Inversion of Control: Callbacks & Frameworks

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Overview

- Inversion of Control
- Callbacks
- Frameworks

Program Flow

- So far:
  - Your code manages its own execution flow
  - You control method calls

- What if you cant?
  - Respond to external event
    - Currently: block on a read, etc. Requires a thread to sit idle. Threads aren't free…
- What if you don't want to?
  - Your code is plugged into a much larger framework

Inversion of Control

- Instead of you calling other code
- Other code calls you

- External events:
  - calls your code to handle the event
- Larger frameworks:
  - Call your code to provide functionality

Pattern: Callback

- Your code needs some other component to do something for it.
- The other component is going to manage its own lifecycle.
- It's going to make calls to your code as it goes along… but from within its own execution context.
- Idea: Instead of *returning* a result, it calls one of your methods to post the result.
- Ex:
  - AsynchTask<> in Android

Ex: java.util.Timer & TimerTask

- Timer class manages execution flow
- TimerTask class defines its callback interface
- Every 30 mills,
  - The run() method on the “Tick” class gets called
    - Repaint the component so that it animates smoothly

```
5  public class ClockComponent extends JComponent {
6
7    public ClockComponent() {
8      new java.util.Timer().scheduleAtFixedRate(new Tick(), 30, 30);
9    }
10
11   public void paint(Graphics g) {
27
28   private class Tick extends java.util.TimerTask {
29     public void run() {
30       repaint();
31     }
32   }
33 }
```

Ex: javax.swing.Timer & ActionListener

```
int delay = 1000; //milliseconds
ActionListener taskPerformer = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        //...Perform a task...
    }
};
new Timer(delay, taskPerformer).start();
```

```
8   public ClockComponent() {
9       new Timer(1000/60, (a) -> {this.repaint();}).start();
10  }
```

```
public interface ActionListener
extends EventListener
```

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

Since:

`1.1`

See Also:

`ActionEvent, How to Write an Action Listener`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
void	**`actionPerformed(ActionEvent e)`** Invoked when an action occurs.

Method Detail

actionPerformed

```
void actionPerformed(ActionEvent e)
```

Invoked when an action occurs.

Ex: SwingWorker

- Problem:
  - Only the UI thread is allowed to updated UI components
  - Long running tasks should not be run on the UI thread because the UI is frozen while they run
    - the UI cannot accept input because its thread is busy

- Solution:
  - Provide a mechanism to
    1. Allow the UI thread to start long running tasks
    2. Have those tasks run in another thread
    3. Allow that task to execute some operations on the UI thread

javax.swing.SwingWorker

final void	execute()
protected abstract ***T***	***doInBackground****()*
final void	addPropertyChangeListener(PropertyChangeListener listener)
final protected void	setProgress(int progress)
final int	getProgress()
final boolean	cancel(boolean mayInterruptIfRunning)
final boolean	isCancelled()
final boolean	isDone()
protected void	**done**()
final T	get()
final T	get(long timeout, TimeUnit unit)
final protected void	publish(V... chunks)
protected void	**process(List<V> chunks)**

```java
private class CoinFlipTask extends SwingWorker<Void, FlipRecord> {
    protected Void doInBackground() {
        // runs on background/worker thread
        Random rand = ThreadLocalRandom.current();
        FlipRecord flip;
        long heads = 0;
        long tails = 0;
        while(!isCancelled()) {
            if(rand.nextBoolean()) {
                heads++;
            } else {
                tails++;
            }
            publish(new FlipRecord(heads, tails));
        }
        return null;
    }

    protected void process(List<FlipRecord> records) {
        // runs on UI thread
        FlipRecord last = records.get(records.size() - 1);
        lblHeads.setText(Long.toString(last.heads));
        lblTails.setText(Long.toString(last.tails));
        int diff = (int) (last.heads - last.tails);
        lblDiff.setText(Long.toString(diff));
        long total = last.heads + last.tails;
        double var = diff / (double) total;
        lblDiffPercent.setText(Double.toString(var));
    }
}
```

Pattern: IoC in Frameworks

- Your code is like a plugin for a larger service
- The service does all the lifecycle and execution flow management
- It calls your code at the appropriate time to do things
  - Your code executes (in the framework's context)
  - When done, it returns execution flow to the framework
- Timer & TimerTask fit this lens
  - Timer manages flow
  - Calls TimerTask as needed
  - Gets flow back when TimerTask is done

Ex: Timer & TimerTask

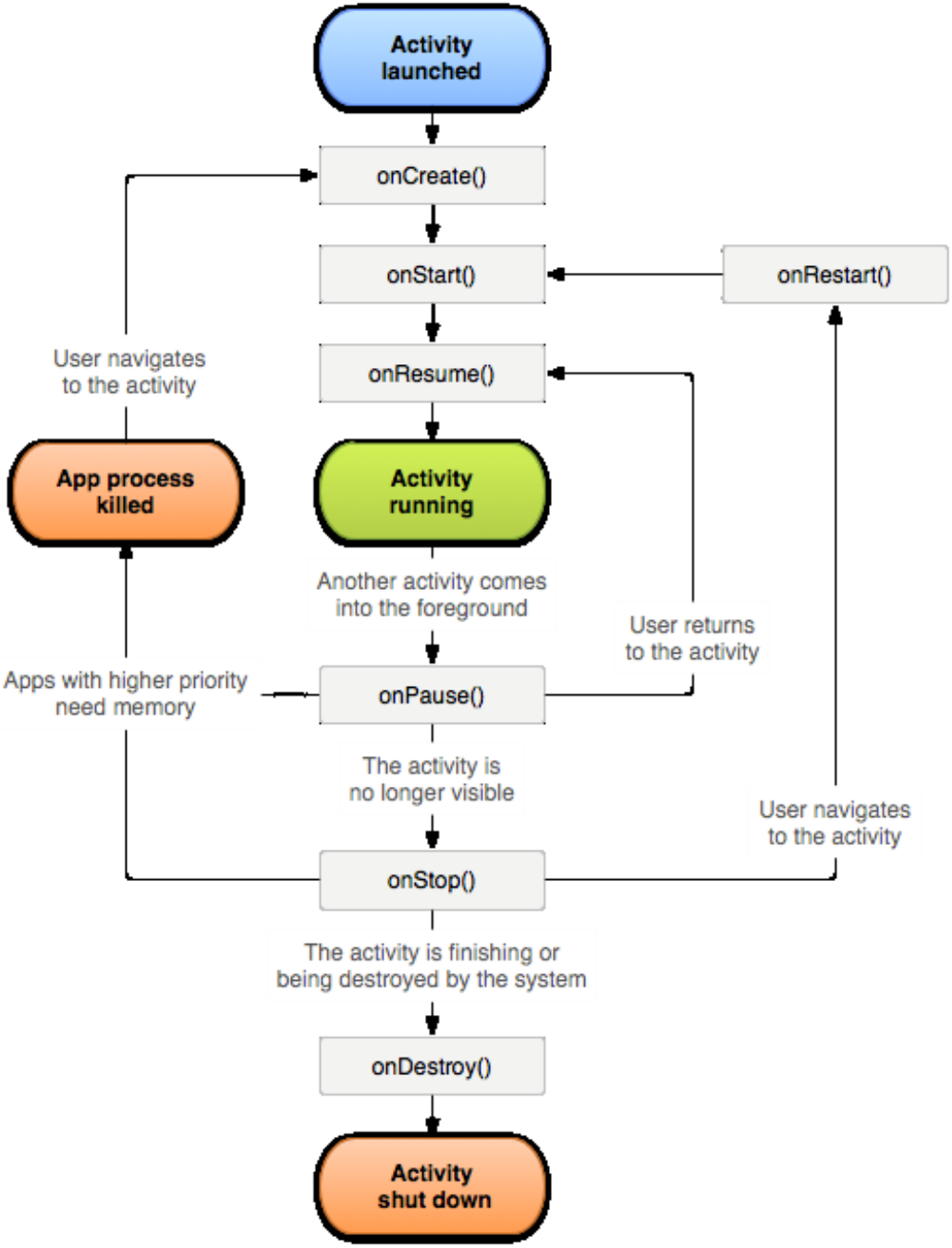
- Timer is the class managing execution flow
- TimerTick is the class defining its callback interface
- Every 30 mills,
  - The run() method on our “Tick” class gets called
    - We repaint our clock so that it animates smoothly
    - Note: our inner class can access things in the outer class

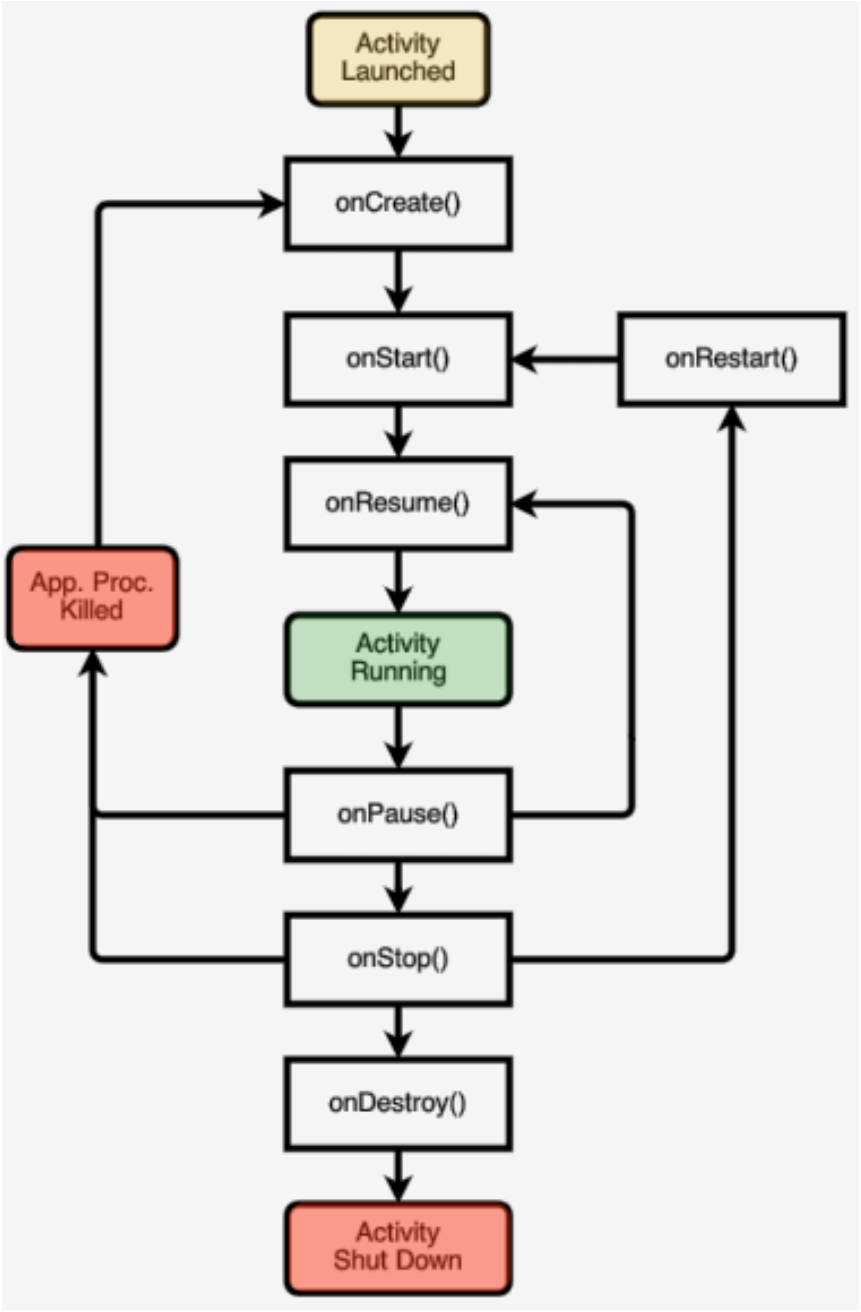
```
5  public class ClockComponent extends JComponent {
6
7    public ClockComponent() {
8      new java.util.Timer().scheduleAtFixedRate(new Tick(), 30, 30);
9    }
10
11   public void paint(Graphics g) {
27
28   private class Tick extends java.util.TimerTask {
29     public void run() {
30       repaint();
31     }
32   }
33 }
```

Ex: Android Activity

- To write an Android App, you create Activities
- An activity represents a single screen in the UI
  - Must draw UI components
  - Must respond to UI events
  - Must respond to other lifecycle events
- Activity lifecycle is managed by the Android framework

- https://developer.android.com/reference/android/app/Activity.html


Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down


Activity Launched
onCreate()
onStart()
onRestart()
onResume()
App. Proc. Killed
Activity Running
onPause()
onStop()
onDestroy()
Activity Shut Down


alive
visible
interactive

Activity & Fragment

```
3  import android.os.Bundle;
4  import android.support.v4.app.FragmentActivity;
5
6  public class SomeActivity extends FragmentActivity {
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_empty);
12
13         getSupportFragmentManager().beginTransaction()
14                 .replace(R.id.frag_empty, SomeDetailFragment.newInstance(getIntent()))
15                 .commit();
16     }
17 }
```

```
1  public class SomeDetailFragment extends Fragment {
2
3      private final static String ARG_ITEM = Item.Factory.KEY;
4
5      private Item item;
6
7      public static SomeDetailFragment newInstance(Item item) {
10
11     public static SomeDetailFragment newInstance(Intent i) {
14
15     private static SomeDetailFragment newInstance(Item item) {
22
23     @Override
24     public void onCreate(Bundle savedInstanceState) {
25         super.onCreate(savedInstanceState);
26
27         Bundle args = getArguments();
28         item = args == null ? null :
29                 Item.Factory.make(args.getBundle(ARG_ITEM));
30     }
31
32     @Override
33     public View onCreateView(LayoutInflater inflater, ViewGroup container,
34                              Bundle savedInstanceState) {
35         View v;
36         if (item != null) {
37             v = inflater.inflate(R.layout.frag_item_detail, container, false);
38             // ... a mountain of UI code ...
39         } else {
40             v = new View(inflater.getContext());
41             v.setVisibility(View.INVISIBLE);
42         }
43         return v;
44     }
45 }
```

Recap

- Inversion of Control
  - Program flow control
- Callbacks
  - You start something
  - It doesn’t return, it calls your method
- Frameworks
  - Manage lifecycle and program flow
  - Calls your methods as needed to provide functionality
  - Think of your code as a sort of plugin