MULTI-THREADING

Thread

- A Thread is a **concurrent** unit of execution.
- The 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.

Creating Threads (method 1)

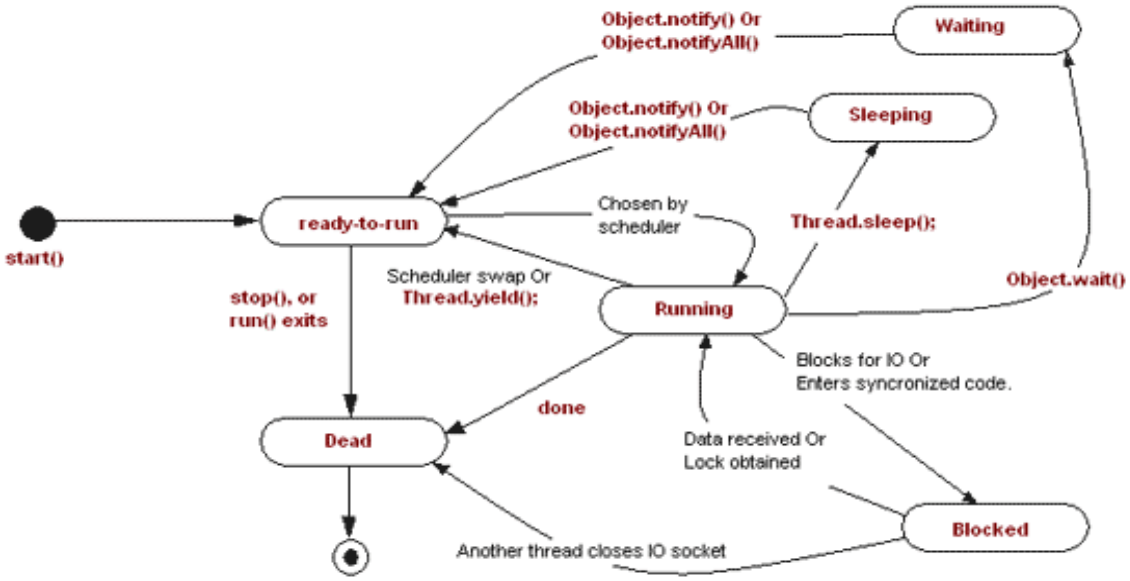
- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

Creating Threads (method 2)

- **Create thread by implementing Runnable interface:**
  - The simplest way to make a thread is to make a class that implements the runnable interface.
  - To implement the runnable, a class require only implement a single method called 'run( )', which is declared as following : `public void run( )`
  - After you make a class that implements 'Runnable', you will instantiate an Object of type 'Thread' from within that class. The thread defines various constructors. The one that we shall use is representing here: `Thread(Runnable threadObj, String threadname);`
  - After the new thread is made, it will not start running until you call it 'start( )' method, which is declared within the Thread. The 'start( )' method is representing here: `void start( );`

Thread State

- Thread State



Thread.State	Description
BLOCKED	The thread is blocked and waiting for a lock.
NEW	The thread has been created, but has never been started.
RUNNABLE	The thread may be run.
TERMINATED	The thread has been terminated.
TIMED_WAITING	The thread is waiting for a specified amount of time.
WAITING	The thread is waiting.

Multithreading

- A multithreading is a specialized form of multitasking. Multithreading requires less overhead than multitasking processing.

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.*

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:

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*.

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

Main thread

- All Android application components run on the main application thread
  - *Activities, Services, and Broadcast Receivers*
- Time-consuming
- Blocking operation
- In any component
  - *will block all other components including Services and the visible Activity*

Time-consuming & blocking operations

- File operations
- Network lookups
- Database transactions
- Complex calculations
- etc

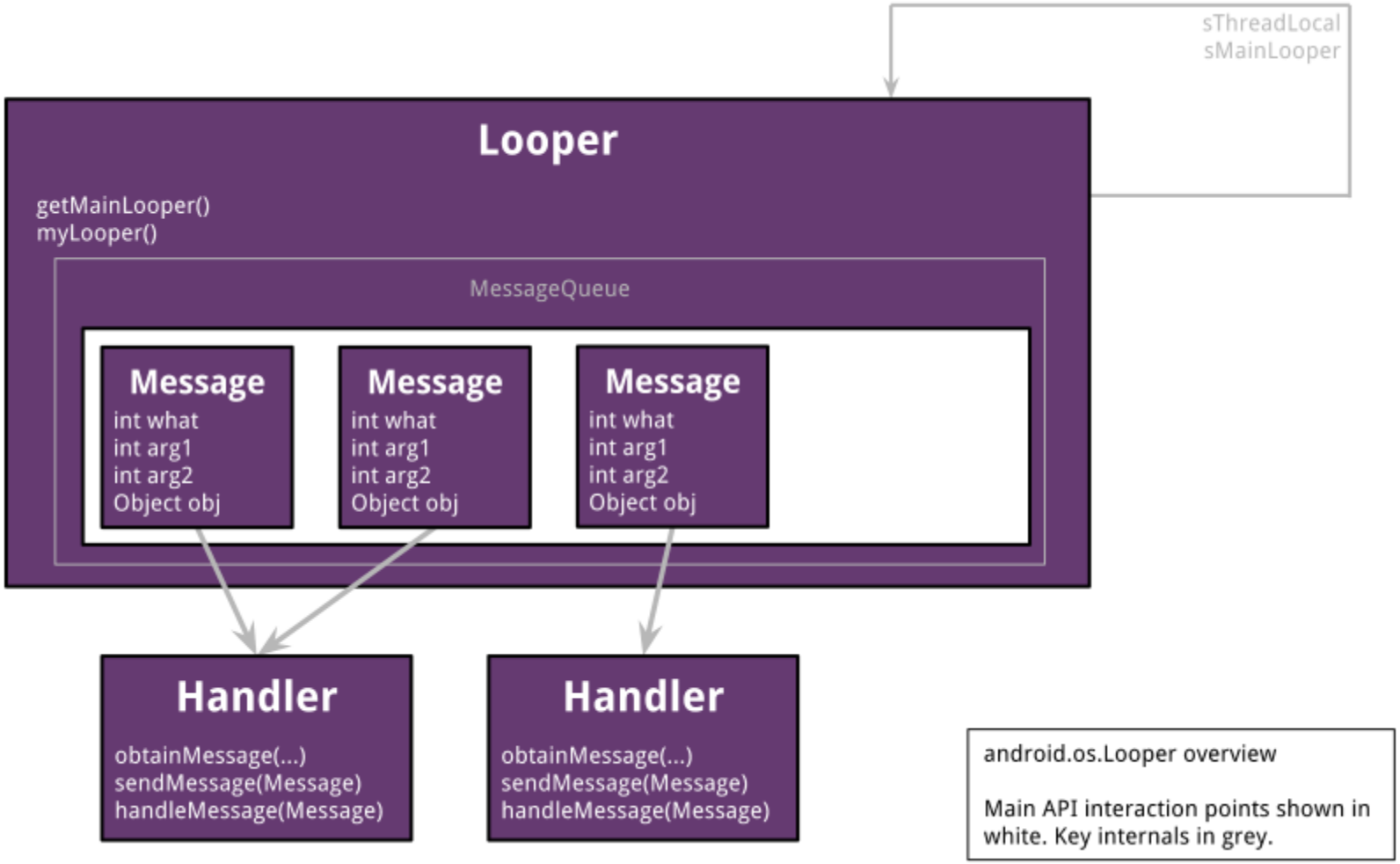
Unresponsive program

- Android OS save himself from application does not response for input events.
- Unresponsive program
- Activities
  - within 5 seconds
- Broadcast Receivers
  - onReceive handlers within 10 seconds.

Unresponsive Exception

Inter-Thread Communication

Looper and Handler



Looper

- Wrapper class to attach a message queue to a thread and manage this queue
- •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, and then loop() to have it process messages until the loop is stopped
- •The main thread (a.k.a. UI thread) in an Android application sets up as a looper thread before your application is created.
- •Most interaction with a message loop is through the Handler class.

Handler

- A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue.
- Each Handler instance is associated with a single thread and that thread's message queue.
- When you create a new Handler, it is bound to the thread / 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.

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

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.

Send Message to MessageQueue

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

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);
```

- 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.

Multi-Threading

Using Post

Main Thread

```
...
Handler myHandler = new Handler(); // this is the "Runnable" object
@Override                          // that executes the background thread
public void onCreate(
            Bundle savedInstanceState) { private Runnable backgroundTask
                                                = new Runnable () {
  ...                                     @Override
 Thread myThread1 =                       public void run() {
          new Thread(backgroundTask,        ... Do some background work here
                          "backAlias1");     myHandler.post(foregroundTask);
 myThread1.start();
                                           }//run
}//onCreate
                                        };//backgroundTask
...
//this is the foreground runnable
private Runnable foregroundTask
    = new Runnable() {
  @Override
  public void run() {
    // work on the UI if needed
}
...
```

Background Thread

Multi-Threading

Using Post

Main Thread

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

    @Override
    public void handleMessage(Message msg) {

        // do something wi h the message... t
        // 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(); ...
```

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

Example 1. Progress Bar - 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.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
android:id="@+id/widget28"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:background="#ff009999" android:orientation="vertical"
xmlns:android="http://schemas.android.com/apk/res/android" >

<TextView
android:id="@+id/TextView01"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Working ...."
android:textSize="18sp"
android:textStyle="bold" />
<ProgressBar
android:id="@+id/progress"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
style="?android:attr/progressBarStyleHorizontal" />
```

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

<TextView
android:id="@+id/TextView02"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="returned from thread..."
android:textSize="14sp"
android:background="#ff0000ff"
android:textStyle="bold"
android:layout_margin="7px"/>

</LinearLayout>
```

Multi-Threading

- **Progress Bar – Using Message Passing**

```
import java.util.Random;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.View;
import android.widget.ProgressBar;
import android.widget.TextView;

public class ThreadDemo1ProgressBar extends Activity {
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;
```

Multi-Threading

- Progress Bar – Using Message Passing

```
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.getLayoutParams().height = 0; bar2.getLayoutParams().height = 0;
}
else {
msgWorking.setText("Working..." +
bar1.getProgress() );
}
}
}; //handler
```

Multi-Threading

- Progress Bar – Using Message Passing

```
@Override
public void onCreate(Bundle icicle) {

super.onCreate(icicle);
setContentView(R.layout.main);

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;


}//onCreate

 public void onStop() {

super.onStop();
isRunning = false;
}
```

Multi-Threading

```
public 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 busy work here
Thread.sleep(1000); //one second at a time
Random rnd = new Random();

// this is a locally generated value
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
}
}//run
});//background
isRunning = true;
background.start();
}//onStart
} //class
```

ASYNCTASK

AsyncTask

- The AsyncTask class encapsulates the creation of Threads and Handlers.
- To use AsyncTask you must subclass it. AsyncTask uses generics and varargs. The parameters are the following AsyncTask <TypeOfVarArg Params , ProgressValue , ResultValue>
- AsyncTask has four steps:
- onPreExecute (Invoked on UI Thread)
  - is invoked before the execution.
- doInBackground (Invoked on background Thread)
  - the main operation. Write your heavy operation here.
- onPostExecute (Invoked on UI Thread)
  - is invoked after the execution.
- onProgressUpdate (Invoked on UI Thread)
  - Indication to the user on progress. It is invoked every time publishProgress() is called.

AsyncTask

- An AsyncTask is started via the execute() method.
- The execute() method calls the doInBackground() and the onPostExecute() method.
- The doInBackground() method contains the coding instruction which should be performed in a background thread. This method runs automatically in a separate Thread.
- The onPostExecute() method synchronizes itself again with the user interface thread and allows it to be updated. This method is called by the framework once the doInBackground() method finishes.

AsyncTask - Example

```
public class Main extends Activity {
Button btnSlowWork;
Button btnQuickWork; EditText etMsg;
Long startingMillis;
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        etMsg = (EditText) findViewById(R.id.EditText01);
        btnSlowWork = (Button) findViewById(R.id.Button01);
// slow work...for example: delete all data from a database or get data from Internet
this.btnSlowWork.setOnClickListener(new OnClickListener() {
public void onClick(final View v) {
new VerySlowTask().execute();
        } });
btnQuickWork = (Button) findViewById(R.id.Button02);
// delete all data from database (when delete button is clicked)
this.btnQuickWork.setOnClickListener(new OnClickListener() {
public void onClick(final View v) {
etMsg.setText((new Date()).toLocaleString());
}
});
}// onCreate
```

AsyncTask - Example

```
private class VerySlowTask extends AsyncTask <String, Long, Void>
      {
      private final ProgressDialog dialog = new ProgressDialog(Main.this);
        // can use UI thread here
        protected void onPreExecute()
        {
                startingMillis = System.currentTimeMillis();
               etMsg.setText("Start Time: " + startingMillis);
              this.dialog.setMessage("Wait\nSome SLOW job is being done...");
               this.dialog.show();
        }
    // automatically done on worker thread (separate from UI thread)
      protected Void doInBackground(final String... args)
       {
   try {
        // simulate here the slow activity
         for (Long i = 0L; i < 3L; i++)
          {
                Thread.sleep(2000);
            publishProgress((Long)i);
        }
}
catch (InterruptedException e)
 {
    Log.v("slow-job interrupted", e.getMessage())
}
return null;
}
```

AsyncTask - Example

```
// periodic updates - it is OK to change UI
      @Override
      protected void onProgressUpdate(Long... value)
      {
            super.onProgressUpdate(value);
         etMsg.append("\nworking..." + value[0]);
      }
      // can use UI thread here
  protected void onPostExecute(final Void unused)
  {
        if (this.dialog.isShowing())
        {
        this.dialog.dismiss();
                }
      // cleaning-up, all done
      etMsg.append("\nEnd Time:“ + (System.currentTimeMillis()-startingMillis)/1000);
      etMsg.append("\ndone!");
      }
   }//AsyncTask
}// Main
```