Thread

Lecture 12
Threads

• A thread is a concurrent unit of execution
• Each thread has its own call stack for methods being invoked, their arguments and local variables.
• Each virtual machine instance has at least one main Thread running when it is started; typically, there are several others for housekeeping.
• The application might decide to launch additional threads for specific purposes.
Threads

• Threads in the same VM interact and synchronize by the use of shared objects and monitors associated with these objects.

• There are basically two main ways of having a thread execute application code.
  – Create a new class that extends Thread and override its run() method.
  – Create a new Thread instance passing to it a Runnable object.

• In both cases, the start() method must be called to actually execute the new Thread.
Two Techniques

class MyRunnable implements Runnable {
    public void run() {
        for(int i=0; i<5; i++){
            System.out.println(i);
        }
    }
}

Thread t = new Thread(new MyRunnable()).start();

class MyRunnableThread extends Thread {
    public void run() {
        for(int i=0; i<5; i++){
            System.out.println(i);
        }
    }
}

Thread t = new MyRunnableThread().start();
Difference

What is the difference between implementing Runnable and extending Thread?

• “One difference between implementing Runnable and extending Thread is that by extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same object instance.”

• “In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.”
public class Program {
    public static void main (String[] args) {
        Runner r = new Runner();
        Thread t1 = new Thread(r, "Thread A");
        Thread t2 = new Thread(r, "Thread B");
        Thread s1 = new Strider("Thread C");
        Thread s2 = new Strider("Thread D");
        t1.start();
        t2.start();
        s1.start();
        s2.start();
    }
}
public class Runner implements Runnable {
    private int counter;
    public void run() {
        try {
            for (int i = 0; i != 2; i++) {
                System.out.println(Thread.currentThread().getName() + ":: " + counter++);
                Thread.sleep(1000);
            }
        } // for loop
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class Strider extends Thread {
    private int counter;
    Strider(String name) {
        super(name);
    }
    public void run() {
        try {
            for (int i = 0; i != 2; i++) {
                System.out.println(Thread.currentThread().getName() + ": ": counter++);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
Multi-Threading

**Process 1** (Dalvik Virtual Machine 1)

- Common memory resources
- Main thread
- Thread-1
- Thread-2

**Process 2** (Dalvik Virtual Machine 2)

- Common memory resources
- main thread
Multi-Threading

Advantages of Multi-Threading

• Threads share the process' resources but are able to execute independently.

• Applications responsibilities can be separated
  • main thread runs UI, and
  • slow tasks are sent to background threads.

• Threading provides an useful abstraction of concurrent execution.

• Particularly useful in the case of a single process that spawns multiple threads on top of a multiprocessor system. In this case real parallelism is achieved.

• Consequently, a multithreaded program operates faster on computer systems that have multiple CPUs.
Multi-Threading

• Multithreading:
  – A thread is a single sequence of execution within a program
  – Refers to multiple threads of control within a single program
  – Each program can run multiple threads of control

• Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
Concurrent multithreading systems give the appearance of several tasks executing at once, but these tasks are actually split up into chunks that share the processor with chunks from other tasks.

In parallel systems, two tasks are actually performed simultaneously. Parallelism requires a multi-CPU system.
Multi-Threading

• Disadvantages of Multi-Threading
  – Code tends to be more complex
  – Need to detect, avoid, resolve **deadlocks**
Android‘s Approach to Slow Activities

An application may involve a time-consuming operation, however we want the UI to be responsive to the user. Android offers two ways for dealing with this scenario:

– Do expensive operations in a background service, using notifications to inform users about next step

– Do the slow work in a background thread.

Interaction between Android threads is accomplished using (a) Handler objects and (b) posting Runnable objects to the main view.
Handler Class

• When a process is created for your application, its *main thread* is dedicated to running a *message queue* that takes care of managing the top-level application objects (activities, intent receivers, etc) and any windows they create.
• You can create your own secondary threads, and communicate back with the main application thread through a **Handler**.
• When you create a new Handler, it is bound to the message queue of the thread that is creating it -- from that point on, it will deliver *messages* and *runnables* to that message queue and execute them as they come out of the message queue.
Handler Class

There are two main uses for a Handler:
(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
Threads and UI

Warning

• Background threads are not allowed to interact with the UI.
• Only the main process can access the (main) activity’s view.
• (Global) class variables can be seen and updated in the threads
Handler‘s MessageQueue

• A secondary thread that wants to communicate with the main thread must request a message token using the `obtainMessage()` method.

• Once obtained, the background thread can fill data into the message token and attach it to the Handler‘s message queue using the `sendMessage()` method.

• The Handler uses the `handleMessage()` method to continuously attend new messages arriving to the main thread.

• A message extracted from the process‘ queue can either return some data to the main process or request the execution of runnable objects through the `post()` method.
Multi-threading

Handler
handleMessage

Message Queue
Free Tokens

Runnable r1
Runnable r2

View
Code
Stack
Memory

What 1
Data
Some object

.sendMessage()
.obtainMessage()

Background Thread 1

.post(r1)
.postAtFrontOfQueue(...)

What 2
Data
Some object

.sendMessage(msg)
.obtainMessage()

Background Thread 2
Multi-threading

Using Messages

**Main Thread**

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

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

    //handleMessage
}
```

**Background Thread**

```java
Thread backJob = 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();
```
Multi-threading

Using Post

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
    }
} //run

Background Thread
// this is the "Runnable" object
// that executes the background thread
private Runnable backgroundTask
    = new Runnable () {
    public void run() {
        ... Do some background work here
    }
} //backgroundTask

myHandler.post(foregroundTask); //run
}; //backgroundTask
Messages

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

• There are a few forms of `obtainMessage()`, allowing you to just create an empty Message object, or messages holding arguments.

• Example

  // thread 1 produces some local data
  String localData = “Greeting from thread 1”;

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

You deliver the message using one of the sendMessage... () family of methods, such as ...

• 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
Processing Messages

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

    handleMessage(...)  

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.
Example

ThreadProgressBar – using message passing

• The main thread displays a horizontal and a circular progress bar widget showing the progress of a slow background operation.

• Some random data is periodically sent from the background thread and the messages are displayed in the main view.
public class MainActivity extends AppCompatActivity {
    ProgressBar bar1;
    ProgressBar bar2;
    TextView msgWorking;
    TextView msgReturned;

    boolean isRunning = false;
    final int MAX_SEC = 60; // (seconds) lifetime for background thread
    String strTest = "global value seen by all threads ";
    int intTest = 0;
Handler handler = new Handler() {

    @Override
    public void handleMessage(Message msg) {
        String returnedValue = (String) msg.obj;
        //do something with the value sent by the background thread here ...

        msgReturned.setText("returned by background thread: \n\n" + returnedValue);
        bar1.incrementProgressBy(2);

        //testing thread’s termination
        if (bar1.getProgress() == MAX_SEC) {
            msgReturned.setText("Done \n back thread has been stopped");
            isRunning = false;
        }
        if (bar1.getProgress() == bar1.getMax()) {
            msgWorking.setText("Done");
            bar1.setVisibility(View.INVISIBLE);
            bar2.setVisibility(View.INVISIBLE);
            bar1.setLayoutParams().height = 0;
            bar2.setLayoutParams().height = 0;
        } else {
            msgWorking.setText("Working..." + bar1.getProgress());
        }
    }
};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    bar1 = (ProgressBar) findViewById(R.id.progress);
    bar2 = (ProgressBar) findViewById(R.id.progress2);
    bar1.setMax(MAX_SEC);
    bar1.setProgress(0);
    msgWorking = (TextView) findViewById(R.id.TextView01);
    msgReturned = (TextView) findViewById(R.id.TextView02);
    strTest += "-01"; // slightly change the global string
    intTest = 1;
}

@Override
protected void onStop() {
    super.onStop();
    isRunning = false;
}
@Override
protected void onStart()
{
    super.onStart();
    // bar1.setProgress(0);
    Thread background = new Thread(new Runnable()
    {
        public void run()
        {
            try {
                for (int i = 0; i < MAX_SEC && isRunning; i++) {
                    //try a Toast method here (will not work!)
                    //fake busy work here
                    Thread.sleep(1000); //one second at a time
                    Random rnd = new Random();
                    String data = "Thread Value: " + (int) rnd.nextInt(101);
                    //we can see and change (global) class variables
                    data += "\n" + strTest + " " + intTest;
                    intTest++;
                    //request a message token and put some data in it
                    Message msg = handler.obtainMessage(1, (String)data);
                    // if thread is still alive send the message
                    if (isRunning) {
                        handler.sendMessage(msg);
                    }
                }
            } catch (Throwable t) {
                // just end the background thread
            }
        }
    }); //background
    isRunning = true;
    background.start();
} //onStart
Multi-threading

ThreadPost -- Using Handler post method

• We will try the similar problem presented earlier (a slow background task and a responsive foreground UI) this time using the posting mechanism to execute foreground runnables.

• Demo ThreadPost APP
public class MainActivity extends AppCompatActivity {

    ProgressBar myBar;
    TextView lblTopCaption;
    EditText txtBox1;
    Button btnDoSomething;

    int globalVar = 0; // to be used by threads to exchange data

    int accum = 0;
    long startingMills = System.currentTimeMillis();
    boolean isRunning = false;
    String PATIENCE = "Some important data is being collected now. " +
    "\nPlease be patient...wait... ";

    Handler myHandler = new Handler();
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    lblTopCaption = (TextView) findViewById(R.id.lblTopCaption);
    myBar = (ProgressBar) findViewById(R.id.myBar);
    myBar.setMax(100); // range goes from 0..100
    txtBox1 = (EditText) findViewById(R.id.txtBox1);
    txtBox1.setHint("Foreground distraction. Enter some data here");
    btnDoSomething = (Button) findViewById(R.id.btnDoSomething);
    btnDoSomething.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Editable txt = txtBox1.getText();
            Toast.makeText(getBaseContext(), "You said >> " + txt, Toast.LENGTH_LONG).show();
        }//onClick
    }); //setOnClickLister
}
ThreadPost -- Using Handler post method

@Override
protected void onStart(){
    super.onStart();

    // create & execute background thread were the busy work will be done
    Thread myThreadBack = new Thread(backgroundTask, "backAlias1");
    myThreadBack.start();
    myBar.incrementProgressBy(0);
}

private Runnable foregroundTask = new Runnable() {

    @Override
    public void run() {
        try {
            int progressStep = 5;
            double totalTime = (System.currentTimeMillis() - startingMills) / 1000;
            synchronized(this) {
                globalVar += 100;
            }

            lblTopCaption.setText(PATIENCE + totalTime + " -- " + globalVar);
            myBar.incrementProgressBy(progressStep);
            accum += progressStep;
            if (accum >= myBar.getMax()) {
                lblTopCaption.setText("Background work is OVER!");
                myBar.setVisibility(View.INVISIBLE);
            }
        } catch (Exception e) {
            Log.e("<<foregroundTask>>", e.getMessage());
        }
    }
}; //foregroundTask
/this is the "Runnable" object that executes the background thread
private Runnable backgroundTask = new Runnable () {
    @Override
    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);
                // now talk to the main thread
                // optionally change some global variable such as: globalVar

                    synchronized(this) {
                        globalVar += 1;
                    }
                    myHandler.post(foregroundTask);
            }
        } catch (InterruptedException e) {
            Log.e("<<foregroundTask>>", e.getMessage());
        }
    //run
}; //backgroundTask