App Development
in Android Studio

HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Lab 2: Threads

December 6, 2017

Jón Steinn Elíasson

jonsteinn@gmail.com

Contents

1 Java Threads

This section offers a brief introduction to threading in Java but anyone with prior knowledge is free to skip to the next section. The source code for all code examples is available here and a programming session here. The programming session is done in IntelliJ and there is a video of it being setup here

method	functionality
`Thread(Runnable runnable)`	Creates a new thread
`void start()`	Causes this thread to begin execution
`void join()`	Makes the calling thread wait until this one dies.
`boolean isAlive()`	Returns true iff thread has started and is not dead
`String getName()`	Returns the thread's name

Table 1: Parts of the Thread API

Threads, in its simplest form, are a way to execute two (or more) blocks of code at the same time while sharing certain resources. Each thread executes a run method, line by line, and once started, we have limited control over the order of which thread is being executed[1]. In the following we have three threads, the thread that starts the other two (the main thread) which will print `X` and `thread1` and `thread2`, printing `A` and `1` respectively. Running this program produces an nondeterministic output, in fact all possible permutations of `A`, `1` and `X` can be printed.

```
public static void main(String[] args) {
    Thread thread1 = new Thread(() -> System.out.print("A"));
    Thread thread2 = new Thread(() -> System.out.print("1"));
    thread1.start();
    thread2.start();
    System.out.print("X");
}
```

One reason to use threads is to keep an UI from locking while a long task is happening in the background. Here we see a task being run in the background while the main thread keeps the user informed. A more familiar version of this idea is a spinning wheel for the user to look at while he waits.

```
public class Main {
    public static void main(String[] args) {
        try {
            Thread carpenter = new Thread(() -> buildHouse(3));
            System.out.println("Ready to work");
            carpenter.start();
            while (true) {
                System.out.println("Work in progress...");
                carpenter.join(500); // wait 0.5s for thread to finish
                if (carpenter.isAlive()) continue;
                System.out.println("Work complete");
                break;
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    public static void buildHouse(int sec) {
        long timePoint = System.currentTimeMillis() + sec * 1000;
        while(System.currentTimeMillis() < timePoint); // loop for sec seconds
    }
}
```

A race conditions can occur when two threads access and edit the same resource in an unexpected order. Suppose thread A and B are both running and their job is to increment a shared variable X, which is set as 0. Thread A loads the value of X as 0 into registry when a context switch occurs and thread B loads the value of X into registry as 0 too, then B adds one and stores the value in X before another context switch when thread

[1] A context switch is when an operating system switches execution from one thread to another

A does the same, both adding 1 to 0, so the variable ends up as 1, not 2. The part of the code where race conditions can occur is called a critical section. We must pay close attention to them when dealing with threads.

In the program below we start thousand threads that increment two variables which are initially equal and then check if they are still equal, upon which we increment a counter. A sequential run of such a program would always increment the variable one at a time, check if they are equal which they will always be and then increment the counter, resulting in him being 1000 at the end. What can happen here is that thread i is running and has just incremented the variable `a` when a context switch occurs and thread j starts to run which now increments `a` again and then `b` and when done, checks if they are equal which they are not since thread i only incremented `a` so j will not increment the counter variable. Of course we can get lucky and everything happens in the correct order, even most of the time, but a slim chance of a race condition is all that is needed to make an entire program ruined.

```
public class Main {
    private static int counter = 0, a = 0, b = 0;
    public static void main(String[] args) {
        try {
            Thread[] threads = new Thread[1000];
            for (int i = 0; i < threads.length; i++) {
                threads[i] = new Thread(() -> {
                    if (++a == ++b) counter++;
                });
                threads[i].start();
            }
            for (Thread t : threads) t.join();
            System.out.println(counter);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

To deal with race conditions, we can identify the critical section and make sure that only one thread can operate there at any given time. That does however come at the cost of execution speed so we must place such lockouts wisely. When some thread arrives at a critical section, it should either be locked and make the thread wait or as soon as the thread 'steps in', the section should lock other threads out, until the current one is done. This can be done with the Java keyword `synchronized`. Synchronized methods can only be run by one thread at a time. Synchronized can also be used on scopes. Another approach is to use a mutex. One can think of a mutex like a single key which you must acquire to reach a critical section but once used, no one else can use it until the current key user has returned it after leaving the critical section. In Java, mutexes are instances of the class `Semaphore`, a more general case of a mutex for arbitrary number of keys (even a negative amount!). Using a Semaphore initialized with a key count of 1, the following mutex will fix our race condition even though, locking the entire part of a thread does not make good use of them.

```
threads[i] = new Thread(() -> {
  try {
    mutex.acquire();                    // Get the only key
    /* Critical section begins */
    if (++a == ++b) counter++;
    /* Critical section ends */
    mutex.release();                    // Give back the only key
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
});
```

Using such locks brings about another problem. What if Thread i is waiting on thread j and thread j is waiting on thread i? The waiting will go on forever and is called a deadlock. In the following code, suppose that the thread `t1` start and acquires `mutex1` but as soon as he has, a context switch occurs and the thread `t2` starts to run and acquires `mutex2`. Now it does not matter which thread is running, both mutexes are locked and both threads are unable to acquire the 'key'.

```
public class Main {
    public static void main(String[] args) {
        Semaphore mutex1 = new Semaphore(1);
        Semaphore mutex2 = new Semaphore(1);
        Thread t1 = new Thread(() -> {
            try {
                mutex1.acquire();
                // Thread 1 stuck here
                mutex2.acquire();
                mutex1.release();
                mutex2.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(() -> {
            try {
                mutex2.acquire();
                // Thread 2 stuck here
                mutex1.acquire();
                mutex2.release();
                mutex1.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t1.start();
        t2.start();
    }
}
```

Thread pools are used to gain more control over threads in Java. We can control the upper bound of how many threads are active at any given time from a pool. This is done with `Executor` and `ExecutorService`. Here we have a thread pool with a thread limit of five but 15 threads to start. As soon as 5 threads are already active, no more are added until at least one of them finishes.

```
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(5);
    for (int i = 0; i < 15; i++) {
        executor.execute(() -> {
            try {
                System.out.println(Thread.currentThread().getName() + ": Hi");
                Thread.sleep(1000);
                System.out.println(Thread.currentThread().getName() + ": Bye");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }/
        });
    }
    executor.shutdown();
}
```

2 The Android UI thread

Upon starting an Android app, a thread is created which will run our launcher activity and from there on, any component (which we will look at later). It also handles all events and interaction concerning the UI. No other threads should interact directly with the UI. This thread is called the main thread or the UI thread.

Suppose our app needs to run a long and expensive task, taking 10 seconds to execute. If we were to run that on the main thread, then it would not be able to listen for any events during the task and the UI would essentially be locked and eventually the operating system would warn the user that the app is not

responding. To avoid this, all expensive tasks should be performed by new threads in the background of the main one. Tasks like calling a web service, downloading and many more.

Just as the UI thread should not run long tasks, the vice versa holds, that long task running threads should never update the UI. The two rules of thumb are

- **Never run a long task on the UI thread**
- **Never update your UI from any thread other than the UI thread**

There are multiple Android specific ways to create and managing threads but we will only look at few here. Background tasks in Android can bring about some memory issues which we will address when looking at activities.

3 Asynchronous tasks

The abstract class `AsyncTask` provides an easy way to perform a background task which can talk to the UI thread. Asynchronous tasks are used for a short task required to run in a background thread, preferably not running longer than a few seconds. They are however bound to the lifetime of their activity.

The `AsyncTask` class has 3 generic types,

- `Params`. The type of parameter sent to task upon execution.
- `Progress`. The type of progress unit published during task.
- `Result` The type of result returned by the task.

All of these can be of the type `Void` which will omit that type from the task. Asynchronous tasks also has 4 stages,

- `onPreExecute`. Runs on the UI thread before task is executed. Used as a initializer for the background task.
- `doInBackground`. Runs on a separate thread, performing the actual task, publishing progress and returning the result.
- `onProgressUpdate`. Runs on the UI thread where receives progress updates from the background task.
- `onPostExecute`. Runs on the UI thread where it processes the result of the background task once done.

Additionally `onCancel` is called if the tasked is cancelled instead of `onPostExecute`. The cancellation should be done by the UI thread but once cancelled, the `doInBackground` method will not stop unless it regularly checks if the task has been canceled.

Asynchronous task should always be created and started by the UI thread. To start a asynchronous task, we can call `execute` or `executeOnExecutor`, the latter when using a Thread pool. To communicate with the UI thread we must either declare our task class (that inhertis from `AsyncTask`) as a subclass of our activity or use a callback interface.

In our provided example, we have a program that upon a button click creates a fake background job and updates a progress bar while working in the background. The job can be cancelled with another button and its status is displayed on the screen. The source code can be found here and the programming session here.

4 Loopers and Handlers

A looper provides a thread with a message queue (message being some data or job to process) and keeps checking if there is any data to process. Each thread can only have one looper but does not need to have

one[2]. The UI thread uses this mechanic to process events. Messages are handled by handlers. Each thread can have many handlers. Handlers are bound to a single thread that must have a Looper (and therefore, a message queue) and can post data to a Looper's message queue or process it. Handlers can add a message to the message queue in a thread from another thread that will be processed eventually. A handler can therefore easily supoort communication from some thread to the UI thread.

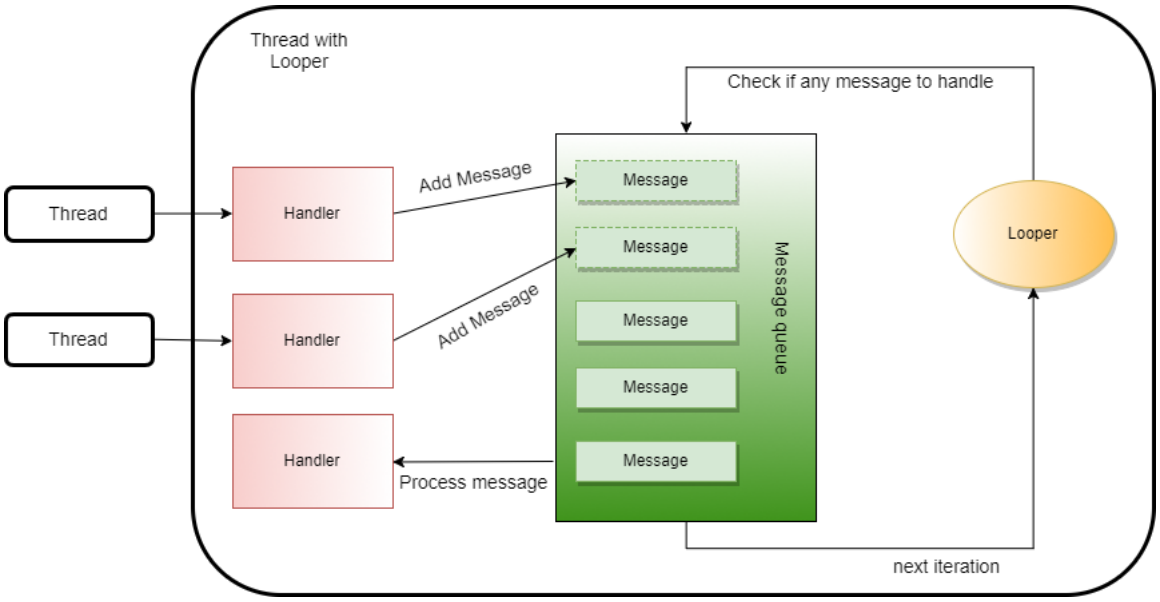

Figure 1: Simple calculator

A typical scenario would have a handler bound to the UI thread's Looper, being accessed from a second thread asking him to post a task to perform on the UI thread. This task would typically envolve UI updates from the second thread. This can be seen in the provided example, where upon clicking a switch we create a new thread which posts the task to disable the switch and start showing the progress bar to the UI thread. After that the new thread just waits for 5 seconds which can resemble any task taking some time and once he has finished he posts another task to the UI thread's message queue to update the UI once again. The source code can be found here and a programming session here.

5 RxAndroid

An Android specific binding for the RxJava library is a very powerful 3rd party option to handle background tasks. Its main components are schedulers, observers and observables. An observable can perform a background task while notifying an observer on a specified scheduler. Before we can use RxAndroid we must add the following dependencies in our `build.gradle`.

```
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
compile 'io.reactivex.rxjava2:rxjava:2.1.3'
```

In the provided example, we use schedulers, ovservers and observables to perform a fake time consuming task while keeping the UI thread from locking. The source code can be found here and a programming session here.

6 Assignment - Bouncing ball

Given a template for a view that draws graphics and a mathematical circle, your task is to make the circle move indefinitely in a diagonal direction but never leaving the screen (bouncing when colliding with its edges). You are free to use whatever threading technique you want to solve this task.

[2] A thread with a looper is called a looper thread

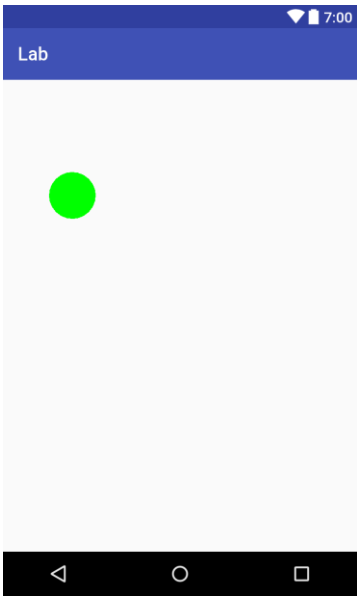

Figure 2: Bouncing ball

The following class should not be changed (other than the color, if you want) and you can use the `update` method to redraw the circle.

```
public class GraphicView extends View {
    private Paint paint;
    public GraphicView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        paint = new Paint();
        paint.setARGB(255, 0 /* R */, 255 /* G */, 0 /* B */);
        paint.setStyle(Paint.Style.FILL_AND_STROKE);
    }
    @Override
    public void onDraw(Canvas canvas) {
        canvas.drawCircle(
                Circle.getInstance().getX(),
                Circle.getInstance().getY(),
                Circle.getInstance().getRadius(),
                paint
        );
    }
    public void update() {
        this.invalidate();
    }
}
```

You are free to add what you need to the `Circle` class. It must however support accessing these 3 geometrical variables in the way that is provided.

```
public final class Circle {
    // Geometrical variables
    private float x;
    private float y;
    public final float radius;

    // Singleton pattern
    private static final Circle INSTANCE = new Circle();
    public static Circle getInstance() {
        return INSTANCE;
    }

    // A private constructor to guarantee a single instance
    private Circle() {
        x = 100f;
```

```
        y = 100f;
        radius = 50f;
    }

    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }

    public float getRadius() {
        return radius;
    }
}
```

You can get the width and the height of the view with their `getWidth` and `getHeight` methods for the collision test. Note that the coordinate system of the canvas has the y value grow downwards as is shown in figure 3.

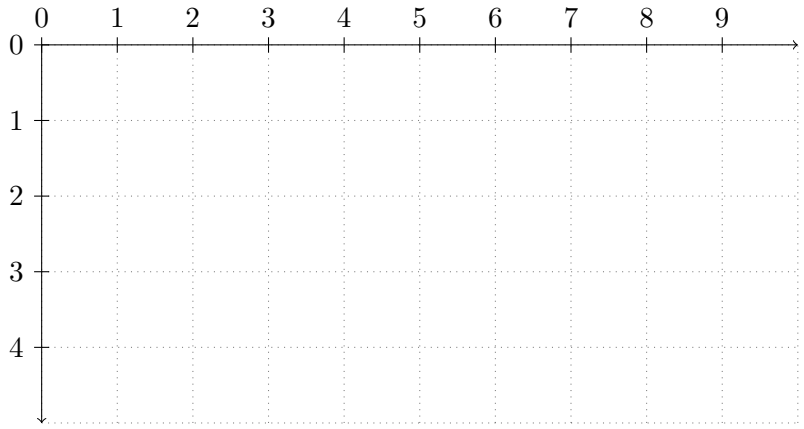

Figure 3: Canvas coordinate system