

ANDROID NÂNG CAO

BÀI 1: Multithreading + Background Works

Nội dung

1. Multithreading

- Threads
- Multithreading
- Ưu/nhược điểm của multithread

2. Tiếp cận của Android

3. Handler

- Messages
- Runnable object

4. AsyncTask

5. Timer

Phần 1

Multithreading

Threads

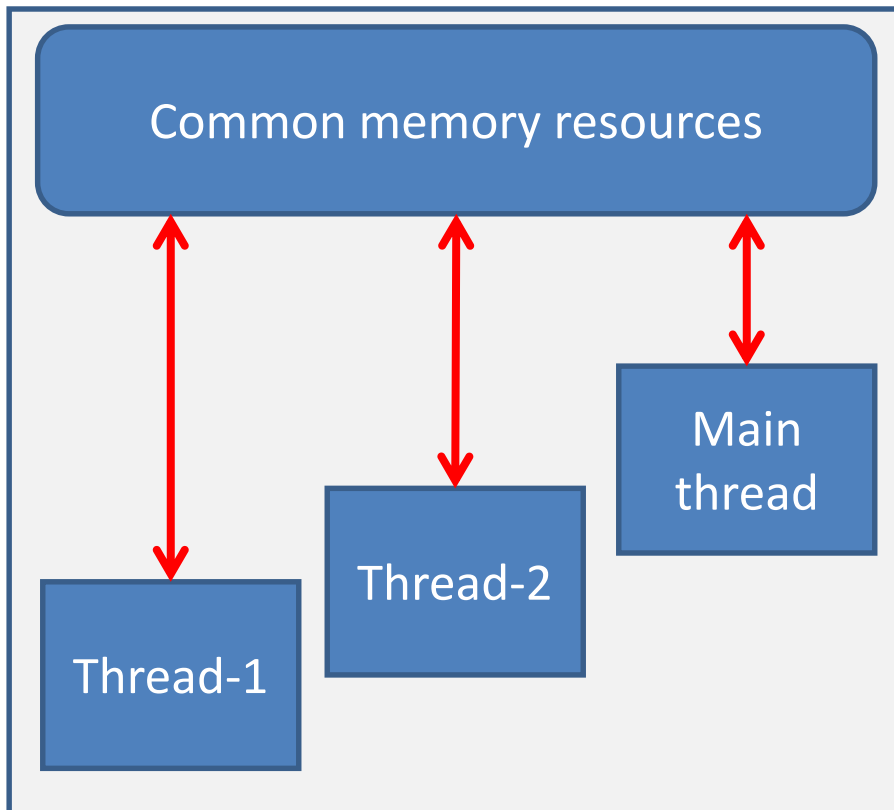
- **Process** (tiến trình): đơn vị thực thi nhỏ nhất quản lý ở mức độ hệ điều hành; mỗi process được cấp bộ nhớ, tài nguyên và lượng CPU riêng; các process không ảnh hưởng lẫn nhau
- **Thread** (mạch/luồng): đoạn mã được thực thi bởi CPU một cách liên tục; chia sẻ bộ nhớ, tài nguyên và CPU với các thread khác thuộc cùng process
- **Application** (ứng dụng) khi chạy có 1 thread chính, ứng dụng kết thúc khi mọi thread của nó kết thúc, ứng dụng có thể tạo thêm các thread con nếu cần

Threads

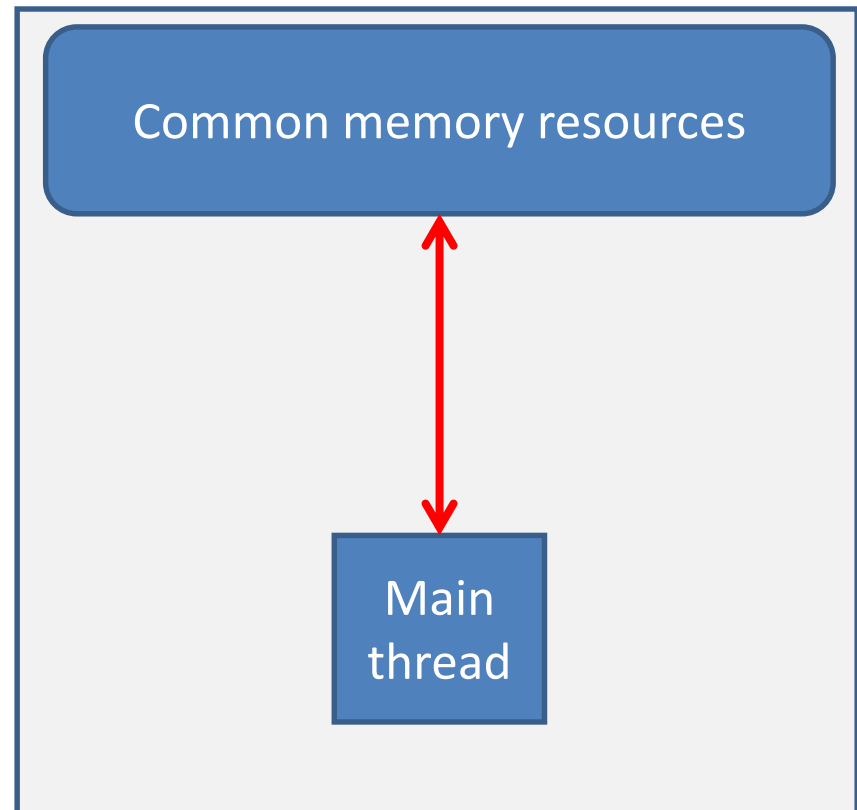
- Mỗi thread có thuộc tính **priority** là mức độ ưu tiên của thread đó, độ ưu tiên càng cao thì lượng CPU cấp cho nó càng nhiều
- Thread có độ ưu tiên thấp nhất là **daemon thread** (**idle thread** – trong Windows), chỉ chạy khi CPU rỗi
- Các thread trong cùng một process tương tác và đồng bộ hóa với nhau qua việc sử dụng các đối tượng dùng chung và các biến monitor
- Java dùng cơ chế **synchronized** và **wait-notify** để xử lý tình huống tranh chấp tài nguyên giữa các thread

Threads

Process 1 (Dalvik Virtual Machine 1)



Process 2 (Dalvik Virtual Machine 2)



Multithreading

- **Multi-user** (đa người dùng): phục vụ cùng lúc nhiều người dùng
- **Multi-tasking** (đa nhiệm): chạy đồng thời nhiều process
- **Multi-threading** (đa luồng): thực thi đồng thời nhiều thread trong cùng một process
 - Với CPU đơn, việc xử lý multithreading là giả lập
 - Với CPU nhiều nhân hoặc luồng, multithreading thực sự là thực hiện song song (parallel), việc thực hiện song song nâng cao sức mạnh thiết bị lên nhiều lần nhưng cũng phức tạp hóa việc lập trình

Multithreading trong java

// viết thread theo cách thứ nhất

```
class MyThread extends Thread {  
    public void run() {  
        // phần thực thi của thread viết ở đây  
    }  
}
```

// viết thread theo cách thứ hai

```
class MyRunnable implements Runnable {  
    public void run() {  
        // phần thực thi của thread viết ở đây  
    }  
}
```

// tạo và chạy các thread

```
MyThread t1 = new MyThread(); t1.start();  
MyRunnable t2 = new MyRunnable(); new Thread(t2).start();
```


Ưu/nhược điểm của multithread

- Tận dụng tốt năng lực của các thiết bị đa nhân
- Trong một số tình huống giải quyết được tình trạng nghẽn cổ chai, kết quả là hệ thống đa luồng sẽ vận hành nhanh hơn đáng kể so với đơn luồng
 - Ví dụ: nếu một tiến trình chờ dữ liệu từ I/O lúc này CPU có thể rảnh rỗi, nhưng trên hệ thống đa luồng thì CPU sẽ được chuyển ngay cho thread khác để làm việc
- Lập trình viên chỉ cần tập trung vào việc viết tốt mã cho một nhiệm vụ cụ thể mà không cần quan tâm tới việc thực thi đa luồng thế nào, vì thế công việc của người viết code trở nên đơn giản hơn

Ưu/nhược điểm của multithread

- **Lập trình viên phải làm quen với kiến thức mới và một số nguyên tắc khi viết ứng dụng**
 - Android duy trì một main thread chạy UI
 - Không cho phép thực hiện các thao tác I/O trên main thread (kết nối và lấy dữ liệu từ mạng, đọc dữ liệu từ stream,...)
 - Các thread con không được trực tiếp làm việc với UI
 - Chia sẻ và đồng bộ dữ liệu giữa các tiến trình
- **Ứng dụng đa luồng khó gỡ lỗi hơn, đặc biệt là khi xảy ra lỗi giữa các thread**
- **Việc phát hiện và giải quyết deadlock rất phức tạp**

Phần 2

Tiếp cận của Android

Cách tiếp cận của Android

- **Android vẫn hỗ trợ các phương pháp làm việc với thread truyền thống của Java**
- **Các bổ sung của android hướng tới 2 mục tiêu**
 - Đơn giản hóa việc trao đổi dữ liệu giữa các thread
 - Phối hợp tốt hơn giữa UI thread và các thread khác
- **Để đảm bảo trải nghiệm người dùng với UI, có hai giải pháp cho những thao tác xử lý nặng**
 - Thực hiện thao tác đó trong một service và dùng hệ thống notification để thông báo cho người dùng
 - Thực hiện thao tác đó trong một background work

Cách tiếp cận của Android

- **UI thread là thread đảm bảo việc xử lý các công việc liên quan tới giao diện**
- **UI thread cần tốc độ nên loại thread-unsafe (nhanh nhưng không an toàn khi làm việc với thread)**
 - Hệ quả là các thread muốn cập nhật giao diện sẽ phải xử lý phức tạp hơn
- **Android bổ sung hai phương pháp để các thread làm việc với nhau và với UI thread**
 - AsyncTask
 - Handler

Cách tiếp cận của Android

- **Handler** là cơ chế trao đổi kiểu ủy thác, trong đó handler đóng vai trò đối tượng trung gian trong các giao tiếp giữa các thread, thread ủy thác công việc cho handler theo hai cách
 - Ra lệnh cho handler bằng message
 - Truyền cho handler đối tượng thực thi kiểu Runnable
- Một số cơ chế cũng sử dụng handler nhưng che dấu phần code phức tạp
- **AsyncTask** là cơ chế cho phép lập trình viên định nghĩa công việc theo mẫu có sẵn của android

4 kiểu mã cơ bản

```
Runnable x = new Runnable() {  
    public void run() {  
        // cập nhật giao diện ...  
    }  
};
```

```
1/ MainActivity.this.runOnUiThread(x);  
2/ MainActivity.this.myView.post(x);  
3/ new Handler().post(x);
```

```
private class BackgroundTask extends AsyncTask<String, Void, Bitmap> {  
    protected void onPostExecute(Bitmap result) {  
        // cập nhật giao diện ...  
    }  
}
```

Phần 3

Handler

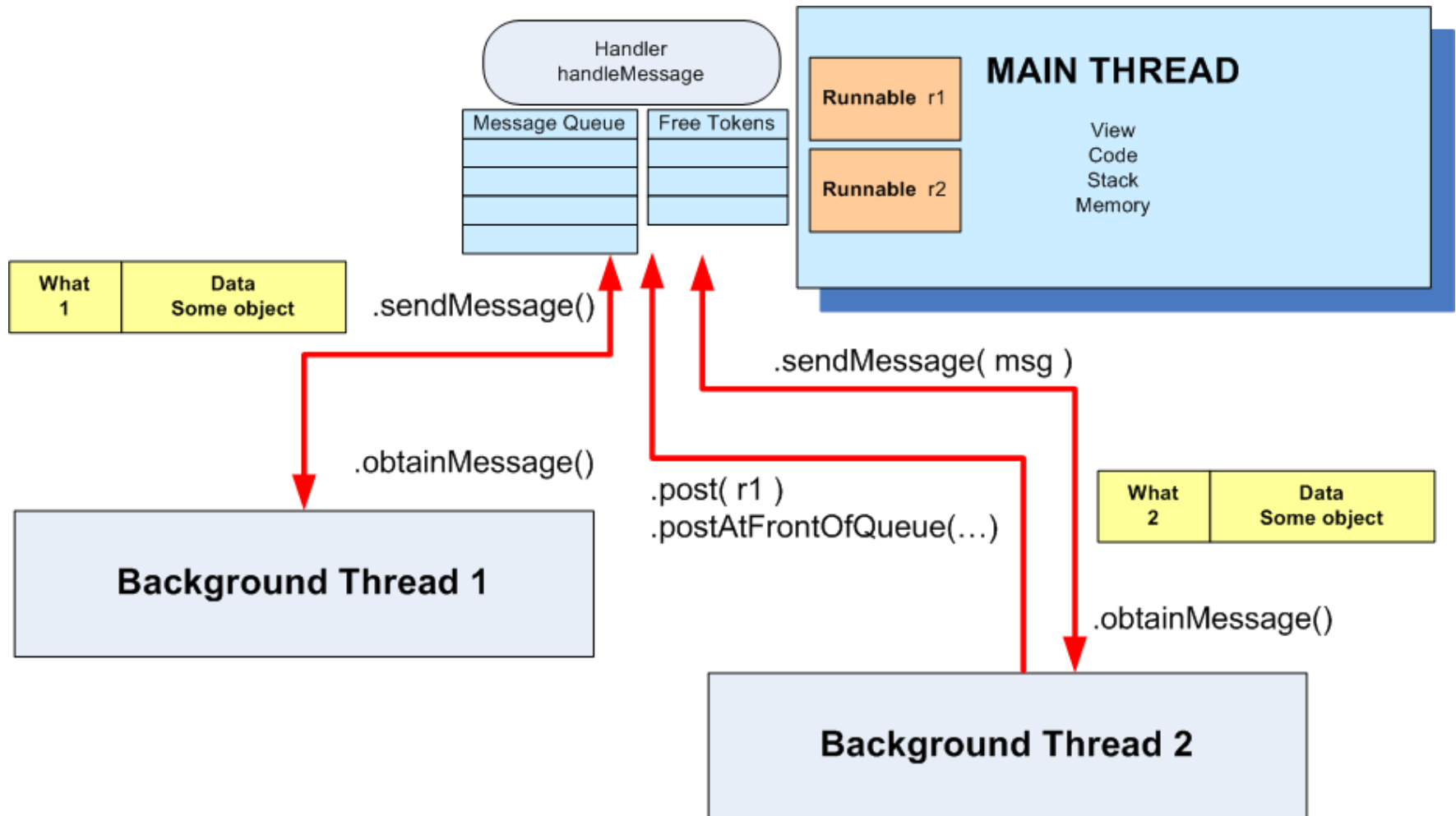
Handler

- Handler là đối tượng đặc biệt, chuyên sử dụng để tiếp nhận các yêu cầu từ thread khác
- Khi được tạo ra bằng `new Handler()`, đối tượng handler tự động liên kết với thread hiện tại
- Handler có một hàng đợi thông điệp (`message pool`)
- Các thread khác có thể “nhờ vả” handler bằng cách
 - Gửi một message đến handler
 - Gửi một đối tượng Runnable đến handler
- Khi thread hiện tại “rảnh rỗi”, nó cho phép handler lấy các thông điệp hoặc Runnable ra thực thi

Handler – message

- Thread phụ cần liên lạc với thread chính thì cần lấy một message token từ pool của handler bằng cách gọi **obtainMessage**
- Có token, thread phụ ghi yêu cầu vào token và gửi nó cho handler bằng cách dùng **sendMessage**, token được đặt vào pool để đợi handler xử lý
- Mỗi khi có message trong pool, phương thức **handleMessage** sẽ được gọi ra để xử lý message đó
- Như vậy handler cần viết lại **handlerMessage** để xử lý các message một cách phù hợp

Handler – message



Handler – message

Main Thread

```
// đối tượng myHandler được tự động
// gắn với main thread
Handler hdr = new Handler() {
    @Override
    public void handleMessage(Message x)
    {
        // xử lý các message
    }
};
```

Background Thread

```
job = new Thread (new Runnable () {
    @Override
    public void run() {
        // lấy msg khỏi pool
        Message msg = hlr.obtainMessage();
        // điền dữ liệu vào msg
        ...
        // gửi lại msg cho handler
        hlr.sendMessage(msg);
    }
});
// chạy thread
job.start();
```

Using Messages

Handler – gửi message

Handler cung cấp nhiều cách gửi message sử dụng trong các tình huống khác nhau, tất cả các hàm này đều trả về false nếu thất bại

- `sendMessage(msg)`: đặt message vào cuối pool
- `sendMessageToFrontOfQueue(msg)`: đặt message vào đầu pool (chứ không phải cuối như mặc định)
- `sendMessageAtTime(msg, T)`: chờ đến thời điểm T thì đặt message vào pool, T đo bằng millisecond theo uptime của hệ thống (`SystemClock.uptimeMillis()+x`)
- `sendMessageDelayed(msg, T)`: đặt message vào queue sau T millisecond

Handler – xử lý message

- Để xử lý các message mà các thread khác gửi đến, handler cần viết lại phương thức
`public void handleMessage(Message x)`
- Phương thức này sẽ được tự động gọi mỗi khi thread chính rảnh rỗi và trong message pool có xuất hiện message
- Trong khi `handleMessage` được gọi, handler có thể cập nhật UI nếu cần. Tuy nhiên, nó cần làm việc đó thật nhanh, vì các nhiệm vụ UI khác bị treo cho đến khi handler thực hiện xong

Handler message example

```
public class MainActivity extends Activity {
    TextView txt;
    Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            txt.setText(txt.getText()+"Item "+msg.getData().getString("x")+"\n");
        }
    };
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txt = (TextView) findViewById(R.id.txt);
    }
    protected void onStart() {
        super.onStart();
        background.start();
    }
}
```

Handler message example

```
Thread background = new Thread(new Runnable() {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            try {  
                Thread.sleep(1000);  
                Message msg = new Message();  
                Bundle b = new Bundle();  
                b.putString("x", "My Value: " + i);  
                msg.setData(b);  
                handler.sendMessage(msg);  
            } catch (Exception e) {}  
        }  
    }  
});  
}
```

Multi Threading Demo

```
Item My Value: 0  
Item My Value: 1  
Item My Value: 2  
Item My Value: 3  
Item My Value: 4  
Item My Value: 5  
Item My Value: 6  
Item My Value: 7  
Item My Value: 8  
Item My Value: 9
```


Handler – post runnable object

```
new Thread(new Runnable() {
    private Bitmap load(String url) {
        InputStream ist = (InputStream) new URL(url).getContent();
        return BitmapFactory.decodeStream(ist);
    }
    public void run() {
        final Bitmap bitmap = load("http://example.com/image.png");
        mImageView.post(new Runnable() {
            public void run() { mImageView.setImageBitmap(bitmap); }
        });
    }
}).start();
```

Phần 4

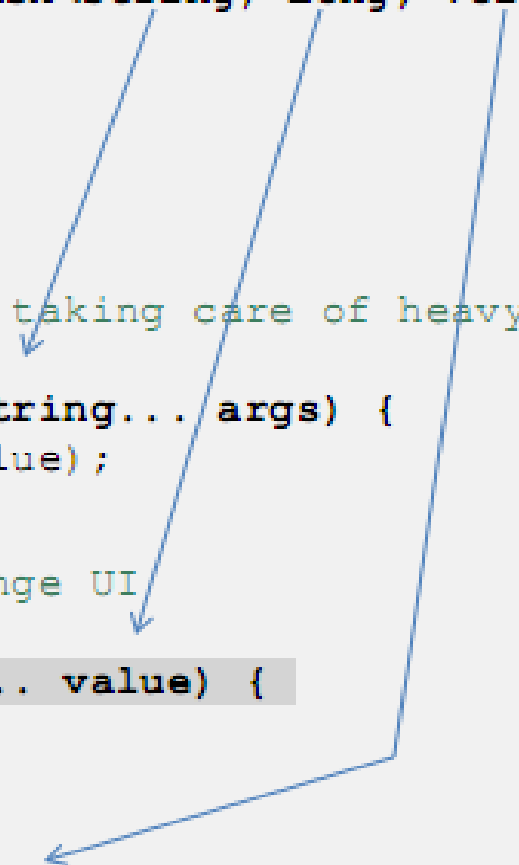
AsyncTask

AsyncTask

- Chúng ta có thể thấy hầu hết các công việc chạy nền đều tuân theo một kịch bản rất giống nhau, đó là quá trình 4 bước cơ bản:
(chuẩn bị) => ((chạy) <=> (cập nhật)) => (kết thúc)
- Android cung cấp mẫu AsyncTask theo kịch bản đó
- AsyncTask cho phép tiến trình nền dễ dàng cập nhật các UI thread mà không quan tâm tới handler hay những cơ chế tương tự
- Không phải công việc chạy nền nào cũng theo kịch bản trên (nghĩa là vẫn cần thread)

AsyncTask

```
private class VerySlowTask extends AsyncTask<String, Long, Void> {  
  
    // Begin - can use UI thread here  
    protected void onPreExecute() {  
  
    }  
  
    // this is the SLOW background thread taking care of heavy tasks  
    // cannot directly change UI  
    protected Void doInBackground(final String... args) {  
        ... publishProgress((Long) someLongValue);  
    }  
  
    // periodic updates - it is OK to change UI  
    @Override  
    protected void onProgressUpdate(Long... value) {  
  
    }  
  
    // End - can use UI thread here  
    protected void onPostExecute(final Void unused) {  
  
    }  
  
}
```



AsyncTask

3 kiểu tổng quát
(generic)

Params,
Progress,
Result

4 trạng thái chính

onPreExecute,
doInBackground,
onProgressUpdate,
onPostExecute

1 phương thức hỗ trợ

publishProgress

Các tham số kiểu trong AsyncTask

AsyncTask <Params, Progress, Result>

Params: kiểu dữ liệu tham số phương thức **doInBackground**, đây là các dữ liệu sẽ được gửi cho background thread

Progress: kiểu dữ liệu sẽ được gửi cho **onProgressUpdate** để công bố kết quả lên UI thread

Result: kiểu dữ liệu của kết quả tính toán do **doInBackground** trả về, dữ liệu này cũng là tham số của phương thức **onPostExecute**

■ Chú ý:

- Không phải AsyncTask nào cũng dùng đến cả ba kiểu dữ liệu. Đối với kiểu không dùng đến, ta dùng kiểu “Void”
- Cú pháp “String...” tương tự “String[]”

Hoạt động của AsyncTask

AsyncTask's methods

onPreExecute: được gọi tại UI thread để chuẩn bị cho việc chạy background. Hàm này thường dùng để setup tác vụ, chẳng hạn để hiện một progress bar tại UI

doInBackground: được gọi tại background thread ngay sau khi ***onPreExecute*** chạy xong. Hàm này thực hiện các tính toán background. Hàm phải trả về kết quả tính toán và sẽ được truyền thành tham số của ***onPostExecute***. Trong khi thực thi, hàm này có thể dùng ***publishProgress*** để báo cáo tiến độ hoạt động lên UI, hàm ***publishProgress*** sẽ tự động gọi ***onProgressUpdate*** vào thời điểm phù hợp

onProgressUpdate: được gọi tự động bởi ***publishProgress***. Thời điểm thực thi không xác định. Hàm này thường dùng để hiển thị thông báo tiến độ lên UI. Ví dụ cập nhật progress bar hoặc hiện log trong một text field

onPostExecute: được gọi bởi UI thread sau khi công việc tính toán tại background đã xong. Kết quả tính toán của background được truyền cho hàm này bằng tham số

Ví dụ: layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/readwebpage"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="readwebpage"
        android:text="Load webpage" >
    </Button>

    <TextView
        android:id="@+id/TextView01"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Example Text" >
    </TextView>

</LinearLayout>
```


Ví dụ

```
class DownloadWebPageTask extends AsyncTask<String, Void, String> {  
    protected String doInBackground(String... urls) {  
        String response = "", s = "";  
        for (String url : urls) {  
            DefaultHttpClient client = new DefaultHttpClient();  
            HttpGet httpGet = new HttpGet(url);  
            try {  
                InputStream content =  
                    client.execute(httpGet).getEntity().getContent();  
                BufferedReader buffer =  
                    new BufferedReader(new InputStreamReader(content));  
            }  
        }  
    }  
}
```

Ví dụ

```
        while ((s = buffer.readLine()) != null) response += s;
    } catch (Exception e) {}
}
return response;
}
protected void onPostExecute(String result) {
    textView.setText(result);
}
}
```

Ví dụ

```
public class ReadWebpageAsyncTask extends Activity {
    private TextView textView;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        textView = (TextView) findViewById(R.id.TextView01);
    }
    public void readWebpage(View view) {
        DownloadWebPageTask task = new DownloadWebPageTask();
        task.execute(new String[] { "http://www.mobipro.vn" });
    }
}
```

Phần 5

Timer

Timer

- Timer là tính năng có sẵn của Java chứ không phải là sáng tạo của android
- Cho phép lập lịch và thực thi một tác vụ vào một thời điểm định sẵn
- Tác vụ có thể thực hiện 1 lần hoặc lặp lại sau một khoảng thời gian
- Timer không hoạt động đúng theo thời gian thực, vì thế cần thận trọng khi sử dụng class này đối với các tác vụ thời gian thực
- Bộ đôi Timer và TimerTask luôn đi với nhau

Timer

- **Timer**: class giúp ta đặt lịch
- **TimerTask**: class giúp ta định nghĩa công việc cần thực hiện công việc khi có lịch
- Viết lại TimerTask bằng cách viết lại hàm “public void run()” (tương tự như thread, bản thân TimerTask implements Runnable)
- TimerTask có một số hàm hữu ích:
 - boolean cancel(): giúp hủy tác vụ
 - long scheduledExecutionTime(): trả về thời điểm thực hiện tác vụ lần cuối

Timer

- Hai phương thức cơ bản giúp đặt lịch chạy một tác vụ:
 - `schedule(TimerTask, when)`: thực hiện `TimerTask` vào thời điểm `when`
 - Ví dụ: `timer.schedule(tasknew, new Date());`
 - `scheduleAtFixedRate(TimerTask, when, period)`: thực hiện lặp đi lặp lại `TimerTask` bắt đầu từ thời điểm `when` và lặp lại sau `period` millisecond
- Một số phương thức thú vị nên tìm hiểu thêm
 - `schedule(TimerTask task, long delay, long period)`
 - `schedule(TimerTask task, long delay)`

Timer

```
TextView tv = (TextView) findViewById(R.id.main_timer_text);
Timer t = new Timer();
t.scheduleAtFixedRate(new TimerTask() {
    public void run() {
        runOnUiThread(new Runnable() {
            public void run() {
                tv.setText(String.valueOf(time++));
            }
        });
    }
}, 0, 1000);
```