Threads and the User Interface

Mario Mata
mario.mata@gcu.ac.uk
M215a

GCU
Glasgow Caledonian University

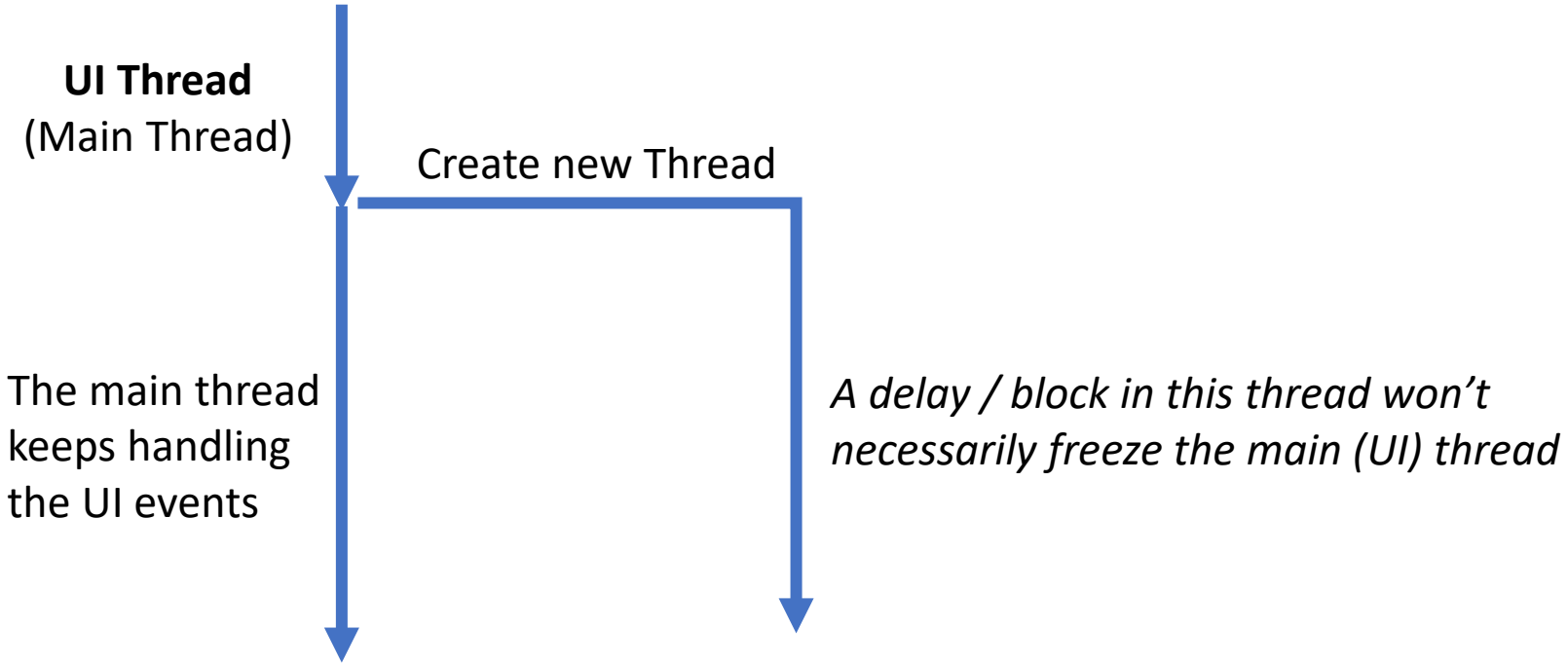
University for the Common Good

What is a thread?

- A **Thread** is a **separate execution path** within a *process* (a program)
  - Core element for concurrent and/or parallel programming
  - Support for threads is provided in any high-level programming language, and implemented by the OS
- Threads intend to improve performance:
  - exploit multiple processing cores, as each core can run one thread (parallel execution)
  - more memory-efficient than separate programs
    - each thread has its own stack, but **shares the heap** with other threads created within the same process (**= shares parent class *members***)
- When an app starts, the Android system starts a new Linux process for it with a single thread of execution
  - By default, all components of the same application run in the same process and thread, called the ***main thread***
    - As this thread is the one handling the UI, it's also called the ***UI thread***

Keeping the app UI responsive

- Threads help keeping the User Interface (UI) responsive:
  - a **main thread** services the UI (handles events from widgets)
  - Any lengthy background actions (network operations, database calls, loading of certain components) can run in other threads
    - those background (worker) threads should **not** handle the UI directly



The UI thread in Android

- In Android, the UI (main) thread is the one intended to make changes to the User Interface
- Performing a long-lasting operation in the UI thread (heavy calculations, accessing a database or a web resource) causes the UI code to “block” until it's finished
- This has a relevant usability impact:
  - Events triggered by user actions on widgets won't be able to execute their callback listeners immediately
  - The user will feel that the app “freezes”!
- Indeed, Android will display an “Application not responding” dialogue if a running activity does not respond within 5 seconds
  - The user is given the option to close the app at that point

Creating threads in Java

- To develop code that can run in a separate thread in Java, you need to create a class that either:
  - *Extends* the `Thread` class
    - Simpler, but it exhausts the class's inheritance ability
  - Or *implements* the `Runnable` interface. That's the preferred approach as then you can:
    - implement other interfaces, and extend other class if needed
    - pass it around as a task within a single-threaded application
    - pass it on to a separate service
    - cleaner separation between code and the implementation of threads

- Regardless of the approach taken, the class must implement a method called `run()` with signature:

```
public void run(){…}
```

  - The `run()` method contains the **action** -the code that you want to run in a separate thread of execution

Starting a thread

- You don't call the `run()` method directly: the thread needs to be sent to the **JVM thread scheduler:**
  - The scheduler is then responsible for starting the thread, that competes with other threads for the available resources
  - The scheduler will eventually call its `run()` method

- To start the execution of an object that implements a thread, you call the **`start()`** method on that object
  - Calling the `start()` method indicates to the JVM thread scheduler that the thread is ready to run

Starting a thread: example

- A first example on starting a thread in Java.
- We create a class implementing the `Runnable` interface – which gives us access to the `run()` method:

```
public class DemoRunnable implements Runnable {
    public void run(){
      //Code to run in separate thread
    }
}
```

- We create a Thread object using an instance of our class, then call `start()` to send it to the thread scheduler:

```
Thread t = new Thread(new DemoRunnable());
t.start();
```

- Or as a single-liner:

```
new Thread(new DemoRunnable()).start();
```

UI thread and background processing

- We use background (or worker) threads to *offload* time-consuming operations from the UI thread
  - This way we don't "block" the UI thread
- If any result from a background thread needs to be *updated on the UI*, we must utilise a specific technique to perform the updating
  - **do not update an UI widget** directly from a background thread
  - this is because Android UI widgets are not “thread safe”
    - they can end up with an inconsistent state because of an update from a background thread (the main thread may not "see" this update)
- **How to “connect” the worker thread to the UI, then?**

Worker thread updating the UI

- Worker threads can safely perform an action on an UI's widget by **wrapping it in another `Runnable()`**, and then following one of those approaches:

1. ***Post*** the new Runnable to the target widget with its `post()` method (inherited from the `View` class):

   ```
   target_widget.post(Runnable);
   or target_widget.postDelayed(Runnable, long);
   ```

2. Use method `runOnUiThread()` from the main UI thread (inherited from the parent `Activity` class):

   ```
   Activity.runOnUiThread(Runnable);
   ```

3. Use a **`Handler`** object to post the Runnable (or a *Message*) using the threads ***MessageQueue***
   - this is the most complex (but most powerful) way

1. post() example (*ThreadTest1*)

- A thread handles calculation, and updates result on a View object (widget) using its **post()** method:

```
public class MainActivity extends AppCompatActivity
implements View.OnClickListener{
 …… //Main Activity
 public void onClick(View v){
   new Thread(new Runnable(){ //calculation thread
      public void run() {
        result = calculate(); //some time-consuming task
        // Now display on UI component (add to its queue)
        answerTextView.post(new Runnable(){
              public void run(){
                answerTextView.setText(result);
              }
        }); //posted action wrapped on its own Runnable
      }
   }).start(); //starts calculation thread
 } //End of onClick
```

2. runOnUiThread() example (*ThreadTest2*)

- A thread handles calculation, then calls the Activity's runOnUiThread() to update a specific View object:

```
public void onClick(View v){
  //Create a new Thread to handle a long running task
  new Thread(){
    public void run(){
      result = calculate(); //some time-consuming task
      //Queue action on UI event queue
      runOnUiThread(new Runnable() {
        @Override
        public void run(){
          answerTextView.setText(result);
        }
      }); //posted action wrapped on its own Runnable
    }
  }.start(); //starts calculation thread
} // End of onClick
```

You can use this approach in the Coursework but won't get marks for it!

3. The `Handler` class

- Summarised from
  http://developer.android.com/reference/android/os/Handler.html:

  "A **`Handler`** allows you to send and process Message and **Runnable** objects to a thread's MessageQueue.

  When you create a new `Handler`, it is **bound** to the thread that is creating it (and to the *message queue* of that thread).

  It will deliver Messages and Runnables to that message queue, and execute them as they come out of this queue."

- Therefore, we can add stuff (*messages* and runnable actions) to the **main UI thread handler**, and they'll then be handled from the main UI thread message queue
- Messages are useful to **just send data** to the main thread

Use of the `Handler` class

- There are two main uses for a Handler:
  - to **schedule** *messages* and *runnables* to be executed at some point in the future in the current thread
  - to **enqueue an action** to be performed on a *different* thread than your own (by using that thread's handler)
    - This is often used to update a UI widget (running on the main UI thread) from a background thread
- There are two commonly used methods:

  `post(Runnable)`

  `sendmessage(Message)`

- The given Runnable or Message will be scheduled in the Handler's *message queue* and processed when appropriate (as soon as possible, or with a delay)

Handler usage: *HandlerTestProject2*

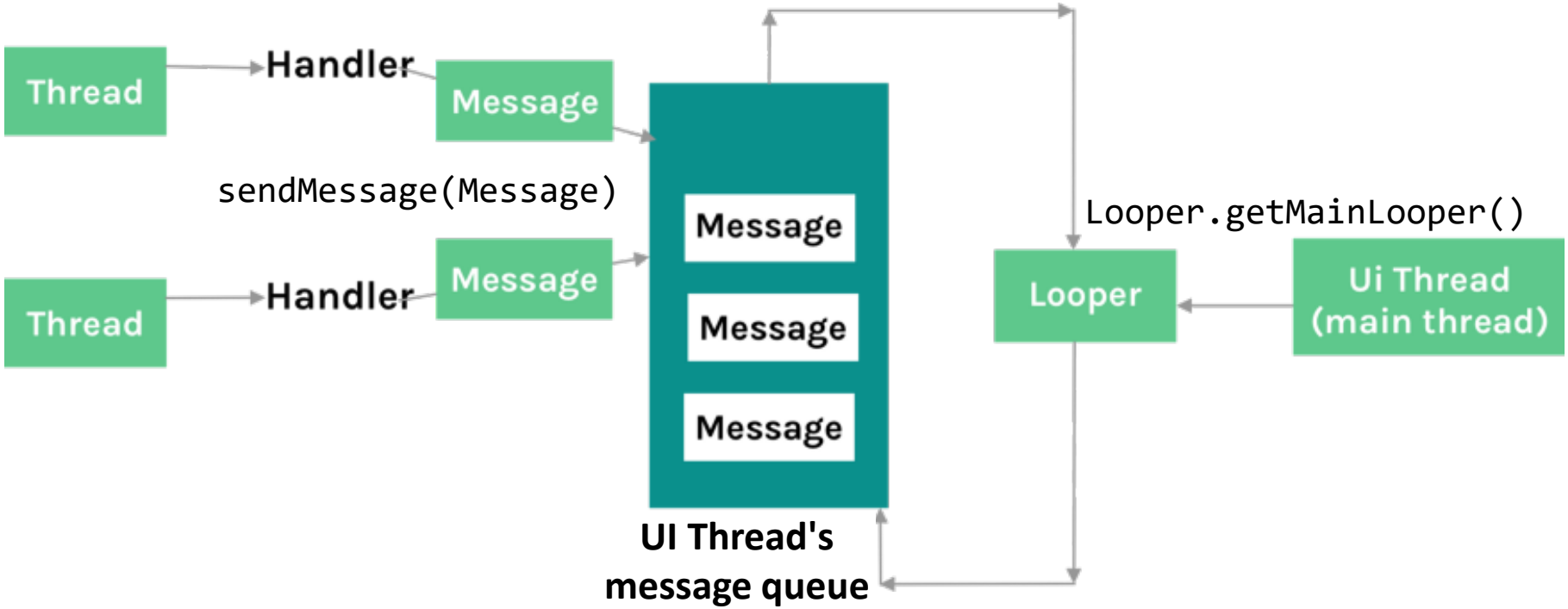
- Here, the worker thread result is actioned by *posting a runnable to the UI thread Handler* (so it runs on the UI):

Created on main thread → connected to main thread's message queue

```
import android.os.Handler;
… //Main Activity
public void onClick(View v){
 mHandler = new Handler(); //UI handler – used to update UI via post()
 //Create a Thread to handle some "long-running calculation"
 new Thread(new Runnable() {
     @Override
     public void run(){
        result = calculate(); // a potentially time consuming task
        // Update the value from worker thread to UI thread
        // This is done in a further short-life Runnable
        // to ensure the is no blocking of the "calculation" thread
        mHandler.post(new Runnable(){
           @Override
           public void run(){
              displayAnswerTextView.setText(result);
           }
        });
     }
 }).start();
} // End of onClick
```

Using Handlers to queue Messages

- Worker threads can use handlers to send **Messages** to the main thread's message queue
  - Its **Looper** will then process them from that queue
- This is useful to let the main thread know when some data is available (may be useful for your CW)
  - The main thread can then use it as needed



Message: *HandlerTestProject1* (I)

- This example creates a Runnable in which to carry out the long-running task (within *doATaskAsynchronously();*)
  - Once result is calculated, a **`Message`** object is sent using an UI handler (created within *createUpdateUiHandler();*)

```
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
… //main Activity
private Handler updateUIHandler = null;
private final static int MESSAGE_UPDATE_TEXT_CHILD_THREAD = 1;
@Override protected void onCreate(Bundle savedInstanceState){
  …
  createUpdateUiHandler(); // Initializes the UI Handler
}
public void onClick(View view){
   if (view == performTaskButton){
       doATaskAsynchronously();
   }
}
```

This handler is in scope for all Runnables created inside the same Activity class

Custom constant to indicate the type of Message

Message: *HandlerTestProject1* (II)

- This action runs in a worker thread, and sends the result as a Message to the UI handler's message queue

```
private void doATaskAsynchronously(){
 // Code here will create a separate Thread to do some
 // work that may be time consuming
 Thread workerThread = new Thread(){
   @Override
   public void run(){
     String answer = calculation(); //get result to display
     // Build Message object
     Message message = new Message();
     message.what = MESSAGE_UPDATE_TEXT_CHILD_THREAD; //Set type
     message.obj = answer; //Set message payload
     // Send message to main thread Handler
     updateUIHandler.sendMessage(message);
   }
 };
 workerThread.start();
}
```

Message: *HandlerTestProject1* (III)

- The message handling runs on the UI thread (so it can also update UI widgets if we wish):
  - Gets a handler to the UI thread's **message queue looper**
  - overrides `handleMessage()` to process our own messages
    - ***on a message received, we use its contents as we wish!***

```
// Create Handler object in main thread
private void createUpdateUiHandler(){
  if(updateUIHandler == null){
    updateUIHandler = new Handler(Looper.getMainLooper()){
      @Override
      public void handleMessage(Message msg){
         // Check the message type
         if(msg.what == MESSAGE_UPDATE_TEXT_CHILD_THREAD){
            // Do something with data (update UI in this example)
            answerTextView.setText((String) msg.obj);
         }
      }
    };
  }
}
```

Executors. ExecutorTest1

- An **Executor** in the Java/Android world provides a way of selecting one or more threads from a **thread pool** to distribute work
  - This is alternative to using Thread. We can use Executor for the same purposes
  - The advantage is that threads in the **thread pool** are already created (just waiting to be used), so they start up faster!
- Example *ExecutorTest1* demonstrates how to use an Executor object to schedule a Runnable to carry out the blocking task:

```
@Override
public void onClick(View v){
  Executor myExecutor =
     Executors.newSingleThreadExecutor();
  myExecutor.execute(runnable);
}
```

Executors. Updating the UI (II)

- The runnable can update the UI as seen before:

```
private Runnable runnable = new Runnable() {
  @Override                                   Runs in worker thread
  public void run(){
    // Do some work that takes 50 milliseconds
    // a potentially time consuming task
    result = calculate() ;
    // Update the UI with progress
    runOnUiThread(new Runnable(){
       @Override
       public void run(){
          answerTextView.setText(result);
       }
    });                                       Runs in main thread
  }
};
```