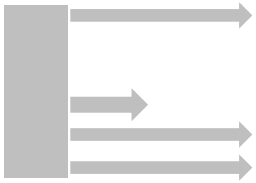
Lesson 11

Concurrency Control

Victor Matos
Cleveland State University

Concurrency Control

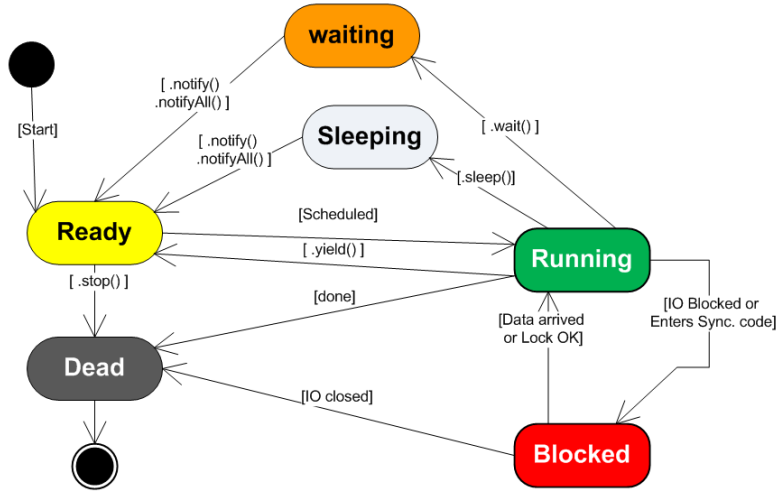
Android's Threads



1. On certain occasions a single app may want to do more than one 'thing' at the same time. For instance, show an animation, download a large file from a website, and maintain a responsive UI for the user to enter data. One solution is to have the app run those individual **concurrent** actions in separate **threads**.
2. The Java Virtual-Machine provides its own **Multi-Threading** architecture (as a consequence the JVM & Dalvik-VM are hardware independence).
3. Threads in the same VM interact and synchronize by the use of **shared objects** and **monitors** .
4. Each virtual machine instance has at least one **main thread** .
5. Each thread has its own call **stack**. The call stack is used on method calling, parameter passing, and storage of the called method's local variables.

Life Cycle of a Java Thread

Java *threading* provides its own abstraction of **concurrent** execution (which is hardware & OS independent). The activity diagram below shows the different possible states a Java thread could reach during its life-cycle.



Concurrency Control

Creating and Executing Threads

The following are two strategies for creating and executing a Java Thread

Style1. Create a new **Thread** instance passing to it a **Runnable** object.

```
Runnable myRunnable1 = new MyRunnableClass();
Thread t1 = new Thread(myRunnable1);
t1.start();
```

Style2. Create a new custom sub-class that *extends* **Thread** and override its **run()** method.

```
MyThread t2 = new MyThread();
t2.start();
```

In both cases, the **start()** method must be called to execute the new Thread. (Use **runnable** on classes that want to fork but already extend another class)

Concurrency Control

Classic Java JDK Monitors (Mutex)

A monitor is a region of critical code executed by only one thread at the time. To implement a *Java Monitor* you may use the **synchronized** modifier, and obtain a mutually **exclusive lock** on an object (data or code). When a thread acquires a lock of an object (for reading or writing), other threads must wait until the lock on that object is released.

```
public synchronized void methodToBeMonitored() {
   // place here your code to be lock-protected
   // (only one thread at the time!)
}

public synchronized int getGlobalVar() {
   return globalVar;
}

public synchronized void setGlobalVar(int newGlobalVar) {
   this.globalVar = newGlobalVar;
}

public synchronized int increaseGlobalVar(int inc) {
   return globalVar += inc;
}
```

Warning

synchronized doesn't support separate locks for reading and writing.

This restriction creates lower than desired performance as no multiple-readers are allowed on a resource.

A better solution is ReadWriteLocks

Concurrency Control

Classic Java JDK Monitors (Mutex)

Another common expression to obtain a mutually **exclusive lock** on an object follows:

```
synchronized ( object ) {
  // place here your code to exclusively
  // work on the locked object
  // (only one thread at the time!)
}
```


Concurrency Control

Java JDK ReadWriteLocks

Better performance occurs when multiple threads are allowed to simultaneously read from a shared resource. Still, only one writer should be allowed in the critical region. Java supports dual Read/Write locks as shown below:

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();

rwLock.readLock().lock();
    // multiple readers can enter this section
    // (as long as no writer has acquired the lock)
rwLock.readLock().unlock();


rwLock.writeLock().lock();
    // only one writer can enter this section,
    // (as long as no current readers locking)
rwLock.writeLock().unlock();
```

Reference:
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

Concurrency Control

Java JDK Semaphores

Counting Semaphores maintain a pool of *n* permits. They can act as a gate guardian that allows up to *n* threads at a time, as well as (2) a mechanism for sending signals between two threads.

In the fragment below a semaphore reserves up to *n* permits. A thread trying to enter the critical section will first try to acquire *n1* of the remaining passes, if all of the n1 are obtained it enter the critical section, and then release *n2* passes. If all requested passes cannot be obtained the thread waits in the semaphore until they become available (Caution: starvation, seniority rights)

```
int n = 1;
Semaphore semaphore = new Semaphore(n);

semaphore.acquire(n1);
    // put your critical code here
semaphore.release(n2);
```

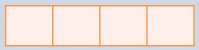
Concurrency Control

Java JDK BlockingQueues

The **BlockingQueue** class exposes a synchronized queue to any number of producers and consumers. It is implemented using one of the following concrete classes: ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, PriorityBlockingQueue, and SynchronousQueue.

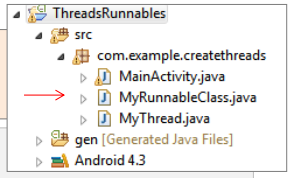
```
ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(4);
Producer producer = new Producer(queue);
Consumer consumer = new Consumer(queue);
new Thread(producer).start();
new Thread(consumer).start();
```

Producer(s)	BlockingQueue	Consumer(s)

`queue.put("Data1");`
`...`
`queue.put("Data2");`
		`queue.take()`
		`...`
		`queue.take()`


Concurrency Control

Example1. A Complete Android Example Creating Two Threads




```
public class MainActivity extends Activity {
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

1       Runnable myRunnable1 = new MyRunnableClass();
        Thread t1 = new Thread(myRunnable1);
        t1.start();

2       MyThread t2 = new MyThread();
        t2.start();

    }//onCreate
```

1. (Style1) Create a common Thread, pass a custom Runnable.
2. (Style2) Create a custom Thread, override its run() method.

Concurrency Control

Example1. Creating Threads – Implementing Runnable

```
1
public class MyRunnableClass implements Runnable {
   @Override
   public void run() {
      try {
         for (int i = 100; i < 105; i++){
2           Thread.sleep(1000);
            Log.e ("t1:<<runnable>>", "runnable talking: " + i);
         }
      } catch (InterruptedException e) {
          Log.e ("t1:<<runnable>>",  e.getMessage() );
      }

   }//run

}//class
```

1. You need to *implement* the Runnable interface and provide a version of its mandatory run() method.
2. Thread.sleep(1000) fakes busy work, the thread sleeps 1000 milisec. (see **LogCat**)

Concurrency Control

Example1. Creating Threads – A Custom Thread

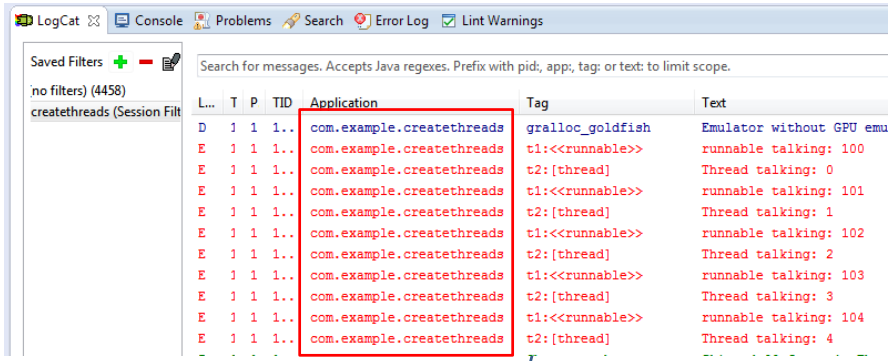
```
1
public class MyThread extends Thread{

   @Override
   public void run() {
      super.run();
      try {
         for(int i=0; i<5; i++){
2           Thread.sleep(1000);
            Log.e("t2:[thread]", "Thread talking: " + i);
         }
      } catch (InterruptedException e) {
          Log.e("t2:[thread]", e.getMessage() );
      }
   }//run
}//MyThread
```

1. You need to *extend* the Thread class and provide a version of its mandatory run() method.
2. Thread,sleep(1000) fakes busy work, the thread sleeps 1000 milisec. (see **LogCat**)

Concurrency Control

Example1. Creating Threads – Testing

Creating (executing) two threads using different programming styles.



Interleaved execution of threads: t1 and t2. Both are part of the CreateThreads process

Concurrency Control

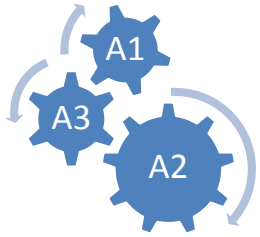
Advantages of Multi-threading

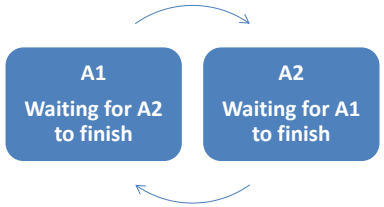
1. The various functional components of an application could be abstracted around the notion of serial or parallel actions.
2. Serial actions could be implemented using common class methods, while parallel activity could be assigned to independent threads.
3. Threads could share the data resources held in the process that contain them.
4. Responsive applications can be easily created by placing the logic controlling the user's interaction with the UI in the application's main thread, while slow processes can be assigned to background threads.
5. A multithreaded program operates ***faster*** on computer systems that have ***multiple CPUs***. Observe that most current Android devices do provide multiple processors.

Concurrency Control

Disadvantages of Multi-threading

1. Code tends to be more **complex**
2. Need to detect, avoid, resolve **deadlocks**





Concurrency Control

Android's Strategies for Execution of Slow Activities

Problem: An application may involve the use of a time-consuming operation. When the slow portion of logic executes the other parts of the application are blocked.

Goal: We want the **UI** (and perhaps other components) to be responsive to the user in spite of its heavy load.

Solution: Android offers two ways for dealing with this scenario:

1. Do expensive operations in a background ***service***, using *notifications* to inform users about next step.
2. Do the slow work in a ***background thread***.

Using Threads: Interaction between Android threads (Main and background) is accomplished using

(a) a main thread ***Handler*** object and

(b) posting ***Runnable*** objects to the main view.

Concurrency Control

Android's Handler class

- The main thread may use its ***MessageQueue*** to manage interactions between the main and background threads it creates.
- The message queue acts as a semaphore protected priority-queue with the capacity to enqueue tokens containing messages or runnables sent by the secondary threads.
- By protocol, children threads must request empty tokens from the ancestor's queue, fill them up, and then send back to the parent's queue.
- In Android's architecture each thread has a MessageQueue. To use it, a **Handler** object must be created.
- The Handler will enqueue *messages* and *runnables* to the parent's message queue. Those requests will later be execute in the order in which they are removed (dequeue) from the message queue.

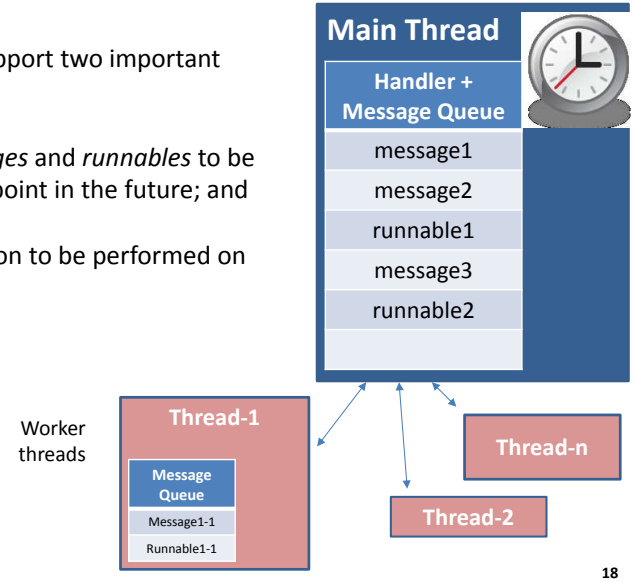
Concurrency Control

Android's Handler class

A **Handler** is used to support two important operations:

(1) to schedule *messages* and *runnables* to be executed as some point in the future; and

(2) to enqueue an action to be performed on another thread



Concurrency Control

Threads cannot touch the app's UI

Warning

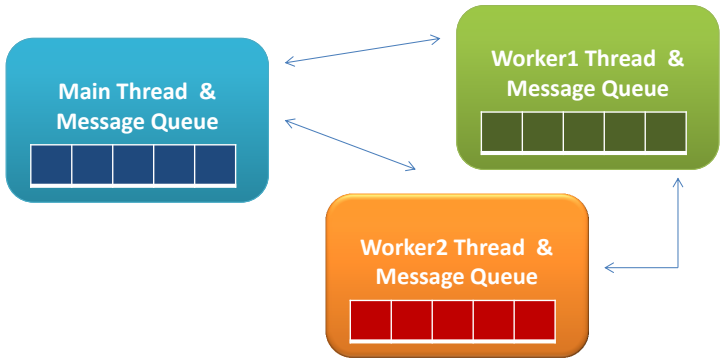
Android's background threads are not allowed to interact with the UI.

- Only the main process can access the activity's view and interact with the user. Consequently all input/output involving what the user sees or supplies must be performed by the main thread.
- A simple experiment. Add a Toast message to the run() methods implemented in Example1. Both should fail!
- Class variables (defined in the Main thread) can be seen and updated by the threads

Concurrency Control

Inter-Thread Communications.

- Typically the main UI thread sets a handler to get messages from its worker threads; however *each worker thread could also define its own handler*.
- A handler in the worker thread creates a local message-queue which could be used to receive messages from other threads (including main).



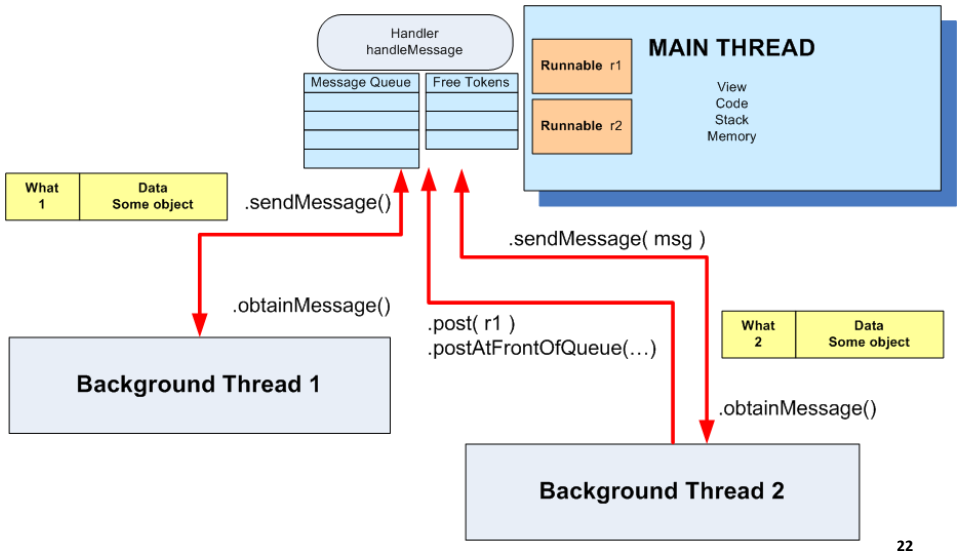
Concurrency Control

Android's Handler-Message Protocol

1. A *background-to-foreground* thread communication is initiated by the background worker (producer) by requesting a message **token** from the main thread (consumer). The ***obtainMessage*()** method is used to negotiate the acquisition of the token, which acts as a special envelope with various pre-defined compartments for data to be inserted.
2. After the empty token is received, the background thread can enter its local data into the message **token.** Local data could be anything ranging from a few numeric values to any custom object. Finally the token is attached to the Handler's **message queue** using the ***sendMessage*()** method.
3. The consumer's Handler uses the ***handleMessage*()** method to listen for new messages arriving from the producers.
4. A **message** taken from the queue to be serviced, could either
   - Pass some **data** to the main activity or
   - Request the **execution** of runnable objects through the ***post*()** method.

Concurrency Control

Android's Handler-Message Architecture



Concurrency Control

Handler. Using Messages

Main Thread

```
...
Handler myHandler= new Handler() {

  @Override
  public void handleMessage(Message msg) {
     // do something with the message...
     // update GUI if needed!
     ...
  }//handleMessage


};//myHandler
...
```

Background Thread

```
...
Thread backgJob = new Thread (new Runnable (){
   @Override
   public void run() {

     // do some busy work here
     // ...
     // get a token to be added to
     // the main's message queue
     Message msg= myHandler.obtainMessage();
      ...
     // deliver message to the
     // main's message-queue
     myHandler.sendMessage(msg);
   }//run

});//Thread

// this call executes the parallel thread
backgroundJob.start();
. . .
```


Concurrency Control

Handler. Using Runnables

Main Thread

```
...
 Handler     myHandler = new Handler();
 @Override
 public void onCreate(Bundle
                      savedInstanceState){
  ...
  Thread myThread1 = new Thread(
                           backgroundTask,
                           "backAlias1");
    myThread1.start();

 }//onCreate

 ...
 // this is the foreground runnable
 private Runnable foregroundTask
    = new Runnable() {
   @Override
   public void run() {
    // work on the UI if needed
          }
 ...
```

Background Thread

```
// this is the "Runnable" object
// representing the background thread

private Runnable backgroundTask
                  = new Runnable () {
     @Override
     public void run() {
         // Do some background work here
         myHandler.post(foregroundTask);

     }//run
};//backgroundTask
```

Concurrency Control

Handler: obtainMessage Method

To send a Message to a Handler, the thread must first invoke obtainMessage() to get the Message object out of the pool.

There are various versions of **obtainMessage().** They allow you to create an empty Message object, or messages holding arguments

Example

```
//  assume thread 1 produces some local data
String localData = "Greetings from thread 1";

//  thread 1 requests a message & adds localData to it
Message mgs = myHandler.obtainMessage (1, localData);
```


Concurrency Control

Handler: sendMessage Methods

There is a number of **sendMessage...()** methods that can be used by secondary threads to send messages to their corresponding primary thread.

- **sendMessage**() puts the message at the end of the queue immediately
- **sendMessageAtFrontOfQueue**() puts the message at the front of the queue immediately (versus the back, as is the default), so your message takes priority over all others
- **sendMessageAtTime**() puts the message on the queue at the stated time, expressed in the form of milliseconds based on system uptime (SystemClock.uptimeMillis())
- **sendMessageDelayed**() puts the message on the queue after a delay, expressed in milliseconds

Concurrency Control

Handler: Processing Incoming Messages

To process messages sent by the background threads, your Handler needs to implement the listener

```
handleMessage( Message msg )
```

which will be called with *each* message that appears on the message queue.

There, the handler can update the UI as needed. However, it should still do that work quickly, as other UI work is suspended until the Handler is done.

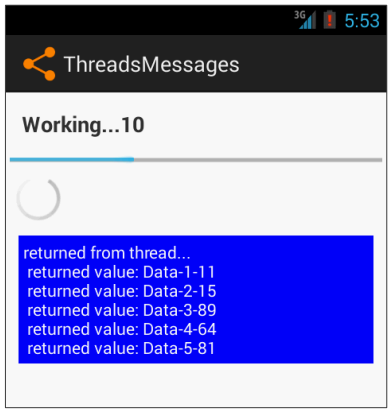
Concurrency Control

Example 2. Main-Background Communication Using Messages

In this example, the main thread presents a horizontal and a circular *progress bar widget* signaling the progress made by a slow cooperative background operation.

To simulate the job performed by the worker thread, some randomly generated result is periodically sent to the main thread.

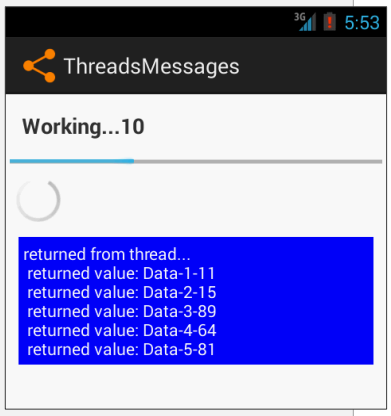
These values are used to update the app's UI and maintain the user informed of the actions realized by the background process.



Concurrency Control

Example 2. Using Handler & Messages - XML Layout

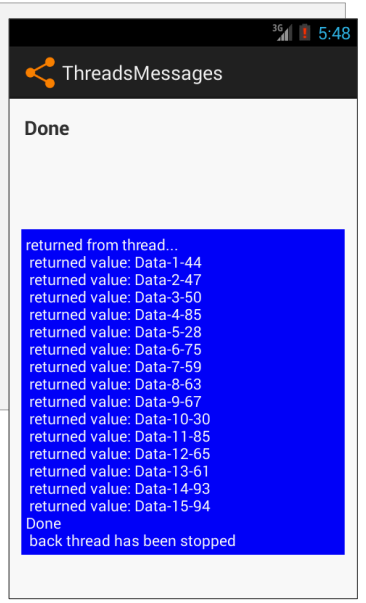
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#44ffff00"
    android:orientation="vertical"
    android:padding="4dp" >

    <TextView
        android:id="@+id/txtWorkProgress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:text="Working ...."
        android:textSize="18sp"
        android:textStyle="bold" />

    <ProgressBar
        android:id="@+id/progress1"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <ProgressBar
        android:id="@+id/progress2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```



Concurrency Control

Example 2. Using Handler & Messages - XML Layout cont. 1

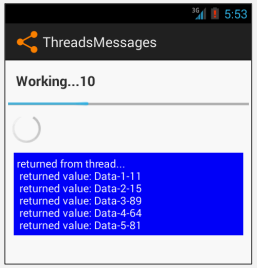
```
    <ScrollView
        android:id="@+id/myscroller"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <TextView
            android:id="@+id/txtReturnedValues"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="7dp"
            android:background="#ff0000ff"
            android:padding="4dp "
            android:text="returned from thread..."
            android:textColor="@android:color/white"
            android:textSize="14sp" />
    </ScrollView>

</LinearLayout>
```



Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java

```
public class ThreadsMessages extends Activity {

   ProgressBar bar1;
   ProgressBar bar2;

   TextView msgWorking;
   TextView msgReturned;
   ScrollView myScrollView;


   // this is a control var used by backg. threads
   protected boolean isRunning = false;

   // lifetime (in seconds) for background thread
   protected final int MAX_SEC = 30;

   // global value seen by all threads - add synchonized get/set
   protected int globalIntTest = 0;
```



Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java cont. 1

```
1  Handler handler = new Handler() {
      @Override
      public void handleMessage(Message msg) {
2        String returnedValue = (String)msg.obj;

         //do something with the value sent by the background thread here
         msgReturned.append("\n returned value: " + returnedValue );
3        myScrollView.fullScroll(View.FOCUS_DOWN);
         bar1.incrementProgressBy(1);

         //testing early termination
         if (bar1.getProgress() == MAX_SEC){
            msgReturned.append(" \nDone \n back thread has been stopped");
            isRunning = false;
         }

         if (bar1.getProgress() == bar1.getMax()){
            msgWorking.setText("Done");
4           bar1.setVisibility(View.INVISIBLE);
            bar2.setVisibility(View.INVISIBLE);
         }
         else {
            msgWorking.setText("Working..." + bar1.getProgress() );
         }
      }
   }; //handler
```

Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java cont. 2

```
@Override
public void onCreate(Bundle icicle) {
   super.onCreate(icicle);
   setContentView(R.layout.main);

   bar1 = (ProgressBar) findViewById(R.id.progress1);
5  bar1.setProgress(0);
   bar1.setMax(MAX_SEC);

   bar2 = (ProgressBar) findViewById(R.id.progress2);

   msgWorking = (TextView)findViewById(R.id.txtWorkProgress);
   msgReturned = (TextView)findViewById(R.id.txtReturnedValues);
   myScrollView = (ScrollView)findViewById(R.id.myscroller);

   // set global var (to be accessed by background thread(s) )
   globalIntTest = 1;

}//onCreate
```


Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java cont. 3

```
public void onStart() {
   super.onStart();
   // this code creates the background activity where busy work is done
   Thread background = new Thread(new Runnable() {
      public void run() {
         try {
            for (int i = 0; i < MAX_SEC && isRunning; i++) {
               // try a Toast method here (it will not work!)
               // fake busy busy work here
6              Thread.sleep(1000);  // 1000 msec.
               // this is a locally generated value between 0-100
               Random rnd = new Random();
               int localData = (int) rnd.nextInt(101);
               // we can see and change (global) class variables [unsafe!]
               // use SYNCHRONIZED get-set accessor MONITORs
               String data = "Data-" + getGlobalIntTest() + "-" + localData;
               increaseGlobalIntTest(1);
               // request a message token and put some data in it
7              Message msg = handler.obtainMessage(1, (String)data);
               // if this thread is still alive send the message
               if (isRunning) {
8                 handler.sendMessage(msg);
               }
            }
         }
```


Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java cont. 4

```
catch (Throwable t) {
          // just end the background thread
          isRunning = false;
          }
       }
    });// Tread

    isRunning = true;
    background.start();
 }//onStart

 public void onStop() {
9   super.onStop();
    isRunning = false;
 }//onStop

 // safe thread access to global var (not needed here-only one backthread!)
X public synchronized int getGlobalIntTest() {
    return globalIntTest;
 }

 public synchronized int increaseGlobalIntTest(int inc) {
    return globalIntTest += inc;
 }
}//class
```

Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java

Comments

1. The **MainActivity** creates a **Handler** object to centralize communications with a background thread that it plans to spawn.
2. The listener **handleMessage** accepts each of the messages sent by the worker class. Both have agreed on passing a string. Here **msg** -the input data object- is casted to String type.
3. Each arriving **msg** is displayed in the app's UI. The horizontal progress bar is advanced, and (if needed) the ScrollView is forced to show its last entry (which is appended at the bottom of its multiline TextView).
4. When the simulation time is over, the progress bars visibility is changed (another option we could apply is **View.GONE**, which dismisses the views and reclaims their space).
5. The maximum value the horizontal progress bar can reach is set to be **MAX_SEC.** The statement **`bar1.setProgress(0)`** moves the progress indicator to the beginning of the bar.

Concurrency Control

Example 2. Using Handler & Messages - MainActivity.java

Comments

6. The worker thread simulates busy work by sleeping 1000 miliseconds. Afterward, a randomly generated number (0-100) is produced and attached to an outgoing string. The variable `globalIntTest` defined in the main thread can be seen and changed by the back worker. After incrementing, its updated value is also attached to the outgoing message.
7. The background thread obtains an empty message token from the main's thread message queue. An basic empty message has compartments for an integer and an object. The statement `handler.obtainMessage(1, (String)data)` moves the value 1 to 'What' (the integer) and the locally produced string **data** to the object container.
8. The global variable **isRunning** becomes false when the main thread is stopped. The secondary thread checks this variable to guarantee it is not sending a message to a non-active thread.
9. When the main thread reaches its termination (onStop) it changes the boolean **isRunning** to false. Background thread uses this flag to decide whether or not to send a message. When *false* no message is delivered.

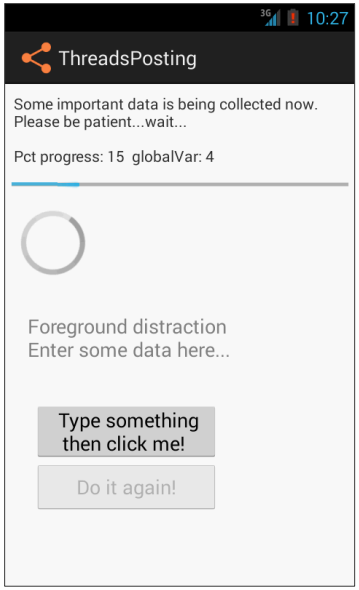
Concurrency Control

Example 3. Using Handler & Post & Runnables

We will tackle again the problem presented earlier as Example2.

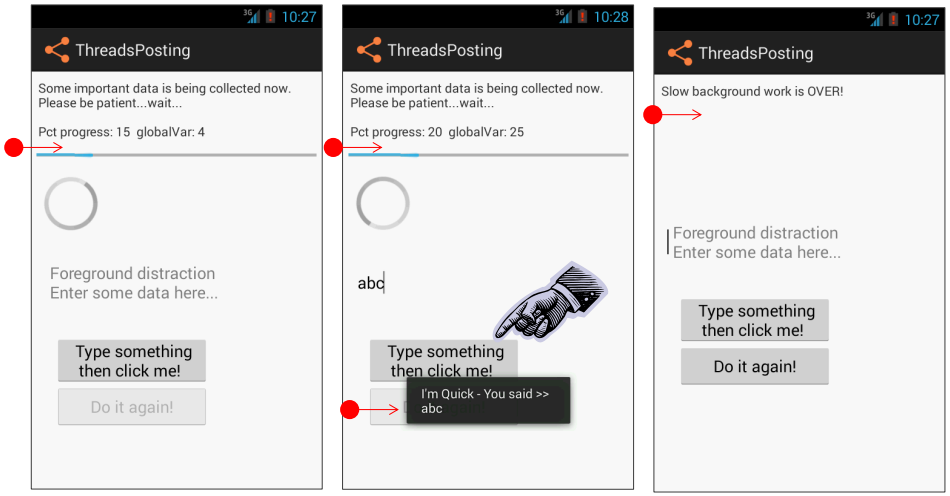
We want to emphasize two new aspects of the problem: it continues to have a slow background task but it is coupled to a fast and responsive foreground UI.

This time we will provide a solution using the **posting mechanism** to execute foreground ***runnables***.



Concurrency Control

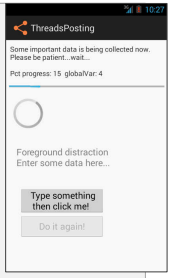
Example 3. Using Handler & Post & Runnables



Concurrency Control

Example 3. Using Handler & Post & Runnables LAYOUT 1 of 2

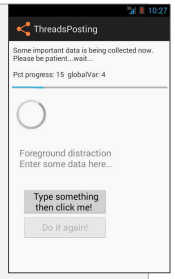
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#22002222"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/lblTopCaption"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="2dp"
        android:text=
            "Some important data is been collected now. Patience please..." />

    <ProgressBar
        android:id="@+id/myBarHor"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="30dp" />

    <ProgressBar
        android:id="@+id/myBarCir"
        style="?android:attr/progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```



Concurrency Control

Example 3. Using Handler & Post & Runnables LAYOUT 2 of 2

```
    <EditText
        android:id="@+id/txtBox1"
        android:layout_width="match_parent"
        android:layout_height="78dp"
        android:layout_margin="10dp"
        android:background="#ffffffff"
        android:textSize="18sp" />

    <Button
        android:id="@+id/btnDoSomething"
        android:layout_width="170dp"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginTop="10dp"
        android:padding="4dp"
        android:text=" Type Something Then click me! " />

    <Button
        android:id="@+id/btnDoItAgain"
        android:layout_width="170dp"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:padding="4dp"
        android:text=" Do it Again! " />

</LinearLayout>
```



Concurrency Control

Example 3. Handler & Post & Runnables - MainActivity 1 of 5

```
public class MainActivity extends Activity {
   ProgressBar myBarHorizontal;
   ProgressBar myBarCircular;

   TextView lblTopCaption;
   EditText txtDataBox;
   Button btnDoSomething;
   Button btnDoItAgain;
   int progressStep = 5;
   final int MAX_PROGRESS = 100;

   int globalVar = 0;
   int accum = 0;

   long startingMills = System.currentTimeMillis();
   boolean isRunning = false;
   String PATIENCE = "Some important data is being collected now. "
         + "\nPlease be patient...wait...\n ";

1  Handler myHandler = new Handler();

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      lblTopCaption = (TextView) findViewById(R.id.lblTopCaption);
```


Concurrency Control

Example 3. Handler & Post & Runnables - MainActivity 2 of 5

```
      myBarHorizontal = (ProgressBar) findViewById(R.id.myBarHor);
      myBarCircular = (ProgressBar) findViewById(R.id.myBarCir);

      txtDataBox = (EditText) findViewById(R.id.txtBox1);
      txtDataBox.setHint(" Foreground distraction\n Enter some data here...");

      btnDoItAgain = (Button) findViewById(R.id.btnDoItAgain);
      btnDoItAgain.setOnClickListener(new OnClickListener() {
         @Override
         public void onClick(View v) {
            onStart();
         }// onClick
      });// setOnClickListener

      btnDoSomething = (Button) findViewById(R.id.btnDoSomething);
      btnDoSomething.setOnClickListener(new OnClickListener() {
         @Override
         public void onClick(View v) {
2           String text = txtDataBox.getText().toString();
            Toast.makeText(MainActivity.this, "I'm quick - You said >> \n"
                        + text, 1).show();
         }// onClick
      });// setOnClickListener

   }// onCreate
```


Concurrency Control

Example 3. Handler & Post & Runnables - MainActivity 3 of 5

```
   @Override
   protected void onStart() {
      super.onStart();
      // prepare UI components
      txtDataBox.setText("");
      btnDoItAgain.setEnabled(false);

      // reset and show progress bars
      accum = 0;
      myBarHorizontal.setMax(MAX_PROGRESS);
      myBarHorizontal.setProgress(0);
      myBarHorizontal.setVisibility(View.VISIBLE);
      myBarCircular.setVisibility(View.VISIBLE);

      // create-start background thread were the busy work will be done
3     Thread myBackgroundThread = new Thread( backgroundTask, "backAlias1" );
      myBackgroundThread.start();

   }
```

Concurrency Control

Example 3. Handler & Post & Runnables - MainActivity 4 of 5

```
// FOREGROUND
// this foreground Runnable works on behave of the background thread,
// its mission is to update the main UI which is unreachable to back worker
private Runnable foregroundRunnable = new Runnable() {
   @Override
4  public void run() {
      try {
         // update UI, observe globalVar is changed in back thread
         lblTopCaption.setText( PATIENCE
                     + "\nPct progress: " + accum
                     + "  globalVar: " + globalVar );
         // advance ProgressBar
         myBarHorizontal.incrementProgressBy(progressStep);
         accum += progressStep;

         // are we done yet?
         if (accum >= myBarHorizontal.getMax()) {
            lblTopCaption.setText("Slow background work is OVER!");
            myBarHorizontal.setVisibility(View.INVISIBLE);
            myBarCircular.setVisibility(View.INVISIBLE);
            btnDoItAgain.setEnabled(true);
         }
      } catch (Exception e) {
         Log.e("<<foregroundTask>>", e.getMessage());
      }
   }
}; // foregroundTask
```

Foreground runnable is defined but not started !

Background thread will requests its execution later

Concurrency Control

Example 3. Handler & Post & Runnables - MainActivity 5 of 5

```
// BACKGROUND
// this is the back runnable that executes the slow work

private Runnable backgroundTask = new Runnable() {
   @Override
5  public void run() {
      // busy work goes here...
      try {
         for (int n = 0; n < 20; n++) {
            // this simulates 1 sec. of busy activity
            Thread.sleep(1000);
            // change a global variable here...
            globalVar++;
            // try: next two UI operations should NOT work
            // Toast.makeText(getApplication(), "Hi ", 1).show();
            // txtDataBox.setText("Hi ");

            // wake up foregroundRunnable delegate to speak for you
6           myHandler.post(foregroundRunnable);
         }
      } catch (InterruptedException e) {
         Log.e("<<foregroundTask>>", e.getMessage());
      }
   }// run
};// backgroundTask

}// ThreadsPosting
```

Tell foreground runnable to do something for us...

Example 3. Handler & Post & Runnables - MainActivity

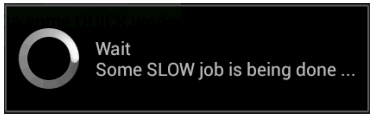
Comments

1. The **MainActivity** defines a message **Handler** to communicate with its background thread.
2. This Toast operation is used to prove that although the application is running a very slow background work, its UI is quick and responsive.
3. The background thread is created and started. We have opted for instantiating a common Thread object and passing to it a new custom Runnable (in our example: 'backgroundTask ').
4. The runnable **foregroundRunnable** will be called to act on behalf of the back worker to update the UI(which is unreachable to it). In our example the progress bar will be advanced, and the value of **globalVar** (defined in the main thread but updated by the back worker) will be displayed.
5. The back worker **backgroundTask** will simulate slow work (one second on each step). Then it will change the value of the variable **globalVar** which is part of the 'common resources' shared by both threads.
6. The command **myHandler.post(foregroundRunnable)** places a request in the main's MessageQueue for its foreground delegate to update the UI.

Concurrency Control

Using the AsyncTask Class

1. The **AsyncTask** class allows the execution of background operations and the publishing of results on the UI's thread without having to manipulate threads and/or handlers.



2. An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

3. An asynchronous task class is defined by the following Types, States, and Method

Generic Types	Main States	Auxiliary Method
Params, Progress, Result	onPreExecute, doInBackground, onProgressUpdate onPostExecute.	publishProgress

Concurrency Control

Using the AsyncTask Class

```
AsyncTask <Params, Progress, Result>
```

AsyncTask's generic types	
Params:	the type of the input parameters sent to the task at execution.
Progress:	the type of the progress units published during the background computation.
Result:	the type of the result of the background computation.

To mark a type as unused, use the type **Void**

Note:
The Java notation "**String ...**" called **Varargs** indicates an array of String values. This syntax is somehow equivalent to "**String[]**" (see Appendix B).

Concurrency Control

Using the AsyncTask Class

```
private class VerySlowTask extends AsyncTask<String, Long, Void> {

    // Begin - can use UI thread here
1   protected void onPreExecute() {

    }

    // this is the SLOW background thread taking care of heavy tasks
    // cannot directly change UI
2   protected Void doInBackground(final String... args) {
    ... publishProgress((Long) someLongValue);
    }

    // periodic updates - it is OK to change UI
    @Override
3   protected void onProgressUpdate(Long... value) {

    }

    // End - can use UI thread here
4   protected void onPostExecute(final Void unused) {

    }
}
```


Concurrency Control

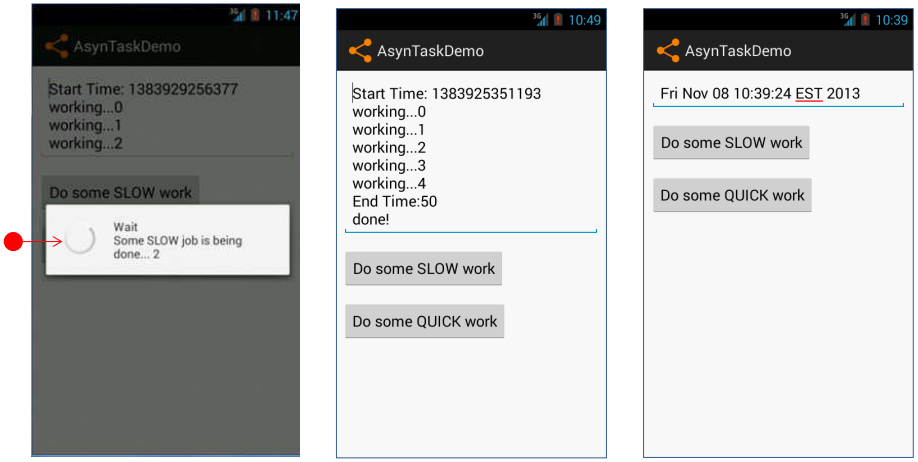
Using the AsyncTask Class

Methods
onPreExecute(), invoked on the UI thread immediately after the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
doInBackground(Params...), invoked on the background thread immediately after *onPreExecute*() finishes executing. This step is used to perform background computation that can take a long time. This step can also use *publishProgress(Progress...)* to publish one or more units of progress. These values are published on the UI thread, in the *onProgressUpdate(Progress...)* step.
onProgressUpdate(Progress...), invoked on the UI thread after a call to *publishProgress(Progress...)*. This method is used to inform of any form of progress in the user interface while the background computation is still executing.
onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Reference: http://developer.android.com/reference/android/os/AsyncTask.html

Concurrency Control

Example 4: Using the AsyncTask Class

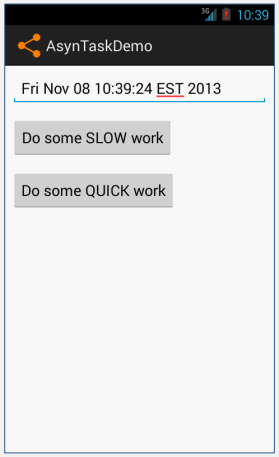


The main task invokes an **AsyncTask** to do some slow job. The AsyncTask method **doInBackgroud(…)** performs the required computation and periodically uses the **onProgressUpdate(…)** function to refresh the main's UI. In our the example, the AsyncTask manages the writing of progress lines in the UI's text box, and displays a **ProgressDialog** box.

Concurrency Control

Example 4: Using the AsyncTask Class - XML Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/txtMsg"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="7dp" />

    <Button
        android:id="@+id/btnSlow"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="7dp"
        android:text="Do some SLOW work" />

    <Button
        android:id="@+id/btnQuick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="7dp"
        android:text="Do some QUICK work" />

</LinearLayout>
```



Concurrency Control

Example 4: Using the AsyncTask Class - XML Layout

```
public class MainActivity extends Activity {
   Button btnSlowWork;
   Button btnQuickWork;
   EditText txtMsg;
   Long startingMillis;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      txtMsg = (EditText) findViewById(R.id.txtMsg);

      // slow work...for example: delete databases: “dummy1” and “dummy2”
      btnSlowWork = (Button) findViewById(R.id.btnSlow);
      this.btnSlowWork.setOnClickListener(new OnClickListener() {
         public void onClick(final View v) {
1 →         new VerySlowTask().execute("dummy1", "dummy2");
         }
      });

      btnQuickWork = (Button) findViewById(R.id.btnQuick);
      this.btnQuickWork.setOnClickListener(new OnClickListener() {
         public void onClick(final View v) {
            txtMsg.setText((new Date()).toString()); // quickly show today’s date
         }
      });
   }// onCreate
```


Concurrency Control

Example 4: Using the AsyncTask Class - XML Layout

```
   private class VerySlowTask extends AsyncTask<String, Long, Void> {
      private final ProgressDialog dialog = new ProgressDialog(MainActivity.this);
      String waitMsg = "Wait\nSome SLOW job is being done... ";

2 →   protected void onPreExecute() {
         startingMillis = System.currentTimeMillis();
         txtMsg.setText("Start Time: " + startingMillis);
         this.dialog.setMessage(waitMsg);
         this.dialog.setCancelable(false); //outside touch doesn't dismiss you
         this.dialog.show();
      }

3 →   protected Void doInBackground(final String... args) {
         // show on Log.e the supplied dummy arguments
         Log.e("doInBackground>>", "Total args: " + args.length );
         Log.e("doInBackground>>", "args[0] = " + args[0] );

         try {
            for (Long i = 0L; i < 5L; i++) {
               Thread.sleep(10000); // simulate the slow job here . . .
               publishProgress((Long) i);
            }
         } catch (InterruptedException e) {
            Log.e("slow-job interrupted", e.getMessage());
         }
         return null;
      }
```


Concurrency Control

Example 4: Using the AsyncTask Class - XML Layout

```
      // periodic updates - it is OK to change UI
      @Override
4 →   protected void onProgressUpdate(Long... value) {
         super.onProgressUpdate(value);
         dialog.setMessage(waitMsg + value[0]);
         txtMsg.append("\nworking..." + value[0]);
      }

      // can use UI thread here
5 →   protected void onPostExecute(final Void unused) {

         if (this.dialog.isShowing()) {
            this.dialog.dismiss();
         }

         // cleaning-up, all done
         txtMsg.append("\nEnd Time:"
               + (System.currentTimeMillis() - startingMillis) / 1000);
         txtMsg.append("\ndone!");
      }

   }// AsyncTask

}// MainActivity
```

Example 4: Using the AsyncTask Class

Comments

1. The **MainActivity** instantiates our AsyncTask passing dummy parameters.
2. VerySlowTask sets a ProgressDialog box to keep the user aware of the slow job. The box is defined as *not cancellable*, so touches on the UI will not dismiss it (as it would do otherwise).
3. **doInBackground** accepts the parameters supplied by the **.execute(…)** method. It fakes slow progress by sleeping various cycles of 10 seconds each. After awaking it asks the onProgressUpdate() method to refresh the ProgressDialog box as well as the user's UI.
4. The **onProgressUpdate**() method receives one argument coming from the busy background method (observe it is defined to accept multiple input arguments). The arriving argument is reported in the UI's textbox and the dialog box.
5. The **OnPostExecute**() method performs house-cleaning, in our case it dismisses the dialog box and adds a "Done" message on the UI.

Questions

Appendix A. Processes and Threads

Processes

1. A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources (memory, system's stack, ports, interruptions, semaphores, …)
2. Most operating systems support *Inter Process Communication* (IPC) resources such as pipes and sockets.
3. Most implementations of the Java virtual machine run as a single process.

Threads

1. Threads exist within a process. Threads share the process's resources (including memory).
2. Every process has at least one thread (called *Main* thread).
3. Each thread has the ability to create additional threads.

Reference: http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

Appendix B. Java Varargs Example

What for?

The clause **(Type ellipsis … varargs)** plays a role in facilitating the creation of Java methods accepting a variable number of arguments all of the same type. It provides for a more flexible method calling approach, as shown in the example below.

```
public  void sum(Integer... items) {
  int sum = 0;
  for (int i = 0; i < items.length; i++) {
    sum += items[i];
  }
  Log.e("SUM", "The sum is " + sum);
}
```

The **sum** method accepts a **Varargs** of Integer values. It could be called with

```
sum(1, 2, 3, 4);
```

or alternatively

```
sum(new Integer[] {1, 2, 3, 4} );
```

Clearly the syntax used in the first call is simpler.

Concurrency Control

Appendix C. Temporary Relief From Android's Watchful Eye

Looking the other way

Your application's main thread should remain responsive at all times, failure to do so generates dreaded ANR dialog boxes (**A**pplication **N**ot **R**esponding).

However you may briefly escape from your obligation to write well behaved, quick responding apps (as it may happen in the rush to test an idea *you know well* will be caught by the Activity Monitor as unacceptably slow). To do so, temporarily disable the system's monitoring by adding to your activity the following code fragment.

```
StrictMode.ThreadPolicy policy = new
                StrictMode.ThreadPolicy.Builder().permitAll().build();

StrictMode.setThreadPolicy(policy);
```

Please notice this is an *extremely poor* remedy and should be replaced by a better strategy such as using Threads, AsyncTasks, or Background Services.