Lecture 4: Programming Event-Driven Systems

Engineering Design with Embedded Systems

Patrick Lam
University of Waterloo

January 14, 2013

Goal

Be able to program for systems which use event-based models (eg Android).

Refresher: Object-oriented Programming

Fundamental idea: use “objects” to encapsulate data.

class[1] Point { int x, y; }

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

[1] Java doesn’t have `struct`.

Classes versus Instances

Which is the class? Which is the instance?

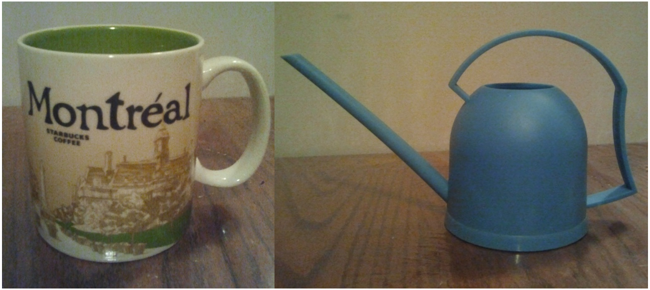
Subclassing

You should also have seen the notion of subclassing:

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

(Note: Java uses the “extends” keyword rather than the : symbol.)

Conceptual Meaning of Subclassing

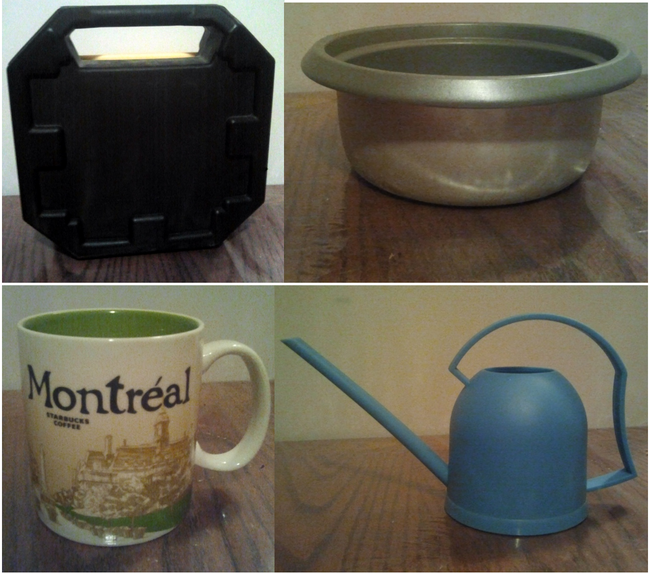


```
class Mug extends Donut { ... }
class WateringCan extends Donut { ... }
```

Subclassing encodes the “is-a” relationship.

Interfaces

One of these things is not like the others:



Interfaces

All but one of these things implement the `HasHandle` interface.

Using Interfaces

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

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

Random Request

In 10 minutes, please remind me that I’m supposed to do something.

Relevance of OO to Android: Events

What happens when you press this?

Android sends an event to the event listener.

Relevance of OO to Android: Events

What happens when you press this?

Android sends an event to the event listener.

Events

An *event* is a notification of a change to the state of your system.

About event-based programming

Reactive, not proactive.

Event Listeners

- 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.

Event Listeners

- 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 (painfully)

We need to pass something to `setOnClickListener()`. What?

This method takes a `View.OnClickListener` object.

You could declare one:

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

...
go.setOnClickListener(new MyClickListener());
```

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

```
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);
      }
      });
  }
}
```

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

Alternative to Inner Classes

You have another option. From the Android documentation[2]:

```
<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
}
```

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

Callback methods

We've been programming with callback methods.

This is also known as "inversion of control".

Key idea: system (user) decides what happens when.

Leveraging callback methods

You can also structure your program with callback methods. Say you have a time-consuming task (TCT).

1. register a callback upon completion of TCT;
2. spawn the TCT in another thread, don’t wait for it;
3. continue normally.

Once the TCT finishes, the callback notifies the main application, which collects results.

Also known as asynchronous, or non-blocking, execution.

Synchronous versus Asynchronous Execution

ECE150: Synchronous, or sequential, programs:

- all instructions execute in sequence;
- an instruction only executes after its predecessor completes.

Also true for function calls.

ECE155, ECE254: Asynchronous, or concurrent, programs:

- most instructions execute in sequence; but
- main program may spawn a function to run concurrently with it.
- Communication via shared memory or via events.

Permits higher performance on multicores, or more relevant structuring. Callbacks are a tool.