Threads in Android

Processes and Threads

- A **process** has a self-contained execution environment.
  - Each process has its own resources and memory space.
  - No sharing of resources and memory among processes.
  - JVM run in a single process.
- A **thread** is a light-weight **concurrent** unit of execution.
  - Threads exist within a process.
  - Every process has at least one thread.
  - Threads share the process's resources, including memory.

Outline

- Processes and threads
- The UI thread
- Custom *Views*
- Drawing on *Canvas*
- Drawing on *Surface View*
**Android Processes and Threads**

- By default, one application, one process
  - When an app is launched the first time, the Android system starts a new process for the app with a single thread – the main thread.
  - When an app starts, and there is an existing process for the app, the app starts in the existing process and uses the existing thread.
  - Idle processes may be killed by the system.
- By default, all components of the same app run in the same process and thread
  - An app may create additional threads

**The Main Thread**

- Also know as the UI thread
- Responsible for dispatching events to UI widgets
  - Including the drawing events
- All components and widgets are instantiated in the UI thread
- All system callbacks are dispatched from the UI thread
  - Callbacks are executed on the UI thread.

**The Single Thread Model**

- The Android UI toolkit is not thread-safe
  - Less overhead, more responsive
  - All UI operations must run on the UI thread
- Performing long operations on the UI thread will block the UI thread
  - UI events are not dispatched
  - UI becomes non-responsive
- If the UI thread is blocked for 5s, the “Application Not Responding” (ANR) dialog will pop up
  - User may kill the app

**The UI Thread & Message Queue**

- Handler
- Message
- Runnable
- Message Queue
- Looper
  - An infinite loop
  - The UI thread
Looper and Handler

- The UI thread runs a `Looper` object
- A `Looper` is associated with one thread and a `Message Queue`
  - A `Message Queue` holds `Messages` and `Runnables` to be dispatched by the `Looper`
- Multiple `Handlers` can be associated with a `Message Queue`.
  - `Handlers` are responsible for handling (adding, removing, dispatching) `Messages` in the `Message Queue`.

Do Not Block the UI Thread

- Time consuming operations, e.g., network operations, should be performed on separate threads
  - Known as `background` or `worker` threads
- Options:
  - Directly create threads
  - Using `Async Task`

The UI Thread Rules

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

Thread and Runnable

- The `Thread` class represents a thread.
- The `Runnable` interface represents a unit of code that can be executed
  - An abstraction of things that run!
  - A single method: `void run()`
- Two ways to create a `Thread` object
  - Create a `Thread` using a `Runnable` object
  - Extend the `Thread` class and override the `run()` method
Implement Runnable

- Define a `Runnable` class
  
  ```java
  public class MyRunnable implements Runnable {
      public void run() {
          Do the work.
      }
  }
  ```

- Start a thread with a `Runnable` object
  
  ```java
  (new Thread(new MyRunnable())).start();
  ```

Extend Thread

- Define a subclass of `Thread` class
  
  ```java
  public class MyThread extends Thread {
      public void run() {
          Do the work.
      }
  }
  ```

- Start a thread with a `Thread` object
  
  ```java
  (new MyThread()).start();
  ```

Execute Code in a Thread

- Call the `start()` method of `Thread`
- The `run()` method gets executed on a new thread
- The thread terminates with the `run()` method returns

Communicate with the UI Thread

- Android provides several ways to access the UI thread from other threads
- Through an `Activity`
  
  ```java
  void runOnUiThread(Runnable action)
  ```
- Run the `Runnable` on the UI thread.
- Run immediately if it is on the UI thread, otherwise add to the message queue
- Through a `View` object
  
  ```java
  boolean post(Runnable action)
  boolean postDelayed(Runnable action, long millis)
  ```
- Add the `Runnable` to the message queue
Communicate with the UI Thread

- Through a *Handler* object
- Each *Handler* object is associated with a single thread and that thread's message queue.
  - Bound to the thread in which it is created
- Enqueue *Runnables* and *Messages*
  ```java
  boolean post(Runnable action)
  boolean postAtTime(Runnable action, long when)
  boolean postDelayed(Runnable action, long millis)
  ```
  - Add the *Runnable* to the message queue

Progress Bar Demo

Progress Bar

- A widget to provide visual indication of progress in some operation.

Progress Bar Demo – The Activity `onCreate()`

```java
public class MyActivity extends Activity {
  private ProgressBar progressBar;
  private TextView textView;
  private int status = 0;

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

    progressBar = (ProgressBar) findViewById(R.id.progressbar);
    progressBar.setProgress(0);
    textView = (TextView) findViewById(R.id.loading);
    ...}
```

**Progress Bar Demo**

- Simulate Long Operation

```java
private int doWork() {
    try {
        Thread.currentThread().sleep(100);
    } catch (InterruptedException e) {
    }
    return ++status;
}
```

Sleep for 100 millisecond.

**Progress Bar Demo**

- Update the Progress (Wrong Way)

```java
public void start(View view) {
    status = 0;
    progressBar.setProgress(0);
    textView.setText("Loading ...");
    new Thread(new Runnable() {
        public void run() {
            while (status <= 100) {
                status = doWork();
                progressBar.setProgress(status);
                if (status == 100) {
                    textView.setText("Done");
                }
            }
        }
    }).start();
}
```

Simulate long operation.

**Progress Bar Demo**

- Update the Progress

```java
private Handler mHandler = new Handler();
public void start(View view) {
    status = 0;
    progressBar.setProgress(0);
    textView.setText("Loading ...");
    new Thread(new Runnable() {
        public void run() {
            while (status <= 100) {
                status = doWork();
                mHandler.post(new Runnable() {
                    public void run() {
                        progressBar.setProgress(status);
                        if (status == 100) textView.setText("Done");
                    }
                });
            }
        }
    }).start();
}
```

A Handler associated with the UI thread.

Deliver a Runnable object to the Message Queue of the UI thread.
Perform Asynchronous Task

- `AsyncTask` class for performing asynchronous work for the UI thread
  - Perform blocking operations in a background thread
  - Publish results on the UI thread
  - Without dealing with threads and/or handlers directly.
- Proper and easy use of the UI thread and background
- Subclass `AsyncTask` and implement its callbacks

Async Task Class

- The `AsyncTask` class has three generic types
  `AsyncTask<Params, Progress, Result>`
- The generic types are
  - `Params` - The type of the input parameters sent to the task
  - `Progress` - The type of the progress unit for reporting during execution
  - `Result` - The type of the result of the task
- All three types must be specified
  - If not used, use the type `Void`
  - An object type with no valid values other than `null`. Not instantiatable. Not to be confused with `void`. Used as a place holder.

Async Task Callbacks

- `Result doInBackground(Params...)` - Runs in a pool of background threads.
  - May call `publishProgress(Progress...)` to publish the progress
- `void onPreExecute()` - Update the UI before and after the task. Invoked on the UI thread.
- `void onPostExecute(Result)` - Update the UI after during the execution of the task, after a call to `publishProgress()`. Invoked on the UI thread.

ProgressBar with AsyncTask – Extend AsyncTask

```java
public class MyActivity extends Activity {
    private ProgressBar progressBar;
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) { ... }

    private class LongTask extends AsyncTask<Void, Integer, Void> {
        private void doWork() { ... }

        @Override
        protected Void doInBackground(Void... params) { }

        @Override
        protected void onPreExecute() { ... }

        @Override
        protected void onPostExecute(Void result) { ... }
    }
}
```
Progress Bar with AsyncTask
– The Background Work

```java
private class LongTask extends AsyncTask<Void, Integer, Void> {
    private int status = 0;
    @Override protected Void doInBackground(Void... params) {
        status = 0;
        while (status <= 100) {
            doWork();
            publishProgress(status);
        }
        return null;
    }
    private void doWork() {
        ++status;
    }
}
```

Runs on a background thread.

Need to update UI.

Simulate a long operation.

---

Progress Bar with AsyncTask
– Start the Async Task

```java
public class MyActivity extends Activity {
    private ProgressBar progressBar;
    private TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) { … }
    public void start(View view) {
        new LongTask().execute();
    }
}
```

The onClick action for the “Start” Button.

Async Task Execution

From your activity

The UI thread

A background thread

execute()

preExecute()

doInBackground()

publishProgress()

onProgressUpdate()

postExecute()
2D Graphics Drawing

Drawing on the UI

- To draw on a View object, you need to
  - Define a custom View class, i.e., extend the View class or a subclass of View
    - Use a custom View in layout
  - Override the onDraw() method
    - Use Canvas, Paint, Drawable, and Shape etc. to draw stuff

Define a Custom View Class

- Extend the View class or one of its subclasses
  - Should provide at least two constructors

```java
public class MyView extends View {
    public MyView(Context context) {
        super(context); ...
    }
    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs); ...
        ...
    }
    ...
}
```

For instantiating MyView programmatically.

For instantiating MyView in XML layout resources.

Attributes of the XML element. Allow user-defined attributes

Custom Drawing

- Override the onDraw(Canvas) method
  - A callback method executed on the UI thread
  - Do not call the method directly
  - The Canvas class defines methods for drawing text, shapes, bitmaps, and other graphics primitives.
- Need to handle the layout
  - To discover the size, override onSizeChanged()
  - For greater control, to interact with the Layout Manager, override onMeasure()
**Canvas**

- The `Canvas` class defines methods for drawing
- Fill canvas with color: `drawColor(color)`  
  - Color is int.
- Draw text: `drawText(text, x, y, paint)`
- Draw bitmap: `drawBitmap(bitmap, x, y, paint)`
- Draw shapes:
  - `drawRect(rect, paint)`
  - `drawOval(rect, paint)`
  - `drawCircle(x, y, radius, paint)`
  - `drawLine(x1, y1, x2, y2, paint)`
- Draw path: `drawPath(path, paint)`

**Canvas Coordinate System**

- The origin (0, 0) is at the top-left corner.
- X-axis points right
- Y-axis points down (sometimes known as the inverted coordinate system)
- Unit: `pixels`

```
(0, 0)  x
```

```
(0, 0)  y
```

```
(width, height)
```

**Paint**

- The `Paint` class holds the style information for drawing
  - Color
  - Line join/cap style
  - Font, text style, text size, alignment, etc.
- Most drawing methods takes a `Paint` object as an argument

**Trigger Redrawing**

- When the `View` object becomes visible.
- When the `invalidate()` method is called on the `View` object.
  - Never call `onDraw()` directly. System controls redrawing.
- Simple animation can be accomplished by calling `invalidate()` in short intervals
- `onDraw()` may be called frequently (up to 60 fps)
  - Optimize the implementation makes a big difference in performance.
Bouncing Ball App

- A simple animation app with a ball bouncing off four sides of the view.
- A custom View
- A layout using the custom View
- Implement `onDraw()`
- Simple animation

The Custom View Class

```java
public class MyView extends View {
    private int x, y, dx = 5, dy = 5, r = 30;
    private int width, height;
    public MyView(Context context) {
        super(context);
    }
    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        width = w;
        height = h;
        x = width / 2;
        y = height / 2;
    }
    private Paint paint = new Paint();
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);
        paint.setColor(Color.BLUE);
        if (x + r >= width || x - r < 0) dx = -dx;
        if (y + r >= height || y - r < 0) dy = -dy;
        x += dx;
        y += dy;
        canvas.drawCircle(x, y, r, paint);
    }
}
```

A shared `Paint` object.
Fill the `Canvas` background.
Adjust the ball position.
Draw the ball.

But the ball is not moving!
### The Custom View Class

**– Prepare a Runnable**

```java
public class MyView extends View {
    private Runnable update = new Runnable() {
        @Override
        public void run() {
            invalidate();
        }
    };
    ..
}
```

A `Runnable` object defined as an instance of an anonymous inner class. Invalidate this view. When this `Runnable` object is placed in the Message Queue of the UI thread, the view will be redrawn.

### The Custom View Class

**– Animation**

```java
public class MyView extends View {
    private int x, y, dx = 5, dy = 5, r = 30;
    private int width, height;
    private Paint paint = new Paint();
    private boolean paused;
    private Handler mHandler = new Handler();

    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);
        paint.setColor(Color.BLUE);
        if (x + r >= width || x - r < 0) dx = -dx;
        if (y + r >= height || y - r < 0) dy = -dy;
        x += dx;
        y += dy;
        canvas.drawCircle(x, y, r, paint);
        if (!paused) mHandler.postDelayed(update, 15);
    }
    ..
}
```

An `Handler` object associated with the UI thread. Post the update object to the UI thread after 15ms delay.

### The Custom View Class

**– Start and Stop the Animation**

```java
public class MyView extends View {
    private boolean paused;
    private long frameCount = 0;
    private long timeStart = 0;

    public void startAnimation() {
        paused = false;
        frameCount = 0;
        timeStart = System.currentTimeMillis();
        mHandler.post(update);
    }
    public void stopAnimation() {
        paused = true;
    }
}
```

Post the initial update object when the animation starts.

### The Custom View Class

**– Display FPS Stats**

```java
protected void onDraw(Canvas canvas) {
    frameCount+=;
    long timeNow = System.currentTimeMillis();
    long elapsedTime = timeNow - timeStart;
    float fps = (float) frameCount / elapsedTime * 1000L;
    canvas.drawText("Frame count=" + frameCount + " Elapsed time=" + elapsedTime + " FPS=" + fps, 20, 40, paint);
    if (!paused) mHandler.postDelayed(update, 15);
    canvas.drawText("FPS=" + fps, 20, 40, paint);
}
```

For calculating frame rate. See next slide. Draw background and the ball. Draw the FPS stats.
Layout Using Custom View

Fully qualified name for the custom View class.

All the attributes defined in the View class are available.

Spinning Text App

- Another simple animation app
- Drawing text
- Using colors and fonts
- Canvas coordinate transformations
- Similar design to the Bouncing Ball app
  - Main difference is in `onDraw()`

Bouncing Ball App – The Activity

```java
public class MyActivity extends Activity {
    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
    }
    @Override protected void onResume() {
        super.onResume();
        MyView v1 = (MyView) findViewById(R.id.v1);
        v1.startAnimation();
    }
    @Override protected void onPause() {
        super.onPause();
        MyView v1 = (MyView) findViewById(R.id.v1);
        v1.stopAnimation();
    }
    ...
}
```

Spinning Text App

- Another simple animation app
- Drawing text
- Using colors and fonts
- Canvas coordinate transformations
- Similar design to the Bouncing Ball app
  - Main difference is in `onDraw()`
Coordinate Transformations

- The Canvas class supports methods for transforming the coordinate system used for drawing.
- **Translate**
  - `translate(dx, dy)`
- **Rotate**
  - `rotate(degrees)`
  - Rotate the coordinate system at the origin by a specified angle (0 – 360)
- **Scale**
  - `scale(sx, sy)`
  - Scale the view by the specified multiplier in each direction.
    - Useful to scale the unit from px to dp.

Spinning Text App – The Custom View Class

```java
public class MyView extends View {
    private int rotation = 0, dr = 5;
    private int width, height;
    private boolean paused;
    private Paint paint = new Paint();
    private int color = Color.BLUE;
    private Typeface typeface = Typeface.DEFAULT;
    private Handler mHandler = new Handler();

    public MyView(Context context) { ... }
    public MyView(Context context, AttributeSet attrs) { ... }
    public void startAnimation() { ... }
    public void stopAnimation() { ... }
    private Runnable update = new Runnable() { ... }
    protected void onSizeChanged(int w, int h, int oldw, int oldh) { ... }
    protected void onDraw(Canvas canvas) { ... }
}
```

The rotation of the text, and the velocity of change.
The width and height of the view.
Similar to Bouncing Ball.
The Custom View Class
– Some Utilities for Style

```java
public class MyView extends View {
    @Override
    private static int getRandomColor() {
        Random random = new Random();
        return Color.rgb(random.nextInt(256), random.nextInt(256),
                         random.nextInt(256));
    }
    private static Typeface getRandomTypeface() {
        Random random = new Random();
        return Typeface.create(TYPFACE[random.nextInt(4)],
                                STYLE[random.nextInt(4)]);
    }
    static final Typeface[] TYPFACE = {
        Typeface.DEFAULT, Typeface.MONOSPACE,
        Typeface.SERIF, Typeface.SANS Serif};
    static final int[] STYLE = {
        Typeface.NORMAL, Typeface.BOLD,
        Typeface.ITALIC, Typeface.BOLD_ITALIC};
    }
```

The Custom View onDraw()
– Adjusting the Rotation & Style

```java
@Override protected void onDraw(Canvas canvas) {
    rotation += dr;
    if (rotation >= 360) {
        rotation %= 360;
        color = getRandomColor();
        typeface = getRandomTypeface();
    }
}
```

Generate a random color with randomly selected RGB values (0 – 255).

Set the style for drawing stats. Text is left aligned.

The Custom View onDraw()
– Drawing the FPS Stats

```java
@Override protected void onDraw(Canvas canvas) {
    frameCount++;
    long timeNow = System.currentTimeMillis();
    long elapsedTime = timeNow - timeStart;
    float fps = (float) frameCount / elapsedTime * 1000L;
    canvas.drawColor(Color.WHITE);
    paint.setAntiAlias(true);
    paint.setColor(Color.BLACK);
    paint.setTextSize(20);
    paint.setTypeface(Typeface.DEFAULT);
    paint.setPaintWith = Paint.Align.CENTER;
    canvas.drawText("Frame count:" + frameCount +
                    " Elapsed time:" + elapsedTime +
                    " FPS=" + fps, 20, 40, paint);
}
```

Set the style for drawing spinning text. Text is left aligned.

Translate the origin to the center, then rotate.

Draw text at the origin with center alignment.
Bouncing Objects App

An extension of the Bouncing Ball app

Many bouncing objects, with random colors, sizes and velocities

Drawing several types of objects

- Shape Drawable
- Bitmap

Shape Drawable

- A Drawable object that draws primitive shapes.
  - Has a Shape object
  - Manages its presence on the screen
  - Bounds: its location & size
  - Paint: its style
- A Shape object can be
  - Rect Shape
  - Oval Shape
  - Round Rect Shape
  - Path Shape

Bouncing Objects – Custom View Overview

```java
public class MyView extends View {
    private int width, height;
    private boolean paused;

    class MyShape { … }
    class MyBitmap extends MyShape { … }
    List<MyShape> shapes = new ArrayList<MyShape>();

    public MyView(Context context) { … }
    public MyView(Context context, AttributeSet attrs) { … }
    public void startAnimation() { … }
    public void stopAnimation() { … }
    private Runnable update = new Runnable() { … }
    protected void onSizeChanged(int neww, int newh, int oldw, int oldh) { … }
    protected void onDraw(Canvas canvas) { … }
}
```
The Inner Classes – The Design

- Design goals:
  - Keep a list of different objects we can draw on canvas
  - Each object manages its size, position, style, movement, and drawing
  - Simple uniform way to deal with all types of objects
- The Strategy design pattern
- Two inner classes representing different types of shape objects
  - MyShape: different types of Shape Drawables
  - MyBitmap: bitmap images. Subclass of MyShape

Inner Class – MyShape

class MyShape {
  ShapeDrawable drawable;
  int dx = 5, dy = 5;
  MyShape(Shape shape) {
    drawable = new ShapeDrawable(shape);
  }
  MyShape() {};
  void move() {
    Rect bounds = drawable.getBounds();
    if (bounds.right >= width || bounds.left < 0) dx = -dx;
    if (bounds.bottom >= height || bounds.top < 0) dy = -dy;
    bounds.left += dx;
    bounds.right += dx;
    bounds.top += dy;
    bounds.bottom += dy;
  }
  ...}

Inner Class – MyBitmap

class MyBitmap extends MyShape {
  Bitmap bitmap;
  int x, y;
  MyBitmap(int resId) {
    bitmap = BitmapFactory.decodeResource(getResources(), resId);
  }
  void setBounds(int left, int top, int right, int bottom) {
    x = left; y = top;
  }
  void move() {
    int w = bitmap.getWidth();
    int h = bitmap.getHeight();
    if (x + w >= width || x < 0) dx = -dx;
    if (y + h >= height || y < 0) dy = -dy;
    x += dx; y += dy;
  }
  void draw(Canvas canvas) {
    canvas.drawBitmap(bitmap, x, y, null);
  }
}
public class MyView extends View {
    public MyView(Context context) {
        super(context);
        initShapes();
    }
    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initShapes();
    }
    private List<MyShape> shapes = new ArrayList<MyShape>();
    private Handler mHandler = new Handler();
    private Random random = new Random();
    private Paint paint = new Paint();
    ...}

    @Override
    protected void onSizeChanged(int newW, int newH, int oldW, int oldH) {
        super.onSizeChanged(newW, newH, oldW, oldH);
        width = newW;
        height = newH;
        positionShapes();
    }
    private void positionShapes() {
        for (MyShape s : shapes) {
            int w = random.nextInt(50) + 30;
            int h = random.nextInt(50) + 30;
            int x = random.nextInt(width - 2 * w) + w;
            int y = random.nextInt(height - 2 * h) + h;
            s.setColor(getRandomColor());
            s.setBounds(x, y, x + w, y + h);
            s.setVelocity(random.nextInt(5) + 2, random.nextInt(5) + 2);
        }
    }

    private Random random = new Random();
    private static int getRandomColor() {
        Random random = new Random();
        return Color.argb(random.nextInt(256),
                          random.nextInt(256),
                          random.nextInt(256),
                          random.nextInt(256));
    }
}
Bouncing Objects Custom View – Drawing Shapes

```java
@Override
protected void onDraw(Canvas canvas) {
    frameCount++;
    long timeNow = System.currentTimeMillis();
    long elapsedTime = timeNow - timeStart;
    float fps = (float) frameCount / elapsedTime * 1000L;
    canvas.drawRectColor(Color.WHITE);
    for (MyShape shape : shapes) {
        shape.move();
        shape.draw(canvas);
    }
    paint.setColor(Color.BLACK);
    paint.setTextSize(20);
    canvas.drawText("Frame count=" + frameCount + "    Elapsed time=" + elapsedTime + "    FPS=" + fps, 20, 40, paint);
    if (!paused) mHandler.postDelayed(update, 15);
}
```

Handle a list of different types of objects in a uniform way.

Bouncing Objects – The Layout

```xml
<LinearLayout ...
    <edu.depaul.csc472.bounncingobjects.MyView
        android:id="@+id/v1"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"/>
</LinearLayout>
```

Bouncing Objects – The Activity

```java
public class MyActivity extends Activity {
    MyView animView;
    Spinner spinner;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
        spinner = (Spinner) findViewById(R.id.num);
        animView = (MyView) findViewById(R.id.v1);
        spinner.setAdapter(new ArrayAdapter<Integer>(this, android.R.layout.simple_spinner_item, NUMBERS));
    }
    
    static final Integer[] NUMBERS = {
        5, 10, 15, 20, 25, 30, 35, 40, 45, 50
    };
}
```

Called from the Activity.

static final Integer[] NUMBERS = {
    5, 10, 15, 20, 25, 30, 35, 40, 45, 50
};
Bouncing Objects – The Activity

```java
public class MyActivity extends Activity {
    MyView animView;
    Spinner spinner;
    ...
    @Override protected void onResume() {
        super.onResume();
        animView.startAnimation();
    }
    @Override protected void onPause() {
        super.onPause();
        animView.stopAnimation();
    }
    public void restart(View view) {
        animView.restart((Integer) spinner.getSelectedItem());
    }
}
```

Bouncing Objects with Surface View

- An alternative version of the Bouncing Objects app using Surface View
- Identical objects and behaviors, different design
- Design and implementation techniques
  - Surface View callbacks
  - Use your own rendering thread
  - Synchronization among different threads

Drawing in Surface View

Surface View

- A subclass of View
  - Can reside in the view hierarchy, which controls its position
- Provides a dedicated drawing surface embedded inside of a view hierarchy.
  - Dedicated buffer, separate from the buffer shared by all other views
  - Consumes more memory
- Recommended for fast-moving applications, such as games, videos, or camera previews etc.
Why Surface View?

- The rendering can be done by a background thread (non-UI thread)
  - Better control of the timing of the updates. (up to 60 fps)
  - More regular updates. Smoother animation.
- Normal view updates through the UI thread
  - Call `invalidate()`
  - The `onDraw()` method will be called on the UI thread
    - Not immediately. The UI thread is responsible for many things.
    - All updates appear after the next VSYNC cycle.

Using Surface View

- The underlying drawing surface is accessed via a `Surface Holder` object
  - Obtained by calling `getHolder()` on the `Surface View` object
- A custom `Surface View` must implement the `Surface Holder Callback` interface
  - To be notified when the underlying surface is created, changed, and destroyed.

Retrieve the Canvas

- A `Canvas` object associated with the drawing surface can be retrieved from the `Surface Holder` object
- You must do the following when you draw each frame
  1. `Canvas c = holder.lockCanvas();`
  2. Do the drawing on canvas
  3. `holder.unlockCanvasAndPost(c);`
- Must draw the whole canvas each frame
- The frame will be visible after unlock the canvas.

A Dedicated Rendering Thread

- You may use your own dedicated rendering thread to do the drawing
  - Call your drawing method on the rendering thread directly
  - Do not use `invalidate()` and `onDraw()`
- The rendering thread can only touch the drawing surface between the time the surface is created and destroyed
  - Notified through the callbacks
Synchronization Among Threads

- *Surface Holder* callbacks are invoked on the UI thread
- Drawing is done on the rendering thread
- Must prevent interfering operations on the same object from different threads
  - Modifying an object on one thread, while it is used in another thread will cause a crash.
- Must properly synchronize any object that is touched by the rendering thread and the UI thread

Synchronized Statement

- Java provides a *synchronized statement*

```
  synchronized (object) {
    statement
  }
```
- Execute a synchronized statement
  - The *statement* will be executed after an exclusive lock associated with the *object* has been obtained.
  - The lock is retained during the execution of the *statement*.
  - The lock is released after the execution is complete.
- No two threads can simultaneously execute statements synchronized on the *same object*.

Custom Surface View

- **Overview**
  ```java
  public class MyView extends SurfaceView implements SurfaceHolder.Callback {
    class MyShape { ... }
    class MyBitmap extends MyShape { ... }
    List<MyShape> shapes = new ArrayList<MyShape>();
    public void surfaceCreated(SurfaceHolder surfaceHolder) {... }
    public void surfaceChanged(SurfaceHolder surfaceHolder, int format, int width, int height) {... }
    public void surfaceDestroyed(SurfaceHolder surfaceHolder) {... }
    protected void doDraw(Canvas canvas) {... }
  }
  ```

  - Implement the *Surface Holder* callbacks.
  - Our drawing method, *Not onDraw()*

- **Constructors**
  ```java
  public class MyView extends SurfaceView implements SurfaceHolder.Callback {
    private boolean surfaceAvailable;
    private SurfaceHolder holder;
    public MyView(Context context) {
      super(context);
      holder = getHolder();
      holder.addCallback(this);
      initShapes(5);
    }
    public MyView(Context context, AttributeSet attrs) {
      super(context, attrs);
      holder = getHolder();
      holder.addCallback(this);
      initShapes(5);
    }
  }
  ```

  - Set itself as a listener to the callbacks from the holder.
Custom Surface View
– Surface Holder Callbacks

```java
public class MyView extends SurfaceView implements SurfaceHolder.Callback {
    private boolean surfaceAvailable;
    private SurfaceHolder holder;

    @Override
    public void surfaceCreated(SurfaceHolder surfaceHolder) {
        surfaceAvailable = true;
        startAnimation();
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder surfaceHolder) {
        surfaceAvailable = false;
        stopAnimation();
    }

    ...}
```

When the surface is available for drawing

```java
public class MyView extends SurfaceView implements SurfaceHolder.Callback {
    private int width, height;
    private boolean surfaceAvailable;
    private SurfaceHolder holder;

    @Override
    public void surfaceChanged(SurfaceHolder surfaceHolder, int format, int width, int height) {
        this.width = width; this.height = height;
        synchronized (holder) {
            positionShapes();
        }
        startAnimation();
    }

    ...}
```

The width and height of the drawing surface.

Modifies the shape list. All operations involving reading/writing the list are synchronized on the holder.

Custom Surface View
– Do Drawing

```java
@Override protected void doDraw(Canvas canvas) {
    frameCount++;
    long timeNow = System.currentTimeMillis();
    long elapsedTime = timeNow - startTime;
    float fps = (float) frameCount / elapsedTime * 1000L;
    canvas.drawColor(Color.WHITE);
    for (MyShape shape : shapes) {
        shape.move();
        shape.draw(canvas);
    }
    paint.setColor(Color.BLACK);
    paint.setTextSize(20);
    canvas.drawText("Frame count=", + frameCount + ", Elapsed time=", + elapsedTime + " FPS=" + fps, 20, 40, paint);
    if (!paused) mHandler.postDelayed(update, 15);
}
```

Same drawing operations.

Custom Surface View
– The Rendering Thread

```java
private void startRenderingThread() {
    new Thread(new Runnable() {
        public void run() {
            while (!done) {
                Canvas c = null;
                try {
                    c = holder.lockCanvas();
                    synchronized (holder) {
                        doDraw(c);
                    }
                } finally {
                    if (c != null) holder.unlockCanvasAndPost(c);
                }
            }
        }
    }).start();
```

Modifies the shape list. All operations involving reading/writing the list are synchronized on the holder.
Custom Surface View
– Start & Stop Animation

```java
public void startAnimation() {
    done = false;
    if (surfaceAvailable) startRenderingThread();
}
public void stopAnimation() {
    done = true;
}
public void restart(int n) {
    stopAnimation();
    synchronized (holder) {
        initShapes(n);
        positionShapes();
    }
    startAnimation();
}
```

The Sample Code

- The sample apps in this lecture are available in D2L
  - ProgressBarDemo.zip
  - ProgressBarAsync.zip
  - BouncingBall.zip
  - SpinningText.zip
  - BouncingObjects.zip
  - BouncingObjects-Surface.zip
- Each zip archive contains the entire project folder
- Unzip the file and import to Android Studio

Next …

- Touch events
- Gestures
- Multi-touch gestures

* Android is a trademark of Google Inc.
* Some contents and diagrams are from Google Inc.
  Licensed under Apache 2.0 and Creative Commons Attribution 2.5.