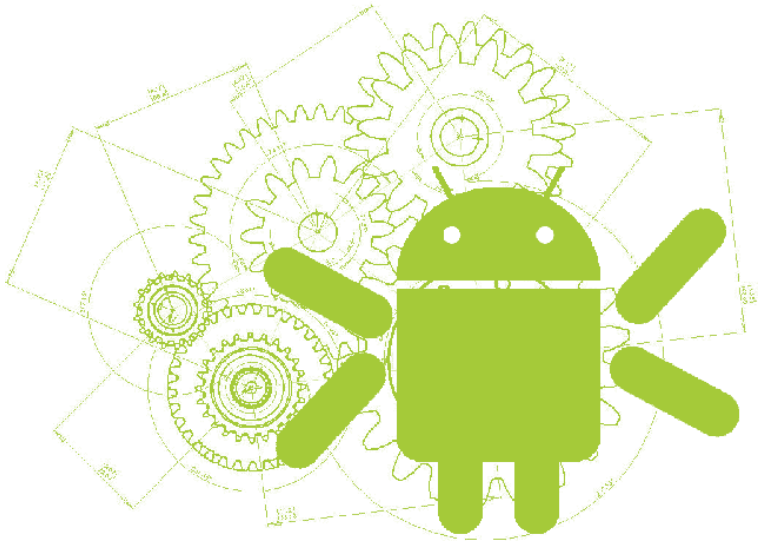
Threads, AsyncTasks, & Handlers

Programming the Android Platform

CS 282
Principles of Operating Systems II
Systems Programming for Android

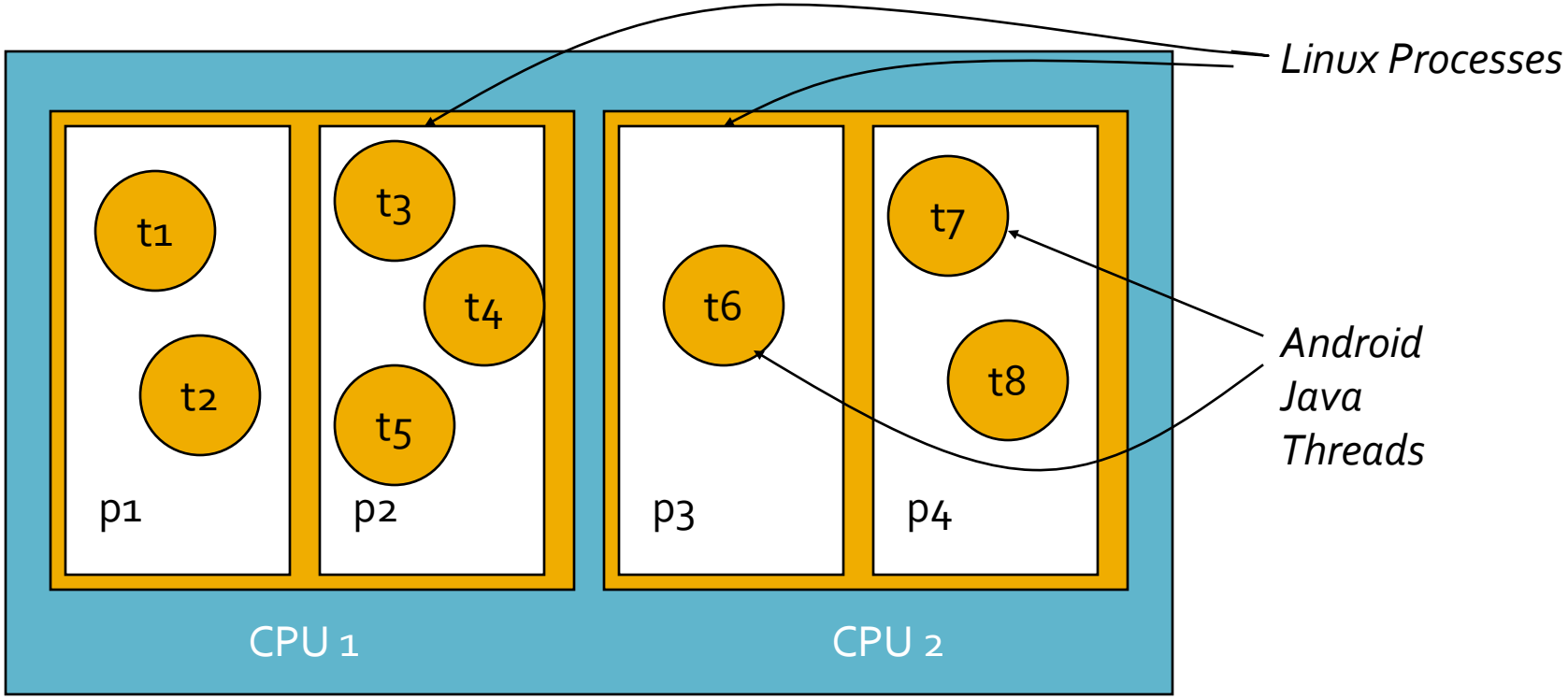
Android Threading

- Android implements Java threads & concurrency classes
- Conceptual view
  - Concurrent computations running in a process
- Implementation view
  - Each Thread has a program counter & a stack (unique)
  - The heap & static areas are shared across threads (common)



http://developer.android.com/guide/components/processes-and-threads.html

Computation Abstractions



Android Mobile Device

Java Threads

- Java provides access to thread creation through the Thread class & Runnable interface
- The Java Thread class is used to create & control a thread
- You provide an implementation of the Runnable interface that specifies what the Thread should do

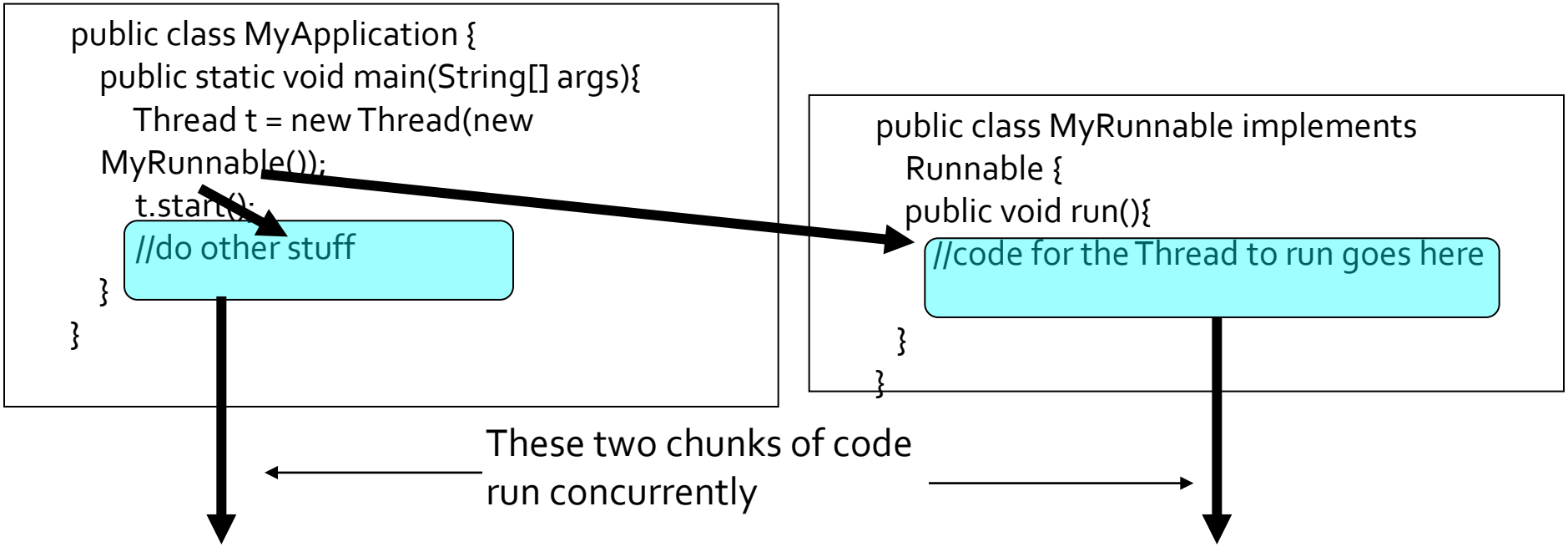
```
public interface Runnable { public void run(); }
```

```
public class MyRunnable implements
   Runnable {
   public void run(){
      //code to run goes here
   }
}
MyRunnable myr = new
     MyRunnable();
new Thread (myr).start();
```

```
public class MyRunnable extends
   Thread {
   public void run(){
      //code to run goes here
   }
}
MyRunnable myr = new
     MyRunnable();
myr.start();
```

Java Threads (cont'd)

- Java provides access to thread creation through the Thread class & Runnable interface
- The Java Thread class is used to create & control a thread
- You provide an implementation of the Runnable interface that specifies what the Thread should do



Java Threads (cont'd)

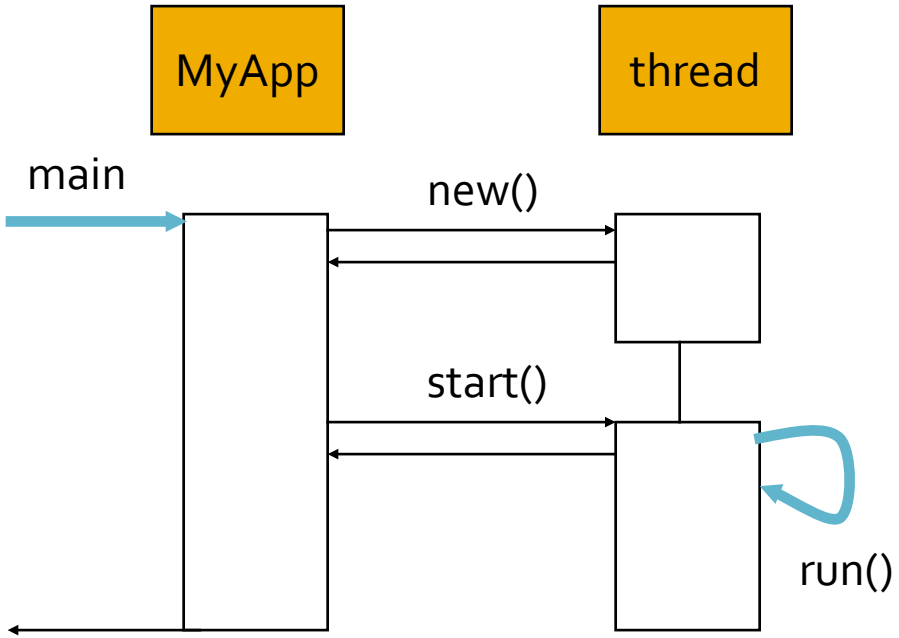
- All threads must be given some code to run
- You specify the code that should be run by implementing the Runnable interface
- Runnables have a "run()" method that is called by the new thread after it starts up
- The thread stays active until run() returns
  - If you want thread to run forever, you need to have a while(true) statement in that run() method
- You can run any block of code in a separate thread, but it must be inside of a run() method or called from a run() method of a Runnable

```
public interface Runnable {
    // ...
        public void run();
   // ...
}
```

Java Threads (cont'd)

- Starting a thread using an inner class as the Runnable

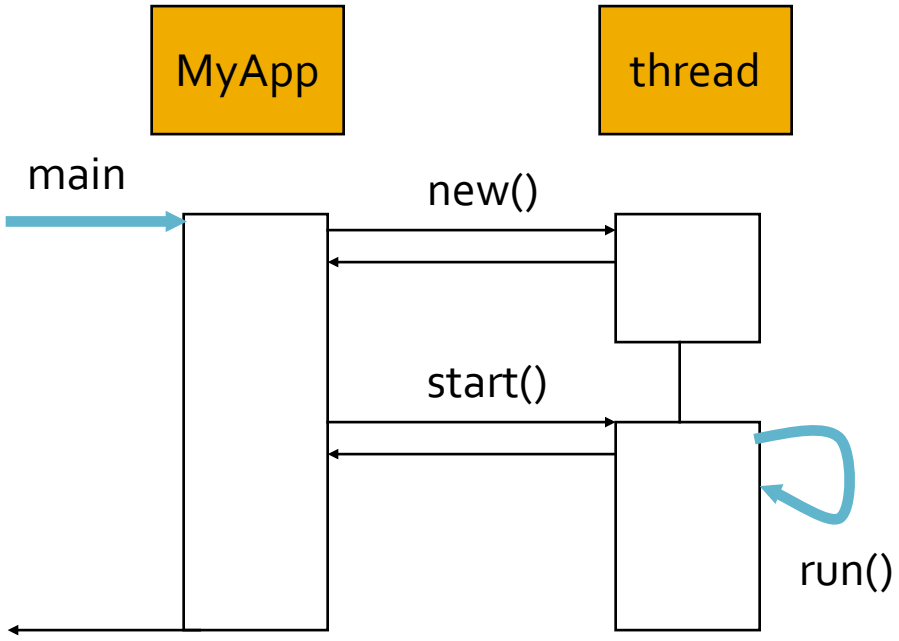
```
public class MyApp {
   private class MyWorker implements
                  Runnable {
        public void run()
        {  /* your concurrent
              code here */ }
   }

public static void main(String args[] ) {
       Thread t = new Thread(new
                        MyWorker());
       t.start();
       // Do other stuff concurrently
   }
}
```



Java Threads (cont'd)

- Starting a thread using an anonymous inner class as the Runnable

```
public class MyApp {
    public static void main(String
                                args[] ) {
        new Thread(new Runnable() {
            public void run(){
                //code to run in parallel
            }
        }).start();
    // Do other stuff concurrently
}
```



Some Java Thread Methods

- void start() - starts the Thread
- void interrupt() - send an interrupt request to calling Thread
- void sleep(long time) - sleep for the given period
- void join() - wait for a thread to die

```
for (int i = 0; i < importantInfo.length; i++) {
  // Pause for 4 seconds
  try { Thread.sleep(4000); }
  catch (InterruptedException e) {
    // We're interrupted: no more messages.
    return;
  } // Print a message
  System.out.println(importantInfo[i]); }
```

```
for (int i = 0; i < inputs.length; i++) {
   heavyCrunch(inputs[i]);
   if (Thread.interrupted())
      { // We're interrupted: no more crunching.
        throw InterruptedException(); }
}
```

Stopping Java Threads

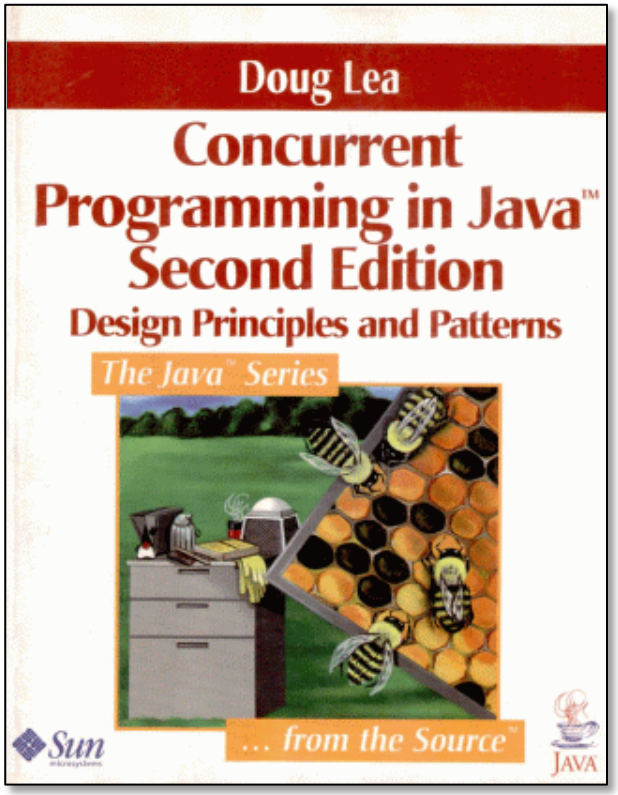
- There is no safe "stop" method for a Thread in Java
- If you are going to create a long running operation inside of your run() method, you must ensure your code can stop!
- If you don't want to use the interrupt() method described earlier, a simple way to have a "stop" flag
  - Add a boolean flag "running_" to your class that implements Runnable
  - Initially, set "running_" to true
  - Have a stop() method that sets "running_" to false

```
public class MyTask{
    private boolean running_ = true;
    public void stop(){running_ = false; }
      public void run(){
      while(running_) { /* do stuff */ }
    }
}
```

Some Static Java Thread Methods

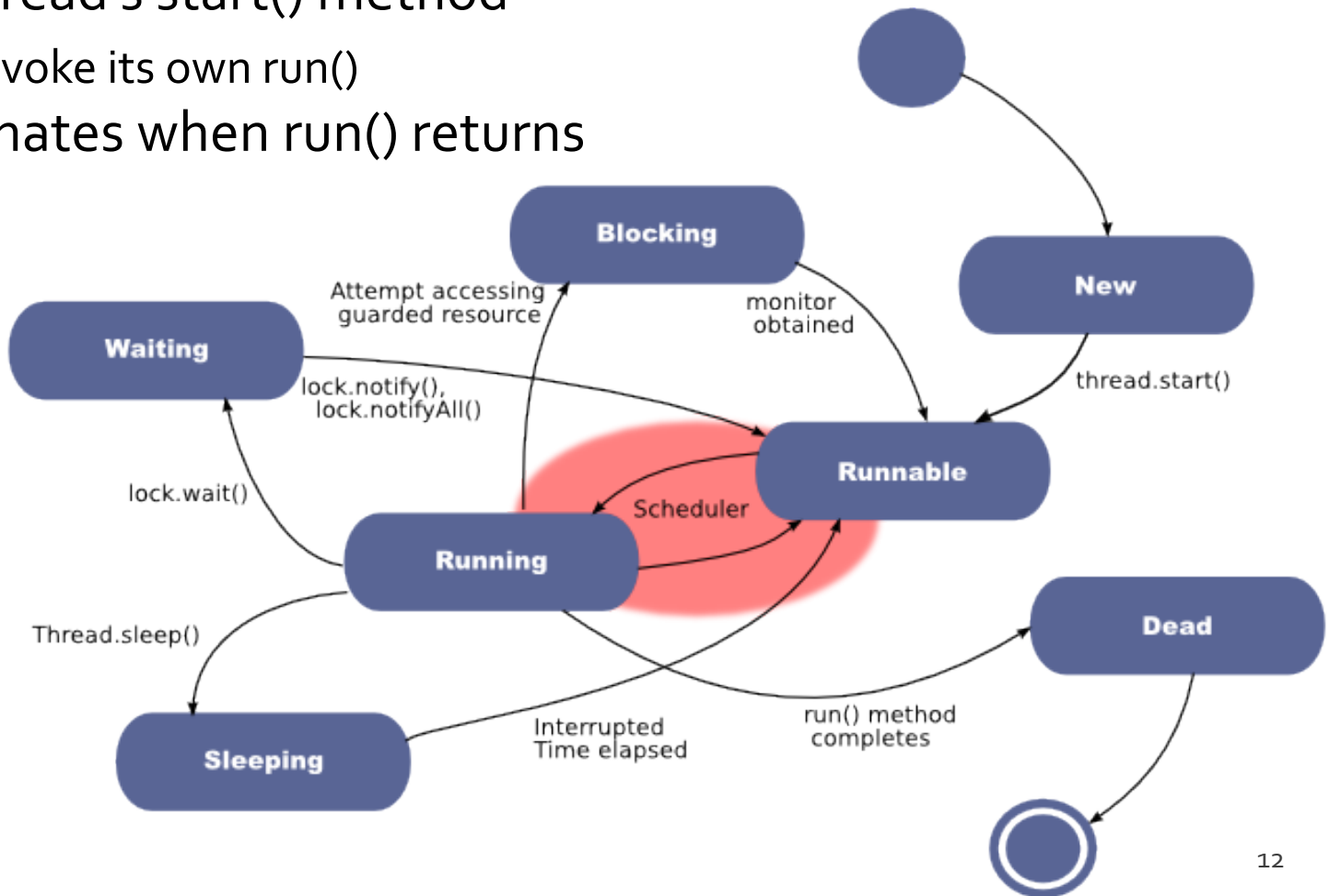
- boolean isAlive() - returns true if the thread has been started, but hasn't yet terminated
- Thread currentThread() - thread object for currently executing thread
- Boolean holdsLock(Object object) - returns true if calling Thread holds an intrinsic lock on object

- For more info see Doug Lea's book on Java concurrency at http://gee.cs.oswego.edu/dl/cpj



Basic Thread Use Case

- Instantiate a Thread object
- Invoke the Thread's start() method
  - Thread will invoke its own run()
- Thread terminates when run() returns



Java Synchronization

- Problem: How do you protect critical sections of code from being executed by two threads in parallel
- What is the result of calling startMultiThreadedAccount()?

```
public class MyBankAccount{
  private int balance_ = 1000;
  private void withdrawFunds(int amount) { balance_ -= amount; }
  public int getBalance(){ return balance_; }

public void startMultiThreadedAccount(){
    Thread t1 = new Thread(
      new Runnable(){ public void run(){ withdrawFunds(100);} });
    Thread t2 = new Thread(
      new Runnable(){ public void run(){ withdrawFunds(100);}});
    t1.start();
    t2.start();
  }
}
```

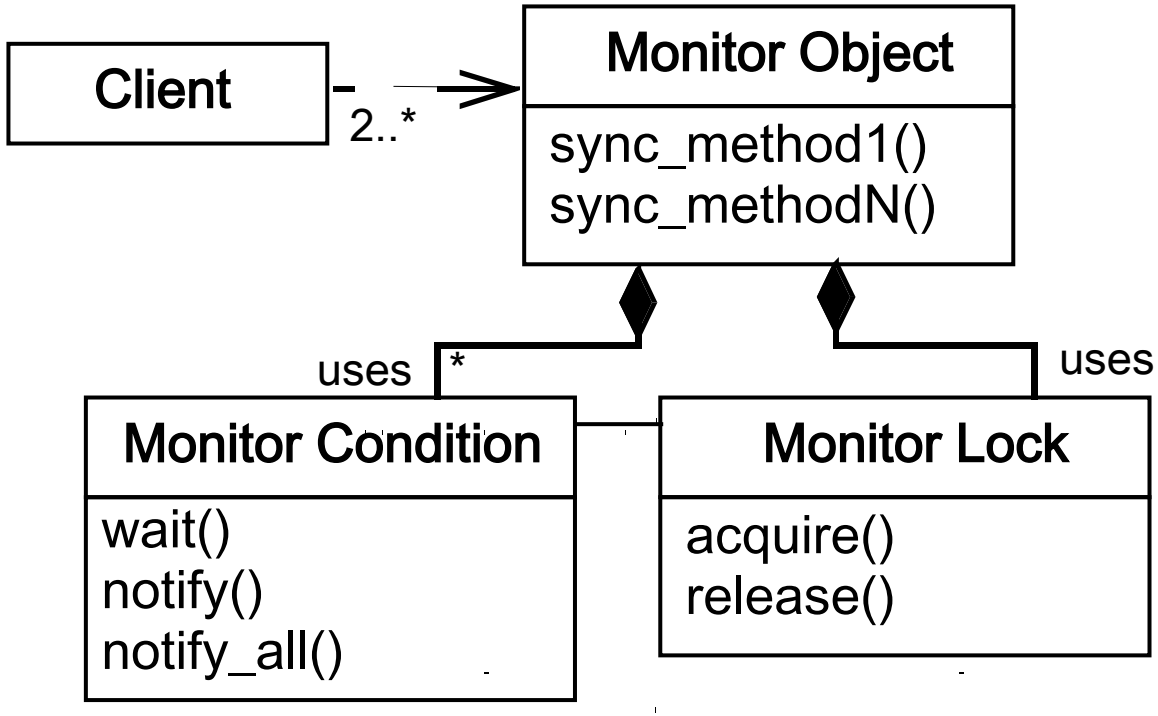
Synchronization via Monitor Objects

- Java provides the "synchronized" keyword to specify sections of code that cannot be accessed concurrently by two threads
- Only a single synchronized method can be active in an object
- All Java objects use the Monitor pattern
- withdrawFunds & getBalance can never execute concurrently:

```
public class MyBankAccount {
  private int balance_ = 1000;

  public synchronized void withdrawFunds(int amount) {
      balance_ -= amount;
  }
  public synchronized int getBalance() { return balance_; }
}
```

Monitor Object Pattern

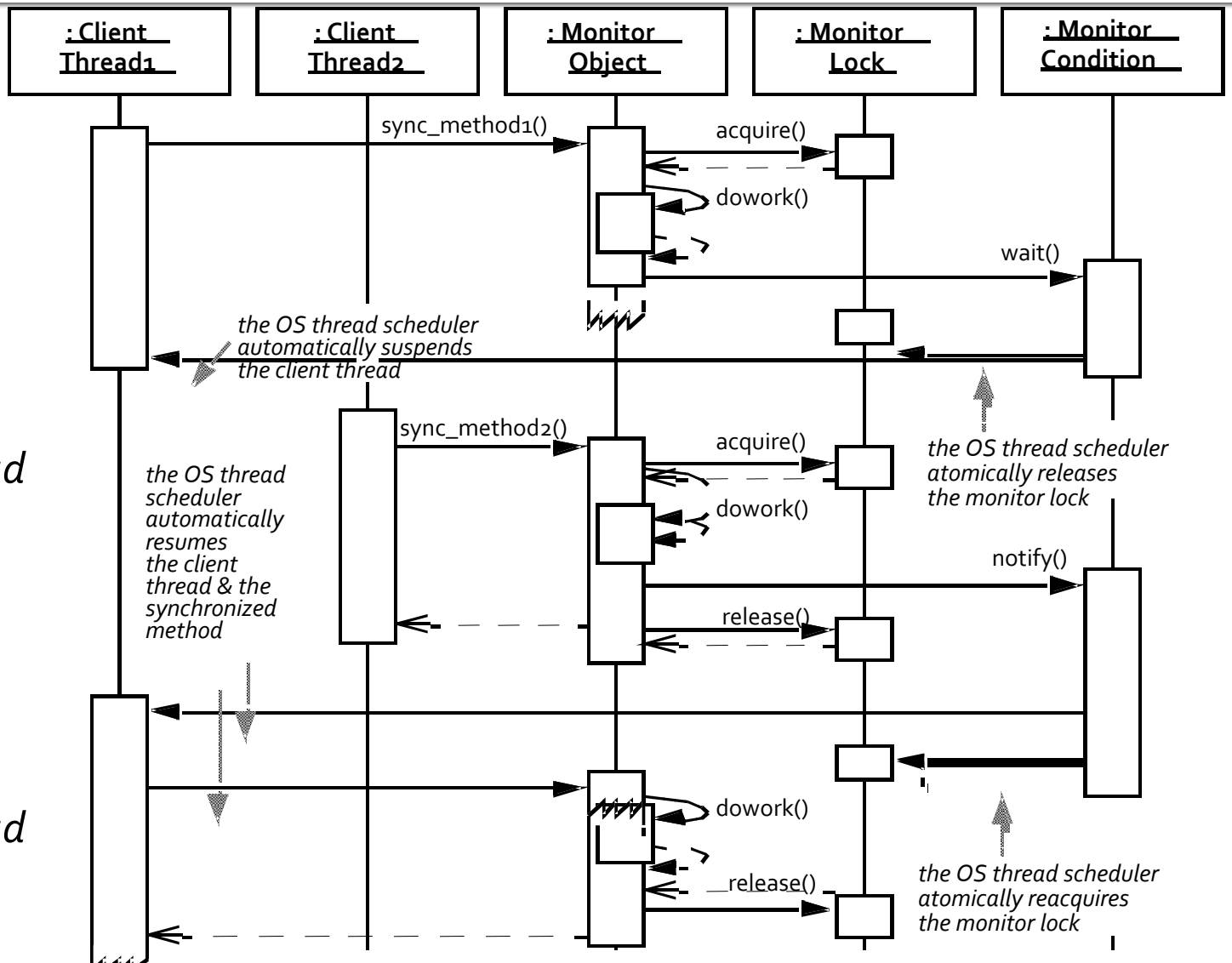
- This pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object
- It also allows an object's methods to cooperatively schedule their execution sequences



http://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2

Monitor Object Dynamics

1. *Synchronized method invocation & serialization*
2. *Synchronized method thread suspension*
3. *Monitor condition notification*
4. *Synchronized method thread resumption*



Monitor Objects in Java

- All objects in Java can be Monitor Objects
- Access to a synchronized method is serialized w/other synchronized methods
- Java objects have wait() & notify()/notifyAll() methods to allow callers to wait for a condition to become true
- Calling wait() on an object will suspend the current thread until a corresponding notify() call is made on the same object

```
public class MessageQueue {
    public synchronized void put(Msg m){…}
    public synchronized Msg get(){…}
}
```

```
public class MessageQueue {
    public synchronized void put(Msg m){
        ….
        notify();
    }

    public synchronized Msg get(){
        while (size() == 0){ wait(); }
        …
    }
}
```

Monitor Objects in Java (cont'd)

What is the output of this program? How would you fix it?

```
public class MyBankAccount{
  private List queue_ = new ArrayList();
  public synchronized void publish
     (String msg){
     queue_.add(msg);
    }
  public synchronized String
    consume(){
     return queue_.remove(0);
  }
}
```

```
public void start(){
    Thread t = new Thread(new Runnable(){
    public void run(){
     for(int i = 0; i < 10; i++) publish("foo");
    });
  Thread t2 = new Thread(new
       Runnable(){
        public void run(){
          while(true){
           System.out.println(consume());}});
  t1.start();
  t2.start();
```

Monitor Objects in Java (cont'd)

- Inside of a synchronized method, you can request that a thread "wait" for a condition, e.g.:
  - if you have a queue, a method to get from the queue can check the queue size & wait() if the queue is empty
  - The thread calling the method blocks on the wait() call & does not continue execution until another thread tells it that the queue has data to process
  - When the thread is notified, it wakes up, obtains the lock for the method again, & continues execution after the wait() call

```
public class MyBankAccount{
  private List queue_ = new
    ArrayList();
  …

  public synchronized String
    consume(){
    while (queue_.size() == 0) {
          wait();
    }
    return queue_.remove(0);
  }

…
```

Monitor Objects in Java (cont'd)

- When a thread is waiting on a condition, another thread can use notify() to wake up the waiting thread
- notify() is called on the *condition* & not the waiting thread, e.g.:
  - Thread A calls wait() on Object 1
  - Thread B calls notify() on Object 1 to wake up Thread A
  - If Thread B calls notify() on Object 2, it would have no affect on Thread A b/c it is waiting on Object 1 (e.g. the condition)

```
public class MyBankAccount{
  private List queue_ = new
    ArrayList();

  public synchronized void
    publish(String msg){
    queue_.add(msg);
    notify();
    }
...
```

- When notify() is called, one thread waiting on the notified object will *eventually* be give the lock back for the synchronized method it was running in & allowed to continue
  - notifyAll() can be used to wake up all waiting threads

Pros & Cons of Monitor Object Pattern

This pattern provides two **benefits**:

- ***Simplifies concurrency control***
  - The Monitor Object pattern presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations
- ***Simplification of scheduling method execution***
  - Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution & that of collaborating monitor objects

This pattern can also incur **liabilities**:

- The use of a single monitor lock can ***limit scalability*** due to increased contention when multiple threads serialize on a monitor object
- ***Complicated extensibility semantics***
  - These result from the coupling between a monitor object's functionality & its synchronization mechanisms
- It's hard to inherit from a monitor object due to the ***inheritance anomaly*** problem
- ***Nested monitor lockout***
  - This problem is similar to the preceding liability & can occur when a monitor object is nested within another monitor object

http://www.dre.vanderbilt.edu/~schmidt/C++2java.html

Simple Thread Example

```
public class SimpleThreadingExample extends Activity {
  private Bitmap bitmap;
  public void onCreate(Bundle savedInstanceState) {
  …
  final ImageView iview = …
  new Thread(new Runnable() {
     public void run() {
       synchronized (iview) {
         bitmap = BitmapFactory
              .decodeResource(getResources(),R.drawable.icon);
         iview.notify();
       }
     }
}).start();
…
```

Thread Example (cont.)

```
final Button button = …
button.setOnClickListener(new OnClickListener() {
  public void onClick(View v) {
    synchronized (iview) {
      while (null == bitmap) {
        try {
          iview.wait();
        } catch (InterruptedException e) {…}
      }
      iview.setImageBitmap(bitmap);
    }
  }
});
…
```

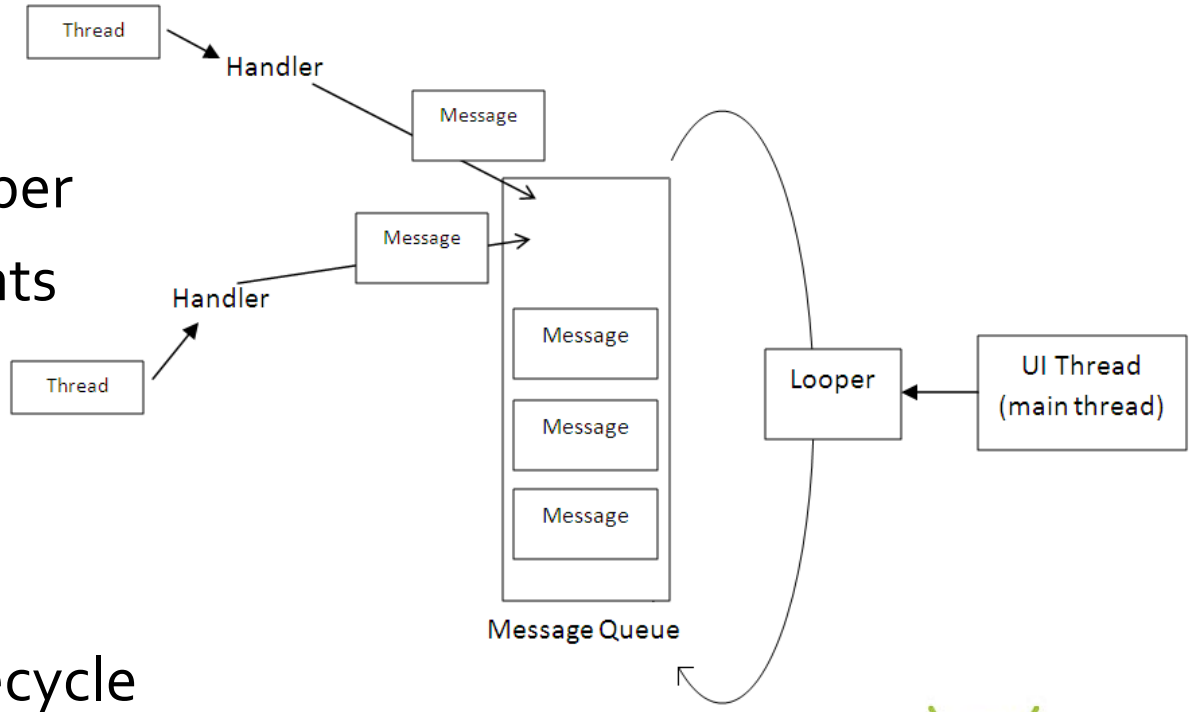
The Looper

- Android class for providing message queue for threads
  - Threads by default do not have a message loop associated with them; to create one, call prepare() in the thread that is to run the loop, & then loop() to have it process messages until the loop is stopped
- Most interaction with a message loop is through Handlers
- HandlerThread
  - Handy class for starting a new thread that has a looper

```
class LooperThread extends Thread {
    public Handler mHandler;

    public void run() {
      Looper.prepare();

      mHandler = new Handler() {
        public void handleMessage
                              (Message msg) {
          // process incoming messages here
        }
      };

      Looper.loop();
    }
}
```

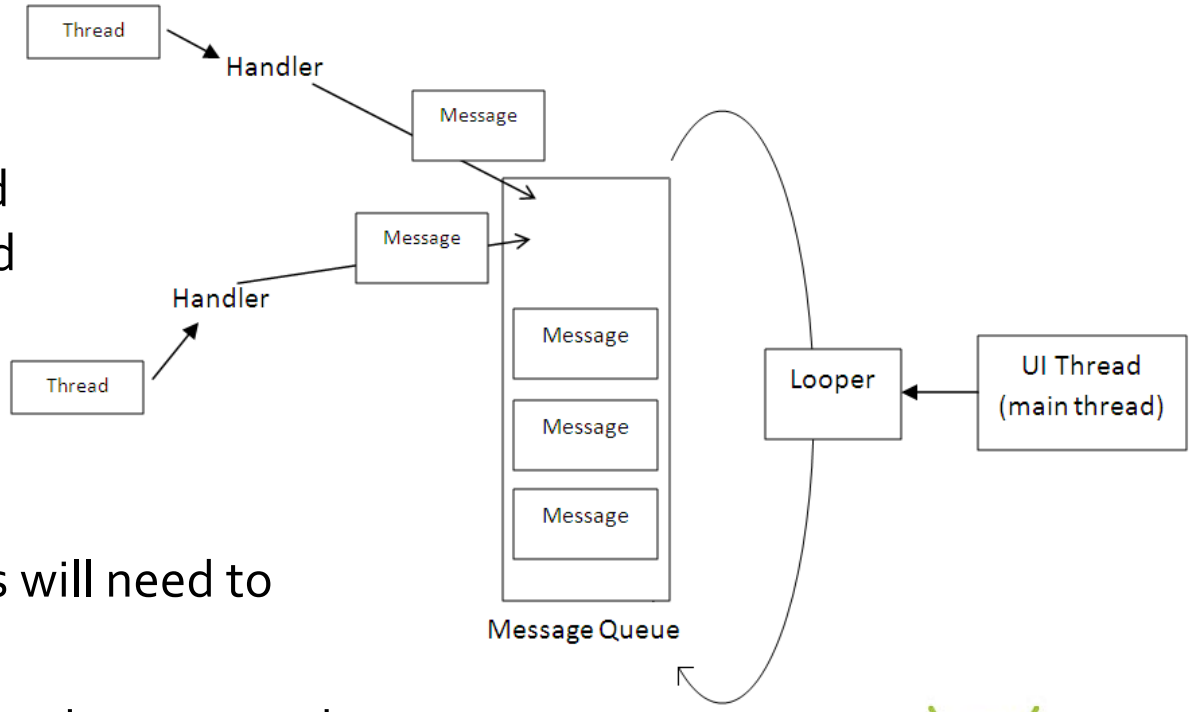
The UIThread

- Applications have a main thread (the UI thread), which is a looper
- Application components in the same process use the same main thread
- User interaction, system callbacks & lifecycle methods handled in the UI thread
- The Android UI toolkit is not thread safe



UI Thread Implications

- Blocking the UI thread hurts responsiveness
  - Long-running ops should run in background thread
- Don't access the UI toolkit from non-UI thread
  - UI & background threads will need to communicate
  - A typical approach is to post messages to the looper thread's message queue



Posting Runnables on UI thread

```
public class SimpleThreadingExample extends Activity {
  private Bitmap bitmap;
  public void onCreate(Bundle savedInstanceState) {
  …
    final ImageView iview = …
    final Button button = …
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        new Thread(new Runnable() {
          public void run() {
            Bitmap  = …
            iview.post(new Runnable() {
              public void run() { iview.setImageBitmap(bitmap);}
            });
          }
        }).start();
…
```

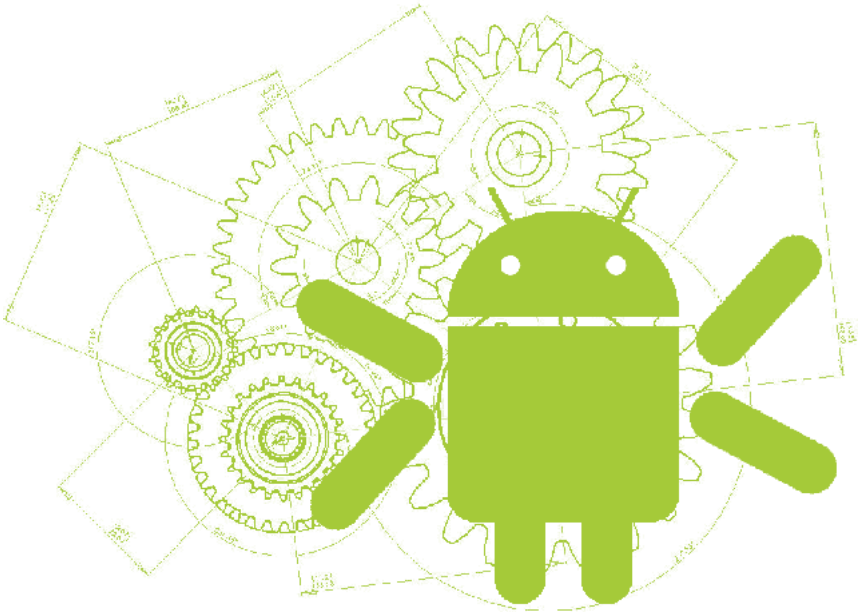
Posting Runnables on UI thread

```
public class SimpleThreadingExample extends Activity {
  private Bitmap bitmap;
  public void onCreate(Bundle savedInstanceState) {
  …
    final ImageView iview = …
    final Button button = …
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        new Thread(new Runnable() {
          public void run() {
            Bitmap  = …
            SimpleThreadingExample.this
                    .runOnUiThread( new Runnable() {
                            public void run() { iview.setImageBitmap(bitmap);}
            });
          }
        }).start();
…
```

AsyncTask

- Structured way to manage work involving background &UI threads
  - Simplifies creation of long-running tasks that need to communicate with the UI
- In background thread
  - Perform work
- In UI Thread
  - Setup
  - Indicate progress
  - Publish results

- Must be subclassed
- Instance must be created on UI thread
- Instance can only be executed once



AsyncTask (cont.)

- Generic class

```
class AsyncTask<Params, Progress, Result> {
  …
}
```

- Generic type parameters
  - Params – Types used in background work
  - Progress – Types used when indicating progress
  - Result – Types of result

AsyncTask (cont.)

- void onPreExecute()
  - Runs before doInBackground()
- Result doInBackground (Params… params)
  - Performs work "in the background"
  - Can call void publishProgress(Progress… values) periodically
- void onProgressUpdate (Progress… values)
  - Invoked in response to publishProgress()
- void onPostExecute (Result result)
  - Runs after doInBackground()

AsyncTask (cont.)

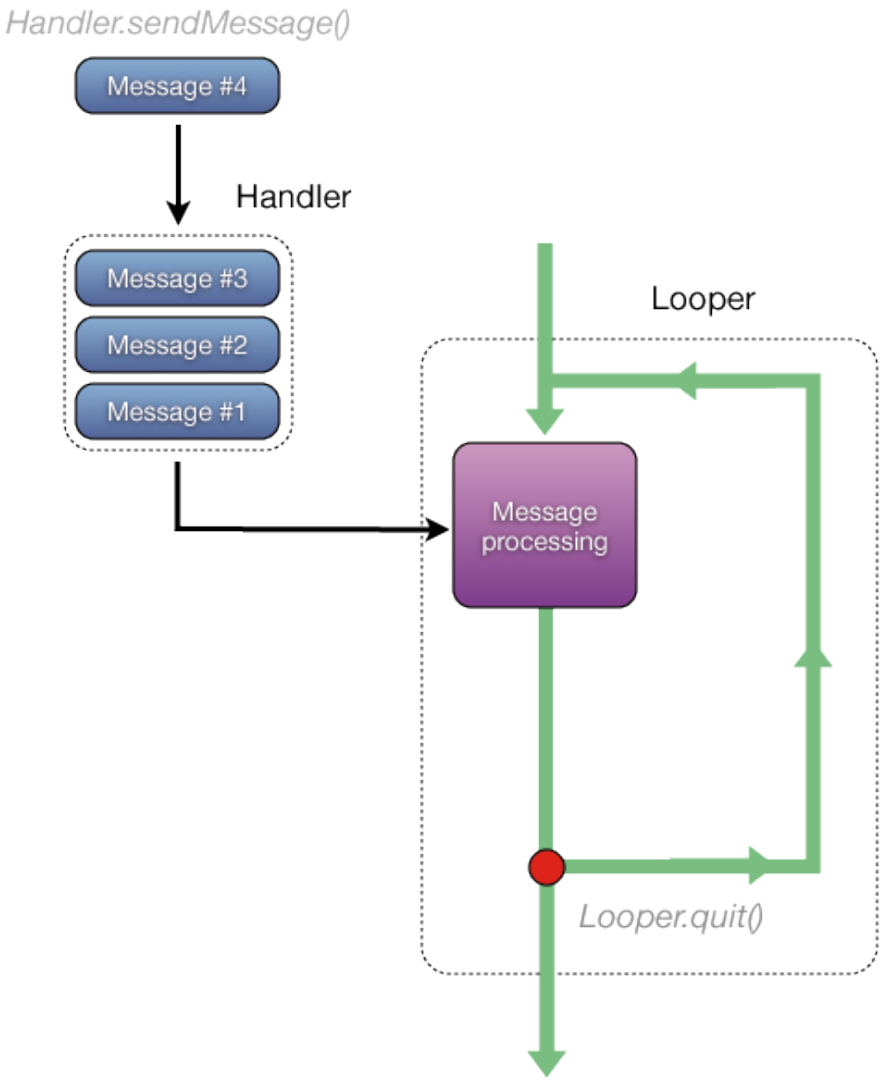
```
public class SimpleThreadingExample extends Activity {
  ImageView iview;
  ProgressBar progress;
  public void onCreate(Bundle savedInstanceState) {
    ...
    iview = ...
    progress = ...
    final Button button = ...
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        new LoadIconTask().execute(R.drawable.icon);
      }
    });
  }
...
```

AsyncTask (cont.)

```
class LoadIconTask extends AsyncTask<Integer, Integer, Bitmap> {
  protected Bitmap doInBackground(Integer... resId) {
    Bitmap tmp = BitmapFactory.decodeResource(
    getResources(), resId[0]);
    // simulate long-running operation
       publishProgress(…);
    return tmp;
  }
  protected void onProgressUpdate(Integer... values) {
    progress.setProgress(values[0]);
  }
  protected void onPostExecute(Bitmap result) {
    iview.setImageBitmap(result);
  }
…
```

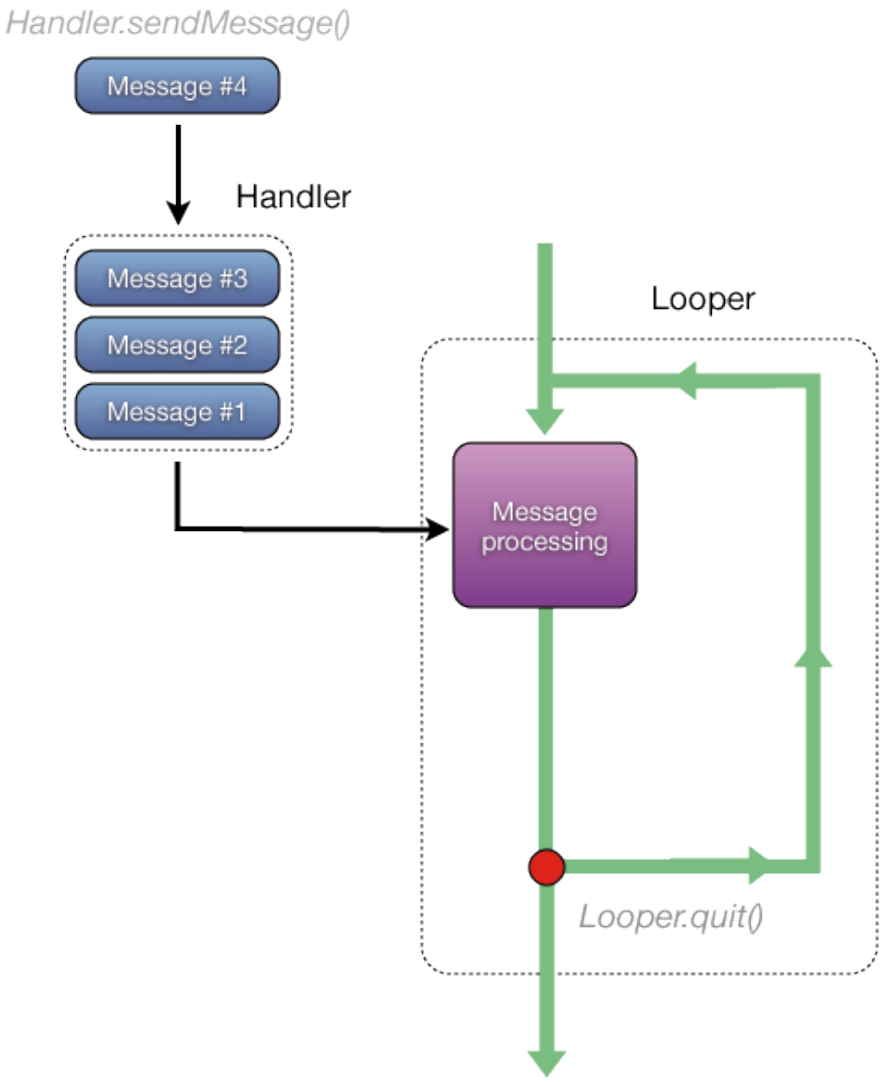
Handler

- Threads can also communicate by exchanging Messages & Runnables
- Connects to thread's looper
  - One per Thread
  - Manages MessageQueue
  - Dispatches MessageQueue entries
- Handler
  - Sends Messages & Runnables to Thread
  - Implements processing for Messages
  - Thread-safe



Handler (cont.)

- Two main uses for a Handler
  - Schedule Message/Runnable for future execution
  - Enqueue action to be performed on a different thread
- Extend the Handler base class & override the hook method handleMessage(Message msg)



Runnables & Handlers

- boolean post(Runnable r)
  - Add Runnable to the MessageQueue
- boolean postAtTime(Runnable r, long uptimeMillis)
  - Add Runnable to the MessageQueue. Run at a specific time (based on SystemClock.upTimeMillis())
- boolean postDelayed(Runnable r, long delayMillis)
  - Add Runnable to the message queue. Run after the specified amount of time elapses

Runnables & Handlers (cont.)

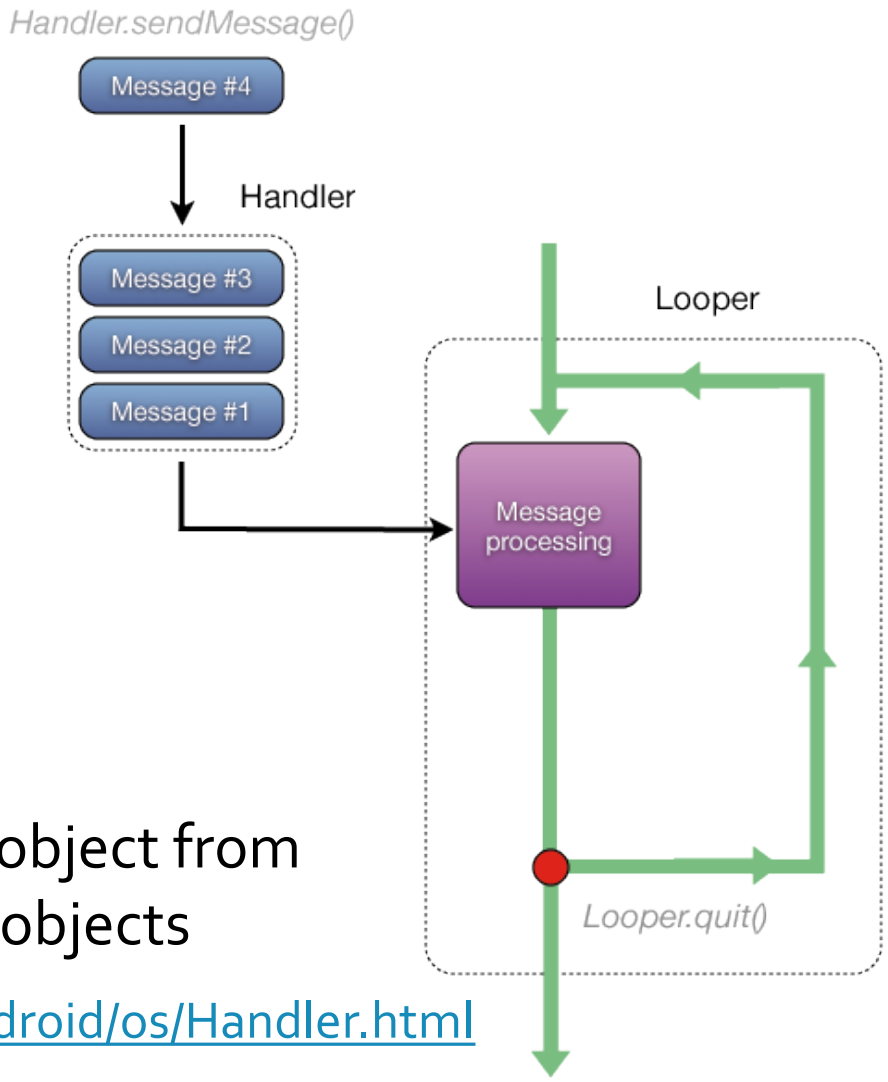
```
public class SimpleThreadingExample extends Activity {
  private ImageView iview;
  private Handler handler = new Handler();
  public void onCreate(Bundle savedInstanceState) {
    …
    iview = …
    final Button = …
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        new Thread(new LoadIconTask(R.drawable.icon)).start();
      }
    });
  }
…
```

Runnables & Handlers (cont.)

```
private class LoadIconTask implements Runnable {
  int resId;

  LoadIconTask(int resId) {
    this.resId = resId;
  }

  public void run() {
    final Bitmap tmp =
                    BitmapFactory.decodeResource(getResources(),resId);
    handler.post(new Runnable() {
        public void run() {
           iview.setImageBitmap(tmp);
        }
     });
  }
...
```

Messages & Handlers

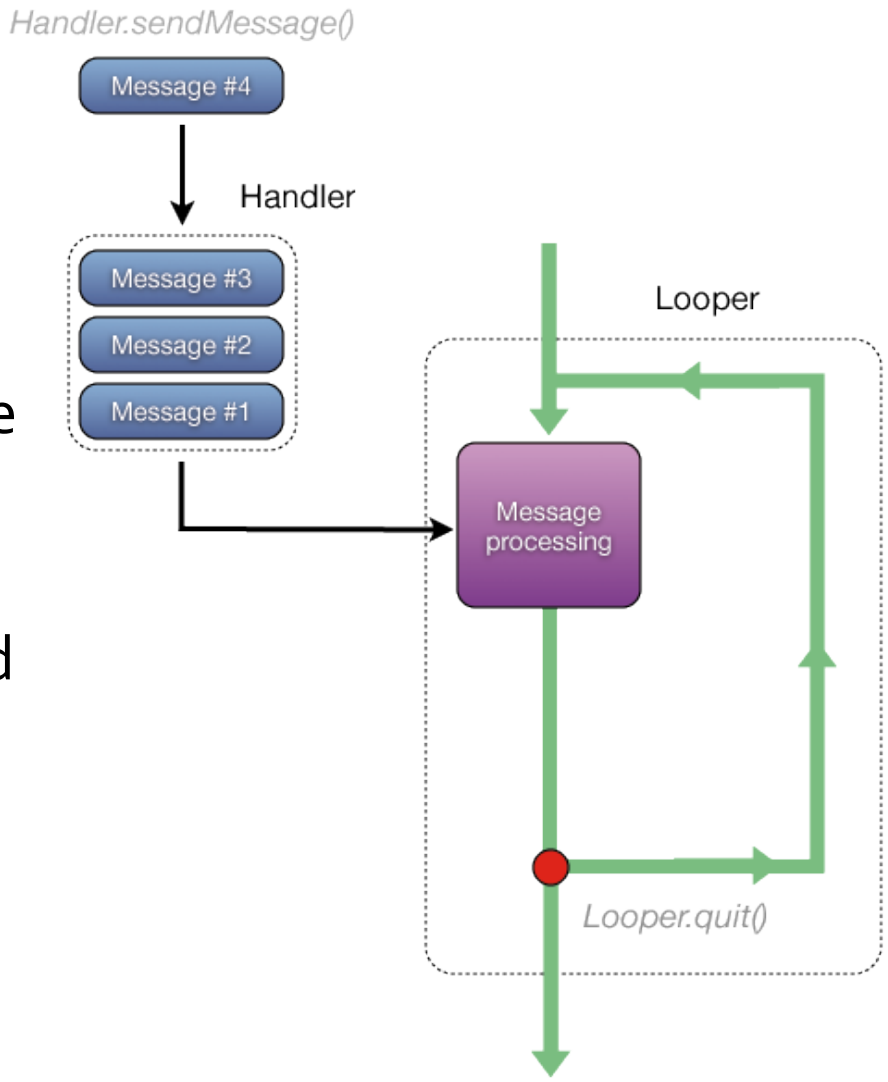
- Create Message & set Message content
  - Handler.obtainMessage()
  - Message.obtain()
  - Many variants, see documentation
- Message parameters include
  - int arg1, arg2
  - int what
  - Object obj
  - Bundle data
- Message.obtain() returns message object from global pool to avoid allocating new objects



http://developer.android.com/reference/android/os/Handler.html

Messages & Handlers (cont.)

- sendMessage() - puts the Message at the end of the queue immediately
- sendMessageAtFrontOfQueue() – puts the Message at the front of the queue immediately
- sendMessageAtTime() – puts the message on the queue at the stated time
- sendMessageDelayed() – puts the message after the delay time has passed



Messages & Handlers (cont.)

```
public class SimpleThreadingExample extends Activity {
  …
  Handler handler = new Handler() {
    public void handleMessage(Message msg) {
      switch (msg.what) {
        case SET_PROGRESS_BAR_VISIBILITY: {
              progress.setVisibility((Integer) msg.obj);  break; }
        case PROGRESS_UPDATE: {
              progress.setProgress((Integer) msg.obj); break; }
        case SET_BITMAP: {
              iview.setImageBitmap((Bitmap) msg.obj);  break; }
      }
…
```

Messages & Handlers (cont.)

```
public void onCreate(Bundle savedInstanceState) {
  ...
  iview = ...
  progress = ...
  final Button button = ...
  button.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
      new Thread(
              new LoadIconTask(R.drawable.icon, handler)).start();
    }
  });
 }
...
```

Messages & Handlers (cont.)

```
private class LoadIconTask implements Runnable {
...
public void run() {
  Message msg = handler.obtainMessage (
                SET_PROGRESS_BAR_VISIBILITY, ProgressBar.VISIBLE);
  handler.sendMessage(msg);
  final Bitmap tmp =
                BitmapFactory.decodeResource(getResources(),resId);
  for (int i = 1; i < 11; i++) {
    msg = handler.obtainMessage(PROGRESS_UPDATE, i * 10);
    handler.sendMessageDelayed(msg, i * 200);
  }
...
```

Messages & Handlers (cont.)

```
...
   msg = handler.obtainMessage(SET_BITMAP, tmp);
   handler.sendMessageAtTime(msg, 11 * 200);
   msg = handler.obtainMessage(
                  SET_PROGRESS_BAR_VISIBILITY, ProgressBar.INVISIBLE);
   handler.sendMessageAtTime(msg, 11 * 200);
   }
  }
}
```

Lab Assignment