**Event Driven Programming**

Table of Contents

[Overview 2](#_Toc423620423)

[Event Driven paradigm 2](#_Toc423620424)

[Event Driven Programming for the “click” event 2](#_Toc423620425)

[Implementing Event Listeners 2](#_Toc423620426)

[Callback methods (A method to be called in the future) 3](#_Toc423620427)

[Finite-State machines 3](#_Toc423620428)

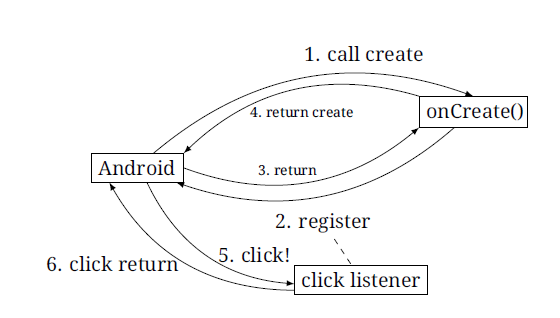
[Polling vs. Interrupts 3](#_Toc423620429)

# Overview

• An event is a notification of a change to the state of your system. Events apply to Embedded systems, Graphical User Interface(GUI).

• You don’t create events yourself, the system create events (sensors, timers, processor) and calls your event listener.

# Event Driven paradigm



# Event Driven Programming for the “click” event

• To receive Click events:

* The app registers an event listener with the object representing the button. **go.setOnClickListener(…)**;

• When the user clicks the button:

* The system executes the click event listener (which is **go.setOnClickListener(…)** )

# Implementing Event Listeners

• **Method 1:**

**Go.setOnClickListener ( new MyClickListener());**

**Class MyClickListener extends View.OnClickListener {**

**Public void onClick(View v) {**

**Log.d (“A2”, “clicked!”); } }**

• **Method 2: (inner class: input a class with its implementation)**

Go.setOnClickListener(new View.OnClickListener() {

Public void onClick(View v) {

Log.d(“A2”, “clicked!”; }} ) ;

Why Inner class? 1. Don’t litter your code with one-time-use classes

2. They can access fields and (final) local variables

# Callback methods (A method to be called in the future)

* The app **registers** a callback to run a time-consuming task
* The app **spawns** the time-consuming task (in different thread), and doesn’t wait for it to finish
* App continues normally, forgetting about the time-consuming task for now
* Once the time-consuming task finishes, the callback executes, notifying the main app about the completion

It permits **asynchronous execution**.

# Finite-State machines

* The app **registers** a callback to run a time-consuming task
* The app **spawns** the time-consuming task (in different thread), and doesn’t wait for it to finish
* App continues normally, forgetting about the time-consuming task for now
* Once the time-consuming task finishes, the callback executes, notifying the main app about the completion

It permits **asynchronous execution**.

# Polling vs. Interrupts

* Refer to my OS notes , IO section.