Université Mohammed V
FACULTE DES SCIENCES
RABAT / FSR
Département informatique

Master IAO
Master II–Semestre 3
Cours

Mobile & Cloud Computing

Pr. REDA Oussama Mohammed

2019-2020

Android Concurrency

Handlers, Messages and Loopers

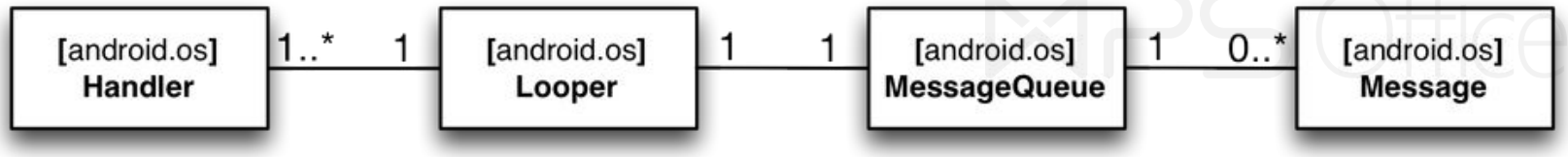
Android Thread communication

- Thread communication with regular Java and classical mechanisms—pipes, shared memory and blocking queues— impose problems for the UI thread

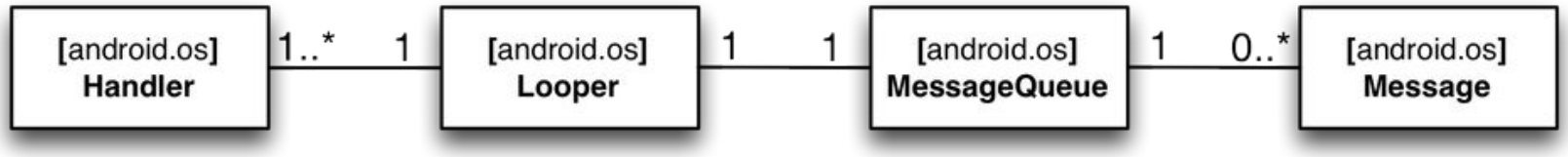
  The UI thread responsiveness is at risk because it may occasionally hang.

- Often, communication
  UI thread **<----------> worker threads.
  Hence, in Android :
  UI thread can offload long tasks by **sending data messages** be processed on **background threads**
  (*non-blocking* consumer-producer pattern).

- The message handling mechanism is fundamental in the Android platform and the API is located in the android.os package, with a set of classes shown in Figure that implement the functionality.



Handlers, Messages and Loopers



API overview.

android.os.Looper

A message dispatcher associated with the one and only consumer thread.

android.os.Handler

Consumer thread message processor, and the interface for a producer thread to insert messages into the queue. A Looper can have many associated Handlers, but they all insert messages into the same queue.

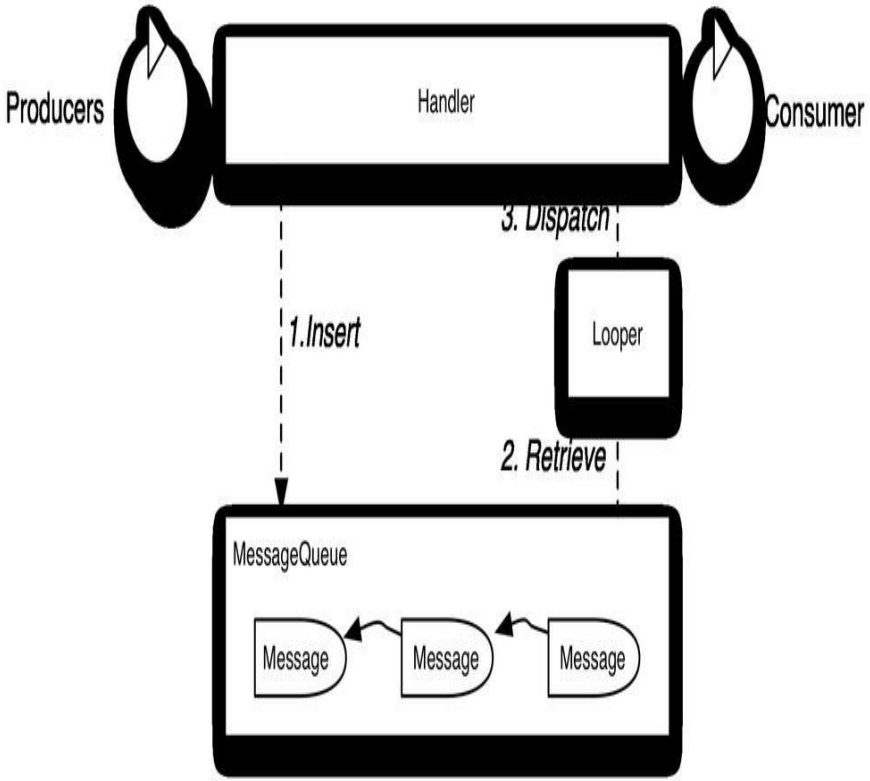
android.os.MessageQueue

Unbounded linked list of messages to be processed on the consumer thread. Every Looper—and Thread—has at most one MessageQueue.

android.os.Message

Message to be executed on the consumer thread.

Android Message Passing



Message passing mechanism between multiple producer threads and one consumer thread. Every message refers to to the next message in the queue.

Messages are inserted by producer threads and processed by the consumer thread, as illustrated in the figure.

Insert

The *producer thread inserts* messages in the queue by using the *Handler* connected to the consumer thread, as shown later.

Retrieve

The *Looper*, discussed later, runs in the consumer thread and *retrieves messages* from the queue in a sequential order.

Dispatch

The *Handler* is responsible for *processing the messages* on the consumer thread. A thread may have multiple Handler instances for processing messages; the Looper ensures that messages are dispatched to the correct Handler.

Basic Message Passing Example

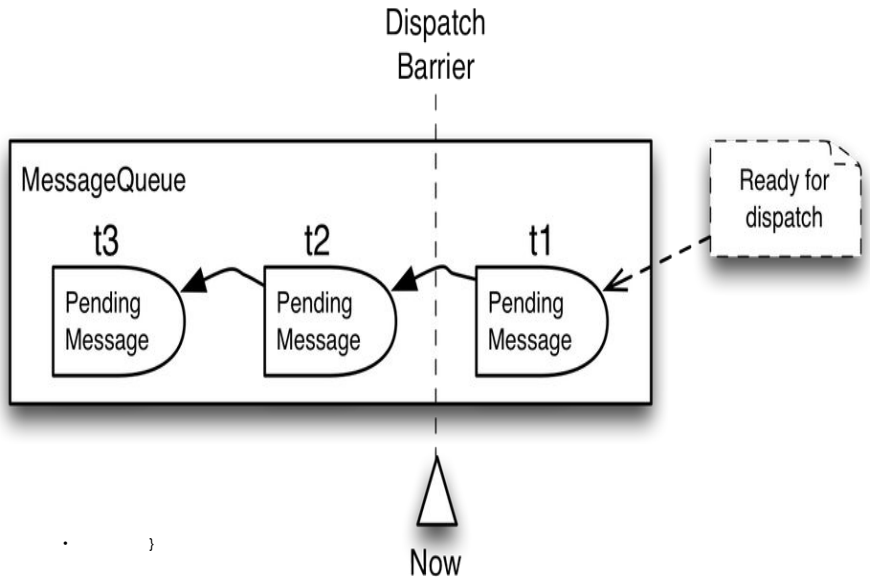
- Fundamental message passing example:
- The following code implements what is probably one of the most common use cases.
  - The user presses a button on the screen that could trigger a long operation, e.g., a network operation.
  - To avoid stalling the rendering of the UI, the long operation, represented here by a dummy doLongRunningOperation() method, has to be executed on a worker thread.
- Hence, the setup is merely one producer thread (the UI thread) and one consumer thread (LooperThread).
- Our code sets up a message queue. It handles the button click as usual in the onClick() callback, which executes on the UI thread.
- In our implementation, the callback inserts a dummy message into the message queue. For sake of brevity, layouts and UI components have been left out of the example code.

Basic Message Passing Example

```
•public class LooperActivity extends Activity {
•       LooperThread mLooperThread;

•       private static class LooperThread extends Thread{
•               public Handler mHandler;

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

•mHandler = new Handler() {
• public void handleMessage(Message msg){
• if(msg.what == 0) {
•
doLongRunningOperation();

•                               }
•                       }
•                   };

•                   Looper.loop(); }
•       }
•    public void onCreate(Bundle savedInstanceState) {
•       super.onCreate(savedInstanceState);
•       mLooperThread = new LooperThread();
mLooperThread.start();

•    }
```

```
•    public void onClick(View v) {

•       if (mLooperThread.mHandler != null) {
•Message msg = mLooperThread.mHandler.obtainMessage(0);
mLooperThread.mHandler.sendMessage(msg); }

•    }




•    private void doLongRunningOperation() {

•           // Add long running operation here.

•       }

•    protected void onDestroy() {

•       mLooperThread.mHandler.getLooper().quit(); }}
```

MessageQueue

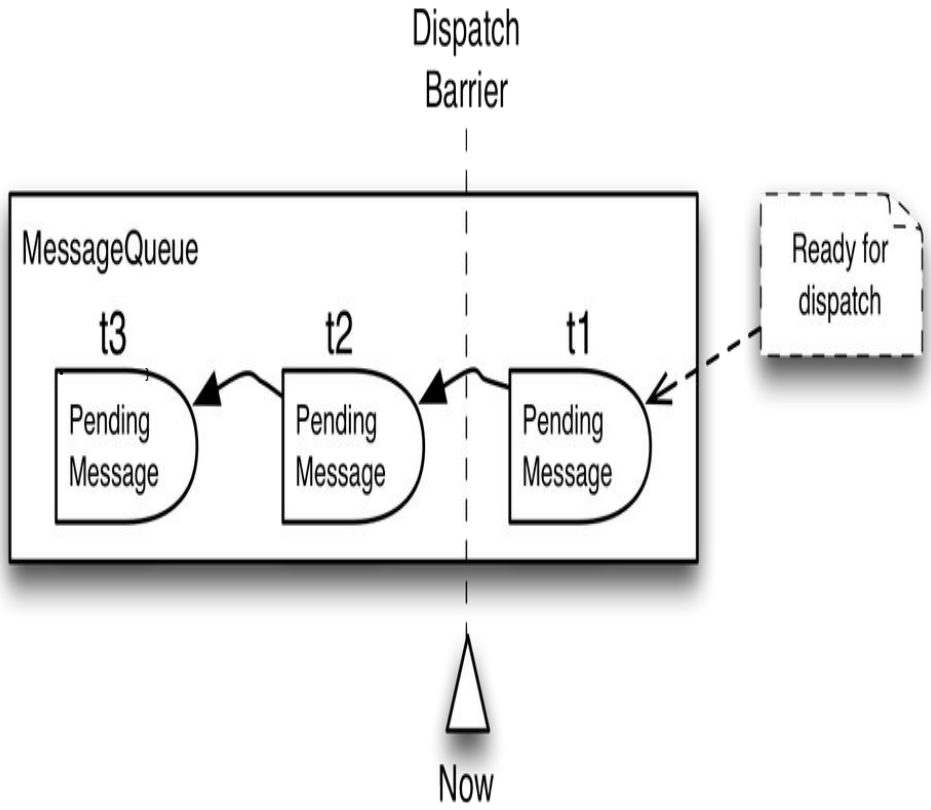


•The figure illustrates a message queue with three pending messages, sorted with timestamps where t1 < t2 < t3. Only one message has passed the dispatch barrier, i.e., the current time. Messages eligible for dispatch have a timestamp value less than the current time, i.e. "Now" in the figure.

•Pending messages in the queue. The rightmost message is first in queue to be processed.

- The message queue <=> `android.os.MessageQueue` class.
- It is a one-directional linked list.
- Producer threads insert messages that will later be dispatched to the consumer.
- The messages are sorted based on timestamps.
- The pending message with the lowest timestamp value is first in line for dispatch to the consumer. However, a message is dispatched only if the timestamp value is less than the current time. If not, the dispatch will wait until the current time has passed the timestamp value.

MessageQueue



If no message has passed the dispatch barrier when the Looper is ready to retrieve the next message, the consumer thread blocks.

Execution is resumed as soon as a message passes the dispatch barrier. The producers can insert new messages in the queue at any time and on any position in the queue.

The insert position in the queue is based on the timestamp value. If a new message has the lowest timestamp value compared to the pending messages in the queue, it will occupy the first position in the queue, which is next to be dispatched. Insertions always comply to the timestamp sorting order.

Message

- Each item on the MessageQueue is of the *android.os.Message* class.
- *android.os.Message* is a container object carrying either a data item or a task, never both.
- Data is processed by the consumer thread, whereas a task is simply executed when it is dequeued and you have no other processing to do.

Note

- The message knows its recipient processor—i.e. Handler —and can enqueue itself through *Message.sendToTarget():*

- *Message m = Message.obtain(handler, runnable);*
  m.sendToTarget();

- As we will see in Handler, the handler is most commonly used for message enqueuing, as it offers more flexibility with regard to message insertion.

Message
(Message parameters)

Parameter name	Type	Usage
what	int	Message identifier. Communicates intention of the message.
arg1, arg2	int	Simple data values to handle the common use case of handing over integers. If a maximum of two integer values are to be passed to the consumer, these parameters are more efficient than allocating a Bundle, as explained under the `data` parameter.
obj	Object	Arbitrary object. If the object is handed off to a thread in another process, it has to implement Parcelable.
data	Bundle	Container of arbitrary data values.
replyTo	Messenger	Reference to Handler in some other process. Enables inter-process message communication, as described in .
callback	Runnable	Task to execute on a thread. This is an internal instance field that holds the Runnable object from the `Handler.post` methods in .

Task message
The task is represented by a java.lang.Runnable object to be executed on the consumer thread. Task messages cannot contain any data beyond the task itself.

Message

- A MessageQueue can contain any combination of data and task messages.
- The consumer thread processes them in a sequential manner, independent of the type.
- If a message is a data message, the consumer processes the data.
- Task messages are handled by letting the Runnable execute on the consumer thread, but the consumer thread does not receive a message to be processed in Handler.handleMessage(Message), as it does with data messages.

Message construction

- Explicit object construction
- Message m = new Message();
- Factory methods
  - **Empty message**
- **Message m = Message.obtain();**

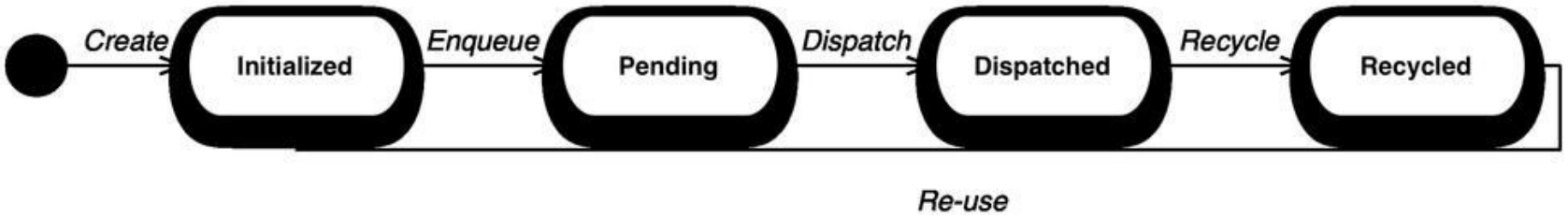
- Data message

- **Message m = Message.obtain(Handler h);**
- **Message m = Message.obtain(Handler h, int what);**
- **Message m = Message.obtain(Handler h, int what, Object o);**
- **Message m = Message.obtain(Handler h, int what, int arg1, int arg2);**
- **Message m = Message.obtain(Handler h, int what, int arg1, int arg2, Object o);**

- Task message
- **Message m = Message.obtain(Handler h, Runnable task);**
- Copy constructor
- Message m = Message.obtain(Message originalMsg);

Messages Lifecycle

- The lifecycle of a message is simple: the producer creates the message, and eventually it is processed by the consumer.
- This description suffices for most use cases, but when a problem arises, a deeper understanding of message handling is invaluable.
- Let us take a look into what actually happens with the message during its lifecycle, which can be split up into four principal states shown in Figure below.

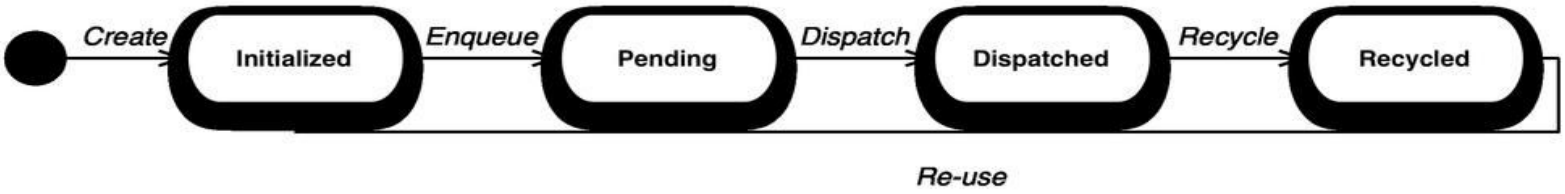


- The runtime stores message objects in an application-wide pool to enable the reuse of previous messages; this avoids the overhead of creating new instances for every hand-off.

- The message object execution time is normally very short and many messages are processed per time unit.

Messages Lifecycle

- The state transfers are partly controlled by the application and partly by the platform.
- States are not observable, and an application cannot follow the changes from one state to another (although there are ways to follow the movement of messages, explained later in Observing the Message Queue).
- Therefore, an application should not make any assumptions about the current state when handling a message.



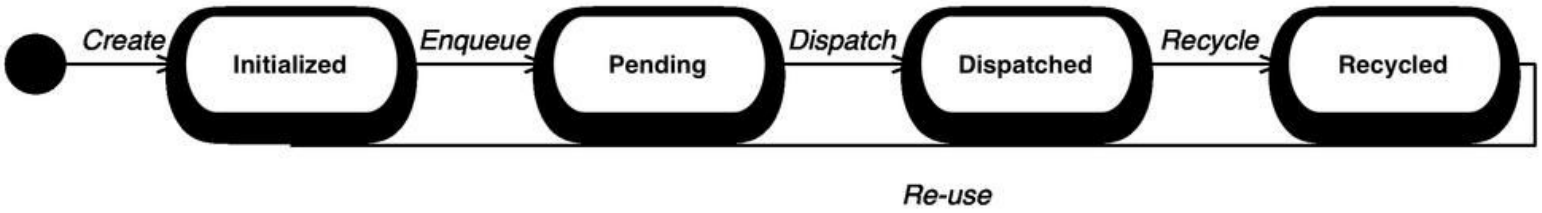
- Initialized

•In the initialized state, a message object with mutable state has been created and, if it is a data message, populated with data. The application is responsible for creating the message object using one of the calls to message methods creation. They take an object from the object pool.

- Pending

•The message has been inserted into the queue by the producer thread, and it is waiting to be dispatched to the consumer thread.

Messages Lifecycle



Dispatched

- In this state, the Looper has retrieved and removed the message from the queue. The message has been dispatched to the consumer thread and is currently being processed.
- When the looper dispatches a message, it checks the delivery information of the message, and delivers the message to the correct recipient. Once dispatched, the message is executed on the consumer thread.

Recycled

- At this point in the lifecycle, the message state is cleared and the instance is returned to the message pool.
- The Looper handles the recycling of the message when it has finished executing on the consumer thread.

Recycling of messages is handler by the runtime and should not be done explicitly by the application.

Looper

- The *android.os.Looper* class handles the dispatch of messages in the queue to the associated handler. All messages that have passed the dispatch barrier, as illustrated in a precedent figure, are eligible for dispatch by the Looper. As long as the queue has messages eligible for dispatch, the Looper will ensure that the consumer thread receives the messages. When no messages have passed the dispatch barrier, the consumer thread will block until a message has passed the dispatch barrier.

- The consumer thread does not interact with the message queue directly to retrieve the messages. Instead, a message queue is added to the thread when the looper has been attached. The looper manages the message queue and facilitates the dispatch of messages to the consumer thread.

- By default, only the UI thread has a Looper; threads created in the application need to get a Looper associated explicitly. When the Looper is created for a thread, it is connected to a message queue. The Looper acts as the intermediator between the queue and the thread. The Looper setup is done in the run method of the thread:

Looper

•class ConsumerThread extends Thread {

- @Override
- public void run() {
- Looper.prepare(); // Handler creation omitted.
- Looper.loop(); }}
- The first step is to create the Looper, which is done with the static prepare() method; it will create a message queue and associate it with the current thread. At this point, the message queue is ready for insertion of messages, but they are not dispatched to the consumer thread.
- *Looper.loop()* Starts handling messages in the message queue. This is a blocking method that ensures the run() method is not finished; while run() blocks, the Looper dispatches messages to the consumer thread for processing.

Looper

- A thread can have only one associated Looper; a runtime error will occur if the application tries to set up a second one.
- Consequently, a thread can have only one message queue, meaning that messages sent by multiple producer threads are processed sequentially on the consumer thread.
- Hence, the currently executing message will postpone subsequent messages until it has been processed. Messages with long execution times shall not be used if they can delay other important tasks in the queue (particularly in the UI thread).

Looper

- The Looper is requested to stop processing messages with either quit or quitSafely.
- ***quit()*** stops the looper from dispatching any more messages from the queue; all pending messages in the queue-including those that have passed the dispatch barrier will be discarded.
- ***quitSafely()***, on the other hand, only discards the messages that has not passed the dispatch barrier. Pending messages that are eligable for dispatch will be processed before the Looper is terminated.

- ***quitSafely*** was added in API level 18 (Jelly Bean 4.3). Previous API levels only support *quit*.

- Terminating a Looper does not terminate the thread; it merely exits Looper.loop() and lets the thread resume running in the method that invoked the loop call. But you cannot start the old looper or a new one, so the thread can no longer enqueue or handle messages. If you call Looper.prepare(), it will throw RuntimeException because the thread already has an attached Looper. If you call Looper.loop(), it will block, but no messages will be dispatched from the queue.

The UI thread Looper

- The UI thread is the only thread with an associated Looper by default. It is a regular thread, like any other thread created by the application itself, bút the Looper is associated with the thread before the application components are initialized.

- There are a few practical differences between the UI thread Looper and other application thread Loopers:

  – It is accessible from everywhere, through the ***Looper.getMainLooper()*** method.

  – It cannot be terminated. Looper.quit() throws RuntimeException.

  – The runtime associates a Looper to the UI thread by *Looper.prepareMainLooper()*. This can be done only once per application. Thus, trying to attach the main looper to another thread will throw an exception.

Handlers

- So far, the focus has been on the internals of Android thread communication, but an application mostly interacts with the ***android.os.Handler*** class.

- It is a two-sided API that both handles the insertion of messages into the queue and the message processing. As in next figure, it is invoked from both the producer and consumer thread typically used for:

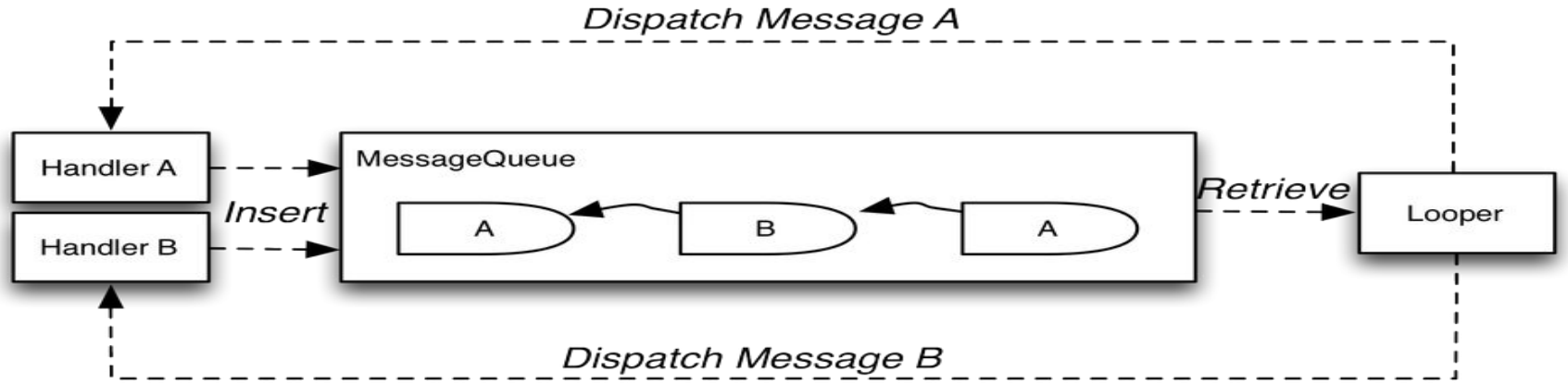
- Creating messages

- Inserting messages into the queue

- Processing messages on the consumer thread

- Managing messages in the queue

Handlers



Multiple handlers using one Looper. The handler inserting a message is the same handler processing the message.

- While carrying out its responsibilities, the Handler interacts with the Looper, message queue, and message.
- As figure above illustrates, the only direct instance relation is to the Looper, which is used to connect to the MessageQueue.
- Without a Looper, a Handler can not function; it cannot couple with a queue to insert messages and consequently it will not receive any messages to process. Hence, a Handler instance is bound to a Looper instance already at construction time:

Handlers

- Constructors without an explicit Looper bind to the Looper of the current thread.
- ***new Handler();*** ***new Handler(Handler.Callback);***
- Constructors with an explicit Looper bind to that Looper.
- ***new Handler(Looper);*** ***new Handler(Looper, Handler.Callback);***
- If the constructors without an explicit Looper are called on a thread without a Looper (i.e., it has not called Looper.prepare()), there is nothing the Handler can bind to, leading to a RuntimeException.
- Once a handler is bound to a Looper, the binding is final.
- A thread can have multiple Handlers; messages from them coexist in the queue but are dispatched to the correct Handler instance, as shown in previous figure.

Multiple handlers will not enable concurrent execution. The messages are still in the same queue and are processed sequentially.

Message creation

- For simplicity, the Handler class offers wrapper functions for the factory methods shown previously to create objects of the Message class.

- The message obtained from a Handler is retrieved from the message pool and implicitly connected to the Handler instance that requested it. This connection enables the looper to dispatch each message to the correct handler.

- Message obtainMessage(int what, int arg1, int arg2)
- Message obtainMessage()
- Message obtainMessage(int what, int arg1, int arg2, Object obj)
- Message obtainMessage(int what)
- Message obtainMessage(int what, Object obj)

Message insertion

- The handler inserts messages in the message queue in various ways depending on the message type. Task messages are inserted through methods whose names begin with post, whereas data messages are inserted through methods whose names begin with send:

- Add a task to the message queue.

- ***boolean post(Runnable r)***
- ***boolean postAtFrontOfQueue(Runnable r)***
- ***boolean postAtTime(Runnable r, Object token, long uptimeMillis)***
- ***boolean postAtTime(Runnable r, long uptimeMillis)***
- ***boolean postDelayed(Runnable r, long delayMillis)***

- Add a data object to the message queue.

- ***boolean sendMessage(Message msg)***
- ***boolean sendMessageAtFrontOfQueue(Message msg)***
- ***boolean sendMessageAtTime(Message msg, long uptimeMillis)***
- ***boolean sendMessageDelayed(Message msg, long delayMillis)***

- Add simple data object to the message queue.

- ***boolean sendEmptyMessage(int what)***
- ***boolean sendEmptyMessageAtTime(int what, long uptimeMillis)***
- ***boolean sendEmptyMessageDelayed(int what, long delayMillis)***

Message time parameters

- All insertion methods put a new Message object in the queue, even though the application does not create the Message object explicitly. The objects, such as Runnable in a task post and what in a send, are wrapped into Message objects, because those are the only data types allowed in the queue.

- Every message inserted in the queue comes with a time parameter indicating the time when the message is eligible for dispatch to the consumer thread. The sorting is based on the time parameter, and it is the only way an application can affect the dispatch order.

- **default**
- Immediately eligible for dispatch.

- **at_front**
- This message is eligible for dispatch at time 0. Hence, it will be the next dispatched message,
- unless another is inserted at the front before this one is processed.

- **delay**
- The amount of time after which this message is eligible for dispatch.

- **uptime**
- The absolute time at which this message is eligible for dispatch.

- Even though explicit delays or uptimescan be specified, the time required to process each message is still indeterminate. It depends both on whatever existing messages need to be processed first and the operating system scheduling.
- Inserting a message in the queue is not failsafe. Some errors can occur.

Example: Two-way message passing

- The HandlerExampleActivity simulates a long running operation that is started when the user clicks a button.
- The long running task is executed on a background thread; meanwhile, the UI displays a progress bar that is removed when the background thread reports the result back to the UI thread.

Two-way message passing

```
•public class HandlerExampleActivity extends Activity {

•private final static int SHOW_PROGRESS_BAR = 1;

•private final static int HIDE_PROGRESS_BAR = 0;

•    private BackgroundThread mBackgroundThread;

•    private TextView mText;

•    private Button mButton;

•    private ProgressBar mProgressBar;

•    @Override

•    public void onCreate(Bundle savedInstanceState) {

•        super.onCreate(savedInstanceState);

•
setContentView(R.layout.activity_handler_example);
```

```
•    mBackgroundThread = new BackgroundThread();

•        mBackgroundThread.start();
•mText = (TextView) findViewById(R.id.text);

•        mProgressBar = (ProgressBar)
findViewById(R.id.progress);

•        mButton = (Button) findViewById(R.id.button);

•        mButton.setOnClickListener(new OnClickListener()
{

• @Override  public void onClick(View v) {

•                mBackgroundThread.doWork(); }
•        });
•    }
•@Override     protected void onDestroy() {

•        super.onDestroy();

•        mBackgroundThread.exit();}
•    // ... The rest of the Activity is defined further down}
```

Two-way message passing

- A background thread with a message queue is started when the HandlerExampleActivity is created. It handles tasks from the UI thread.

- When the user clicks a button, a new task is sent to the background thread. As the tasks will be executed sequentially on the background thread, multiple button clicks may lead to queueing of tasks before they are processed.

- The background thread is stopped when the HandlerExampleActivity is destroyed.

Two-way message passing

•BackgroundThread is used to offload tasks from the UI thread.

•It runs—and can receive messages—during the lifetime of the HandlerExampleActivity.

•It does not expose its internal Handler; instead it wraps all accesses to the Handler in public methods doWork and exit.

•private class BackgroundThread extends Thread {

• private Handler mBackgroundHandler;

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

• mBackgroundHandler = new Handler(); Looper.loop();

. }

.

• public void doWork() {

• mBackgroundHandler.post(new Runnable() {
•@Override public void run() {
•Message uiMsg = mUiHandler.obtainMessage(
• SHOW_PROGRESS_BAR, 0, 0, null); mUiHandler.sendMessage(uiMsg);
•Random r = new Random();

• int randomInt = r.nextInt(5000);

• SystemClock.sleep(randomInt);
•uiMsg = mUiHandler.obtainMessage(

• HIDE_PROGRESS_BAR, randomInt, 0, null); mUiHandler.sendMessage(uiMsg); }
• });
• }
• public void exit()
{ mBackgroundHandler.getLooper().quit();
• }
•}

Key points in the code

•1) Associate a Looper with the thread.

•2) The Handler processes only Runnables. Hence, it is not required to implement Handler.handleMessage.

•3) Post a long task to be executed in the background.

•4) Create a Message object that contains only a what argument with a command—SHOW_PROGRESS_BAR—to the UI thread so that it can show the progress bar.

•5) Send the start message to the UI thread.

•6) Simulate a long task of random length, that produces some data randomInt.

•7`) Create a Message object with the result randomInt, that is passed in the arg1 parameter. The what parameter contains a command—HIDE_PROGRESS_BAR—to remove the progress bar.

•8) The message with the end result that both informs the UI thread that the task is finished and delivers a result.

•9 Quit the Looper so that the thread can finish.

Two-way message passing

•The UI thread defines its own Handler that can receive commands to control the progress bar and update the UI with results from the background thread.

```
•private final Handler mUiHandler = new Handler() {
•   public void handleMessage(Message msg) {
•       switch(msg.what) {
•           case SHOW_PROGRESS_BAR: mProgressBar.setVisibility(View.VISIBLE);
•               break;
•           case HIDE_PROGRESS_BAR: mText.setText(String.valueOf(msg.arg1));
•               mProgressBar.setVisibility(View.INVISIBLE);
•               break;
•       }
•   }};
```

•Show the progress bar.
•Hide the progress bar and update the TextView with the produced result.

Message processing

- Messages dispatched by the Looper are processed by the Handler on the consumer thread. The message type determines the processing:
- Task messages
  - Task messages contain only a Runnable and no data. Hence, the processing to be executed is defined in the run method of the Runnable, which is executed automatically on the consumer thread, without invoking ***Handler.handleMessage().***
- Data messages
  - When the message contains data, the handler is the receiver of the data, and is responsible for its processing. The consumer thread processes the data by overriding the ***Handler.handleMessage(Message msg)*** method. There are two ways to do this, described in the text that follows.
- One way to define handleMessage is to do it as part of creating a Handler. The method should be defined as soon as the message queue is available (after *Looper.prepare()* is called) but before the message retrieval starts (before *Looper.loop()* is called).

Message processing

•A template for setting up the handling of data messages is as follows:

```
•class ConsumerThread extends Thread {
•       Handler mHandler;
•       @Override
•       public void run() {

•               Looper.prepare();
•               mHandler = new Handler() {
•                       public void handleMessage(Message msg) {
•                               // Process data message here
•                       }
•               };)

•Looper.loop();    }

•}
```

- In this code, the Handler is defined as an anonymous inner class, but it could as well have been defined as a regular or inner class.

Message Processing

- A convenient alternative to extending the Handler class is to use the Handler.Callback interface, which defines a handleMessage method with an additional return parameter not in Handler.handleMessage().

- ***public interface Callback {***

- ***public boolean handleMessage(Message msg);***

- ***}***

- With the Callback interface, it is not necessary to extend the Handler class. Instead, the Callback implementation can be passed to the Handler constructor, and it will then receive the dispatched messages for processing.

The *Callback* interface

- With the Callback interface, it is not necessary to extend the Handler class. Instead, the Callback implementation can be passed to the Handler constructor, and it will then receive the dispatched messages for processing.

```
•public class HandlerCallbackActivity extends Activity implements Handler.Callback {

•     Handler mUiHandler;

•     @Override

•     public void onCreate(Bundle savedInstanceState)

•     {       super.onCreate(savedInstanceState);

•     mUiHandler = new Handler(this); }

•     @Override

•     public boolean handleMessage(Message message) {
•                    // Process messages
•               return  true;
•             }
•}
```

The *Callback* interface

- **Callback.handleMessage** should **return true** if the message is handled, which guarantees that **no further processing** of the message is done.

- **If**, however, **false** is returned, the message is passed on to the **Handler.handleMessage** method for f**urther processing.**

- Note that the **Callback does not override Handler.handleMessage**. Instead, it adds a message preprocessor that is invoked before the Handlers own method.

- The Callback preprocessor can intercept and change messages before the Handler receives them.

The *Callback* interface

•The following code shows the principle for intercepting messages with the Callback:

```
•public class HandlerCallbackActivity extends Activity implements Handler.Callback {

•       @Override
•       public boolean handleMessage(Message msg) {

•               Logg.d(TAG, "Primary Handler- msg = " + msg.what);

•               switch (msg.what) {

•               case 1:             msg.what = 11;

•               return true;

•               default:            msg.what = 22;


•               return false;       }


•       }
•
```

.

```
•       // Invoked on button click

•public void onHandlerCallback(View v) {

•               Handler handler = new Handler(this) {

•       @Override

•       public void handleMessage(Message msg) {


•Log.d(TAG, "Secondary Handler - msg = " + msg.what); }

•               };


•handler.sendEmptyMessage(1);


•handler.sendEmptyMessage(2);
•   }

•}
```

Notes on the code

- The HandlerCallbackActivity implements the Callback interface to intercept messages.

The Callback implementation intercepts messages.

- If msg.what is 1, it returns true—the message is handled. Otherwise, it changes the value of msg.what to 22 and returns false—the message is not handled, so it is passed on to the Handler implementation of handleMessage.

- Log the messages sent to the secondary handleMessage

- Insert a message with msg.what == 1. The message is intercepted by the Callback as it returns true.

- Insert a message with msg.what == 2. The message is changed by the Callback and passed on to the Handler that prints Secondary Handler - msg = 22.

Removing messages and runnables

- After enqueuing a message, the producer can invoke a method of the Handler class to remove the message, so long as it has not been dequeued by the Looper.

- Sometimes an application may want to clean the message queue by removing all messages, which is possible, but most often a more fine-grained approach is desired: an application wants to target only a subset of the messages.

- For that, it needs to be able to identify the correct messages. Hence, messages can be identified from certain properties:

- The handler identifier is mandatory for every message, because a message always knows what handler it will be dispatched to. This requirement implicitly restricts each Handler to removing only messages belonging to that Handler. It is not possible for a Handler to remove messages in the queue that were inserted by another Handler.

Removing messages and runnables

- The methods available in the Handler class for managing the message queue are:
- *Remove a task from the message queue.*
  - **removeCallbacks(Runnable r)**
  - **removeCallbacks(Runnable r, Object token)**
- *Remove a data message from the message queue.*
  - **removeMessages(int what)**
  - **removeMessages(int what, Object object)**
- *Remove tasks and data messages from the message queue.*
  - **removeCallbacksAndMessages(Object token)**

- The Object identifier used in both the data and task message. Hence, it can be assigned to messages as a kind of tag, allowing you later to remove related messages that you have tagged with the same Object.

Removing messages and runnables

•The following excerpt inserts two messages in the queue, to make it possible to remove them later based on the tag.

•Object ***tag*** = new Object();

•Handler handler = new Handler() {

• public void handleMessage(Message msg) {

• // Process message

• Log.d("Example", "Processing message"); }};

• Message **message** = handler.obtainMessage(0, **tag**); **handler.sendMessage(message);**

• handler.postAtTime(**new Runnable() {**

• **public void run() {**

• // Left empty for brevity **}}, tag,** SystemClock.uptimeMillis()**);**

• *handler.removeCallbacksAndMessages(**tag**);*

Removing messages and runnables

- The message tag identifier, common to both the task and data message.
- The object in a Message instance is used both as data container and implicitly defined message tag.
- Post a task message with an explicitly defined message tag.
- Remove all messages with the tag.
- As indicated before, you have no way to find out whether a message was dispatched and handled before you issues a call to remove it. Once the message is dispatched, the producer thread that enqueued it cannot stop its task from executing or its data from being processed.

Observing the Message Queue

- It is possible to observe pending messages and the dispatching of messages from a Looper to the associated Handlers. The Android platform offers two observing mechanisms. Let us take a look at them by example.

- The first example shows how it is possible to log the current snapshot of pending messages in the queue.

- **Taking a snapshot of the current message queue**

- This example creates a worker thread when the Activity is created. When the user presses a button, causing onClick to be called, six messages are added to the queue in different ways. Afterwards we observe the state of the message queue.

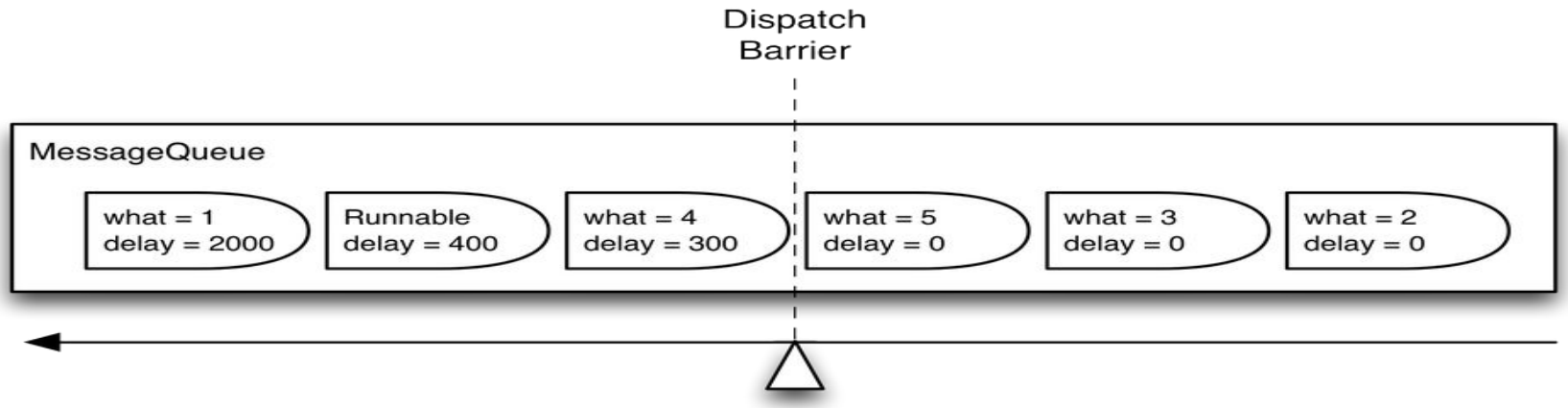
Observing the Message Queue

```
public class MQDebugActivity
         extends Activity {
 private static final String TAG = "EAT";
 Handler mWorkerHandler;
public void onCreate(Bundle
savedInstanceState){

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_mqdebug);
  Thread t = new Thread() {
      @Override
  public void run() {
 Looper.prepare();
     mWorkerHandler = new Handler()
{    @Override
public  void handleMessage(Message msg)
{ Log.d(TAG, "handleMessage - what = " +
msg.what);
                    }
                };
              Looper.loop();
          }
      };
       t.start();
}
```

•Called on button click, i.e. from the UI thread.

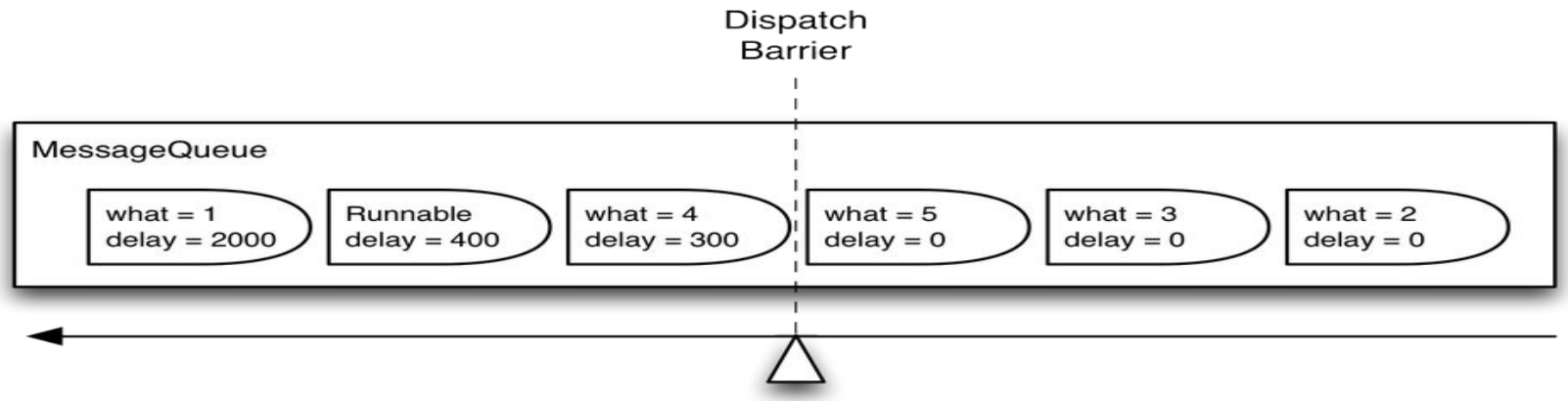
```
 public void onClick(View v)
{         mWorkerHandler.
         sendEmptyMessageDelayed(1, 2000);
mWorkerHandler.sendEmptyMessage(2);
mWorkerHandler.
    obtainMessage(3, 0, 0, new Object()).
          sendToTarget();
mWorkerHandler.
         sendEmptyMessageDelayed(4, 300);
mWorkerHandler.postDelayed(new Runnable()
{  @Override
    public void run() {
       Log.d(TAG, Execute");
                          }      }, 400);
mWorkerHandler.sendEmptyMessage(5);

mWorkerHandler.
dump(new LogPrinter(Log.DEBUG, TAG), "");

     }
}
```

Observing the Message Queue



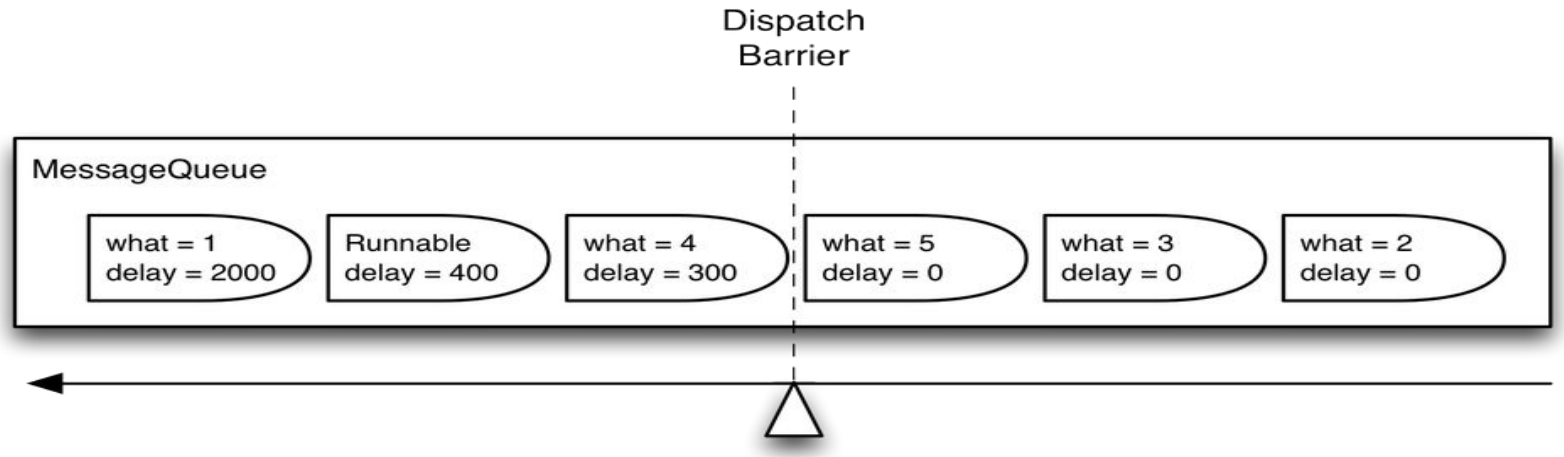
- Six messages, with the parameters shown in the figure, are added to the queue.
- Right after the messages are added to the queue, a snapshot is printed to the log. **Only pending messages are observed**. Hence, the number of messages actually observed depends on how many messages have already been dispatched to the handler.
- Three of the messages are added without a delay, which makes them eligible for dispatch at the time of the snapshot.

Observing the Message Queue



•49.397: handleMessage - what = 2
•49.397: handleMessage - what = 3
•49.397: handleMessage - what = 5
•49.397: Handler (com.wifill.eat.ui.MQDebugActivity$1$1) {412cb3d8} @
•5994288
•49.407: Looper{412cb070}
•49.407: mRun=true
•49.407: mThread=Thread[Thread-111,5,main]
•49.407: mQueue=android.os.MessageQueue@412cb090
•49.407: Message 0: { what=4 when=+293ms }
•49.407: Message 1: { what=0 when=+394ms }
•49.407: Message 2: { what=1 when=+1s990ms }
•49.407: (Total messages: 3)
•49.707: handleMessage - what = 4
•49.808: Execute
•51.407: handleMessage - what = 1

Observing the Message Queue



- The snapshot of the message queue shows that the messages with what parameters (0, 1, and 4) are pending in the queue. These are the messages added to the queue with a dispatch delay, whereas the others without a dispatch delay apparently have been dispatched already. This is a reasonable result because the handler processing is very short—just a print to the log.

- The snapshot also shows how much time is left before each message in the queue will pass the dispatch barrier. For instance, the next message to pass the barrier is Message 0 (what= 4) in 293 ms. Messages still pending in the queue but eligible for dispatch will have a negative time indication in the log, e.g., when is less than zero.

Tracing the message queue processing

- The message processing information can be printed to the log.
- Message queue logging is enabled from the Looper class.

- The following call enables logging on the message queue of the calling thread:

```
Looper.
  myLooper().
    setMessageLogging(
      new LogPrinter(Log.DEBUG, TAG));
```

- An example of tracing a message that is posted to the UI thread:

```
mHandler.post(new Runnable() {
  @Override
      public void run() {
   Log.d(TAG, "Executing Runnable");
    }
});
mHandler.sendEmptyMessage(42);
```

Tracing the message queue processing

- The example posts two events to the message queue: first a Runnable followed by an empty message. As expected, with the sequential execution in mind, the Runnable is processed first and consequently the first to be logged:

•>>>>> Dispatching to Handler (android.os.Handler) {4111ef40} com.wifill.eat.ui.MessageTracingActivity$1@41130820: 0

•Executing Runnable

•<<<<< Finished to Handler (android.os.Handler) {4111ef40} com.wifill.eat.ui.MessageTracingActivity$1@41130820

Tracing the message queue processing

- The trace prints the start and end of the event identified by three properties:
- Handler instance
  - android.os.Handler 4111ef40
- Task instance
  - com.wifill.eat.ui.MessageTracingActivity$1@41130820
- The what parameter
  - 0 (Runnable tasks do not carry a what parameter)

Tracing the message queue processing

- Similarly the trace of a message with the what parameter set to 42 prints the message argument but not any Runnable instance:

•>>>>> Dispatching to Handler (android.os.Handler) {4111ef40} null: 42

•<<<<< Finished to Handler (android.os.Handler) {4111ef40} null

- Combining the two techniques of message queue snapshots and dispatch tracing allows the application to observe message passing in detail.

Uithread

- The UI thread is the only thread in an application that has an associated Looper by default, which is associated on the thread before the first Android component is started.

- The UI thread can be a consumer, to which other threads can pass messages. It's important to send only short-lived tasks to the UI thread.

- The UI thread is application global and processes both android component and system messages sequentially. Hence, long-lived tasks will have a global impact across the application.

- Messages are passed to the UI thread through its Looper that is accessible globally in the application from all threads with **Looper.getMainLooper()**:

  –Runnable task = new Runnable() {...};
  –new Handler(Looper.getMainLooper()).post(task);

Uithread()

- Independent of the posting thread, the message is inserted in the queue of the UI thread.
- *If it is the UI thread that posts the message to itself, the message can be processed at the earliest after the current message is done:*

```
// Method called on UI thread.
private void postFromUiThreadToUiThread() {
        new Handler().post(new Runnable() { ... });
        // The code at this point is part of a message being processed
        // and is executed before the posted message.
 }
```

However, a task message that is posted from the UI thread to itself can bypass the message passing and execute immediately within the currently processed message on the UI thread with the convenience method ***Activity.runOnUiThread(Runnable)***:

```
// Method called on UI thread.
private void postFromUiThreadToUiThread() {
        runOnUiThread(new Runnable() { ... });
        // The code at this point is executed after the message.
 }
```

Uithread()

- If it is called outside the UI thread, the message is inserted in the queue.
- The runOnUiThread method can only be executed from an Activity instance, but the same behavior can be implemented by tracking the ID of the UI thread, for example with a convenience method customRunOnUiThread in an Application subclass.
- The customRunOnUiThread inserts a message in the queue like the following example:

```
•public class EatApplication extends Application {

•       private long mUiThreadId;
•       private Handler mUiHandler;
•       @Override
•       public void onCreate() {
•               super.onCreate();
•               mUiThreadId =
Thread.currentThread().getId();
•               mUiHandler = new Handler();
•               }

•       public void customRunOnUiThread(Runnable
action) {
•               if (Thread.currentThread().getId() !=
mUiThreadId) {
        mUiHandler.post(action);        }

•               else {
•                       action.run();
•                       }
•               }
•       }

.
```

HandlerThread

- ***A thread with a built-in looper***

HandlerThread

- ***HandlerThread*** is a handy class for starting a new thread that has a Looper (hence an associated MessageQueue) attached so that one doesn't have to go through creating a Thread and calling Looper.prepare(), Looper.loop(), etc.

- Generally a thread attached with a Looper is needed when we want sequential execution of tasks without race conditions and keep a thread alive even after a particular task is completed so that it can be reused so that you don't have to create new thread instances.

- Starting the same thread object again raises ***IllegalThreadStateException*** with a Thread already started message.

HandlerThread

•Once a HandlerThread is started, it sets up queuing through a Looper and MessageQueue and waits for incoming messages to process:

•***HandlerThread handlerThread = new HandlerThread("HandlerThread");***

•***handlerThread.start();***

•*// Create a handler attached to the HandlerThread's Looper*

•***mHandler = new Handler(handlerThread.getLooper()) {***

• *@Override*

• *public void handleMessage(Message msg) {*

• *// Process messages here*

• *}*

•*};*

•*// Now send messages using mHandler.sendMessage()*

HandlerThread

- There's only one associated MessageQueue on the thread, hence execution is guaranteed to be sequential and therefore thread-safe.

- Behind the scenes, HandlerThread guarantees no race condition between the Looper creation and sending messages by making handlerThread.getLooper() a blocking call until it is ready to receive messages.

- This is an important reason why HandlerThread should be used over manual setup with Looper.prepare(), Looper.loop(), Looper.quit(), etc.

HandlerThread

- The *HandlerThread.onLooperPrepared()* method can be used to execute some sort of setup before the Looper loops, like creating a Handler that will be associated with the HandlerThread.

- This method is invoked on the background thread when the Looper is prepared (after the Looper.prepare() call).

- If you want to prevent access to the Handler that is used to pass a data message or a task to a HandlerThread and ensure that the Looper is also not accessible, then you can create a separate class with a private Handler and public methods that actually does the job of passing messages or tasks. Something like this:

HandlerThread

```
class MyHandlerThread extends HandlerThread {


        private Handler mHandler;

        public MyHandlerThread() {

                super("MyHandlerThread",

        Process.THREAD_PRIORITY_BACKGROUND);

        }

        @Override

        protected void onLooperPrepared() {

                super.onLooperPrepared();

                mHandler = new Handler(getLooper()) {

        @Override

public void handleMessage(Message msg) {

       switch (msg.what) {
```

```
case 1:
        // Handle message

        break;

case 2:
       // Handle message

        break;

                                                        }

                                        }

                };

        }

        public void taskOne() {

                mHandler.sendEmptyMessage(1);

        }

        public void taskTwo() {

                mHandler.sendEmptyMessage(2);

        }

}
```

HandlerThread

•Pretty straightforward code. Although one important line of code to notice is this –

•super("MyHandlerThread",
• Process.THREAD_PRIORITY_BACKGROUND);

•This HandlerThread constructor takes a:

- name argument required for debugging purposes so that the thread can be found easily in logs.
- priority argument specified via Process.setThreadPriority() with values supplied from Process.
- The default priority is Process.THREAD_PRIORITY_DEFAULT (same as that of the UI thread) but can be lowered down to Process.THREAD_PRIORITY_BACKGROUND (less important tasks).

Android Concurrency

Handlers, Messages and Loopers

Efficient Android Threading: Asynchronous Processing Techniques for Android Applications 1st Edition
by Anders Goransson

Université Mohammed V
FACULTE DES SCIENCES
RABAT / FSR
Département informatique

Master IAO
Master II–Semestre 3
Cours

Mobile & Cloud Computing

Pr. REDA Oussama Mohammed

2019-2020