```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 android.view;

import static android.os.Build.VERSION_CODES.JELLY_BEAN_MR1;

import android.animation.LayoutTransition;
import android.annotation.CallSuper;
import android.annotation.IdRes;
import android.annotation.NonNull;
import android.annotation.TestApi;
import android.annotation.UiThread;
import android.content.ClipData;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.res.Configuration;
import android.content.res.TypedArray;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Insets;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.PointF;
import android.graphics.Rect;
import android.graphics.RectF;
import android.graphics.Region;
import android.os.Build;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.SystemClock;
import android.util.AttributeSet;
import android.util.Log;
import android.util.Pools;
import android.util.Pools.SynchronizedPool;
import android.util.SparseArray;
import android.util.SparseBooleanArray;
import android.view.accessibility.AccessibilityEvent;
import android.view.accessibility.AccessibilityManager;
import android.view.accessibility.AccessibilityNodeInfo;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.view.animation.LayoutAnimationController;
import android.view.animation.Transformation;

import com.android.internal.R;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;

/**
 * <p>
 * A <code>ViewGroup</code> is a special view that can contain other views
 * (called children.) The view group is the base class for layouts and views
 * containers. This class also defines the
 * {@link android.view.ViewGroup.LayoutParams} class which serves as the base
 * class for layouts parameters.
 * </p>
 *
```

```
 * <p>
 * Also see {@link LayoutParams} for layout attributes.
 * </p>
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For more information about creating user interface layouts, read the
 * <a href="{@docRoot}guide/topics/ui/declaring-layout.html">XML Layouts</a> developer
 * guide.</p></div>
 *
 * <p>Here is a complete implementation of a custom ViewGroup that implements
 * a simple {@link android.widget.FrameLayout} along with the ability to stack
 * children in left and right gutters.</p>
 *
 * {@sample development/samples/ApiDemos/src/com/example/android/apis/view/CustomLayout.java
 *      Complete}
 *
 * <p>If you are implementing XML layout attributes as shown in the example, this is the
 * corresponding definition for them that would go in <code>res/values/attrs.xml</code>:</p>
 *
 * {@sample development/samples/ApiDemos/res/values/attrs.xml CustomLayout}
 *
 * <p>Finally the layout manager can be used in an XML layout like so:</p>
 *
 * {@sample development/samples/ApiDemos/res/layout/custom_layout.xml Complete}
 *
 * @attr ref android.R.styleable#ViewGroup_clipChildren
 * @attr ref android.R.styleable#ViewGroup_clipToPadding
 * @attr ref android.R.styleable#ViewGroup_layoutAnimation
 * @attr ref android.R.styleable#ViewGroup_animationCache
 * @attr ref android.R.styleable#ViewGroup_persistentDrawingCache
 * @attr ref android.R.styleable#ViewGroup_alwaysDrawnWithCache
 * @attr ref android.R.styleable#ViewGroup_addStatesFromChildren
 * @attr ref android.R.styleable#ViewGroup_descendantFocusability
 * @attr ref android.R.styleable#ViewGroup_animateLayoutChanges
 * @attr ref android.R.styleable#ViewGroup_splitMotionEvents
 * @attr ref android.R.styleable#ViewGroup_layoutMode
 */
@UiThread
public abstract class ViewGroup extends View implements ViewParent, ViewManager {
    private static final String TAG = "ViewGroup";

    private static final boolean DBG = false;

    /**
     * Views which have been hidden or removed which need to be animated on
     * their way out.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    protected ArrayList<View> mDisappearingChildren;

    /**
     * Listener used to propagate events indicating when children are added
     * and/or removed from a view group.
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    protected OnHierarchyChangeListener mOnHierarchyChangeListener;

    // The view contained within this ViewGroup that has or contains focus.
    private View mFocused;
    // The view contained within this ViewGroup (excluding nested keyboard navigation clusters)
    // that is or contains a default-focus view.
    private View mDefaultFocus;
    // The last child of this ViewGroup which held focus within the current cluster
    View mFocusedInCluster;

    /**
     * A Transformation used when drawing children, to
     * apply on the child being drawn.
     */
    private Transformation mChildTransformation;

    /**
     * Used to track the current invalidation region.
     */
    RectF mInvalidateRegion;

    /**
     * A Transformation used to calculate a correct
     * invalidation area when the application is autoscaled.
```

```
     */
    Transformation mInvalidationTransformation;

    // Current frontmost child that can accept drag and lies under the drag location.
    // Used only to generate ENTER/EXIT events for pre-Nougat aps.
    private View mCurrentDragChild;

    // Metadata about the ongoing drag
    private DragEvent mCurrentDragStartEvent;
    private boolean mIsInterestedInDrag;
    private HashSet<View> mChildrenInterestedInDrag;

    // Used during drag dispatch
    private PointF mLocalPoint;

    // Lazily-created holder for point computations.
    private float[] mTempPoint;

    // Layout animation
    private LayoutAnimationController mLayoutAnimationController;
    private Animation.AnimationListener mAnimationListener;

    // First touch target in the linked list of touch targets.
    private TouchTarget mFirstTouchTarget;

    // For debugging only.  You can see these in hierarchyviewer.
    @SuppressWarnings({"FieldCanBeLocal", "UnusedDeclaration"})
    @ViewDebug.ExportedProperty(category = "events")
    private long mLastTouchDownTime;
    @ViewDebug.ExportedProperty(category = "events")
    private int mLastTouchDownIndex = -1;
    @SuppressWarnings({"FieldCanBeLocal", "UnusedDeclaration"})
    @ViewDebug.ExportedProperty(category = "events")
    private float mLastTouchDownX;
    @SuppressWarnings({"FieldCanBeLocal", "UnusedDeclaration"})
    @ViewDebug.ExportedProperty(category = "events")
    private float mLastTouchDownY;

    // First hover target in the linked list of hover targets.
    // The hover targets are children which have received ACTION_HOVER_ENTER.
    // They might not have actually handled the hover event, but we will
    // continue sending hover events to them as long as the pointer remains over
    // their bounds and the view group does not intercept hover.
    private HoverTarget mFirstHoverTarget;

    // True if the view group itself received a hover event.
    // It might not have actually handled the hover event.
    private boolean mHoveredSelf;

    // The child capable of showing a tooltip and currently under the pointer.
    private View mTooltipHoverTarget;

    // True if the view group is capable of showing a tooltip and the pointer is directly
    // over the view group but not one of its child views.
    private boolean mTooltipHoveredSelf;

    /**
     * Internal flags.
     *
     * This field should be made private, so it is hidden from the SDK.
     * {@hide}
     */
    @ViewDebug.ExportedProperty(flagMapping = {
            @ViewDebug.FlagToString(mask = FLAG_CLIP_CHILDREN, equals = FLAG_CLIP_CHILDREN,
                    name = "CLIP_CHILDREN"),
            @ViewDebug.FlagToString(mask = FLAG_CLIP_TO_PADDING, equals = FLAG_CLIP_TO_PADDING,
                    name = "CLIP_TO_PADDING"),
            @ViewDebug.FlagToString(mask = FLAG_PADDING_NOT_NULL, equals = FLAG_PADDING_NOT_NULL,
                    name = "PADDING_NOT_NULL")
    }, formatToHexString = true)
    protected int mGroupFlags;

    /**
     * Either {@link #LAYOUT_MODE_CLIP_BOUNDS} or {@link #LAYOUT_MODE_OPTICAL_BOUNDS}.
     */
    private int mLayoutMode = LAYOUT_MODE_UNDEFINED;

    /**
     * NOTE: If you change the flags below make sure to reflect the changes
     *       the DisplayList class
     */
```

```
// When set, ViewGroup invalidates only the child's rectangle
// Set by default
static final int FLAG_CLIP_CHILDREN = 0x1;

// When set, ViewGroup excludes the padding area from the invalidate rectangle
// Set by default
private static final int FLAG_CLIP_TO_PADDING = 0x2;

// When set, dispatchDraw() will invoke invalidate(); this is set by drawChild() when
// a child needs to be invalidated and FLAG_OPTIMIZE_INVALIDATE is set
static final int FLAG_INVALIDATE_REQUIRED  = 0x4;

// When set, dispatchDraw() will run the layout animation and unset the flag
private static final int FLAG_RUN_ANIMATION = 0x8;

// When set, there is either no layout animation on the ViewGroup or the layout
// animation is over
// Set by default
static final int FLAG_ANIMATION_DONE = 0x10;

// If set, this ViewGroup has padding; if unset there is no padding and we don't need
// to clip it, even if FLAG_CLIP_TO_PADDING is set
private static final int FLAG_PADDING_NOT_NULL = 0x20;

/** @deprecated - functionality removed */
@Deprecated
private static final int FLAG_ANIMATION_CACHE = 0x40;

// When set, this ViewGroup converts calls to invalidate(Rect) to invalidate() during a
// layout animation; this avoid clobbering the hierarchy
// Automatically set when the layout animation starts, depending on the animation's
// characteristics
static final int FLAG_OPTIMIZE_INVALIDATE = 0x80;

// When set, the next call to drawChild() will clear mChildTransformation's matrix
static final int FLAG_CLEAR_TRANSFORMATION = 0x100;

// When set, this ViewGroup invokes mAnimationListener.onAnimationEnd() and removes
// the children's Bitmap caches if necessary
// This flag is set when the layout animation is over (after FLAG_ANIMATION_DONE is set)
private static final int FLAG_NOTIFY_ANIMATION_LISTENER = 0x200;

/**
 * When set, the drawing method will call {@link #getChildDrawingOrder(int, int)}
 * to get the index of the child to draw for that iteration.
 *
 * @hide
 */
protected static final int FLAG_USE_CHILD_DRAWING_ORDER = 0x400;

/**
 * When set, this ViewGroup supports static transformations on children; this causes
 * {@link #getChildStaticTransformation(View, android.view.animation.Transformation)} to be
 * invoked when a child is drawn.
 *
 * Any subclass overriding
 * {@link #getChildStaticTransformation(View, android.view.animation.Transformation)} should
 * set this flags in {@link #mGroupFlags}.
 *
 * {@hide}
 */
protected static final int FLAG_SUPPORT_STATIC_TRANSFORMATIONS = 0x800;

// UNUSED FLAG VALUE: 0x1000;

/**
 * When set, this ViewGroup's drawable states also include those
 * of its children.
 */
private static final int FLAG_ADD_STATES_FROM_CHILDREN = 0x2000;

/** @deprecated functionality removed */
@Deprecated
private static final int FLAG_ALWAYS_DRAWN_WITH_CACHE = 0x4000;

/** @deprecated functionality removed */
@Deprecated
private static final int FLAG_CHILDREN_DRAWN_WITH_CACHE = 0x8000;

/**
 * When set, this group will go through its list of children to notify them of
 * any drawable state change.
```

```
     */
    private static final int FLAG_NOTIFY_CHILDREN_ON_DRAWABLE_STATE_CHANGE = 0x10000;

    private static final int FLAG_MASK_FOCUSABILITY = 0x60000;

    /**
     * This view will get focus before any of its descendants.
     */
    public static final int FOCUS_BEFORE_DESCENDANTS = 0x20000;

    /**
     * This view will get focus only if none of its descendants want it.
     */
    public static final int FOCUS_AFTER_DESCENDANTS = 0x40000;

    /**
     * This view will block any of its descendants from getting focus, even
     * if they are focusable.
     */
    public static final int FOCUS_BLOCK_DESCENDANTS = 0x60000;

    /**
     * Used to map between enum in attrubutes and flag values.
     */
    private static final int[] DESCENDANT_FOCUSABILITY_FLAGS =
            {FOCUS_BEFORE_DESCENDANTS, FOCUS_AFTER_DESCENDANTS,
                    FOCUS_BLOCK_DESCENDANTS};

    /**
     * When set, this ViewGroup should not intercept touch events.
     * {@hide}
     */
    protected static final int FLAG_DISALLOW_INTERCEPT = 0x80000;

    /**
     * When set, this ViewGroup will split MotionEvents to multiple child Views when appropriate.
     */
    private static final int FLAG_SPLIT_MOTION_EVENTS = 0x200000;

    /**
     * When set, this ViewGroup will not dispatch onAttachedToWindow calls
     * to children when adding new views. This is used to prevent multiple
     * onAttached calls when a ViewGroup adds children in its own onAttached method.
     */
    private static final int FLAG_PREVENT_DISPATCH_ATTACHED_TO_WINDOW = 0x400000;

    /**
     * When true, indicates that a layoutMode has been explicitly set, either with
     * an explicit call to {@link #setLayoutMode(int)} in code or from an XML resource.
     * This distinguishes the situation in which a layout mode was inherited from
     * one of the ViewGroup's ancestors and cached locally.
     */
    private static final int FLAG_LAYOUT_MODE_WAS_EXPLICITLY_SET = 0x800000;

    static final int FLAG_IS_TRANSITION_GROUP = 0x1000000;

    static final int FLAG_IS_TRANSITION_GROUP_SET = 0x2000000;

    /**
     * When set, focus will not be permitted to enter this group if a touchscreen is present.
     */
    static final int FLAG_TOUCHSCREEN_BLOCKS_FOCUS = 0x4000000;

    /**
     * When true, indicates that a call to startActionModeForChild was made with the type parameter
     * and should not be ignored. This helps in backwards compatibility with the existing method
     * without a type.
     *
     * @see #startActionModeForChild(View, android.view.ActionMode.Callback)
     * @see #startActionModeForChild(View, android.view.ActionMode.Callback, int)
     */
    private static final int FLAG_START_ACTION_MODE_FOR_CHILD_IS_TYPED = 0x8000000;

    /**
     * When true, indicates that a call to startActionModeForChild was made without the type
     * parameter. This helps in backwards compatibility with the existing method
     * without a type.
     *
     * @see #startActionModeForChild(View, android.view.ActionMode.Callback)
     * @see #startActionModeForChild(View, android.view.ActionMode.Callback, int)
     */
    private static final int FLAG_START_ACTION_MODE_FOR_CHILD_IS_NOT_TYPED = 0x10000000;
```

```java
/**
 * When set, indicates that a call to showContextMenuForChild was made with explicit
 * coordinates within the initiating child view.
 */
private static final int FLAG_SHOW_CONTEXT_MENU_WITH_COORDS = 0x20000000;

/**
 * Indicates which types of drawing caches are to be kept in memory.
 * This field should be made private, so it is hidden from the SDK.
 * {@hide}
 */
protected int mPersistentDrawingCache;

/**
 * Used to indicate that no drawing cache should be kept in memory.
 */
public static final int PERSISTENT_NO_CACHE = 0x0;

/**
 * Used to indicate that the animation drawing cache should be kept in memory.
 */
public static final int PERSISTENT_ANIMATION_CACHE = 0x1;

/**
 * Used to indicate that the scrolling drawing cache should be kept in memory.
 */
public static final int PERSISTENT_SCROLLING_CACHE = 0x2;

/**
 * Used to indicate that all drawing caches should be kept in memory.
 */
public static final int PERSISTENT_ALL_CACHES = 0x3;

// Layout Modes

private static final int LAYOUT_MODE_UNDEFINED = -1;

/**
 * This constant is a {@link #setLayoutMode(int) layoutMode}.
 * Clip bounds are the raw values of {@link #getLeft() left}, {@link #getTop() top},
 * {@link #getRight() right} and {@link #getBottom() bottom}.
 */
public static final int LAYOUT_MODE_CLIP_BOUNDS = 0;

/**
 * This constant is a {@link #setLayoutMode(int) layoutMode}.
 * Optical bounds describe where a widget appears to be. They sit inside the clip
 * bounds which need to cover a larger area to allow other effects,
 * such as shadows and glows, to be drawn.
 */
public static final int LAYOUT_MODE_OPTICAL_BOUNDS = 1;

/** @hide */
public static int LAYOUT_MODE_DEFAULT = LAYOUT_MODE_CLIP_BOUNDS;

/**
 * We clip to padding when FLAG_CLIP_TO_PADDING and FLAG_PADDING_NOT_NULL
 * are set at the same time.
 */
protected static final int CLIP_TO_PADDING_MASK = FLAG_CLIP_TO_PADDING | FLAG_PADDING_NOT_NULL;

// Index of the child's left position in the mLocation array
private static final int CHILD_LEFT_INDEX = 0;
// Index of the child's top position in the mLocation array
private static final int CHILD_TOP_INDEX = 1;

// Child views of this ViewGroup
private View[] mChildren;
// Number of valid children in the mChildren array, the rest should be null or not
// considered as children
private int mChildrenCount;

// Whether layout calls are currently being suppressed, controlled by calls to
// suppressLayout()
boolean mSuppressLayout = false;

// Whether any layout calls have actually been suppressed while mSuppressLayout
// has been true. This tracks whether we need to issue a requestLayout() when
// layout is later re-enabled.
private boolean mLayoutCalledWhileSuppressed = false;
```

```
private static final int ARRAY_INITIAL_CAPACITY = 12;
private static final int ARRAY_CAPACITY_INCREMENT = 12;

private static float[] sDebugLines;

// Used to draw cached views
Paint mCachePaint;

// Used to animate add/remove changes in layout
private LayoutTransition mTransition;

// The set of views that are currently being transitioned. This list is used to track views
// being removed that should not actually be removed from the parent yet because they are
// being animated.
private ArrayList<View> mTransitioningViews;

// List of children changing visibility. This is used to potentially keep rendering
// views during a transition when they otherwise would have become gone/invisible
private ArrayList<View> mVisibilityChangingChildren;

// Temporary holder of presorted children, only used for
// input/software draw dispatch for correctly Z ordering.
private ArrayList<View> mPreSortedChildren;

// Indicates how many of this container's child subtrees contain transient state
@ViewDebug.ExportedProperty(category = "layout")
private int mChildCountWithTransientState = 0;

/**
 * Currently registered axes for nested scrolling. Flag set consisting of
 * {@link #SCROLL_AXIS_HORIZONTAL} {@link #SCROLL_AXIS_VERTICAL} or {@link #SCROLL_AXIS_NONE}
 * for null.
 */
private int mNestedScrollAxes;

// Used to manage the list of transient views, added by addTransientView()
private List<Integer> mTransientIndices = null;
private List<View> mTransientViews = null;


/**
 * Empty ActionMode used as a sentinel in recursive entries to startActionModeForChild.
 *
 * @see #startActionModeForChild(View, android.view.ActionMode.Callback)
 * @see #startActionModeForChild(View, android.view.ActionMode.Callback, int)
 */
private static final ActionMode SENTINEL_ACTION_MODE = new ActionMode() {
    @Override
    public void setTitle(CharSequence title) {}

    @Override
    public void setTitle(int resId) {}

    @Override
    public void setSubtitle(CharSequence subtitle) {}

    @Override
    public void setSubtitle(int resId) {}

    @Override
    public void setCustomView(View view) {}

    @Override
    public void invalidate() {}

    @Override
    public void finish() {}

    @Override
    public Menu getMenu() {
        return null;
    }

    @Override
    public CharSequence getTitle() {
        return null;
    }

    @Override
    public CharSequence getSubtitle() {
        return null;
    }
```

```
    @Override
    public View getCustomView() {
        return null;
    }

    @Override
    public MenuInflater getMenuInflater() {
        return null;
    }
};

public ViewGroup(Context context) {
    this(context, null);
}

public ViewGroup(Context context, AttributeSet attrs) {
    this(context, attrs, 0);
}

public ViewGroup(Context context, AttributeSet attrs, int defStyleAttr) {
    this(context, attrs, defStyleAttr, 0);
}

public ViewGroup(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {
    super(context, attrs, defStyleAttr, defStyleRes);

    initViewGroup();
    initFromAttributes(context, attrs, defStyleAttr, defStyleRes);
}

private void initViewGroup() {
    // ViewGroup doesn't draw by default
    if (!debugDraw()) {
        setFlags(WILL_NOT_DRAW, DRAW_MASK);
    }
    mGroupFlags |= FLAG_CLIP_CHILDREN;
    mGroupFlags |= FLAG_CLIP_TO_PADDING;
    mGroupFlags |= FLAG_ANIMATION_DONE;
    mGroupFlags |= FLAG_ANIMATION_CACHE;
    mGroupFlags |= FLAG_ALWAYS_DRAWN_WITH_CACHE;

    if (mContext.getApplicationInfo().targetSdkVersion >= Build.VERSION_CODES.HONEYCOMB) {
        mGroupFlags |= FLAG_SPLIT_MOTION_EVENTS;
    }

    setDescendantFocusability(FOCUS_BEFORE_DESCENDANTS);

    mChildren = new View[ARRAY_INITIAL_CAPACITY];
    mChildrenCount = 0;

    mPersistentDrawingCache = PERSISTENT_SCROLLING_CACHE;
}

private void initFromAttributes(
        Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {
    final TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.ViewGroup, defStyleAttr,
            defStyleRes);

    final int N = a.getIndexCount();
    for (int i = 0; i < N; i++) {
        int attr = a.getIndex(i);
        switch (attr) {
            case R.styleable.ViewGroup_clipChildren:
                setClipChildren(a.getBoolean(attr, true));
                break;
            case R.styleable.ViewGroup_clipToPadding:
                setClipToPadding(a.getBoolean(attr, true));
                break;
            case R.styleable.ViewGroup_animationCache:
                setAnimationCacheEnabled(a.getBoolean(attr, true));
                break;
            case R.styleable.ViewGroup_persistentDrawingCache:
                setPersistentDrawingCache(a.getInt(attr, PERSISTENT_SCROLLING_CACHE));
                break;
            case R.styleable.ViewGroup_addStatesFromChildren:
                setAddStatesFromChildren(a.getBoolean(attr, false));
                break;
            case R.styleable.ViewGroup_alwaysDrawnWithCache:
                setAlwaysDrawnWithCacheEnabled(a.getBoolean(attr, true));
                break;
            case R.styleable.ViewGroup_layoutAnimation:
```

```
                int id = a.getResourceId(attr, -1);
                if (id > 0) {
                    setLayoutAnimation(AnimationUtils.loadLayoutAnimation(mContext, id));
                }
                break;
            case R.styleable.ViewGroup_descendantFocusability:
                setDescendantFocusability(DESCENDANT_FOCUSABILITY_FLAGS[a.getInt(attr, 0)]);
                break;
            case R.styleable.ViewGroup_splitMotionEvents:
                setMotionEventSplittingEnabled(a.getBoolean(attr, false));
                break;
            case R.styleable.ViewGroup_animateLayoutChanges:
                boolean animateLayoutChanges = a.getBoolean(attr, false);
                if (animateLayoutChanges) {
                    setLayoutTransition(new LayoutTransition());
                }
                break;
            case R.styleable.ViewGroup_layoutMode:
                setLayoutMode(a.getInt(attr, LAYOUT_MODE_UNDEFINED));
                break;
            case R.styleable.ViewGroup_transitionGroup:
                setTransitionGroup(a.getBoolean(attr, false));
                break;
            case R.styleable.ViewGroup_touchscreenBlocksFocus:
                setTouchscreenBlocksFocus(a.getBoolean(attr, false));
                break;
        }
    }

    a.recycle();
}

/**
 * Gets the descendant focusability of this view group.  The descendant
 * focusability defines the relationship between this view group and its
 * descendants when looking for a view to take focus in
 * {@link #requestFocus(int, android.graphics.Rect)}.
 *
 * @return one of {@link #FOCUS_BEFORE_DESCENDANTS}, {@link #FOCUS_AFTER_DESCENDANTS},
 *   {@link #FOCUS_BLOCK_DESCENDANTS}.
 */
@ViewDebug.ExportedProperty(category = "focus", mapping = {
    @ViewDebug.IntToString(from = FOCUS_BEFORE_DESCENDANTS, to = "FOCUS_BEFORE_DESCENDANTS"),
    @ViewDebug.IntToString(from = FOCUS_AFTER_DESCENDANTS, to = "FOCUS_AFTER_DESCENDANTS"),
    @ViewDebug.IntToString(from = FOCUS_BLOCK_DESCENDANTS, to = "FOCUS_BLOCK_DESCENDANTS")
})
public int getDescendantFocusability() {
    return mGroupFlags & FLAG_MASK_FOCUSABILITY;
}

/**
 * Set the descendant focusability of this view group. This defines the relationship
 * between this view group and its descendants when looking for a view to
 * take focus in {@link #requestFocus(int, android.graphics.Rect)}.
 *
 * @param focusability one of {@link #FOCUS_BEFORE_DESCENDANTS}, {@link #FOCUS_AFTER_DESCENDANTS},
 *   {@link #FOCUS_BLOCK_DESCENDANTS}.
 */
public void setDescendantFocusability(int focusability) {
    switch (focusability) {
        case FOCUS_BEFORE_DESCENDANTS:
        case FOCUS_AFTER_DESCENDANTS:
        case FOCUS_BLOCK_DESCENDANTS:
            break;
        default:
            throw new IllegalArgumentException("must be one of FOCUS_BEFORE_DESCENDANTS, "
                    + "FOCUS_AFTER_DESCENDANTS, FOCUS_BLOCK_DESCENDANTS");
    }
    mGroupFlags &= ~FLAG_MASK_FOCUSABILITY;
    mGroupFlags |= (focusability & FLAG_MASK_FOCUSABILITY);
}

@Override
void handleFocusGainInternal(int direction, Rect previouslyFocusedRect) {
    if (mFocused != null) {
        mFocused.unFocus(this);
        mFocused = null;
        mFocusedInCluster = null;
    }
    super.handleFocusGainInternal(direction, previouslyFocusedRect);
}
```

```
@Override
public void requestChildFocus(View child, View focused) {
    if (DBG) {
        System.out.println(this + " requestChildFocus()");
    }
    if (getDescendantFocusability() == FOCUS_BLOCK_DESCENDANTS) {
        return;
    }

    // Unfocus us, if necessary
    super.unFocus(focused);

    // We had a previous notion of who had focus. Clear it.
    if (mFocused != child) {
        if (mFocused != null) {
            mFocused.unFocus(focused);
        }

        mFocused = child;
    }
    if (mParent != null) {
        mParent.requestChildFocus(this, focused);
    }
}

void setDefaultFocus(View child) {
    // Stop at any higher view which is explicitly focused-by-default
    if (mDefaultFocus != null && mDefaultFocus.isFocusedByDefault()) {
        return;
    }

    mDefaultFocus = child;

    if (mParent instanceof ViewGroup) {
        ((ViewGroup) mParent).setDefaultFocus(this);
    }
}

/**
 * Clears the default-focus chain from {@param child} up to the first parent which has another
 * default-focusable branch below it or until there is no default-focus chain.
 *
 * @param child
 */
void clearDefaultFocus(View child) {
    // Stop at any higher view which is explicitly focused-by-default
    if (mDefaultFocus != child && mDefaultFocus != null
            && mDefaultFocus.isFocusedByDefault()) {
        return;
    }

    mDefaultFocus = null;

    // Search child siblings for default focusables.
    for (int i = 0; i < mChildrenCount; ++i) {
        View sibling = mChildren[i];
        if (sibling.isFocusedByDefault()) {
            mDefaultFocus = sibling;
            return;
        } else if (mDefaultFocus == null && sibling.hasDefaultFocus()) {
            mDefaultFocus = sibling;
        }
    }

    if (mParent instanceof ViewGroup) {
        ((ViewGroup) mParent).clearDefaultFocus(this);
    }
}

@Override
boolean hasDefaultFocus() {
    return mDefaultFocus != null || super.hasDefaultFocus();
}

/**
 * Removes {@code child} (and associated focusedInCluster chain) from the cluster containing
 * it.
 * <br>
 * This is intended to be run on {@code child}'s immediate parent. This is necessary because
 * the chain is sometimes cleared after {@code child} has been detached.
 */
void clearFocusedInCluster(View child) {
```

```
    if (mFocusedInCluster != child) {
        return;
    }
    clearFocusedInCluster();
}

/**
 * Removes the focusedInCluster chain from this up to the cluster containing it.
 */
void clearFocusedInCluster() {
    View top = findKeyboardNavigationCluster();
    ViewParent parent = this;
    do {
        ((ViewGroup) parent).mFocusedInCluster = null;
        if (parent == top) {
            break;
        }
        parent = parent.getParent();
    } while (parent instanceof ViewGroup);
}

@Override
public void focusableViewAvailable(View v) {
    if (mParent != null
            // shortcut: don't report a new focusable view if we block our descendants from
            // getting focus or if we're not visible
            && (getDescendantFocusability() != FOCUS_BLOCK_DESCENDANTS)
            && ((mViewFlags & VISIBILITY_MASK) == VISIBLE)
            && (isFocusableInTouchMode() || !shouldBlockFocusForTouchscreen())
            // shortcut: don't report a new focusable view if we already are focused
            // (and we don't prefer our descendants)
            //
            // note: knowing that mFocused is non-null is not a good enough reason
            // to break the traversal since in that case we'd actually have to find
            // the focused view and make sure it wasn't FOCUS_AFTER_DESCENDANTS and
            // an ancestor of v; this will get checked for at ViewAncestor
            && !(isFocused() && getDescendantFocusability() != FOCUS_AFTER_DESCENDANTS)) {
        mParent.focusableViewAvailable(v);
    }
}

@Override
public boolean showContextMenuForChild(View originalView) {
    if (isShowingContextMenuWithCoords()) {
        // We're being called for compatibility. Return false and let the version
        // with coordinates recurse up.
        return false;
    }
    return mParent != null && mParent.showContextMenuForChild(originalView);
}

/**
 * @hide used internally for compatibility with existing app code only
 */
public final boolean isShowingContextMenuWithCoords() {
    return (mGroupFlags & FLAG_SHOW_CONTEXT_MENU_WITH_COORDS) != 0;
}

@Override
public boolean showContextMenuForChild(View originalView, float x, float y) {
    try {
        mGroupFlags |= FLAG_SHOW_CONTEXT_MENU_WITH_COORDS;
        if (showContextMenuForChild(originalView)) {
            return true;
        }
    } finally {
        mGroupFlags &= ~FLAG_SHOW_CONTEXT_MENU_WITH_COORDS;
    }
    return mParent != null && mParent.showContextMenuForChild(originalView, x, y);
}

@Override
public ActionMode startActionModeForChild(View originalView, ActionMode.Callback callback) {
    if ((mGroupFlags & FLAG_START_ACTION_MODE_FOR_CHILD_IS_TYPED) == 0) {
        // This is the original call.
        try {
            mGroupFlags |= FLAG_START_ACTION_MODE_FOR_CHILD_IS_NOT_TYPED;
            return startActionModeForChild(originalView, callback, ActionMode.TYPE_PRIMARY);
        } finally {
            mGroupFlags &= ~FLAG_START_ACTION_MODE_FOR_CHILD_IS_NOT_TYPED;
        }
    } else {
```

```
        // We are being called from the new method with type.
        return SENTINEL_ACTION_MODE;
    }
}

@Override
public ActionMode startActionModeForChild(
        View originalView, ActionMode.Callback callback, int type) {
    if ((mGroupFlags & FLAG_START_ACTION_MODE_FOR_CHILD_IS_NOT_TYPED) == 0
            && type == ActionMode.TYPE_PRIMARY) {
        ActionMode mode;
        try {
            mGroupFlags |= FLAG_START_ACTION_MODE_FOR_CHILD_IS_TYPED;
            mode = startActionModeForChild(originalView, callback);
        } finally {
            mGroupFlags &= ~FLAG_START_ACTION_MODE_FOR_CHILD_IS_TYPED;
        }
        if (mode != SENTINEL_ACTION_MODE) {
            return mode;
        }
    }
    if (mParent != null) {
        try {
            return mParent.startActionModeForChild(originalView, callback, type);
        } catch (AbstractMethodError ame) {
            // Custom view parents might not implement this method.
            return mParent.startActionModeForChild(originalView, callback);
        }
    }
    return null;
}

/**
 * @hide
 */
@Override
public boolean dispatchActivityResult(
        String who, int requestCode, int resultCode, Intent data) {
    if (super.dispatchActivityResult(who, requestCode, resultCode, data)) {
        return true;
    }
    int childCount = getChildCount();
    for (int i = 0; i < childCount; i++) {
        View child = getChildAt(i);
        if (child.dispatchActivityResult(who, requestCode, resultCode, data)) {
            return true;
        }
    }
    return false;
}

/**
 * Find the nearest view in the specified direction that wants to take
 * focus.
 *
 * @param focused The view that currently has focus
 * @param direction One of FOCUS_UP, FOCUS_DOWN, FOCUS_LEFT, and
 *        FOCUS_RIGHT, or 0 for not applicable.
 */
@Override
public View focusSearch(View focused, int direction) {
    if (isRootNamespace()) {
        // root namespace means we should consider ourselves the top of the
        // tree for focus searching; otherwise we could be focus searching
        // into other tabs.  see LocalActivityManager and TabHost for more info.
        return FocusFinder.getInstance().findNextFocus(this, focused, direction);
    } else if (mParent != null) {
        return mParent.focusSearch(focused, direction);
    }
    return null;
}

@Override
public boolean requestChildRectangleOnScreen(View child, Rect rectangle, boolean immediate) {
    return false;
}

@Override
public boolean requestSendAccessibilityEvent(View child, AccessibilityEvent event) {
    ViewParent parent = mParent;
    if (parent == null) {
        return false;
```

```
    }
    final boolean propagate = onRequestSendAccessibilityEvent(child, event);
    if (!propagate) {
        return false;
    }
    return parent.requestSendAccessibilityEvent(this, event);
}

/**
 * Called when a child has requested sending an {@link AccessibilityEvent} and
 * gives an opportunity to its parent to augment the event.
 * <p>
 * If an {@link android.view.View.AccessibilityDelegate} has been specified via calling
 * {@link android.view.View#setAccessibilityDelegate(android.view.View.AccessibilityDelegate)} its
 * {@link android.view.View.AccessibilityDelegate#onRequestSendAccessibilityEvent(ViewGroup, View, AccessibilityEvent)}
 * is responsible for handling this call.
 * </p>
 *
 * @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 #requestSendAccessibilityEvent(View, AccessibilityEvent)
 */
public boolean onRequestSendAccessibilityEvent(View child, AccessibilityEvent event) {
    if (mAccessibilityDelegate != null) {
        return mAccessibilityDelegate.onRequestSendAccessibilityEvent(this, child, event);
    } else {
        return onRequestSendAccessibilityEventInternal(child, event);
    }
}

/**
 * @see #onRequestSendAccessibilityEvent(View, AccessibilityEvent)
 *
 * Note: Called from the default {@link View.AccessibilityDelegate}.
 *
 * @hide
 */
public boolean onRequestSendAccessibilityEventInternal(View child, AccessibilityEvent event) {
    return true;
}

/**
 * Called when a child view has changed whether or not it is tracking transient state.
 */
@Override
public void childHasTransientStateChanged(View child, boolean childHasTransientState) {
    final boolean oldHasTransientState = hasTransientState();
    if (childHasTransientState) {
        mChildCountWithTransientState++;
    } else {
        mChildCountWithTransientState--;
    }

    final boolean newHasTransientState = hasTransientState();
    if (mParent != null && oldHasTransientState != newHasTransientState) {
        try {
            mParent.childHasTransientStateChanged(this, newHasTransientState);
        } catch (AbstractMethodError e) {
            Log.e(TAG, mParent.getClass().getSimpleName() +
                    " does not fully implement ViewParent", e);
        }
    }
}

@Override
public boolean hasTransientState() {
    return mChildCountWithTransientState > 0 || super.hasTransientState();
}

@Override
public boolean dispatchUnhandledMove(View focused, int direction) {
    return mFocused != null &&
            mFocused.dispatchUnhandledMove(focused, direction);
}

@Override
public void clearChildFocus(View child) {
    if (DBG) {
        System.out.println(this + " clearChildFocus()");
    }
```

```
    mFocused = null;
    if (mParent != null) {
        mParent.clearChildFocus(this);
    }
}

@Override
public void clearFocus() {
    if (DBG) {
        System.out.println(this + " clearFocus()");
    }
    if (mFocused == null) {
        super.clearFocus();
    } else {
        View focused = mFocused;
        mFocused = null;
        focused.clearFocus();
    }
}

@Override
void unFocus(View focused) {
    if (DBG) {
        System.out.println(this + " unFocus()");
    }
    if (mFocused == null) {
        super.unFocus(focused);
    } else {
        mFocused.unFocus(focused);
        mFocused = null;
    }
}

/**
 * Returns the focused child of this view, if any. The child may have focus
 * or contain focus.
 *
 * @return the focused child or null.
 */
public View getFocusedChild() {
    return mFocused;
}

View getDeepestFocusedChild() {
    View v = this;
    while (v != null) {
        if (v.isFocused()) {
            return v;
        }
        v = v instanceof ViewGroup ? ((ViewGroup) v).getFocusedChild() : null;
    }
    return null;
}

/**
 * Returns true if this view has or contains focus
 *
 * @return true if this view has or contains focus
 */
@Override
public boolean hasFocus() {
    return (mPrivateFlags & PFLAG_FOCUSED) != 0 || mFocused != null;
}

/*
 * (non-Javadoc)
 *
 * @see android.view.View#findFocus()
 */
@Override
public View findFocus() {
    if (DBG) {
        System.out.println("Find focus in " + this + ": flags="
                + isFocused() + ", child=" + mFocused);
    }

    if (isFocused()) {
        return this;
    }

    if (mFocused != null) {
```

```
        return mFocused.findFocus();
    }
    return null;
}

@Override
boolean hasFocusable(boolean allowAutoFocus, boolean dispatchExplicit) {
    // This should probably be super.hasFocusable, but that would change
    // behavior. Historically, we have not checked the ancestor views for
    // shouldBlockFocusForTouchscreen() in ViewGroup.hasFocusable.

    // 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;
    }

    // Determine whether we have a focused descendant.
    final int descendantFocusability = getDescendantFocusability();
    if (descendantFocusability != FOCUS_BLOCK_DESCENDANTS) {
        return hasFocusableChild(dispatchExplicit);
    }

    return false;
}

boolean hasFocusableChild(boolean dispatchExplicit) {
    // Determine whether we have a focusable descendant.
    final int count = mChildrenCount;
    final View[] children = mChildren;

    for (int i = 0; i < count; i++) {
        final View child = children[i];

        // In case the subclass has overridden has[Explicit]Focusable, dispatch
        // to the expected one for each child even though we share logic here.
        if ((dispatchExplicit && child.hasExplicitFocusable())
                || (!dispatchExplicit && child.hasFocusable())) {
            return true;
        }
    }

    return false;
}

@Override
public void addFocusables(ArrayList<View> views, int direction, int focusableMode) {
    final int focusableCount = views.size();

    final int descendantFocusability = getDescendantFocusability();
    final boolean blockFocusForTouchscreen = shouldBlockFocusForTouchscreen();
    final boolean focusSelf = (isFocusableInTouchMode() || !blockFocusForTouchscreen);

    if (descendantFocusability == FOCUS_BLOCK_DESCENDANTS) {
        if (focusSelf) {
            super.addFocusables(views, direction, focusableMode);
        }
        return;
    }

    if (blockFocusForTouchscreen) {
        focusableMode |= FOCUSABLES_TOUCH_MODE;
    }

    if ((descendantFocusability == FOCUS_BEFORE_DESCENDANTS) && focusSelf) {
        super.addFocusables(views, direction, focusableMode);
    }

    int count = 0;
    final View[] children = new View[mChildrenCount];
    for (int i = 0; i < mChildrenCount; ++i) {
        View child = mChildren[i];
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
            children[count++] = child;
        }
    }
    FocusFinder.sort(children, 0, count, this, isLayoutRtl());
    for (int i = 0; i < count; ++i) {
```

```
            children[i].addFocusables(views, direction, focusableMode);
        }

        // When set to FOCUS_AFTER_DESCENDANTS, we only add ourselves if
        // there aren't any focusable descendants.  this is
        // to avoid the focus search finding layouts when a more precise search
        // among the focusable children would be more interesting.
        if ((descendantFocusability == FOCUS_AFTER_DESCENDANTS) && focusSelf
                && focusableCount == views.size()) {
            super.addFocusables(views, direction, focusableMode);
        }
    }

    @Override
    public void addKeyboardNavigationClusters(Collection<View> views, int direction) {
        final int focusableCount = views.size();

        if (isKeyboardNavigationCluster()) {
            // Cluster-navigation can enter a touchscreenBlocksFocus cluster, so temporarily
            // disable touchscreenBlocksFocus to evaluate whether it contains focusables.
            final boolean blockedFocus = getTouchscreenBlocksFocus();
            try {
                setTouchscreenBlocksFocusNoRefocus(false);
                super.addKeyboardNavigationClusters(views, direction);
            } finally {
                setTouchscreenBlocksFocusNoRefocus(blockedFocus);
            }
        } else {
            super.addKeyboardNavigationClusters(views, direction);
        }

        if (focusableCount != views.size()) {
            // No need to look for groups inside a group.
            return;
        }

        if (getDescendantFocusability() == FOCUS_BLOCK_DESCENDANTS) {
            return;
        }

        int count = 0;
        final View[] visibleChildren = new View[mChildrenCount];
        for (int i = 0; i < mChildrenCount; ++i) {
            final View child = mChildren[i];
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
                visibleChildren[count++] = child;
            }
        }
        FocusFinder.sort(visibleChildren, 0, count, this, isLayoutRtl());
        for (int i = 0; i < count; ++i) {
            visibleChildren[i].addKeyboardNavigationClusters(views, direction);
        }
    }

    /**
     * Set whether this ViewGroup should ignore focus requests for itself and its children.
     * If this option is enabled and the ViewGroup or a descendant currently has focus, focus
     * will proceed forward.
     *
     * @param touchscreenBlocksFocus true to enable blocking focus in the presence of a touchscreen
     */
    public void setTouchscreenBlocksFocus(boolean touchscreenBlocksFocus) {
        if (touchscreenBlocksFocus) {
            mGroupFlags |= FLAG_TOUCHSCREEN_BLOCKS_FOCUS;
            if (hasFocus() && !isKeyboardNavigationCluster()) {
                final View focusedChild = getDeepestFocusedChild();
                if (!focusedChild.isFocusableInTouchMode()) {
                    final View newFocus = focusSearch(FOCUS_FORWARD);
                    if (newFocus != null) {
                        newFocus.requestFocus();
                    }
                }
            }
        } else {
            mGroupFlags &= ~FLAG_TOUCHSCREEN_BLOCKS_FOCUS;
        }
    }

    private void setTouchscreenBlocksFocusNoRefocus(boolean touchscreenBlocksFocus) {
        if (touchscreenBlocksFocus) {
            mGroupFlags |= FLAG_TOUCHSCREEN_BLOCKS_FOCUS;
        } else {
```

```
        mGroupFlags &= ~FLAG_TOUCHSCREEN_BLOCKS_FOCUS;
    }
}

/**
 * Check whether this ViewGroup should ignore focus requests for itself and its children.
 */
@ViewDebug.ExportedProperty(category = "focus")
public boolean getTouchscreenBlocksFocus() {
    return (mGroupFlags & FLAG_TOUCHSCREEN_BLOCKS_FOCUS) != 0;
}

boolean shouldBlockFocusForTouchscreen() {
    // There is a special case for keyboard-navigation clusters. We allow cluster navigation
    // to jump into blockFocusForTouchscreen ViewGroups which are clusters. Once in the
    // cluster, focus is free to move around within it.
    return getTouchscreenBlocksFocus() &&
            mContext.getPackageManager().hasSystemFeature(PackageManager.FEATURE_TOUCHSCREEN)
            && !(isKeyboardNavigationCluster()
                    && (hasFocus() || (findKeyboardNavigationCluster() != this)));
}

@Override
public void findViewsWithText(ArrayList<View> outViews, CharSequence text, int flags) {
    super.findViewsWithText(outViews, text, flags);
    final int childrenCount = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < childrenCount; i++) {
        View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE
                && (child.mPrivateFlags & PFLAG_IS_ROOT_NAMESPACE) == 0) {
            child.findViewsWithText(outViews, text, flags);
        }
    }
}

/** @hide */
@Override
public View findViewByAccessibilityIdTraversal(int accessibilityId) {
    View foundView = super.findViewByAccessibilityIdTraversal(accessibilityId);
    if (foundView != null) {
        return foundView;
    }

    if (getAccessibilityNodeProvider() != null) {
        return null;
    }

    final int childrenCount = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < childrenCount; i++) {
        View child = children[i];
        foundView = child.findViewByAccessibilityIdTraversal(accessibilityId);
        if (foundView != null) {
            return foundView;
        }
    }

    return null;
}

/** @hide */
@Override
public View findViewByAutofillIdTraversal(int autofillId) {
    View foundView = super.findViewByAutofillIdTraversal(autofillId);
    if (foundView != null) {
        return foundView;
    }

    final int childrenCount = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < childrenCount; i++) {
        View child = children[i];
        foundView = child.findViewByAutofillIdTraversal(autofillId);
        if (foundView != null) {
            return foundView;
        }
    }

    return null;
}
```

```java
@Override
public void dispatchWindowFocusChanged(boolean hasFocus) {
    super.dispatchWindowFocusChanged(hasFocus);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchWindowFocusChanged(hasFocus);
    }
}

@Override
public void addTouchables(ArrayList<View> views) {
    super.addTouchables(views);

    final int count = mChildrenCount;
    final View[] children = mChildren;

    for (int i = 0; i < count; i++) {
        final View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
            child.addTouchables(views);
        }
    }
}

/**
 * @hide
 */
@Override
public void makeOptionalFitsSystemWindows() {
    super.makeOptionalFitsSystemWindows();
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].makeOptionalFitsSystemWindows();
    }
}

@Override
public void dispatchDisplayHint(int hint) {
    super.dispatchDisplayHint(hint);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchDisplayHint(hint);
    }
}

/**
 * Called when a view's visibility has changed. Notify the parent to take any appropriate
 * action.
 *
 * @param child The view whose visibility has changed
 * @param oldVisibility The previous visibility value (GONE, INVISIBLE, or VISIBLE).
 * @param newVisibility The new visibility value (GONE, INVISIBLE, or VISIBLE).
 * @hide
 */
protected void onChildVisibilityChanged(View child, int oldVisibility, int newVisibility) {
    if (mTransition != null) {
        if (newVisibility == VISIBLE) {
            mTransition.showChild(this, child, oldVisibility);
        } else {
            mTransition.hideChild(this, child, newVisibility);
            if (mTransitioningViews != null && mTransitioningViews.contains(child)) {
                // Only track this on disappearing views - appearing views are already visible
                // and don't need special handling during drawChild()
                if (mVisibilityChangingChildren == null) {
                    mVisibilityChangingChildren = new ArrayList<View>();
                }
                mVisibilityChangingChildren.add(child);
                addDisappearingView(child);
            }
        }
    }

    // in all cases, for drags
    if (newVisibility == VISIBLE && mCurrentDragStartEvent != null) {
        if (!mChildrenInterestedInDrag.contains(child)) {
            notifyChildOfDragStart(child);
        }
    }
}
```

```
@Override
protected void dispatchVisibilityChanged(View changedView, int visibility) {
    super.dispatchVisibilityChanged(changedView, visibility);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchVisibilityChanged(changedView, visibility);
    }
}

@Override
public void dispatchWindowVisibilityChanged(int visibility) {
    super.dispatchWindowVisibilityChanged(visibility);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchWindowVisibilityChanged(visibility);
    }
}

@Override
boolean dispatchVisibilityAggregated(boolean isVisible) {
    isVisible = super.dispatchVisibilityAggregated(isVisible);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        // Only dispatch to visible children. Not visible children and their subtrees already
        // know that they aren't visible and that's not going to change as a result of
        // whatever triggered this dispatch.
        if (children[i].getVisibility() == VISIBLE) {
            children[i].dispatchVisibilityAggregated(isVisible);
        }
    }
    return isVisible;
}

@Override
public void dispatchConfigurationChanged(Configuration newConfig) {
    super.dispatchConfigurationChanged(newConfig);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchConfigurationChanged(newConfig);
    }
}

@Override
public void recomputeViewAttributes(View child) {
    if (mAttachInfo != null && !mAttachInfo.mRecomputeGlobalAttributes) {
        ViewParent parent = mParent;
        if (parent != null) parent.recomputeViewAttributes(this);
    }
}

@Override
void dispatchCollectViewAttributes(AttachInfo attachInfo, int visibility) {
    if ((visibility & VISIBILITY_MASK) == VISIBLE) {
        super.dispatchCollectViewAttributes(attachInfo, visibility);
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            final View child = children[i];
            child.dispatchCollectViewAttributes(attachInfo,
                    visibility | (child.mViewFlags&VISIBILITY_MASK));
        }
    }
}

@Override
public void bringChildToFront(View child) {
    final int index = indexOfChild(child);
    if (index >= 0) {
        removeFromArray(index);
        addInArray(child, mChildrenCount);
        child.mParent = this;
        requestLayout();
        invalidate();
    }
}

private PointF getLocalPoint() {
```

```
    if (mLocalPoint == null) mLocalPoint = new PointF();
    return mLocalPoint;
}

@Override
boolean dispatchDragEnterExitInPreN(DragEvent event) {
    if (event.mAction == DragEvent.ACTION_DRAG_EXITED && mCurrentDragChild != null) {
        // The drag exited a sub-tree of views; notify of the exit all descendants that are in
        // entered state.
        // We don't need this recursive delivery for ENTERED events because they get generated
        // from the recursive delivery of LOCATION/DROP events, and hence, don't need their own
        // recursion.
        mCurrentDragChild.dispatchDragEnterExitInPreN(event);
        mCurrentDragChild = null;
    }
    return mIsInterestedInDrag && super.dispatchDragEnterExitInPreN(event);
}

// TODO: Write real docs
@Override
public boolean dispatchDragEvent(DragEvent event) {
    boolean retval = false;
    final float tx = event.mX;
    final float ty = event.mY;
    final ClipData td = event.mClipData;

    // Dispatch down the view hierarchy
    final PointF localPoint = getLocalPoint();

    switch (event.mAction) {
    case DragEvent.ACTION_DRAG_STARTED: {
        // Clear the state to recalculate which views we drag over.
        mCurrentDragChild = null;

        // Set up our tracking of drag-started notifications
        mCurrentDragStartEvent = DragEvent.obtain(event);
        if (mChildrenInterestedInDrag == null) {
            mChildrenInterestedInDrag = new HashSet<View>();
        } else {
            mChildrenInterestedInDrag.clear();
        }

        // Now dispatch down to our children, caching the responses
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            final View child = children[i];
            child.mPrivateFlags2 &= ~View.DRAG_MASK;
            if (child.getVisibility() == VISIBLE) {
                if (notifyChildOfDragStart(children[i])) {
                    retval = true;
                }
            }
        }

        // Notify itself of the drag start.
        mIsInterestedInDrag = super.dispatchDragEvent(event);
        if (mIsInterestedInDrag) {
            retval = true;
        }

        if (!retval) {
            // Neither us nor any of our children are interested in this drag, so stop tracking
            // the current drag event.
            mCurrentDragStartEvent.recycle();
            mCurrentDragStartEvent = null;
        }
    } break;

    case DragEvent.ACTION_DRAG_ENDED: {
        // Release the bookkeeping now that the drag lifecycle has ended
        final HashSet<View> childrenInterestedInDrag = mChildrenInterestedInDrag;
        if (childrenInterestedInDrag != null) {
            for (View child : childrenInterestedInDrag) {
                // If a child was interested in the ongoing drag, it's told that it's over
                if (child.dispatchDragEvent(event)) {
                    retval = true;
                }
            }
            childrenInterestedInDrag.clear();
        }
        if (mCurrentDragStartEvent != null) {
```

```
            mCurrentDragStartEvent.recycle();
            mCurrentDragStartEvent = null;
        }

        if (mIsInterestedInDrag) {
            if (super.dispatchDragEvent(event)) {
                retval = true;
            }
            mIsInterestedInDrag = false;
        }
    } break;

    case DragEvent.ACTION_DRAG_LOCATION:
    case DragEvent.ACTION_DROP: {
        // Find the [possibly new] drag target
        View target = findFrontmostDroppableChildAt(event.mX, event.mY, localPoint);

        if (target != mCurrentDragChild) {
            if (sCascadedDragDrop) {
                // For pre-Nougat apps, make sure that the whole hierarchy of views that contain
                // the drag location is kept in the state between ENTERED and EXITED events.
                // (Starting with N, only the innermost view will be in that state).

                final int action = event.mAction;
                // Position should not be available for ACTION_DRAG_ENTERED and
                // ACTION_DRAG_EXITED.
                event.mX = 0;
                event.mY = 0;
                event.mClipData = null;

                if (mCurrentDragChild != null) {
                    event.mAction = DragEvent.ACTION_DRAG_EXITED;
                    mCurrentDragChild.dispatchDragEnterExitInPreN(event);
                }

                if (target != null) {
                    event.mAction = DragEvent.ACTION_DRAG_ENTERED;
                    target.dispatchDragEnterExitInPreN(event);
                }

                event.mAction = action;
                event.mX = tx;
                event.mY = ty;
                event.mClipData = td;
            }
            mCurrentDragChild = target;
        }

        if (target == null && mIsInterestedInDrag) {
            target = this;
        }

        // Dispatch the actual drag notice, localized into the target coordinates.
        if (target != null) {
            if (target != this) {
                event.mX = localPoint.x;
                event.mY = localPoint.y;

                retval = target.dispatchDragEvent(event);

                event.mX = tx;
                event.mY = ty;

                if (mIsInterestedInDrag) {
                    final boolean eventWasConsumed;
                    if (sCascadedDragDrop) {
                        eventWasConsumed = retval;
                    } else {
                        eventWasConsumed = event.mEventHandlerWasCalled;
                    }

                    if (!eventWasConsumed) {
                        retval = super.dispatchDragEvent(event);
                    }
                }
            } else {
                retval = super.dispatchDragEvent(event);
            }
        }
    } break;
    }
```

```
    return retval;
}

// Find the frontmost child view that lies under the given point, and calculate
// the position within its own local coordinate system.
View findFrontmostDroppableChildAt(float x, float y, PointF outLocalPoint) {
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = count - 1; i >= 0; i--) {
        final View child = children[i];
        if (!child.canAcceptDrag()) {
            continue;
        }

        if (isTransformedTouchPointInView(x, y, child, outLocalPoint)) {
            return child;
        }
    }
    return null;
}

boolean notifyChildOfDragStart(View child) {
    // The caller guarantees that the child is not in mChildrenInterestedInDrag yet.

    if (ViewDebug.DEBUG_DRAG) {
        Log.d(View.VIEW_LOG_TAG, "Sending drag-started to view: " + child);
    }

    final float tx = mCurrentDragStartEvent.mX;
    final float ty = mCurrentDragStartEvent.mY;

    final float[] point = getTempPoint();
    point[0] = tx;
    point[1] = ty;
    transformPointToViewLocal(point, child);

    mCurrentDragStartEvent.mX = point[0];
    mCurrentDragStartEvent.mY = point[1];
    final boolean canAccept = child.dispatchDragEvent(mCurrentDragStartEvent);
    mCurrentDragStartEvent.mX = tx;
    mCurrentDragStartEvent.mY = ty;
    mCurrentDragStartEvent.mEventHandlerWasCalled = false;
    if (canAccept) {
        mChildrenInterestedInDrag.add(child);
        if (!child.canAcceptDrag()) {
            child.mPrivateFlags2 |= View.PFLAG2_DRAG_CAN_ACCEPT;
            child.refreshDrawableState();
        }
    }
    return canAccept;
}

@Override
public void dispatchWindowSystemUiVisiblityChanged(int visible) {
    super.dispatchWindowSystemUiVisiblityChanged(visible);

    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i=0; i <count; i++) {
        final View child = children[i];
        child.dispatchWindowSystemUiVisiblityChanged(visible);
    }
}

@Override
public void dispatchSystemUiVisibilityChanged(int visible) {
    super.dispatchSystemUiVisibilityChanged(visible);

    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i=0; i <count; i++) {
        final View child = children[i];
        child.dispatchSystemUiVisibilityChanged(visible);
    }
}

@Override
boolean updateLocalSystemUiVisibility(int localValue, int localChanges) {
    boolean changed = super.updateLocalSystemUiVisibility(localValue, localChanges);

    final int count = mChildrenCount;
    final View[] children = mChildren;
```

```
        for (int i=0; i <count; i++) {
            final View child = children[i];
            changed |= child.updateLocalSystemUiVisibility(localValue, localChanges);
        }
        return changed;
    }

    @Override
    public boolean dispatchKeyEventPreIme(KeyEvent event) {
        if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
                == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
            return super.dispatchKeyEventPreIme(event);
        } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
                == PFLAG_HAS_BOUNDS) {
            return mFocused.dispatchKeyEventPreIme(event);
        }
        return false;
    }

    @Override
    public boolean dispatchKeyEvent(KeyEvent event) {
        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onKeyEvent(event, 1);
        }

        if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
                == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
            if (super.dispatchKeyEvent(event)) {
                return true;
            }
        } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
                == PFLAG_HAS_BOUNDS) {
            if (mFocused.dispatchKeyEvent(event)) {
                return true;
            }
        }

        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onUnhandledEvent(event, 1);
        }
        return false;
    }

    @Override
    public boolean dispatchKeyShortcutEvent(KeyEvent event) {
        if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
                == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
            return super.dispatchKeyShortcutEvent(event);
        } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
                == PFLAG_HAS_BOUNDS) {
            return mFocused.dispatchKeyShortcutEvent(event);
        }
        return false;
    }

    @Override
    public boolean dispatchTrackballEvent(MotionEvent event) {
        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onTrackballEvent(event, 1);
        }

        if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
                == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
            if (super.dispatchTrackballEvent(event)) {
                return true;
            }
        } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
                == PFLAG_HAS_BOUNDS) {
            if (mFocused.dispatchTrackballEvent(event)) {
                return true;
            }
        }

        if (mInputEventConsistencyVerifier != null) {
            mInputEventConsistencyVerifier.onUnhandledEvent(event, 1);
        }
        return false;
    }

    @Override
    public boolean dispatchCapturedPointerEvent(MotionEvent event) {
        if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
```

```
            == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
        if (super.dispatchCapturedPointerEvent(event)) {
            return true;
        }
    } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
            == PFLAG_HAS_BOUNDS) {
        if (mFocused.dispatchCapturedPointerEvent(event)) {
            return true;
        }
    }
    return false;
}

@Override
public void dispatchPointerCaptureChanged(boolean hasCapture) {
    exitHoverTargets();

    super.dispatchPointerCaptureChanged(hasCapture);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchPointerCaptureChanged(hasCapture);
    }
}

@Override
public PointerIcon onResolvePointerIcon(MotionEvent event, int pointerIndex) {
    final float x = event.getX(pointerIndex);
    final float y = event.getY(pointerIndex);
    if (isOnScrollbarThumb(x, y) || isDraggingScrollBar()) {
        return PointerIcon.getSystemIcon(mContext, PointerIcon.TYPE_ARROW);
    }
    // Check what the child under the pointer says about the pointer.
    final int childrenCount = mChildrenCount;
    if (childrenCount != 0) {
        final ArrayList<View> preorderedList = buildOrderedChildList();
        final boolean customOrder = preorderedList == null
                && isChildrenDrawingOrderEnabled();
        final View[] children = mChildren;
        for (int i = childrenCount - 1; i >= 0; i--) {
            final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
            final View child = getAndVerifyPreorderedView(preorderedList, children, childIndex);
            if (!canViewReceivePointerEvents(child)
                    || !isTransformedTouchPointInView(x, y, child, null)) {
                continue;
            }
            final PointerIcon pointerIcon =
                    dispatchResolvePointerIcon(event, pointerIndex, child);
            if (pointerIcon != null) {
                if (preorderedList != null) preorderedList.clear();
                return pointerIcon;
            }
        }
        if (preorderedList != null) preorderedList.clear();
    }

    // The pointer is not a child or the child has no preferences, returning the default
    // implementation.
    return super.onResolvePointerIcon(event, pointerIndex);
}

private PointerIcon dispatchResolvePointerIcon(MotionEvent event, int pointerIndex,
        View child) {
    final PointerIcon pointerIcon;
    if (!child.hasIdentityMatrix()) {
        MotionEvent transformedEvent = getTransformedMotionEvent(event, child);
        pointerIcon = child.onResolvePointerIcon(transformedEvent, pointerIndex);
        transformedEvent.recycle();
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        event.offsetLocation(offsetX, offsetY);
        pointerIcon = child.onResolvePointerIcon(event, pointerIndex);
        event.offsetLocation(-offsetX, -offsetY);
    }
    return pointerIcon;
}

private int getAndVerifyPreorderedIndex(int childrenCount, int i, boolean customOrder) {
    final int childIndex;
    if (customOrder) {
        final int childIndex1 = getChildDrawingOrder(childrenCount, i);
```

```
        if (childIndex1 >= childrenCount) {
            throw new IndexOutOfBoundsException("getChildDrawingOrder() "
                    + "returned invalid index " + childIndex1
                    + " (child count is " + childrenCount + ")");
        }
        childIndex = childIndex1;
    } else {
        childIndex = i;
    }
    return childIndex;
}

@SuppressWarnings({"ConstantConditions"})
@Override
protected boolean dispatchHoverEvent(MotionEvent event) {
    final int action = event.getAction();

    // First check whether the view group wants to intercept the hover event.
    final boolean interceptHover = onInterceptHoverEvent(event);
    event.setAction(action); // restore action in case it was changed

    MotionEvent eventNoHistory = event;
    boolean handled = false;

    // Send events to the hovered children and build a new list of hover targets until
    // one is found that handles the event.
    HoverTarget firstOldHoverTarget = mFirstHoverTarget;
    mFirstHoverTarget = null;
    if (!interceptHover && action != MotionEvent.ACTION_HOVER_EXIT) {
        final float x = event.getX();
        final float y = event.getY();
        final int childrenCount = mChildrenCount;
        if (childrenCount != 0) {
            final ArrayList<View> preorderedList = buildOrderedChildList();
            final boolean customOrder = preorderedList == null
                    && isChildrenDrawingOrderEnabled();
            final View[] children = mChildren;
            HoverTarget lastHoverTarget = null;
            for (int i = childrenCount - 1; i >= 0; i--) {
                final int childIndex = getAndVerifyPreorderedIndex(
                        childrenCount, i, customOrder);
                final View child = getAndVerifyPreorderedView(
                        preorderedList, children, childIndex);
                if (!canViewReceivePointerEvents(child)
                        || !isTransformedTouchPointInView(x, y, child, null)) {
                    continue;
                }

                // Obtain a hover target for this child.  Dequeue it from the
                // old hover target list if the child was previously hovered.
                HoverTarget hoverTarget = firstOldHoverTarget;
                final boolean wasHovered;
                for (HoverTarget predecessor = null; ;) {
                    if (hoverTarget == null) {
                        hoverTarget = HoverTarget.obtain(child);
                        wasHovered = false;
                        break;
                    }

                    if (hoverTarget.child == child) {
                        if (predecessor != null) {
                            predecessor.next = hoverTarget.next;
                        } else {
                            firstOldHoverTarget = hoverTarget.next;
                        }
                        hoverTarget.next = null;
                        wasHovered = true;
                        break;
                    }

                    predecessor = hoverTarget;
                    hoverTarget = hoverTarget.next;
                }

                // Enqueue the hover target onto the new hover target list.
                if (lastHoverTarget != null) {
                    lastHoverTarget.next = hoverTarget;
                } else {
                    mFirstHoverTarget = hoverTarget;
                }
                lastHoverTarget = hoverTarget;
```

```
            // Dispatch the event to the child.
            if (action == MotionEvent.ACTION_HOVER_ENTER) {
                if (!wasHovered) {
                    // Send the enter as is.
                    handled |= dispatchTransformedGenericPointerEvent(
                            event, child); // enter
                }
            } else if (action == MotionEvent.ACTION_HOVER_MOVE) {
                if (!wasHovered) {
                    // Synthesize an enter from a move.
                    eventNoHistory = obtainMotionEventNoHistoryOrSelf(eventNoHistory);
                    eventNoHistory.setAction(MotionEvent.ACTION_HOVER_ENTER);
                    handled |= dispatchTransformedGenericPointerEvent(
                            eventNoHistory, child); // enter
                    eventNoHistory.setAction(action);

                    handled |= dispatchTransformedGenericPointerEvent(
                            eventNoHistory, child); // move
                } else {
                    // Send the move as is.
                    handled |= dispatchTransformedGenericPointerEvent(event, child);
                }
            }
            if (handled) {
                break;
            }
        }
        if (preorderedList != null) preorderedList.clear();
    }
}

// Send exit events to all previously hovered children that are no longer hovered.
while (firstOldHoverTarget != null) {
    final View child = firstOldHoverTarget.child;

    // Exit the old hovered child.
    if (action == MotionEvent.ACTION_HOVER_EXIT) {
        // Send the exit as is.
        handled |= dispatchTransformedGenericPointerEvent(
                event, child); // exit
    } else {
        // Synthesize an exit from a move or enter.
        // Ignore the result because hover focus has moved to a different view.
        if (action == MotionEvent.ACTION_HOVER_MOVE) {
            final boolean hoverExitPending = event.isHoverExitPending();
            event.setHoverExitPending(true);
            dispatchTransformedGenericPointerEvent(
                    event, child); // move
            event.setHoverExitPending(hoverExitPending);
        }
        eventNoHistory = obtainMotionEventNoHistoryOrSelf(eventNoHistory);
        eventNoHistory.setAction(MotionEvent.ACTION_HOVER_EXIT);
        dispatchTransformedGenericPointerEvent(
                eventNoHistory, child); // exit
        eventNoHistory.setAction(action);
    }

    final HoverTarget nextOldHoverTarget = firstOldHoverTarget.next;
    firstOldHoverTarget.recycle();
    firstOldHoverTarget = nextOldHoverTarget;
}

// Send events to the view group itself if no children have handled it and the view group
// itself is not currently being hover-exited.
boolean newHoveredSelf = !handled &&
        (action != MotionEvent.ACTION_HOVER_EXIT) && !event.isHoverExitPending();
if (newHoveredSelf == mHoveredSelf) {
    if (newHoveredSelf) {
        // Send event to the view group as before.
        handled |= super.dispatchHoverEvent(event);
    }
} else {
    if (mHoveredSelf) {
        // Exit the view group.
        if (action == MotionEvent.ACTION_HOVER_EXIT) {
            // Send the exit as is.
            handled |= super.dispatchHoverEvent(event); // exit
        } else {
            // Synthesize an exit from a move or enter.
            // Ignore the result because hover focus is moving to a different view.
            if (action == MotionEvent.ACTION_HOVER_MOVE) {
                super.dispatchHoverEvent(event); // move
```

```
                }
                eventNoHistory = obtainMotionEventNoHistoryOrSelf(eventNoHistory);
                eventNoHistory.setAction(MotionEvent.ACTION_HOVER_EXIT);
                super.dispatchHoverEvent(eventNoHistory); // exit
                eventNoHistory.setAction(action);
            }
            mHoveredSelf = false;
        }

        if (newHoveredSelf) {
            // Enter the view group.
            if (action == MotionEvent.ACTION_HOVER_ENTER) {
                // Send the enter as is.
                handled |= super.dispatchHoverEvent(event); // enter
                mHoveredSelf = true;
            } else if (action == MotionEvent.ACTION_HOVER_MOVE) {
                // Synthesize an enter from a move.
                eventNoHistory = obtainMotionEventNoHistoryOrSelf(eventNoHistory);
                eventNoHistory.setAction(MotionEvent.ACTION_HOVER_ENTER);
                handled |= super.dispatchHoverEvent(eventNoHistory); // enter
                eventNoHistory.setAction(action);

                handled |= super.dispatchHoverEvent(eventNoHistory); // move
                mHoveredSelf = true;
            }
        }
    }

    // Recycle the copy of the event that we made.
    if (eventNoHistory != event) {
        eventNoHistory.recycle();
    }

    // Done.
    return handled;
}

private void exitHoverTargets() {
    if (mHoveredSelf || mFirstHoverTarget != null) {
        final long now = SystemClock.uptimeMillis();
        MotionEvent event = MotionEvent.obtain(now, now,
                MotionEvent.ACTION_HOVER_EXIT, 0.0f, 0.0f, 0);
        event.setSource(InputDevice.SOURCE_TOUCHSCREEN);
        dispatchHoverEvent(event);
        event.recycle();
    }
}

private void cancelHoverTarget(View view) {
    HoverTarget predecessor = null;
    HoverTarget target = mFirstHoverTarget;
    while (target != null) {
        final HoverTarget next = target.next;
        if (target.child == view) {
            if (predecessor == null) {
                mFirstHoverTarget = next;
            } else {
                predecessor.next = next;
            }
            target.recycle();

            final long now = SystemClock.uptimeMillis();
            MotionEvent event = MotionEvent.obtain(now, now,
                    MotionEvent.ACTION_HOVER_EXIT, 0.0f, 0.0f, 0);
            event.setSource(InputDevice.SOURCE_TOUCHSCREEN);
            view.dispatchHoverEvent(event);
            event.recycle();
            return;
        }
        predecessor = target;
        target = next;
    }
}

@Override
boolean dispatchTooltipHoverEvent(MotionEvent event) {
    final int action = event.getAction();
    switch (action) {
        case MotionEvent.ACTION_HOVER_ENTER:
            break;

        case MotionEvent.ACTION_HOVER_MOVE:
```

```
            View newTarget = null;

            // Check what the child under the pointer says about the tooltip.
            final int childrenCount = mChildrenCount;
            if (childrenCount != 0) {
                final float x = event.getX();
                final float y = event.getY();

                final ArrayList<View> preorderedList = buildOrderedChildList();
                final boolean customOrder = preorderedList == null
                        && isChildrenDrawingOrderEnabled();
                final View[] children = mChildren;
                for (int i = childrenCount - 1; i >= 0; i--) {
                    final int childIndex =
                            getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
                    final View child =
                            getAndVerifyPreorderedView(preorderedList, children, childIndex);
                    if (!canViewReceivePointerEvents(child)
                            || !isTransformedTouchPointInView(x, y, child, null)) {
                        continue;
                    }
                    if (dispatchTooltipHoverEvent(event, child)) {
                        newTarget = child;
                        break;
                    }
                }
                if (preorderedList != null) preorderedList.clear();
            }

            if (mTooltipHoverTarget != newTarget) {
                if (mTooltipHoverTarget != null) {
                    event.setAction(MotionEvent.ACTION_HOVER_EXIT);
                    mTooltipHoverTarget.dispatchTooltipHoverEvent(event);
                    event.setAction(action);
                }
                mTooltipHoverTarget = newTarget;
            }

            if (mTooltipHoverTarget != null) {
                if (mTooltipHoveredSelf) {
                    mTooltipHoveredSelf = false;
                    event.setAction(MotionEvent.ACTION_HOVER_EXIT);
                    super.dispatchTooltipHoverEvent(event);
                    event.setAction(action);
                }
                return true;
            }

            mTooltipHoveredSelf = super.dispatchTooltipHoverEvent(event);
            return mTooltipHoveredSelf;

        case MotionEvent.ACTION_HOVER_EXIT:
            if (mTooltipHoverTarget != null) {
                mTooltipHoverTarget.dispatchTooltipHoverEvent(event);
                mTooltipHoverTarget = null;
            } else if (mTooltipHoveredSelf) {
                super.dispatchTooltipHoverEvent(event);
                mTooltipHoveredSelf = false;
            }
            break;
    }
    return false;
}

private boolean dispatchTooltipHoverEvent(MotionEvent event, View child) {
    final boolean result;
    if (!child.hasIdentityMatrix()) {
        MotionEvent transformedEvent = getTransformedMotionEvent(event, child);
        result = child.dispatchTooltipHoverEvent(transformedEvent);
        transformedEvent.recycle();
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        event.offsetLocation(offsetX, offsetY);
        result = child.dispatchTooltipHoverEvent(event);
        event.offsetLocation(-offsetX, -offsetY);
    }
    return result;
}

private void exitTooltipHoverTargets() {
    if (mTooltipHoveredSelf || mTooltipHoverTarget != null) {
```

```
        final long now = SystemClock.uptimeMillis();
        MotionEvent event = MotionEvent.obtain(now, now,
                MotionEvent.ACTION_HOVER_EXIT, 0.0f, 0.0f, 0);
        event.setSource(InputDevice.SOURCE_TOUCHSCREEN);
        dispatchTooltipHoverEvent(event);
        event.recycle();
    }
}

/** @hide */
@Override
protected boolean hasHoveredChild() {
    return mFirstHoverTarget != null;
}

@Override
public void addChildrenForAccessibility(ArrayList<View> outChildren) {
    if (getAccessibilityNodeProvider() != null) {
        return;
    }
    ChildListForAccessibility children = ChildListForAccessibility.obtain(this, true);
    try {
        final int childrenCount = children.getChildCount();
        for (int i = 0; i < childrenCount; i++) {
            View child = children.getChildAt(i);
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
                if (child.includeForAccessibility()) {
                    outChildren.add(child);
                } else {
                    child.addChildrenForAccessibility(outChildren);
                }
            }
        }
    } finally {
        children.recycle();
    }
}

/**
 * Implement this method to intercept hover events before they are handled
 * by child views.
 * <p>
 * This method is called before dispatching a hover event to a child of
 * the view group or to the view group's own {@link #onHoverEvent} to allow
 * the view group a chance to intercept the hover event.
 * This method can also be used to watch all pointer motions that occur within
 * the bounds of the view group even when the pointer is hovering over
 * a child of the view group rather than over the view group itself.
 * </p><p>
 * The view group can prevent its children from receiving hover events by
 * implementing this method and returning <code>true</code> to indicate
 * that it would like to intercept hover events.  The view group must
 * continuously return <code>true</code> from {@link #onInterceptHoverEvent}
 * for as long as it wishes to continue intercepting hover events from
 * its children.
 * </p><p>
 * Interception preserves the invariant that at most one view can be
 * hovered at a time by transferring hover focus from the currently hovered
 * child to the view group or vice-versa as needed.
 * </p><p>
 * If this method returns <code>true</code> and a child is already hovered, then the
 * child view will first receive a hover exit event and then the view group
 * itself will receive a hover enter event in {@link #onHoverEvent}.
 * Likewise, if this method had previously returned <code>true</code> to intercept hover
 * events and instead returns <code>false</code> while the pointer is hovering
 * within the bounds of one of a child, then the view group will first receive a
 * hover exit event in {@link #onHoverEvent} and then the hovered child will
 * receive a hover enter event.
 * </p><p>
 * The default implementation handles mouse hover on the scroll bars.
 * </p>
 *
 * @param event The motion event that describes the hover.
 * @return True if the view group would like to intercept the hover event
 * and prevent its children from receiving it.
 */
public boolean onInterceptHoverEvent(MotionEvent event) {
    if (event.isFromSource(InputDevice.SOURCE_MOUSE)) {
        final int action = event.getAction();
        final float x = event.getX();
        final float y = event.getY();
        if ((action == MotionEvent.ACTION_HOVER_MOVE
```

```
                || action == MotionEvent.ACTION_HOVER_ENTER) && isOnScrollbar(x, y)) {
            return true;
        }
    }
    return false;
}

private static MotionEvent obtainMotionEventNoHistoryOrSelf(MotionEvent event) {
    if (event.getHistorySize() == 0) {
        return event;
    }
    return MotionEvent.obtainNoHistory(event);
}

@Override
protected boolean dispatchGenericPointerEvent(MotionEvent event) {
    // Send the event to the child under the pointer.
    final int childrenCount = mChildrenCount;
    if (childrenCount != 0) {
        final float x = event.getX();
        final float y = event.getY();

        final ArrayList<View> preorderedList = buildOrderedChildList();
        final boolean customOrder = preorderedList == null
                && isChildrenDrawingOrderEnabled();
        final View[] children = mChildren;
        for (int i = childrenCount - 1; i >= 0; i--) {
            final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
            final View child = getAndVerifyPreorderedView(preorderedList, children, childIndex);
            if (!canViewReceivePointerEvents(child)
                    || !isTransformedTouchPointInView(x, y, child, null)) {
                continue;
            }

            if (dispatchTransformedGenericPointerEvent(event, child)) {
                if (preorderedList != null) preorderedList.clear();
                return true;
            }
        }
        if (preorderedList != null) preorderedList.clear();
    }

    // No child handled the event.  Send it to this view group.
    return super.dispatchGenericPointerEvent(event);
}

@Override
protected boolean dispatchGenericFocusedEvent(MotionEvent event) {
    // Send the event to the focused child or to this view group if it has focus.
    if ((mPrivateFlags & (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS))
            == (PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) {
        return super.dispatchGenericFocusedEvent(event);
    } else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS)
            == PFLAG_HAS_BOUNDS) {
        return mFocused.dispatchGenericMotionEvent(event);
    }
    return false;
}

/**
 * Dispatches a generic pointer event to a child, taking into account
 * transformations that apply to the child.
 *
 * @param event The event to send.
 * @param child The view to send the event to.
 * @return {@code true} if the child handled the event.
 */
private boolean dispatchTransformedGenericPointerEvent(MotionEvent event, View child) {
    boolean handled;
    if (!child.hasIdentityMatrix()) {
        MotionEvent transformedEvent = getTransformedMotionEvent(event, child);
        handled = child.dispatchGenericMotionEvent(transformedEvent);
        transformedEvent.recycle();
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        event.offsetLocation(offsetX, offsetY);
        handled = child.dispatchGenericMotionEvent(event);
        event.offsetLocation(-offsetX, -offsetY);
    }
    return handled;
}
```

```java
/**
 * Returns a MotionEvent that's been transformed into the child's local coordinates.
 *
 * It's the responsibility of the caller to recycle it once they're finished with it.
 * @param event The event to transform.
 * @param child The view whose coordinate space is to be used.
 * @return A copy of the the given MotionEvent, transformed into the given View's coordinate
 *         space.
 */
private MotionEvent getTransformedMotionEvent(MotionEvent event, View child) {
    final float offsetX = mScrollX - child.mLeft;
    final float offsetY = mScrollY - child.mTop;
    final MotionEvent transformedEvent = MotionEvent.obtain(event);
    transformedEvent.offsetLocation(offsetX, offsetY);
    if (!child.hasIdentityMatrix()) {
        transformedEvent.transform(child.getInverseMatrix());
    }
    return transformedEvent;
}

@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(ev, 1);
    }

    // If the event targets the accessibility focused view and this is it, start
    // normal event dispatch. Maybe a descendant is what will handle the click.
    if (ev.isTargetAccessibilityFocus() && isAccessibilityFocusedViewOrHost()) {
        ev.setTargetAccessibilityFocus(false);
    }

    boolean handled = false;
    if (onFilterTouchEventForSecurity(ev)) {
        final int action = ev.getAction();
        final int actionMasked = action & MotionEvent.ACTION_MASK;

        // Handle an initial down.
        if (actionMasked == MotionEvent.ACTION_DOWN) {
            // Throw away all previous state when starting a new touch gesture.
            // The framework may have dropped the up or cancel event for the previous gesture
            // due to an app switch, ANR, or some other state change.
            cancelAndClearTouchTargets(ev);
            resetTouchState();
        }

        // Check for interception.
        final boolean intercepted;
        if (actionMasked == MotionEvent.ACTION_DOWN
                || mFirstTouchTarget != null) {
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                intercepted = onInterceptTouchEvent(ev);
                ev.setAction(action); // restore action in case it was changed
            } else {
                intercepted = false;
            }
        } else {
            // There are no touch targets and this action is not an initial down
            // so this view group continues to intercept touches.
            intercepted = true;
        }

        // If intercepted, start normal event dispatch. Also if there is already
        // a view that is handling the gesture, do normal event dispatch.
        if (intercepted || mFirstTouchTarget != null) {
            ev.setTargetAccessibilityFocus(false);
        }

        // Check for cancelation.
        final boolean canceled = resetCancelNextUpFlag(this)
                || actionMasked == MotionEvent.ACTION_CANCEL;

        // Update list of touch targets for pointer down, if needed.
        final boolean split = (mGroupFlags & FLAG_SPLIT_MOTION_EVENTS) != 0;
        TouchTarget newTouchTarget = null;
        boolean alreadyDispatchedToNewTouchTarget = false;
        if (!canceled && !intercepted) {

            // If the event is targeting accessiiblity focus we give it to the
            // view that has accessibility focus and if it does not handle it
```

```
// we clear the flag and dispatch the event to all children as usual.
// We are looking up the accessibility focused host to avoid keeping
// state since these events are very rare.
View childWithAccessibilityFocus = ev.isTargetAccessibilityFocus()
        ? findChildWithAccessibilityFocus() : null;

if (actionMasked == MotionEvent.ACTION_DOWN
        || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
        || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
    final int actionIndex = ev.getActionIndex(); // always 0 for down
    final int idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex)
            : TouchTarget.ALL_POINTER_IDS;

    // Clean up earlier touch targets for this pointer id in case they
    // have become out of sync.
    removePointersFromTouchTargets(idBitsToAssign);

    final int childrenCount = mChildrenCount;
    if (newTouchTarget == null && childrenCount != 0) {
        final float x = ev.getX(actionIndex);
        final float y = ev.getY(actionIndex);
        // Find a child that can receive the event.
        // Scan children from front to back.
        final ArrayList<View> preorderedList = buildTouchDispatchChildList();
        final boolean customOrder = preorderedList == null
                && isChildrenDrawingOrderEnabled();
        final View[] children = mChildren;
        for (int i = childrenCount - 1; i >= 0; i--) {
            final int childIndex = getAndVerifyPreorderedIndex(
                    childrenCount, i, customOrder);
            final View child = getAndVerifyPreorderedView(
                    preorderedList, children, childIndex);

            // If there is a view that has accessibility focus we want it
            // to get the event first and if not handled we will perform a
            // normal dispatch. We may do a double iteration but this is
            // safer given the timeframe.
            if (childWithAccessibilityFocus != null) {
                if (childWithAccessibilityFocus != child) {
                    continue;
                }
                childWithAccessibilityFocus = null;
                i = childrenCount - 1;
            }

            if (!canViewReceivePointerEvents(child)
                    || !isTransformedTouchPointInView(x, y, child, null)) {
                ev.setTargetAccessibilityFocus(false);
                continue;
            }

            newTouchTarget = getTouchTarget(child);
            if (newTouchTarget != null) {
                // Child is already receiving touch within its bounds.
                // Give it the new pointer in addition to the ones it is handling.
                newTouchTarget.pointerIdBits |= idBitsToAssign;
                break;
            }

            resetCancelNextUpFlag(child);
            if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                // Child wants to receive touch within its bounds.
                mLastTouchDownTime = ev.getDownTime();
                if (preorderedList != null) {
                    // childIndex points into presorted list, find original index
                    for (int j = 0; j < childrenCount; j++) {
                        if (children[childIndex] == mChildren[j]) {
                            mLastTouchDownIndex = j;
                            break;
                        }
                    }
                } else {
                    mLastTouchDownIndex = childIndex;
                }
                mLastTouchDownX = ev.getX();
                mLastTouchDownY = ev.getY();
                newTouchTarget = addTouchTarget(child, idBitsToAssign);
                alreadyDispatchedToNewTouchTarget = true;
                break;
            }

            // The accessibility focus didn't handle the event, so clear
```

```
                        // the flag and do a normal dispatch to all children.
                        ev.setTargetAccessibilityFocus(false);
                    }
                    if (preorderedList != null) preorderedList.clear();
                }

                if (newTouchTarget == null && mFirstTouchTarget != null) {
                    // Did not find a child to receive the event.
                    // Assign the pointer to the least recently added target.
                    newTouchTarget = mFirstTouchTarget;
                    while (newTouchTarget.next != null) {
                        newTouchTarget = newTouchTarget.next;
                    }
                    newTouchTarget.pointerIdBits |= idBitsToAssign;
                }
            }
        }

        // Dispatch to touch targets.
        if (mFirstTouchTarget == null) {
            // No touch targets so treat this as an ordinary view.
            handled = dispatchTransformedTouchEvent(ev, canceled, null,
                    TouchTarget.ALL_POINTER_IDS);
        } else {
            // Dispatch to touch targets, excluding the new touch target if we already
            // dispatched to it.  Cancel touch targets if necessary.
            TouchTarget predecessor = null;
            TouchTarget target = mFirstTouchTarget;
            while (target != null) {
                final TouchTarget next = target.next;
                if (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {
                    handled = true;
                } else {
                    final boolean cancelChild = resetCancelNextUpFlag(target.child)
                            || intercepted;
                    if (dispatchTransformedTouchEvent(ev, cancelChild,
                            target.child, target.pointerIdBits)) {
                        handled = true;
                    }
                    if (cancelChild) {
                        if (predecessor == null) {
                            mFirstTouchTarget = next;
                        } else {
                            predecessor.next = next;
                        }
                        target.recycle();
                        target = next;
                        continue;
                    }
                }
                predecessor = target;
                target = next;
            }
        }

        // Update list of touch targets for pointer up or cancel, if needed.
        if (canceled
                || actionMasked == MotionEvent.ACTION_UP
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
            resetTouchState();
        } else if (split && actionMasked == MotionEvent.ACTION_POINTER_UP) {
            final int actionIndex = ev.getActionIndex();
            final int idBitsToRemove = 1 << ev.getPointerId(actionIndex);
            removePointersFromTouchTargets(idBitsToRemove);
        }
    }

    if (!handled && mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(ev, 1);
    }
    return handled;
}

/**
 * Provide custom ordering of views in which the touch will be dispatched.
 *
 * This is called within a tight loop, so you are not allowed to allocate objects, including
 * the return array. Instead, you should return a pre-allocated list that will be cleared
 * after the dispatch is finished.
 * @hide
 */
public ArrayList<View> buildTouchDispatchChildList() {
```

```
    return buildOrderedChildList();
}

/**
 * Finds the child which has accessibility focus.
 *
 * @return The child that has focus.
 */
private View findChildWithAccessibilityFocus() {
    ViewRootImpl viewRoot = getViewRootImpl();
    if (viewRoot == null) {
        return null;
    }

    View current = viewRoot.getAccessibilityFocusedHost();
    if (current == null) {
        return null;
    }

    ViewParent parent = current.getParent();
    while (parent instanceof View) {
        if (parent == this) {
            return current;
        }
        current = (View) parent;
        parent = current.getParent();
    }

    return null;
}

/**
 * Resets all touch state in preparation for a new cycle.
 */
private void resetTouchState() {
    clearTouchTargets();
    resetCancelNextUpFlag(this);
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
    mNestedScrollAxes = SCROLL_AXIS_NONE;
}

/**
 * Resets the cancel next up flag.
 * Returns true if the flag was previously set.
 */
private static boolean resetCancelNextUpFlag(@NonNull View view) {
    if ((view.mPrivateFlags & PFLAG_CANCEL_NEXT_UP_EVENT) != 0) {
        view.mPrivateFlags &= ~PFLAG_CANCEL_NEXT_UP_EVENT;
        return true;
    }
    return false;
}

/**
 * Clears all touch targets.
 */
private void clearTouchTargets() {
    TouchTarget target = mFirstTouchTarget;
    if (target != null) {
        do {
            TouchTarget next = target.next;
            target.recycle();
            target = next;
        } while (target != null);
        mFirstTouchTarget = null;
    }
}

/**
 * Cancels and clears all touch targets.
 */
private void cancelAndClearTouchTargets(MotionEvent event) {
    if (mFirstTouchTarget != null) {
        boolean syntheticEvent = false;
        if (event == null) {
            final long now = SystemClock.uptimeMillis();
            event = MotionEvent.obtain(now, now,
                    MotionEvent.ACTION_CANCEL, 0.0f, 0.0f, 0);
            event.setSource(InputDevice.SOURCE_TOUCHSCREEN);
            syntheticEvent = true;
        }
```

```
        for (TouchTarget target = mFirstTouchTarget; target != null; target = target.next) {
            resetCancelNextUpFlag(target.child);
            dispatchTransformedTouchEvent(event, true, target.child, target.pointerIdBits);
        }
        clearTouchTargets();

        if (syntheticEvent) {
            event.recycle();
        }
    }
}

/**
 * Gets the touch target for specified child view.
 * Returns null if not found.
 */
private TouchTarget getTouchTarget(@NonNull View child) {
    for (TouchTarget target = mFirstTouchTarget; target != null; target = target.next) {
        if (target.child == child) {
            return target;
        }
    }
    return null;
}

/**
 * Adds a touch target for specified child to the beginning of the list.
 * Assumes the target child is not already present.
 */
private TouchTarget addTouchTarget(@NonNull View child, int pointerIdBits) {
    final TouchTarget target = TouchTarget.obtain(child, pointerIdBits);
    target.next = mFirstTouchTarget;
    mFirstTouchTarget = target;
    return target;
}

/**
 * Removes the pointer ids from consideration.
 */
private void removePointersFromTouchTargets(int pointerIdBits) {
    TouchTarget predecessor = null;
    TouchTarget target = mFirstTouchTarget;
    while (target != null) {
        final TouchTarget next = target.next;
        if ((target.pointerIdBits & pointerIdBits) != 0) {
            target.pointerIdBits &= ~pointerIdBits;
            if (target.pointerIdBits == 0) {
                if (predecessor == null) {
                    mFirstTouchTarget = next;
                } else {
                    predecessor.next = next;
                }
                target.recycle();
                target = next;
                continue;
            }
        }
        predecessor = target;
        target = next;
    }
}

private void cancelTouchTarget(View view) {
    TouchTarget predecessor = null;
    TouchTarget target = mFirstTouchTarget;
    while (target != null) {
        final TouchTarget next = target.next;
        if (target.child == view) {
            if (predecessor == null) {
                mFirstTouchTarget = next;
            } else {
                predecessor.next = next;
            }
            target.recycle();

            final long now = SystemClock.uptimeMillis();
            MotionEvent event = MotionEvent.obtain(now, now,
                    MotionEvent.ACTION_CANCEL, 0.0f, 0.0f, 0);
            event.setSource(InputDevice.SOURCE_TOUCHSCREEN);
            view.dispatchTouchEvent(event);
            event.recycle();
            return;
```

```
            }
            predecessor = target;
            target = next;
        }
    }

    /**
     * Returns true if a child view can receive pointer events.
     * @hide
     */
    private static boolean canViewReceivePointerEvents(@NonNull View child) {
        return (child.mViewFlags & VISIBILITY_MASK) == VISIBLE
                || child.getAnimation() != null;
    }

    private float[] getTempPoint() {
        if (mTempPoint == null) {
            mTempPoint = new float[2];
        }
        return mTempPoint;
    }

    /**
     * Returns true if a child view contains the specified point when transformed
     * into its coordinate space.
     * Child must not be null.
     * @hide
     */
    protected boolean isTransformedTouchPointInView(float x, float y, View child,
            PointF outLocalPoint) {
        final float[] point = getTempPoint();
        point[0] = x;
        point[1] = y;
        transformPointToViewLocal(point, child);
        final boolean isInView = child.pointInView(point[0], point[1]);
        if (isInView && outLocalPoint != null) {
            outLocalPoint.set(point[0], point[1]);
        }
        return isInView;
    }

    /**
     * @hide
     */
    public void transformPointToViewLocal(float[] point, View child) {
        point[0] += mScrollX - child.mLeft;
        point[1] += mScrollY - child.mTop;

        if (!child.hasIdentityMatrix()) {
            child.getInverseMatrix().mapPoints(point);
        }
    }

    /**
     * Transforms a motion event into the coordinate space of a particular child view,
     * filters out irrelevant pointer ids, and overrides its action if necessary.
     * If child is null, assumes the MotionEvent will be sent to this ViewGroup instead.
     */
    private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
            View child, int desiredPointerIdBits) {
        final boolean handled;

        // Canceling motions is a special case.  We don't need to perform any transformations
        // or filtering.  The important part is the action, not the contents.
        final int oldAction = event.getAction();
        if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
            event.setAction(MotionEvent.ACTION_CANCEL);
            if (child == null) {
                handled = super.dispatchTouchEvent(event);
            } else {
                handled = child.dispatchTouchEvent(event);
            }
            event.setAction(oldAction);
            return handled;
        }

        // Calculate the number of pointers to deliver.
        final int oldPointerIdBits = event.getPointerIdBits();
        final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;

        // If for some reason we ended up in an inconsistent state where it looks like we
        // might produce a motion event with no pointers in it, then drop the event.
```

```
    if (newPointerIdBits == 0) {
        return false;
    }

    // If the number of pointers is the same and we don't need to perform any fancy
    // irreversible transformations, then we can reuse the motion event for this
    // dispatch as long as we are careful to revert any changes we make.
    // Otherwise we need to make a copy.
    final MotionEvent transformedEvent;
    if (newPointerIdBits == oldPointerIdBits) {
        if (child == null || child.hasIdentityMatrix()) {
            if (child == null) {
                handled = super.dispatchTouchEvent(event);
            } else {
                final float offsetX = mScrollX - child.mLeft;
                final float offsetY = mScrollY - child.mTop;
                event.offsetLocation(offsetX, offsetY);

                handled = child.dispatchTouchEvent(event);

                event.offsetLocation(-offsetX, -offsetY);
            }
            return handled;
        }
        transformedEvent = MotionEvent.obtain(event);
    } else {
        transformedEvent = event.split(newPointerIdBits);
    }

    // Perform any necessary transformations and dispatch.
    if (child == null) {
        handled = super.dispatchTouchEvent(transformedEvent);
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        transformedEvent.offsetLocation(offsetX, offsetY);
        if (! child.hasIdentityMatrix()) {
            transformedEvent.transform(child.getInverseMatrix());
        }

        handled = child.dispatchTouchEvent(transformedEvent);
    }

    // Done.
    transformedEvent.recycle();
    return handled;
}

/**
 * Enable or disable the splitting of MotionEvents to multiple children during touch event
 * dispatch. This behavior is enabled by default for applications that target an
 * SDK version of {@link Build.VERSION_CODES#HONEYCOMB} or newer.
 *
 * <p>When this option is enabled MotionEvents may be split and dispatched to different child
 * views depending on where each pointer initially went down. This allows for user interactions
 * such as scrolling two panes of content independently, chording of buttons, and performing
 * independent gestures on different pieces of content.
 *
 * @param split <code>true</code> to allow MotionEvents to be split and dispatched to multiple
 *              child views. <code>false</code> to only allow one child view to be the target of
 *              any MotionEvent received by this ViewGroup.
 * @attr ref android.R.styleable#ViewGroup_splitMotionEvents
 */
public void setMotionEventSplittingEnabled(boolean split) {
    // TODO Applications really shouldn't change this setting mid-touch event,
    // but perhaps this should handle that case and send ACTION_CANCELs to any child views
    // with gestures in progress when this is changed.
    if (split) {
        mGroupFlags |= FLAG_SPLIT_MOTION_EVENTS;
    } else {
        mGroupFlags &= ~FLAG_SPLIT_MOTION_EVENTS;
    }
}

/**
 * Returns true if MotionEvents dispatched to this ViewGroup can be split to multiple children.
 * @return true if MotionEvents dispatched to this ViewGroup can be split to multiple children.
 */
public boolean isMotionEventSplittingEnabled() {
    return (mGroupFlags & FLAG_SPLIT_MOTION_EVENTS) == FLAG_SPLIT_MOTION_EVENTS;
}
```

```java
/**
 * Returns true if this ViewGroup should be considered as a single entity for removal
 * when executing an Activity transition. If this is false, child elements will move
 * individually during the transition.
 *
 * @return True if the ViewGroup should be acted on together during an Activity transition.
 * The default value is true when there is a non-null background or if
 * {@link #getTransitionName()} is not null or if a
 * non-null {@link android.view.ViewOutlineProvider} other than
 * {@link android.view.ViewOutlineProvider#BACKGROUND} was given to
 * {@link #setOutlineProvider(ViewOutlineProvider)} and false otherwise.
 */
public boolean isTransitionGroup() {
    if ((mGroupFlags & FLAG_IS_TRANSITION_GROUP_SET) != 0) {
        return ((mGroupFlags & FLAG_IS_TRANSITION_GROUP) != 0);
    } else {
        final ViewOutlineProvider outlineProvider = getOutlineProvider();
        return getBackground() != null || getTransitionName() != null ||
                (outlineProvider != null && outlineProvider != ViewOutlineProvider.BACKGROUND);
    }
}

/**
 * Changes whether or not this ViewGroup should be treated as a single entity during
 * Activity Transitions.
 * @param isTransitionGroup Whether or not the ViewGroup should be treated as a unit
 *                          in Activity transitions. If false, the ViewGroup won't transition,
 *                          only its children. If true, the entire ViewGroup will transition
 *                          together.
 * @see android.app.ActivityOptions#makeSceneTransitionAnimation(android.app.Activity,
 * android.util.Pair[])
 */
public void setTransitionGroup(boolean isTransitionGroup) {
    mGroupFlags |= FLAG_IS_TRANSITION_GROUP_SET;
    if (isTransitionGroup) {
        mGroupFlags |= FLAG_IS_TRANSITION_GROUP;
    } else {
        mGroupFlags &= ~FLAG_IS_TRANSITION_GROUP;
    }
}

@Override
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {

    if (disallowIntercept == ((mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0)) {
        // We're already in this state, assume our ancestors are too
        return;
    }

    if (disallowIntercept) {
        mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
    } else {
        mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
    }

    // Pass it up to our parent
    if (mParent != null) {
        mParent.requestDisallowInterceptTouchEvent(disallowIntercept);
    }
}

/**
 * Implement this method to intercept all touch screen motion events.  This
 * allows you to watch events as they are dispatched to your children, and
 * take ownership of the current gesture at any point.
 *
 * <p>Using this function takes some care, as it has a fairly complicated
 * interaction with {@link View#onTouchEvent(MotionEvent)
 * View.onTouchEvent(MotionEvent)}, and using it requires implementing
 * that method as well as this one in the correct way.  Events will be
 * received in the following order:
 *
 * <ol>
 * <li> You will receive the down event here.
 * <li> The down event will be handled either by a child of this view
 * group, or given to your own onTouchEvent() method to handle; this means
 * you should implement onTouchEvent() to return true, so you will
 * continue to see the rest of the gesture (instead of looking for
 * a parent view to handle it).  Also, by returning true from
 * onTouchEvent(), you will not receive any following
 * events in onInterceptTouchEvent() and all touch processing must
 * happen in onTouchEvent() like normal.
```

```
     * <li> For as long as you return false from this function, each following
     * event (up to and including the final up) will be delivered first here
     * and then to the target's onTouchEvent().
     * <li> If you return true from here, you will not receive any
     * following events: the target view will receive the same event but
     * with the action {@link MotionEvent#ACTION_CANCEL}, and all further
     * events will be delivered to your onTouchEvent() method and no longer
     * appear here.
     * </ol>
     *
     * @param ev The motion event being dispatched down the hierarchy.
     * @return Return true to steal motion events from the children and have
     * them dispatched to this ViewGroup through onTouchEvent().
     * The current target will receive an ACTION_CANCEL event, and no further
     * messages will be delivered here.
     */
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        if (ev.isFromSource(InputDevice.SOURCE_MOUSE)
                && ev.getAction() == MotionEvent.ACTION_DOWN
                && ev.isButtonPressed(MotionEvent.BUTTON_PRIMARY)
                && isOnScrollbarThumb(ev.getX(), ev.getY())) {
            return true;
        }
        return false;
    }

    /**
     * {@inheritDoc}
     *
     * Looks for a view to give focus to respecting the setting specified by
     * {@link #getDescendantFocusability()}.
     *
     * Uses {@link #onRequestFocusInDescendants(int, android.graphics.Rect)} to
     * find focus within the children of this group when appropriate.
     *
     * @see #FOCUS_BEFORE_DESCENDANTS
     * @see #FOCUS_AFTER_DESCENDANTS
     * @see #FOCUS_BLOCK_DESCENDANTS
     * @see #onRequestFocusInDescendants(int, android.graphics.Rect)
     */
    @Override
    public boolean requestFocus(int direction, Rect previouslyFocusedRect) {
        if (DBG) {
            System.out.println(this + " ViewGroup.requestFocus direction="
                    + direction);
        }
        int descendantFocusability = getDescendantFocusability();

        switch (descendantFocusability) {
            case FOCUS_BLOCK_DESCENDANTS:
                return super.requestFocus(direction, previouslyFocusedRect);
            case FOCUS_BEFORE_DESCENDANTS: {
                final boolean took = super.requestFocus(direction, previouslyFocusedRect);
                return took ? took : onRequestFocusInDescendants(direction, previouslyFocusedRect);
            }
            case FOCUS_AFTER_DESCENDANTS: {
                final boolean took = onRequestFocusInDescendants(direction, previouslyFocusedRect);
                return took ? took : super.requestFocus(direction, previouslyFocusedRect);
            }
            default:
                throw new IllegalStateException("descendant focusability must be "
                        + "one of FOCUS_BEFORE_DESCENDANTS, FOCUS_AFTER_DESCENDANTS, FOCUS_BLOCK_DESCENDANTS "
                        + "but is " + descendantFocusability);
        }
    }

    /**
     * Look for a descendant to call {@link View#requestFocus} on.
     * Called by {@link ViewGroup#requestFocus(int, android.graphics.Rect)}
     * when it wants to request focus within its children.  Override this to
     * customize how your {@link ViewGroup} requests focus within its children.
     * @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 focus was taken.
     */
    @SuppressWarnings({"ConstantConditions"})
    protected boolean onRequestFocusInDescendants(int direction,
            Rect previouslyFocusedRect) {
        int index;
        int increment;
```

```
    int end;
    int count = mChildrenCount;
    if ((direction & FOCUS_FORWARD) != 0) {
        index = 0;
        increment = 1;
        end = count;
    } else {
        index = count - 1;
        increment = -1;
        end = -1;
    }
    final View[] children = mChildren;
    for (int i = index; i != end; i += increment) {
        View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
            if (child.requestFocus(direction, previouslyFocusedRect)) {
                return true;
            }
        }
    }
    return false;
}

@Override
public boolean restoreDefaultFocus() {
    if (mDefaultFocus != null
            && getDescendantFocusability() != FOCUS_BLOCK_DESCENDANTS
            && (mDefaultFocus.mViewFlags & VISIBILITY_MASK) == VISIBLE
            && mDefaultFocus.restoreDefaultFocus()) {
        return true;
    }
    return super.restoreDefaultFocus();
}

/**
 * @hide
 */
@TestApi
@Override
public boolean restoreFocusInCluster(@FocusRealDirection int direction) {
    // Allow cluster-navigation to enter touchscreenBlocksFocus ViewGroups.
    if (isKeyboardNavigationCluster()) {
        final boolean blockedFocus = getTouchscreenBlocksFocus();
        try {
            setTouchscreenBlocksFocusNoRefocus(false);
            return restoreFocusInClusterInternal(direction);
        } finally {
            setTouchscreenBlocksFocusNoRefocus(blockedFocus);
        }
    } else {
        return restoreFocusInClusterInternal(direction);
    }
}

private boolean restoreFocusInClusterInternal(@FocusRealDirection int direction) {
    if (mFocusedInCluster != null && getDescendantFocusability() != FOCUS_BLOCK_DESCENDANTS
            && (mFocusedInCluster.mViewFlags & VISIBILITY_MASK) == VISIBLE
            && mFocusedInCluster.restoreFocusInCluster(direction)) {
        return true;
    }
    return super.restoreFocusInCluster(direction);
}

/**
 * @hide
 */
@Override
public boolean restoreFocusNotInCluster() {
    if (mFocusedInCluster != null) {
        // since clusters don't nest; we can assume that a non-null mFocusedInCluster
        // will refer to a view not-in a cluster.
        return restoreFocusInCluster(View.FOCUS_DOWN);
    }
    if (isKeyboardNavigationCluster() || (mViewFlags & VISIBILITY_MASK) != VISIBLE) {
        return false;
    }
    int descendentFocusability = getDescendantFocusability();
    if (descendentFocusability == FOCUS_BLOCK_DESCENDANTS) {
        return super.requestFocus(FOCUS_DOWN, null);
    }
    if (descendentFocusability == FOCUS_BEFORE_DESCENDANTS
            && super.requestFocus(FOCUS_DOWN, null)) {
```

```
            return true;
        }
        for (int i = 0; i < mChildrenCount; ++i) {
            View child = mChildren[i];
            if (!child.isKeyboardNavigationCluster()
                    && child.restoreFocusNotInCluster()) {
                return true;
            }
        }
        if (descendentFocusability == FOCUS_AFTER_DESCENDANTS && !hasFocusableChild(false)) {
            return super.requestFocus(FOCUS_DOWN, null);
        }
        return false;
    }

    /**
     * {@inheritDoc}
     *
     * @hide
     */
    @Override
    public void dispatchStartTemporaryDetach() {
        super.dispatchStartTemporaryDetach();
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            children[i].dispatchStartTemporaryDetach();
        }
    }

    /**
     * {@inheritDoc}
     *
     * @hide
     */
    @Override
    public void dispatchFinishTemporaryDetach() {
        super.dispatchFinishTemporaryDetach();
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            children[i].dispatchFinishTemporaryDetach();
        }
    }

    @Override
    void dispatchAttachedToWindow(AttachInfo info, int visibility) {
        mGroupFlags |= FLAG_PREVENT_DISPATCH_ATTACHED_TO_WINDOW;
        super.dispatchAttachedToWindow(info, visibility);
        mGroupFlags &= ~FLAG_PREVENT_DISPATCH_ATTACHED_TO_WINDOW;

        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            final View child = children[i];
            child.dispatchAttachedToWindow(info,
                    combineVisibility(visibility, child.getVisibility()));
        }
        final int transientCount = mTransientIndices == null ? 0 : mTransientIndices.size();
        for (int i = 0; i < transientCount; ++i) {
            View view = mTransientViews.get(i);
            view.dispatchAttachedToWindow(info,
                    combineVisibility(visibility, view.getVisibility()));
        }
    }

    @Override
    void dispatchScreenStateChanged(int screenState) {
        super.dispatchScreenStateChanged(screenState);

        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            children[i].dispatchScreenStateChanged(screenState);
        }
    }

    @Override
    void dispatchMovedToDisplay(Display display, Configuration config) {
        super.dispatchMovedToDisplay(display, config);

        final int count = mChildrenCount;
```

```
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchMovedToDisplay(display, config);
    }
}

/** @hide */
@Override
public boolean dispatchPopulateAccessibilityEventInternal(AccessibilityEvent event) {
    boolean handled = false;
    if (includeForAccessibility()) {
        handled = super.dispatchPopulateAccessibilityEventInternal(event);
        if (handled) {
            return handled;
        }
    }
    // Let our children have a shot in populating the event.
    ChildListForAccessibility children = ChildListForAccessibility.obtain(this, true);
    try {
        final int childCount = children.getChildCount();
        for (int i = 0; i < childCount; i++) {
            View child = children.getChildAt(i);
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
                handled = child.dispatchPopulateAccessibilityEvent(event);
                if (handled) {
                    return handled;
                }
            }
        }
    } finally {
        children.recycle();
    }
    return false;
}

/**
 * Dispatch creation of {@link ViewStructure} down the hierarchy.  This implementation
 * adds in all child views of the view group, in addition to calling the default View
 * implementation.
 */
@Override
public void dispatchProvideStructure(ViewStructure structure) {
    super.dispatchProvideStructure(structure);
    if (isAssistBlocked() || structure.getChildCount() != 0) {
        return;
    }
    final int childrenCount = mChildrenCount;
    if (childrenCount <= 0) {
        return;
    }

    if (!isLaidOut()) {
        Log.v(VIEW_LOG_TAG, "dispatchProvideStructure(): not laid out, ignoring "
                + childrenCount + " children of " + getAccessibilityViewId());
        return;
    }

    structure.setChildCount(childrenCount);
    ArrayList<View> preorderedList = buildOrderedChildList();
    boolean customOrder = preorderedList == null
            && isChildrenDrawingOrderEnabled();
    for (int i = 0; i < childrenCount; i++) {
        int childIndex;
        try {
            childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
        } catch (IndexOutOfBoundsException e) {
            childIndex = i;
            if (mContext.getApplicationInfo().targetSdkVersion < Build.VERSION_CODES.M) {
                Log.w(TAG, "Bad getChildDrawingOrder while collecting assist @ "
                        + i + " of " + childrenCount, e);
                // At least one app is failing when we call getChildDrawingOrder
                // at this point, so deal semi-gracefully with it by falling back
                // on the basic order.
                customOrder = false;
                if (i > 0) {
                    // If we failed at the first index, there really isn't
                    // anything to do -- we will just proceed with the simple
                    // sequence order.
                    // Otherwise, we failed in the middle, so need to come up
                    // with an order for the remaining indices and use that.
                    // Failed at the first one, easy peasy.
                    int[] permutation = new int[childrenCount];
```

```
                    SparseBooleanArray usedIndices = new SparseBooleanArray();
                    // Go back and collected the indices we have done so far.
                    for (int j = 0; j < i; j++) {
                        permutation[j] = getChildDrawingOrder(childrenCount, j);
                        usedIndices.put(permutation[j], true);
                    }
                    // Fill in the remaining indices with indices that have not
                    // yet been used.
                    int nextIndex = 0;
                    for (int j = i; j < childrenCount; j++) {
                        while (usedIndices.get(nextIndex, false)) {
                            nextIndex++;
                        }
                        permutation[j] = nextIndex;
                        nextIndex++;
                    }
                    // Build the final view list.
                    preorderedList = new ArrayList<>(childrenCount);
                    for (int j = 0; j < childrenCount; j++) {
                        final int index = permutation[j];
                        final View child = mChildren[index];
                        preorderedList.add(child);
                    }
                }
            } else {
                throw e;
            }
        }
        final View child = getAndVerifyPreorderedView(preorderedList, mChildren,
                childIndex);
        final ViewStructure cstructure = structure.newChild(i);
        child.dispatchProvideStructure(cstructure);
    }
    if (preorderedList != null) {
        preorderedList.clear();
    }
}

/**
 * {@inheritDoc}
 *
 * <p>This implementation adds in all child views of the view group, in addition to calling the
 * default {@link View} implementation.
 */
@Override
public void dispatchProvideAutofillStructure(ViewStructure structure,
        @AutofillFlags int flags) {
    super.dispatchProvideAutofillStructure(structure, flags);
    if (structure.getChildCount() != 0) {
        return;
    }

    if (!isLaidOut()) {
        Log.v(VIEW_LOG_TAG, "dispatchProvideAutofillStructure(): not laid out, ignoring "
                + mChildrenCount + " children of " + getAutofillId());
        return;
    }

    final ChildListForAutoFill children = getChildrenForAutofill(flags);
    final int childrenCount = children.size();
    structure.setChildCount(childrenCount);
    for (int i = 0; i < childrenCount; i++) {
        final View child = children.get(i);
        final ViewStructure cstructure = structure.newChild(i);
        child.dispatchProvideAutofillStructure(cstructure, flags);
    }
    children.recycle();
}

/**
 * Gets the children for autofill. Children for autofill are the first
 * level descendants that are important for autofill. The returned
 * child list object is pooled and the caller must recycle it once done.
 * @hide */
private @NonNull ChildListForAutoFill getChildrenForAutofill(@AutofillFlags int flags) {
    final ChildListForAutoFill children = ChildListForAutoFill.obtain();
    populateChildrenForAutofill(children, flags);
    return children;
}

/** @hide */
private void populateChildrenForAutofill(ArrayList<View> list, @AutofillFlags int flags) {
```

```
    final int childrenCount = mChildrenCount;
    if (childrenCount <= 0) {
        return;
    }
    final ArrayList<View> preorderedList = buildOrderedChildList();
    final boolean customOrder = preorderedList == null
            && isChildrenDrawingOrderEnabled();
    for (int i = 0; i < childrenCount; i++) {
        final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
        final View child = (preorderedList == null)
                ? mChildren[childIndex] : preorderedList.get(childIndex);
        if ((flags & AUTOFILL_FLAG_INCLUDE_NOT_IMPORTANT_VIEWS) != 0
                || child.isImportantForAutofill()) {
            list.add(child);
        } else if (child instanceof ViewGroup) {
            ((ViewGroup) child).populateChildrenForAutofill(list, flags);
        }
    }
}

private static View getAndVerifyPreorderedView(ArrayList<View> preorderedList, View[] children,
        int childIndex) {
    final View child;
    if (preorderedList != null) {
        child = preorderedList.get(childIndex);
        if (child == null) {
            throw new RuntimeException("Invalid preorderedList contained null child at index "
                    + childIndex);
        }
    } else {
        child = children[childIndex];
    }
    return child;
}

/** @hide */
@Override
public void onInitializeAccessibilityNodeInfoInternal(AccessibilityNodeInfo info) {
    super.onInitializeAccessibilityNodeInfoInternal(info);
    if (getAccessibilityNodeProvider() != null) {
        return;
    }
    if (mAttachInfo != null) {
        final ArrayList<View> childrenForAccessibility = mAttachInfo.mTempArrayList;
        childrenForAccessibility.clear();
        addChildrenForAccessibility(childrenForAccessibility);
        final int childrenForAccessibilityCount = childrenForAccessibility.size();
        for (int i = 0; i < childrenForAccessibilityCount; i++) {
            final View child = childrenForAccessibility.get(i);
            info.addChildUnchecked(child);
        }
        childrenForAccessibility.clear();
    }
}

@Override
public CharSequence getAccessibilityClassName() {
    return ViewGroup.class.getName();
}

@Override
public void notifySubtreeAccessibilityStateChanged(View child, View source, int changeType) {
    // If this is a live region, we should send a subtree change event
    // from this view. Otherwise, we can let it propagate up.
    if (getAccessibilityLiveRegion() != ACCESSIBILITY_LIVE_REGION_NONE) {
        notifyViewAccessibilityStateChangedIfNeeded(
                AccessibilityEvent.CONTENT_CHANGE_TYPE_SUBTREE);
    } else if (mParent != null) {
        try {
            mParent.notifySubtreeAccessibilityStateChanged(this, source, changeType);
        } catch (AbstractMethodError e) {
            Log.e(VIEW_LOG_TAG, mParent.getClass().getSimpleName() +
                    " does not fully implement ViewParent", e);
        }
    }
}

/** @hide */
@Override
public void notifySubtreeAccessibilityStateChangedIfNeeded() {
    if (!AccessibilityManager.getInstance(mContext).isEnabled() || mAttachInfo == null) {
        return;
```

```
    }
    // If something important for a11y is happening in this subtree, make sure it's dispatched
    // from a view that is important for a11y so it doesn't get lost.
    if ((getImportantForAccessibility() != IMPORTANT_FOR_ACCESSIBILITY_NO_HIDE_DESCENDANTS)
            && !isImportantForAccessibility() && (getChildCount() > 0)) {
        ViewParent a11yParent = getParentForAccessibility();
        if (a11yParent instanceof View) {
            ((View) a11yParent).notifySubtreeAccessibilityStateChangedIfNeeded();
            return;
        }
    }
    super.notifySubtreeAccessibilityStateChangedIfNeeded();
}

@Override
void resetSubtreeAccessibilityStateChanged() {
    super.resetSubtreeAccessibilityStateChanged();
    View[] children = mChildren;
    final int childCount = mChildrenCount;
    for (int i = 0; i < childCount; i++) {
        children[i].resetSubtreeAccessibilityStateChanged();
    }
}

/**
 * Counts the number of children of this View that will be sent to an accessibility service.
 *
 * @return The number of children an {@code AccessibilityNodeInfo} rooted at this View
 * would have.
 */
int getNumChildrenForAccessibility() {
    int numChildrenForAccessibility = 0;
    for (int i = 0; i < getChildCount(); i++) {
        View child = getChildAt(i);
        if (child.includeForAccessibility()) {
            numChildrenForAccessibility++;
        } else if (child instanceof ViewGroup) {
            numChildrenForAccessibility += ((ViewGroup) child)
                    .getNumChildrenForAccessibility();
        }
    }
    return numChildrenForAccessibility;
}

/**
 * {@inheritDoc}
 *
 * <p>Subclasses should always call <code>super.onNestedPrePerformAccessibilityAction</code></p>
 *
 * @param target The target view dispatching this action
 * @param action Action being performed; see
 *               {@link android.view.accessibility.AccessibilityNodeInfo}
 * @param args Optional action arguments
 * @return false by default. Subclasses should return true if they handle the event.
 */
@Override
public boolean onNestedPrePerformAccessibilityAction(View target, int action, Bundle args) {
    return false;
}

@Override
void dispatchDetachedFromWindow() {
    // If we still have a touch target, we are still in the process of
    // dispatching motion events to a child; we need to get rid of that
    // child to avoid dispatching events to it after the window is torn
    // down. To make sure we keep the child in a consistent state, we
    // first send it an ACTION_CANCEL motion event.
    cancelAndClearTouchTargets(null);

    // Similarly, set ACTION_EXIT to all hover targets and clear them.
    exitHoverTargets();
    exitTooltipHoverTargets();

    // In case view is detached while transition is running
    mLayoutCalledWhileSuppressed = false;

    // Tear down our drag tracking
    mChildrenInterestedInDrag = null;
    mIsInterestedInDrag = false;
    if (mCurrentDragStartEvent != null) {
        mCurrentDragStartEvent.recycle();
        mCurrentDragStartEvent = null;
```

```
    }

    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        children[i].dispatchDetachedFromWindow();
    }
    clearDisappearingChildren();
    final int transientCount = mTransientViews == null ? 0 : mTransientIndices.size();
    for (int i = 0; i < transientCount; ++i) {
        View view = mTransientViews.get(i);
        view.dispatchDetachedFromWindow();
    }
    super.dispatchDetachedFromWindow();
}

/**
 * @hide
 */
@Override
protected void internalSetPadding(int left, int top, int right, int bottom) {
    super.internalSetPadding(left, top, right, bottom);

    if ((mPaddingLeft | mPaddingTop | mPaddingRight | mPaddingBottom) != 0) {
        mGroupFlags |= FLAG_PADDING_NOT_NULL;
    } else {
        mGroupFlags &= ~FLAG_PADDING_NOT_NULL;
    }
}

@Override
protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
    super.dispatchSaveInstanceState(container);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        View c = children[i];
        if ((c.mViewFlags & PARENT_SAVE_DISABLED_MASK) != PARENT_SAVE_DISABLED) {
            c.dispatchSaveInstanceState(container);
        }
    }
}

/**
 * Perform dispatching of a {@link #saveHierarchyState(android.util.SparseArray)}  freeze()}
 * to only this view, not to its children.  For use when overriding
 * {@link #dispatchSaveInstanceState(android.util.SparseArray)}  dispatchFreeze()} to allow
 * subclasses to freeze their own state but not the state of their children.
 *
 * @param container the container
 */
protected void dispatchFreezeSelfOnly(SparseArray<Parcelable> container) {
    super.dispatchSaveInstanceState(container);
}

@Override
protected void dispatchRestoreInstanceState(SparseArray<Parcelable> container) {
    super.dispatchRestoreInstanceState(container);
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        View c = children[i];
        if ((c.mViewFlags & PARENT_SAVE_DISABLED_MASK) != PARENT_SAVE_DISABLED) {
            c.dispatchRestoreInstanceState(container);
        }
    }
}

/**
 * Perform dispatching of a {@link #restoreHierarchyState(android.util.SparseArray)}
 * to only this view, not to its children.  For use when overriding
 * {@link #dispatchRestoreInstanceState(android.util.SparseArray)} to allow
 * subclasses to thaw their own state but not the state of their children.
 *
 * @param container the container
 */
protected void dispatchThawSelfOnly(SparseArray<Parcelable> container) {
    super.dispatchRestoreInstanceState(container);
}

/**
 * Enables or disables the drawing cache for each child of this view group.
```

```
     *
     * @param enabled true to enable the cache, false to dispose of it
     */
    protected void setChildrenDrawingCacheEnabled(boolean enabled) {
        if (enabled || (mPersistentDrawingCache & PERSISTENT_ALL_CACHES) != PERSISTENT_ALL_CACHES) {
            final View[] children = mChildren;
            final int count = mChildrenCount;
            for (int i = 0; i < count; i++) {
                children[i].setDrawingCacheEnabled(enabled);
            }
        }
    }

    /**
     * @hide
     */
    @Override
    public Bitmap createSnapshot(Bitmap.Config quality, int backgroundColor, boolean skipChildren) {
        int count = mChildrenCount;
        int[] visibilities = null;

        if (skipChildren) {
            visibilities = new int[count];
            for (int i = 0; i < count; i++) {
                View child = getChildAt(i);
                visibilities[i] = child.getVisibility();
                if (visibilities[i] == View.VISIBLE) {
                    child.mViewFlags = (child.mViewFlags & ~View.VISIBILITY_MASK)
                            | (View.INVISIBLE & View.VISIBILITY_MASK);
                }
            }
        }

        Bitmap b = super.createSnapshot(quality, backgroundColor, skipChildren);

        if (skipChildren) {
            for (int i = 0; i < count; i++) {
                View child = getChildAt(i);
                child.mViewFlags = (child.mViewFlags & ~View.VISIBILITY_MASK)
                        | (visibilities[i] & View.VISIBILITY_MASK);
            }
        }

        return b;
    }

    /** Return true if this ViewGroup is laying out using optical bounds. */
    boolean isLayoutModeOptical() {
        return mLayoutMode == LAYOUT_MODE_OPTICAL_BOUNDS;
    }

    @Override
    Insets computeOpticalInsets() {
        if (isLayoutModeOptical()) {
            int left = 0;
            int top = 0;
            int right = 0;
            int bottom = 0;
            for (int i = 0; i < mChildrenCount; i++) {
                View child = getChildAt(i);
                if (child.getVisibility() == VISIBLE) {
                    Insets insets = child.getOpticalInsets();
                    left =   Math.max(left,   insets.left);
                    top =    Math.max(top,    insets.top);
                    right =  Math.max(right,  insets.right);
                    bottom = Math.max(bottom, insets.bottom);
                }
            }
            return Insets.of(left, top, right, bottom);
        } else {
            return Insets.NONE;
        }
    }

    private static void fillRect(Canvas canvas, Paint paint, int x1, int y1, int x2, int y2) {
        if (x1 != x2 && y1 != y2) {
            if (x1 > x2) {
                int tmp = x1; x1 = x2; x2 = tmp;
            }
            if (y1 > y2) {
                int tmp = y1; y1 = y2; y2 = tmp;
            }
```

```
            canvas.drawRect(x1, y1, x2, y2, paint);
        }
    }

    private static int sign(int x) {
        return (x >= 0) ? 1 : -1;
    }

    private static void drawCorner(Canvas c, Paint paint, int x1, int y1, int dx, int dy, int lw) {
        fillRect(c, paint, x1, y1, x1 + dx, y1 + lw * sign(dy));
        fillRect(c, paint, x1, y1, x1 + lw * sign(dx), y1 + dy);
    }

    private static void drawRectCorners(Canvas canvas, int x1, int y1, int x2, int y2, Paint paint,
            int lineLength, int lineWidth) {
        drawCorner(canvas, paint, x1, y1, lineLength, lineLength, lineWidth);
        drawCorner(canvas, paint, x1, y2, lineLength, -lineLength, lineWidth);
        drawCorner(canvas, paint, x2, y1, -lineLength, lineLength, lineWidth);
        drawCorner(canvas, paint, x2, y2, -lineLength, -lineLength, lineWidth);
    }

    private static void fillDifference(Canvas canvas,
            int x2, int y2, int x3, int y3,
            int dx1, int dy1, int dx2, int dy2, Paint paint) {
        int x1 = x2 - dx1;
        int y1 = y2 - dy1;

        int x4 = x3 + dx2;
        int y4 = y3 + dy2;

        fillRect(canvas, paint, x1, y1, x4, y2);
        fillRect(canvas, paint, x1, y2, x2, y3);
        fillRect(canvas, paint, x3, y2, x4, y3);
        fillRect(canvas, paint, x1, y3, x4, y4);
    }

    /**
     * @hide
     */
    protected void onDebugDrawMargins(Canvas canvas, Paint paint) {
        for (int i = 0; i < getChildCount(); i++) {
            View c = getChildAt(i);
            c.getLayoutParams().onDebugDraw(c, canvas, paint);
        }
    }

    /**
     * @hide
     */
    protected void onDebugDraw(Canvas canvas) {
        Paint paint = getDebugPaint();

        // Draw optical bounds
        {
            paint.setColor(Color.RED);
            paint.setStyle(Paint.Style.STROKE);

            for (int i = 0; i < getChildCount(); i++) {
                View c = getChildAt(i);
                if (c.getVisibility() != View.GONE) {
                    Insets insets = c.getOpticalInsets();

                    drawRect(canvas, paint,
                            c.getLeft() + insets.left,
                            c.getTop() + insets.top,
                            c.getRight() - insets.right - 1,
                            c.getBottom() - insets.bottom - 1);
                }
            }
        }

        // Draw margins
        {
            paint.setColor(Color.argb(63, 255, 0, 255));
            paint.setStyle(Paint.Style.FILL);

            onDebugDrawMargins(canvas, paint);
        }

        // Draw clip bounds
        {
            paint.setColor(DEBUG_CORNERS_COLOR);
```

```
        paint.setStyle(Paint.Style.FILL);

        int lineLength = dipsToPixels(DEBUG_CORNERS_SIZE_DIP);
        int lineWidth = dipsToPixels(1);
        for (int i = 0; i < getChildCount(); i++) {
            View c = getChildAt(i);
            if (c.getVisibility() != View.GONE) {
                drawRectCorners(canvas, c.getLeft(), c.getTop(), c.getRight(), c.getBottom(),
                        paint, lineLength, lineWidth);
            }
        }
    }
}

@Override
protected void dispatchDraw(Canvas canvas) {
    boolean usingRenderNodeProperties = canvas.isRecordingFor(mRenderNode);
    final int childrenCount = mChildrenCount;
    final View[] children = mChildren;
    int flags = mGroupFlags;

    if ((flags & FLAG_RUN_ANIMATION) != 0 && canAnimate()) {
        final boolean buildCache = !isHardwareAccelerated();
        for (int i = 0; i < childrenCount; i++) {
            final View child = children[i];
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE) {
                final LayoutParams params = child.getLayoutParams();
                attachLayoutAnimationParameters(child, params, i, childrenCount);
                bindLayoutAnimation(child);
            }
        }

        final LayoutAnimationController controller = mLayoutAnimationController;
        if (controller.willOverlap()) {
            mGroupFlags |= FLAG_OPTIMIZE_INVALIDATE;
        }

        controller.start();

        mGroupFlags &= ~FLAG_RUN_ANIMATION;
        mGroupFlags &= ~FLAG_ANIMATION_DONE;

        if (mAnimationListener != null) {
            mAnimationListener.onAnimationStart(controller.getAnimation());
        }
    }

    int clipSaveCount = 0;
    final boolean clipToPadding = (flags & CLIP_TO_PADDING_MASK) == CLIP_TO_PADDING_MASK;
    if (clipToPadding) {
        clipSaveCount = canvas.save(Canvas.CLIP_SAVE_FLAG);
        canvas.clipRect(mScrollX + mPaddingLeft, mScrollY + mPaddingTop,
                mScrollX + mRight - mLeft - mPaddingRight,
                mScrollY + mBottom - mTop - mPaddingBottom);
    }

    // We will draw our child's animation, let's reset the flag
    mPrivateFlags &= ~PFLAG_DRAW_ANIMATION;
    mGroupFlags &= ~FLAG_INVALIDATE_REQUIRED;

    boolean more = false;
    final long drawingTime = getDrawingTime();

    if (usingRenderNodeProperties) canvas.insertReorderBarrier();
    final int transientCount = mTransientIndices == null ? 0 : mTransientIndices.size();
    int transientIndex = transientCount != 0 ? 0 : -1;
    // Only use the preordered list if not HW accelerated, since the HW pipeline will do the
    // draw reordering internally
    final ArrayList<View> preorderedList = usingRenderNodeProperties
            ? null : buildOrderedChildList();
    final boolean customOrder = preorderedList == null
            && isChildrenDrawingOrderEnabled();
    for (int i = 0; i < childrenCount; i++) {
        while (transientIndex >= 0 && mTransientIndices.get(transientIndex) == i) {
            final View transientChild = mTransientViews.get(transientIndex);
            if ((transientChild.mViewFlags & VISIBILITY_MASK) == VISIBLE ||
                    transientChild.getAnimation() != null) {
                more |= drawChild(canvas, transientChild, drawingTime);
            }
            transientIndex++;
            if (transientIndex >= transientCount) {
                transientIndex = -1;
```

```
            }
        }

        final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
        final View child = getAndVerifyPreorderedView(preorderedList, children, childIndex);
        if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null) {
            more |= drawChild(canvas, child, drawingTime);
        }
    }
    while (transientIndex >= 0) {
        // there may be additional transient views after the normal views
        final View transientChild = mTransientViews.get(transientIndex);
        if ((transientChild.mViewFlags & VISIBILITY_MASK) == VISIBLE ||
                transientChild.getAnimation() != null) {
            more |= drawChild(canvas, transientChild, drawingTime);
        }
        transientIndex++;
        if (transientIndex >= transientCount) {
            break;
        }
    }
    if (preorderedList != null) preorderedList.clear();

    // Draw any disappearing views that have animations
    if (mDisappearingChildren != null) {
        final ArrayList<View> disappearingChildren = mDisappearingChildren;
        final int disappearingCount = disappearingChildren.size() - 1;
        // Go backwards -- we may delete as animations finish
        for (int i = disappearingCount; i >= 0; i--) {
            final View child = disappearingChildren.get(i);
            more |= drawChild(canvas, child, drawingTime);
        }
    }
    if (usingRenderNodeProperties) canvas.insertInorderBarrier();

    if (debugDraw()) {
        onDebugDraw(canvas);
    }

    if (clipToPadding) {
        canvas.restoreToCount(clipSaveCount);
    }

    // mGroupFlags might have been updated by drawChild()
    flags = mGroupFlags;

    if ((flags & FLAG_INVALIDATE_REQUIRED) == FLAG_INVALIDATE_REQUIRED) {
        invalidate(true);
    }

    if ((flags & FLAG_ANIMATION_DONE) == 0 && (flags & FLAG_NOTIFY_ANIMATION_LISTENER) == 0 &&
            mLayoutAnimationController.isDone() && !more) {
        // We want to erase the drawing cache and notify the listener after the
        // next frame is drawn because one extra invalidate() is caused by
        // drawChild() after the animation is over
        mGroupFlags |= FLAG_NOTIFY_ANIMATION_LISTENER;
        final Runnable end = new Runnable() {
           @Override
           public void run() {
               notifyAnimationListener();
           }
        };
        post(end);
    }
}

/**
 * Returns the ViewGroupOverlay for this view group, creating it if it does
 * not yet exist. In addition to {@link ViewOverlay}'s support for drawables,
 * {@link ViewGroupOverlay} allows views to be added to the overlay. These
 * views, like overlay drawables, are visual-only; they do not receive input
 * events and should not be used as anything other than a temporary
 * representation of a view in a parent container, such as might be used
 * by an animation effect.
 *
 * <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 ViewGroupOverlay object for this view.
 * @see ViewGroupOverlay
 */
```

```java
@Override
public ViewGroupOverlay getOverlay() {
    if (mOverlay == null) {
        mOverlay = new ViewGroupOverlay(mContext, this);
    }
    return (ViewGroupOverlay) mOverlay;
}

/**
 * Returns the index of the child to draw for this iteration. Override this
 * if you want to change the drawing order of children. By default, it
 * returns i.
 * <p>
 * NOTE: In order for this method to be called, you must enable child ordering
 * first by calling {@link #setChildrenDrawingOrderEnabled(boolean)}.
 *
 * @param i The current iteration.
 * @return The index of the child to draw this iteration.
 *
 * @see #setChildrenDrawingOrderEnabled(boolean)
 * @see #isChildrenDrawingOrderEnabled()
 */
protected int getChildDrawingOrder(int childCount, int i) {
    return i;
}

private boolean hasChildWithZ() {
    for (int i = 0; i < mChildrenCount; i++) {
        if (mChildren[i].getZ() != 0) return true;
    }
    return false;
}

/**
 * Populates (and returns) mPreSortedChildren with a pre-ordered list of the View's children,
 * sorted first by Z, then by child drawing order (if applicable). This list must be cleared
 * after use to avoid leaking child Views.
 *
 * Uses a stable, insertion sort which is commonly O(n) for ViewGroups with very few elevated
 * children.
 */
ArrayList<View> buildOrderedChildList() {
    final int childrenCount = mChildrenCount;
    if (childrenCount <= 1 || !hasChildWithZ()) return null;

    if (mPreSortedChildren == null) {
        mPreSortedChildren = new ArrayList<>(childrenCount);
    } else {
        // callers should clear, so clear shouldn't be necessary, but for safety...
        mPreSortedChildren.clear();
        mPreSortedChildren.ensureCapacity(childrenCount);
    }

    final boolean customOrder = isChildrenDrawingOrderEnabled();
    for (int i = 0; i < childrenCount; i++) {
        // add next child (in child order) to end of list
        final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
        final View nextChild = mChildren[childIndex];
        final float currentZ = nextChild.getZ();

        // insert ahead of any Views with greater Z
        int insertIndex = i;
        while (insertIndex > 0 && mPreSortedChildren.get(insertIndex - 1).getZ() > currentZ) {
            insertIndex--;
        }
        mPreSortedChildren.add(insertIndex, nextChild);
    }
    return mPreSortedChildren;
}

private void notifyAnimationListener() {
    mGroupFlags &= ~FLAG_NOTIFY_ANIMATION_LISTENER;
    mGroupFlags |= FLAG_ANIMATION_DONE;

    if (mAnimationListener != null) {
       final Runnable end = new Runnable() {
           @Override
           public void run() {
               mAnimationListener.onAnimationEnd(mLayoutAnimationController.getAnimation());
           }
       };
       post(end);
```

```
    }

    invalidate(true);
}

/**
 * This method is used to cause children of this ViewGroup 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
 */
@Override
protected void dispatchGetDisplayList() {
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        final View child = children[i];
        if (((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null)) {
            recreateChildDisplayList(child);
        }
    }
    if (mOverlay != null) {
        View overlayView = mOverlay.getOverlayView();
        recreateChildDisplayList(overlayView);
    }
    if (mDisappearingChildren != null) {
        final ArrayList<View> disappearingChildren = mDisappearingChildren;
        final int disappearingCount = disappearingChildren.size();
        for (int i = 0; i < disappearingCount; ++i) {
            final View child = disappearingChildren.get(i);
            recreateChildDisplayList(child);
        }
    }
}

private void recreateChildDisplayList(View child) {
    child.mRecreateDisplayList = (child.mPrivateFlags & PFLAG_INVALIDATED) != 0;
    child.mPrivateFlags &= ~PFLAG_INVALIDATED;
    child.updateDisplayListIfDirty();
    child.mRecreateDisplayList = false;
}

/**
 * Draw one child of this View Group. This method is responsible for getting
 * the canvas in the right state. This includes clipping, translating so
 * that the child's scrolled origin is at 0, 0, and applying any animation
 * transformations.
 *
 * @param canvas The canvas on which to draw the child
 * @param child Who to draw
 * @param drawingTime The time at which draw is occurring
 * @return True if an invalidate() was issued
 */
protected boolean drawChild(Canvas canvas, View child, long drawingTime) {
    return child.draw(canvas, this, drawingTime);
}

@Override
void getScrollIndicatorBounds(@NonNull Rect out) {
    super.getScrollIndicatorBounds(out);

    // If we have padding and we're supposed to clip children to that
    // padding, offset the scroll indicators to match our clip bounds.
    final boolean clipToPadding = (mGroupFlags & CLIP_TO_PADDING_MASK) == CLIP_TO_PADDING_MASK;
    if (clipToPadding) {
        out.left += mPaddingLeft;
        out.right -= mPaddingRight;
        out.top += mPaddingTop;
        out.bottom -= mPaddingBottom;
    }
}

/**
 * Returns whether this group's children are clipped to their bounds before drawing.
 * The default value is true.
 * @see #setClipChildren(boolean)
 *
 * @return True if the group's children will be clipped to their bounds,
 * false otherwise.
 */
```

```java
@ViewDebug.ExportedProperty(category = "drawing")
public boolean getClipChildren() {
    return ((mGroupFlags & FLAG_CLIP_CHILDREN) != 0);
}

/**
 * By default, children are clipped to their bounds before drawing. This
 * allows view groups to override this behavior for animations, etc.
 *
 * @param clipChildren true to clip children to their bounds,
 *        false otherwise
 * @attr ref android.R.styleable#ViewGroup_clipChildren
 */
public void setClipChildren(boolean clipChildren) {
    boolean previousValue = (mGroupFlags & FLAG_CLIP_CHILDREN) == FLAG_CLIP_CHILDREN;
    if (clipChildren != previousValue) {
        setBooleanFlag(FLAG_CLIP_CHILDREN, clipChildren);
        for (int i = 0; i < mChildrenCount; ++i) {
            View child = getChildAt(i);
            if (child.mRenderNode != null) {
                child.mRenderNode.setClipToBounds(clipChildren);
            }
        }
        invalidate(true);
    }
}

/**
 * Sets whether this ViewGroup will clip its children to its padding and resize (but not
 * clip) any EdgeEffect to the padded region, if padding is present.
 * <p>
 * By default, children are clipped to the padding of their parent
 * ViewGroup. This clipping behavior is only enabled if padding is non-zero.
 *
 * @param clipToPadding true to clip children to the padding of the group, and resize (but
 *        not clip) any EdgeEffect to the padded region. False otherwise.
 * @attr ref android.R.styleable#ViewGroup_clipToPadding
 */
public void setClipToPadding(boolean clipToPadding) {
    if (hasBooleanFlag(FLAG_CLIP_TO_PADDING) != clipToPadding) {
        setBooleanFlag(FLAG_CLIP_TO_PADDING, clipToPadding);
        invalidate(true);
    }
}

/**
 * Returns whether this ViewGroup will clip its children to its padding, and resize (but
 * not clip) any EdgeEffect to the padded region, if padding is present.
 * <p>
 * By default, children are clipped to the padding of their parent
 * Viewgroup. This clipping behavior is only enabled if padding is non-zero.
 *
 * @return true if this ViewGroup clips children to its padding and resizes (but doesn't
 *         clip) any EdgeEffect to the padded region, false otherwise.
 *
 * @attr ref android.R.styleable#ViewGroup_clipToPadding
 */
@ViewDebug.ExportedProperty(category = "drawing")
public boolean getClipToPadding() {
    return hasBooleanFlag(FLAG_CLIP_TO_PADDING);
}

@Override
public void dispatchSetSelected(boolean selected) {
    final View[] children = mChildren;
    final int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        children[i].setSelected(selected);
    }
}

@Override
public void dispatchSetActivated(boolean activated) {
    final View[] children = mChildren;
    final int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        children[i].setActivated(activated);
    }
}

@Override
protected void dispatchSetPressed(boolean pressed) {
```

```
    final View[] children = mChildren;
    final int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        final View child = children[i];
        // Children that are clickable on their own should not
        // show a pressed state when their parent view does.
        // Clearing a pressed state always propagates.
        if (!pressed || (!child.isClickable() && !child.isLongClickable())) {
            child.setPressed(pressed);
        }
    }
}

/**
 * Dispatches drawable hotspot changes to child views that meet at least
 * one of the following criteria:
 * <ul>
 *     <li>Returns {@code false} from both {@link View#isClickable()} and
 *     {@link View#isLongClickable()}</li>
 *     <li>Requests duplication of parent state via
 *     {@link View#setDuplicateParentStateEnabled(boolean)}</li>
 * </ul>
 *
 * @param x hotspot x coordinate
 * @param y hotspot y coordinate
 * @see #drawableHotspotChanged(float, float)
 */
@Override
public void dispatchDrawableHotspotChanged(float x, float y) {
    final int count = mChildrenCount;
    if (count == 0) {
        return;
    }

    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        final View child = children[i];
        // Children that are clickable on their own should not
        // receive hotspots when their parent view does.
        final boolean nonActionable = !child.isClickable() && !child.isLongClickable();
        final boolean duplicatesState = (child.mViewFlags & DUPLICATE_PARENT_STATE) != 0;
        if (nonActionable || duplicatesState) {
            final float[] point = getTempPoint();
            point[0] = x;
            point[1] = y;
            transformPointToViewLocal(point, child);
            child.drawableHotspotChanged(point[0], point[1]);
        }
    }
}

@Override
void dispatchCancelPendingInputEvents() {
    super.dispatchCancelPendingInputEvents();

    final View[] children = mChildren;
    final int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        children[i].dispatchCancelPendingInputEvents();
    }
}

/**
 * When this property is set to true, this ViewGroup supports static transformations on
 * children; this causes
 * {@link #getChildStaticTransformation(View, android.view.animation.Transformation)} to be
 * invoked when a child is drawn.
 *
 * Any subclass overriding
 * {@link #getChildStaticTransformation(View, android.view.animation.Transformation)} should
 * set this property to true.
 *
 * @param enabled True to enable static transformations on children, false otherwise.
 *
 * @see #getChildStaticTransformation(View, android.view.animation.Transformation)
 */
protected void setStaticTransformationsEnabled(boolean enabled) {
    setBooleanFlag(FLAG_SUPPORT_STATIC_TRANSFORMATIONS, enabled);
}

/**
 * Sets  <code>t</code> to be the static transformation of the child, if set, returning a
```

```
     * boolean to indicate whether a static transform was set. The default implementation
     * simply returns <code>false</code>; subclasses may override this method for different
     * behavior. {@link #setStaticTransformationsEnabled(boolean)} must be set to true
     * for this method to be called.
     *
     * @param child The child view whose static transform is being requested
     * @param t The Transformation which will hold the result
     * @return true if the transformation was set, false otherwise
     * @see #setStaticTransformationsEnabled(boolean)
     */
    protected boolean getChildStaticTransformation(View child, Transformation t) {
        return false;
    }

    Transformation getChildTransformation() {
        if (mChildTransformation == null) {
            mChildTransformation = new Transformation();
        }
        return mChildTransformation;
    }

    /**
     * {@hide}
     */
    @Override
    protected <T extends View> T findViewTraversal(@IdRes int id) {
        if (id == mID) {
            return (T) this;
        }

        final View[] where = mChildren;
        final int len = mChildrenCount;

        for (int i = 0; i < len; i++) {
            View v = where[i];

            if ((v.mPrivateFlags & PFLAG_IS_ROOT_NAMESPACE) == 0) {
                v = v.findViewById(id);

                if (v != null) {
                    return (T) v;
                }
            }
        }

        return null;
    }

    /**
     * {@hide}
     */
    @Override
    protected <T extends View> T findViewWithTagTraversal(Object tag) {
        if (tag != null && tag.equals(mTag)) {
            return (T) this;
        }

        final View[] where = mChildren;
        final int len = mChildrenCount;

        for (int i = 0; i < len; i++) {
            View v = where[i];

            if ((v.mPrivateFlags & PFLAG_IS_ROOT_NAMESPACE) == 0) {
                v = v.findViewWithTag(tag);

                if (v != null) {
                    return (T) v;
                }
            }
        }

        return null;
    }

    /**
     * {@hide}
     */
    @Override
    protected <T extends View> T findViewByPredicateTraversal(Predicate<View> predicate,
            View childToSkip) {
        if (predicate.test(this)) {
```

```
            return (T) this;
        }

        final View[] where = mChildren;
        final int len = mChildrenCount;

        for (int i = 0; i < len; i++) {
            View v = where[i];

            if (v != childToSkip && (v.mPrivateFlags & PFLAG_IS_ROOT_NAMESPACE) == 0) {
                v = v.findViewByPredicate(predicate);

                if (v != null) {
                    return (T) v;
                }
            }
        }

        return null;
    }

    /**
     * This method adds a view to this container at the specified index purely for the
     * purposes of allowing that view to draw even though it is not a normal child of
     * the container. That is, the view does not participate in layout, focus, accessibility,
     * input, or other normal view operations; it is purely an item to be drawn during the normal
     * rendering operation of this container. The index that it is added at is the order
     * in which it will be drawn, with respect to the other views in the container.
     * For example, a transient view added at index 0 will be drawn before all other views
     * in the container because it will be drawn first (including before any real view
     * at index 0). There can be more than one transient view at any particular index;
     * these views will be drawn in the order in which they were added to the list of
     * transient views. The index of transient views can also be greater than the number
     * of normal views in the container; that just means that they will be drawn after all
     * other views are drawn.
     *
     * <p>Note that since transient views do not participate in layout, they must be sized
     * manually or, more typically, they should just use the size that they had before they
     * were removed from their container.</p>
     *
     * <p>Transient views are useful for handling animations of views that have been removed
     * from the container, but which should be animated out after the removal. Adding these
     * views as transient views allows them to participate in drawing without side-effecting
     * the layout of the container.</p>
     *
     * <p>Transient views must always be explicitly {@link #removeTransientView(View) removed}
     * from the container when they are no longer needed. For example, a transient view
     * which is added in order to fade it out in its old location should be removed
     * once the animation is complete.</p>
     *
     * @param view The view to be added
     * @param index The index at which this view should be drawn, must be >= 0.
     * This value is relative to the {@link #getChildAt(int) index} values in the normal
     * child list of this container, where any transient view at a particular index will
     * be drawn before any normal child at that same index.
     *
     * @hide
     */
    public void addTransientView(View view, int index) {
        if (index < 0) {
            return;
        }
        if (mTransientIndices == null) {
            mTransientIndices = new ArrayList<Integer>();
            mTransientViews = new ArrayList<View>();
        }
        final int oldSize = mTransientIndices.size();
        if (oldSize > 0) {
            int insertionIndex;
            for (insertionIndex = 0; insertionIndex < oldSize; ++insertionIndex) {
                if (index < mTransientIndices.get(insertionIndex)) {
                    break;
                }
            }
            mTransientIndices.add(insertionIndex, index);
            mTransientViews.add(insertionIndex, view);
        } else {
            mTransientIndices.add(index);
            mTransientViews.add(view);
        }
        view.mParent = this;
        view.dispatchAttachedToWindow(mAttachInfo, (mViewFlags&VISIBILITY_MASK));
```

```
    invalidate(true);
}

/**
 * Removes a view from the list of transient views in this container. If there is no
 * such transient view, this method does nothing.
 *
 * @param view The transient view to be removed
 *
 * @hide
 */
public void removeTransientView(View view) {
    if (mTransientViews == null) {
        return;
    }
    final int size = mTransientViews.size();
    for (int i = 0; i < size; ++i) {
        if (view == mTransientViews.get(i)) {
            mTransientViews.remove(i);
            mTransientIndices.remove(i);
            view.mParent = null;
            view.dispatchDetachedFromWindow();
            invalidate(true);
            return;
        }
    }
}

/**
 * Returns the number of transient views in this container. Specific transient
 * views and the index at which they were added can be retrieved via
 * {@link #getTransientView(int)} and {@link #getTransientViewIndex(int)}.
 *
 * @see #addTransientView(View, int)
 * @return The number of transient views in this container
 *
 * @hide
 */
public int getTransientViewCount() {
    return mTransientIndices == null ? 0 : mTransientIndices.size();
}

/**
 * Given a valid position within the list of transient views, returns the index of
 * the transient view at that position.
 *
 * @param position The position of the index being queried. Must be at least 0
 * and less than the value returned by {@link #getTransientViewCount()}.
 * @return The index of the transient view stored in the given position if the
 * position is valid, otherwise -1
 *
 * @hide
 */
public int getTransientViewIndex(int position) {
    if (position < 0 || mTransientIndices == null || position >= mTransientIndices.size()) {
        return -1;
    }
    return mTransientIndices.get(position);
}

/**
 * Given a valid position within the list of transient views, returns the
 * transient view at that position.
 *
 * @param position The position of the view being queried. Must be at least 0
 * and less than the value returned by {@link #getTransientViewCount()}.
 * @return The transient view stored in the given position if the
 * position is valid, otherwise null
 *
 * @hide
 */
public View getTransientView(int position) {
    if (mTransientViews == null || position >= mTransientViews.size()) {
        return null;
    }
    return mTransientViews.get(position);
}

/**
 * <p>Adds a child view. If no layout parameters are already set on the child, the
 * default parameters for this ViewGroup are set on the child.</p>
 *
```

```
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     *
     * @param child the child view to add
     *
     * @see #generateDefaultLayoutParams()
     */
    public void addView(View child) {
        addView(child, -1);
    }

    /**
     * Adds a child view. If no layout parameters are already set on the child, the
     * default parameters for this ViewGroup are set on the child.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     *
     * @param child the child view to add
     * @param index the position at which to add the child
     *
     * @see #generateDefaultLayoutParams()
     */
    public void addView(View child, int index) {
        if (child == null) {
            throw new IllegalArgumentException("Cannot add a null child view to a ViewGroup");
        }
        LayoutParams params = child.getLayoutParams();
        if (params == null) {
            params = generateDefaultLayoutParams();
            if (params == null) {
                throw new IllegalArgumentException("generateDefaultLayoutParams() cannot return null");
            }
        }
        addView(child, index, params);
    }

    /**
     * Adds a child view with this ViewGroup's default layout parameters and the
     * specified width and height.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     *
     * @param child the child view to add
     */
    public void addView(View child, int width, int height) {
        final LayoutParams params = generateDefaultLayoutParams();
        params.width = width;
        params.height = height;
        addView(child, -1, params);
    }

    /**
     * Adds a child view with the specified layout parameters.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     *
     * @param child the child view to add
     * @param params the layout parameters to set on the child
     */
    @Override
    public void addView(View child, LayoutParams params) {
        addView(child, -1, params);
    }

    /**
     * Adds a child view with the specified layout parameters.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     *
     * @param child the child view to add
     * @param index the position at which to add the child or -1 to add last
     * @param params the layout parameters to set on the child
     */
```

```
public void addView(View child, int index, LayoutParams params) {
    if (DBG) {
        System.out.println(this + " addView");
    }

    if (child == null) {
        throw new IllegalArgumentException("Cannot add a null child view to a ViewGroup");
    }

    // addViewInner() will call child.requestLayout() when setting the new LayoutParams
    // therefore, we call requestLayout() on ourselves before, so that the child's request
    // will be blocked at our level
    requestLayout();
    invalidate(true);
    addViewInner(child, index, params, false);
}

@Override
public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
    if (!checkLayoutParams(params)) {
        throw new IllegalArgumentException("Invalid LayoutParams supplied to " + this);
    }
    if (view.mParent != this) {
        throw new IllegalArgumentException("Given view not a child of " + this);
    }
    view.setLayoutParams(params);
}

protected boolean checkLayoutParams(ViewGroup.LayoutParams p) {
    return  p != null;
}

/**
 * Interface definition for a callback to be invoked when the hierarchy
 * within this view changed. The hierarchy changes whenever a child is added
 * to or removed from this view.
 */
public interface OnHierarchyChangeListener {
    /**
     * Called when a new child is added to a parent view.
     *
     * @param parent the view in which a child was added
     * @param child the new child view added in the hierarchy
     */
    void onChildViewAdded(View parent, View child);

    /**
     * Called when a child is removed from a parent view.
     *
     * @param parent the view from which the child was removed
     * @param child the child removed from the hierarchy
     */
    void onChildViewRemoved(View parent, View child);
}

/**
 * Register a callback to be invoked when a child is added to or removed
 * from this view.
 *
 * @param listener the callback to invoke on hierarchy change
 */
public void setOnHierarchyChangeListener(OnHierarchyChangeListener listener) {
    mOnHierarchyChangeListener = listener;
}

void dispatchViewAdded(View child) {
    onViewAdded(child);
    if (mOnHierarchyChangeListener != null) {
        mOnHierarchyChangeListener.onChildViewAdded(this, child);
    }
}

/**
 * Called when a new child is added to this ViewGroup. Overrides should always
 * call super.onViewAdded.
 *
 * @param child the added child view
 */
public void onViewAdded(View child) {
}

void dispatchViewRemoved(View child) {
```

```
    onViewRemoved(child);
    if (mOnHierarchyChangeListener != null) {
        mOnHierarchyChangeListener.onChildViewRemoved(this, child);
    }
}

/**
 * Called when a child view is removed from this ViewGroup. Overrides should always
 * call super.onViewRemoved.
 *
 * @param child the removed child view
 */
public void onViewRemoved(View child) {
}

private void clearCachedLayoutMode() {
    if (!hasBooleanFlag(FLAG_LAYOUT_MODE_WAS_EXPLICITLY_SET)) {
       mLayoutMode = LAYOUT_MODE_UNDEFINED;
    }
}

@Override
protected void onAttachedToWindow() {
    super.onAttachedToWindow();
    clearCachedLayoutMode();
}

@Override
protected void onDetachedFromWindow() {
    super.onDetachedFromWindow();
    clearCachedLayoutMode();
}

/** @hide */
@Override
protected void destroyHardwareResources() {
    super.destroyHardwareResources();
    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        getChildAt(i).destroyHardwareResources();
    }
}

/**
 * Adds a view during layout. This is useful if in your onLayout() method,
 * you need to add more views (as does the list view for example).
 *
 * If index is negative, it means put it at the end of the list.
 *
 * @param child the view to add to the group
 * @param index the index at which the child must be added or -1 to add last
 * @param params the layout parameters to associate with the child
 * @return true if the child was added, false otherwise
 */
protected boolean addViewInLayout(View child, int index, LayoutParams params) {
    return addViewInLayout(child, index, params, false);
}

/**
 * Adds a view during layout. This is useful if in your onLayout() method,
 * you need to add more views (as does the list view for example).
 *
 * If index is negative, it means put it at the end of the list.
 *
 * @param child the view to add to the group
 * @param index the index at which the child must be added or -1 to add last
 * @param params the layout parameters to associate with the child
 * @param preventRequestLayout if true, calling this method will not trigger a
 *        layout request on child
 * @return true if the child was added, false otherwise
 */
protected boolean addViewInLayout(View child, int index, LayoutParams params,
        boolean preventRequestLayout) {
    if (child == null) {
        throw new IllegalArgumentException("Cannot add a null child view to a ViewGroup");
    }
    child.mParent = null;
    addViewInner(child, index, params, preventRequestLayout);
    child.mPrivateFlags = (child.mPrivateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;
    return true;
}
```

```
/**
 * Prevents the specified child to be laid out during the next layout pass.
 *
 * @param child the child on which to perform the cleanup
 */
protected void cleanupLayoutState(View child) {
    child.mPrivateFlags &= ~View.PFLAG_FORCE_LAYOUT;
}

private void addViewInner(View child, int index, LayoutParams params,
        boolean preventRequestLayout) {

    if (mTransition != null) {
        // Don't prevent other add transitions from completing, but cancel remove
        // transitions to let them complete the process before we add to the container
        mTransition.cancel(LayoutTransition.DISAPPEARING);
    }

    if (child.getParent() != null) {
        throw new IllegalStateException("The specified child already has a parent. " +
                "You must call removeView() on the child's parent first.");
    }

    if (mTransition != null) {
        mTransition.addChild(this, child);
    }

    if (!checkLayoutParams(params)) {
        params = generateLayoutParams(params);
    }

    if (preventRequestLayout) {
        child.mLayoutParams = params;
    } else {
        child.setLayoutParams(params);
    }

    if (index < 0) {
        index = mChildrenCount;
    }

    addInArray(child, index);

    // tell our children
    if (preventRequestLayout) {
        child.assignParent(this);
    } else {
        child.mParent = this;
    }

    final boolean childHasFocus = child.hasFocus();
    if (childHasFocus) {
        requestChildFocus(child, child.findFocus());
    }

    AttachInfo ai = mAttachInfo;
    if (ai != null && (mGroupFlags & FLAG_PREVENT_DISPATCH_ATTACHED_TO_WINDOW) == 0) {
        boolean lastKeepOn = ai.mKeepScreenOn;
        ai.mKeepScreenOn = false;
        child.dispatchAttachedToWindow(mAttachInfo, (mViewFlags&VISIBILITY_MASK));
        if (ai.mKeepScreenOn) {
            needGlobalAttributesUpdate(true);
        }
        ai.mKeepScreenOn = lastKeepOn;
    }

    if (child.isLayoutDirectionInherited()) {
        child.resetRtlProperties();
    }

    dispatchViewAdded(child);

    if ((child.mViewFlags & DUPLICATE_PARENT_STATE) == DUPLICATE_PARENT_STATE) {
        mGroupFlags |= FLAG_NOTIFY_CHILDREN_ON_DRAWABLE_STATE_CHANGE;
    }

    if (child.hasTransientState()) {
        childHasTransientStateChanged(child, true);
    }

    if (child.getVisibility() != View.GONE) {
        notifySubtreeAccessibilityStateChangedIfNeeded();
```

```
    }

    if (mTransientIndices != null) {
        final int transientCount = mTransientIndices.size();
        for (int i = 0; i < transientCount; ++i) {
            final int oldIndex = mTransientIndices.get(i);
            if (index <= oldIndex) {
                mTransientIndices.set(i, oldIndex + 1);
            }
        }
    }

    if (mCurrentDragStartEvent != null && child.getVisibility() == VISIBLE) {
        notifyChildOfDragStart(child);
    }

    if (child.hasDefaultFocus()) {
        // When adding a child that contains default focus, either during inflation or while
        // manually assembling the hierarchy, update the ancestor default-focus chain.
        setDefaultFocus(child);
    }
}

private void addInArray(View child, int index) {
    View[] children = mChildren;
    final int count = mChildrenCount;
    final int size = children.length;
    if (index == count) {
        if (size == count) {
            mChildren = new View[size + ARRAY_CAPACITY_INCREMENT];
            System.arraycopy(children, 0, mChildren, 0, size);
            children = mChildren;
        }
        children[mChildrenCount++] = child;
    } else if (index < count) {
        if (size == count) {
            mChildren = new View[size + ARRAY_CAPACITY_INCREMENT];
            System.arraycopy(children, 0, mChildren, 0, index);
            System.arraycopy(children, index, mChildren, index + 1, count - index);
            children = mChildren;
        } else {
            System.arraycopy(children, index, children, index + 1, count - index);
        }
        children[index] = child;
        mChildrenCount++;
        if (mLastTouchDownIndex >= index) {
            mLastTouchDownIndex++;
        }
    } else {
        throw new IndexOutOfBoundsException("index=" + index + " count=" + count);
    }
}

// This method also sets the child's mParent to null
private void removeFromArray(int index) {
    final View[] children = mChildren;
    if (!(mTransitioningViews != null && mTransitioningViews.contains(children[index]))) {
        children[index].mParent = null;
    }
    final int count = mChildrenCount;
    if (index == count - 1) {
        children[--mChildrenCount] = null;
    } else if (index >= 0 && index < count) {
        System.arraycopy(children, index + 1, children, index, count - index - 1);
        children[--mChildrenCount] = null;
    } else {
        throw new IndexOutOfBoundsException();
    }
    if (mLastTouchDownIndex == index) {
        mLastTouchDownTime = 0;
        mLastTouchDownIndex = -1;
    } else if (mLastTouchDownIndex > index) {
        mLastTouchDownIndex--;
    }
}

// This method also sets the children's mParent to null
private void removeFromArray(int start, int count) {
    final View[] children = mChildren;
    final int childrenCount = mChildrenCount;

    start = Math.max(0, start);
```

```
    final int end = Math.min(childrenCount, start + count);

    if (start == end) {
        return;
    }

    if (end == childrenCount) {
        for (int i = start; i < end; i++) {
            children[i].mParent = null;
            children[i] = null;
        }
    } else {
        for (int i = start; i < end; i++) {
            children[i].mParent = null;
        }

        // Since we're looping above, we might as well do the copy, but is arraycopy()
        // faster than the extra 2 bounds checks we would do in the loop?
        System.arraycopy(children, end, children, start, childrenCount - end);

        for (int i = childrenCount - (end - start); i < childrenCount; i++) {
            children[i] = null;
        }
    }

    mChildrenCount -= (end - start);
}

private void bindLayoutAnimation(View child) {
    Animation a = mLayoutAnimationController.getAnimationForView(child);
    child.setAnimation(a);
}

/**
 * Subclasses should override this method to set layout animation
 * parameters on the supplied child.
 *
 * @param child the child to associate with animation parameters
 * @param params the child's layout parameters which hold the animation
 *        parameters
 * @param index the index of the child in the view group
 * @param count the number of children in the view group
 */
protected void attachLayoutAnimationParameters(View child,
        LayoutParams params, int index, int count) {
    LayoutAnimationController.AnimationParameters animationParams =
                params.layoutAnimationParameters;
    if (animationParams == null) {
        animationParams = new LayoutAnimationController.AnimationParameters();
        params.layoutAnimationParameters = animationParams;
    }

    animationParams.count = count;
    animationParams.index = index;
}

/**
 * {@inheritDoc}
 *
 * <p><strong>Note:</strong> do not invoke this method from
 * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
 * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
 */
@Override
public void removeView(View view) {
    if (removeViewInternal(view)) {
        requestLayout();
        invalidate(true);
    }
}

/**
 * Removes a view during layout. This is useful if in your onLayout() method,
 * you need to remove more views.
 *
 * <p><strong>Note:</strong> do not invoke this method from
 * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
 * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
 *
 * @param view the view to remove from the group
 */
public void removeViewInLayout(View view) {
```

```
    removeViewInternal(view);
}

/**
 * Removes a range of views during layout. This is useful if in your onLayout() method,
 * you need to remove more views.
 *
 * <p><strong>Note:</strong> do not invoke this method from
 * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
 * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
 *
 * @param start the index of the first view to remove from the group
 * @param count the number of views to remove from the group
 */
public void removeViewsInLayout(int start, int count) {
    removeViewsInternal(start, count);
}

/**
 * Removes the view at the specified position in the group.
 *
 * <p><strong>Note:</strong> do not invoke this method from
 * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
 * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
 *
 * @param index the position in the group of the view to remove
 */
public void removeViewAt(int index) {
    removeViewInternal(index, getChildAt(index));
    requestLayout();
    invalidate(true);
}

/**
 * Removes the specified range of views from the group.
 *
 * <p><strong>Note:</strong> do not invoke this method from
 * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
 * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
 *
 * @param start the first position in the group of the range of views to remove
 * @param count the number of views to remove
 */
public void removeViews(int start, int count) {
    removeViewsInternal(start, count);
    requestLayout();
    invalidate(true);
}

private boolean removeViewInternal(View view) {
    final int index = indexOfChild(view);
    if (index >= 0) {
        removeViewInternal(index, view);
        return true;
    }
    return false;
}

private void removeViewInternal(int index, View view) {
    if (mTransition != null) {
        mTransition.removeChild(this, view);
    }

    boolean clearChildFocus = false;
    if (view == mFocused) {
        view.unFocus(null);
        clearChildFocus = true;
    }
    if (view == mFocusedInCluster) {
        clearFocusedInCluster(view);
    }

    view.clearAccessibilityFocus();

    cancelTouchTarget(view);
    cancelHoverTarget(view);

    if (view.getAnimation() != null ||
            (mTransitioningViews != null && mTransitioningViews.contains(view))) {
        addDisappearingView(view);
    } else if (view.mAttachInfo != null) {
       view.dispatchDetachedFromWindow();
```

```
    }

    if (view.hasTransientState()) {
        childHasTransientStateChanged(view, false);
    }

    needGlobalAttributesUpdate(false);

    removeFromArray(index);

    if (view == mDefaultFocus) {
        clearDefaultFocus(view);
    }
    if (clearChildFocus) {
        clearChildFocus(view);
        if (!rootViewRequestFocus()) {
            notifyGlobalFocusCleared(this);
        }
    }

    dispatchViewRemoved(view);

    if (view.getVisibility() != View.GONE) {
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }

    int transientCount = mTransientIndices == null ? 0 : mTransientIndices.size();
    for (int i = 0; i < transientCount; ++i) {
        final int oldIndex = mTransientIndices.get(i);
        if (index < oldIndex) {
            mTransientIndices.set(i, oldIndex - 1);
        }
    }

    if (mCurrentDragStartEvent != null) {
        mChildrenInterestedInDrag.remove(view);
    }
}

/**
 * Sets the LayoutTransition object for this ViewGroup. If the LayoutTransition object is
 * not null, changes in layout which occur because of children being added to or removed from
 * the ViewGroup will be animated according to the animations defined in that LayoutTransition
 * object. By default, the transition object is null (so layout changes are not animated).
 *
 * <p>Replacing a non-null transition will cause that previous transition to be
 * canceled, if it is currently running, to restore this container to
 * its correct post-transition state.</p>
 *
 * @param transition The LayoutTransition object that will animated changes in layout. A value
 * of <code>null</code> means no transition will run on layout changes.
 * @attr ref android.R.styleable#ViewGroup_animateLayoutChanges
 */
public void setLayoutTransition(LayoutTransition transition) {
    if (mTransition != null) {
        LayoutTransition previousTransition = mTransition;
        previousTransition.cancel();
        previousTransition.removeTransitionListener(mLayoutTransitionListener);
    }
    mTransition = transition;
    if (mTransition != null) {
        mTransition.addTransitionListener(mLayoutTransitionListener);
    }
}

/**
 * Gets the LayoutTransition object for this ViewGroup. If the LayoutTransition object is
 * not null, changes in layout which occur because of children being added to or removed from
 * the ViewGroup will be animated according to the animations defined in that LayoutTransition
 * object. By default, the transition object is null (so layout changes are not animated).
 *
 * @return LayoutTranstion The LayoutTransition object that will animated changes in layout.
 * A value of <code>null</code> means no transition will run on layout changes.
 */
public LayoutTransition getLayoutTransition() {
    return mTransition;
}

private void removeViewsInternal(int start, int count) {
    final int end = start + count;

    if (start < 0 || count < 0 || end > mChildrenCount) {
```

```
            throw new IndexOutOfBoundsException();
        }

        final View focused = mFocused;
        final boolean detach = mAttachInfo != null;
        boolean clearChildFocus = false;
        View clearDefaultFocus = null;

        final View[] children = mChildren;

        for (int i = start; i < end; i++) {
            final View view = children[i];

            if (mTransition != null) {
                mTransition.removeChild(this, view);
            }

            if (view == focused) {
                view.unFocus(null);
                clearChildFocus = true;
            }
            if (view == mDefaultFocus) {
                clearDefaultFocus = view;
            }
            if (view == mFocusedInCluster) {
                clearFocusedInCluster(view);
            }

            view.clearAccessibilityFocus();

            cancelTouchTarget(view);
            cancelHoverTarget(view);

            if (view.getAnimation() != null ||
                (mTransitioningViews != null && mTransitioningViews.contains(view))) {
                addDisappearingView(view);
            } else if (detach) {
               view.dispatchDetachedFromWindow();
            }

            if (view.hasTransientState()) {
                childHasTransientStateChanged(view, false);
            }

            needGlobalAttributesUpdate(false);

            dispatchViewRemoved(view);
        }

        removeFromArray(start, count);

        if (clearDefaultFocus != null) {
            clearDefaultFocus(clearDefaultFocus);
        }
        if (clearChildFocus) {
            clearChildFocus(focused);
            if (!rootViewRequestFocus()) {
                notifyGlobalFocusCleared(focused);
            }
        }
    }

    /**
     * Call this method to remove all child views from the
     * ViewGroup.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     */
    public void removeAllViews() {
        removeAllViewsInLayout();
        requestLayout();
        invalidate(true);
    }

    /**
     * Called by a ViewGroup subclass to remove child views from itself,
     * when it must first know its size on screen before it can calculate how many
     * child views it will render. An example is a Gallery or a ListView, which
     * may "have" 50 children, but actually only render the number of children
     * that can currently fit inside the object on screen. Do not call
```

```
     * this method unless you are extending ViewGroup and understand the
     * view measuring and layout pipeline.
     *
     * <p><strong>Note:</strong> do not invoke this method from
     * {@link #draw(android.graphics.Canvas)}, {@link #onDraw(android.graphics.Canvas)},
     * {@link #dispatchDraw(android.graphics.Canvas)} or any related method.</p>
     */
    public void removeAllViewsInLayout() {
        final int count = mChildrenCount;
        if (count <= 0) {
            return;
        }

        final View[] children = mChildren;
        mChildrenCount = 0;

        final View focused = mFocused;
        final boolean detach = mAttachInfo != null;
        boolean clearChildFocus = false;

        needGlobalAttributesUpdate(false);

        for (int i = count - 1; i >= 0; i--) {
            final View view = children[i];

            if (mTransition != null) {
                mTransition.removeChild(this, view);
            }

            if (view == focused) {
                view.unFocus(null);
                clearChildFocus = true;
            }

            view.clearAccessibilityFocus();

            cancelTouchTarget(view);
            cancelHoverTarget(view);

            if (view.getAnimation() != null ||
                    (mTransitioningViews != null && mTransitioningViews.contains(view))) {
                addDisappearingView(view);
            } else if (detach) {
               view.dispatchDetachedFromWindow();
            }

            if (view.hasTransientState()) {
                childHasTransientStateChanged(view, false);
            }

            dispatchViewRemoved(view);

            view.mParent = null;
            children[i] = null;
        }

        if (mDefaultFocus != null) {
            clearDefaultFocus(mDefaultFocus);
        }
        if (mFocusedInCluster != null) {
            clearFocusedInCluster(mFocusedInCluster);
        }
        if (clearChildFocus) {
            clearChildFocus(focused);
            if (!rootViewRequestFocus()) {
                notifyGlobalFocusCleared(focused);
            }
        }
    }

    /**
     * Finishes the removal of a detached view. This method will dispatch the detached from
     * window event and notify the hierarchy change listener.
     * <p>
     * This method is intended to be lightweight and makes no assumptions about whether the
     * parent or child should be redrawn. Proper use of this method will include also making
     * any appropriate {@link #requestLayout()} or {@link #invalidate()} calls.
     * For example, callers can {@link #post(Runnable) post} a {@link Runnable}
     * which performs a {@link #requestLayout()} on the next frame, after all detach/remove
     * calls are finished, causing layout to be run prior to redrawing the view hierarchy.
     *
     * @param child the child to be definitely removed from the view hierarchy
```

```
     * @param animate if true and the view has an animation, the view is placed in the
     *                disappearing views list, otherwise, it is detached from the window
     *
     * @see #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)
     * @see #detachAllViewsFromParent()
     * @see #detachViewFromParent(View)
     * @see #detachViewFromParent(int)
     */
    protected void removeDetachedView(View child, boolean animate) {
        if (mTransition != null) {
            mTransition.removeChild(this, child);
        }

        if (child == mFocused) {
            child.clearFocus();
        }
        if (child == mDefaultFocus) {
            clearDefaultFocus(child);
        }
        if (child == mFocusedInCluster) {
            clearFocusedInCluster(child);
        }

        child.clearAccessibilityFocus();

        cancelTouchTarget(child);
        cancelHoverTarget(child);

        if ((animate && child.getAnimation() != null) ||
                (mTransitioningViews != null && mTransitioningViews.contains(child))) {
            addDisappearingView(child);
        } else if (child.mAttachInfo != null) {
            child.dispatchDetachedFromWindow();
        }

        if (child.hasTransientState()) {
            childHasTransientStateChanged(child, false);
        }

        dispatchViewRemoved(child);
    }

    /**
     * Attaches a view to this view group. Attaching a view assigns this group as the parent,
     * sets the layout parameters and puts the view in the list of children so that
     * it can be retrieved by calling {@link #getChildAt(int)}.
     * <p>
     * This method is intended to be lightweight and makes no assumptions about whether the
     * parent or child should be redrawn. Proper use of this method will include also making
     * any appropriate {@link #requestLayout()} or {@link #invalidate()} calls.
     * For example, callers can {@link #post(Runnable) post} a {@link Runnable}
     * which performs a {@link #requestLayout()} on the next frame, after all detach/attach
     * calls are finished, causing layout to be run prior to redrawing the view hierarchy.
     * <p>
     * This method should be called only for views which were detached from their parent.
     *
     * @param child the child to attach
     * @param index the index at which the child should be attached
     * @param params the layout parameters of the child
     *
     * @see #removeDetachedView(View, boolean)
     * @see #detachAllViewsFromParent()
     * @see #detachViewFromParent(View)
     * @see #detachViewFromParent(int)
     */
    protected void attachViewToParent(View child, int index, LayoutParams params) {
        child.mLayoutParams = params;

        if (index < 0) {
            index = mChildrenCount;
        }

        addInArray(child, index);

        child.mParent = this;
        child.mPrivateFlags = (child.mPrivateFlags & ~PFLAG_DIRTY_MASK
                        & ~PFLAG_DRAWING_CACHE_VALID)
                | PFLAG_DRAWN | PFLAG_INVALIDATED;
        this.mPrivateFlags |= PFLAG_INVALIDATED;

        if (child.hasFocus()) {
            requestChildFocus(child, child.findFocus());
```

```
    }
    dispatchVisibilityAggregated(isAttachedToWindow() && getWindowVisibility() == VISIBLE
            && isShown());
}

/**
 * Detaches a view from its parent. Detaching a view should be followed
 * either by a call to
 * {@link #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)}
 * or a call to {@link #removeDetachedView(View, boolean)}. Detachment should only be
 * temporary; reattachment or removal should happen within the same drawing cycle as
 * detachment. When a view is detached, its parent is null and cannot be retrieved by a
 * call to {@link #getChildAt(int)}.
 *
 * @param child the child to detach
 *
 * @see #detachViewFromParent(int)
 * @see #detachViewsFromParent(int, int)
 * @see #detachAllViewsFromParent()
 * @see #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)
 * @see #removeDetachedView(View, boolean)
 */
protected void detachViewFromParent(View child) {
    removeFromArray(indexOfChild(child));
}

/**
 * Detaches a view from its parent. Detaching a view should be followed
 * either by a call to
 * {@link #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)}
 * or a call to {@link #removeDetachedView(View, boolean)}. Detachment should only be
 * temporary; reattachment or removal should happen within the same drawing cycle as
 * detachment. When a view is detached, its parent is null and cannot be retrieved by a
 * call to {@link #getChildAt(int)}.
 *
 * @param index the index of the child to detach
 *
 * @see #detachViewFromParent(View)
 * @see #detachAllViewsFromParent()
 * @see #detachViewsFromParent(int, int)
 * @see #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)
 * @see #removeDetachedView(View, boolean)
 */
protected void detachViewFromParent(int index) {
    removeFromArray(index);
}

/**
 * Detaches a range of views from their parents. Detaching a view should be followed
 * either by a call to
 * {@link #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)}
 * or a call to {@link #removeDetachedView(View, boolean)}. Detachment should only be
 * temporary; reattachment or removal should happen within the same drawing cycle as
 * detachment. When a view is detached, its parent is null and cannot be retrieved by a
 * call to {@link #getChildAt(int)}.
 *
 * @param start the first index of the childrend range to detach
 * @param count the number of children to detach
 *
 * @see #detachViewFromParent(View)
 * @see #detachViewFromParent(int)
 * @see #detachAllViewsFromParent()
 * @see #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)
 * @see #removeDetachedView(View, boolean)
 */
protected void detachViewsFromParent(int start, int count) {
    removeFromArray(start, count);
}

/**
 * Detaches all views from the parent. Detaching a view should be followed
 * either by a call to
 * {@link #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)}
 * or a call to {@link #removeDetachedView(View, boolean)}. Detachment should only be
 * temporary; reattachment or removal should happen within the same drawing cycle as
 * detachment. When a view is detached, its parent is null and cannot be retrieved by a
 * call to {@link #getChildAt(int)}.
 *
 * @see #detachViewFromParent(View)
 * @see #detachViewFromParent(int)
 * @see #detachViewsFromParent(int, int)
 * @see #attachViewToParent(View, int, android.view.ViewGroup.LayoutParams)
```

```
     * @see #removeDetachedView(View, boolean)
     */
    protected void detachAllViewsFromParent() {
        final int count = mChildrenCount;
        if (count <= 0) {
            return;
        }

        final View[] children = mChildren;
        mChildrenCount = 0;

        for (int i = count - 1; i >= 0; i--) {
            children[i].mParent = null;
            children[i] = null;
        }
    }

    @Override
    @CallSuper
    public void onDescendantInvalidated(@NonNull View child, @NonNull View target) {
        /*
         * HW-only, Rect-ignoring damage codepath
         *
         * We don't deal with rectangles here, since RenderThread native code computes damage for
         * everything drawn by HWUI (and SW layer / drawing cache doesn't keep track of damage area)
         */

        // if set, combine the animation flag into the parent
        mPrivateFlags |= (target.mPrivateFlags & PFLAG_DRAW_ANIMATION);

        if ((target.mPrivateFlags & ~PFLAG_DIRTY_MASK) != 0) {
            // We lazily use PFLAG_DIRTY, since computing opaque isn't worth the potential
            // optimization in provides in a DisplayList world.
            mPrivateFlags = (mPrivateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DIRTY;

            // simplified invalidateChildInParent behavior: clear cache validity to be safe...
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        }

        // ... and mark inval if in software layer that needs to repaint (hw handled in native)
        if (mLayerType == LAYER_TYPE_SOFTWARE) {
            // Layered parents should be invalidated. Escalate to a full invalidate (and note that
            // we do this after consuming any relevant flags from the originating descendant)
            mPrivateFlags |= PFLAG_INVALIDATED | PFLAG_DIRTY;
            target = this;
        }

        if (mParent != null) {
            mParent.onDescendantInvalidated(this, target);
        }
    }


    /**
     * Don't call or override this method. It is used for the implementation of
     * the view hierarchy.
     *
     * @deprecated Use {@link #onDescendantInvalidated(View, View)} instead to observe updates to
     * draw state in descendants.
     */
    @Deprecated
    @Override
    public final void invalidateChild(View child, final Rect dirty) {
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null && attachInfo.mHardwareAccelerated) {
            // HW accelerated fast path
            onDescendantInvalidated(child, child);
            return;
        }

        ViewParent parent = this;
        if (attachInfo != null) {
            // If the child is drawing an animation, we want to copy this flag onto
            // ourselves and the parent to make sure the invalidate request goes
            // through
            final boolean drawAnimation = (child.mPrivateFlags & PFLAG_DRAW_ANIMATION) != 0;

            // Check whether the child that requests the invalidate is fully opaque
            // Views being animated or transformed are not considered opaque because we may
            // be invalidating their old position and need the parent to paint behind them.
            Matrix childMatrix = child.getMatrix();
            final boolean isOpaque = child.isOpaque() && !drawAnimation &&
```

```
                child.getAnimation() == null && childMatrix.isIdentity();
        // Mark the child as dirty, using the appropriate flag
        // Make sure we do not set both flags at the same time
        int opaqueFlag = isOpaque ? PFLAG_DIRTY_OPAQUE : PFLAG_DIRTY;

        if (child.mLayerType != LAYER_TYPE_NONE) {
            mPrivateFlags |= PFLAG_INVALIDATED;
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        }

        final int[] location = attachInfo.mInvalidateChildLocation;
        location[CHILD_LEFT_INDEX] = child.mLeft;
        location[CHILD_TOP_INDEX] = child.mTop;
        if (!childMatrix.isIdentity() ||
                (mGroupFlags & ViewGroup.FLAG_SUPPORT_STATIC_TRANSFORMATIONS) != 0) {
            RectF boundingRect = attachInfo.mTmpTransformRect;
            boundingRect.set(dirty);
            Matrix transformMatrix;
            if ((mGroupFlags & ViewGroup.FLAG_SUPPORT_STATIC_TRANSFORMATIONS) != 0) {
                Transformation t = attachInfo.mTmpTransformation;
                boolean transformed = getChildStaticTransformation(child, t);
                if (transformed) {
                    transformMatrix = attachInfo.mTmpMatrix;
                    transformMatrix.set(t.getMatrix());
                    if (!childMatrix.isIdentity()) {
                        transformMatrix.preConcat(childMatrix);
                    }
                } else {
                    transformMatrix = childMatrix;
                }
            } else {
                transformMatrix = childMatrix;
            }
            transformMatrix.mapRect(boundingRect);
            dirty.set((int) Math.floor(boundingRect.left),
                    (int) Math.floor(boundingRect.top),
                    (int) Math.ceil(boundingRect.right),
                    (int) Math.ceil(boundingRect.bottom));
        }

        do {
            View view = null;
            if (parent instanceof View) {
                view = (View) parent;
            }

            if (drawAnimation) {
                if (view != null) {
                    view.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
                } else if (parent instanceof ViewRootImpl) {
                    ((ViewRootImpl) parent).mIsAnimating = true;
                }
            }

            // If the parent is dirty opaque or not dirty, mark it dirty with the opaque
            // flag coming from the child that initiated the invalidate
            if (view != null) {
                if ((view.mViewFlags & FADING_EDGE_MASK) != 0 &&
                        view.getSolidColor() == 0) {
                    opaqueFlag = PFLAG_DIRTY;
                }
                if ((view.mPrivateFlags & PFLAG_DIRTY_MASK) != PFLAG_DIRTY) {
                    view.mPrivateFlags = (view.mPrivateFlags & ~PFLAG_DIRTY_MASK) | opaqueFlag;
                }
            }

            parent = parent.invalidateChildInParent(location, dirty);
            if (view != null) {
                // Account for transform on current parent
                Matrix m = view.getMatrix();
                if (!m.isIdentity()) {
                    RectF boundingRect = attachInfo.mTmpTransformRect;
                    boundingRect.set(dirty);
                    m.mapRect(boundingRect);
                    dirty.set((int) Math.floor(boundingRect.left),
                            (int) Math.floor(boundingRect.top),
                            (int) Math.ceil(boundingRect.right),
                            (int) Math.ceil(boundingRect.bottom));
                }
            }
        } while (parent != null);
    }
```

```
}

/**
 * Don't call or override this method. It is used for the implementation of
 * the view hierarchy.
 *
 * This implementation returns null if this ViewGroup does not have a parent,
 * if this ViewGroup is already fully invalidated or if the dirty rectangle
 * does not intersect with this ViewGroup's bounds.
 *
 * @deprecated Use {@link #onDescendantInvalidated(View, View)} instead to observe updates to
 * draw state in descendants.
 */
@Deprecated
@Override
public ViewParent invalidateChildInParent(final int[] location, final Rect dirty) {
    if ((mPrivateFlags & (PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID)) != 0) {
        // either DRAWN, or DRAWING_CACHE_VALID
        if ((mGroupFlags & (FLAG_OPTIMIZE_INVALIDATE | FLAG_ANIMATION_DONE))
                != FLAG_OPTIMIZE_INVALIDATE) {
            dirty.offset(location[CHILD_LEFT_INDEX] - mScrollX,
                    location[CHILD_TOP_INDEX] - mScrollY);
            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == 0) {
                dirty.union(0, 0, mRight - mLeft, mBottom - mTop);
            }

            final int left = mLeft;
            final int top = mTop;

            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == FLAG_CLIP_CHILDREN) {
                if (!dirty.intersect(0, 0, mRight - left, mBottom - top)) {
                    dirty.setEmpty();
                }
            }

            location[CHILD_LEFT_INDEX] = left;
            location[CHILD_TOP_INDEX] = top;
        } else {

            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == FLAG_CLIP_CHILDREN) {
                dirty.set(0, 0, mRight - mLeft, mBottom - mTop);
            } else {
                // in case the dirty rect extends outside the bounds of this container
                dirty.union(0, 0, mRight - mLeft, mBottom - mTop);
            }
            location[CHILD_LEFT_INDEX] = mLeft;
            location[CHILD_TOP_INDEX] = mTop;

            mPrivateFlags &= ~PFLAG_DRAWN;
        }
        mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        if (mLayerType != LAYER_TYPE_NONE) {
            mPrivateFlags |= PFLAG_INVALIDATED;
        }

        return mParent;
    }

    return null;
}

/**
 * Offset a rectangle that is in a descendant's coordinate
 * space into our coordinate space.
 * @param descendant A descendant of this view
 * @param rect A rectangle defined in descendant's coordinate space.
 */
public final void offsetDescendantRectToMyCoords(View descendant, Rect rect) {
    offsetRectBetweenParentAndChild(descendant, rect, true, false);
}

/**
 * Offset a rectangle that is in our coordinate space into an ancestor's
 * coordinate space.
 * @param descendant A descendant of this view
 * @param rect A rectangle defined in descendant's coordinate space.
 */
public final void offsetRectIntoDescendantCoords(View descendant, Rect rect) {
    offsetRectBetweenParentAndChild(descendant, rect, false, false);
}

/**
```

```
 * Helper method that offsets a rect either from parent to descendant or
 * descendant to parent.
 */
void offsetRectBetweenParentAndChild(View descendant, Rect rect,
        boolean offsetFromChildToParent, boolean clipToBounds) {

    // already in the same coord system :)
    if (descendant == this) {
        return;
    }

    ViewParent theParent = descendant.mParent;

    // search and offset up to the parent
    while ((theParent != null)
            && (theParent instanceof View)
            && (theParent != this)) {

        if (offsetFromChildToParent) {
            rect.offset(descendant.mLeft - descendant.mScrollX,
                    descendant.mTop - descendant.mScrollY);
            if (clipToBounds) {
                View p = (View) theParent;
                boolean intersected = rect.intersect(0, 0, p.mRight - p.mLeft,
                        p.mBottom - p.mTop);
                if (!intersected) {
                    rect.setEmpty();
                }
            }
        } else {
            if (clipToBounds) {
                View p = (View) theParent;
                boolean intersected = rect.intersect(0, 0, p.mRight - p.mLeft,
                        p.mBottom - p.mTop);
                if (!intersected) {
                    rect.setEmpty();
                }
            }
            rect.offset(descendant.mScrollX - descendant.mLeft,
                    descendant.mScrollY - descendant.mTop);
        }

        descendant = (View) theParent;
        theParent = descendant.mParent;
    }

    // now that we are up to this view, need to offset one more time
    // to get into our coordinate space
    if (theParent == this) {
        if (offsetFromChildToParent) {
            rect.offset(descendant.mLeft - descendant.mScrollX,
                    descendant.mTop - descendant.mScrollY);
        } else {
            rect.offset(descendant.mScrollX - descendant.mLeft,
                    descendant.mScrollY - descendant.mTop);
        }
    } else {
        throw new IllegalArgumentException("parameter must be a descendant of this view");
    }
}

/**
 * Offset the vertical location of all children of this view by the specified number of pixels.
 *
 * @param offset the number of pixels to offset
 *
 * @hide
 */
public void offsetChildrenTopAndBottom(int offset) {
    final int count = mChildrenCount;
    final View[] children = mChildren;
    boolean invalidate = false;

    for (int i = 0; i < count; i++) {
        final View v = children[i];
        v.mTop += offset;
        v.mBottom += offset;
        if (v.mRenderNode != null) {
            invalidate = true;
            v.mRenderNode.offsetTopAndBottom(offset);
        }
    }
```

```
    if (invalidate) {
        invalidateViewProperty(false, false);
    }
    notifySubtreeAccessibilityStateChangedIfNeeded();
}

@Override
public boolean getChildVisibleRect(View child, Rect r, android.graphics.Point offset) {
    return getChildVisibleRect(child, r, offset, false);
}

/**
 * @param forceParentCheck true to guarantee that this call will propagate to all ancestors,
 *      false otherwise
 *
 * @hide
 */
public boolean getChildVisibleRect(
        View child, Rect r, android.graphics.Point offset, boolean forceParentCheck) {
    // It doesn't make a whole lot of sense to call this on a view that isn't attached,
    // but for some simple tests it can be useful. If we don't have attach info this
    // will allocate memory.
    final RectF rect = mAttachInfo != null ? mAttachInfo.mTmpTransformRect : new RectF();
    rect.set(r);

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

    final int dx = child.mLeft - mScrollX;
    final int dy = child.mTop - mScrollY;

    rect.offset(dx, dy);

    if (offset != null) {
        if (!child.hasIdentityMatrix()) {
            float[] position = mAttachInfo != null ? mAttachInfo.mTmpTransformLocation
                    : new float[2];
            position[0] = offset.x;
            position[1] = offset.y;
            child.getMatrix().mapPoints(position);
            offset.x = Math.round(position[0]);
            offset.y = Math.round(position[1]);
        }
        offset.x += dx;
        offset.y += dy;
    }

    final int width = mRight - mLeft;
    final int height = mBottom - mTop;

    boolean rectIsVisible = true;
    if (mParent == null ||
            (mParent instanceof ViewGroup && ((ViewGroup) mParent).getClipChildren())) {
        // Clip to bounds.
        rectIsVisible = rect.intersect(0, 0, width, height);
    }

    if ((forceParentCheck || rectIsVisible)
            && (mGroupFlags & CLIP_TO_PADDING_MASK) == CLIP_TO_PADDING_MASK) {
        // Clip to padding.
        rectIsVisible = rect.intersect(mPaddingLeft, mPaddingTop,
                width - mPaddingRight, height - mPaddingBottom);
    }

    if ((forceParentCheck || rectIsVisible) && mClipBounds != null) {
        // Clip to clipBounds.
        rectIsVisible = rect.intersect(mClipBounds.left, mClipBounds.top, mClipBounds.right,
                mClipBounds.bottom);
    }
    r.set((int) Math.floor(rect.left), (int) Math.floor(rect.top),
            (int) Math.ceil(rect.right), (int) Math.ceil(rect.bottom));

    if ((forceParentCheck || rectIsVisible) && mParent != null) {
        if (mParent instanceof ViewGroup) {
            rectIsVisible = ((ViewGroup) mParent)
                    .getChildVisibleRect(this, r, offset, forceParentCheck);
        } else {
            rectIsVisible = mParent.getChildVisibleRect(this, r, offset);
        }
    }
```

```
    return rectIsVisible;
}

@Override
public final void layout(int l, int t, int r, int b) {
    if (!mSuppressLayout && (mTransition == null || !mTransition.isChangingLayout())) {
        if (mTransition != null) {
            mTransition.layoutChange(this);
        }
        super.layout(l, t, r, b);
    } else {
        // record the fact that we noop'd it; request layout when transition finishes
        mLayoutCalledWhileSuppressed = true;
    }
}

@Override
protected abstract void onLayout(boolean changed,
        int l, int t, int r, int b);

/**
 * Indicates whether the view group has the ability to animate its children
 * after the first layout.
 *
 * @return true if the children can be animated, false otherwise
 */
protected boolean canAnimate() {
    return mLayoutAnimationController != null;
}

/**
 * Runs the layout animation. Calling this method triggers a relayout of
 * this view group.
 */
public void startLayoutAnimation() {
    if (mLayoutAnimationController != null) {
        mGroupFlags |= FLAG_RUN_ANIMATION;
        requestLayout();
    }
}

/**
 * Schedules the layout animation to be played after the next layout pass
 * of this view group. This can be used to restart the layout animation
 * when the content of the view group changes or when the activity is
 * paused and resumed.
 */
public void scheduleLayoutAnimation() {
    mGroupFlags |= FLAG_RUN_ANIMATION;
}

/**
 * Sets the layout animation controller used to animate the group's
 * children after the first layout.
 *
 * @param controller the animation controller
 */
public void setLayoutAnimation(LayoutAnimationController controller) {
    mLayoutAnimationController = controller;
    if (mLayoutAnimationController != null) {
        mGroupFlags |= FLAG_RUN_ANIMATION;
    }
}

/**
 * Returns the layout animation controller used to animate the group's
 * children.
 *
 * @return the current animation controller
 */
public LayoutAnimationController getLayoutAnimation() {
    return mLayoutAnimationController;
}

/**
 * Indicates whether the children's drawing cache is used during a layout
 * animation. By default, the drawing cache is enabled but this will prevent
 * nested layout animations from working. To nest animations, you must disable
 * the cache.
 *
 * @return true if the animation cache is enabled, false otherwise
 *
```

```
     * @see #setAnimationCacheEnabled(boolean)
     * @see View#setDrawingCacheEnabled(boolean)
     *
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Caching behavior of children may be controlled through {@link View#setLayerType(int, Paint)}.
     */
    @Deprecated
    public boolean isAnimationCacheEnabled() {
        return (mGroupFlags & FLAG_ANIMATION_CACHE) == FLAG_ANIMATION_CACHE;
    }

    /**
     * Enables or disables the children's drawing cache during a layout animation.
     * By default, the drawing cache is enabled but this will prevent nested
     * layout animations from working. To nest animations, you must disable the
     * cache.
     *
     * @param enabled true to enable the animation cache, false otherwise
     *
     * @see #isAnimationCacheEnabled()
     * @see View#setDrawingCacheEnabled(boolean)
     *
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Caching behavior of children may be controlled through {@link View#setLayerType(int, Paint)}.
     */
    @Deprecated
    public void setAnimationCacheEnabled(boolean enabled) {
        setBooleanFlag(FLAG_ANIMATION_CACHE, enabled);
    }

    /**
     * Indicates whether this ViewGroup will always try to draw its children using their
     * drawing cache. By default this property is enabled.
     *
     * @return true if the animation cache is enabled, false otherwise
     *
     * @see #setAlwaysDrawnWithCacheEnabled(boolean)
     * @see #setChildrenDrawnWithCacheEnabled(boolean)
     * @see View#setDrawingCacheEnabled(boolean)
     *
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Child views may no longer have their caching behavior disabled by parents.
     */
    @Deprecated
    public boolean isAlwaysDrawnWithCacheEnabled() {
        return (mGroupFlags & FLAG_ALWAYS_DRAWN_WITH_CACHE) == FLAG_ALWAYS_DRAWN_WITH_CACHE;
    }

    /**
     * Indicates whether this ViewGroup will always try to draw its children using their
     * drawing cache. This property can be set to true when the cache rendering is
     * slightly different from the children's normal rendering. Renderings can be different,
     * for instance, when the cache's quality is set to low.
     *
     * When this property is disabled, the ViewGroup will use the drawing cache of its
     * children only when asked to. It's usually the task of subclasses to tell ViewGroup
     * when to start using the drawing cache and when to stop using it.
     *
     * @param always true to always draw with the drawing cache, false otherwise
     *
     * @see #isAlwaysDrawnWithCacheEnabled()
     * @see #setChildrenDrawnWithCacheEnabled(boolean)
     * @see View#setDrawingCacheEnabled(boolean)
     * @see View#setDrawingCacheQuality(int)
     *
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Child views may no longer have their caching behavior disabled by parents.
     */
    @Deprecated
    public void setAlwaysDrawnWithCacheEnabled(boolean always) {
        setBooleanFlag(FLAG_ALWAYS_DRAWN_WITH_CACHE, always);
    }

    /**
     * Indicates whether the ViewGroup is currently drawing its children using
     * their drawing cache.
     *
     * @return true if children should be drawn with their cache, false otherwise
     *
     * @see #setAlwaysDrawnWithCacheEnabled(boolean)
     * @see #setChildrenDrawnWithCacheEnabled(boolean)
     *
```

```
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Child views may no longer be forced to cache their rendering state by their parents.
     * Use {@link View#setLayerType(int, Paint)} on individual Views instead.
     */
    @Deprecated
    protected boolean isChildrenDrawnWithCacheEnabled() {
        return (mGroupFlags & FLAG_CHILDREN_DRAWN_WITH_CACHE) == FLAG_CHILDREN_DRAWN_WITH_CACHE;
    }

    /**
     * Tells the ViewGroup to draw its children using their drawing cache. This property
     * is ignored when {@link #isAlwaysDrawnWithCacheEnabled()} is true. A child's drawing cache
     * will be used only if it has been enabled.
     *
     * Subclasses should call this method to start and stop using the drawing cache when
     * they perform performance sensitive operations, like scrolling or animating.
     *
     * @param enabled true if children should be drawn with their cache, false otherwise
     *
     * @see #setAlwaysDrawnWithCacheEnabled(boolean)
     * @see #isChildrenDrawnWithCacheEnabled()
     *
     * @deprecated As of {@link android.os.Build.VERSION_CODES#M}, this property is ignored.
     * Child views may no longer be forced to cache their rendering state by their parents.
     * Use {@link View#setLayerType(int, Paint)} on individual Views instead.
     */
    @Deprecated
    protected void setChildrenDrawnWithCacheEnabled(boolean enabled) {
        setBooleanFlag(FLAG_CHILDREN_DRAWN_WITH_CACHE, enabled);
    }

    /**
     * Indicates whether the ViewGroup is drawing its children in the order defined by
     * {@link #getChildDrawingOrder(int, int)}.
     *
     * @return true if children drawing order is defined by {@link #getChildDrawingOrder(int, int)},
     *         false otherwise
     *
     * @see #setChildrenDrawingOrderEnabled(boolean)
     * @see #getChildDrawingOrder(int, int)
     */
    @ViewDebug.ExportedProperty(category = "drawing")
    protected boolean isChildrenDrawingOrderEnabled() {
        return (mGroupFlags & FLAG_USE_CHILD_DRAWING_ORDER) == FLAG_USE_CHILD_DRAWING_ORDER;
    }

    /**
     * Tells the ViewGroup whether to draw its children in the order defined by the method
     * {@link #getChildDrawingOrder(int, int)}.
     * <p>
     * Note that {@link View#getZ() Z} reordering, done by {@link #dispatchDraw(Canvas)},
     * will override custom child ordering done via this method.
     *
     * @param enabled true if the order of the children when drawing is determined by
     *        {@link #getChildDrawingOrder(int, int)}, false otherwise
     *
     * @see #isChildrenDrawingOrderEnabled()
     * @see #getChildDrawingOrder(int, int)
     */
    protected void setChildrenDrawingOrderEnabled(boolean enabled) {
        setBooleanFlag(FLAG_USE_CHILD_DRAWING_ORDER, enabled);
    }

    private boolean hasBooleanFlag(int flag) {
        return (mGroupFlags & flag) == flag;
    }

    private void setBooleanFlag(int flag, boolean value) {
        if (value) {
            mGroupFlags |= flag;
        } else {
            mGroupFlags &= ~flag;
        }
    }

    /**
     * Returns an integer indicating what types of drawing caches are kept in memory.
     *
     * @see #setPersistentDrawingCache(int)
     * @see #setAnimationCacheEnabled(boolean)
     *
     * @return one or a combination of {@link #PERSISTENT_NO_CACHE},
```

```
     *         {@link #PERSISTENT_ANIMATION_CACHE}, {@link #PERSISTENT_SCROLLING_CACHE}
     *         and {@link #PERSISTENT_ALL_CACHES}
     */
    @ViewDebug.ExportedProperty(category = "drawing", mapping = {
        @ViewDebug.IntToString(from = PERSISTENT_NO_CACHE,        to = "NONE"),
        @ViewDebug.IntToString(from = PERSISTENT_ANIMATION_CACHE, to = "ANIMATION"),
        @ViewDebug.IntToString(from = PERSISTENT_SCROLLING_CACHE, to = "SCROLLING"),
        @ViewDebug.IntToString(from = PERSISTENT_ALL_CACHES,      to = "ALL")
    })
    public int getPersistentDrawingCache() {
        return mPersistentDrawingCache;
    }

    /**
     * Indicates what types of drawing caches should be kept in memory after
     * they have been created.
     *
     * @see #getPersistentDrawingCache()
     * @see #setAnimationCacheEnabled(boolean)
     *
     * @param drawingCacheToKeep one or a combination of {@link #PERSISTENT_NO_CACHE},
     *        {@link #PERSISTENT_ANIMATION_CACHE}, {@link #PERSISTENT_SCROLLING_CACHE}
     *        and {@link #PERSISTENT_ALL_CACHES}
     */
    public void setPersistentDrawingCache(int drawingCacheToKeep) {
        mPersistentDrawingCache = drawingCacheToKeep & PERSISTENT_ALL_CACHES;
    }

    private void setLayoutMode(int layoutMode, boolean explicitly) {
        mLayoutMode = layoutMode;
        setBooleanFlag(FLAG_LAYOUT_MODE_WAS_EXPLICITLY_SET, explicitly);
    }

    /**
     * Recursively traverse the view hierarchy, resetting the layoutMode of any
     * descendants that had inherited a different layoutMode from a previous parent.
     * Recursion terminates when a descendant's mode is:
     * <ul>
     *     <li>Undefined</li>
     *     <li>The same as the root node's</li>
     *     <li>A mode that had been explicitly set</li>
     * <ul/>
     * The first two clauses are optimizations.
     * @param layoutModeOfRoot
     */
    @Override
    void invalidateInheritedLayoutMode(int layoutModeOfRoot) {
        if (mLayoutMode == LAYOUT_MODE_UNDEFINED ||
            mLayoutMode == layoutModeOfRoot ||
            hasBooleanFlag(FLAG_LAYOUT_MODE_WAS_EXPLICITLY_SET)) {
            return;
        }
        setLayoutMode(LAYOUT_MODE_UNDEFINED, false);

        // apply recursively
        for (int i = 0, N = getChildCount(); i < N; i++) {
            getChildAt(i).invalidateInheritedLayoutMode(layoutModeOfRoot);
        }
    }

    /**
     * Returns the basis of alignment during layout operations on this ViewGroup:
     * either {@link #LAYOUT_MODE_CLIP_BOUNDS} or {@link #LAYOUT_MODE_OPTICAL_BOUNDS}.
     * <p>
     * If no layoutMode was explicitly set, either programmatically or in an XML resource,
     * the method returns the layoutMode of the view's parent ViewGroup if such a parent exists,
     * otherwise the method returns a default value of {@link #LAYOUT_MODE_CLIP_BOUNDS}.
     *
     * @return the layout mode to use during layout operations
     *
     * @see #setLayoutMode(int)
     */
    public int getLayoutMode() {
        if (mLayoutMode == LAYOUT_MODE_UNDEFINED) {
            int inheritedLayoutMode = (mParent instanceof ViewGroup) ?
                    ((ViewGroup) mParent).getLayoutMode() : LAYOUT_MODE_DEFAULT;
            setLayoutMode(inheritedLayoutMode, false);
        }
        return mLayoutMode;
    }

    /**
```

```
     * Sets the basis of alignment during the layout of this ViewGroup.
     * Valid values are either {@link #LAYOUT_MODE_CLIP_BOUNDS} or
     * {@link #LAYOUT_MODE_OPTICAL_BOUNDS}.
     *
     * @param layoutMode the layout mode to use during layout operations
     *
     * @see #getLayoutMode()
     * @attr ref android.R.styleable#ViewGroup_layoutMode
     */
    public void setLayoutMode(int layoutMode) {
        if (mLayoutMode != layoutMode) {
            invalidateInheritedLayoutMode(layoutMode);
            setLayoutMode(layoutMode, layoutMode != LAYOUT_MODE_UNDEFINED);
            requestLayout();
        }
    }

    /**
     * Returns a new set of layout parameters based on the supplied attributes set.
     *
     * @param attrs the attributes to build the layout parameters from
     *
     * @return an instance of {@link android.view.ViewGroup.LayoutParams} or one
     *         of its descendants
     */
    public LayoutParams generateLayoutParams(AttributeSet attrs) {
        return new LayoutParams(getContext(), attrs);
    }

    /**
     * Returns a safe set of layout parameters based on the supplied layout params.
     * When a ViewGroup is passed a View whose layout params do not pass the test of
     * {@link #checkLayoutParams(android.view.ViewGroup.LayoutParams)}, this method
     * is invoked. This method should return a new set of layout params suitable for
     * this ViewGroup, possibly by copying the appropriate attributes from the
     * specified set of layout params.
     *
     * @param p The layout parameters to convert into a suitable set of layout parameters
     *          for this ViewGroup.
     *
     * @return an instance of {@link android.view.ViewGroup.LayoutParams} or one
     *         of its descendants
     */
    protected LayoutParams generateLayoutParams(ViewGroup.LayoutParams p) {
        return p;
    }

    /**
     * Returns a set of default layout parameters. These parameters are requested
     * when the View passed to {@link #addView(View)} has no layout parameters
     * already set. If null is returned, an exception is thrown from addView.
     *
     * @return a set of default layout parameters or null
     */
    protected LayoutParams generateDefaultLayoutParams() {
        return new LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT);
    }

    @Override
    protected void debug(int depth) {
        super.debug(depth);
        String output;

        if (mFocused != null) {
            output = debugIndent(depth);
            output += "mFocused";
            Log.d(VIEW_LOG_TAG, output);
            mFocused.debug(depth + 1);
        }
        if (mDefaultFocus != null) {
            output = debugIndent(depth);
            output += "mDefaultFocus";
            Log.d(VIEW_LOG_TAG, output);
            mDefaultFocus.debug(depth + 1);
        }
        if (mFocusedInCluster != null) {
            output = debugIndent(depth);
            output += "mFocusedInCluster";
            Log.d(VIEW_LOG_TAG, output);
            mFocusedInCluster.debug(depth + 1);
        }
        if (mChildrenCount != 0) {
```

```
        output = debugIndent(depth);
        output += "{";
        Log.d(VIEW_LOG_TAG, output);
    }
    int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        View child = mChildren[i];
        child.debug(depth + 1);
    }

    if (mChildrenCount != 0) {
        output = debugIndent(depth);
        output += "}";
        Log.d(VIEW_LOG_TAG, output);
    }
}

/**
 * Returns the position in the group of the specified child view.
 *
 * @param child the view for which to get the position
 * @return a positive integer representing the position of the view in the
 *         group, or -1 if the view does not exist in the group
 */
public int indexOfChild(View child) {
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        if (children[i] == child) {
            return i;
        }
    }
    return -1;
}

/**
 * Returns the number of children in the group.
 *
 * @return a positive integer representing the number of children in
 *         the group
 */
public int getChildCount() {
    return mChildrenCount;
}

/**
 * Returns the view at the specified position in the group.
 *
 * @param index the position at which to get the view from
 * @return the view at the specified position or null if the position
 *         does not exist within the group
 */
public View getChildAt(int index) {
    if (index < 0 || index >= mChildrenCount) {
        return null;
    }
    return mChildren[index];
}

/**
 * Ask all of the children of this view to measure themselves, taking into
 * account both the MeasureSpec requirements for this view and its padding.
 * We skip children that are in the GONE state The heavy lifting is done in
 * getChildMeasureSpec.
 *
 * @param widthMeasureSpec The width requirements for this view
 * @param heightMeasureSpec The height requirements for this view
 */
protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {
    final int size = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < size; ++i) {
        final View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {
            measureChild(child, widthMeasureSpec, heightMeasureSpec);
        }
    }
}

/**
 * Ask one of the children of this view to measure itself, taking into
 * account both the MeasureSpec requirements for this view and its padding.
```

```
     * The heavy lifting is done in getChildMeasureSpec.
     *
     * @param child The child to measure
     * @param parentWidthMeasureSpec The width requirements for this view
     * @param parentHeightMeasureSpec The height requirements for this view
     */
    protected void measureChild(View child, int parentWidthMeasureSpec,
            int parentHeightMeasureSpec) {
        final LayoutParams lp = child.getLayoutParams();

        final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
                mPaddingLeft + mPaddingRight, lp.width);
        final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
                mPaddingTop + mPaddingBottom, lp.height);

        child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
    }

    /**
     * Ask one of the children of this view to measure itself, taking into
     * account both the MeasureSpec requirements for this view and its padding
     * and margins. The child must have MarginLayoutParams The heavy lifting is
     * done in getChildMeasureSpec.
     *
     * @param child The child to measure
     * @param parentWidthMeasureSpec The width requirements for this view
     * @param widthUsed Extra space that has been used up by the parent
     *        horizontally (possibly by other children of the parent)
     * @param parentHeightMeasureSpec The height requirements for this view
     * @param heightUsed Extra space that has been used up by the parent
     *        vertically (possibly by other children of the parent)
     */
    protected void measureChildWithMargins(View child,
            int parentWidthMeasureSpec, int widthUsed,
            int parentHeightMeasureSpec, int heightUsed) {
        final MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();

        final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
                mPaddingLeft + mPaddingRight + lp.leftMargin + lp.rightMargin
                        + widthUsed, lp.width);
        final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
                mPaddingTop + mPaddingBottom + lp.topMargin + lp.bottomMargin
                        + heightUsed, lp.height);

        child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
    }

    /**
     * Does the hard part of measureChildren: figuring out the MeasureSpec to
     * pass to a particular child. This method figures out the right MeasureSpec
     * for one dimension (height or width) of one child view.
     *
     * The goal is to combine information from our MeasureSpec with the
     * LayoutParams of the child to get the best possible results. For example,
     * if the this view knows its size (because its MeasureSpec has a mode of
     * EXACTLY), and the child has indicated in its LayoutParams that it wants
     * to be the same size as the parent, the parent should ask the child to
     * layout given an exact size.
     *
     * @param spec The requirements for this view
     * @param padding The padding of this view for the current dimension and
     *        margins, if applicable
     * @param childDimension How big the child wants to be in the current
     *        dimension
     * @return a MeasureSpec integer for the child
     */
    public static int getChildMeasureSpec(int spec, int padding, int childDimension) {
        int specMode = MeasureSpec.getMode(spec);
        int specSize = MeasureSpec.getSize(spec);

        int size = Math.max(0, specSize - padding);

        int resultSize = 0;
        int resultMode = 0;

        switch (specMode) {
        // Parent has imposed an exact size on us
        case MeasureSpec.EXACTLY:
            if (childDimension >= 0) {
                resultSize = childDimension;
                resultMode = MeasureSpec.EXACTLY;
            } else if (childDimension == LayoutParams.MATCH_PARENT) {
```

```
            // Child wants to be our size. So be it.
            resultSize = size;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size. It can't be
            // bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        }
        break;

    // Parent has imposed a maximum size on us
    case MeasureSpec.AT_MOST:
        if (childDimension >= 0) {
            // Child wants a specific size... so be it
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.MATCH_PARENT) {
            // Child wants to be our size, but our size is not fixed.
            // Constrain child to not be bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size. It can't be
            // bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        }
        break;

    // Parent asked to see how big we want to be
    case MeasureSpec.UNSPECIFIED:
        if (childDimension >= 0) {
            // Child wants a specific size... let him have it
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.MATCH_PARENT) {
            // Child wants to be our size... find out how big it should
            // be
            resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
            resultMode = MeasureSpec.UNSPECIFIED;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size.... find out how
            // big it should be
            resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
            resultMode = MeasureSpec.UNSPECIFIED;
        }
        break;
    }
    //noinspection ResourceType
    return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
}


/**
 * Removes any pending animations for views that have been removed. Call
 * this if you don't want animations for exiting views to stack up.
 */
public void clearDisappearingChildren() {
    final ArrayList<View> disappearingChildren = mDisappearingChildren;
    if (disappearingChildren != null) {
        final int count = disappearingChildren.size();
        for (int i = 0; i < count; i++) {
            final View view = disappearingChildren.get(i);
            if (view.mAttachInfo != null) {
                view.dispatchDetachedFromWindow();
            }
            view.clearAnimation();
        }
        disappearingChildren.clear();
        invalidate();
    }
}

/**
 * Add a view which is removed from mChildren but still needs animation
 *
 * @param v View to add
 */
private void addDisappearingView(View v) {
    ArrayList<View> disappearingChildren = mDisappearingChildren;
```

```
    if (disappearingChildren == null) {
        disappearingChildren = mDisappearingChildren = new ArrayList<View>();
    }

    disappearingChildren.add(v);
}

/**
 * Cleanup a view when its animation is done. This may mean removing it from
 * the list of disappearing views.
 *
 * @param view The view whose animation has finished
 * @param animation The animation, cannot be null
 */
void finishAnimatingView(final View view, Animation animation) {
    final ArrayList<View> disappearingChildren = mDisappearingChildren;
    if (disappearingChildren != null) {
        if (disappearingChildren.contains(view)) {
            disappearingChildren.remove(view);

            if (view.mAttachInfo != null) {
                view.dispatchDetachedFromWindow();
            }

            view.clearAnimation();
            mGroupFlags |= FLAG_INVALIDATE_REQUIRED;
        }
    }

    if (animation != null && !animation.getFillAfter()) {
        view.clearAnimation();
    }

    if ((view.mPrivateFlags & PFLAG_ANIMATION_STARTED) == PFLAG_ANIMATION_STARTED) {
        view.onAnimationEnd();
        // Should be performed by onAnimationEnd() but this avoid an infinite loop,
        // so we'd rather be safe than sorry
        view.mPrivateFlags &= ~PFLAG_ANIMATION_STARTED;
        // Draw one more frame after the animation is done
        mGroupFlags |= FLAG_INVALIDATE_REQUIRED;
    }
}

/**
 * Utility function called by View during invalidation to determine whether a view that
 * is invisible or gone should still be invalidated because it is being transitioned (and
 * therefore still needs to be drawn).
 */
boolean isViewTransitioning(View view) {
    return (mTransitioningViews != null && mTransitioningViews.contains(view));
}

/**
 * This method tells the ViewGroup that the given View object, which should have this
 * ViewGroup as its parent,
 * should be kept around  (re-displayed when the ViewGroup draws its children) even if it
 * is removed from its parent. This allows animations, such as those used by
 * {@link android.app.Fragment} and {@link android.animation.LayoutTransition} to animate
 * the removal of views. A call to this method should always be accompanied by a later call
 * to {@link #endViewTransition(View)}, such as after an animation on the View has finished,
 * so that the View finally gets removed.
 *
 * @param view The View object to be kept visible even if it gets removed from its parent.
 */
public void startViewTransition(View view) {
    if (view.mParent == this) {
        if (mTransitioningViews == null) {
            mTransitioningViews = new ArrayList<View>();
        }
        mTransitioningViews.add(view);
    }
}

/**
 * This method should always be called following an earlier call to
 * {@link #startViewTransition(View)}. The given View is finally removed from its parent
 * and will no longer be displayed. Note that this method does not perform the functionality
 * of removing a view from its parent; it just discontinues the display of a View that
 * has previously been removed.
 *
 * @return view The View object that has been removed but is being kept around in the visible
 * hierarchy by an earlier call to {@link #startViewTransition(View)}.
```

```
     */
    public void endViewTransition(View view) {
        if (mTransitioningViews != null) {
            mTransitioningViews.remove(view);
            final ArrayList<View> disappearingChildren = mDisappearingChildren;
            if (disappearingChildren != null && disappearingChildren.contains(view)) {
                disappearingChildren.remove(view);
                if (mVisibilityChangingChildren != null &&
                        mVisibilityChangingChildren.contains(view)) {
                    mVisibilityChangingChildren.remove(view);
                } else {
                    if (view.mAttachInfo != null) {
                        view.dispatchDetachedFromWindow();
                    }
                    if (view.mParent != null) {
                        view.mParent = null;
                    }
                }
                invalidate();
            }
        }
    }

    private LayoutTransition.TransitionListener mLayoutTransitionListener =
            new LayoutTransition.TransitionListener() {
        @Override
        public void startTransition(LayoutTransition transition, ViewGroup container,
                View view, int transitionType) {
            // We only care about disappearing items, since we need special logic to keep
            // those items visible after they've been 'removed'
            if (transitionType == LayoutTransition.DISAPPEARING) {
                startViewTransition(view);
            }
        }

        @Override
        public void endTransition(LayoutTransition transition, ViewGroup container,
                View view, int transitionType) {
            if (mLayoutCalledWhileSuppressed && !transition.isChangingLayout()) {
                requestLayout();
                mLayoutCalledWhileSuppressed = false;
            }
            if (transitionType == LayoutTransition.DISAPPEARING && mTransitioningViews != null) {
                endViewTransition(view);
            }
        }
    };

    /**
     * Tells this ViewGroup to suppress all layout() calls until layout
     * suppression is disabled with a later call to suppressLayout(false).
     * When layout suppression is disabled, a requestLayout() call is sent
     * if layout() was attempted while layout was being suppressed.
     *
     * @hide
     */
    public void suppressLayout(boolean suppress) {
        mSuppressLayout = suppress;
        if (!suppress) {
            if (mLayoutCalledWhileSuppressed) {
                requestLayout();
                mLayoutCalledWhileSuppressed = false;
            }
        }
    }

    /**
     * Returns whether layout calls on this container are currently being
     * suppressed, due to an earlier call to {@link #suppressLayout(boolean)}.
     *
     * @return true if layout calls are currently suppressed, false otherwise.
     *
     * @hide
     */
    public boolean isLayoutSuppressed() {
        return mSuppressLayout;
    }

    @Override
    public boolean gatherTransparentRegion(Region region) {
        // If no transparent regions requested, we are always opaque.
        final boolean meOpaque = (mPrivateFlags & View.PFLAG_REQUEST_TRANSPARENT_REGIONS) == 0;
```

```
    if (meOpaque && region == null) {
        // The caller doesn't care about the region, so stop now.
        return true;
    }
    super.gatherTransparentRegion(region);
    // Instead of naively traversing the view tree, we have to traverse according to the Z
    // order here. We need to go with the same order as dispatchDraw().
    // One example is that after surfaceView punch a hole, we will still allow other views drawn
    // on top of that hole. In this case, those other views should be able to cut the
    // transparent region into smaller area.
    final int childrenCount = mChildrenCount;
    boolean noneOfTheChildrenAreTransparent = true;
    if (childrenCount > 0) {
        final ArrayList<View> preorderedList = buildOrderedChildList();
        final boolean customOrder = preorderedList == null
                && isChildrenDrawingOrderEnabled();
        final View[] children = mChildren;
        for (int i = 0; i < childrenCount; i++) {
            final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
            final View child = getAndVerifyPreorderedView(preorderedList, children, childIndex);
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null) {
                if (!child.gatherTransparentRegion(region)) {
                    noneOfTheChildrenAreTransparent = false;
                }
            }
        }
        if (preorderedList != null) preorderedList.clear();
    }
    return meOpaque || noneOfTheChildrenAreTransparent;
}

@Override
public void requestTransparentRegion(View child) {
    if (child != null) {
        child.mPrivateFlags |= View.PFLAG_REQUEST_TRANSPARENT_REGIONS;
        if (mParent != null) {
            mParent.requestTransparentRegion(this);
        }
    }
}

@Override
public WindowInsets dispatchApplyWindowInsets(WindowInsets insets) {
    insets = super.dispatchApplyWindowInsets(insets);
    if (!insets.isConsumed()) {
        final int count = getChildCount();
        for (int i = 0; i < count; i++) {
            insets = getChildAt(i).dispatchApplyWindowInsets(insets);
            if (insets.isConsumed()) {
                break;
            }
        }
    }
    return insets;
}

/**
 * Returns the animation listener to which layout animation events are
 * sent.
 *
 * @return an {@link android.view.animation.Animation.AnimationListener}
 */
public Animation.AnimationListener getLayoutAnimationListener() {
    return mAnimationListener;
}

@Override
protected void drawableStateChanged() {
    super.drawableStateChanged();

    if ((mGroupFlags & FLAG_NOTIFY_CHILDREN_ON_DRAWABLE_STATE_CHANGE) != 0) {
        if ((mGroupFlags & FLAG_ADD_STATES_FROM_CHILDREN) != 0) {
            throw new IllegalStateException("addStateFromChildren cannot be enabled if a"
                    + " child has duplicateParentState set to true");
        }

        final View[] children = mChildren;
        final int count = mChildrenCount;

        for (int i = 0; i < count; i++) {
            final View child = children[i];
            if ((child.mViewFlags & DUPLICATE_PARENT_STATE) != 0) {
```

```
                child.refreshDrawableState();
            }
        }
    }
}

@Override
public void jumpDrawablesToCurrentState() {
    super.jumpDrawablesToCurrentState();
    final View[] children = mChildren;
    final int count = mChildrenCount;
    for (int i = 0; i < count; i++) {
        children[i].jumpDrawablesToCurrentState();
    }
}

@Override
protected int[] onCreateDrawableState(int extraSpace) {
    if ((mGroupFlags & FLAG_ADD_STATES_FROM_CHILDREN) == 0) {
        return super.onCreateDrawableState(extraSpace);
    }

    int need = 0;
    int n = getChildCount();
    for (int i = 0; i < n; i++) {
        int[] childState = getChildAt(i).getDrawableState();

        if (childState != null) {
            need += childState.length;
        }
    }

    int[] state = super.onCreateDrawableState(extraSpace + need);

    for (int i = 0; i < n; i++) {
        int[] childState = getChildAt(i).getDrawableState();

        if (childState != null) {
            state = mergeDrawableStates(state, childState);
        }
    }

    return state;
}

/**
 * Sets whether this ViewGroup's drawable states also include
 * its children's drawable states.  This is used, for example, to
 * make a group appear to be focused when its child EditText or button
 * is focused.
 */
public void setAddStatesFromChildren(boolean addsStates) {
    if (addsStates) {
        mGroupFlags |= FLAG_ADD_STATES_FROM_CHILDREN;
    } else {
        mGroupFlags &= ~FLAG_ADD_STATES_FROM_CHILDREN;
    }

    refreshDrawableState();
}

/**
 * Returns whether this ViewGroup's drawable states also include
 * its children's drawable states.  This is used, for example, to
 * make a group appear to be focused when its child EditText or button
 * is focused.
 */
public boolean addStatesFromChildren() {
    return (mGroupFlags & FLAG_ADD_STATES_FROM_CHILDREN) != 0;
}

/**
 * If {@link #addStatesFromChildren} is true, refreshes this group's
 * drawable state (to include the states from its children).
 */
@Override
public void childDrawableStateChanged(View child) {
    if ((mGroupFlags & FLAG_ADD_STATES_FROM_CHILDREN) != 0) {
        refreshDrawableState();
    }
}
```

```java
/**
 * Specifies the animation listener to which layout animation events must
 * be sent. Only
 * {@link android.view.animation.Animation.AnimationListener#onAnimationStart(Animation)}
 * and
 * {@link android.view.animation.Animation.AnimationListener#onAnimationEnd(Animation)}
 * are invoked.
 *
 * @param animationListener the layout animation listener
 */
public void setLayoutAnimationListener(Animation.AnimationListener animationListener) {
    mAnimationListener = animationListener;
}

/**
 * This method is called by LayoutTransition when there are 'changing' animations that need
 * to start after the layout/setup phase. The request is forwarded to the ViewAncestor, who
 * starts all pending transitions prior to the drawing phase in the current traversal.
 *
 * @param transition The LayoutTransition to be started on the next traversal.
 *
 * @hide
 */
public void requestTransitionStart(LayoutTransition transition) {
    ViewRootImpl viewAncestor = getViewRootImpl();
    if (viewAncestor != null) {
        viewAncestor.requestTransitionStart(transition);
    }
}

/**
 * @hide
 */
@Override
public boolean resolveRtlPropertiesIfNeeded() {
    final boolean result = super.resolveRtlPropertiesIfNeeded();
    // We dont need to resolve the children RTL properties if nothing has changed for the parent
    if (result) {
        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isLayoutDirectionInherited()) {
                child.resolveRtlPropertiesIfNeeded();
            }
        }
    }
    return result;
}

/**
 * @hide
 */
@Override
public boolean resolveLayoutDirection() {
    final boolean result = super.resolveLayoutDirection();
    if (result) {
        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isLayoutDirectionInherited()) {
                child.resolveLayoutDirection();
            }
        }
    }
    return result;
}

/**
 * @hide
 */
@Override
public boolean resolveTextDirection() {
    final boolean result = super.resolveTextDirection();
    if (result) {
        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isTextDirectionInherited()) {
                child.resolveTextDirection();
            }
        }
    }
```

```
    return result;
}

/**
 * @hide
 */
@Override
public boolean resolveTextAlignment() {
    final boolean result = super.resolveTextAlignment();
    if (result) {
        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isTextAlignmentInherited()) {
                child.resolveTextAlignment();
            }
        }
    }
    return result;
}

/**
 * @hide
 */
@Override
public void resolvePadding() {
    super.resolvePadding();
    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        if (child.isLayoutDirectionInherited() && !child.isPaddingResolved()) {
            child.resolvePadding();
        }
    }
}

/**
 * @hide
 */
@Override
protected void resolveDrawables() {
    super.resolveDrawables();
    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        if (child.isLayoutDirectionInherited() && !child.areDrawablesResolved()) {
            child.resolveDrawables();
        }
    }
}

/**
 * @hide
 */
@Override
public void resolveLayoutParams() {
    super.resolveLayoutParams();
    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        child.resolveLayoutParams();
    }
}

/**
 * @hide
 */
@Override
public void resetResolvedLayoutDirection() {
    super.resetResolvedLayoutDirection();

    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        if (child.isLayoutDirectionInherited()) {
            child.resetResolvedLayoutDirection();
        }
    }
}

/**
 * @hide
```

```
     */
    @Override
    public void resetResolvedTextDirection() {
        super.resetResolvedTextDirection();

        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isTextDirectionInherited()) {
                child.resetResolvedTextDirection();
            }
        }
    }

    /**
     * @hide
     */
    @Override
    public void resetResolvedTextAlignment() {
        super.resetResolvedTextAlignment();

        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isTextAlignmentInherited()) {
                child.resetResolvedTextAlignment();
            }
        }
    }

    /**
     * @hide
     */
    @Override
    public void resetResolvedPadding() {
        super.resetResolvedPadding();

        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isLayoutDirectionInherited()) {
                child.resetResolvedPadding();
            }
        }
    }

    /**
     * @hide
     */
    @Override
    protected void resetResolvedDrawables() {
        super.resetResolvedDrawables();

        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.isLayoutDirectionInherited()) {
                child.resetResolvedDrawables();
            }
        }
    }

    /**
     * Return true if the pressed state should be delayed for children or descendants of this
     * ViewGroup. Generally, this should be done for containers that can scroll, such as a List.
     * This prevents the pressed state from appearing when the user is actually trying to scroll
     * the content.
     *
     * The default implementation returns true for compatibility reasons. Subclasses that do
     * not scroll should generally override this method and return false.
     */
    public boolean shouldDelayChildPressedState() {
        return true;
    }

    /**
     * @inheritDoc
     */
    @Override
    public boolean onStartNestedScroll(View child, View target, int nestedScrollAxes) {
        return false;
    }
```

```java
/**
 * @inheritDoc
 */
@Override
public void onNestedScrollAccepted(View child, View target, int axes) {
    mNestedScrollAxes = axes;
}

/**
 * @inheritDoc
 *
 * <p>The default implementation of onStopNestedScroll calls
 * {@link #stopNestedScroll()} to halt any recursive nested scrolling in progress.</p>
 */
@Override
public void onStopNestedScroll(View child) {
    // Stop any recursive nested scrolling.
    stopNestedScroll();
    mNestedScrollAxes = 0;
}

/**
 * @inheritDoc
 */
@Override
public void onNestedScroll(View target, int dxConsumed, int dyConsumed,
        int dxUnconsumed, int dyUnconsumed) {
    // Re-dispatch up the tree by default
    dispatchNestedScroll(dxConsumed, dyConsumed, dxUnconsumed, dyUnconsumed, null);
}

/**
 * @inheritDoc
 */
@Override
public void onNestedPreScroll(View target, int dx, int dy, int[] consumed) {
    // Re-dispatch up the tree by default
    dispatchNestedPreScroll(dx, dy, consumed, null);
}

/**
 * @inheritDoc
 */
@Override
public boolean onNestedFling(View target, float velocityX, float velocityY, boolean consumed) {
    // Re-dispatch up the tree by default
    return dispatchNestedFling(velocityX, velocityY, consumed);
}

/**
 * @inheritDoc
 */
@Override
public boolean onNestedPreFling(View target, float velocityX, float velocityY) {
    // Re-dispatch up the tree by default
    return dispatchNestedPreFling(velocityX, velocityY);
}

/**
 * Return the current axes of nested scrolling for this ViewGroup.
 *
 * <p>A ViewGroup returning something other than {@link #SCROLL_AXIS_NONE} is currently
 * acting as a nested scrolling parent for one or more descendant views in the hierarchy.</p>
 *
 * @return Flags indicating the current axes of nested scrolling
 * @see #SCROLL_AXIS_HORIZONTAL
 * @see #SCROLL_AXIS_VERTICAL
 * @see #SCROLL_AXIS_NONE
 */
public int getNestedScrollAxes() {
    return mNestedScrollAxes;
}

/** @hide */
protected void onSetLayoutParams(View child, LayoutParams layoutParams) {
    requestLayout();
}

/** @hide */
@Override
public void captureTransitioningViews(List<View> transitioningViews) {
```

```
    if (getVisibility() != View.VISIBLE) {
        return;
    }
    if (isTransitionGroup()) {
        transitioningViews.add(this);
    } else {
        int count = getChildCount();
        for (int i = 0; i < count; i++) {
            View child = getChildAt(i);
            child.captureTransitioningViews(transitioningViews);
        }
    }
}

/** @hide */
@Override
public void findNamedViews(Map<String, View> namedElements) {
    if (getVisibility() != VISIBLE && mGhostView == null) {
        return;
    }
    super.findNamedViews(namedElements);
    int count = getChildCount();
    for (int i = 0; i < count; i++) {
        View child = getChildAt(i);
        child.findNamedViews(namedElements);
    }
}

/**
 * LayoutParams are used by views to tell their parents how they want to be
 * laid out. See
 * {@link android.R.styleable#ViewGroup_Layout ViewGroup Layout Attributes}
 * for a list of all child view attributes that this class supports.
 *
 * <p>
 * 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>FILL_PARENT (renamed MATCH_PARENT in API Level 8 and higher), which
 * means that 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)
 * <li> an exact number
 * </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>
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>For more information about creating user interface layouts, read the
 * <a href="{@docRoot}guide/topics/ui/declaring-layout.html">XML Layouts</a> developer
 * guide.</p></div>
 *
 * @attr ref android.R.styleable#ViewGroup_Layout_layout_height
 * @attr ref android.R.styleable#ViewGroup_Layout_layout_width
 */
public static class LayoutParams {
    /**
     * Special value for the height or width requested by a View.
     * FILL_PARENT means that the view wants to be as big as its parent,
     * minus the parent's padding, if any. This value is deprecated
     * starting in API Level 8 and replaced by {@link #MATCH_PARENT}.
     */
    @SuppressWarnings({"UnusedDeclaration"})
    @Deprecated
    public static final int FILL_PARENT = -1;

    /**
     * Special value for the height or width requested by a View.
     * MATCH_PARENT means that the view wants to be as big as its parent,
     * minus the parent's padding, if any. Introduced in API Level 8.
     */
    public static final int MATCH_PARENT = -1;

    /**
     * Special value for the height or width requested by a View.
     * WRAP_CONTENT means that the view wants to be just large enough to fit
     * its own internal content, taking its own padding into account.
     */
    public static final int WRAP_CONTENT = -2;
```

```
/**
 * Information about how wide the view wants to be. Can be one of the
 * constants FILL_PARENT (replaced by MATCH_PARENT
 * in API Level 8) or WRAP_CONTENT, or an exact size.
 */
@ViewDebug.ExportedProperty(category = "layout", mapping = {
    @ViewDebug.IntToString(from = MATCH_PARENT, to = "MATCH_PARENT"),
    @ViewDebug.IntToString(from = WRAP_CONTENT, to = "WRAP_CONTENT")
})
public int width;

/**
 * Information about how tall the view wants to be. Can be one of the
 * constants FILL_PARENT (replaced by MATCH_PARENT
 * in API Level 8) or WRAP_CONTENT, or an exact size.
 */
@ViewDebug.ExportedProperty(category = "layout", mapping = {
    @ViewDebug.IntToString(from = MATCH_PARENT, to = "MATCH_PARENT"),
    @ViewDebug.IntToString(from = WRAP_CONTENT, to = "WRAP_CONTENT")
})
public int height;

/**
 * Used to animate layouts.
 */
public LayoutAnimationController.AnimationParameters layoutAnimationParameters;

/**
 * Creates a new set of layout parameters. The values are extracted from
 * the supplied attributes set and context. The XML attributes mapped
 * to this set of layout parameters are:
 *
 * <ul>
 *   <li><code>layout_width</code>: the width, either an exact value,
 *   {@link #WRAP_CONTENT}, or {@link #FILL_PARENT} (replaced by
 *   {@link #MATCH_PARENT} in API Level 8)</li>
 *   <li><code>layout_height</code>: the height, either an exact value,
 *   {@link #WRAP_CONTENT}, or {@link #FILL_PARENT} (replaced by
 *   {@link #MATCH_PARENT} in API Level 8)</li>
 * </ul>
 *
 * @param c the application environment
 * @param attrs the set of attributes from which to extract the layout
 *              parameters' values
 */
public LayoutParams(Context c, AttributeSet attrs) {
    TypedArray a = c.obtainStyledAttributes(attrs, R.styleable.ViewGroup_Layout);
    setBaseAttributes(a,
            R.styleable.ViewGroup_Layout_layout_width,
            R.styleable.ViewGroup_Layout_layout_height);
    a.recycle();
}

/**
 * Creates a new set of layout parameters with the specified width
 * and height.
 *
 * @param width the width, either {@link #WRAP_CONTENT},
 *        {@link #FILL_PARENT} (replaced by {@link #MATCH_PARENT} in
 *        API Level 8), or a fixed size in pixels
 * @param height the height, either {@link #WRAP_CONTENT},
 *        {@link #FILL_PARENT} (replaced by {@link #MATCH_PARENT} in
 *        API Level 8), or a fixed size in pixels
 */
public LayoutParams(int width, int height) {
    this.width = width;
    this.height = height;
}

/**
 * Copy constructor. Clones the width and height values of the source.
 *
 * @param source The layout params to copy from.
 */
public LayoutParams(LayoutParams source) {
    this.width = source.width;
    this.height = source.height;
}

/**
 * Used internally by MarginLayoutParams.
 * @hide
```

```
     */
    LayoutParams() {
    }

    /**
     * Extracts the layout parameters from the supplied attributes.
     *
     * @param a the style attributes to extract the parameters from
     * @param widthAttr the identifier of the width attribute
     * @param heightAttr the identifier of the height attribute
     */
    protected void setBaseAttributes(TypedArray a, int widthAttr, int heightAttr) {
        width = a.getLayoutDimension(widthAttr, "layout_width");
        height = a.getLayoutDimension(heightAttr, "layout_height");
    }

    /**
     * Resolve layout parameters depending on the layout direction. Subclasses that care about
     * layoutDirection changes should override this method. The default implementation does
     * nothing.
     *
     * @param layoutDirection the direction of the layout
     *
     * {@link View#LAYOUT_DIRECTION_LTR}
     * {@link View#LAYOUT_DIRECTION_RTL}
     */
    public void resolveLayoutDirection(int layoutDirection) {
    }

    /**
     * Returns a String representation of this set of layout parameters.
     *
     * @param output the String to prepend to the internal representation
     * @return a String with the following format: output +
     *         "ViewGroup.LayoutParams={ width=WIDTH, height=HEIGHT }"
     *
     * @hide
     */
    public String debug(String output) {
        return output + "ViewGroup.LayoutParams={ width="
                + sizeToString(width) + ", height=" + sizeToString(height) + " }";
    }

    /**
     * Use {@code canvas} to draw suitable debugging annotations for these LayoutParameters.
     *
     * @param view the view that contains these layout parameters
     * @param canvas the canvas on which to draw
     *
     * @hide
     */
    public void onDebugDraw(View view, Canvas canvas, Paint paint) {
    }

    /**
     * Converts the specified size to a readable String.
     *
     * @param size the size to convert
     * @return a String instance representing the supplied size
     *
     * @hide
     */
    protected static String sizeToString(int size) {
        if (size == WRAP_CONTENT) {
            return "wrap-content";
        }
        if (size == MATCH_PARENT) {
            return "match-parent";
        }
        return String.valueOf(size);
    }

    /** @hide */
    void encode(@NonNull ViewHierarchyEncoder encoder) {
        encoder.beginObject(this);
        encodeProperties(encoder);
        encoder.endObject();
    }

    /** @hide */
    protected void encodeProperties(@NonNull ViewHierarchyEncoder encoder) {
        encoder.addProperty("width", width);
```

```
            encoder.addProperty("height", height);
        }
    }

    /**
     * Per-child layout information for layouts that support margins.
     * See
     * {@link android.R.styleable#ViewGroup_MarginLayout ViewGroup Margin Layout Attributes}
     * for a list of all child view attributes that this class supports.
     *
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_margin
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginHorizontal
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginVertical
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginLeft
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginTop
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginRight
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginBottom
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginStart
     * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginEnd
     */
    public static class MarginLayoutParams extends ViewGroup.LayoutParams {
        /**
         * The left margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        public int leftMargin;

        /**
         * The top margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        public int topMargin;

        /**
         * The right margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        public int rightMargin;

        /**
         * The bottom margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        public int bottomMargin;

        /**
         * The start margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        private int startMargin = DEFAULT_MARGIN_RELATIVE;

        /**
         * The end margin in pixels of the child. Margin values should be positive.
         * Call {@link ViewGroup#setLayoutParams(LayoutParams)} after reassigning a new value
         * to this field.
         */
        @ViewDebug.ExportedProperty(category = "layout")
        private int endMargin = DEFAULT_MARGIN_RELATIVE;

        /**
         * The default start and end margin.
         * @hide
         */
        public static final int DEFAULT_MARGIN_RELATIVE = Integer.MIN_VALUE;

        /**
         * Bit  0: layout direction
         * Bit  1: layout direction
         * Bit  2: left margin undefined
         * Bit  3: right margin undefined
         * Bit  4: is RTL compatibility mode
         * Bit  5: need resolution
```

```
     *
     * Bit 6 to 7 not used
     *
     * @hide
     */
    @ViewDebug.ExportedProperty(category = "layout", flagMapping = {
            @ViewDebug.FlagToString(mask = LAYOUT_DIRECTION_MASK,
                    equals = LAYOUT_DIRECTION_MASK, name = "LAYOUT_DIRECTION"),
            @ViewDebug.FlagToString(mask = LEFT_MARGIN_UNDEFINED_MASK,
                    equals = LEFT_MARGIN_UNDEFINED_MASK, name = "LEFT_MARGIN_UNDEFINED_MASK"),
            @ViewDebug.FlagToString(mask = RIGHT_MARGIN_UNDEFINED_MASK,
                    equals = RIGHT_MARGIN_UNDEFINED_MASK, name = "RIGHT_MARGIN_UNDEFINED_MASK"),
            @ViewDebug.FlagToString(mask = RTL_COMPATIBILITY_MODE_MASK,
                    equals = RTL_COMPATIBILITY_MODE_MASK, name = "RTL_COMPATIBILITY_MODE_MASK"),
            @ViewDebug.FlagToString(mask = NEED_RESOLUTION_MASK,
                    equals = NEED_RESOLUTION_MASK, name = "NEED_RESOLUTION_MASK")
    }, formatToHexString = true)
    byte mMarginFlags;

    private static final int LAYOUT_DIRECTION_MASK = 0x00000003;
    private static final int LEFT_MARGIN_UNDEFINED_MASK = 0x00000004;
    private static final int RIGHT_MARGIN_UNDEFINED_MASK = 0x00000008;
    private static final int RTL_COMPATIBILITY_MODE_MASK = 0x00000010;
    private static final int NEED_RESOLUTION_MASK = 0x00000020;

    private static final int DEFAULT_MARGIN_RESOLVED = 0;
    private static final int UNDEFINED_MARGIN = DEFAULT_MARGIN_RELATIVE;

    /**
     * Creates a new set of layout parameters. The values are extracted from
     * the supplied attributes set and context.
     *
     * @param c the application environment
     * @param attrs the set of attributes from which to extract the layout
     *              parameters' values
     */
    public MarginLayoutParams(Context c, AttributeSet attrs) {
        super();

        TypedArray a = c.obtainStyledAttributes(attrs, R.styleable.ViewGroup_MarginLayout);
        setBaseAttributes(a,
                R.styleable.ViewGroup_MarginLayout_layout_width,
                R.styleable.ViewGroup_MarginLayout_layout_height);

        int margin = a.getDimensionPixelSize(
                com.android.internal.R.styleable.ViewGroup_MarginLayout_layout_margin, -1);
        if (margin >= 0) {
            leftMargin = margin;
            topMargin = margin;
            rightMargin= margin;
            bottomMargin = margin;
        } else {
            int horizontalMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginHorizontal, -1);
            int verticalMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginVertical, -1);

            if (horizontalMargin >= 0) {
                leftMargin = horizontalMargin;
                rightMargin = horizontalMargin;
            } else {
                leftMargin = a.getDimensionPixelSize(
                        R.styleable.ViewGroup_MarginLayout_layout_marginLeft,
                        UNDEFINED_MARGIN);
                if (leftMargin == UNDEFINED_MARGIN) {
                    mMarginFlags |= LEFT_MARGIN_UNDEFINED_MASK;
                    leftMargin = DEFAULT_MARGIN_RESOLVED;
                }
                rightMargin = a.getDimensionPixelSize(
                        R.styleable.ViewGroup_MarginLayout_layout_marginRight,
                        UNDEFINED_MARGIN);
                if (rightMargin == UNDEFINED_MARGIN) {
                    mMarginFlags |= RIGHT_MARGIN_UNDEFINED_MASK;
                    rightMargin = DEFAULT_MARGIN_RESOLVED;
                }
            }

            startMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginStart,
                    DEFAULT_MARGIN_RELATIVE);
            endMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginEnd,
```

```
                DEFAULT_MARGIN_RELATIVE);

        if (verticalMargin >= 0) {
            topMargin = verticalMargin;
            bottomMargin = verticalMargin;
        } else {
            topMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginTop,
                    DEFAULT_MARGIN_RESOLVED);
            bottomMargin = a.getDimensionPixelSize(
                    R.styleable.ViewGroup_MarginLayout_layout_marginBottom,
                    DEFAULT_MARGIN_RESOLVED);
        }

        if (isMarginRelative()) {
           mMarginFlags |= NEED_RESOLUTION_MASK;
        }
    }

    final boolean hasRtlSupport = c.getApplicationInfo().hasRtlSupport();
    final int targetSdkVersion = c.getApplicationInfo().targetSdkVersion;
    if (targetSdkVersion < JELLY_BEAN_MR1 || !hasRtlSupport) {
        mMarginFlags |= RTL_COMPATIBILITY_MODE_MASK;
    }

    // Layout direction is LTR by default
    mMarginFlags |= LAYOUT_DIRECTION_LTR;

    a.recycle();
}

public MarginLayoutParams(int width, int height) {
    super(width, height);

    mMarginFlags |= LEFT_MARGIN_UNDEFINED_MASK;
    mMarginFlags |= RIGHT_MARGIN_UNDEFINED_MASK;

    mMarginFlags &= ~NEED_RESOLUTION_MASK;
    mMarginFlags &= ~RTL_COMPATIBILITY_MODE_MASK;
}

/**
 * Copy constructor. Clones the width, height and margin values of the source.
 *
 * @param source The layout params to copy from.
 */
public MarginLayoutParams(MarginLayoutParams source) {
    this.width = source.width;
    this.height = source.height;

    this.leftMargin = source.leftMargin;
    this.topMargin = source.topMargin;
    this.rightMargin = source.rightMargin;
    this.bottomMargin = source.bottomMargin;
    this.startMargin = source.startMargin;
    this.endMargin = source.endMargin;

    this.mMarginFlags = source.mMarginFlags;
}

public MarginLayoutParams(LayoutParams source) {
    super(source);

    mMarginFlags |= LEFT_MARGIN_UNDEFINED_MASK;
    mMarginFlags |= RIGHT_MARGIN_UNDEFINED_MASK;

    mMarginFlags &= ~NEED_RESOLUTION_MASK;
    mMarginFlags &= ~RTL_COMPATIBILITY_MODE_MASK;
}

/**
 * @hide Used internally.
 */
public final void copyMarginsFrom(MarginLayoutParams source) {
    this.leftMargin = source.leftMargin;
    this.topMargin = source.topMargin;
    this.rightMargin = source.rightMargin;
    this.bottomMargin = source.bottomMargin;
    this.startMargin = source.startMargin;
    this.endMargin = source.endMargin;

    this.mMarginFlags = source.mMarginFlags;
```

```
}

/**
 * Sets the margins, in pixels. A call to {@link android.view.View#requestLayout()} needs
 * to be done so that the new margins are taken into account. Left and right margins may be
 * overriden by {@link android.view.View#requestLayout()} depending on layout direction.
 * Margin values should be positive.
 *
 * @param left the left margin size
 * @param top the top margin size
 * @param right the right margin size
 * @param bottom the bottom margin size
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginLeft
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginTop
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginRight
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginBottom
 */
public void setMargins(int left, int top, int right, int bottom) {
    leftMargin = left;
    topMargin = top;
    rightMargin = right;
    bottomMargin = bottom;
    mMarginFlags &= ~LEFT_MARGIN_UNDEFINED_MASK;
    mMarginFlags &= ~RIGHT_MARGIN_UNDEFINED_MASK;
    if (isMarginRelative()) {
        mMarginFlags |= NEED_RESOLUTION_MASK;
    } else {
        mMarginFlags &= ~NEED_RESOLUTION_MASK;
    }
}

/**
 * Sets the relative margins, in pixels. A call to {@link android.view.View#requestLayout()}
 * needs to be done so that the new relative margins are taken into account. Left and right
 * margins may be overriden by {@link android.view.View#requestLayout()} depending on layout
 * direction. Margin values should be positive.
 *
 * @param start the start margin size
 * @param top the top margin size
 * @param end the right margin size
 * @param bottom the bottom margin size
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginStart
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginTop
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginEnd
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginBottom
 *
 * @hide
 */
public void setMarginsRelative(int start, int top, int end, int bottom) {
    startMargin = start;
    topMargin = top;
    endMargin = end;
    bottomMargin = bottom;
    mMarginFlags |= NEED_RESOLUTION_MASK;
}

/**
 * Sets the relative start margin. Margin values should be positive.
 *
 * @param start the start margin size
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginStart
 */
public void setMarginStart(int start) {
    startMargin = start;
    mMarginFlags |= NEED_RESOLUTION_MASK;
}

/**
 * Returns the start margin in pixels.
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginStart
 *
 * @return the start margin in pixels.
 */
public int getMarginStart() {
    if (startMargin != DEFAULT_MARGIN_RELATIVE) return startMargin;
    if ((mMarginFlags & NEED_RESOLUTION_MASK) == NEED_RESOLUTION_MASK) {
        doResolveMargins();
    }
```

```
    switch(mMarginFlags & LAYOUT_DIRECTION_MASK) {
        case View.LAYOUT_DIRECTION_RTL:
            return rightMargin;
        case View.LAYOUT_DIRECTION_LTR:
        default:
            return leftMargin;
    }
}

/**
 * Sets the relative end margin. Margin values should be positive.
 *
 * @param end the end margin size
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginEnd
 */
public void setMarginEnd(int end) {
    endMargin = end;
    mMarginFlags |= NEED_RESOLUTION_MASK;
}

/**
 * Returns the end margin in pixels.
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginEnd
 *
 * @return the end margin in pixels.
 */
public int getMarginEnd() {
    if (endMargin != DEFAULT_MARGIN_RELATIVE) return endMargin;
    if ((mMarginFlags & NEED_RESOLUTION_MASK) == NEED_RESOLUTION_MASK) {
        doResolveMargins();
    }
    switch(mMarginFlags & LAYOUT_DIRECTION_MASK) {
        case View.LAYOUT_DIRECTION_RTL:
            return leftMargin;
        case View.LAYOUT_DIRECTION_LTR:
        default:
            return rightMargin;
    }
}

/**
 * Check if margins are relative.
 *
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginStart
 * @attr ref android.R.styleable#ViewGroup_MarginLayout_layout_marginEnd
 *
 * @return true if either marginStart or marginEnd has been set.
 */
public boolean isMarginRelative() {
    return (startMargin != DEFAULT_MARGIN_RELATIVE || endMargin != DEFAULT_MARGIN_RELATIVE);
}

/**
 * Set the layout direction
 * @param layoutDirection the layout direction.
 *        Should be either {@link View#LAYOUT_DIRECTION_LTR}
 *                      or {@link View#LAYOUT_DIRECTION_RTL}.
 */
public void setLayoutDirection(int layoutDirection) {
    if (layoutDirection != View.LAYOUT_DIRECTION_LTR &&
            layoutDirection != View.LAYOUT_DIRECTION_RTL) return;
    if (layoutDirection != (mMarginFlags & LAYOUT_DIRECTION_MASK)) {
        mMarginFlags &= ~LAYOUT_DIRECTION_MASK;
        mMarginFlags |= (layoutDirection & LAYOUT_DIRECTION_MASK);
        if (isMarginRelative()) {
            mMarginFlags |= NEED_RESOLUTION_MASK;
        } else {
            mMarginFlags &= ~NEED_RESOLUTION_MASK;
        }
    }
}

/**
 * Retuns the layout direction. Can be either {@link View#LAYOUT_DIRECTION_LTR} or
 * {@link View#LAYOUT_DIRECTION_RTL}.
 *
 * @return the layout direction.
 */
public int getLayoutDirection() {
    return (mMarginFlags & LAYOUT_DIRECTION_MASK);
```

```
}

/**
 * This will be called by {@link android.view.View#requestLayout()}. Left and Right margins
 * may be overridden depending on layout direction.
 */
@Override
public void resolveLayoutDirection(int layoutDirection) {
    setLayoutDirection(layoutDirection);

    // No relative margin or pre JB-MR1 case or no need to resolve, just dont do anything
    // Will use the left and right margins if no relative margin is defined.
    if (!isMarginRelative() ||
            (mMarginFlags & NEED_RESOLUTION_MASK) != NEED_RESOLUTION_MASK) return;

    // Proceed with resolution
    doResolveMargins();
}

private void doResolveMargins() {
    if ((mMarginFlags & RTL_COMPATIBILITY_MODE_MASK) == RTL_COMPATIBILITY_MODE_MASK) {
        // if left or right margins are not defined and if we have some start or end margin
        // defined then use those start and end margins.
        if ((mMarginFlags & LEFT_MARGIN_UNDEFINED_MASK) == LEFT_MARGIN_UNDEFINED_MASK
                && startMargin > DEFAULT_MARGIN_RELATIVE) {
            leftMargin = startMargin;
        }
        if ((mMarginFlags & RIGHT_MARGIN_UNDEFINED_MASK) == RIGHT_MARGIN_UNDEFINED_MASK
                && endMargin > DEFAULT_MARGIN_RELATIVE) {
            rightMargin = endMargin;
        }
    } else {
        // We have some relative margins (either the start one or the end one or both). So use
        // them and override what has been defined for left and right margins. If either start
        // or end margin is not defined, just set it to default "0".
        switch(mMarginFlags & LAYOUT_DIRECTION_MASK) {
            case View.LAYOUT_DIRECTION_RTL:
                leftMargin = (endMargin > DEFAULT_MARGIN_RELATIVE) ?
                        endMargin : DEFAULT_MARGIN_RESOLVED;
                rightMargin = (startMargin > DEFAULT_MARGIN_RELATIVE) ?
                        startMargin : DEFAULT_MARGIN_RESOLVED;
                break;
            case View.LAYOUT_DIRECTION_LTR:
            default:
                leftMargin = (startMargin > DEFAULT_MARGIN_RELATIVE) ?
                        startMargin : DEFAULT_MARGIN_RESOLVED;
                rightMargin = (endMargin > DEFAULT_MARGIN_RELATIVE) ?
                        endMargin : DEFAULT_MARGIN_RESOLVED;
                break;
        }
    }
    mMarginFlags &= ~NEED_RESOLUTION_MASK;
}

/**
 * @hide
 */
public boolean isLayoutRtl() {
    return ((mMarginFlags & LAYOUT_DIRECTION_MASK) == View.LAYOUT_DIRECTION_RTL);
}

/**
 * @hide
 */
@Override
public void onDebugDraw(View view, Canvas canvas, Paint paint) {
    Insets oi = isLayoutModeOptical(view.mParent) ? view.getOpticalInsets() : Insets.NONE;

    fillDifference(canvas,
            view.getLeft()   + oi.left,
            view.getTop()    + oi.top,
            view.getRight()  - oi.right,
            view.getBottom() - oi.bottom,
            leftMargin,
            topMargin,
            rightMargin,
            bottomMargin,
            paint);
}

/** @hide */
@Override
```

```
    protected void encodeProperties(@NonNull ViewHierarchyEncoder encoder) {
        super.encodeProperties(encoder);
        encoder.addProperty("leftMargin", leftMargin);
        encoder.addProperty("topMargin", topMargin);
        encoder.addProperty("rightMargin", rightMargin);
        encoder.addProperty("bottomMargin", bottomMargin);
        encoder.addProperty("startMargin", startMargin);
        encoder.addProperty("endMargin", endMargin);
    }
}

/* Describes a touched view and the ids of the pointers that it has captured.
 *
 * This code assumes that pointer ids are always in the range 0..31 such that
 * it can use a bitfield to track which pointer ids are present.
 * As it happens, the lower layers of the input dispatch pipeline also use the
 * same trick so the assumption should be safe here...
 */
private static final class TouchTarget {
    private static final int MAX_RECYCLED = 32;
    private static final Object sRecycleLock = new Object[0];
    private static TouchTarget sRecycleBin;
    private static int sRecycledCount;

    public static final int ALL_POINTER_IDS = -1; // all ones

    // The touched child view.
    public View child;

    // The combined bit mask of pointer ids for all pointers captured by the target.
    public int pointerIdBits;

    // The next target in the target list.
    public TouchTarget next;

    private TouchTarget() {
    }

    public static TouchTarget obtain(@NonNull View child, int pointerIdBits) {
        if (child == null) {
            throw new IllegalArgumentException("child must be non-null");
        }

        final TouchTarget target;
        synchronized (sRecycleLock) {
            if (sRecycleBin == null) {
                target = new TouchTarget();
            } else {
                target = sRecycleBin;
                sRecycleBin = target.next;
                 sRecycledCount--;
                target.next = null;
            }
        }
        target.child = child;
        target.pointerIdBits = pointerIdBits;
        return target;
    }

    public void recycle() {
        if (child == null) {
            throw new IllegalStateException("already recycled once");
        }

        synchronized (sRecycleLock) {
            if (sRecycledCount < MAX_RECYCLED) {
                next = sRecycleBin;
                sRecycleBin = this;
                sRecycledCount += 1;
            } else {
                next = null;
            }
            child = null;
        }
    }
}

/* Describes a hovered view. */
private static final class HoverTarget {
    private static final int MAX_RECYCLED = 32;
    private static final Object sRecycleLock = new Object[0];
    private static HoverTarget sRecycleBin;
```

```
    private static int sRecycledCount;

    // The hovered child view.
    public View child;

    // The next target in the target list.
    public HoverTarget next;

    private HoverTarget() {
    }

    public static HoverTarget obtain(@NonNull View child) {
        if (child == null) {
            throw new IllegalArgumentException("child must be non-null");
        }

        final HoverTarget target;
        synchronized (sRecycleLock) {
            if (sRecycleBin == null) {
                target = new HoverTarget();
            } else {
                target = sRecycleBin;
                sRecycleBin = target.next;
                sRecycledCount--;
                target.next = null;
            }
        }
        target.child = child;
        return target;
    }

    public void recycle() {
        if (child == null) {
            throw new IllegalStateException("already recycled once");
        }

        synchronized (sRecycleLock) {
            if (sRecycledCount < MAX_RECYCLED) {
                next = sRecycleBin;
                sRecycleBin = this;
                sRecycledCount += 1;
            } else {
                next = null;
            }
            child = null;
        }
    }
}

/**
 * Pooled class that to hold the children for autifill.
 */
static class ChildListForAutoFill extends ArrayList<View> {
    private static final int MAX_POOL_SIZE = 32;

    private static final Pools.SimplePool<ChildListForAutoFill> sPool =
            new Pools.SimplePool<>(MAX_POOL_SIZE);

    public static ChildListForAutoFill obtain() {
        ChildListForAutoFill list = sPool.acquire();
        if (list == null) {
            list = new ChildListForAutoFill();
        }
        return list;
    }

    public void recycle() {
        clear();
        sPool.release(this);
    }
}

/**
 * Pooled class that orderes the children of a ViewGroup from start
 * to end based on how they are laid out and the layout direction.
 */
static class ChildListForAccessibility {

    private static final int MAX_POOL_SIZE = 32;

    private static final SynchronizedPool<ChildListForAccessibility> sPool =
            new SynchronizedPool<ChildListForAccessibility>(MAX_POOL_SIZE);
```

```
    private final ArrayList<View> mChildren = new ArrayList<View>();

    private final ArrayList<ViewLocationHolder> mHolders = new ArrayList<ViewLocationHolder>();

    public static ChildListForAccessibility obtain(ViewGroup parent, boolean sort) {
        ChildListForAccessibility list = sPool.acquire();
        if (list == null) {
            list = new ChildListForAccessibility();
        }
        list.init(parent, sort);
        return list;
    }

    public void recycle() {
        clear();
        sPool.release(this);
    }

    public int getChildCount() {
        return mChildren.size();
    }

    public View getChildAt(int index) {
        return mChildren.get(index);
    }

    private void init(ViewGroup parent, boolean sort) {
        ArrayList<View> children = mChildren;
        final int childCount = parent.getChildCount();
        for (int i = 0; i < childCount; i++) {
            View child = parent.getChildAt(i);
            children.add(child);
        }
        if (sort) {
            ArrayList<ViewLocationHolder> holders = mHolders;
            for (int i = 0; i < childCount; i++) {
                View child = children.get(i);
                ViewLocationHolder holder = ViewLocationHolder.obtain(parent, child);
                holders.add(holder);
            }
            sort(holders);
            for (int i = 0; i < childCount; i++) {
                ViewLocationHolder holder = holders.get(i);
                children.set(i, holder.mView);
                holder.recycle();
            }
            holders.clear();
        }
    }

    private void sort(ArrayList<ViewLocationHolder> holders) {
        // This is gross but the least risky solution. The current comparison
        // strategy breaks transitivity but produces very good results. Coming
        // up with a new strategy requires time which we do not have, so ...
        try {
            ViewLocationHolder.setComparisonStrategy(
                    ViewLocationHolder.COMPARISON_STRATEGY_STRIPE);
            Collections.sort(holders);
        } catch (IllegalArgumentException iae) {
            // Note that in practice this occurs extremely rarely in a couple
            // of pathological cases.
            ViewLocationHolder.setComparisonStrategy(
                    ViewLocationHolder.COMPARISON_STRATEGY_LOCATION);
            Collections.sort(holders);
        }
    }

    private void clear() {
        mChildren.clear();
    }
}

/**
 * Pooled class that holds a View and its location with respect to
 * a specified root. This enables sorting of views based on their
 * coordinates without recomputing the position relative to the root
 * on every comparison.
 */
static class ViewLocationHolder implements Comparable<ViewLocationHolder> {

    private static final int MAX_POOL_SIZE = 32;
```

```
private static final SynchronizedPool<ViewLocationHolder> sPool =
        new SynchronizedPool<ViewLocationHolder>(MAX_POOL_SIZE);

public static final int COMPARISON_STRATEGY_STRIPE = 1;

public static final int COMPARISON_STRATEGY_LOCATION = 2;

private static int sComparisonStrategy = COMPARISON_STRATEGY_STRIPE;

private final Rect mLocation = new Rect();

public View mView;

private int mLayoutDirection;

public static ViewLocationHolder obtain(ViewGroup root, View view) {
    ViewLocationHolder holder = sPool.acquire();
    if (holder == null) {
        holder = new ViewLocationHolder();
    }
    holder.init(root, view);
    return holder;
}

public static void setComparisonStrategy(int strategy) {
    sComparisonStrategy = strategy;
}

public void recycle() {
    clear();
    sPool.release(this);
}

@Override
public int compareTo(ViewLocationHolder another) {
    // This instance is greater than an invalid argument.
    if (another == null) {
        return 1;
    }

    if (sComparisonStrategy == COMPARISON_STRATEGY_STRIPE) {
        // First is above second.
        if (mLocation.bottom - another.mLocation.top <= 0) {
            return -1;
        }
        // First is below second.
        if (mLocation.top - another.mLocation.bottom >= 0) {
            return 1;
        }
    }

    // We are ordering left-to-right, top-to-bottom.
    if (mLayoutDirection == LAYOUT_DIRECTION_LTR) {
        final int leftDifference = mLocation.left - another.mLocation.left;
        if (leftDifference != 0) {
            return leftDifference;
        }
    } else { // RTL
        final int rightDifference = mLocation.right - another.mLocation.right;
        if (rightDifference != 0) {
            return -rightDifference;
        }
    }
    // We are ordering left-to-right, top-to-bottom.
    final int topDifference = mLocation.top - another.mLocation.top;
    if (topDifference != 0) {
        return topDifference;
    }
    // Break tie by height.
    final int heightDiference = mLocation.height() - another.mLocation.height();
    if (heightDiference != 0) {
        return -heightDiference;
    }
    // Break tie by width.
    final int widthDiference = mLocation.width() - another.mLocation.width();
    if (widthDiference != 0) {
        return -widthDiference;
    }
    // Just break the tie somehow. The accessibliity ids are unique
    // and stable, hence this is deterministic tie breaking.
    return mView.getAccessibilityViewId() - another.mView.getAccessibilityViewId();
```

```
        }

        private void init(ViewGroup root, View view) {
            Rect viewLocation = mLocation;
            view.getDrawingRect(viewLocation);
            root.offsetDescendantRectToMyCoords(view, viewLocation);
            mView = view;
            mLayoutDirection = root.getLayoutDirection();
        }

        private void clear() {
            mView = null;
            mLocation.set(0, 0, 0, 0);
        }
    }

    private static void drawRect(Canvas canvas, Paint paint, int x1, int y1, int x2, int y2) {
        if (sDebugLines== null) {
            // TODO: This won't work with multiple UI threads in a single process
            sDebugLines = new float[16];
        }

        sDebugLines[0] = x1;
        sDebugLines[1] = y1;
        sDebugLines[2] = x2;
        sDebugLines[3] = y1;

        sDebugLines[4] = x2;
        sDebugLines[5] = y1;
        sDebugLines[6] = x2;
        sDebugLines[7] = y2;

        sDebugLines[8] = x2;
        sDebugLines[9] = y2;
        sDebugLines[10] = x1;
        sDebugLines[11] = y2;

        sDebugLines[12] = x1;
        sDebugLines[13] = y2;
        sDebugLines[14] = x1;
        sDebugLines[15] = y1;

        canvas.drawLines(sDebugLines, paint);
    }

    /** @hide */
    @Override
    protected void encodeProperties(@NonNull ViewHierarchyEncoder encoder) {
        super.encodeProperties(encoder);

        encoder.addProperty("focus:descendantFocusability", getDescendantFocusability());
        encoder.addProperty("drawing:clipChildren", getClipChildren());
        encoder.addProperty("drawing:clipToPadding", getClipToPadding());
        encoder.addProperty("drawing:childrenDrawingOrderEnabled", isChildrenDrawingOrderEnabled());
        encoder.addProperty("drawing:persistentDrawingCache", getPersistentDrawingCache());

        int n = getChildCount();
        encoder.addProperty("meta:__childCount__", (short)n);
        for (int i = 0; i < n; i++) {
            encoder.addPropertyKey("meta:__child__" + i);
            getChildAt(i).encode(encoder);
        }
    }
}
```