CSC 472 / 372 Mobile Application Development for Android

Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

Outline

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

Threads in Android

Processes and Threads

- A *process* is a self-contained *concurrent* 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 thread.
  - Threads share the process's resources, including memory.

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 use 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

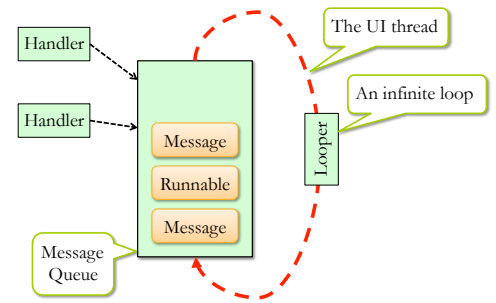
The Main Thread

- 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



The Looper and the 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*.

The UI Thread Rules

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

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*

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

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

- Start a thread with a *Runnable* object

```
(new Thread(new MyRunnable())).start();
```


Extend Thread

- Define a subclass of *Thread* class

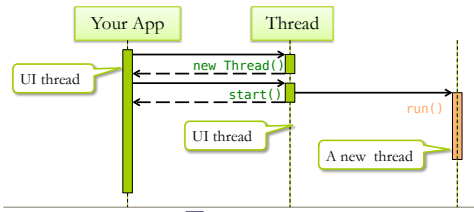
```
public class MyThread extends Thread {
  public void run() {
    Do the work.
  }
}
```

- Start a thread with a *Thread* object

```
(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*

```
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

```
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*

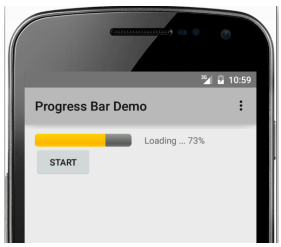
```
boolean post(Runnable action)
boolean postAtTime(Runnable action, long when)
boolean postDelayed(Runnable action, long millis)
```

  - Add the *Runnable* to the message queue

Progress Bar Demo

Progress Bar

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



DePaul University 22

Progress Bar Demo – The Main Activity onCreate()

```
public class MainActivity 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_main);

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

DePaul University 23

Progress Bar Demo – Simulate a Long Operation

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

Sleep for 100 millisecond.

DePaul University 25

Progress Bar Demo – Update the Progress (Wrong Way)

```
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("Loading ... " + status + "%");
        } else {
          textView.setText("Done");
        }
      }
    }
  }).start();
}
```

The onClick action for the "Start" *Button*.

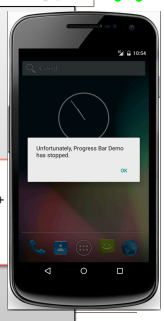
Simulate long operation.

Update the progress.

29

Progress Bar Demo – Update the Progress (Wrong Way)

```
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("Loading ... " + status +
        } else {
          textView.setText("Done");
        }
      }
    }
  }).start();
}
```



30

Progress Bar Demo – Update the Progress

```
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("Loading ... " + status + "%");
            else
              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

DePaul University 34

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`

Must be class or interface types

An *object* type with no valid values other than `null`. Not instantiatable. Not to be confused with `void`. Used as a place holder.

DePaul University 38

Async Task Callbacks

`Result doInBackground(Params...)`

Variable arguments. Accessed as an array.

- **Runs in a pool of background threads.**
- May call `publishProgress(Progress...)` to publish the progress

`void onPreExecute()`

`void onPostExecute(Result)`

- Update the UI before and after the task. Invoked on the UI thread.

`void onProgressUpdate(Progress...)`

- Update the UI after during the execution of the task, after a call to `publishProgress()`. Invoked on the UI thread.

DePaul University 40

Progress Bar with AsyncTask – Extend Async Task

```
public class MainActivity 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 onProgressUpdate(Integer... progress) { … }
    @Override protected void onPreExecute() { … }
    @Override protected void onPostExecute(Void result) { … }
  }
  …
}
```

DePaul University 41

Progress Bar with AsyncTask – The Background Work

```
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() {
    Do the work.
    ++status;
  }
  …
}
```

Runs on a background thread.

Need to update UI.

Simulate a long operation.

Progress Bar with AsyncTask

```
private class LongTask extends AsyncTask<Void, Integer, Void> {
  …
  @Override protected Void doInBackground(Void... params) {
    …
    publishProgress(status);
    …
  }
  @Override protected void onProgressUpdate(Integer... progress) {
    progressBar.setProgress(progress[0]);
    textView.setText("Loading ... " + status
  }
  @Override protected void onPreExecute() {
    textView.setText("Loading ...");
  }
  @Override protected void onPostExecute(Void result) {
    textView.setText("Done");
  }
}
```

Runs on a background thread.

Runs on the UI thread.

Runs on the UI thread.

Runs on the UI thread.

Progress Bar with AsyncTask – Start the Async Task

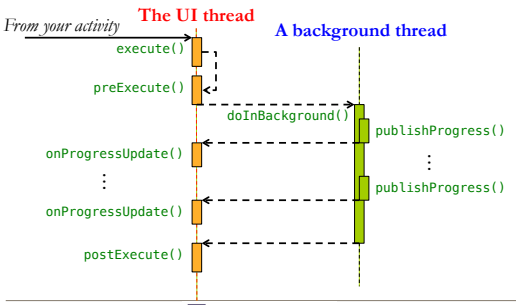
```
public class MainActivity extends Activity {
  private ProgressBar progressBar;
  private TextView textView;
  @Override
  protected void onCreate(Bundle savedInstanceState) { … }

  public void start(View view) {
    new LongTask().execute();
  }

  private class LongTask extends AsyncTask<Void, Integer, Void> {
    …
  }
  …
}
```

The onClick action for the "Start" *Button*.

Async Task Execution



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

```
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()`

Next topic

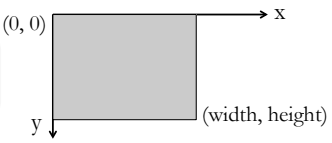
Canvas

- The *Canvas* class defines the methods for drawing
- Fill canvas with color: `drawColor(color)` — Color is an 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*

Not *dp*. May need to adjust for screen density



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 take a *Paint* object as an argument

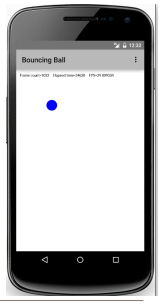
Trigger Redrawing

- 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)
  - Optimizing the implementation makes a big difference in performance.

Bouncing Ball App

Bouncing Ball App

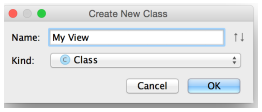
- A simple animation app with a ball bouncing off four sides of the view.
- A custom View
- A layout using the custom View
- Implement `onDraw()`
- Simple animation



Adding a Custom View Class

- Start with a *Blank Activity*
- Add a custom view class
- From the menu bar

  File | New | Java Class
  - Name: *MyView*
  - Kind: *Class*
- A custom *View* class extends the `android.viewView` class, i.e., a subclass of *View*



The Custom View Class

```
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) {
    super.onSizeChanged(w, h, oldw, oldh);
    width = w; height = h;
    x = width / 2; y = height / 2;
  }
  …
}
```

The position and radius of the ball, and the velocity of change.

The width and height of the view

The constructors.

Discover the width and height of the view. Start the ball in the center.

The Custom View Class – onDraw()

```
public class MyView extends View {
  private int x, y, dx = 5, dy = 5, r = 30;
  private int width, height;
  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

```
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

```
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();
  @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);
    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

```
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;
  }
}
```

For calculating frame rate. See next slide.

Post the initial update object when the animation starts.

The Custom View Class – Display FPS Stats

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

DePaul University 92

Layout Using Custom View

Fully qualified name for the custom *View* class.

```
<LinearLayout … >
  <edu.depaul.csc472.bounncingball.MyView
    android:id="@+id/v1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</LinearLayout>
```

All the attributes defined in the *View* class are available.

DePaul University 95

Bouncing Ball App – The Main Activity

```
public class MainActivity extends Activity {
  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
  @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();
  }
  …
}
```

DePaul University 96

Spinning Text App

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()`

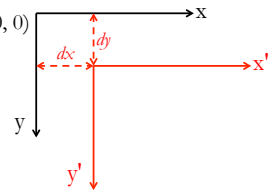
DePaul University 98

Coordinate Transformations

- The Canvas class supports methods for transforming the coordinate system used for drawing
- Translate
  - `translate(dx, dy)`
- Shift the origin by the specified amount

$x = x' + dx$

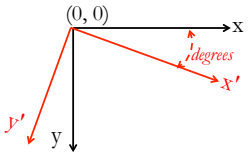
$y = y' + dy$



DePaul University 99

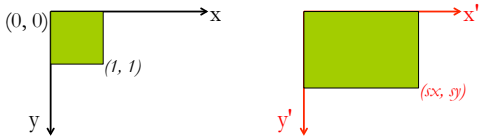
Coordinate Transformations

- Rotate
  - `rotate(degrees)`
- Rotate the coordinate system at the origin by a specified angle, in degrees, clockwise:
  0° – 360°



Coordinate Transformations

- 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

```
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

```
public class MyView extends View {
  …
  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(TYPEFACE[random.nextInt(4)],
                           STYLE[random.nextInt(4)]);
  }
  static final Typeface[] TYPEFACE = { Typeface.DEFAULT,
    Typeface.MONOSPACE, Typeface.SERIF, Typeface.SANS_SERIF };
  static final int[] STYLE = { Typeface.NORMAL, Typeface.BOLD,
    Typeface.ITALIC, Typeface.BOLD_ITALIC };
  …
}
```

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

The Custom View onDraw() – Drawing the FPS Stats

```
@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.setTextAlign(Paint.Align.LEFT);
  canvas.drawText("Frame count=" + frameCount +
          "    Elapsed time=" + elapsedTime +
          "    FPS=" + fps, 20, 40, paint);
  …
```

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

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

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

Adjust the rotation

If it has completed a full rotation (360°), change the text color and typeface.

The Custom View onDraw() – Drawing the Spinning Text

```
@Override protected void onDraw(Canvas canvas) {
  …
  paint.setColor(color);
  paint.setTextSize(50);
  paint.setTypeface(typeface);
  paint.setTextAlign(Paint.Align.CENTER);

  canvas.translate(width/2, height/2);
  canvas.rotate(rotation);
  canvas.drawText("Android 6.0 Marshmallow", 0, 0, paint);

  if (!paused) mHandler.postDelayed(update, 15);
}
```

Set the style for drawing spinning text. *Text is left aligned, by default.*

Translate the origin to the center, then rotate.

Draw text at the origin with center alignment

Bouncing Objects App

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 simple shapes.
  - Has a *Shape* object
  - Manages its presence on the screen
    - Bounds: its location & size
    - Paint: its style
- A *Shape* object can be one of
  - *Rect Shape*
  - *Oval Shape*
  - *Round Rect Shape*
  - *Path Shape*

Bouncing Objects – Custom View Overview

```
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) { … }
  …
}
```

Two inner classes representing various types of objects to draw

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;
  }
  …
```

The *Shape Drawable* object.

The velocity of movement.

A constructor for subclasses.

Move the shape by adjusting its bounds. Handle the bouncing behavior.

Inner Class – MyShape

```
class MyShape {
  …
  void setBounds(int left, int top, int right, int bottom) {
    drawable.setBounds(left, top, right, bottom);
  }
  void setVelocity(int dx, int dy) {
    this.dx = dx;
    this.dy = dy;
  }
  void setColor(int color) {
    if (drawable != null) drawable.getPaint().setColor(color);
  }
  void draw(Canvas canvas) {
    drawable.draw(canvas);
  }
}
```

Draw the shape.

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);
  }
}
```

The *Bitmap* object and its position.

Move the bitmap by adjusting its position. Handle the bouncing behavior.

Draw the bitmap.

Bouncing Objects Custom View – Constructors & List of Shapes

```
public class MyView extends View {
  public MyView(Context context) {
    super(context);
    initShapes(5);
  }
  public MyView(Context context, AttributeSet attrs) {
    super(context, attrs);
    initShapes(5);
  }
  private List<MyShape> shapes = new ArrayList<MyShape>();
  private Handler mHandler = new Handler();
  private Random random = new Random();
  private Paint paint = new Paint();
  …
}
```

A list of different types of objects .

Bouncing Objects Custom View – Initialize the Shape List

```
public class MyView extends View {
  …
  private void initShapes(int n) {
    float[] outR = new float[] {6,6,6,6,6,6,6,6};
    shapes.clear();
    for (int i = 0; i < n; i++) {
      switch (i % 8) {
        case 0: shapes.add(new MyShape(new RectShape())); break;
        case 1: shapes.add(new MyShape(new OvalShape())); break;
        case 2: shapes.add(new MyShape(
                  new RoundRectShape(outR, null, null))); break;
        case 3: shapes.add(new MyBitmap(R.drawable.soccer)); break;
        …
      }
    }
  }
```

Create different shape objects and add to the list.

Bouncing Objects Custom View – Set Initial Positions

```
@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);
  }
}
```

Set random position, size, color, and velocity to each shape object.

Bouncing Objects Custom View – Some Utilities

```
private Runnable update = new Runnable() {
  @Override public void run() {
    invalidate();
  }
};

private static int getRandomColor() {
  Random random = new Random();
  return Color.argb(random.nextInt(230) + 26,
                    random.nextInt(256),
                    random.nextInt(256),
                    random.nextInt(256));
}
```

Generate a random color with a random transparency value.

Bouncing Objects Custom View – Drawing Shapes

```
@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);
  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 Custom View – Start and Stop Animation

```
public void startAnimation() {
  paused = false;
  frameCount = 0;
  timeStart = System.currentTimeMillis();
  mHandler.post(update);
}
public void stopAnimation() {
  paused = true;
}
public void restart(int n) {
  if (width > 0 && height > 0) {
    stopAnimation();
    initShapes(n);
    positionShapes();
    startAnimation();
  }
}
```

Called from the *Activity*.

Bouncing Objects – The Layout

```
<LinearLayout … >
  <edu.depaul.csc472.bounncingobjects.MyView
     android:id="@+id/v1"
     android:layout_width="match_parent"
     android:layout_height="0dp"
     android:layout_weight="1"/>
  <LinearLayout … >
    <TextView … android:text="# of objects: "/>
    <Spinner android:id="@+id/num"
       android:layout_width="wrap_content"
       android:layout_height=" wrap_content" />
  </LinearLayout>
</LinearLayout>
```

Bouncing Objects – The Main Activity

```
public class MainActivity extends Activity {
  MyView animView;
  Spinner spinner;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    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
  };
  …
```


Bouncing Objects – The Main Activity

```
public class MainActivity extends Activity {
  …
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    spinner = (Spinner) findViewById(R.id.num);
    …
    spinner.setOnItemSelectedListener(
      new AdapterView.OnItemSelectedListener() {
        @Override public void onItemSelected(AdapterView<?> parent,
                                View view, int position, long id) {
          restart();
        }
        @Override public void onNothingSelected(AdapterView<?> parent) {}
    });
  }
```

Bouncing Objects – The Main Activity

```
public class MainActivity 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() {
     animView.restart((Integer) spinner.getSelectedItem());
  }
}
```

Called from the *Listener*

Drawing in Surface View

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

Surface View

- A subclass of *View*
  - Can reside in the view hierarchy, which controls its size and 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

Trade-off: *performance* vs. *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 appears after the next VSYNC cycle.

Using Surface View

- The underlying drawing surface is accessed via a *Surface Holder* object
  - Obtained by calling `getHodler()` 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
}
```

The exclusive region.

- 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 the statements synchronized on the *same object*.

Custom Surface View – Overview

```
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) { … }
  …
}
```

Same as before.

Our drawing method. Not `onDraw()`

Implement the *Surface Holder* callbacks.

Custom Surface View – Constructors

```
public class MyView extends SurfaceView
                    implements SurfaceHolder.Callback {
  private boolean surfaceAvailble;
  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

```
public class MyView extends SurfaceView
                    implements SurfaceHolder.Callback {
  private boolean surfaceAvailble;
  private SurfaceHolder holder;
  @Override
  public void surfaceCreated(SurfaceHolder surfaceHolder) {
    surfaceAvailble = true;
    startAnimation();
  }
  @Override
  public void surfaceDestroyed(SurfaceHolder surfaceHolder) {
    surfaceAvailble = false;
    stopAnimation();
  }
  …
```

When the surface is available for drawing

Custom Surface View – Surface Holder Callbacks

```
public class MyView extends SurfaceView
                    implements SurfaceHolder.Callback {
  private int width, height;
  private boolean surfaceAvailble;
  private SurfaceHolder holder;
  …
  @Override
  public void surfaceChanged(SurfaceHolder surfaceHolder,
                             int format, int width, int height) {
    this.width = width; this.height = height;
    stopAnimation();
    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

```
@Override protected void doDraw(Canvas canvas) {
  frameCount++;
  long timeNow = System.currentTimeMillis();
  long elapsedTime = timeNow - timeStart;
  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

```
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.**

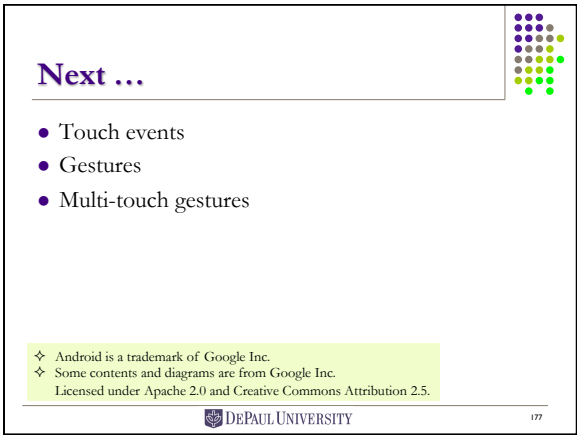
Custom Surface View – Start & Stop Animation

```
public void startAnimation() {
  done = false;
  if (surfaceAvailble) startRenderingThread();
}
public void stopAnimation() {
  done = true;
}
public void restart(int n) {
  if (width > 0 && height > 0) {
    stopAnimation();
    synchronized (holder) {
      initShapes(n);
      positionShapes();
    }
    startAnimation();
  }
}
```

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

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.
DePaul University
177