be between 0 and {@link MotionEvent#getPointerCount()}.
 * @see PointerIcon
 */
public PointerIcon onResolvePointerIcon(MotionEvent event, int pointerIndex) {
    final float x = event.getX(pointerIndex);
    final float y = event.getY(pointerIndex);
    if (isDraggingScrollBar() || isOnScrollbarThumb(x, y)) {
        return PointerIcon.getSystemIcon(mContext, PointerIcon.TYPE_ARROW);
    }
    return mPointerIcon;
}

/**
 * Set the pointer icon for the current view.
 * Passing {@code null} will restore the pointer icon to its default value.
 * @param pointerIcon A PointerIcon instance which will be shown when the mouse hovers.
 */
public void setPointerIcon(PointerIcon pointerIcon) {
    mPointerIcon = pointerIcon;
    if (mAttachInfo == null || mAttachInfo.mHandlingPointerEvent) {
        return;
    }
    try {
        mAttachInfo.mSession.updatePointerIcon(mAttachInfo.mWindow);
    } catch (RemoteException e) {
    }
}

/**
 * Gets the pointer icon for the current view.
 */
public PointerIcon getPointerIcon() {
    return mPointerIcon;
}

/**
 * Checks pointer capture status.
 *
 * @return true if the view has pointer capture.
 * @see #requestPointerCapture()
 * @see #hasPointerCapture()
 */
public boolean hasPointerCapture() {
```

```
    final ViewRootImpl viewRootImpl = getViewRootImpl();
    if (viewRootImpl == null) {
        return false;
    }
    return viewRootImpl.hasPointerCapture();
}

/**
 * Requests pointer capture mode.
 * <p>
 * When the window has pointer capture, the mouse pointer icon will disappear and will not
 * change its position. Further mouse will be dispatched with the source
 * {@link InputDevice#SOURCE_MOUSE_RELATIVE}, and relative position changes will be available
 * through {@link MotionEvent#getX} and {@link MotionEvent#getY}. Non-mouse events
 * (touchscreens, or stylus) will not be affected.
 * <p>
 * If the window already has pointer capture, this call does nothing.
 * <p>
 * The capture may be released through {@link #releasePointerCapture()}, or will be lost
 * automatically when the window loses focus.
 *
 * @see #releasePointerCapture()
 * @see #hasPointerCapture()
 */
public void requestPointerCapture() {
    final ViewRootImpl viewRootImpl = getViewRootImpl();
    if (viewRootImpl != null) {
        viewRootImpl.requestPointerCapture(true);
    }
}


/**
 * Releases the pointer capture.
 * <p>
 * If the window does not have pointer capture, this call will do nothing.
 * @see #requestPointerCapture()
 * @see #hasPointerCapture()
 */
public void releasePointerCapture() {
    final ViewRootImpl viewRootImpl = getViewRootImpl();
    if (viewRootImpl != null) {
        viewRootImpl.requestPointerCapture(false);
    }
}

/**
 * Called when the window has just acquired or lost pointer capture.
 *
 * @param hasCapture True if the view now has pointerCapture, false otherwise.
 */
@CallSuper
public void onPointerCaptureChange(boolean hasCapture) {
}

/**
 * @see #onPointerCaptureChange
 */
public void dispatchPointerCaptureChanged(boolean hasCapture) {
    onPointerCaptureChange(hasCapture);
}

/**
 * Implement this method to handle captured pointer events
 *
 * @param event The captured pointer event.
 * @return True if the event was handled, false otherwise.
 * @see #requestPointerCapture()
 */
public boolean onCapturedPointerEvent(MotionEvent event) {
    return false;
}

/**
 * Interface definition for a callback to be invoked when a captured pointer event
 * is being dispatched this view. The callback will be invoked before the event is
 * given to the view.
 */
public interface OnCapturedPointerListener {
    /**
     * Called when a captured pointer event is dispatched to a view.
     * @param view The view this event has been dispatched to.
```

```
     * @param event The captured event.
     * @return True if the listener has consumed the event, false otherwise.
     */
    boolean onCapturedPointer(View view, MotionEvent event);
}

/**
 * Set a listener to receive callbacks when the pointer capture state of a view changes.
 * @param l  The {@link OnCapturedPointerListener} to receive callbacks.
 */
public void setOnCapturedPointerListener(OnCapturedPointerListener l) {
    getListenerInfo().mOnCapturedPointerListener = l;
}

// Properties
//
/**
 * A Property wrapper around the <code>alpha</code> functionality handled by the
 * {@link View#setAlpha(float)} and {@link View#getAlpha()} methods.
 */
public static final Property<View, Float> ALPHA = new FloatProperty<View>("alpha") {
    @Override
    public void setValue(View object, float value) {
        object.setAlpha(value);
    }

    @Override
    public Float get(View object) {
        return object.getAlpha();
    }
};

/**
 * A Property wrapper around the <code>translationX</code> functionality handled by the
 * {@link View#setTranslationX(float)} and {@link View#getTranslationX()} methods.
 */
public static final Property<View, Float> TRANSLATION_X = new FloatProperty<View>("translationX") {
    @Override
    public void setValue(View object, float value) {
        object.setTranslationX(value);
    }

            @Override
    public Float get(View object) {
        return object.getTranslationX();
    }
};

/**
 * A Property wrapper around the <code>translationY</code> functionality handled by the
 * {@link View#setTranslationY(float)} and {@link View#getTranslationY()} methods.
 */
public static final Property<View, Float> TRANSLATION_Y = new FloatProperty<View>("translationY") {
    @Override
    public void setValue(View object, float value) {
        object.setTranslationY(value);
    }

    @Override
    public Float get(View object) {
        return object.getTranslationY();
    }
};

/**
 * A Property wrapper around the <code>translationZ</code> functionality handled by the
 * {@link View#setTranslationZ(float)} and {@link View#getTranslationZ()} methods.
 */
public static final Property<View, Float> TRANSLATION_Z = new FloatProperty<View>("translationZ") {
    @Override
    public void setValue(View object, float value) {
        object.setTranslationZ(value);
    }

    @Override
    public Float get(View object) {
        return object.getTranslationZ();
    }
};

/**
 * A Property wrapper around the <code>x</code> functionality handled by the
```

```
     * {@link View#setX(float)} and {@link View#getX()} methods.
     */
    public static final Property<View, Float> X = new FloatProperty<View>("x") {
        @Override
        public void setValue(View object, float value) {
            object.setX(value);
        }

        @Override
        public Float get(View object) {
            return object.getX();
        }
    };

    /**
     * A Property wrapper around the <code>y</code> functionality handled by the
     * {@link View#setY(float)} and {@link View#getY()} methods.
     */
    public static final Property<View, Float> Y = new FloatProperty<View>("y") {
        @Override
        public void setValue(View object, float value) {
            object.setY(value);
        }

        @Override
        public Float get(View object) {
            return object.getY();
        }
    };

    /**
     * A Property wrapper around the <code>z</code> functionality handled by the
     * {@link View#setZ(float)} and {@link View#getZ()} methods.
     */
    public static final Property<View, Float> Z = new FloatProperty<View>("z") {
        @Override
        public void setValue(View object, float value) {
            object.setZ(value);
        }

        @Override
        public Float get(View object) {
            return object.getZ();
        }
    };

    /**
     * A Property wrapper around the <code>rotation</code> functionality handled by the
     * {@link View#setRotation(float)} and {@link View#getRotation()} methods.
     */
    public static final Property<View, Float> ROTATION = new FloatProperty<View>("rotation") {
        @Override
        public void setValue(View object, float value) {
            object.setRotation(value);
        }

        @Override
        public Float get(View object) {
            return object.getRotation();
        }
    };

    /**
     * A Property wrapper around the <code>rotationX</code> functionality handled by the
     * {@link View#setRotationX(float)} and {@link View#getRotationX()} methods.
     */
    public static final Property<View, Float> ROTATION_X = new FloatProperty<View>("rotationX") {
        @Override
        public void setValue(View object, float value) {
            object.setRotationX(value);
        }

        @Override
        public Float get(View object) {
            return object.getRotationX();
        }
    };

    /**
     * A Property wrapper around the <code>rotationY</code> functionality handled by the
     * {@link View#setRotationY(float)} and {@link View#getRotationY()} methods.
     */
```

```
public static final Property<View, Float> ROTATION_Y = new FloatProperty<View>("rotationY") {
    @Override
    public void setValue(View object, float value) {
        object.setRotationY(value);
    }

    @Override
    public Float get(View object) {
        return object.getRotationY();
    }
};

/**
 * A Property wrapper around the <code>scaleX</code> functionality handled by the
 * {@link View#setScaleX(float)} and {@link View#getScaleX()} methods.
 */
public static final Property<View, Float> SCALE_X = new FloatProperty<View>("scaleX") {
    @Override
    public void setValue(View object, float value) {
        object.setScaleX(value);
    }

    @Override
    public Float get(View object) {
        return object.getScaleX();
    }
};

/**
 * A Property wrapper around the <code>scaleY</code> functionality handled by the
 * {@link View#setScaleY(float)} and {@link View#getScaleY()} methods.
 */
public static final Property<View, Float> SCALE_Y = new FloatProperty<View>("scaleY") {
    @Override
    public void setValue(View object, float value) {
        object.setScaleY(value);
    }

    @Override
    public Float get(View object) {
        return object.getScaleY();
    }
};

/**
 * A MeasureSpec encapsulates the layout requirements passed from parent to child.
 * Each MeasureSpec represents a requirement for either the width or the height.
 * A MeasureSpec is comprised of a size and a mode. There are three possible
 * modes:
 * <dl>
 * <dt>UNSPECIFIED</dt>
 * <dd>
 * The parent has not imposed any constraint on the child. It can be whatever size
 * it wants.
 * </dd>
 *
 * <dt>EXACTLY</dt>
 * <dd>
 * The parent has determined an exact size for the child. The child is going to be
 * given those bounds regardless of how big it wants to be.
 * </dd>
 *
 * <dt>AT_MOST</dt>
 * <dd>
 * The child can be as large as it wants up to the specified size.
 * </dd>
 * </dl>
 *
 * MeasureSpecs are implemented as ints to reduce object allocation. This class
 * is provided to pack and unpack the &lt;size, mode&gt; tuple into the int.
 */
public static class MeasureSpec {
    private static final int MODE_SHIFT = 30;
    private static final int MODE_MASK  = 0x3 << MODE_SHIFT;

    /** @hide */
    @IntDef({UNSPECIFIED, EXACTLY, AT_MOST})
    @Retention(RetentionPolicy.SOURCE)
    public @interface MeasureSpecMode {}

    /**
     * Measure specification mode: The parent has not imposed any constraint
```

```
     * on the child. It can be whatever size it wants.
     */
    public static final int UNSPECIFIED = 0 << MODE_SHIFT;

    /**
     * Measure specification mode: The parent has determined an exact size
     * for the child. The child is going to be given those bounds regardless
     * of how big it wants to be.
     */
    public static final int EXACTLY     = 1 << MODE_SHIFT;

    /**
     * Measure specification mode: The child can be as large as it wants up
     * to the specified size.
     */
    public static final int AT_MOST     = 2 << MODE_SHIFT;

    /**
     * Creates a measure specification based on the supplied size and mode.
     *
     * The mode must always be one of the following:
     * <ul>
     *  <li>{@link android.view.View.MeasureSpec#UNSPECIFIED}</li>
     *  <li>{@link android.view.View.MeasureSpec#EXACTLY}</li>
     *  <li>{@link android.view.View.MeasureSpec#AT_MOST}</li>
     * </ul>
     *
     * <p><strong>Note:</strong> On API level 17 and lower, makeMeasureSpec's
     * implementation was such that the order of arguments did not matter
     * and overflow in either value could impact the resulting MeasureSpec.
     * {@link android.widget.RelativeLayout} was affected by this bug.
     * Apps targeting API levels greater than 17 will get the fixed, more strict
     * behavior.</p>
     *
     * @param size the size of the measure specification
     * @param mode the mode of the measure specification
     * @return the measure specification based on size and mode
     */
    public static int makeMeasureSpec(@IntRange(from = 0, to = (1 << MeasureSpec.MODE_SHIFT) - 1) int size,
                                      @MeasureSpecMode int mode) {
        if (sUseBrokenMakeMeasureSpec) {
            return size + mode;
        } else {
            return (size & ~MODE_MASK) | (mode & MODE_MASK);
        }
    }

    /**
     * Like {@link #makeMeasureSpec(int, int)}, but any spec with a mode of UNSPECIFIED
     * will automatically get a size of 0. Older apps expect this.
     *
     * @hide internal use only for compatibility with system widgets and older apps
     */
    public static int makeSafeMeasureSpec(int size, int mode) {
        if (sUseZeroUnspecifiedMeasureSpec && mode == UNSPECIFIED) {
            return 0;
        }
        return makeMeasureSpec(size, mode);
    }

    /**
     * Extracts the mode from the supplied measure specification.
     *
     * @param measureSpec the measure specification to extract the mode from
     * @return {@link android.view.View.MeasureSpec#UNSPECIFIED},
     *         {@link android.view.View.MeasureSpec#AT_MOST} or
     *         {@link android.view.View.MeasureSpec#EXACTLY}
     */
    @MeasureSpecMode
    public static int getMode(int measureSpec) {
        //noinspection ResourceType
        return (measureSpec & MODE_MASK);
    }

    /**
     * Extracts the size from the supplied measure specification.
     *
     * @param measureSpec the measure specification to extract the size from
     * @return the size in pixels defined in the supplied measure specification
     */
    public static int getSize(int measureSpec) {
        return (measureSpec & ~MODE_MASK);
```

```
    }

    static int adjust(int measureSpec, int delta) {
        final int mode = getMode(measureSpec);
        int size = getSize(measureSpec);
        if (mode == UNSPECIFIED) {
            // No need to adjust size for UNSPECIFIED mode.
            return makeMeasureSpec(size, UNSPECIFIED);
        }
        size += delta;
        if (size < 0) {
            Log.e(VIEW_LOG_TAG, "MeasureSpec.adjust: new size would be negative! (" + size +
                    ") spec: " + toString(measureSpec) + " delta: " + delta);
            size = 0;
        }
        return makeMeasureSpec(size, mode);
    }

    /**
     * Returns a String representation of the specified measure
     * specification.
     *
     * @param measureSpec the measure specification to convert to a String
     * @return a String with the following format: "MeasureSpec: MODE SIZE"
     */
    public static String toString(int measureSpec) {
        int mode = getMode(measureSpec);
        int size = getSize(measureSpec);

        StringBuilder sb = new StringBuilder("MeasureSpec: ");

        if (mode == UNSPECIFIED)
            sb.append("UNSPECIFIED ");
        else if (mode == EXACTLY)
            sb.append("EXACTLY ");
        else if (mode == AT_MOST)
            sb.append("AT_MOST ");
        else
            sb.append(mode).append(" ");

        sb.append(size);
        return sb.toString();
    }
}

private final class CheckForLongPress implements Runnable {
    private int mOriginalWindowAttachCount;
    private float mX;
    private float mY;
    private boolean mOriginalPressedState;

    @Override
    public void run() {
        if ((mOriginalPressedState == isPressed()) && (mParent != null)
                && mOriginalWindowAttachCount == mWindowAttachCount) {
            if (performLongClick(mX, mY)) {
                mHasPerformedLongPress = true;
            }
        }
    }

    public void setAnchor(float x, float y) {
        mX = x;
        mY = y;
    }

    public void rememberWindowAttachCount() {
        mOriginalWindowAttachCount = mWindowAttachCount;
    }

    public void rememberPressedState() {
        mOriginalPressedState = isPressed();
    }
}

private final class CheckForTap implements Runnable {
    public float x;
    public float y;

    @Override
    public void run() {
        mPrivateFlags &= ~PFLAG_PREPRESSED;
```

```
            setPressed(true, x, y);
            checkForLongClick(ViewConfiguration.getTapTimeout(), x, y);
        }
    }

    private final class PerformClick implements Runnable {
        @Override
        public void run() {
            performClick();
        }
    }

    /**
     * This method returns a ViewPropertyAnimator object, which can be used to animate
     * specific properties on this View.
     *
     * @return ViewPropertyAnimator The ViewPropertyAnimator associated with this View.
     */
    public ViewPropertyAnimator animate() {
        if (mAnimator == null) {
            mAnimator = new ViewPropertyAnimator(this);
        }
        return mAnimator;
    }

    /**
     * Sets the name of the View to be used to identify Views in Transitions.
     * Names should be unique in the View hierarchy.
     *
     * @param transitionName The name of the View to uniquely identify it for Transitions.
     */
    public final void setTransitionName(String transitionName) {
        mTransitionName = transitionName;
    }

    /**
     * Returns the name of the View to be used to identify Views in Transitions.
     * Names should be unique in the View hierarchy.
     *
     * <p>This returns null if the View has not been given a name.</p>
     *
     * @return The name used of the View to be used to identify Views in Transitions or null
     * if no name has been given.
     */
    @ViewDebug.ExportedProperty
    public String getTransitionName() {
        return mTransitionName;
    }

    /**
     * @hide
     */
    public void requestKeyboardShortcuts(List<KeyboardShortcutGroup> data, int deviceId) {
        // Do nothing.
    }

    /**
     * Interface definition for a callback to be invoked when a hardware key event is
     * dispatched to this view. The callback will be invoked before the key event is
     * given to the view. This is only useful for hardware keyboards; a software input
     * method has no obligation to trigger this listener.
     */
    public interface OnKeyListener {
        /**
         * Called when a hardware key is dispatched to a view. This allows listeners to
         * get a chance to respond before the target view.
         * <p>Key presses in software keyboards will generally NOT trigger this method,
         * although some may elect to do so in some situations. Do not assume a
         * software input method has to be key-based; even if it is, it may use key presses
         * in a different way than you expect, so there is no way to reliably catch soft
         * input key presses.
         *
         * @param v The view the key has been dispatched to.
         * @param keyCode The code for the physical key that was pressed
         * @param event The KeyEvent object containing full information about
         *        the event.
         * @return True if the listener has consumed the event, false otherwise.
         */
        boolean onKey(View v, int keyCode, KeyEvent event);
    }

    /**
```

```
 * Interface definition for a callback to be invoked when a touch event is
 * dispatched to this view. The callback will be invoked before the touch
 * event is given to the view.
 */
public interface OnTouchListener {
    /**
     * Called when a touch event is dispatched to a view. This allows listeners to
     * get a chance to respond before the target view.
     *
     * @param v The view the touch event has been dispatched to.
     * @param event The MotionEvent object containing full information about
     *        the event.
     * @return True if the listener has consumed the event, false otherwise.
     */
    boolean onTouch(View v, MotionEvent event);
}

/**
 * Interface definition for a callback to be invoked when a hover event is
 * dispatched to this view. The callback will be invoked before the hover
 * event is given to the view.
 */
public interface OnHoverListener {
    /**
     * Called when a hover event is dispatched to a view. This allows listeners to
     * get a chance to respond before the target view.
     *
     * @param v The view the hover event has been dispatched to.
     * @param event The MotionEvent object containing full information about
     *        the event.
     * @return True if the listener has consumed the event, false otherwise.
     */
    boolean onHover(View v, MotionEvent event);
}

/**
 * Interface definition for a callback to be invoked when a generic motion event is
 * dispatched to this view. The callback will be invoked before the generic motion
 * event is given to the view.
 */
public interface OnGenericMotionListener {
    /**
     * Called when a generic motion event is dispatched to a view. This allows listeners to
     * get a chance to respond before the target view.
     *
     * @param v The view the generic motion event has been dispatched to.
     * @param event The MotionEvent object containing full information about
     *        the event.
     * @return True if the listener has consumed the event, false otherwise.
     */
    boolean onGenericMotion(View v, MotionEvent event);
}

/**
 * Interface definition for a callback to be invoked when a view has been clicked and held.
 */
public interface OnLongClickListener {
    /**
     * Called when a view has been clicked and held.
     *
     * @param v The view that was clicked and held.
     *
     * @return true if the callback consumed the long click, false otherwise.
     */
    boolean onLongClick(View v);
}

/**
 * Interface definition for a callback to be invoked when a drag is being dispatched
 * to this view.  The callback will be invoked before the hosting view's own
 * onDrag(event) method.  If the listener wants to fall back to the hosting view's
 * onDrag(event) behavior, it should return 'false' from this callback.
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For a guide to implementing drag and drop features, read the
 * <a href="{@docRoot}guide/topics/ui/drag-drop.html">Drag and Drop</a> developer guide.</p>
 * </div>
 */
public interface OnDragListener {
    /**
     * Called when a drag event is dispatched to a view. This allows listeners
```

```
     * to get a chance to override base View behavior.
     *
     * @param v The View that received the drag event.
     * @param event The {@link android.view.DragEvent} object for the drag event.
     * @return {@code true} if the drag event was handled successfully, or {@code false}
     * if the drag event was not handled. Note that {@code false} will trigger the View
     * to call its {@link #onDragEvent(DragEvent) onDragEvent()} handler.
     */
    boolean onDrag(View v, DragEvent event);
}

/**
 * Interface definition for a callback to be invoked when the focus state of
 * a view changed.
 */
public interface OnFocusChangeListener {
    /**
     * Called when the focus state of a view has changed.
     *
     * @param v The view whose state has changed.
     * @param hasFocus The new focus state of v.
     */
    void onFocusChange(View v, boolean hasFocus);
}

/**
 * Interface definition for a callback to be invoked when a view is clicked.
 */
public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}

/**
 * Interface definition for a callback to be invoked when a view is context clicked.
 */
public interface OnContextClickListener {
    /**
     * Called when a view is context clicked.
     *
     * @param v The view that has been context clicked.
     * @return true if the callback consumed the context click, false otherwise.
     */
    boolean onContextClick(View v);
}

/**
 * Interface definition for a callback to be invoked when the context menu
 * for this view is being built.
 */
public interface OnCreateContextMenuListener {
    /**
     * Called when the context menu for this view is being built. It is not
     * safe to hold onto the menu after this method returns.
     *
     * @param menu The context menu that is being built
     * @param v The view for which the context menu is being built
     * @param menuInfo Extra information about the item for which the
     *            context menu should be shown. This information will vary
     *            depending on the class of v.
     */
    void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo);
}

/**
 * Interface definition for a callback to be invoked when the status bar changes
 * visibility.  This reports <strong>global</strong> changes to the system UI
 * state, not what the application is requesting.
 *
 * @see View#setOnSystemUiVisibilityChangeListener(android.view.View.OnSystemUiVisibilityChangeListener)
 */
public interface OnSystemUiVisibilityChangeListener {
    /**
     * Called when the status bar changes visibility because of a call to
     * {@link View#setSystemUiVisibility(int)}.
     *
     * @param visibility  Bitwise-or of flags {@link #SYSTEM_UI_FLAG_LOW_PROFILE},
     * {@link #SYSTEM_UI_FLAG_HIDE_NAVIGATION}, and {@link #SYSTEM_UI_FLAG_FULLSCREEN}.
```

```
         * This tells you the <strong>global</strong> state of these UI visibility
         * flags, not what your app is currently applying.
         */
        public void onSystemUiVisibilityChange(int visibility);
    }

    /**
     * Interface definition for a callback to be invoked when this view is attached
     * or detached from its window.
     */
    public interface OnAttachStateChangeListener {
        /**
         * Called when the view is attached to a window.
         * @param v The view that was attached
         */
        public void onViewAttachedToWindow(View v);
        /**
         * Called when the view is detached from a window.
         * @param v The view that was detached
         */
        public void onViewDetachedFromWindow(View v);
    }

    /**
     * Listener for applying window insets on a view in a custom way.
     *
     * <p>Apps may choose to implement this interface if they want to apply custom policy
     * to the way that window insets are treated for a view. If an OnApplyWindowInsetsListener
     * is set, its
     * {@link OnApplyWindowInsetsListener#onApplyWindowInsets(View, WindowInsets) onApplyWindowInsets}
     * method will be called instead of the View's own
     * {@link #onApplyWindowInsets(WindowInsets) onApplyWindowInsets} method. The listener
     * may optionally call the parameter View's <code>onApplyWindowInsets</code> method to apply
     * the View's normal behavior as part of its own.</p>
     */
    public interface OnApplyWindowInsetsListener {
        /**
         * When {@link View#setOnApplyWindowInsetsListener(View.OnApplyWindowInsetsListener) set}
         * on a View, this listener method will be called instead of the view's own
         * {@link View#onApplyWindowInsets(WindowInsets) onApplyWindowInsets} method.
         *
         * @param v The view applying window insets
         * @param insets The insets to apply
         * @return The insets supplied, minus any insets that were consumed
         */
        public WindowInsets onApplyWindowInsets(View v, WindowInsets insets);
    }

    private final class UnsetPressedState implements Runnable {
        @Override
        public void run() {
            setPressed(false);
        }
    }

    /**
     * When a view becomes invisible checks if autofill considers the view invisible too. This
     * happens after the regular removal operation to make sure the operation is finished by the
     * time this is called.
     */
    private static class VisibilityChangeForAutofillHandler extends Handler {
        private final AutofillManager mAfm;
        private final View mView;

        private VisibilityChangeForAutofillHandler(@NonNull AutofillManager afm,
                @NonNull View view) {
            mAfm = afm;
            mView = view;
        }

        @Override
        public void handleMessage(Message msg) {
            mAfm.notifyViewVisibilityChanged(mView, mView.isShown());
        }
    }

    /**
     * Base class for derived classes that want to save and restore their own
     * state in {@link android.view.View#onSaveInstanceState()}.
     */
    public static class BaseSavedState extends AbsSavedState {
        static final int START_ACTIVITY_REQUESTED_WHO_SAVED = 0b1;
```

```
    static final int IS_AUTOFILLED = 0b10;
    static final int AUTOFILL_ID = 0b100;

    // Flags that describe what data in this state is valid
    int mSavedData;
    String mStartActivityRequestWhoSaved;
    boolean mIsAutofilled;
    int mAutofillViewId;

    /**
     * Constructor used when reading from a parcel. Reads the state of the superclass.
     *
     * @param source parcel to read from
     */
    public BaseSavedState(Parcel source) {
        this(source, null);
    }

    /**
     * Constructor used when reading from a parcel using a given class loader.
     * Reads the state of the superclass.
     *
     * @param source parcel to read from
     * @param loader ClassLoader to use for reading
     */
    public BaseSavedState(Parcel source, ClassLoader loader) {
        super(source, loader);
        mSavedData = source.readInt();
        mStartActivityRequestWhoSaved = source.readString();
        mIsAutofilled = source.readBoolean();
        mAutofillViewId = source.readInt();
    }

    /**
     * Constructor called by derived classes when creating their SavedState objects
     *
     * @param superState The state of the superclass of this view
     */
    public BaseSavedState(Parcelable superState) {
        super(superState);
    }

    @Override
    public void writeToParcel(Parcel out, int flags) {
        super.writeToParcel(out, flags);

        out.writeInt(mSavedData);
        out.writeString(mStartActivityRequestWhoSaved);
        out.writeBoolean(mIsAutofilled);
        out.writeInt(mAutofillViewId);
    }

    public static final Parcelable.Creator<BaseSavedState> CREATOR
            = new Parcelable.ClassLoaderCreator<BaseSavedState>() {
        @Override
        public BaseSavedState createFromParcel(Parcel in) {
            return new BaseSavedState(in);
        }

        @Override
        public BaseSavedState createFromParcel(Parcel in, ClassLoader loader) {
            return new BaseSavedState(in, loader);
        }

        @Override
        public BaseSavedState[] newArray(int size) {
            return new BaseSavedState[size];
        }
    };
}

/**
 * A set of information given to a view when it is attached to its parent
 * window.
 */
final static class AttachInfo {
    interface Callbacks {
        void playSoundEffect(int effectId);
        boolean performHapticFeedback(int effectId, boolean always);
    }

    /**
```

```
     * InvalidateInfo is used to post invalidate(int, int, int, int) messages
     * to a Handler. This class contains the target (View) to invalidate and
     * the coordinates of the dirty rectangle.
     *
     * For performance purposes, this class also implements a pool of up to
     * POOL_LIMIT objects that get reused. This reduces memory allocations
     * whenever possible.
     */
    static class InvalidateInfo {
        private static final int POOL_LIMIT = 10;

        private static final SynchronizedPool<InvalidateInfo> sPool =
                new SynchronizedPool<InvalidateInfo>(POOL_LIMIT);

        View target;

        int left;
        int top;
        int right;
        int bottom;

        public static InvalidateInfo obtain() {
            InvalidateInfo instance = sPool.acquire();
            return (instance != null) ? instance : new InvalidateInfo();
        }

        public void recycle() {
            target = null;
            sPool.release(this);
        }
    }

    final IWindowSession mSession;

    final IWindow mWindow;

    final IBinder mWindowToken;

    Display mDisplay;

    final Callbacks mRootCallbacks;

    IWindowId mIWindowId;
    WindowId mWindowId;

    /**
     * The top view of the hierarchy.
     */
    View mRootView;

    IBinder mPanelParentWindowToken;

    boolean mHardwareAccelerated;
    boolean mHardwareAccelerationRequested;
    ThreadedRenderer mThreadedRenderer;
    List<RenderNode> mPendingAnimatingRenderNodes;

    /**
     * The state of the display to which the window is attached, as reported
     * by {@link Display#getState()}.  Note that the display state constants
     * declared by {@link Display} do not exactly line up with the screen state
     * constants declared by {@link View} (there are more display states than
     * screen states).
     */
    int mDisplayState = Display.STATE_UNKNOWN;

    /**
     * Scale factor used by the compatibility mode
     */
    float mApplicationScale;

    /**
     * Indicates whether the application is in compatibility mode
     */
    boolean mScalingRequired;

    /**
     * Left position of this view's window
     */
    int mWindowLeft;

    /**
```

```
     * Top position of this view's window
     */
    int mWindowTop;

    /**
     * Indicates whether views need to use 32-bit drawing caches
     */
    boolean mUse32BitDrawingCache;

    /**
     * For windows that are full-screen but using insets to layout inside
     * of the screen areas, these are the current insets to appear inside
     * the overscan area of the display.
     */
    final Rect mOverscanInsets = new Rect();

    /**
     * For windows that are full-screen but using insets to layout inside
     * of the screen decorations, these are the current insets for the
     * content of the window.
     */
    final Rect mContentInsets = new Rect();

    /**
     * For windows that are full-screen but using insets to layout inside
     * of the screen decorations, these are the current insets for the
     * actual visible parts of the window.
     */
    final Rect mVisibleInsets = new Rect();

    /**
     * For windows that are full-screen but using insets to layout inside
     * of the screen decorations, these are the current insets for the
     * stable system windows.
     */
    final Rect mStableInsets = new Rect();

    /**
     * For windows that include areas that are not covered by real surface these are the outsets
     * for real surface.
     */
    final Rect mOutsets = new Rect();

    /**
     * In multi-window we force show the navigation bar. Because we don't want that the surface
     * size changes in this mode, we instead have a flag whether the navigation bar size should
     * always be consumed, so the app is treated like there is no virtual navigation bar at all.
     */
    boolean mAlwaysConsumeNavBar;

    /**
     * The internal insets given by this window.  This value is
     * supplied by the client (through
     * {@link ViewTreeObserver.OnComputeInternalInsetsListener}) and will
     * be given to the window manager when changed to be used in laying
     * out windows behind it.
     */
    final ViewTreeObserver.InternalInsetsInfo mGivenInternalInsets
            = new ViewTreeObserver.InternalInsetsInfo();

    /**
     * Set to true when mGivenInternalInsets is non-empty.
     */
    boolean mHasNonEmptyGivenInternalInsets;

    /**
     * All views in the window's hierarchy that serve as scroll containers,
     * used to determine if the window can be resized or must be panned
     * to adjust for a soft input area.
     */
    final ArrayList<View> mScrollContainers = new ArrayList<View>();

    final KeyEvent.DispatcherState mKeyDispatchState
            = new KeyEvent.DispatcherState();

    /**
     * Indicates whether the view's window currently has the focus.
     */
    boolean mHasWindowFocus;

    /**
     * The current visibility of the window.
```

```
     */
    int mWindowVisibility;

    /**
     * Indicates the time at which drawing started to occur.
     */
    long mDrawingTime;

    /**
     * Indicates whether or not ignoring the DIRTY_MASK flags.
     */
    boolean mIgnoreDirtyState;

    /**
     * This flag tracks when the mIgnoreDirtyState flag is set during draw(),
     * to avoid clearing that flag prematurely.
     */
    boolean mSetIgnoreDirtyState = false;

    /**
     * Indicates whether the view's window is currently in touch mode.
     */
    boolean mInTouchMode;

    /**
     * Indicates whether the view has requested unbuffered input dispatching for the current
     * event stream.
     */
    boolean mUnbufferedDispatchRequested;

    /**
     * Indicates that ViewAncestor should trigger a global layout change
     * the next time it performs a traversal
     */
    boolean mRecomputeGlobalAttributes;

    /**
     * Always report new attributes at next traversal.
     */
    boolean mForceReportNewAttributes;

    /**
     * Set during a traveral if any views want to keep the screen on.
     */
    boolean mKeepScreenOn;

    /**
     * Set during a traveral if the light center needs to be updated.
     */
    boolean mNeedsUpdateLightCenter;

    /**
     * Bitwise-or of all of the values that views have passed to setSystemUiVisibility().
     */
    int mSystemUiVisibility;

    /**
     * Hack to force certain system UI visibility flags to be cleared.
     */
    int mDisabledSystemUiVisibility;

    /**
     * Last global system UI visibility reported by the window manager.
     */
    int mGlobalSystemUiVisibility = -1;

    /**
     * True if a view in this hierarchy has an OnSystemUiVisibilityChangeListener
     * attached.
     */
    boolean mHasSystemUiListeners;

    /**
     * Set if the window has requested to extend into the overscan region
     * via WindowManager.LayoutParams.FLAG_LAYOUT_IN_OVERSCAN.
     */
    boolean mOverscanRequested;

    /**
     * Set if the visibility of any views has changed.
     */
    boolean mViewVisibilityChanged;
```

```java
/**
 * Set to true if a view has been scrolled.
 */
boolean mViewScrollChanged;

/**
 * Set to true if high contrast mode enabled
 */
boolean mHighContrastText;

/**
 * Set to true if a pointer event is currently being handled.
 */
boolean mHandlingPointerEvent;

/**
 * Global to the view hierarchy used as a temporary for dealing with
 * x/y points in the transparent region computations.
 */
final int[] mTransparentLocation = new int[2];

/**
 * Global to the view hierarchy used as a temporary for dealing with
 * x/y points in the ViewGroup.invalidateChild implementation.
 */
final int[] mInvalidateChildLocation = new int[2];

/**
 * Global to the view hierarchy used as a temporary for dealing with
 * computing absolute on-screen location.
 */
final int[] mTmpLocation = new int[2];

/**
 * Global to the view hierarchy used as a temporary for dealing with
 * x/y location when view is transformed.
 */
final float[] mTmpTransformLocation = new float[2];

/**
 * The view tree observer used to dispatch global events like
 * layout, pre-draw, touch mode change, etc.
 */
final ViewTreeObserver mTreeObserver;

/**
 * A Canvas used by the view hierarchy to perform bitmap caching.
 */
Canvas mCanvas;

/**
 * The view root impl.
 */
final ViewRootImpl mViewRootImpl;

/**
 * A Handler supplied by a view's {@link android.view.ViewRootImpl}. This
 * handler can be used to pump events in the UI events queue.
 */
final Handler mHandler;

/**
 * Temporary for use in computing invalidate rectangles while
 * calling up the hierarchy.
 */
final Rect mTmpInvalRect = new Rect();

/**
 * Temporary for use in computing hit areas with transformed views
 */
final RectF mTmpTransformRect = new RectF();

/**
 * Temporary for use in computing hit areas with transformed views
 */
final RectF mTmpTransformRect1 = new RectF();

/**
 * Temporary list of rectanges.
 */
final List<RectF> mTmpRectList = new ArrayList<>();
```

```java
/**
 * Temporary for use in transforming invalidation rect
 */
final Matrix mTmpMatrix = new Matrix();

/**
 * Temporary for use in transforming invalidation rect
 */
final Transformation mTmpTransformation = new Transformation();

/**
 * Temporary for use in querying outlines from OutlineProviders
 */
final Outline mTmpOutline = new Outline();

/**
 * Temporary list for use in collecting focusable descendents of a view.
 */
final ArrayList<View> mTempArrayList = new ArrayList<View>(24);

/**
 * The id of the window for accessibility purposes.
 */
int mAccessibilityWindowId = AccessibilityWindowInfo.UNDEFINED_WINDOW_ID;

/**
 * Flags related to accessibility processing.
 *
 * @see AccessibilityNodeInfo#FLAG_INCLUDE_NOT_IMPORTANT_VIEWS
 * @see AccessibilityNodeInfo#FLAG_REPORT_VIEW_IDS
 */
int mAccessibilityFetchFlags;

/**
 * The drawable for highlighting accessibility focus.
 */
Drawable mAccessibilityFocusDrawable;

/**
 * The drawable for highlighting autofilled views.
 *
 * @see #isAutofilled()
 */
Drawable mAutofilledDrawable;

/**
 * Show where the margins, bounds and layout bounds are for each view.
 */
boolean mDebugLayout = SystemProperties.getBoolean(DEBUG_LAYOUT_PROPERTY, false);

/**
 * Point used to compute visible regions.
 */
final Point mPoint = new Point();

/**
 * Used to track which View originated a requestLayout() call, used when
 * requestLayout() is called during layout.
 */
View mViewRequestingLayout;

/**
 * Used to track views that need (at least) a partial relayout at their current size
 * during the next traversal.
 */
List<View> mPartialLayoutViews = new ArrayList<>();

/**
 * Swapped with mPartialLayoutViews during layout to avoid concurrent
 * modification. Lazily assigned during ViewRootImpl layout.
 */
List<View> mEmptyPartialLayoutViews;

/**
 * Used to track the identity of the current drag operation.
 */
IBinder mDragToken;

/**
 * The drag shadow surface for the current drag operation.
 */
```

```
    public Surface mDragSurface;


    /**
     * The view that currently has a tooltip displayed.
     */
    View mTooltipHost;

    /**
     * Creates a new set of attachment information with the specified
     * events handler and thread.
     *
     * @param handler the events handler the view must use
     */
    AttachInfo(IWindowSession session, IWindow window, Display display,
            ViewRootImpl viewRootImpl, Handler handler, Callbacks effectPlayer,
            Context context) {
        mSession = session;
        mWindow = window;
        mWindowToken = window.asBinder();
        mDisplay = display;
        mViewRootImpl = viewRootImpl;
        mHandler = handler;
        mRootCallbacks = effectPlayer;
        mTreeObserver = new ViewTreeObserver(context);
    }
}

/**
 * <p>ScrollabilityCache holds various fields used by a View when scrolling
 * is supported. This avoids keeping too many unused fields in most
 * instances of View.</p>
 */
private static class ScrollabilityCache implements Runnable {

    /**
     * Scrollbars are not visible
     */
    public static final int OFF = 0;

    /**
     * Scrollbars are visible
     */
    public static final int ON = 1;

    /**
     * Scrollbars are fading away
     */
    public static final int FADING = 2;

    public boolean fadeScrollBars;

    public int fadingEdgeLength;
    public int scrollBarDefaultDelayBeforeFade;
    public int scrollBarFadeDuration;

    public int scrollBarSize;
    public int scrollBarMinTouchTarget;
    public ScrollBarDrawable scrollBar;
    public float[] interpolatorValues;
    public View host;

    public final Paint paint;
    public final Matrix matrix;
    public Shader shader;

    public final Interpolator scrollBarInterpolator = new Interpolator(1, 2);

    private static final float[] OPAQUE = { 255 };
    private static final float[] TRANSPARENT = { 0.0f };

    /**
     * When fading should start. This time moves into the future every time
     * a new scroll happens. Measured based on SystemClock.uptimeMillis()
     */
    public long fadeStartTime;


    /**
     * The current state of the scrollbars: ON, OFF, or FADING
     */
    public int state = OFF;
```

```
    private int mLastColor;

    public final Rect mScrollBarBounds = new Rect();
    public final Rect mScrollBarTouchBounds = new Rect();

    public static final int NOT_DRAGGING = 0;
    public static final int DRAGGING_VERTICAL_SCROLL_BAR = 1;
    public static final int DRAGGING_HORIZONTAL_SCROLL_BAR = 2;
    public int mScrollBarDraggingState = NOT_DRAGGING;

    public float mScrollBarDraggingPos = 0;

    public ScrollabilityCache(ViewConfiguration configuration, View host) {
        fadingEdgeLength = configuration.getScaledFadingEdgeLength();
        scrollBarSize = configuration.getScaledScrollBarSize();
        scrollBarMinTouchTarget = configuration.getScaledMinScrollbarTouchTarget();
        scrollBarDefaultDelayBeforeFade = ViewConfiguration.getScrollDefaultDelay();
        scrollBarFadeDuration = ViewConfiguration.getScrollBarFadeDuration();

        paint = new Paint();
        matrix = new Matrix();
        // use use a height of 1, and then wack the matrix each time we
        // actually use it.
        shader = new LinearGradient(0, 0, 0, 1, 0xFF000000, 0, Shader.TileMode.CLAMP);
        paint.setShader(shader);
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.DST_OUT));

        this.host = host;
    }

    public void setFadeColor(int color) {
        if (color != mLastColor) {
            mLastColor = color;

            if (color != 0) {
                shader = new LinearGradient(0, 0, 0, 1, color | 0xFF000000,
                        color & 0x00FFFFFF, Shader.TileMode.CLAMP);
                paint.setShader(shader);
                // Restore the default transfer mode (src_over)
                paint.setXfermode(null);
            } else {
                shader = new LinearGradient(0, 0, 0, 1, 0xFF000000, 0, Shader.TileMode.CLAMP);
                paint.setShader(shader);
                paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.DST_OUT));
            }
        }
    }

    public void run() {
        long now = AnimationUtils.currentAnimationTimeMillis();
        if (now >= fadeStartTime) {

            // the animation fades the scrollbars out by changing
            // the opacity (alpha) from fully opaque to fully
            // transparent
            int nextFrame = (int) now;
            int framesCount = 0;

            Interpolator interpolator = scrollBarInterpolator;

            // Start opaque
            interpolator.setKeyFrame(framesCount++, nextFrame, OPAQUE);

            // End transparent
            nextFrame += scrollBarFadeDuration;
            interpolator.setKeyFrame(framesCount, nextFrame, TRANSPARENT);

            state = FADING;

            // Kick off the fade animation
            host.invalidate(true);
        }
    }
}

/**
 * Resuable callback for sending
 * {@link AccessibilityEvent#TYPE_VIEW_SCROLLED} accessibility event.
 */
private class SendViewScrolledAccessibilityEvent implements Runnable {
    public volatile boolean mIsPending;
```

```
    public void run() {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_SCROLLED);
        mIsPending = false;
    }
}

/**
 * <p>
 * This class represents a delegate that can be registered in a {@link View}
 * to enhance accessibility support via composition rather via inheritance.
 * It is specifically targeted to widget developers that extend basic View
 * classes i.e. classes in package android.view, that would like their
 * applications to be backwards compatible.
 * </p>
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For more information about making applications accessible, read the
 * <a href="{@docRoot}guide/topics/ui/accessibility/index.html">Accessibility</a>
 * developer guide.</p>
 * </div>
 * <p>
 * A scenario in which a developer would like to use an accessibility delegate
 * is overriding a method introduced in a later API version than the minimal API
 * version supported by the application. For example, the method
 * {@link View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)} is not available
 * in API version 4 when the accessibility APIs were first introduced. If a
 * developer would like his application to run on API version 4 devices (assuming
 * all other APIs used by the application are version 4 or lower) and take advantage
 * of this method, instead of overriding the method which would break the application's
 * backwards compatibility, he can override the corresponding method in this
 * delegate and register the delegate in the target View if the API version of
 * the system is high enough, i.e. the API version is the same as or higher than the API
 * version that introduced
 * {@link View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)}.
 * </p>
 * <p>
 * Here is an example implementation:
 * </p>
 * <code><pre><p>
 * if (Build.VERSION.SDK_INT >= 14) {
 *     // If the API version is equal of higher than the version in
 *     // which onInitializeAccessibilityNodeInfo was introduced we
 *     // register a delegate with a customized implementation.
 *     View view = findViewById(R.id.view_id);
 *     view.setAccessibilityDelegate(new AccessibilityDelegate() {
 *         public void onInitializeAccessibilityNodeInfo(View host,
 *                 AccessibilityNodeInfo info) {
 *             // Let the default implementation populate the info.
 *             super.onInitializeAccessibilityNodeInfo(host, info);
 *             // Set some other information.
 *             info.setEnabled(host.isEnabled());
 *         }
 *     });
 * }
 * </code></pre></p>
 * <p>
 * This delegate contains methods that correspond to the accessibility methods
 * in View. If a delegate has been specified the implementation in View hands
 * off handling to the corresponding method in this delegate. The default
 * implementation the delegate methods behaves exactly as the corresponding
 * method in View for the case of no accessibility delegate been set. Hence,
 * to customize the behavior of a View method, clients can override only the
 * corresponding delegate method without altering the behavior of the rest
 * accessibility related methods of the host view.
 * </p>
 * <p>
 * <strong>Note:</strong> On platform versions prior to
 * {@link android.os.Build.VERSION_CODES#M API 23}, delegate methods on
 * views in the {@code android.widget.*} package are called <i>before</i>
 * host methods. This prevents certain properties such as class name from
 * being modified by overriding
 * {@link AccessibilityDelegate#onInitializeAccessibilityNodeInfo(View, AccessibilityNodeInfo)},
 * as any changes will be overwritten by the host class.
 * <p>
 * Starting in {@link android.os.Build.VERSION_CODES#M API 23}, delegate
 * methods are called <i>after</i> host methods, which all properties to be
 * modified without being overwritten by the host class.
 */
public static class AccessibilityDelegate {

    /**
```

```
     * Sends an accessibility event of the given type. If accessibility is not
     * enabled this method has no effect.
     * <p>
     * The default implementation behaves as {@link View#sendAccessibilityEvent(int)
     *  View#sendAccessibilityEvent(int)} for the case of no accessibility delegate
     * been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param eventType The type of the event to send.
     *
     * @see View#sendAccessibilityEvent(int) View#sendAccessibilityEvent(int)
     */
    public void sendAccessibilityEvent(View host, int eventType) {
        host.sendAccessibilityEventInternal(eventType);
    }

    /**
     * Performs the specified accessibility action on the view. For
     * possible accessibility actions look at {@link AccessibilityNodeInfo}.
     * <p>
     * The default implementation behaves as
     * {@link View#performAccessibilityAction(int, Bundle)
     *  View#performAccessibilityAction(int, Bundle)} for the case of
     *  no accessibility delegate been set.
     * </p>
     *
     * @param action The action to perform.
     * @return Whether the action was performed.
     *
     * @see View#performAccessibilityAction(int, Bundle)
     *      View#performAccessibilityAction(int, Bundle)
     */
    public boolean performAccessibilityAction(View host, int action, Bundle args) {
        return host.performAccessibilityActionInternal(action, args);
    }

    /**
     * Sends an accessibility event. This method behaves exactly as
     * {@link #sendAccessibilityEvent(View, int)} but takes as an argument an
     * empty {@link AccessibilityEvent} and does not perform a check whether
     * accessibility is enabled.
     * <p>
     * The default implementation behaves as
     * {@link View#sendAccessibilityEventUnchecked(AccessibilityEvent)
     *  View#sendAccessibilityEventUnchecked(AccessibilityEvent)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param event The event to send.
     *
     * @see View#sendAccessibilityEventUnchecked(AccessibilityEvent)
     *      View#sendAccessibilityEventUnchecked(AccessibilityEvent)
     */
    public void sendAccessibilityEventUnchecked(View host, AccessibilityEvent event) {
        host.sendAccessibilityEventUncheckedInternal(event);
    }

    /**
     * Dispatches an {@link AccessibilityEvent} to the host {@link View} first and then
     * to its children for adding their text content to the event.
     * <p>
     * The default implementation behaves as
     * {@link View#dispatchPopulateAccessibilityEvent(AccessibilityEvent)
     *  View#dispatchPopulateAccessibilityEvent(AccessibilityEvent)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param event The event.
     * @return True if the event population was completed.
     *
     * @see View#dispatchPopulateAccessibilityEvent(AccessibilityEvent)
     *      View#dispatchPopulateAccessibilityEvent(AccessibilityEvent)
     */
    public boolean dispatchPopulateAccessibilityEvent(View host, AccessibilityEvent event) {
        return host.dispatchPopulateAccessibilityEventInternal(event);
    }

    /**
     * Gives a chance to the host View to populate the accessibility event with its
```

```
     * text content.
     * <p>
     * The default implementation behaves as
     * {@link View#onPopulateAccessibilityEvent(AccessibilityEvent)
     *  View#onPopulateAccessibilityEvent(AccessibilityEvent)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param event The accessibility event which to populate.
     *
     * @see View#onPopulateAccessibilityEvent(AccessibilityEvent)
     *      View#onPopulateAccessibilityEvent(AccessibilityEvent)
     */
    public void onPopulateAccessibilityEvent(View host, AccessibilityEvent event) {
        host.onPopulateAccessibilityEventInternal(event);
    }

    /**
     * Initializes an {@link AccessibilityEvent} with information about the
     * the host View which is the event source.
     * <p>
     * The default implementation behaves as
     * {@link View#onInitializeAccessibilityEvent(AccessibilityEvent)
     *  View#onInitializeAccessibilityEvent(AccessibilityEvent)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param event The event to initialize.
     *
     * @see View#onInitializeAccessibilityEvent(AccessibilityEvent)
     *      View#onInitializeAccessibilityEvent(AccessibilityEvent)
     */
    public void onInitializeAccessibilityEvent(View host, AccessibilityEvent event) {
        host.onInitializeAccessibilityEventInternal(event);
    }

    /**
     * Initializes an {@link AccessibilityNodeInfo} with information about the host view.
     * <p>
     * The default implementation behaves as
     * {@link View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)
     *  View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param info The instance to initialize.
     *
     * @see View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)
     *      View#onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)
     */
    public void onInitializeAccessibilityNodeInfo(View host, AccessibilityNodeInfo info) {
        host.onInitializeAccessibilityNodeInfoInternal(info);
    }

    /**
     * Adds extra data to an {@link AccessibilityNodeInfo} based on an explicit request for the
     * additional data.
     * <p>
     * This method only needs to be implemented if the View offers to provide additional data.
     * </p>
     * <p>
     * The default implementation behaves as
     * {@link View#addExtraDataToAccessibilityNodeInfo(AccessibilityNodeInfo, String, Bundle)
     * for the case where no accessibility delegate is set.
     * </p>
     *
     * @param host The View hosting the delegate. Never {@code null}.
     * @param info The info to which to add the extra data. Never {@code null}.
     * @param extraDataKey A key specifying the type of extra data to add to the info. The
     *                     extra data should be added to the {@link Bundle} returned by
     *                     the info's {@link AccessibilityNodeInfo#getExtras} method.  Never
     *                     {@code null}.
     * @param arguments A {@link Bundle} holding any arguments relevant for this request.
     *                  May be {@code null} if the if the service provided no arguments.
     *
     * @see AccessibilityNodeInfo#setExtraAvailableData
     */
    public void addExtraDataToAccessibilityNodeInfo(@NonNull View host,
            @NonNull AccessibilityNodeInfo info, @NonNull String extraDataKey,
```

```
            @Nullable Bundle arguments) {
        host.addExtraDataToAccessibilityNodeInfo(info, extraDataKey, arguments);
    }

    /**
     * Called when a child of the host View has requested sending an
     * {@link AccessibilityEvent} and gives an opportunity to the parent (the host)
     * to augment the event.
     * <p>
     * The default implementation behaves as
     * {@link ViewGroup#onRequestSendAccessibilityEvent(View, AccessibilityEvent)
     *  ViewGroup#onRequestSendAccessibilityEvent(View, AccessibilityEvent)} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @param host The View hosting the delegate.
     * @param child The child which requests sending the event.
     * @param event The event to be sent.
     * @return True if the event should be sent
     *
     * @see ViewGroup#onRequestSendAccessibilityEvent(View, AccessibilityEvent)
     *      ViewGroup#onRequestSendAccessibilityEvent(View, AccessibilityEvent)
     */
    public boolean onRequestSendAccessibilityEvent(ViewGroup host, View child,
            AccessibilityEvent event) {
        return host.onRequestSendAccessibilityEventInternal(child, event);
    }

    /**
     * Gets the provider for managing a virtual view hierarchy rooted at this View
     * and reported to {@link android.accessibilityservice.AccessibilityService}s
     * that explore the window content.
     * <p>
     * The default implementation behaves as
     * {@link View#getAccessibilityNodeProvider() View#getAccessibilityNodeProvider()} for
     * the case of no accessibility delegate been set.
     * </p>
     *
     * @return The provider.
     *
     * @see AccessibilityNodeProvider
     */
    public AccessibilityNodeProvider getAccessibilityNodeProvider(View host) {
        return null;
    }

    /**
     * Returns an {@link AccessibilityNodeInfo} representing the host view from the
     * point of view of an {@link android.accessibilityservice.AccessibilityService}.
     * This method is responsible for obtaining an accessibility node info from a
     * pool of reusable instances and calling
     * {@link #onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo)} on the host
     * view to initialize the former.
     * <p>
     * <strong>Note:</strong> The client is responsible for recycling the obtained
     * instance by calling {@link AccessibilityNodeInfo#recycle()} to minimize object
     * creation.
     * </p>
     * <p>
     * The default implementation behaves as
     * {@link View#createAccessibilityNodeInfo() View#createAccessibilityNodeInfo()} for
     * the case of no accessibility delegate been set.
     * </p>
     * @return A populated {@link AccessibilityNodeInfo}.
     *
     * @see AccessibilityNodeInfo
     *
     * @hide
     */
    public AccessibilityNodeInfo createAccessibilityNodeInfo(View host) {
        return host.createAccessibilityNodeInfoInternal();
    }
}

private static class MatchIdPredicate implements Predicate<View> {
    public int mId;

    @Override
    public boolean test(View view) {
        return (view.mID == mId);
    }
}
```

```
private static class MatchLabelForPredicate implements Predicate<View> {
    private int mLabeledId;

    @Override
    public boolean test(View view) {
        return (view.mLabelForId == mLabeledId);
    }
}

/**
 * Dump all private flags in readable format, useful for documentation and
 * sanity checking.
 */
private static void dumpFlags() {
    final HashMap<String, String> found = Maps.newHashMap();
    try {
        for (Field field : View.class.getDeclaredFields()) {
            final int modifiers = field.getModifiers();
            if (Modifier.isStatic(modifiers) && Modifier.isFinal(modifiers)) {
                if (field.getType().equals(int.class)) {
                    final int value = field.getInt(null);
                    dumpFlag(found, field.getName(), value);
                } else if (field.getType().equals(int[].class)) {
                    final int[] values = (int[]) field.get(null);
                    for (int i = 0; i < values.length; i++) {
                        dumpFlag(found, field.getName() + "[" + i + "]", values[i]);
                    }
                }
            }
        }
    } catch (IllegalAccessException e) {
        throw new RuntimeException(e);
    }

    final ArrayList<String> keys = Lists.newArrayList();
    keys.addAll(found.keySet());
    Collections.sort(keys);
    for (String key : keys) {
        Log.d(VIEW_LOG_TAG, found.get(key));
    }
}

private static void dumpFlag(HashMap<String, String> found, String name, int value) {
    // Sort flags by prefix, then by bits, always keeping unique keys
    final String bits = String.format("%32s", Integer.toBinaryString(value)).replace('0', ' ');
    final int prefix = name.indexOf('_');
    final String key = (prefix > 0 ? name.substring(0, prefix) : name) + bits + name;
    final String output = bits + " " + name;
    found.put(key, output);
}

/** {@hide} */
public void encode(@NonNull ViewHierarchyEncoder stream) {
    stream.beginObject(this);
    encodeProperties(stream);
    stream.endObject();
}

/** {@hide} */
@CallSuper
protected void encodeProperties(@NonNull ViewHierarchyEncoder stream) {
    Object resolveId = ViewDebug.resolveId(getContext(), mID);
    if (resolveId instanceof String) {
        stream.addProperty("id", (String) resolveId);
    } else {
        stream.addProperty("id", mID);
    }

    stream.addProperty("misc:transformation.alpha",
            mTransformationInfo != null ? mTransformationInfo.mAlpha : 0);
    stream.addProperty("misc:transitionName", getTransitionName());

    // layout
    stream.addProperty("layout:left", mLeft);
    stream.addProperty("layout:right", mRight);
    stream.addProperty("layout:top", mTop);
    stream.addProperty("layout:bottom", mBottom);
    stream.addProperty("layout:width", getWidth());
    stream.addProperty("layout:height", getHeight());
    stream.addProperty("layout:layoutDirection", getLayoutDirection());
    stream.addProperty("layout:layoutRtl", isLayoutRtl());
```

```
stream.addProperty("layout:hasTransientState", hasTransientState());
stream.addProperty("layout:baseline", getBaseline());

// layout params
ViewGroup.LayoutParams layoutParams = getLayoutParams();
if (layoutParams != null) {
    stream.addPropertyKey("layoutParams");
    layoutParams.encode(stream);
}

// scrolling
stream.addProperty("scrolling:scrollX", mScrollX);
stream.addProperty("scrolling:scrollY", mScrollY);

// padding
stream.addProperty("padding:paddingLeft", mPaddingLeft);
stream.addProperty("padding:paddingRight", mPaddingRight);
stream.addProperty("padding:paddingTop", mPaddingTop);
stream.addProperty("padding:paddingBottom", mPaddingBottom);
stream.addProperty("padding:userPaddingRight", mUserPaddingRight);
stream.addProperty("padding:userPaddingLeft", mUserPaddingLeft);
stream.addProperty("padding:userPaddingBottom", mUserPaddingBottom);
stream.addProperty("padding:userPaddingStart", mUserPaddingStart);
stream.addProperty("padding:userPaddingEnd", mUserPaddingEnd);

// measurement
stream.addProperty("measurement:minHeight", mMinHeight);
stream.addProperty("measurement:minWidth", mMinWidth);
stream.addProperty("measurement:measuredWidth", mMeasuredWidth);
stream.addProperty("measurement:measuredHeight", mMeasuredHeight);

// drawing
stream.addProperty("drawing:elevation", getElevation());
stream.addProperty("drawing:translationX", getTranslationX());
stream.addProperty("drawing:translationY", getTranslationY());
stream.addProperty("drawing:translationZ", getTranslationZ());
stream.addProperty("drawing:rotation", getRotation());
stream.addProperty("drawing:rotationX", getRotationX());
stream.addProperty("drawing:rotationY", getRotationY());
stream.addProperty("drawing:scaleX", getScaleX());
stream.addProperty("drawing:scaleY", getScaleY());
stream.addProperty("drawing:pivotX", getPivotX());
stream.addProperty("drawing:pivotY", getPivotY());
stream.addProperty("drawing:opaque", isOpaque());
stream.addProperty("drawing:alpha", getAlpha());
stream.addProperty("drawing:transitionAlpha", getTransitionAlpha());
stream.addProperty("drawing:shadow", hasShadow());
stream.addProperty("drawing:solidColor", getSolidColor());
stream.addProperty("drawing:layerType", mLayerType);
stream.addProperty("drawing:willNotDraw", willNotDraw());
stream.addProperty("drawing:hardwareAccelerated", isHardwareAccelerated());
stream.addProperty("drawing:willNotCacheDrawing", willNotCacheDrawing());
stream.addProperty("drawing:drawingCacheEnabled", isDrawingCacheEnabled());
stream.addProperty("drawing:overlappingRendering", hasOverlappingRendering());

// focus
stream.addProperty("focus:hasFocus", hasFocus());
stream.addProperty("focus:isFocused", isFocused());
stream.addProperty("focus:focusable", getFocusable());
stream.addProperty("focus:isFocusable", isFocusable());
stream.addProperty("focus:isFocusableInTouchMode", isFocusableInTouchMode());

stream.addProperty("misc:clickable", isClickable());
stream.addProperty("misc:pressed", isPressed());
stream.addProperty("misc:selected", isSelected());
stream.addProperty("misc:touchMode", isInTouchMode());
stream.addProperty("misc:hovered", isHovered());
stream.addProperty("misc:activated", isActivated());

stream.addProperty("misc:visibility", getVisibility());
stream.addProperty("misc:fitsSystemWindows", getFitsSystemWindows());
stream.addProperty("misc:filterTouchesWhenObscured", getFilterTouchesWhenObscured());

stream.addProperty("misc:enabled", isEnabled());
stream.addProperty("misc:soundEffectsEnabled", isSoundEffectsEnabled());
stream.addProperty("misc:hapticFeedbackEnabled", isHapticFeedbackEnabled());

// theme attributes
Resources.Theme theme = getContext().getTheme();
if (theme != null) {
    stream.addPropertyKey("theme");
    theme.encode(stream);
```

```
    }

    // view attribute information
    int n = mAttributes != null ? mAttributes.length : 0;
    stream.addProperty("meta:__attrCount__", n/2);
    for (int i = 0; i < n; i += 2) {
        stream.addProperty("meta:__attr__" + mAttributes[i], mAttributes[i+1]);
    }

    stream.addProperty("misc:scrollBarStyle", getScrollBarStyle());

    // text
    stream.addProperty("text:textDirection", getTextDirection());
    stream.addProperty("text:textAlignment", getTextAlignment());

    // accessibility
    CharSequence contentDescription = getContentDescription();
    stream.addProperty("accessibility:contentDescription",
            contentDescription == null ? "" : contentDescription.toString());
    stream.addProperty("accessibility:labelFor", getLabelFor());
    stream.addProperty("accessibility:importantForAccessibility", getImportantForAccessibility());
}

/**
 * Determine if this view is rendered on a round wearable device and is the main view
 * on the screen.
 */
boolean shouldDrawRoundScrollbar() {
    if (!mResources.getConfiguration().isScreenRound() || mAttachInfo == null) {
        return false;
    }

    final View rootView = getRootView();
    final WindowInsets insets = getRootWindowInsets();

    int height = getHeight();
    int width = getWidth();
    int displayHeight = rootView.getHeight();
    int displayWidth = rootView.getWidth();

    if (height != displayHeight || width != displayWidth) {
        return false;
    }

    getLocationInWindow(mAttachInfo.mTmpLocation);
    return mAttachInfo.mTmpLocation[0] == insets.getStableInsetLeft()
            && mAttachInfo.mTmpLocation[1] == insets.getStableInsetTop();
}

/**
 * Sets the tooltip text which will be displayed in a small popup next to the view.
 * <p>
 * The tooltip will be displayed:
 * <ul>
 * <li>On long click, unless it is handled otherwise (by OnLongClickListener or a context
 * menu). </li>
 * <li>On hover, after a brief delay since the pointer has stopped moving </li>
 * </ul>
 * <p>
 * <strong>Note:</strong> Do not override this method, as it will have no
 * effect on the text displayed in the tooltip.
 *
 * @param tooltipText the tooltip text, or null if no tooltip is required
 * @see #getTooltipText()
 * @attr ref android.R.styleable#View_tooltipText
 */
public void setTooltipText(@Nullable CharSequence tooltipText) {
    if (TextUtils.isEmpty(tooltipText)) {
        setFlags(0, TOOLTIP);
        hideTooltip();
        mTooltipInfo = null;
    } else {
        setFlags(TOOLTIP, TOOLTIP);
        if (mTooltipInfo == null) {
            mTooltipInfo = new TooltipInfo();
            mTooltipInfo.mShowTooltipRunnable = this::showHoverTooltip;
            mTooltipInfo.mHideTooltipRunnable = this::hideTooltip;
        }
        mTooltipInfo.mTooltipText = tooltipText;
    }
}
```

```java
/**
 * @hide Binary compatibility stub. To be removed when we finalize O APIs.
 */
public void setTooltip(@Nullable CharSequence tooltipText) {
    setTooltipText(tooltipText);
}

/**
 * Returns the view's tooltip text.
 *
 * <strong>Note:</strong> Do not override this method, as it will have no
 * effect on the text displayed in the tooltip. You must call
 * {@link #setTooltipText(CharSequence)} to modify the tooltip text.
 *
 * @return the tooltip text
 * @see #setTooltipText(CharSequence)
 * @attr ref android.R.styleable#View_tooltipText
 */
@Nullable
public CharSequence getTooltipText() {
    return mTooltipInfo != null ? mTooltipInfo.mTooltipText : null;
}

/**
 * @hide Binary compatibility stub. To be removed when we finalize O APIs.
 */
@Nullable
public CharSequence getTooltip() {
    return getTooltipText();
}

private boolean showTooltip(int x, int y, boolean fromLongClick) {
    if (mAttachInfo == null || mTooltipInfo == null) {
        return false;
    }
    if ((mViewFlags & ENABLED_MASK) != ENABLED) {
        return false;
    }
    if (TextUtils.isEmpty(mTooltipInfo.mTooltipText)) {
        return false;
    }
    hideTooltip();
    mTooltipInfo.mTooltipFromLongClick = fromLongClick;
    mTooltipInfo.mTooltipPopup = new TooltipPopup(getContext());
    final boolean fromTouch = (mPrivateFlags3 & PFLAG3_FINGER_DOWN) == PFLAG3_FINGER_DOWN;
    mTooltipInfo.mTooltipPopup.show(this, x, y, fromTouch, mTooltipInfo.mTooltipText);
    mAttachInfo.mTooltipHost = this;
    return true;
}

void hideTooltip() {
    if (mTooltipInfo == null) {
        return;
    }
    removeCallbacks(mTooltipInfo.mShowTooltipRunnable);
    if (mTooltipInfo.mTooltipPopup == null) {
        return;
    }
    mTooltipInfo.mTooltipPopup.hide();
    mTooltipInfo.mTooltipPopup = null;
    mTooltipInfo.mTooltipFromLongClick = false;
    if (mAttachInfo != null) {
        mAttachInfo.mTooltipHost = null;
    }
}

private boolean showLongClickTooltip(int x, int y) {
    removeCallbacks(mTooltipInfo.mShowTooltipRunnable);
    removeCallbacks(mTooltipInfo.mHideTooltipRunnable);
    return showTooltip(x, y, true);
}

private void showHoverTooltip() {
    showTooltip(mTooltipInfo.mAnchorX, mTooltipInfo.mAnchorY, false);
}

boolean dispatchTooltipHoverEvent(MotionEvent event) {
    if (mTooltipInfo == null) {
        return false;
    }
    switch(event.getAction()) {
        case MotionEvent.ACTION_HOVER_MOVE:
```

```
            if ((mViewFlags & TOOLTIP) != TOOLTIP || (mViewFlags & ENABLED_MASK) != ENABLED) {
                break;
            }
            if (!mTooltipInfo.mTooltipFromLongClick) {
                if (mTooltipInfo.mTooltipPopup == null) {
                    // Schedule showing the tooltip after a timeout.
                    mTooltipInfo.mAnchorX = (int) event.getX();
                    mTooltipInfo.mAnchorY = (int) event.getY();
                    removeCallbacks(mTooltipInfo.mShowTooltipRunnable);
                    postDelayed(mTooltipInfo.mShowTooltipRunnable,
                            ViewConfiguration.getHoverTooltipShowTimeout());
                }

                // Hide hover-triggered tooltip after a period of inactivity.
                // Match the timeout used by NativeInputManager to hide the mouse pointer
                // (depends on SYSTEM_UI_FLAG_LOW_PROFILE being set).
                final int timeout;
                if ((getWindowSystemUiVisibility() & SYSTEM_UI_FLAG_LOW_PROFILE)
                        == SYSTEM_UI_FLAG_LOW_PROFILE) {
                    timeout = ViewConfiguration.getHoverTooltipHideShortTimeout();
                } else {
                    timeout = ViewConfiguration.getHoverTooltipHideTimeout();
                }
                removeCallbacks(mTooltipInfo.mHideTooltipRunnable);
                postDelayed(mTooltipInfo.mHideTooltipRunnable, timeout);
            }
            return true;

        case MotionEvent.ACTION_HOVER_EXIT:
            if (!mTooltipInfo.mTooltipFromLongClick) {
                hideTooltip();
            }
            break;
    }
    return false;
}

void handleTooltipKey(KeyEvent event) {
    switch (event.getAction()) {
        case KeyEvent.ACTION_DOWN:
            if (event.getRepeatCount() == 0) {
                hideTooltip();
            }
            break;

        case KeyEvent.ACTION_UP:
            handleTooltipUp();
            break;
    }
}

private void handleTooltipUp() {
    if (mTooltipInfo == null || mTooltipInfo.mTooltipPopup == null) {
        return;
    }
    removeCallbacks(mTooltipInfo.mHideTooltipRunnable);
    postDelayed(mTooltipInfo.mHideTooltipRunnable,
            ViewConfiguration.getLongPressTooltipHideTimeout());
}

private int getFocusableAttribute(TypedArray attributes) {
    TypedValue val = new TypedValue();
    if (attributes.getValue(com.android.internal.R.styleable.View_focusable, val)) {
        if (val.type == TypedValue.TYPE_INT_BOOLEAN) {
            return (val.data == 0 ? NOT_FOCUSABLE : FOCUSABLE);
        } else {
            return val.data;
        }
    } else {
        return FOCUSABLE_AUTO;
    }
}

/**
 * @return The content view of the tooltip popup currently being shown, or null if the tooltip
 * is not showing.
 * @hide
 */
@TestApi
public View getTooltipView() {
    if (mTooltipInfo == null || mTooltipInfo.mTooltipPopup == null) {
        return null;
```

```
        }
        return mTooltipInfo.mTooltipPopup.getContentView();
    }
}
```