ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 4 — January 14, 2013

Patrick Lam *version 2*

Skills

In this part of the course, you will learn to write event handling code. You'll see inner classes as a syntactic shorthand for event handlers.

Refresher: Object-Oriented Programming

Java is an object-oriented language. Almost everything is an object (except for a few primitive types, like `int`, `boolean`, `float`, etc.) Objects are instances of classes.

You can define your own classes in Java programs, and all of your code is going to belong to some class. Here's an example of a simple class definition:

```
class Point {
  int x, y;
}
```

(This should be a struct in C#, but Java doesn't have structs.) Once you have a class definition, you can *instantiate* objects of that class:

```
Point p = new Point();
```

Subclassing. Once you have one class, you can then extend it to produce a subclass, for instance:

```
class Mug extends Donut { ... }
```

Note that all `Mug` instances are automatically `Donut` instances; subclassing encodes the "is-a" relationship between classes. Also, Java uses the keyword "extends" rather than the colon (:).

Interfaces. Both C# and Java contain the notion of interfaces, which specify a set of methods that a class must implement. For instance:

```
interface HasHandle {
  void pickup();
}
```

```
class Donut implements HasHandle {
  void pickup() { ... }
  ...
}
```

Any class that implements the `HasHandle` interface must implement the `pickup()` method. So, if you have any `HasHandle` object, you can always call `pickup()` on it.

Event-Driven Programming

We're going to talk about events in today's lecture, because we use the *event-driven paradigm* to receive sensor data in the labs; and user input in the assignments. Other programming paradigms for embedded systems exist, e.g. the superloop structure.

Events are applicable to many application areas beyond embedded systems, notably graphical user interfaces (e.g. mouse clicks, keystrokes are typical events).

An *event* is a notification of a change to the state of your system, such as a button click.

Typically, you don't create events yourself; the system creates them (perhaps in response to notifications from hardware devices—sensors, timers, or the processor itself) and calls your *event listener*, telling you that an event occured. You just wait for events to occur and react to them. Event-driven software is reactive, not proactive: traditionally, events come to you. This represents an inversion of control. (You may, however, schedule an event to be delivered at some point in the future, which includes "right now".)

Why would you use an event-driven programming model?

What are some real-life analogies to event-driven programming?

Here's how to do event-driven programming for the click event, then.

- To receive click events:
  - the application 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.

Implementing Event Listeners. We've seen that the application has to call `setOnClickListener()` on the button. What does it pass to that method? A `View.OnClickListener` object.

You could declare one:

```
class MyClickListener
      extends View.OnClickListener {
  public void onClick(View v) {
    Log.d("A2", "clicked!");
  }
}
```

```
// (somewhere else:)
go.setOnClickListener(new MyClickListener());
```

But, there's a better way:

```
go.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    Log.d("A2", "clicked!");
  }
  });
```

This is called an *inner class*.

Advantages of Inner Classes.

- They don't litter your code with one-time-use classes.
- They can access fields and (final) local variables.

```
class MainActivity {
  int i;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    Button go = (Button) findViewById(R.id.go);
    final int j = 2;
    go.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        Log.d("A2", "i is "+i+" and j is "+j);
      }
      });
  }
}
```

An Alternative to Inner Classes. You have another option. From the Android documentation[1], you could use the following in your Activity XML:

```
<Button
   android:layout_height="wrap_content"
   android:layout_width="wrap_content"
   android:text="@string/self_destruct"
   android:onClick="selfDestruct" />
```

Then, in your activity, you must include the method:

```
public void selfDestruct(View view) {
   // Kabloey
}
```

Callback methods. Applications sometimes need a particular method to be called in the future. *Callback methods* are a way of implementing this design. For instance, event handlers should be called when the appropriate event occurs. One use of callback methods is to structure applications with time-consuming tasks:

- The application registers a callback to run upon completion (or near-completion) of the time-consuming task.
- The application spawns the time-consuming task (perhaps in a different thread), and doesn't wait for it to finish.
- The application continues normally, forgetting about the time-consuming task for now.
- Once the time-consuming task finishes (or is about to finish), the callback executes, notifying the main application about the completion.

Callback methods therefore permit asynchronous (non-blocking) execution.

Can you name a real-life analogy to the workflow described above?

Synchronous vs. Asynchronous Execution. The programs we've seen so far are *synchronous* (or sequential): each instruction executes in sequence, and an instruction can only execute after its predecessor completes. In particular, when a program calls a function, it waits for the function to return before continuing with its own execution, no matter how long the function takes to complete.

Because the application is only doing one thing at a time, it is much easier to understand than asynchronous programs.

It is also possible to write *asynchronous* (or concurrent) programs. In these programs, a main program may spawn a function, or task, and continue executing before the function returns.

Writing asynchronous programs permits higher performance on modern multi-core architectures; for some application domains, concurrency is also a better way to structure the code.

[1] http://developer.android.com/reference/android/widget/Button.html