```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.widget;

import android.annotation.IntDef;
import android.annotation.NonNull;
import android.annotation.Nullable;
import android.content.Context;
import android.content.res.TypedArray;
import android.graphics.Canvas;
import android.graphics.drawable.Drawable;
import android.os.Build;
import android.util.AttributeSet;
import android.view.Gravity;
import android.view.View;
import android.view.ViewDebug;
import android.view.ViewGroup;
import android.view.ViewHierarchyEncoder;
import android.widget.RemoteViews.RemoteView;

import com.android.internal.R;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;


/**
 * A layout that arranges other views either horizontally in a single column
 * or vertically in a single row.
 *
 * <p>The following snippet shows how to include a linear layout in your layout XML file:</p>
 *
 * <pre>&lt;LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 *   android:layout_width="match_parent"
 *   android:layout_height="match_parent"
 *   android:paddingLeft="16dp"
 *   android:paddingRight="16dp"
 *   android:orientation="horizontal"
 *   android:gravity="center"&gt;
 *
 *   &lt;!-- Include other widget or layout tags here. These are considered
 *           "child views" or "children" of the linear layout --&gt;
 *
 * &lt;/LinearLayout&gt;</pre>
 *
 * <p>Set {@link android.R.styleable#LinearLayout_orientation android:orientation} to specify
 * whether child views are displayed in a row or column.</p>
 *
 * <p>To control how linear layout aligns all the views it contains, set a value for
 * {@link android.R.styleable#LinearLayout_gravity android:gravity}.  For example, the
 * snippet above sets android:gravity to "center".  The value you set affects
 * both horizontal and vertical alignment of all child views within the single row or column.</p>
 *
 * <p>You can set
 * {@link android.R.styleable#LinearLayout_Layout_layout_weight android:layout_weight}
 * on individual child views to specify how linear layout divides remaining space amongst
 * the views it contains. See the
 * <a href="https://developer.android.com/guide/topics/ui/layout/linear.html">Linear Layout</a>
 * guide for an example.</p>
 *
 * <p>See
 * {@link android.widget.LinearLayout.LayoutParams LinearLayout.LayoutParams}
 * to learn about other attributes you can set on a child view to affect its
 * position and size in the containing linear layout.</p>
 *
 * @attr ref android.R.styleable#LinearLayout_baselineAligned
 * @attr ref android.R.styleable#LinearLayout_baselineAlignedChildIndex
 * @attr ref android.R.styleable#LinearLayout_gravity
 * @attr ref android.R.styleable#LinearLayout_measureWithLargestChild
 * @attr ref android.R.styleable#LinearLayout_orientation
 * @attr ref android.R.styleable#LinearLayout_weightSum
 */
@RemoteView
public class LinearLayout extends ViewGroup {
    /** @hide */
    @IntDef({HORIZONTAL, VERTICAL})
    @Retention(RetentionPolicy.SOURCE)
    public @interface OrientationMode {}

    public static final int HORIZONTAL = 0;
    public static final int VERTICAL = 1;

    /** @hide */
    @IntDef(flag = true,
            value = {
                SHOW_DIVIDER_NONE,
                SHOW_DIVIDER_BEGINNING,
                SHOW_DIVIDER_MIDDLE,
                SHOW_DIVIDER_END
            })
    @Retention(RetentionPolicy.SOURCE)
    public @interface DividerMode {}

    /**
     * Don't show any dividers.
     */
    public static final int SHOW_DIVIDER_NONE = 0;
    /**
     * Show a divider at the beginning of the group.
     */
    public static final int SHOW_DIVIDER_BEGINNING = 1;
    /**
     * Show dividers between each item in the group.
     */
    public static final int SHOW_DIVIDER_MIDDLE = 2;
    /**
     * Show a divider at the end of the group.
     */
    public static final int SHOW_DIVIDER_END = 4;

    /**
     * Compatibility check. Old versions of the platform would give different
     * results from measurement passes using EXACTLY and non-EXACTLY modes,
     * even when the resulting size was the same.
     */
    private final boolean mAllowInconsistentMeasurement;

    /**
     * Whether the children of this layout are baseline aligned.  Only applicable
     * if {@link #mOrientation} is horizontal.
     */
    @ViewDebug.ExportedProperty(category = "layout")
    private boolean mBaselineAligned = true;

    /**
     * If this layout is part of another layout that is baseline aligned,
     * use the child at this index as the baseline.
     *
```

```java
 * Note: this is orthogonal to {@link #mBaselineAligned}, which is concerned
 * with whether the children of this layout are baseline aligned.
 */
@ViewDebug.ExportedProperty(category = "layout")
private int mBaselineAlignedChildIndex = -1;

/**
 * The additional offset to the child's baseline.
 * We'll calculate the baseline of this layout as we measure vertically; for
 * horizontal linear layouts, the offset of 0 is appropriate.
 */
@ViewDebug.ExportedProperty(category = "measurement")
private int mBaselineChildTop = 0;

@ViewDebug.ExportedProperty(category = "measurement")
private int mOrientation;

@ViewDebug.ExportedProperty(category = "measurement", flagMapping = {
        @ViewDebug.FlagToString(mask = -1,
            equals = -1, name = "NONE"),
        @ViewDebug.FlagToString(mask = Gravity.NO_GRAVITY,
            equals = Gravity.NO_GRAVITY,name = "NONE"),
        @ViewDebug.FlagToString(mask = Gravity.TOP,
            equals = Gravity.TOP, name = "TOP"),
        @ViewDebug.FlagToString(mask = Gravity.BOTTOM,
            equals = Gravity.BOTTOM, name = "BOTTOM"),
        @ViewDebug.FlagToString(mask = Gravity.LEFT,
            equals = Gravity.LEFT, name = "LEFT"),
        @ViewDebug.FlagToString(mask = Gravity.RIGHT,
            equals = Gravity.RIGHT, name = "RIGHT"),
        @ViewDebug.FlagToString(mask = Gravity.START,
            equals = Gravity.START, name = "START"),
        @ViewDebug.FlagToString(mask = Gravity.END,
            equals = Gravity.END, name = "END"),
        @ViewDebug.FlagToString(mask = Gravity.CENTER_VERTICAL,
            equals = Gravity.CENTER_VERTICAL, name = "CENTER_VERTICAL"),
        @ViewDebug.FlagToString(mask = Gravity.FILL_VERTICAL,
            equals = Gravity.FILL_VERTICAL, name = "FILL_VERTICAL"),
        @ViewDebug.FlagToString(mask = Gravity.CENTER_HORIZONTAL,
            equals = Gravity.CENTER_HORIZONTAL, name = "CENTER_HORIZONTAL"),
        @ViewDebug.FlagToString(mask = Gravity.FILL_HORIZONTAL,
            equals = Gravity.FILL_HORIZONTAL, name = "FILL_HORIZONTAL"),
        @ViewDebug.FlagToString(mask = Gravity.CENTER,
            equals = Gravity.CENTER, name = "CENTER"),
        @ViewDebug.FlagToString(mask = Gravity.FILL,
            equals = Gravity.FILL, name = "FILL"),
        @ViewDebug.FlagToString(mask = Gravity.RELATIVE_LAYOUT_DIRECTION,
            equals = Gravity.RELATIVE_LAYOUT_DIRECTION, name = "RELATIVE")
    }, formatToHexString = true)
private int mGravity = Gravity.START | Gravity.TOP;

@ViewDebug.ExportedProperty(category = "measurement")
private int mTotalLength;

@ViewDebug.ExportedProperty(category = "layout")
private float mWeightSum;

@ViewDebug.ExportedProperty(category = "layout")
private boolean mUseLargestChild;

private int[] mMaxAscent;
private int[] mMaxDescent;

private static final int VERTICAL_GRAVITY_COUNT = 4;

private static final int INDEX_CENTER_VERTICAL = 0;
private static final int INDEX_TOP = 1;
private static final int INDEX_BOTTOM = 2;
private static final int INDEX_FILL = 3;

private Drawable mDivider;
private int mDividerWidth;
private int mDividerHeight;
private int mShowDividers;
private int mDividerPadding;

private int mLayoutDirection = View.LAYOUT_DIRECTION_UNDEFINED;

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

public LinearLayout(Context context, @Nullable AttributeSet attrs) {
    this(context, attrs, 0);
}

public LinearLayout(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
    this(context, attrs, defStyleAttr, 0);
}

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

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

    int index = a.getInt(com.android.internal.R.styleable.LinearLayout_orientation, -1);
    if (index >= 0) {
        setOrientation(index);
    }

    index = a.getInt(com.android.internal.R.styleable.LinearLayout_gravity, -1);
    if (index >= 0) {
        setGravity(index);
    }

    boolean baselineAligned = a.getBoolean(R.styleable.LinearLayout_baselineAligned, true);
    if (!baselineAligned) {
        setBaselineAligned(baselineAligned);
    }

    mWeightSum = a.getFloat(R.styleable.LinearLayout_weightSum, -1.0f);

    mBaselineAlignedChildIndex =
            a.getInt(com.android.internal.R.styleable.LinearLayout_baselineAlignedChildIndex, -1);

    mUseLargestChild = a.getBoolean(R.styleable.LinearLayout_measureWithLargestChild, false);

    mShowDividers = a.getInt(R.styleable.LinearLayout_showDividers, SHOW_DIVIDER_NONE);
    mDividerPadding = a.getDimensionPixelSize(R.styleable.LinearLayout_dividerPadding, 0);
    setDividerDrawable(a.getDrawable(R.styleable.LinearLayout_divider));

    final int version = context.getApplicationInfo().targetSdkVersion;
    mAllowInconsistentMeasurement = version <= Build.VERSION_CODES.M;

    a.recycle();
}

/**
 * Returns <code>true</code> if this layout is currently configured to show at least one
 * divider.
 */
private boolean isShowingDividers() {
    return (mShowDividers != SHOW_DIVIDER_NONE) && (mDivider != null);
}

/**
 * Set how dividers should be shown between items in this layout
 *
 * @param showDividers One or more of {@link #SHOW_DIVIDER_BEGINNING},
 *                     {@link #SHOW_DIVIDER_MIDDLE}, or {@link #SHOW_DIVIDER_END}
 *                     to show dividers, or {@link #SHOW_DIVIDER_NONE} to show no dividers.
 */
public void setShowDividers(@DividerMode int showDividers) {
```

```java
        if (showDividers == mShowDividers) {
            return;
        }
        mShowDividers = showDividers;

        setWillNotDraw(!isShowingDividers());
        requestLayout();
    }

    @Override
    public boolean shouldDelayChildPressedState() {
        return false;
    }

    /**
     * @return A flag set indicating how dividers should be shown around items.
     * @see #setShowDividers(int)
     */
    @DividerMode
    public int getShowDividers() {
        return mShowDividers;
    }

    /**
     * @return the divider Drawable that will divide each item.
     *
     * @see #setDividerDrawable(Drawable)
     *
     * @attr ref android.R.styleable#LinearLayout_divider
     */
    public Drawable getDividerDrawable() {
        return mDivider;
    }

    /**
     * Set a drawable to be used as a divider between items.
     *
     * @param divider Drawable that will divide each item.
     *
     * @see #setShowDividers(int)
     *
     * @attr ref android.R.styleable#LinearLayout_divider
     */
    public void setDividerDrawable(Drawable divider) {
        if (divider == mDivider) {
            return;
        }
        mDivider = divider;
        if (divider != null) {
            mDividerWidth = divider.getIntrinsicWidth();
            mDividerHeight = divider.getIntrinsicHeight();
        } else {
            mDividerWidth = 0;
            mDividerHeight = 0;
        }

        setWillNotDraw(!isShowingDividers());
        requestLayout();
    }

    /**
     * Set padding displayed on both ends of dividers. For a vertical layout, the padding is applied
     * to left and right end of dividers. For a horizontal layout, the padding is applied to top and
     * bottom end of dividers.
     *
     * @param padding Padding value in pixels that will be applied to each end
     *
     * @see #setShowDividers(int)
     * @see #setDividerDrawable(Drawable)
     * @see #getDividerPadding()
     */
    public void setDividerPadding(int padding) {
        if (padding == mDividerPadding) {
            return;
        }
        mDividerPadding = padding;

        if (isShowingDividers()) {
            requestLayout();
            invalidate();
        }
    }

    /**
     * Get the padding size used to inset dividers in pixels
     *
     * @see #setShowDividers(int)
     * @see #setDividerDrawable(Drawable)
     * @see #setDividerPadding(int)
     */
    public int getDividerPadding() {
        return mDividerPadding;
    }

    /**
     * Get the width of the current divider drawable.
     *
     * @hide Used internally by framework.
     */
    public int getDividerWidth() {
        return mDividerWidth;
    }

    @Override
    protected void onDraw(Canvas canvas) {
        if (mDivider == null) {
            return;
        }

        if (mOrientation == VERTICAL) {
            drawDividersVertical(canvas);
        } else {
            drawDividersHorizontal(canvas);
        }
    }

    void drawDividersVertical(Canvas canvas) {
        final int count = getVirtualChildCount();
        for (int i = 0; i < count; i++) {
            final View child = getVirtualChildAt(i);
            if (child != null && child.getVisibility() != GONE) {
                if (hasDividerBeforeChildAt(i)) {
                    final LayoutParams lp = (LayoutParams) child.getLayoutParams();
                    final int top = child.getTop() - lp.topMargin - mDividerHeight;
                    drawHorizontalDivider(canvas, top);
                }
            }
        }

        if (hasDividerBeforeChildAt(count)) {
            final View child = getLastNonGoneChild();
            int bottom = 0;
            if (child == null) {
                bottom = getHeight() - getPaddingBottom() - mDividerHeight;
            } else {
                final LayoutParams lp = (LayoutParams) child.getLayoutParams();
                bottom = child.getBottom() + lp.bottomMargin;
            }
            drawHorizontalDivider(canvas, bottom);
        }
    }

    /**
     * Finds the last child that is not gone. The last child will be used as the reference for
```

```java
 * where the end divider should be drawn.
 */
private View getLastNonGoneChild() {
    for (int i = getVirtualChildCount() - 1; i >= 0; i--) {
        final View child = getVirtualChildAt(i);
        if (child != null && child.getVisibility() != GONE) {
            return child;
        }
    }
    return null;
}

void drawDividersHorizontal(Canvas canvas) {
    final int count = getVirtualChildCount();
    final boolean isLayoutRtl = isLayoutRtl();
    for (int i = 0; i < count; i++) {
        final View child = getVirtualChildAt(i);
        if (child != null && child.getVisibility() != GONE) {
            if (hasDividerBeforeChildAt(i)) {
                final LayoutParams lp = (LayoutParams) child.getLayoutParams();
                final int position;
                if (isLayoutRtl) {
                    position = child.getRight() + lp.rightMargin;
                } else {
                    position = child.getLeft() - lp.leftMargin - mDividerWidth;
                }
                drawVerticalDivider(canvas, position);
            }
        }
    }

    if (hasDividerBeforeChildAt(count)) {
        final View child = getLastNonGoneChild();
        int position;
        if (child == null) {
            if (isLayoutRtl) {
                position = getPaddingLeft();
            } else {
                position = getWidth() - getPaddingRight() - mDividerWidth;
            }
        } else {
            final LayoutParams lp = (LayoutParams) child.getLayoutParams();
            if (isLayoutRtl) {
                position = child.getLeft() - lp.leftMargin - mDividerWidth;
            } else {
                position = child.getRight() + lp.rightMargin;
            }
        }
        drawVerticalDivider(canvas, position);
    }
}

void drawHorizontalDivider(Canvas canvas, int top) {
    mDivider.setBounds(getPaddingLeft() + mDividerPadding, top,
            getWidth() - getPaddingRight() - mDividerPadding, top + mDividerHeight);
    mDivider.draw(canvas);
}

void drawVerticalDivider(Canvas canvas, int left) {
    mDivider.setBounds(left, getPaddingTop() + mDividerPadding,
            left + mDividerWidth, getHeight() - getPaddingBottom() - mDividerPadding);
    mDivider.draw(canvas);
}

/**
 * <p>Indicates whether widgets contained within this layout are aligned
 * on their baseline or not.</p>
 *
 * @return true when widgets are baseline-aligned, false otherwise
 */
public boolean isBaselineAligned() {
    return mBaselineAligned;
```

```java
}

/**
 * <p>Defines whether widgets contained in this layout are
 * baseline-aligned or not.</p>
 *
 * @param baselineAligned true to align widgets on their baseline,
 *         false otherwise
 *
 * @attr ref android.R.styleable#LinearLayout_baselineAligned
 */
@android.view.RemotableViewMethod
public void setBaselineAligned(boolean baselineAligned) {
    mBaselineAligned = baselineAligned;
}

/**
 * When true, all children with a weight will be considered having
 * the minimum size of the largest child. If false, all children are
 * measured normally.
 *
 * @return True to measure children with a weight using the minimum
 *         size of the largest child, false otherwise.
 *
 * @attr ref android.R.styleable#LinearLayout_measureWithLargestChild
 */
public boolean isMeasureWithLargestChildEnabled() {
    return mUseLargestChild;
}

/**
 * When set to true, all children with a weight will be considered having
 * the minimum size of the largest child. If false, all children are
 * measured normally.
 *
 * Disabled by default.
 *
 * @param enabled True to measure children with a weight using the
 *        minimum size of the largest child, false otherwise.
 *
 * @attr ref android.R.styleable#LinearLayout_measureWithLargestChild
 */
@android.view.RemotableViewMethod
public void setMeasureWithLargestChildEnabled(boolean enabled) {
    mUseLargestChild = enabled;
}

@Override
public int getBaseline() {
    if (mBaselineAlignedChildIndex < 0) {
        return super.getBaseline();
    }

    if (getChildCount() <= mBaselineAlignedChildIndex) {
        throw new RuntimeException("mBaselineAlignedChildIndex of LinearLayout "
                + "set to an index that is out of bounds.");
    }

    final View child = getChildAt(mBaselineAlignedChildIndex);
    final int childBaseline = child.getBaseline();

    if (childBaseline == -1) {
        if (mBaselineAlignedChildIndex == 0) {
            // this is just the default case, safe to return -1
            return -1;
        }
        // the user picked an index that points to something that doesn't
        // know how to calculate its baseline.
        throw new RuntimeException("mBaselineAlignedChildIndex of LinearLayout "
                + "points to a View that doesn't know how to get its baseline.");
    }
```

```
        // TODO: This should try to take into account the virtual offsets
        // (See getNextLocationOffset and getLocationOffset)
        // We should add to childTop:
        // sum([getNextLocationOffset(getChildAt(i)) / i < mBaselineAlignedChildIndex])
        // and also add:
        // getLocationOffset(child)
        int childTop = mBaselineChildTop;

        if (mOrientation == VERTICAL) {
            final int majorGravity = mGravity & Gravity.VERTICAL_GRAVITY_MASK;
            if (majorGravity != Gravity.TOP) {
               switch (majorGravity) {
                   case Gravity.BOTTOM:
                       childTop = mBottom - mTop - mPaddingBottom - mTotalLength;
                       break;

                   case Gravity.CENTER_VERTICAL:
                       childTop += ((mBottom - mTop - mPaddingTop - mPaddingBottom) -
                               mTotalLength) / 2;
                       break;
               }
            }
        }

        LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams) child.getLayoutParams();
        return childTop + lp.topMargin + childBaseline;
    }

    /**
     * @return The index of the child that will be used if this layout is
     *   part of a larger layout that is baseline aligned, or -1 if none has
     *   been set.
     */
    public int getBaselineAlignedChildIndex() {
        return mBaselineAlignedChildIndex;
    }

    /**
     * @param i The index of the child that will be used if this layout is
     *          part of a larger layout that is baseline aligned.
     *
     * @attr ref android.R.styleable#LinearLayout_baselineAlignedChildIndex
     */
    @android.view.RemotableViewMethod
    public void setBaselineAlignedChildIndex(int i) {
        if ((i < 0) || (i >= getChildCount())) {
            throw new IllegalArgumentException("base aligned child index out "
                    + "of range (0, " + getChildCount() + ")");
        }
        mBaselineAlignedChildIndex = i;
    }

    /**
     * <p>Returns the view at the specified index. This method can be overriden
     * to take into account virtual children. Refer to
     * {@link android.widget.TableLayout} and {@link android.widget.TableRow}
     * for an example.</p>
     *
     * @param index the child's index
     * @return the child at the specified index, may be {@code null}
     */
    @Nullable
    View getVirtualChildAt(int index) {
        return getChildAt(index);
    }

    /**
     * <p>Returns the virtual number of children. This number might be different
     * than the actual number of children if the layout can hold virtual
     * children. Refer to
     * {@link android.widget.TableLayout} and {@link android.widget.TableRow}
     * for an example.</p>
```

```java
     *
     * @return the virtual number of children
     */
    int getVirtualChildCount() {
        return getChildCount();
    }

    /**
     * Returns the desired weights sum.
     *
     * @return A number greater than 0.0f if the weight sum is defined, or
     *         a number lower than or equals to 0.0f if not weight sum is
     *         to be used.
     */
    public float getWeightSum() {
        return mWeightSum;
    }

    /**
     * Defines the desired weights sum. If unspecified the weights sum is computed
     * at layout time by adding the layout_weight of each child.
     *
     * This can be used for instance to give a single child 50% of the total
     * available space by giving it a layout_weight of 0.5 and setting the
     * weightSum to 1.0.
     *
     * @param weightSum a number greater than 0.0f, or a number lower than or equals
     *        to 0.0f if the weight sum should be computed from the children's
     *        layout_weight
     */
    @android.view.RemotableViewMethod
    public void setWeightSum(float weightSum) {
        mWeightSum = Math.max(0.0f, weightSum);
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        if (mOrientation == VERTICAL) {
            measureVertical(widthMeasureSpec, heightMeasureSpec);
        } else {
            measureHorizontal(widthMeasureSpec, heightMeasureSpec);
        }
    }

    /**
     * Determines where to position dividers between children.
     *
     * @param childIndex Index of child to check for preceding divider
     * @return true if there should be a divider before the child at childIndex
     * @hide Pending API consideration. Currently only used internally by the system.
     */
    protected boolean hasDividerBeforeChildAt(int childIndex) {
        if (childIndex == getVirtualChildCount()) {
            // Check whether the end divider should draw.
            return (mShowDividers & SHOW_DIVIDER_END) != 0;
        }
        boolean allViewsAreGoneBefore = allViewsAreGoneBefore(childIndex);
        if (allViewsAreGoneBefore) {
            // This is the first view that's not gone, check if beginning divider is enabled.
            return (mShowDividers & SHOW_DIVIDER_BEGINNING) != 0;
        } else {
            return (mShowDividers & SHOW_DIVIDER_MIDDLE) != 0;
        }
    }

    /**
     * Checks whether all (virtual) child views before the given index are gone.
     */
    private boolean allViewsAreGoneBefore(int childIndex) {
        for (int i = childIndex - 1; i >= 0; i--) {
            final View child = getVirtualChildAt(i);
            if (child != null && child.getVisibility() != GONE) {
                return false;
            }
        }
        return true;
    }

    /**
     * Measures the children when the orientation of this LinearLayout is set
     * to {@link #VERTICAL}.
     *
     * @param widthMeasureSpec Horizontal space requirements as imposed by the parent.
     * @param heightMeasureSpec Vertical space requirements as imposed by the parent.
     *
     * @see #getOrientation()
     * @see #setOrientation(int)
     * @see #onMeasure(int, int)
     */
    void measureVertical(int widthMeasureSpec, int heightMeasureSpec) {
        mTotalLength = 0;
        int maxWidth = 0;
        int childState = 0;
        int alternativeMaxWidth = 0;
        int weightedMaxWidth = 0;
        boolean allFillParent = true;
        float totalWeight = 0;

        final int count = getVirtualChildCount();

        final int widthMode = MeasureSpec.getMode(widthMeasureSpec);
        final int heightMode = MeasureSpec.getMode(heightMeasureSpec);

        boolean matchWidth = false;
        boolean skippedMeasure = false;

        final int baselineChildIndex = mBaselineAlignedChildIndex;
        final boolean useLargestChild = mUseLargestChild;

        int largestChildHeight = Integer.MIN_VALUE;
        int consumedExcessSpace = 0;

        int nonSkippedChildCount = 0;

        // See how tall everyone is. Also remember max width.
        for (int i = 0; i < count; ++i) {
            final View child = getVirtualChildAt(i);
            if (child == null) {
                mTotalLength += measureNullChild(i);
                continue;
            }

            if (child.getVisibility() == View.GONE) {
               i += getChildrenSkipCount(child, i);
               continue;
            }

            nonSkippedChildCount++;
            if (hasDividerBeforeChildAt(i)) {
                mTotalLength += mDividerHeight;
            }

            final LayoutParams lp = (LayoutParams) child.getLayoutParams();

            totalWeight += lp.weight;

            final boolean useExcessSpace = lp.height == 0 && lp.weight > 0;
            if (heightMode == MeasureSpec.EXACTLY && useExcessSpace) {
                // Optimization: don't bother measuring children who are only
                // laid out using excess space. These views will get measured
                // later if we have space to distribute.
                final int totalLength = mTotalLength;
                mTotalLength = Math.max(totalLength, totalLength + lp.topMargin + lp.bottomMargin);
                skippedMeasure = true;
```

```
} else {
    if (useExcessSpace) {
        // The heightMode is either UNSPECIFIED or AT_MOST, and
        // this child is only laid out using excess space. Measure
        // using WRAP_CONTENT so that we can find out the view's
        // optimal height. We'll restore the original height of 0
        // after measurement.
        lp.height = LayoutParams.WRAP_CONTENT;
    }

    // Determine how big this child would like to be. If this or
    // previous children have given a weight, then we allow it to
    // use all available space (and we will shrink things later
    // if needed).
    final int usedHeight = totalWeight == 0 ? mTotalLength : 0;
    measureChildBeforeLayout(child, i, widthMeasureSpec, 0,
            heightMeasureSpec, usedHeight);

    final int childHeight = child.getMeasuredHeight();
    if (useExcessSpace) {
        // Restore the original height and record how much space
        // we've allocated to excess-only children so that we can
        // match the behavior of EXACTLY measurement.
        lp.height = 0;
        consumedExcessSpace += childHeight;
    }

    final int totalLength = mTotalLength;
    mTotalLength = Math.max(totalLength, totalLength + childHeight + lp.topMargin +
           lp.bottomMargin + getNextLocationOffset(child));

    if (useLargestChild) {
        largestChildHeight = Math.max(childHeight, largestChildHeight);
    }
}

/**
 * If applicable, compute the additional offset to the child's baseline
 * we'll need later when asked {@link #getBaseline}.
 */
if ((baselineChildIndex >= 0) && (baselineChildIndex == i + 1)) {
   mBaselineChildTop = mTotalLength;
}

// if we are trying to use a child index for our baseline, the above
// book keeping only works if there are no children above it with
// weight.  fail fast to aid the developer.
if (i < baselineChildIndex && lp.weight > 0) {
    throw new RuntimeException("A child of LinearLayout with index "
            + "less than mBaselineAlignedChildIndex has weight > 0, which "
            + "won't work.  Either remove the weight, or don't set "
            + "mBaselineAlignedChildIndex.");
}

boolean matchWidthLocally = false;
if (widthMode != MeasureSpec.EXACTLY && lp.width == LayoutParams.MATCH_PARENT) {
    // The width of the linear layout will scale, and at least one
    // child said it wanted to match our width. Set a flag
    // indicating that we need to remeasure at least that view when
    // we know our width.
    matchWidth = true;
    matchWidthLocally = true;
}

final int margin = lp.leftMargin + lp.rightMargin;
final int measuredWidth = child.getMeasuredWidth() + margin;
maxWidth = Math.max(maxWidth, measuredWidth);
childState = combineMeasuredStates(childState, child.getMeasuredState());

allFillParent = allFillParent && lp.width == LayoutParams.MATCH_PARENT;
if (lp.weight > 0) {
    /*
```

```
                 * Widths of weighted Views are bogus if we end up
                 * remeasuring, so keep them separate.
                 */
                weightedMaxWidth = Math.max(weightedMaxWidth,
                        matchWidthLocally ? margin : measuredWidth);
            } else {
                alternativeMaxWidth = Math.max(alternativeMaxWidth,
                        matchWidthLocally ? margin : measuredWidth);
            }

            i += getChildrenSkipCount(child, i);
        }

        if (nonSkippedChildCount > 0 && hasDividerBeforeChildAt(count)) {
            mTotalLength += mDividerHeight;
        }

        if (useLargestChild &&
                (heightMode == MeasureSpec.AT_MOST || heightMode == MeasureSpec.UNSPECIFIED)) {
            mTotalLength = 0;

            for (int i = 0; i < count; ++i) {
                final View child = getVirtualChildAt(i);
                if (child == null) {
                    mTotalLength += measureNullChild(i);
                    continue;
                }

                if (child.getVisibility() == GONE) {
                    i += getChildrenSkipCount(child, i);
                    continue;
                }

                final LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams)
                        child.getLayoutParams();
                // Account for negative margins
                final int totalLength = mTotalLength;
                mTotalLength = Math.max(totalLength, totalLength + largestChildHeight +
                        lp.topMargin + lp.bottomMargin + getNextLocationOffset(child));
            }
        }

        // Add in our padding
        mTotalLength += mPaddingTop + mPaddingBottom;

        int heightSize = mTotalLength;

        // Check against our minimum height
        heightSize = Math.max(heightSize, getSuggestedMinimumHeight());

        // Reconcile our calculated size with the heightMeasureSpec
        int heightSizeAndState = resolveSizeAndState(heightSize, heightMeasureSpec, 0);
        heightSize = heightSizeAndState & MEASURED_SIZE_MASK;
        // Either expand children with weight to take up available space or
        // shrink them if they extend beyond our current bounds. If we skipped
        // measurement on any children, we need to measure them now.
        int remainingExcess = heightSize - mTotalLength
                + (mAllowInconsistentMeasurement ? 0 : consumedExcessSpace);
        if (skippedMeasure || remainingExcess != 0 && totalWeight > 0.0f) {
            float remainingWeightSum = mWeightSum > 0.0f ? mWeightSum : totalWeight;

            mTotalLength = 0;

            for (int i = 0; i < count; ++i) {
                final View child = getVirtualChildAt(i);
                if (child == null || child.getVisibility() == View.GONE) {
                    continue;
                }

                final LayoutParams lp = (LayoutParams) child.getLayoutParams();
                final float childWeight = lp.weight;
                if (childWeight > 0) {
                    final int share = (int) (childWeight * remainingExcess / remainingWeightSum);
                    remainingExcess -= share;
                    remainingWeightSum -= childWeight;

                    final int childHeight;
                    if (mUseLargestChild && heightMode != MeasureSpec.EXACTLY) {
                        childHeight = largestChildHeight;
                    } else if (lp.height == 0 && (!mAllowInconsistentMeasurement
                            || heightMode == MeasureSpec.EXACTLY)) {
                        // This child needs to be laid out from scratch using
                        // only its share of excess space.
                        childHeight = share;
                    } else {
                        // This child had some intrinsic height to which we
                        // need to add its share of excess space.
                        childHeight = child.getMeasuredHeight() + share;
                    }

                    final int childHeightMeasureSpec = MeasureSpec.makeMeasureSpec(
                            Math.max(0, childHeight), MeasureSpec.EXACTLY);
                    final int childWidthMeasureSpec = getChildMeasureSpec(widthMeasureSpec,
                            mPaddingLeft + mPaddingRight + lp.leftMargin + lp.rightMargin,
                            lp.width);
                    child.measure(childWidthMeasureSpec, childHeightMeasureSpec);

                    // Child may now not fit in vertical dimension.
                    childState = combineMeasuredStates(childState, child.getMeasuredState()
                            & (MEASURED_STATE_MASK>>MEASURED_HEIGHT_STATE_SHIFT));
                }

                final int margin =  lp.leftMargin + lp.rightMargin;
                final int measuredWidth = child.getMeasuredWidth() + margin;
                maxWidth = Math.max(maxWidth, measuredWidth);

                boolean matchWidthLocally = widthMode != MeasureSpec.EXACTLY &&
                        lp.width == LayoutParams.MATCH_PARENT;

                alternativeMaxWidth = Math.max(alternativeMaxWidth,
                        matchWidthLocally ? margin : measuredWidth);

                allFillParent = allFillParent && lp.width == LayoutParams.MATCH_PARENT;

                final int totalLength = mTotalLength;
                mTotalLength = Math.max(totalLength, totalLength + child.getMeasuredHeight() +
                        lp.topMargin + lp.bottomMargin + getNextLocationOffset(child));
            }

            // Add in our padding
            mTotalLength += mPaddingTop + mPaddingBottom;
            // TODO: Should we recompute the heightSpec based on the new total length?
        } else {
            alternativeMaxWidth = Math.max(alternativeMaxWidth,
                                           weightedMaxWidth);


            // We have no limit, so make all weighted views as tall as the largest child.
            // Children will have already been measured once.
            if (useLargestChild && heightMode != MeasureSpec.EXACTLY) {
                for (int i = 0; i < count; i++) {
                    final View child = getVirtualChildAt(i);
                    if (child == null || child.getVisibility() == View.GONE) {
                        continue;
                    }

                    final LinearLayout.LayoutParams lp =
                            (LinearLayout.LayoutParams) child.getLayoutParams();

                    float childExtra = lp.weight;
                    if (childExtra > 0) {
                        child.measure(
                                MeasureSpec.makeMeasureSpec(child.getMeasuredWidth(),
                                        MeasureSpec.EXACTLY),
                                MeasureSpec.makeMeasureSpec(largestChildHeight,
                                        MeasureSpec.EXACTLY));
                    }
                }
            }
        }

        if (!allFillParent && widthMode != MeasureSpec.EXACTLY) {
            maxWidth = alternativeMaxWidth;
        }

        maxWidth += mPaddingLeft + mPaddingRight;

        // Check against our minimum width
        maxWidth = Math.max(maxWidth, getSuggestedMinimumWidth());

        setMeasuredDimension(resolveSizeAndState(maxWidth, widthMeasureSpec, childState),
                heightSizeAndState);

        if (matchWidth) {
            forceUniformWidth(count, heightMeasureSpec);
        }
    }

    private void forceUniformWidth(int count, int heightMeasureSpec) {
        // Pretend that the linear layout has an exact size.
        int uniformMeasureSpec = MeasureSpec.makeMeasureSpec(getMeasuredWidth(),
                MeasureSpec.EXACTLY);
        for (int i = 0; i< count; ++i) {
           final View child = getVirtualChildAt(i);
           if (child != null && child.getVisibility() != GONE) {
               LinearLayout.LayoutParams lp = ((LinearLayout.LayoutParams)child.getLayoutParams());

               if (lp.width == LayoutParams.MATCH_PARENT) {
                   // Temporarily force children to reuse their old measured height
                   // FIXME: this may not be right for something like wrapping text?
                   int oldHeight = lp.height;
                   lp.height = child.getMeasuredHeight();

                   // Remeasue with new dimensions
                   measureChildWithMargins(child, uniformMeasureSpec, 0, heightMeasureSpec, 0);
                   lp.height = oldHeight;
               }
           }
        }
    }

    /**
     * Measures the children when the orientation of this LinearLayout is set
     * to {@link #HORIZONTAL}.
     *
     * @param widthMeasureSpec Horizontal space requirements as imposed by the parent.
     * @param heightMeasureSpec Vertical space requirements as imposed by the parent.
     *
     * @see #getOrientation()
     * @see #setOrientation(int)
     * @see #onMeasure(int, int)
     */
    void measureHorizontal(int widthMeasureSpec, int heightMeasureSpec) {
        mTotalLength = 0;
        int maxHeight = 0;
        int childState = 0;
        int alternativeMaxHeight = 0;
        int weightedMaxHeight = 0;
        boolean allFillParent = true;
        float totalWeight = 0;

        final int count = getVirtualChildCount();

        final int widthMode = MeasureSpec.getMode(widthMeasureSpec);
        final int heightMode = MeasureSpec.getMode(heightMeasureSpec);
```

```java
boolean matchHeight = false;
boolean skippedMeasure = false;

if (mMaxAscent == null || mMaxDescent == null) {
    mMaxAscent = new int[VERTICAL_GRAVITY_COUNT];
    mMaxDescent = new int[VERTICAL_GRAVITY_COUNT];
}

final int[] maxAscent = mMaxAscent;
final int[] maxDescent = mMaxDescent;

maxAscent[0] = maxAscent[1] = maxAscent[2] = maxAscent[3] = -1;
maxDescent[0] = maxDescent[1] = maxDescent[2] = maxDescent[3] = -1;

final boolean baselineAligned = mBaselineAligned;
final boolean useLargestChild = mUseLargestChild;

final boolean isExactly = widthMode == MeasureSpec.EXACTLY;

int largestChildWidth = Integer.MIN_VALUE;
int usedExcessSpace = 0;

int nonSkippedChildCount = 0;

// See how wide everyone is. Also remember max height.
for (int i = 0; i < count; ++i) {
    final View child = getVirtualChildAt(i);
    if (child == null) {
        mTotalLength += measureNullChild(i);
        continue;
    }

    if (child.getVisibility() == GONE) {
        i += getChildrenSkipCount(child, i);
        continue;
    }

    nonSkippedChildCount++;
    if (hasDividerBeforeChildAt(i)) {
        mTotalLength += mDividerWidth;
    }

    final LayoutParams lp = (LayoutParams) child.getLayoutParams();

    totalWeight += lp.weight;

    final boolean useExcessSpace = lp.width == 0 && lp.weight > 0;
    if (widthMode == MeasureSpec.EXACTLY && useExcessSpace) {
        // Optimization: don't bother measuring children who are only
        // laid out using excess space. These views will get measured
        // later if we have space to distribute.
        if (isExactly) {
            mTotalLength += lp.leftMargin + lp.rightMargin;
        } else {
            final int totalLength = mTotalLength;
            mTotalLength = Math.max(totalLength, totalLength +
                    lp.leftMargin + lp.rightMargin);
        }

        // Baseline alignment requires to measure widgets to obtain the
        // baseline offset (in particular for TextViews). The following
        // defeats the optimization mentioned above. Allow the child to
        // use as much space as it wants because we can shrink things
        // later (and re-measure).
        if (baselineAligned) {
            final int freeWidthSpec = MeasureSpec.makeSafeMeasureSpec(
                    MeasureSpec.getSize(widthMeasureSpec), MeasureSpec.UNSPECIFIED);
            final int freeHeightSpec = MeasureSpec.makeSafeMeasureSpec(
                    MeasureSpec.getSize(heightMeasureSpec), MeasureSpec.UNSPECIFIED);
            child.measure(freeWidthSpec, freeHeightSpec);
        } else {
            skippedMeasure = true;
        }
    } else {
        if (useExcessSpace) {
            // The widthMode is either UNSPECIFIED or AT_MOST, and
            // this child is only laid out using excess space. Measure
            // using WRAP_CONTENT so that we can find out the view's
            // optimal width. We'll restore the original width of 0
            // after measurement.
            lp.width = LayoutParams.WRAP_CONTENT;
        }

        // Determine how big this child would like to be. If this or
        // previous children have given a weight, then we allow it to
        // use all available space (and we will shrink things later
        // if needed).
        final int usedWidth = totalWeight == 0 ? mTotalLength : 0;
        measureChildBeforeLayout(child, i, widthMeasureSpec, usedWidth,
                heightMeasureSpec, 0);

        final int childWidth = child.getMeasuredWidth();
        if (useExcessSpace) {
            // Restore the original width and record how much space
            // we've allocated to excess-only children so that we can
            // match the behavior of EXACTLY measurement.
            lp.width = 0;
            usedExcessSpace += childWidth;
        }

        if (isExactly) {
            mTotalLength += childWidth + lp.leftMargin + lp.rightMargin
                    + getNextLocationOffset(child);
        } else {
            final int totalLength = mTotalLength;
            mTotalLength = Math.max(totalLength, totalLength + childWidth + lp.leftMargin
                    + lp.rightMargin + getNextLocationOffset(child));
        }

        if (useLargestChild) {
            largestChildWidth = Math.max(childWidth, largestChildWidth);
        }
    }

    boolean matchHeightLocally = false;
    if (heightMode != MeasureSpec.EXACTLY && lp.height == LayoutParams.MATCH_PARENT) {
        // The height of the linear layout will scale, and at least one
        // child said it wanted to match our height. Set a flag indicating that
        // we need to remeasure at least that view when we know our height.
        matchHeight = true;
        matchHeightLocally = true;
    }

    final int margin = lp.topMargin + lp.bottomMargin;
    final int childHeight = child.getMeasuredHeight() + margin;
    childState = combineMeasuredStates(childState, child.getMeasuredState());

    if (baselineAligned) {
        final int childBaseline = child.getBaseline();
        if (childBaseline != -1) {
            // Translates the child's vertical gravity into an index
            // in the range 0..VERTICAL_GRAVITY_COUNT
            final int gravity = (lp.gravity < 0 ? mGravity : lp.gravity)
                    & Gravity.VERTICAL_GRAVITY_MASK;
            final int index = ((gravity >> Gravity.AXIS_Y_SHIFT)
                    & ~Gravity.AXIS_SPECIFIED) >> 1;

            maxAscent[index] = Math.max(maxAscent[index], childBaseline);
            maxDescent[index] = Math.max(maxDescent[index], childHeight - childBaseline);
        }
    }

    maxHeight = Math.max(maxHeight, childHeight);
```

```
            allFillParent = allFillParent && lp.height == LayoutParams.MATCH_PARENT;
            if (lp.weight > 0) {
                /*
                 * Heights of weighted Views are bogus if we end up
                 * remeasuring, so keep them separate.
                 */
                weightedMaxHeight = Math.max(weightedMaxHeight,
                        matchHeightLocally ? margin : childHeight);
            } else {
                alternativeMaxHeight = Math.max(alternativeMaxHeight,
                        matchHeightLocally ? margin : childHeight);
            }

            i += getChildrenSkipCount(child, i);
        }

        if (nonSkippedChildCount > 0 && hasDividerBeforeChildAt(count)) {
            mTotalLength += mDividerWidth;
        }

        // Check mMaxAscent[INDEX_TOP] first because it maps to Gravity.TOP,
        // the most common case
        if (maxAscent[INDEX_TOP] != -1 ||
                maxAscent[INDEX_CENTER_VERTICAL] != -1 ||
                maxAscent[INDEX_BOTTOM] != -1 ||
                maxAscent[INDEX_FILL] != -1) {
            final int ascent = Math.max(maxAscent[INDEX_FILL],
                    Math.max(maxAscent[INDEX_CENTER_VERTICAL],
                    Math.max(maxAscent[INDEX_TOP], maxAscent[INDEX_BOTTOM])));
            final int descent = Math.max(maxDescent[INDEX_FILL],
                    Math.max(maxDescent[INDEX_CENTER_VERTICAL],
                    Math.max(maxDescent[INDEX_TOP], maxDescent[INDEX_BOTTOM])));
            maxHeight = Math.max(maxHeight, ascent + descent);
        }

        if (useLargestChild &&
                (widthMode == MeasureSpec.AT_MOST || widthMode == MeasureSpec.UNSPECIFIED)) {
            mTotalLength = 0;

            for (int i = 0; i < count; ++i) {
                final View child = getVirtualChildAt(i);
                if (child == null) {
                    mTotalLength += measureNullChild(i);
                    continue;
                }

                if (child.getVisibility() == GONE) {
                    i += getChildrenSkipCount(child, i);
                    continue;
                }

                final LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams)
                        child.getLayoutParams();
                if (isExactly) {
                    mTotalLength += largestChildWidth + lp.leftMargin + lp.rightMargin +
                            getNextLocationOffset(child);
                } else {
                    final int totalLength = mTotalLength;
                    mTotalLength = Math.max(totalLength, totalLength + largestChildWidth +
                            lp.leftMargin + lp.rightMargin + getNextLocationOffset(child));
                }
            }
        }

        // Add in our padding
        mTotalLength += mPaddingLeft + mPaddingRight;

        int widthSize = mTotalLength;

        // Check against our minimum width
        widthSize = Math.max(widthSize, getSuggestedMinimumWidth());
```

```
// Reconcile our calculated size with the widthMeasureSpec
int widthSizeAndState = resolveSizeAndState(widthSize, widthMeasureSpec, 0);
widthSize = widthSizeAndState & MEASURED_SIZE_MASK;

// Either expand children with weight to take up available space or
// shrink them if they extend beyond our current bounds. If we skipped
// measurement on any children, we need to measure them now.
int remainingExcess = widthSize - mTotalLength
        + (mAllowInconsistentMeasurement ? 0 : usedExcessSpace);
if (skippedMeasure || remainingExcess != 0 && totalWeight > 0.0f) {
    float remainingWeightSum = mWeightSum > 0.0f ? mWeightSum : totalWeight;

    maxAscent[0] = maxAscent[1] = maxAscent[2] = maxAscent[3] = -1;
    maxDescent[0] = maxDescent[1] = maxDescent[2] = maxDescent[3] = -1;
    maxHeight = -1;

    mTotalLength = 0;

    for (int i = 0; i < count; ++i) {
        final View child = getVirtualChildAt(i);
        if (child == null || child.getVisibility() == View.GONE) {
            continue;
        }

        final LayoutParams lp = (LayoutParams) child.getLayoutParams();
        final float childWeight = lp.weight;
        if (childWeight > 0) {
            final int share = (int) (childWeight * remainingExcess / remainingWeightSum);
            remainingExcess -= share;
            remainingWeightSum -= childWeight;

            final int childWidth;
            if (mUseLargestChild && widthMode != MeasureSpec.EXACTLY) {
                childWidth = largestChildWidth;
            } else if (lp.width == 0 && (!mAllowInconsistentMeasurement
                    || widthMode == MeasureSpec.EXACTLY)) {
                // This child needs to be laid out from scratch using
                // only its share of excess space.
                childWidth = share;
            } else {
                // This child had some intrinsic width to which we
                // need to add its share of excess space.
                childWidth = child.getMeasuredWidth() + share;
            }

            final int childWidthMeasureSpec = MeasureSpec.makeMeasureSpec(
                    Math.max(0, childWidth), MeasureSpec.EXACTLY);
            final int childHeightMeasureSpec = getChildMeasureSpec(heightMeasureSpec,
                    mPaddingTop + mPaddingBottom + lp.topMargin + lp.bottomMargin,
                    lp.height);
            child.measure(childWidthMeasureSpec, childHeightMeasureSpec);

            // Child may now not fit in horizontal dimension.
            childState = combineMeasuredStates(childState,
                    child.getMeasuredState() & MEASURED_STATE_MASK);
        }

        if (isExactly) {
            mTotalLength += child.getMeasuredWidth() + lp.leftMargin + lp.rightMargin +
                    getNextLocationOffset(child);
        } else {
            final int totalLength = mTotalLength;
            mTotalLength = Math.max(totalLength, totalLength + child.getMeasuredWidth() +
                    lp.leftMargin + lp.rightMargin + getNextLocationOffset(child));
        }

        boolean matchHeightLocally = heightMode != MeasureSpec.EXACTLY &&
                lp.height == LayoutParams.MATCH_PARENT;

        final int margin = lp.topMargin + lp .bottomMargin;
        int childHeight = child.getMeasuredHeight() + margin;
        maxHeight = Math.max(maxHeight, childHeight);
        alternativeMaxHeight = Math.max(alternativeMaxHeight,
                matchHeightLocally ? margin : childHeight);

        allFillParent = allFillParent && lp.height == LayoutParams.MATCH_PARENT;

        if (baselineAligned) {
            final int childBaseline = child.getBaseline();
            if (childBaseline != -1) {
                // Translates the child's vertical gravity into an index in the range 0..2
                final int gravity = (lp.gravity < 0 ? mGravity : lp.gravity)
                        & Gravity.VERTICAL_GRAVITY_MASK;
                final int index = ((gravity >> Gravity.AXIS_Y_SHIFT)
                        & ~Gravity.AXIS_SPECIFIED) >> 1;

                maxAscent[index] = Math.max(maxAscent[index], childBaseline);
                maxDescent[index] = Math.max(maxDescent[index],
                        childHeight - childBaseline);
            }
        }
    }

    // Add in our padding
    mTotalLength += mPaddingLeft + mPaddingRight;
    // TODO: Should we update widthSize with the new total length?

    // Check mMaxAscent[INDEX_TOP] first because it maps to Gravity.TOP,
    // the most common case
    if (maxAscent[INDEX_TOP] != -1 ||
            maxAscent[INDEX_CENTER_VERTICAL] != -1 ||
            maxAscent[INDEX_BOTTOM] != -1 ||
            maxAscent[INDEX_FILL] != -1) {
        final int ascent = Math.max(maxAscent[INDEX_FILL],
                Math.max(maxAscent[INDEX_CENTER_VERTICAL],
                Math.max(maxAscent[INDEX_TOP], maxAscent[INDEX_BOTTOM])));
        final int descent = Math.max(maxDescent[INDEX_FILL],
                Math.max(maxDescent[INDEX_CENTER_VERTICAL],
                Math.max(maxDescent[INDEX_TOP], maxDescent[INDEX_BOTTOM])));
        maxHeight = Math.max(maxHeight, ascent + descent);
    }
} else {
    alternativeMaxHeight = Math.max(alternativeMaxHeight, weightedMaxHeight);

    // We have no limit, so make all weighted views as wide as the largest child.
    // Children will have already been measured once.
    if (useLargestChild && widthMode != MeasureSpec.EXACTLY) {
        for (int i = 0; i < count; i++) {
            final View child = getVirtualChildAt(i);
            if (child == null || child.getVisibility() == View.GONE) {
                continue;
            }

            final LinearLayout.LayoutParams lp =
                    (LinearLayout.LayoutParams) child.getLayoutParams();

            float childExtra = lp.weight;
            if (childExtra > 0) {
                child.measure(
                        MeasureSpec.makeMeasureSpec(largestChildWidth, MeasureSpec.EXACTLY),
                        MeasureSpec.makeMeasureSpec(child.getMeasuredHeight(),
                                MeasureSpec.EXACTLY));
            }
        }
    }
}

if (!allFillParent && heightMode != MeasureSpec.EXACTLY) {
    maxHeight = alternativeMaxHeight;
}

maxHeight += mPaddingTop + mPaddingBottom;

// Check against our minimum height
```

```
        maxHeight = Math.max(maxHeight, getSuggestedMinimumHeight());

        setMeasuredDimension(widthSizeAndState | (childState&MEASURED_STATE_MASK),
                resolveSizeAndState(maxHeight, heightMeasureSpec,
                        (childState<<MEASURED_HEIGHT_STATE_SHIFT)));

        if (matchHeight) {
            forceUniformHeight(count, widthMeasureSpec);
        }
    }

    private void forceUniformHeight(int count, int widthMeasureSpec) {
        // Pretend that the linear layout has an exact size. This is the measured height of
        // ourselves. The measured height should be the max height of the children, changed
        // to accommodate the heightMeasureSpec from the parent
        int uniformMeasureSpec = MeasureSpec.makeMeasureSpec(getMeasuredHeight(),
                MeasureSpec.EXACTLY);
        for (int i = 0; i < count; ++i) {
           final View child = getVirtualChildAt(i);
           if (child != null && child.getVisibility() != GONE) {
               LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams) child.getLayoutParams();

               if (lp.height == LayoutParams.MATCH_PARENT) {
                   // Temporarily force children to reuse their old measured width
                   // FIXME: this may not be right for something like wrapping text?
                   int oldWidth = lp.width;
                   lp.width = child.getMeasuredWidth();

                   // Remeasure with new dimensions
                   measureChildWithMargins(child, widthMeasureSpec, 0, uniformMeasureSpec, 0);
                   lp.width = oldWidth;
               }
           }
        }
    }

    /**
     * <p>Returns the number of children to skip after measuring/laying out
     * the specified child.</p>
     *
     * @param child the child after which we want to skip children
     * @param index the index of the child after which we want to skip children
     * @return the number of children to skip, 0 by default
     */
    int getChildrenSkipCount(View child, int index) {
        return 0;
    }

    /**
     * <p>Returns the size (width or height) that should be occupied by a null
     * child.</p>
     *
     * @param childIndex the index of the null child
     * @return the width or height of the child depending on the orientation
     */
    int measureNullChild(int childIndex) {
        return 0;
    }

    /**
     * <p>Measure the child according to the parent's measure specs. This
     * method should be overriden by subclasses to force the sizing of
     * children. This method is called by {@link #measureVertical(int, int)} and
     * {@link #measureHorizontal(int, int)}.</p>
     *
     * @param child the child to measure
     * @param childIndex the index of the child in this view
     * @param widthMeasureSpec horizontal space requirements as imposed by the parent
     * @param totalWidth extra space that has been used up by the parent horizontally
     * @param heightMeasureSpec vertical space requirements as imposed by the parent
     * @param totalHeight extra space that has been used up by the parent vertically
     */
    void measureChildBeforeLayout(View child, int childIndex,
            int widthMeasureSpec, int totalWidth, int heightMeasureSpec,
            int totalHeight) {
        measureChildWithMargins(child, widthMeasureSpec, totalWidth,
                heightMeasureSpec, totalHeight);
    }

    /**
     * <p>Return the location offset of the specified child. This can be used
     * by subclasses to change the location of a given widget.</p>
     *
     * @param child the child for which to obtain the location offset
     * @return the location offset in pixels
     */
    int getLocationOffset(View child) {
        return 0;
    }

    /**
     * <p>Return the size offset of the next sibling of the specified child.
     * This can be used by subclasses to change the location of the widget
     * following <code>child</code>.</p>
     *
     * @param child the child whose next sibling will be moved
     * @return the location offset of the next child in pixels
     */
    int getNextLocationOffset(View child) {
        return 0;
    }

    @Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
        if (mOrientation == VERTICAL) {
            layoutVertical(l, t, r, b);
        } else {
            layoutHorizontal(l, t, r, b);
        }
    }

    /**
     * Position the children during a layout pass if the orientation of this
     * LinearLayout is set to {@link #VERTICAL}.
     *
     * @see #getOrientation()
     * @see #setOrientation(int)
     * @see #onLayout(boolean, int, int, int, int)
     * @param left
     * @param top
     * @param right
     * @param bottom
     */
    void layoutVertical(int left, int top, int right, int bottom) {
        final int paddingLeft = mPaddingLeft;

        int childTop;
        int childLeft;

        // Where right end of child should go
        final int width = right - left;
        int childRight = width - mPaddingRight;

        // Space available for child
        int childSpace = width - paddingLeft - mPaddingRight;

        final int count = getVirtualChildCount();

        final int majorGravity = mGravity & Gravity.VERTICAL_GRAVITY_MASK;
        final int minorGravity = mGravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK;

        switch (majorGravity) {
           case Gravity.BOTTOM:
               // mTotalLength contains the padding already
               childTop = mPaddingTop + bottom - top - mTotalLength;
               break;

               // mTotalLength contains the padding already
           case Gravity.CENTER_VERTICAL:
               childTop = mPaddingTop + (bottom - top - mTotalLength) / 2;
               break;

           case Gravity.TOP:
           default:
               childTop = mPaddingTop;
               break;
        }

        for (int i = 0; i < count; i++) {
            final View child = getVirtualChildAt(i);
            if (child == null) {
                childTop += measureNullChild(i);
            } else if (child.getVisibility() != GONE) {
                final int childWidth = child.getMeasuredWidth();
                final int childHeight = child.getMeasuredHeight();

                final LinearLayout.LayoutParams lp =
                        (LinearLayout.LayoutParams) child.getLayoutParams();

                int gravity = lp.gravity;
                if (gravity < 0) {
                    gravity = minorGravity;
                }
                final int layoutDirection = getLayoutDirection();
                final int absoluteGravity = Gravity.getAbsoluteGravity(gravity, layoutDirection);
                switch (absoluteGravity & Gravity.HORIZONTAL_GRAVITY_MASK) {
                    case Gravity.CENTER_HORIZONTAL:
                        childLeft = paddingLeft + ((childSpace - childWidth) / 2)
                                + lp.leftMargin - lp.rightMargin;
                        break;

                    case Gravity.RIGHT:
                        childLeft = childRight - childWidth - lp.rightMargin;
                        break;

                    case Gravity.LEFT:
                    default:
                        childLeft = paddingLeft + lp.leftMargin;
                        break;
                }

                if (hasDividerBeforeChildAt(i)) {
                    childTop += mDividerHeight;
                }

                childTop += lp.topMargin;
                setChildFrame(child, childLeft, childTop + getLocationOffset(child),
                        childWidth, childHeight);
                childTop += childHeight + lp.bottomMargin + getNextLocationOffset(child);

                i += getChildrenSkipCount(child, i);
            }
        }
    }

    @Override
    public void onRtlPropertiesChanged(@ResolvedLayoutDir int layoutDirection) {
        super.onRtlPropertiesChanged(layoutDirection);
        if (layoutDirection != mLayoutDirection) {
            mLayoutDirection = layoutDirection;
            if (mOrientation == HORIZONTAL) {
                requestLayout();
            }
        }
    }
```

```java
/**
 * Position the children during a layout pass if the orientation of this
 * LinearLayout is set to {@link #HORIZONTAL}.
 *
 * @see #getOrientation()
 * @see #setOrientation(int)
 * @see #onLayout(boolean, int, int, int, int)
 * @param left
 * @param top
 * @param right
 * @param bottom
 */
void layoutHorizontal(int left, int top, int right, int bottom) {
    final boolean isLayoutRtl = isLayoutRtl();
    final int paddingTop = mPaddingTop;

    int childTop;
    int childLeft;

    // Where bottom of child should go
    final int height = bottom - top;
    int childBottom = height - mPaddingBottom;

    // Space available for child
    int childSpace = height - paddingTop - mPaddingBottom;

    final int count = getVirtualChildCount();

    final int majorGravity = mGravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK;
    final int minorGravity = mGravity & Gravity.VERTICAL_GRAVITY_MASK;

    final boolean baselineAligned = mBaselineAligned;

    final int[] maxAscent = mMaxAscent;
    final int[] maxDescent = mMaxDescent;

    final int layoutDirection = getLayoutDirection();
    switch (Gravity.getAbsoluteGravity(majorGravity, layoutDirection)) {
        case Gravity.RIGHT:
            // mTotalLength contains the padding already
            childLeft = mPaddingLeft + right - left - mTotalLength;
            break;

        case Gravity.CENTER_HORIZONTAL:
            // mTotalLength contains the padding already
            childLeft = mPaddingLeft + (right - left - mTotalLength) / 2;
            break;

        case Gravity.LEFT:
        default:
            childLeft = mPaddingLeft;
            break;
    }

    int start = 0;
    int dir = 1;
    //In case of RTL, start drawing from the last child.
    if (isLayoutRtl) {
        start = count - 1;
        dir = -1;
    }

    for (int i = 0; i < count; i++) {
        final int childIndex = start + dir * i;
        final View child = getVirtualChildAt(childIndex);
        if (child == null) {
            childLeft += measureNullChild(childIndex);
        } else if (child.getVisibility() != GONE) {
            final int childWidth = child.getMeasuredWidth();
            final int childHeight = child.getMeasuredHeight();
            int childBaseline = -1;
            final LinearLayout.LayoutParams lp =
                    (LinearLayout.LayoutParams) child.getLayoutParams();

            if (baselineAligned && lp.height != LayoutParams.MATCH_PARENT) {
                childBaseline = child.getBaseline();
            }

            int gravity = lp.gravity;
            if (gravity < 0) {
                gravity = minorGravity;
            }

            switch (gravity & Gravity.VERTICAL_GRAVITY_MASK) {
                case Gravity.TOP:
                    childTop = paddingTop + lp.topMargin;
                    if (childBaseline != -1) {
                        childTop += maxAscent[INDEX_TOP] - childBaseline;
                    }
                    break;

                case Gravity.CENTER_VERTICAL:
                    // Removed support for baseline alignment when layout_gravity or
                    // gravity == center_vertical. See bug #1038483.
                    // Keep the code around if we need to re-enable this feature
                    // if (childBaseline != -1) {
                    //     // Align baselines vertically only if the child is smaller than us
                    //     if (childSpace - childHeight > 0) {
                    //         childTop = paddingTop + (childSpace / 2) - childBaseline;
                    //     } else {
                    //         childTop = paddingTop + (childSpace - childHeight) / 2;
                    //     }
                    // } else {
                    childTop = paddingTop + ((childSpace - childHeight) / 2)
                            + lp.topMargin - lp.bottomMargin;
                    break;

                case Gravity.BOTTOM:
                    childTop = childBottom - childHeight - lp.bottomMargin;
                    if (childBaseline != -1) {
                        int descent = child.getMeasuredHeight() - childBaseline;
                        childTop -= (maxDescent[INDEX_BOTTOM] - descent);
                    }
                    break;
                default:
                    childTop = paddingTop;
                    break;
            }

            if (hasDividerBeforeChildAt(childIndex)) {
                childLeft += mDividerWidth;
            }

            childLeft += lp.leftMargin;
            setChildFrame(child, childLeft + getLocationOffset(child), childTop,
                    childWidth, childHeight);
            childLeft += childWidth + lp.rightMargin +
                    getNextLocationOffset(child);

            i += getChildrenSkipCount(child, childIndex);
        }
    }
}

private void setChildFrame(View child, int left, int top, int width, int height) {
    child.layout(left, top, left + width, top + height);
}

/**
 * Should the layout be a column or a row.
 * @param orientation Pass {@link #HORIZONTAL} or {@link #VERTICAL}. Default
 * value is {@link #HORIZONTAL}.
 *
```

```java
     * @attr ref android.R.styleable#LinearLayout_orientation
     */
    public void setOrientation(@OrientationMode int orientation) {
        if (mOrientation != orientation) {
            mOrientation = orientation;
            requestLayout();
        }
    }

    /**
     * Returns the current orientation.
     *
     * @return either {@link #HORIZONTAL} or {@link #VERTICAL}
     */
    @OrientationMode
    public int getOrientation() {
        return mOrientation;
    }

    /**
     * Describes how the child views are positioned. Defaults to GRAVITY_TOP. If
     * this layout has a VERTICAL orientation, this controls where all the child
     * views are placed if there is extra vertical space. If this layout has a
     * HORIZONTAL orientation, this controls the alignment of the children.
     *
     * @param gravity See {@link android.view.Gravity}
     *
     * @attr ref android.R.styleable#LinearLayout_gravity
     */
    @android.view.RemotableViewMethod
    public void setGravity(int gravity) {
        if (mGravity != gravity) {
            if ((gravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK) == 0) {
                gravity |= Gravity.START;
            }

            if ((gravity & Gravity.VERTICAL_GRAVITY_MASK) == 0) {
                gravity |= Gravity.TOP;
            }

            mGravity = gravity;
            requestLayout();
        }
    }

    /**
     * Returns the current gravity. See {@link android.view.Gravity}
     *
     * @return the current gravity.
     * @see #setGravity
     */
    public int getGravity() {
        return mGravity;
    }

    @android.view.RemotableViewMethod
    public void setHorizontalGravity(int horizontalGravity) {
        final int gravity = horizontalGravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK;
        if ((mGravity & Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK) != gravity) {
            mGravity = (mGravity & ~Gravity.RELATIVE_HORIZONTAL_GRAVITY_MASK) | gravity;
            requestLayout();
        }
    }

    @android.view.RemotableViewMethod
    public void setVerticalGravity(int verticalGravity) {
        final int gravity = verticalGravity & Gravity.VERTICAL_GRAVITY_MASK;
        if ((mGravity & Gravity.VERTICAL_GRAVITY_MASK) != gravity) {
            mGravity = (mGravity & ~Gravity.VERTICAL_GRAVITY_MASK) | gravity;
            requestLayout();
        }
    }
```

```java
@Override
public LayoutParams generateLayoutParams(AttributeSet attrs) {
    return new LinearLayout.LayoutParams(getContext(), attrs);
}

/**
 * Returns a set of layout parameters with a width of
 * {@link android.view.ViewGroup.LayoutParams#MATCH_PARENT}
 * and a height of {@link android.view.ViewGroup.LayoutParams#WRAP_CONTENT}
 * when the layout's orientation is {@link #VERTICAL}. When the orientation is
 * {@link #HORIZONTAL}, the width is set to {@link LayoutParams#WRAP_CONTENT}
 * and the height to {@link LayoutParams#WRAP_CONTENT}.
 */
@Override
protected LayoutParams generateDefaultLayoutParams() {
    if (mOrientation == HORIZONTAL) {
        return new LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT);
    } else if (mOrientation == VERTICAL) {
        return new LayoutParams(LayoutParams.MATCH_PARENT, LayoutParams.WRAP_CONTENT);
    }
    return null;
}

@Override
protected LayoutParams generateLayoutParams(ViewGroup.LayoutParams lp) {
    if (sPreserveMarginParamsInLayoutParamConversion) {
        if (lp instanceof LayoutParams) {
            return new LayoutParams((LayoutParams) lp);
        } else if (lp instanceof MarginLayoutParams) {
            return new LayoutParams((MarginLayoutParams) lp);
        }
    }
    return new LayoutParams(lp);
}


// Override to allow type-checking of LayoutParams.
@Override
protected boolean checkLayoutParams(ViewGroup.LayoutParams p) {
    return p instanceof LinearLayout.LayoutParams;
}

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

/** @hide */
@Override
protected void encodeProperties(@NonNull ViewHierarchyEncoder encoder) {
    super.encodeProperties(encoder);
    encoder.addProperty("layout:baselineAligned", mBaselineAligned);
    encoder.addProperty("layout:baselineAlignedChildIndex", mBaselineAlignedChildIndex);
    encoder.addProperty("measurement:baselineChildTop", mBaselineChildTop);
    encoder.addProperty("measurement:orientation", mOrientation);
    encoder.addProperty("measurement:gravity", mGravity);
    encoder.addProperty("measurement:totalLength", mTotalLength);
    encoder.addProperty("layout:totalLength", mTotalLength);
    encoder.addProperty("layout:useLargestChild", mUseLargestChild);
}

/**
 * Per-child layout information associated with ViewLinearLayout.
 *
 * @attr ref android.R.styleable#LinearLayout_Layout_layout_weight
 * @attr ref android.R.styleable#LinearLayout_Layout_layout_gravity
 */
public static class LayoutParams extends ViewGroup.MarginLayoutParams {
    /**
     * Indicates how much of the extra space in the LinearLayout will be
     * allocated to the view associated with these LayoutParams. Specify
     * 0 if the view should not be stretched. Otherwise the extra pixels
     * will be pro-rated among all views whose weight is greater than 0.
     */
    @ViewDebug.ExportedProperty(category = "layout")
    public float weight;

    /**
     * Gravity for the view associated with these LayoutParams.
     *
     * @see android.view.Gravity
     */
    @ViewDebug.ExportedProperty(category = "layout", mapping = {
        @ViewDebug.IntToString(from =  -1,                       to = "NONE"),
        @ViewDebug.IntToString(from = Gravity.NO_GRAVITY,        to = "NONE"),
        @ViewDebug.IntToString(from = Gravity.TOP,               to = "TOP"),
        @ViewDebug.IntToString(from = Gravity.BOTTOM,            to = "BOTTOM"),
        @ViewDebug.IntToString(from = Gravity.LEFT,              to = "LEFT"),
        @ViewDebug.IntToString(from = Gravity.RIGHT,             to = "RIGHT"),
        @ViewDebug.IntToString(from = Gravity.START,             to = "START"),
        @ViewDebug.IntToString(from = Gravity.END,               to = "END"),
        @ViewDebug.IntToString(from = Gravity.CENTER_VERTICAL,   to = "CENTER_VERTICAL"),
        @ViewDebug.IntToString(from = Gravity.FILL_VERTICAL,     to = "FILL_VERTICAL"),
        @ViewDebug.IntToString(from = Gravity.CENTER_HORIZONTAL, to = "CENTER_HORIZONTAL"),
        @ViewDebug.IntToString(from = Gravity.FILL_HORIZONTAL,   to = "FILL_HORIZONTAL"),
        @ViewDebug.IntToString(from = Gravity.CENTER,            to = "CENTER"),
        @ViewDebug.IntToString(from = Gravity.FILL,              to = "FILL")
    })
    public int gravity = -1;

    /**
     * {@inheritDoc}
     */
    public LayoutParams(Context c, AttributeSet attrs) {
        super(c, attrs);
        TypedArray a =
                c.obtainStyledAttributes(attrs, com.android.internal.R.styleable.LinearLayout_Layout);

        weight = a.getFloat(com.android.internal.R.styleable.LinearLayout_Layout_layout_weight, 0);
        gravity = a.getInt(com.android.internal.R.styleable.LinearLayout_Layout_layout_gravity, -1);

        a.recycle();
    }

    /**
     * {@inheritDoc}
     */
    public LayoutParams(int width, int height) {
        super(width, height);
        weight = 0;
    }

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

    /**
     * {@inheritDoc}
     */
    public LayoutParams(ViewGroup.LayoutParams p) {
        super(p);
    }
```

```
        /**
         * {@inheritDoc}
         */
        public LayoutParams(ViewGroup.MarginLayoutParams source) {
            super(source);
        }

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

            this.weight = source.weight;
            this.gravity = source.gravity;
        }

        @Override
        public String debug(String output) {
            return output + "LinearLayout.LayoutParams={width=" + sizeToString(width) +
                    ", height=" + sizeToString(height) + " weight=" + weight +  "}";
        }

        /** @hide */
        @Override
        protected void encodeProperties(@NonNull ViewHierarchyEncoder encoder) {
            super.encodeProperties(encoder);

            encoder.addProperty("layout:weight", weight);
            encoder.addProperty("layout:gravity", gravity);
        }
    }
}
```