

Handler e Looper

Long Operation

```
...
((Button)findViewById(R.id.Button01)).setOnClickListener(
    new OnClickListener() {
        @Override
        public void onClick(View v) {
            int result = doLongOperation();
            updateUI(result);
        }
    });
...
```

Long Operation e Thread

...

```
((Button)findViewById(R.id.Button01)).setOnClickListener(
    new OnClickListener() {
        @Override
        public void onClick(View v) {
            (new Thread(new Runnable() {
                @Override
                public void run() {
                    int result = doLongOperation();
                    updateUI(result);
                }
            })).start();
        }
    }
}
```

...

Long Operation e Thread

```
...
((Button)findViewById(R.id.Button01)).setOnClickListener(
    new OnClickListener() {
        @Override
        public void onClick(View v) {
            (new Thread(new Runnable() {
                @Override
                public void run() {
                    int result = doLongOperation();
                    updateUI(result);
                }
            })).start();
        }
    }
);
...
```

```
FATAL EXCEPTION: Thread-8
android.view.ViewRoot$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
    at android.view.ViewRoot.checkThread(ViewRoot.java:2802)
    at android.view.ViewRoot.requestLayout(ViewRoot.java:594)
```

Single Thread Model

- Il thread creato all'avvio dell'applicazione è quello incaricato di gestire eventi e oggetti grafici
 - è chiamato il main thread
- Due regole:
 - Non bloccare il main thread
 - Non accedere agli oggetti grafici da thread che non sono il main thread

Soluzione 1

- Inviare il codice con l'aggiornamento al main thread
 - Il codice è contenuto nel metodo run di una Runnable
- Metodi di aiuto
 - Activity.**runOnUiThread**(Runnable)
 - View.**post**(Runnable)
 - View.**postDelayed**(Runnable, long)

Esempio

```
...
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap = loadImageFromNetwork(
                "http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run(){
                    mImageView.setImageBitmap(
                        bitmap);
                }
            });
        }
    }).start();
}
...
```

>> La post permette di eseguire Runnable nel main Thread

Soluzione 2

- Inviare un messaggio al main thread
- I messaggi sono gestiti mediante una coda
 - Ogni messaggio viene gestito nel main thread che possiede la coda
- Gli oggetti che gestiscono questi messaggi si chiamano **Handler**
- Alcuni messaggi sono speciali e contengono una Runnable

Inviare un messaggio al main thread

...

```
final Handler myHandler = new Handler(){
    public void handleMessage(Message msg) {
        updateUI((String)msg.obj);
    }
};
```

...

```
(new Thread(new Runnable() {
    public void run() {
```

```
        Message msg = myHandler.obtainMessage();
        msg.obj = doLongOperation();
        myHandler.sendMessage(msg);
```

```
    }
```

```
}).start();
```

...

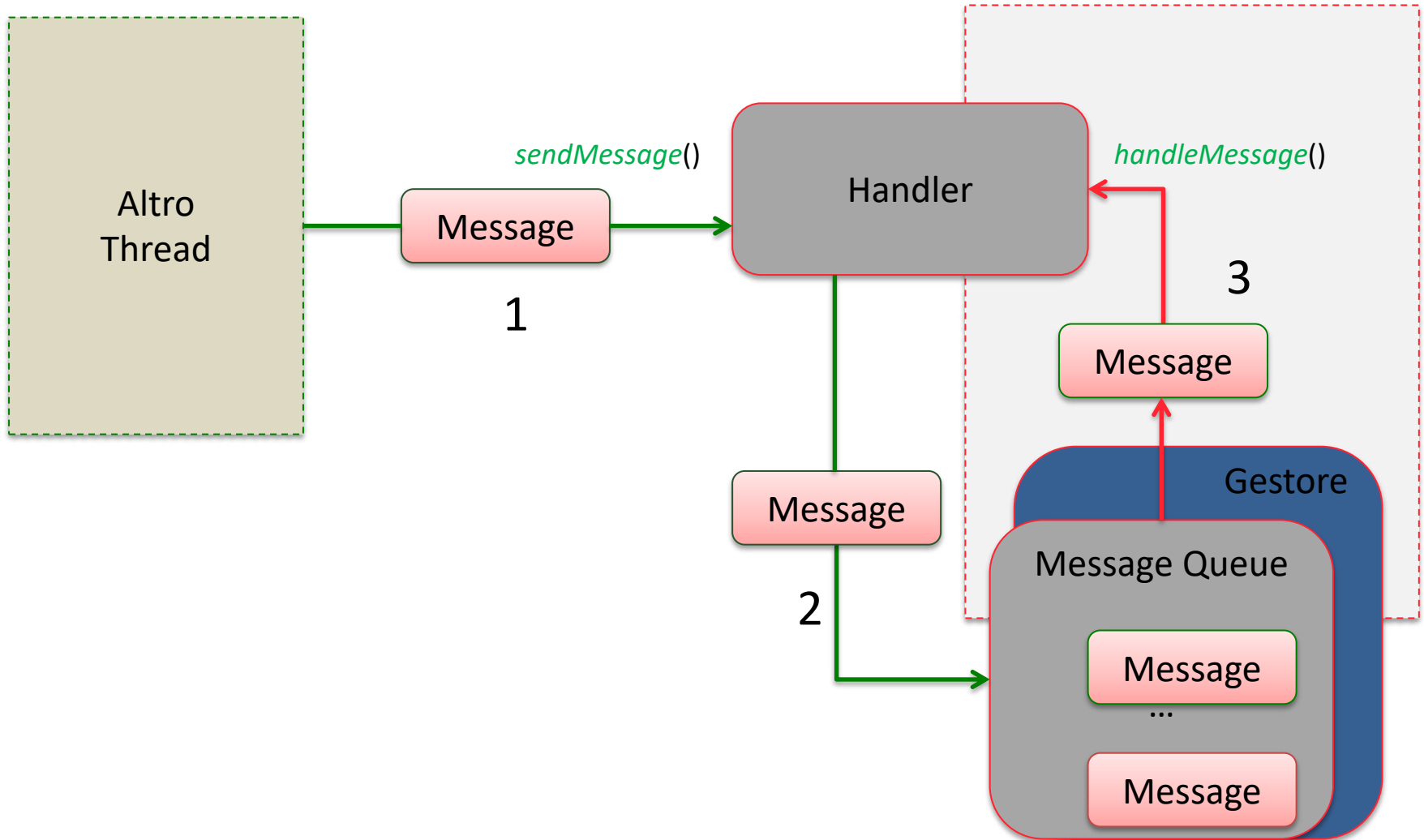
Esempio Handler

```
final Handler handler = new Handler() {
    public void handleMessage(android.os.Message msg) {
        switch (msg.what) {
            case 0:
                doSomething();
                break;
            case 1:
                doSomethingElse();
                break;
            default:
                super.handleMessage(msg);
        }
    }
};
...
```

```
android.os.Message newmsg = handler.obtainMessage(1); // oppure 0 o niente
Bundle b = new Bundle();
b.putString("key", "value");
newmsg.setData(b);
handler.sendMessage(newmsg);
```

Percorso dei Messaggi

Main Thread



Handler

- Un Handler è una classe che permette a due o più thread di scambiarsi informazioni di tipo Message e/o Runnable.
 - il Runnable viene messo nel messaggio
- Un Handler è “associato” **solo** al thread che lo ha creato.
- La coda utilizzata di default dall’Handler per accodare i messaggi è quella del thread che lo ha creato
 - se ne può impostare una diversa passandola come argomento al costruttore
- L’Handler si occupa di inserire i messaggi in coda e di gestire quelli che gli vengono recapitati dalla coda.

Metodi dell'Handler

- Metodi per spedire Messaggi:
 - handler.*sendMessage*(Message)
 - handler.*sendMessageAtFrontOfQueue*(Message)
 - handler.*sendMessageAtTime*(Message,long)
 - handler.*sendMessageDelayed*(Message,long)
- Metodi per inviare Runnable:
 - handler.*post*(Runnable)
 - handler.*postDelayed*(Runnable, long)
 - handler.*postAtTime*(Runnable, long)
- Metodi per gestire la ricezione
 - *dispatchMessage*(Message)
 - *handleMessage*(Message)

Post a Message

- *post(Runnable r)*

```
public final boolean post(Runnable r){
    return sendMessageDelayed(getPostMessage(r), 0);
}
```

- *postDelayed(Runnable, long)*

```
public final boolean postDelayed(Runnable r, long delayMillis){
    return sendMessageDelayed(getPostMessage(r), delayMillis);
}
```

- *postAtTime(Runnable, long)*

```
public final boolean postAtTime(Runnable r, long uptimeMillis){
    return sendMessageAtTime(getPostMessage(r), uptimeMillis);
}
```

Messaggi Runnable

- Runnable to Message
 - *getPostMessage*(Runnable)

```
private final Message getPostMessage(Runnable r)
{
    Message m = Message.obtain(); Ritorna un Message da un pool
    m.callback = r;
    return m;
}
```


Send Message

- *sendMessage(Message)*

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0);
}
```

- *sendMessageDelayed(Message, long)*

```
public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0)
        delayMillis = 0;

    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
```

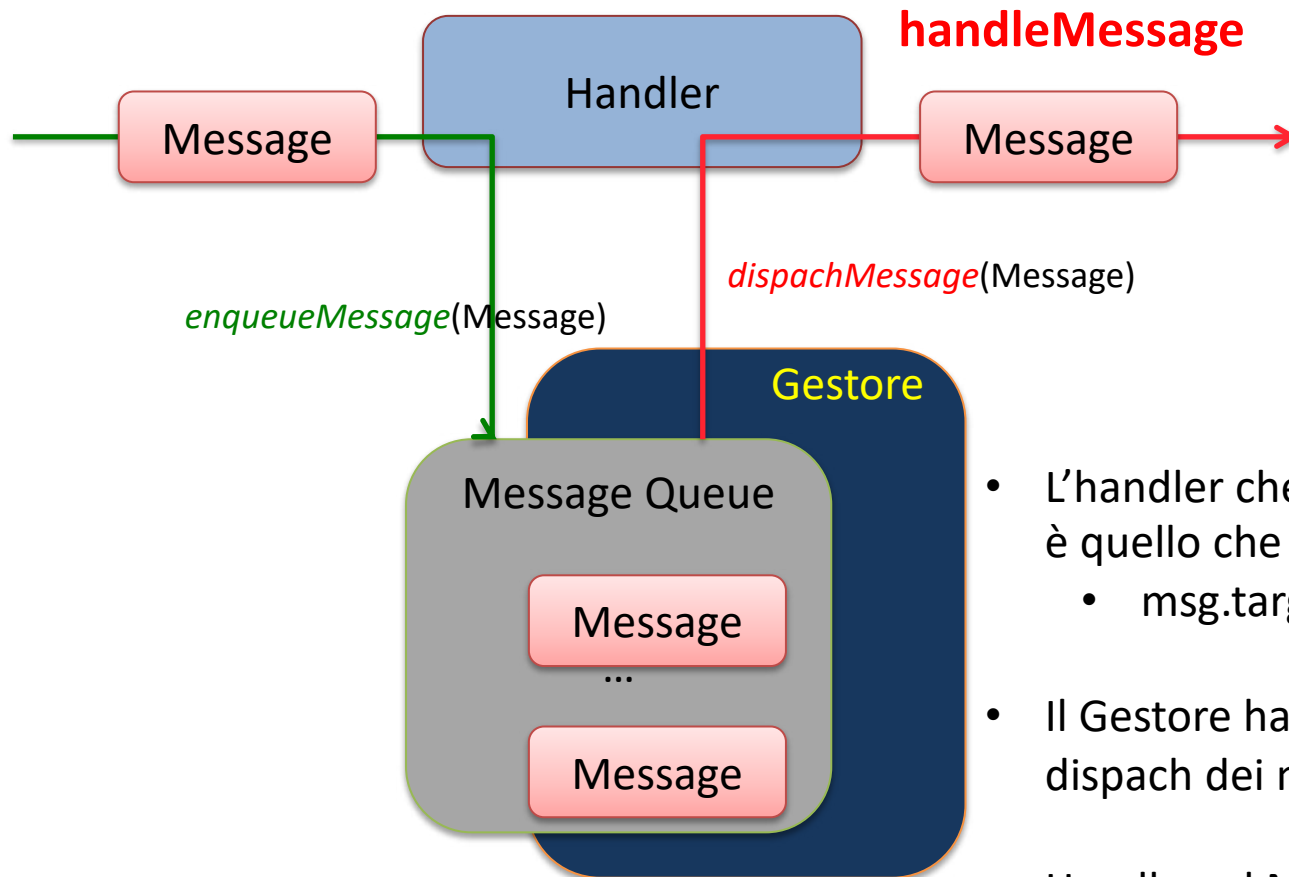
Accodamento Messaggio

- *sendMessageAtTime(Message, long)*

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
    boolean sent = false;
    MessageQueue queue = mQueue;

    if (queue != null) {
        msg.target = this;
        sent = queue.enqueueMessage(msg, uptimeMillis);
    }
    else {
        RuntimeException e = new RuntimeException(
            this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
    }
    return sent;
}
```

Message Queue



- L'handler che "gestisce" il messaggio è quello che lo ha accodato
 - `msg.target`
- Il Gestore ha il compito di operare il dispatch dei messaggi
- Handle nel Main thread
 - accodamento può essere in un altro thread

Message Handling

- *dispatchMessage(Message)*

```
new Handler()
{
    public void dispatchMessage(Message msg)
    {
        if (msg.getCallback() != null)
            executeRunnable(msg.getCallback());
        else
            handleMessage(msg);
    }
};
```

Message Handling

- *handleMessage (Message msg)*

```
new Handler()  
{  
    @Override  
    public void handleMessage (Message msg)  
    {  
        doSomethingWithMessage(msg);  
    }  
};
```


Handler e Runnable

```
final Handler myHandler = new Handler();
```

```
...
```

```
(new Thread(new Runnable() {
    public void run() {
        final String res = doLongOperation();
        myHandler.post(new Runnable() {
            public void run() {
                updateUI(res);
            }
        });
    }
})).start();
```

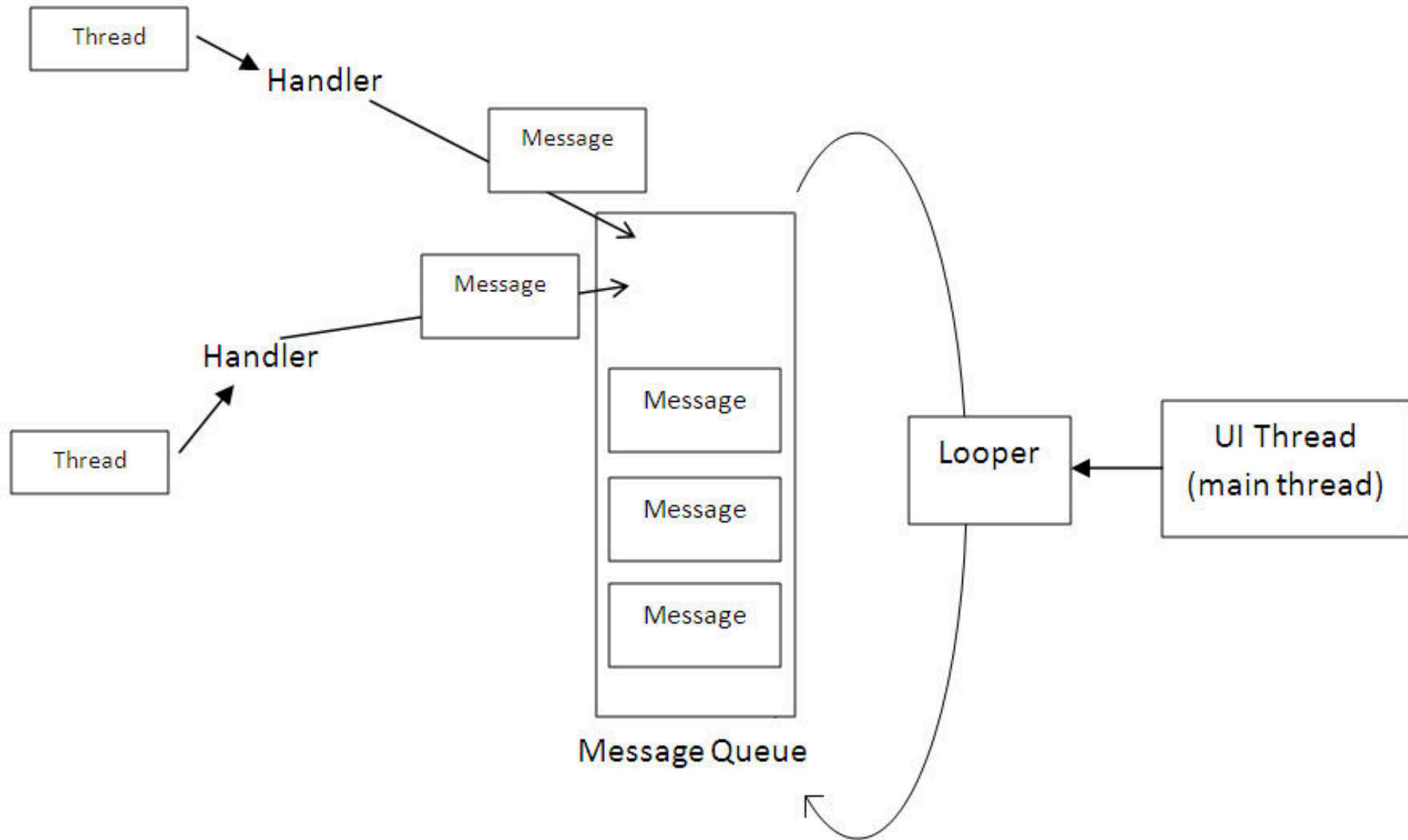
```
...
```

Le activity usano l'handler

...

```
public final void runOnUiThread(Runnable action) {  
    if (Thread.currentThread() != mUiThread) {  
        mHandler.post(action);  
    } else {  
        action.run();  
    }  
}
```

...



Un “Pipeline Thread”

- Il **Pipeline Thread** possiede una coda di “task” da eseguire
 - i task sono “unità” di lavoro da compiere
- Altri thread inseriscono task nella coda
 - lo possono fare in modo asincrono
 - la coda gestisce la concorrenza sull’accesso
- Il **Pipeline Thread** esegue i task uno dopo l’altro e attende nuovi task se la coda è vuota

La classe che trasforma un thread in un pipeline thread si chiama **Looper**

- Il gestore della coda

Looper

- Il Looper è la classe che gestisce la **MessageQueue** associata ad un Thread.
- Un Looper è associato solo ad un singolo Thread
 - ma può essere legato a più Handler.
- Il Looper gestisce dei messaggi in modo asincrono, schedulandoli nel tempo ed inviandoli all'handler che li ha creati
- Il legame tra Thread e Looper si instaura a seguito della chiamata statica `Looper.prepare()`
 - nel caso del main Thread con la chiamata `Looper.prepareMainLooper()`

Creare altri Pipeline Thread

...

```
public void run() {
    try {
        Looper.prepare();
        handler = new Handler();
        Looper.loop();           ← il thread è in loop qui
    } catch (Throwable t) {
        Log.e(TAG, "halted due to an error", t);
    }
}
```

...

```
handler.post(new Runnable() {
    public void run() {
        // viene eseguito nella pipeline
    }
});
```

...

Creazione del looper

Looper.prepare()

```
public static final void prepare()
{
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created
                                   per thread");
    }
    sThreadLocal.set(new Looper());
}
```

Loop

- `Looper.loop()`

```
public static final void loop() {  
    Looper me = myLooper();  
    MessageQueue queue = me.mQueue;  
  
    while (true) {  
        msg = queue.next();  
        if (msg != null) {  
            if (msg.target == null) // No target is a magic identifier for the quit message.  
                return;  
            msg.target.dispatchMessage(msg);  
            msg.recycle();  
        }  
    }  
}
```

Associazione Handler-Looper

```
public Handler()
{
    if (FIND_POTENTIAL_LEAKS)
    {
        final Class<? extends Handler> klass = getClass();
        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&
            (klass.getModifiers() & Modifier.STATIC) == 0) {
            Log.w(TAG, "The following Handler class should be static or leaks might occur: " +
                klass.getCanonicalName());
        }
    }
}
```

```
mLooper = Looper.myLooper();
if (mLooper == null) {
    throw new RuntimeException(
        "Can't create handler inside thread that has not called Looper.prepare()");
}
mQueue = mLooper.mQueue;
mCallback = null;
```

1) Non esiste un Handler senza un looper

2) La coda usata dall'Handler è presa dal Looper

Thread Local

| sThreadLocal | | | |
|--------------|----------|----------|----------|
| Thread 1 | Thread 2 | Thread 3 | Thread 4 |
| Looper 1 | Looper 2 | Looper 3 | Looper 4 |

- Ogni thread “vede” il suo looper
 - variabile statica con scope il thread corrente
- **Pattern ThreadLocal**
 - la classe Looper dichiara un avariabile **static final** di tipo **ThreadLocal**
 - quando chiamo Looper.prepare() nel TL viene creata una associazione Looper-Thread
 - Il TL gestisce una map
 - un Handler richiede un looper con **Looper.myLooper()**
 - viene ritornato quello associato al thread corrente
 - sThreadLocal.get();

Terminare un loop

- quit()

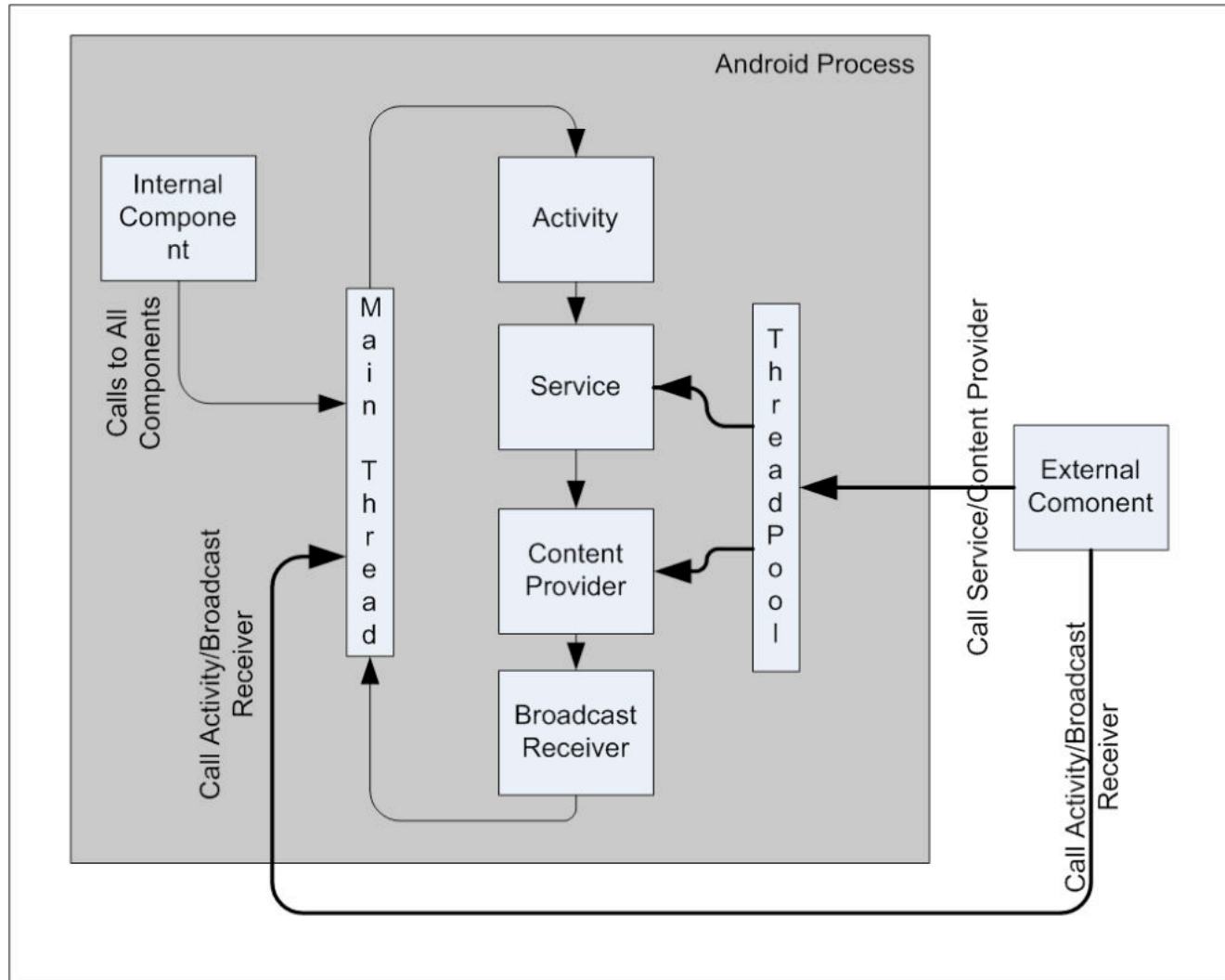
```
public void quit()
{
    // NOTE: By enqueueing directly into the message queue, the
    // message is left with a null target. This is how we know it is
    // a quit message.
    Message msg = Message.obtain();
    mQueue.enqueueMessage(msg, 0);
}
```

- La chiamata `looper.quit()` deve essere fatta da un altro Thread ma sull'istanza di `Looper` che si vuole terminare.

La main dell'app

```
public static void main(String[] args) {  
    ...  
    Looper.prepareMainLooper();  
    if (sMainThreadHandler == null) {  
        sMainThreadHandler = new Handler();  
    }  
    ...  
  
    Looper.loop();  
    throw new RuntimeException(  
        "Main thread loop unexpectedly exited");  
}
```

Android Architecture



Associazione Looper-Handler

Thread A

```
public void run()
{
    Looper.prepare();
    handlerA = new Handler();
    Looper.loop();
}

public Looper getLooper()
{
    return handlerA.getLooper();
}
```

Thread B

```
public void run()
{
    handlerB = new Handler(looperA);

    Message msg = new Message();
    Bundle bundle = new Bundle();
    bundle.putString("key", "value");
    msg.setData(bundle);

    handlerB.sendMessage(msg);
}
```


Errori comuni 1

Class ThreadB extends Thread

```
{  
    Handler handlerB ;  
  
    public void run()  
    {  
        handlerB = new Handler() {  
            @Override  
            public void handleMessage(Message msg)  
            {  
                doSomethingWithMsg(msg);  
            }  
        }  
    }  
}
```

Errori comuni 1

Class ThreadB extends Thread

```
{  
    Handler handlerB ;  
  
    public void run()  
    {  
        handlerB = new Handler() {  
            @Override  
            public void handleMessage(Message msg)  
            {  
                doSomethingWithMsg(msg);  
            }  
        }  
    }  
}
```

- Manca **Looper.prepare()** e **Looper.loop()**

Errori comuni 2

```
class ThreadA extends Thread
{
    Handler handlerA;

    public ThreadA()
    {
        handlerA = new Handler() {
            @Override
            public void handleMessage(Message msg)
            {
                doSomethingWithMsg(msg);
            }
        }
    }

    public run(){
        Looper.prepare();
        Looper.loop();
    }
}
```

Errori comuni 2

```
class ThreadA extends Thread
{
    Handler handlerA;

    public ThreadA()
    {
        handlerA = new Handler() {
            @Override
            public void handleMessage(Message msg)
            {
                doSomethingWithMsg(msg);
            }
        }
    }

    public run(){
        Looper.prepare();
        Looper.loop();
    }
}
```

Handler istanziato nel costruttore – thread “sbagliato”!!

Errori comuni 3

```
Message msg = new Message();  
Bundle bundle = new Bundle();  
msg.setData(bundle);  
msg.setTarget(handlerA);  
handlerB.sendMessage(msg);
```

Errori comuni 3

```
Message msg = new Message();
Bundle bundle = new Bundle();
msg.setData(bundle);
msg.setTarget(handlerA);
handlerB.sendMessage(msg);
```

1 – Questa chiamata sovrascrive il target (Handler di destinazione)

- **Message.setTarget(Handler)**

Se si utilizza la chiamata `sendMessage()`, nel campo `target` del messaggio viene salvato l'Handler dal quale si sta effettuando la chiamata, andando a sovrascrivere eventuali target precedenti.