

안드로이드 프로세스와 스레드

안드로이드 프로세스와 스레드

2

■ 안드로이드는 기본적으로 멀티 프로세스^{Process}와 멀티 스레드^{Thread}를 지원한다.



예를 들어 한 회사를 프로세스로 비유했을 때 회사 내 각종 서류, 책상, 의자, 컴퓨터 등이 자원에 해당하고, 그러한 자원을 가지고 일하는 직원이 작업자라고 할 수 있겠다. 여기서 자원은 **메모리**고 작업자는 **스레드**에 해당한다.

각 프로세스 간에는 자원을 공유할 수 없다. 그러므로 프로세스는 다른 프로세스 간섭 없이 완벽하게 독립적으로 처리되며, 만일 특정 프로세스에 문제가 발생하더라도 다른 프로세스에 영향을 주지 않고 자신만 종료하게 된다.

스레드란?

3

■ 스레드를 이용하면 동시 처리가 가능하다. 단 하나의 스레드는 단 하나의 실행 흐름만 가지지만, 여러 개의 스레드를 생성하면 여러 실행 흐름을 가질 수 있기 때문이다.



스레드란?

4

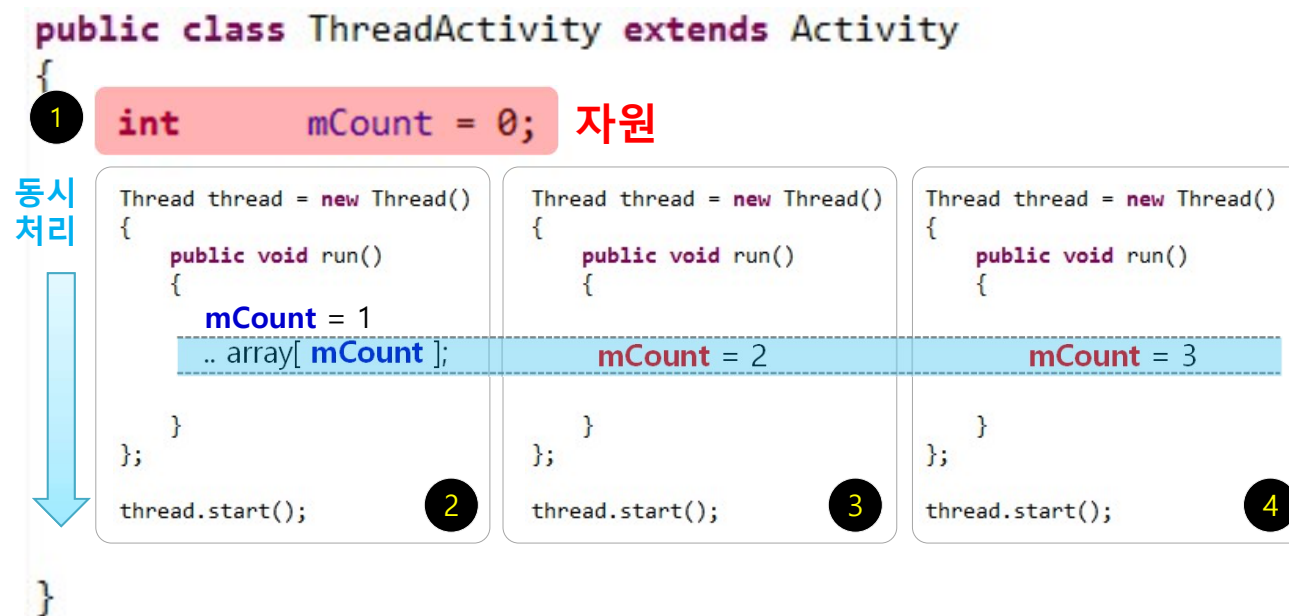
- 스레드를 생성하는 방법은 매우 간단하다.

```
Thread thread = new Thread()  
{  
    public void run()  
    {  
        ↓ 처리해야 할 일  
    }  
};  
thread.start();
```

스레드란?

5

- 하지만 스레드를 관리하는 것이 쉽지 않다.



따라서 스레드에서 공유되는 자원은 잘 관리해주어야 한다. (synchronized 등의 구문 이용...)

메인 스레드와 작업 스레드 – 메인 스레드

6

■ 하나의 프로세스에는 최소 하나의 스레드가 존재해야 한다. 따라서 스레드가 존재하지 않는 프로세스는 종료된 프로세스라고 보면 된다. 그러므로 앱이 실행되면 프로세스와 스레드 하나가 꼭 생성된다.

안드로이드에서 앱이 실행될 때 기본으로 생성되는 스레드를 메인Main 스레드라 한다. 메인 스레드는 각종 생명주기 함수들을 처리하고 화면에 그림을 그리는 등의 역할을 한다.

직접 메인 스레드를 확인해보자.

메인 스레드와 작업 스레드 – 메인 스레드

7

- Application name: Thread
- Company Domain: company.co.kr
- Package name: kr.co.company.thread

메인 스레드와 작업 스레드 - 메인 스레드

8

□ activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="20dp" >

    <TextView
        android:id="@+id/count_textview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Count :" />

</LinearLayout>
```


메인 스레드와 작업 스레드 - 메인 스레드

9

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);
```

```
        // 1. 생명주기 onCreate 함수가 호출되었을 때 로그를 출력해준다.
```

```
        // =====
```

```
        Log.d("superdroid", "onCreate()");
```

```
        // =====
```

```
        // 2. 레이아웃을 액티비티에 반영한다.
```

```
        // =====
```

```
        setContentView(R.layout.activity_main);
```

```
        // =====
```

```
    }
```

```
}
```

메인 스레드와 작업 스레드 – 메인 스레드

The screenshot shows the Android Studio interface with the Android Device Monitor. The 'Devices' tab is selected, showing a list of processes. The 'Threads' tab is also visible, showing a list of threads for the selected process. The 'LogCat' tab is at the bottom, showing a log message for the 'kr.co.company.thread' process.

Devices List:

Name	Online
emulator-5554	Online
com.google.android.googlequicksearchbox:search	2112
com.android.calendar	2626
com.android.inputmethod.latin	1859
com.google.android.gms.unstable	3078
com.google.android.gms	2567
com.android.exchange	2952
com.google.android.gms.persistent	1931
system_process	1559
android.process.acore	2007
com.android.systemui	1688
com.android.phone	1946
com.google.process.gapps	2141
com.android.settings	2589
com.android.providers.calendar	2658
android.process.media	1701
com.android.launcher3	1962
com.android.deskclock	2541
com.google.android.apps.photos	3183
com.google.android.googlequicksearchbox:interactor	1840
kr.co.company.thread	2289

Threads List:

ID	Tid	Status	utime	stime	Name
1	2289	Runnable	15	16	main
*2	2294	Wait	0	0	Signal Catcher
*3	2295	Runnable	6	6	JDWP
*4	2297	Wait	0	0	FinalizerDaemon
*5	2296	Wait	0	0	ReferenceQueueDaemon
*6	2299	Monitor	0	1	HeapTaskDaemon
*7	2298	Wait	0	0	FinalizerWatchdogDaemon
8	2300	Runnable	0	0	Binder_1
9	2301	Runnable	0	0	Binder_2
10	2326	Runnable	0	0	Binder_3
11	2473	Runnable	0	0	Binder_4
12	2610	Runnable	0	0	Thread-82
13	2622	Runnable	0	1	RenderThread
14	2623	Runnable	0	0	hwuiTask1

LogCat:

tag:superdroid

L...	Time	PID	TID	Application	Tag	Text
D	11-17 04:25:04.381	2289	2289	kr.co.company.thread	superdroid	onCreate ()

메인 스레드와 작업 스레드 - 메인 스레드의 한계

11

- 개발자가 별도의 작업 스레드를 만들지 않는 이상 구현되는 모든 코드는 메인 스레드에서 동작한다.
그러나 아쉽게도 모든 작업을 메인 스레드 처리할 수는 없다.

메인 스레드와 작업 스레드 - 메인 스레드와 ANR

12

□ activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="20dp" >

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="ANR Test"
        android:onClick="onClick"/>

</LinearLayout>
```

메인 스레드와 작업 스레드 – 메인 스레드와 ANR

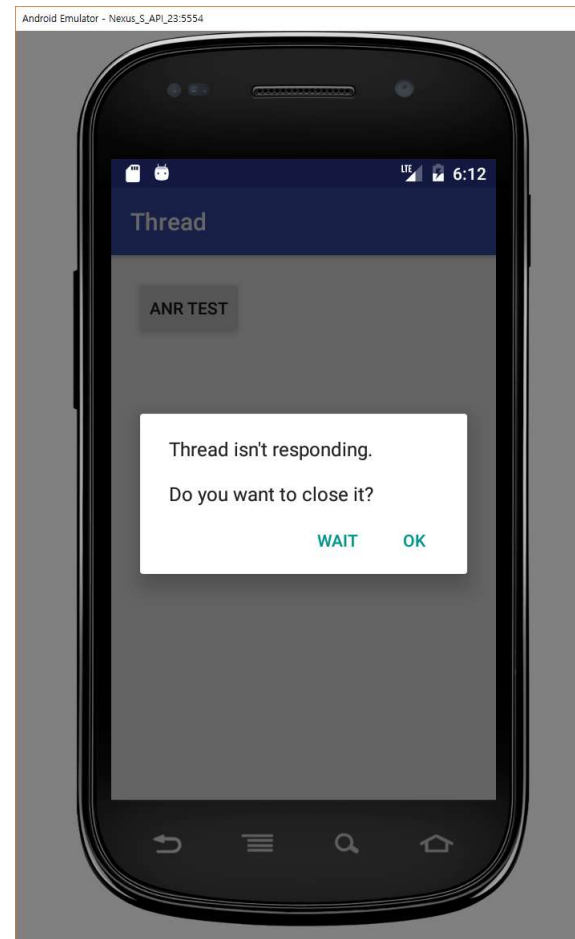
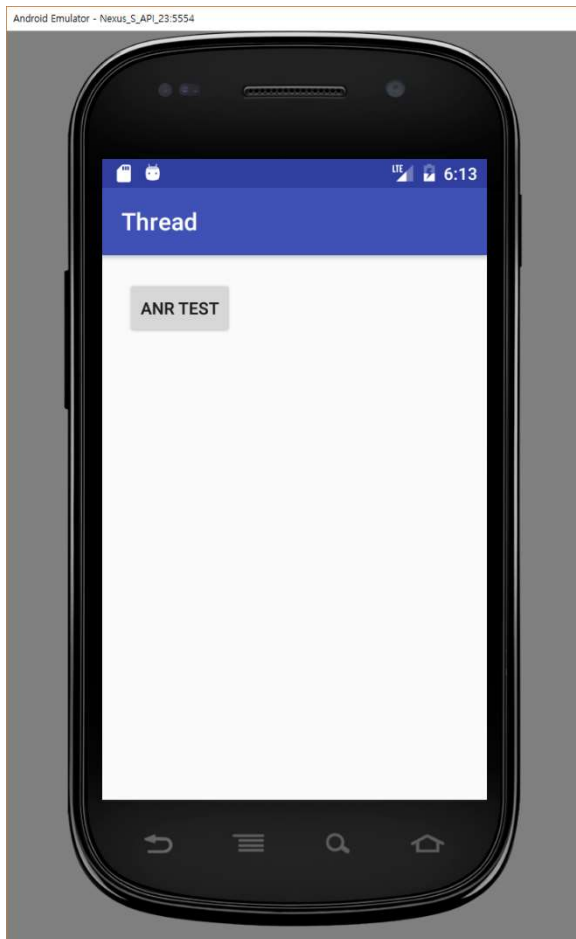
13

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
  
    public void onClick(View v) {  
        try {  
            for (int i = 0; i < 100; i++)  
                Thread.sleep(100);  
        } catch ( InterruptedException e ) {  
            e.printStackTrace();  
        }  
    }  
}
```

메인 스레드와 작업 스레드 - 메인 스레드와 ANR

14



메인 스레드와 작업 스레드 – 메인 스레드와 ANR

15

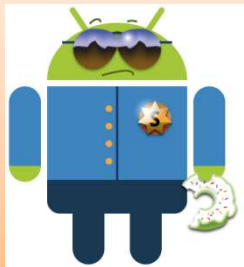
- 메인 스레드는 사용자와 상호 작용하는 작업을 처리하기 때문에 **UI 스레드** 라고도 부른다.
- 안드로이드는 **반응성(Responsiveness)**을 굉장히 중요하게 여기며 액티비티 관리자와 윈도우 관리자가 항상 백그라운드에서 응용 프로그램의 반응성을 감시한다.
- 안드로이드는 다음 조건 중 하나가 발견되면, **ANR(Application Not Responding)** 대화상자를 띄운다.
 - ▣ 액티비티가 5초 이상 사용자의 입력에 반응하지 않을 때
 - ▣ 브로드캐스트 리시버가 10초 혹은 60초 내로 리턴하지 않을 때
 - ▣ 서비스가 20초 동안 메인 스레드를 잡고 있을 때

메인 스레드와 작업 스레드 - 메인 스레드와 스트릭트 모드

16

■ 안드로이드는 앱이 끊김 없이 원활히 동작하도록 메인 스레드에서 처리할 수 없는 일들을 정의했다.

메인 스레드



1. 디스크에서 파일 쓰기
2. 디스크에서 파일 읽기
3. 네트워크 사용

안드로이드에서는 크게 세 가지 사항을 메인 스레드 사용 위반 사례로 보는데 이를 스트릭트 모드 Strict Mode 위반이라 부른다.

특히 네트워크 사용은 네트워크 상태와 서버의 사정에 따라 소모되는 시간을 유추하기 힘들고, 시간이 많이 소모될 확률이 크다. 그러므로 **API 10부터는 기본으로 메인 스레드의 네트워크 사용을 강제적으로 막고 있다.**

메인 스레드와 작업 스레드 - 메인 스레드와 스트릭트 모드

17

□ AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="kr.co.company.thread">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <uses-permission android:name="android.permission.INTERNET" />

        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

메인 스레드와 작업 스레드 - 메인 스레드와 스트릭트 모드

18

□ MainActivity.java

```
public void onClick(View v) {  
    try {  
        URL url = new URL("http://m.naver.com");  
  
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();  
        if (connection != null) {  
            connection.setConnectTimeout(10000);  
            connection.setRequestMethod("GET");  
            connection.setDoInput(true);  
            connection.setDoOutput(true);  
        }  
  
        int responseCode = connection.getResponseCode();  
        if (responseCode == HttpURLConnection.HTTP_OK) {  
            BufferedReader reader = new BufferedReader(  
                new InputStreamReader(connection.getInputStream()));  
            String line = null;  
            while (true) {  
                line = reader.readLine();  
                if (line == null) break;  
            }  
            reader.close();  
            connection.disconnect();  
        }  
    } catch (Exception e) {  
        Log.e("superdroid", "Exception on click");  
        e.printStackTrace();  
    }  
}
```

activity_main.xml x MainActivity.java x AndroidManifest.xml x

```

26     if (connection != null) {
27         connection.setConnectTimeout(10000);
28         connection.setRequestMethod("GET");
29         connection.setDoInput(true);
30         connection.setDoOutput(true);
31     }
32
33     int responseCode = connection.getResponseCode();
34     if (responseCode == HttpURLConnection.HTTP_OK) {
35         BufferedReader reader = new BufferedReader(
36             new InputStreamReader(connection.getInputStream()));
37         String line = null;
38         while (true) {
39             line = reader.readLine();
40             if (line == null) break;
41         }
42         reader.close();
43         connection.disconnect();
44     }
45 } catch (Exception e) {
46     Log.e("superdroid", "Exception on click");

```

o.company.thread (17436)

Verbose

```

ad E/superdroid: Exception on click
ad W/System.err: android.os.NetworkOnMainThreadException
ad W/System.err: at android.os.StrictMode$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1273)
ad W/System.err: at java.net.InetAddress.lookupHostByName(InetAddress.java:431)
ad W/System.err: at java.net.InetAddress.getAllByNameImpl(InetAddress.java:252)
ad W/System.err: at java.net.InetAddress.getAllByName(InetAddress.java:215)
ad W/System.err: at com.android.okhttp.internal.Network$1.resolveInetAddresses(Network.java:29)
ad W/System.err: at com.android.okhttp.internal.http.RouteSelector.resetNextInetSocketAddress(RouteSelect
ad W/System.err: at com.android.okhttp.internal.http.RouteSelector.nextProxy(RouteSelector.java:157)
ad W/System.err: at com.android.okhttp.internal.http.RouteSelector.next(RouteSelector.java:100)
ad W/System.err: at com.android.okhttp.internal.http.HttpEngine.createNextConnection(HttpEngine.java:357)
ad W/System.err: at com.android.okhttp.internal.http.HttpEngine.nextConnection(HttpEngine.java:340)
ad W/System.err: at com.android.okhttp.internal.http.HttpEngine.connect(HttpEngine.java:330)
ad W/System.err: at com.android.okhttp.internal.http.HttpEngine.sendRequest(HttpEngine.java:248)
ad W/System.err: at com.android.okhttp.internal.huc.HttpURLConnectionImpl.execute(HttpURLConnectionImpl.j
ad W/System.err: at com.android.okhttp.internal.huc.HttpURLConnectionImpl.getResponse(HttpURLConnectionIm
ad W/System.err: at com.android.okhttp.internal.huc.HttpURLConnectionImpl.getResponseCode(HttpURLConnection
ad W/System.err: at kr.co.company.thread.MainActivity.onClick(MainActivity.java:33) <1 internal calls>

```

메인 스레드와 작업 스레드 - 작업 스레드의 필요성

20

- 그렇다면 해결 방법은 무엇일까? 간단하다. 메인 스레드에서 긴 작업을 처리하지 않으면 된다.
즉 긴 작업은 모두 작업 스레드에서 처리한다.

메인 스레드와 작업 스레드 - 작업 스레드의 필요성

21

□ activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="20dp" >

    <TextView
        android:id="@+id/count_textview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Count :" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Get Current Count"
        android:onClick="onClick"/>

</LinearLayout>
```

메인 스레드와 작업 스레드 - 작업 스레드의 필요성

22

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {
```

```
    int mCount = 0;
```

```
    TextView mCountTextView = null;
```

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

```
        // 현재까지 카운트 된 수를 텍스트뷰에 출력한다.
```

```
        // =====
```

```
        mCountTextView.setText( "Count : " + mCount );
```

```
        // =====
```

```
    }
```

메인 스레드와 작업 스레드 – 작업 스레드의 필요성

```
□ MainActivity.java @Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // 1. 생명주기 onCreate 함수가 호출되었을 때 로그를 출력해 둔다.
    // =====
    Log.d("superdroid", "onCreate()");
    // =====

    // 2. 레이아웃을 액티비티에 반영 및 텍스트뷰 객체를 얻어온다.
    // =====
    mCountTextView = (TextView)findViewById(R.id.count_textview);
    // =====

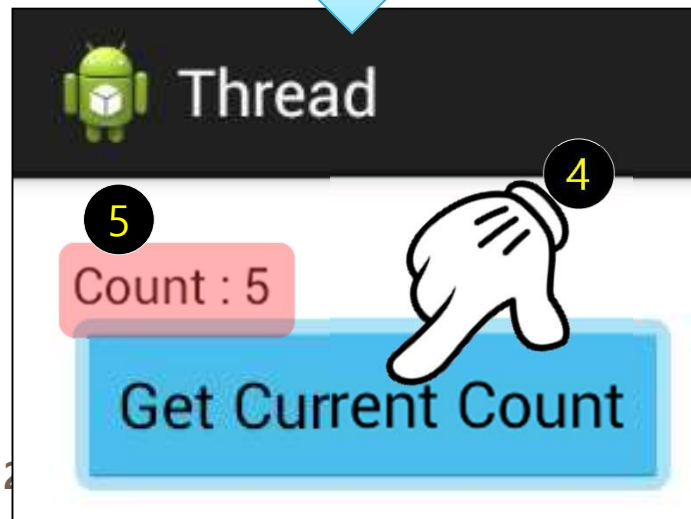
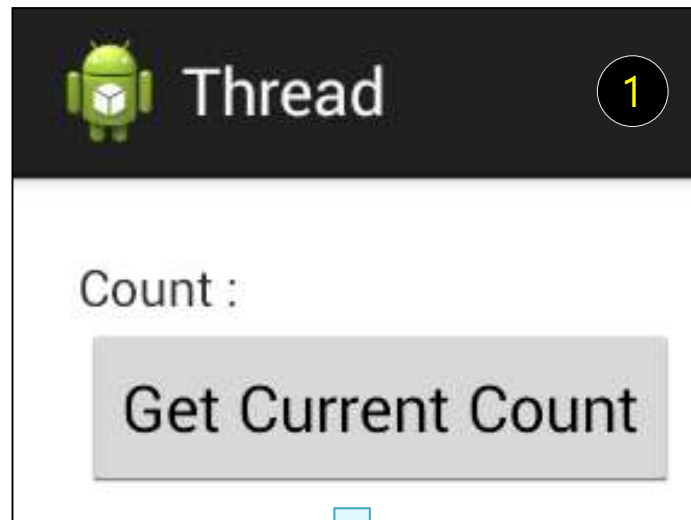
    // 3. 10초 동안 1씩 카운트하는 스레드 생성 및 시작
    // =====
    Thread countThread = new Thread("Count Thread") {
        public void run() {
            for (int i = 0 ; i < 10 ; i++) {
                mCount++;

                // 현재 카운트된 값을 로그로 출력한다.
                // -----
                Log.i("superdroid", "Current Count : " + mCount);
                // -----

                try {
                    Thread.sleep(1000);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    };

    countThread.start();
    // =====
}
```

메인 스레드와 작업 스레드 - 작업 스레드의 필요성



● DDMS 스레드 정보

ID	Tid	Status	utime	stime	Name
1	943	Native	23	10	main
*2	948	VmWait	0	0	GC
*3	949	VmWait	0	0	Signal Catcher
...					
12	959	Native	0	0	Binder_4
13	960	TimedW...	0	0	Count Thread

● 로그 출력 결과

PID	TID	Text
943	943	onCreate ()
943	960	Current Count : 1
943	960	Current Count : 2
943	960	Current Count : 3

메인 스레드와 작업 스레드 - 작업 스레드의 필요성

■ 메인 스레드와 작업 스레드를 자바 코드 흐름으로 살펴보자.

메인 스레드

```
protected void onCreate( )  
{  
    ...  
    Thread countThread ...  
    {  
        ...  
    };  
}
```

1 **countThread.start();**

...
...
...
...
...
...
...
...
...
...



시간 흐름

동시 실행
구간

카운트 스레드

```
public void run() 2  
{  
    for ( int i = 0 ; i < 10 ; i ++ )  
    {  
        try { Thread.sleep( 1000 ); }  
        catch ...  
        mCount ++;  
        ...  
    }  
}
```

메인 스레드와 작업 스레드 - 화면에 그리는 작업을 할 수 없는 작업 스레드

26

■ 안드로이드에서는 메인 스레드가 아닌 다른 스레드에서 화면에 그리는 작업을 허용하지 않는다.

왜 이러한 제약사항이 존재할까?

여러 스레드에서 뷰를 변경하여 화면을 갱신한다면 동기화 문제가 발생할 수 있기 때문이다.

즉 동시에 실행되는 스레드는 어느 것이 더 빨리 처리될지 순서를 알 수 없고,

화면에 뷰를 그리는 것은 순서가 매우 중요하다.

그러므로 처리되는 순서가 불규칙한 스레드에서 그리는 작업을 하면 화면은 뒤죽박죽이 될 수 있다.

따라서 안드로이드에서는 **메인 스레드에서만 그리는 작업을 허용**하여 그리는 순서를 보장하며,

이를 **단일 스레드 GUI** Graphical User Interface **모델**이라 한다.

즉 단일 스레드 GUI는 하나의 스레드에서만 그리는 작업을 처리하는 것을 말한다.

■ 그렇다면 작업 스레드에서 화면을 그리면 어떻게 될까?

메인 스레드와 작업 스레드 - 화면에 그리는 작업을 할 수 없는 작업 스레드

27

- MainActivity.java
 - ▣ countThread의 run() 메소드만 수정

// 3. 10초 동안 1씩 카운트하는 스레드 생성 및 시작

// =====

```
Thread countThread = new Thread("Count Thread") {
```

```
    public void run() {
```

```
        for (int i = 0 ; i < 10 ; i++) {
```

```
            mCount ++;
```

```
            // 현재까지 카운트 된 수를 텍스트뷰에 출력한다.
```

```
            mCountTextView.setText("Count : " + mCount);
```

```
        try {
```

```
            Thread.sleep(1000);
```

```
        }
```

```
        catch (InterruptedException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

```
};
```

메인 스레드와 작업 스레드 - 화면에 그리는 작업을 할 수 없는 작업 스레드

28

● 로그 출력 결과

```
FATAL EXCEPTION: Count Thread  
android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original th  
read that created a view hierarchy can touch its views.
```

Thread(이)가 중지되었습니다.

확인

따라서 작업 스레드에서 그리는 작업이 필요하다면 메인 스레드로 이관해야 한다.

```
public class MainActivity extends AppCompatActivity {
```

```
...  
Handler mHandler = new Handler();  
...
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
```

```
...
```

```
// 3. 10초 동안 1씩 카운트하는 스레드 생성 및 시작
```

```
// =====
```

```
Thread countThread = new Thread("Count Thread") {
```

```
    public void run() {
```

```
        for (int i = 0 ; i < 10 ; i++) {
```

```
            mCount ++;
```

```
            mHandler.post(new Runnable() {
```

```
                @Override
```

```
                public void run() {
```

```
                    // 현재까지 카운트 된 수를 텍스트뷰에 출력한다.
```

```
                    Log.i("superdroid", "Count : " + mCount);
```

```
                    mCountTextView.setText("Count : " + mCount);
```

```
                }
```

```
            });
```

```
        try {
```

```
            Thread.sleep(1000);
```

```
        }
```

```
        catch (InterruptedException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

```
};
```

```
countThread.start();
```

```
// =====
```

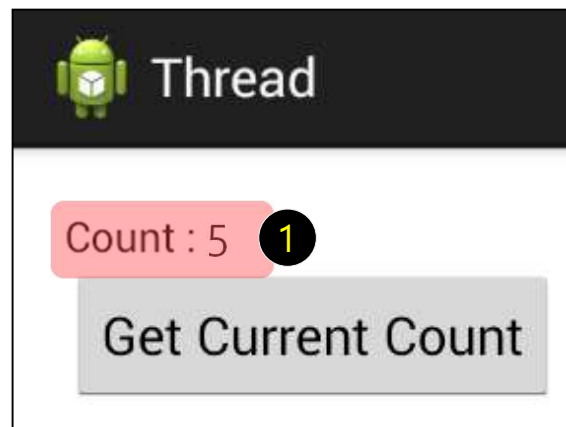
```
}
```

```
}
```

Handler 객체를 통해
그리는 작업을 메인 스레드로
이관하고 있다.

메인 스레드와 작업 스레드 - 화면에 그리는 작업을 할 수 없는 작업 스레드

30



● DDMS 스레드 정보

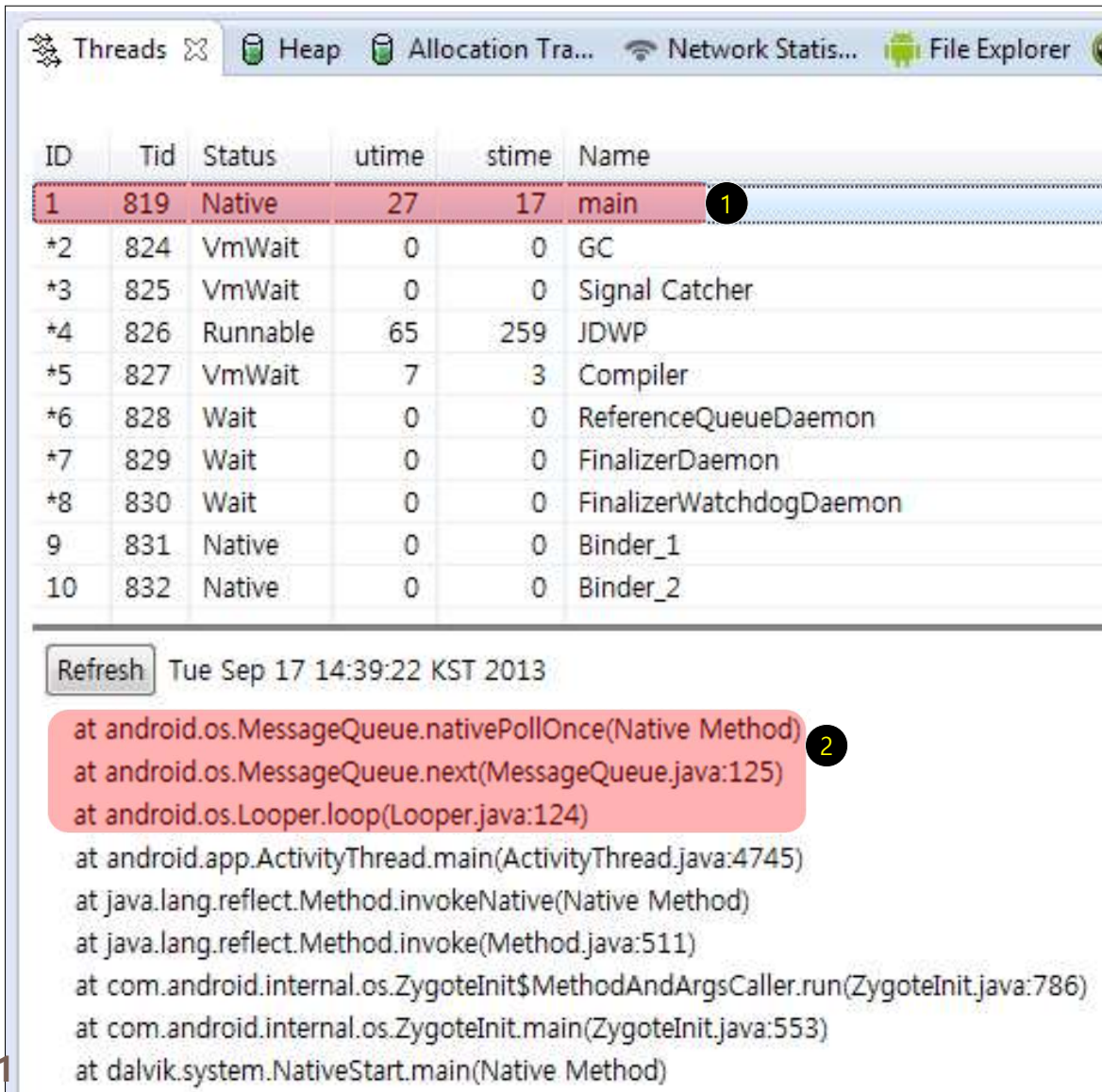
ID	Tid	Status	u...	s...	Name
1	819	Native	16	15	main 2
*2	824	VmWait	0	0	GC
...					
11	833	Timed...	0	0	Count Thread

● 로그 출력 결과

PID	TID	Application	Text
819	819	com.superdroid.t...	onCreate()
819 3	819	com.superdroid.t...	Count : 1
819	819	com.superdroid.t...	Count : 2
819	819	com.superdroid.t...	Count : 3
819	819	com.superdroid.t...	Count : 4

Handler에 대해서 설명하려면 메인 스레드의 구조를 이해해야 한다.

안드로이드 메인 스레드 구조



ID	Tid	Status	utime	stime	Name
1	819	Native	27	17	main
*2	824	VmWait	0	0	GC
*3	825	VmWait	0	0	Signal Catcher
*4	826	Runnable	65	259	JDWP
*5	827	VmWait	7	3	Compiler
*6	828	Wait	0	0	ReferenceQueueDaemon
*7	829	Wait	0	0	FinalizerDaemon
*8	830	Wait	0	0	FinalizerWatchdogDaemon
9	831	Native	0	0	Binder_1
10	832	Native	0	0	Binder_2

Refresh Tue Sep 17 14:39:22 KST 2013

```
at android.os.MessageQueue.nativePollOnce(Native Method)
at android.os.MessageQueue.next(MessageQueue.java:125)
at android.os.Looper.loop(Looper.java:124)
at android.app.ActivityThread.main(ActivityThread.java:4745)
at java.lang.reflect.Method.invokeNative(Native Method)
at java.lang.reflect.Method.invoke(Method.java:511)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
at dalvik.system.NativeStart.main(Native Method)
```

메인 스레드 콜 스택을 보면
메시지 큐와
루퍼 객체를 계속 사용하고 있다.

안드로이드 메인 스레드 구조 - 루퍼와 메시지 큐

32

■ 메인 스레드의 루퍼

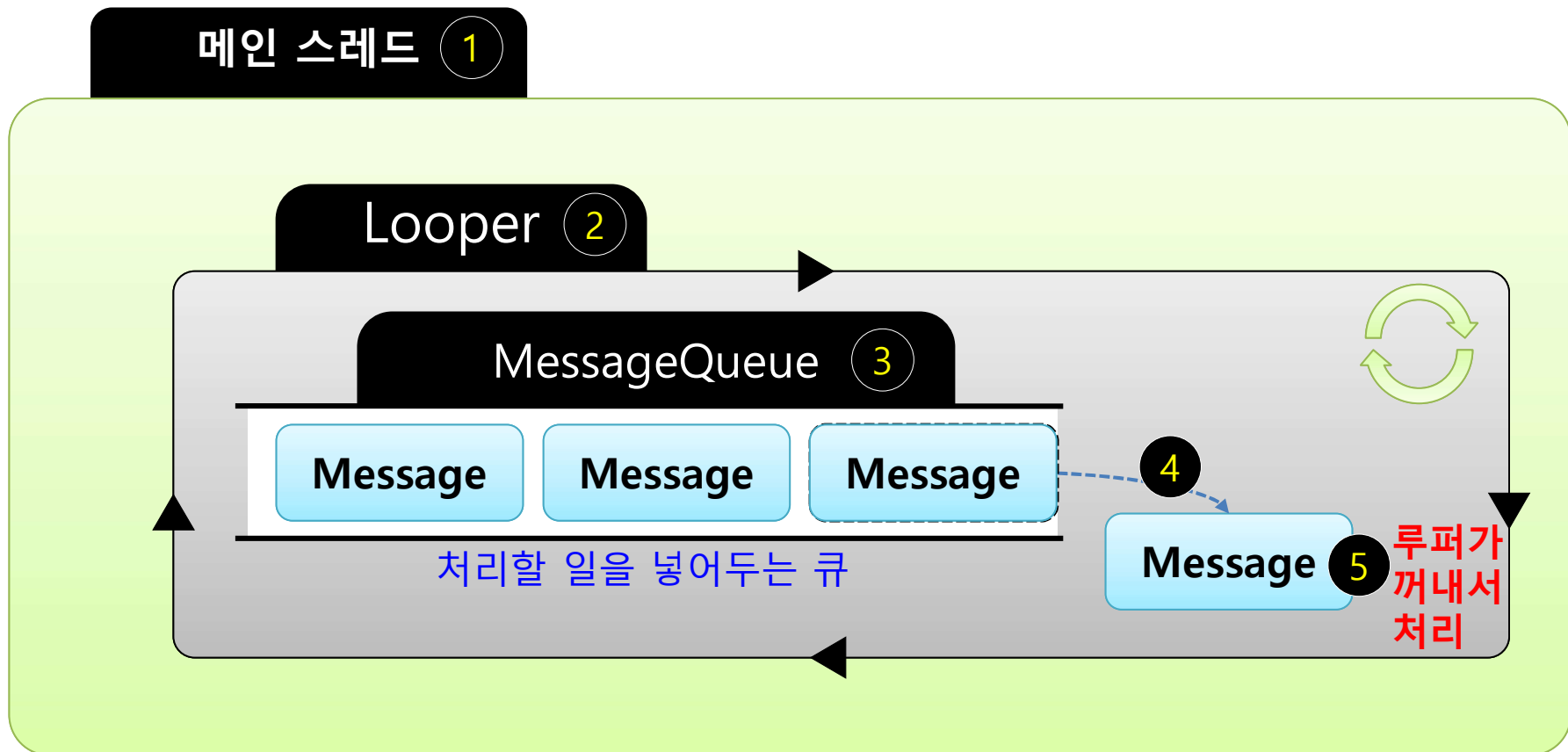


루퍼는 어떤 작업을 하며 끊임없이 반복하고 있을까?

안드로이드 메인 스레드 구조 - 루퍼와 메시지 큐

33

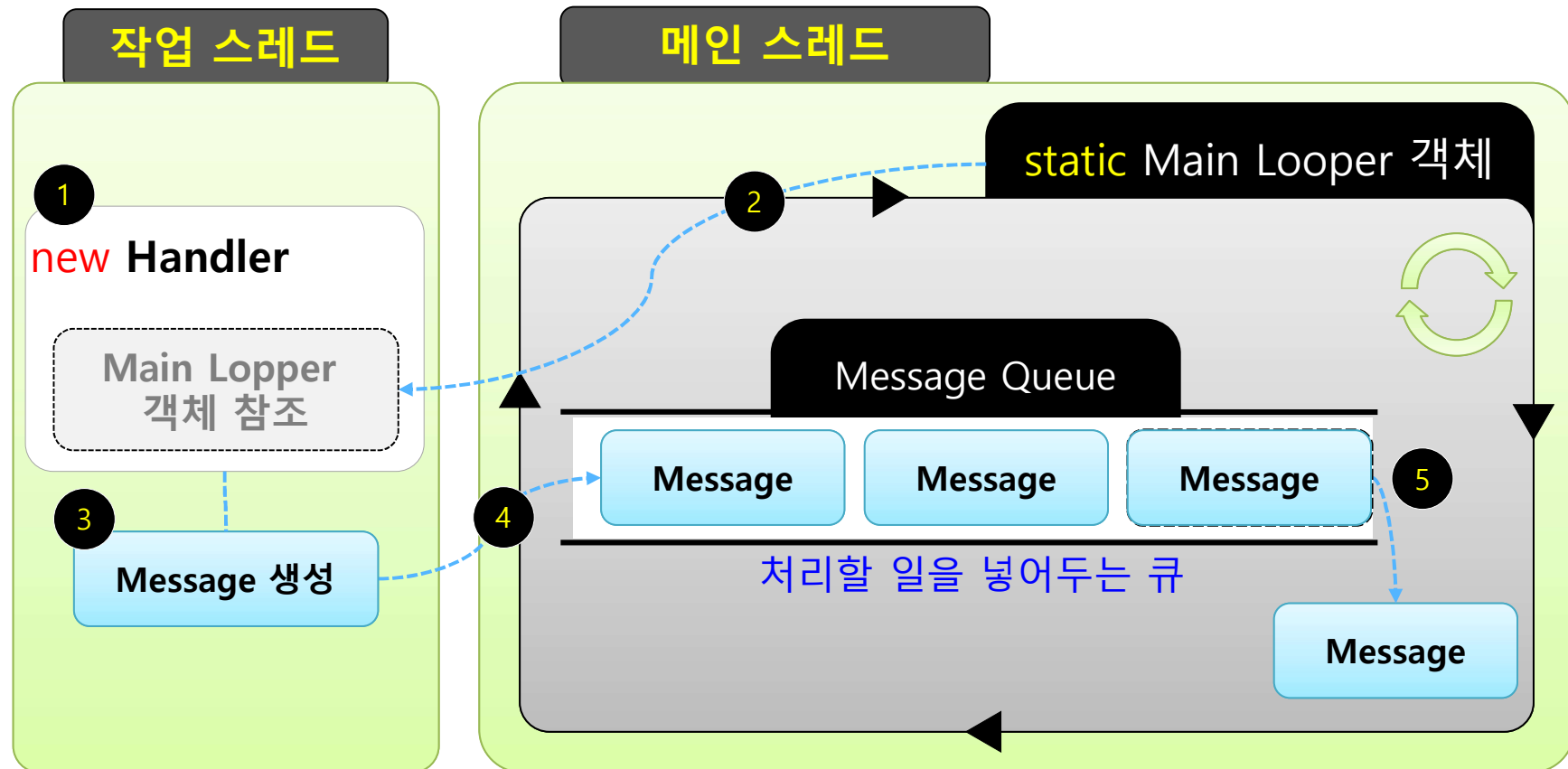
■ 루퍼와 메시지 큐



그렇다면 메시지 큐에 처리할 일에 해당하는 메시지를 추가하는 것은 어떤 것이 담당할까?

안드로이드 메인 스레드 구조 - 핸들러

34



작업 스레드에서만 핸들러를 생성할 수 있는 것은 아니다.

어디서든 핸들러를 생성하면 핸들러 내부적으로 메인 스레드의 루퍼 객체를 참조한다.

이 것이 가능한 이유는 메인 스레드의 루퍼 객체 자체가 **static** 변수로 정의되어 있기 때문이다.

안드로이드 메인 스레드 구조 – 핸들러

35

프레임워크 소스 : `Looper.java`

```
public class Looper
{
    ...

    private static Looper sMainLooper;
    final MessageQueue mQueue;
    ...

    public static Looper getMainLooper()
    {
        synchronized (Looper.class) {
            return sMainLooper;
        }
    }
}
```

핸들러는 내부적으로
루퍼의 `getMainLooper` 함수를
통해 루퍼 객체를 참조한다.

루퍼 객체를 참조하면
루퍼를 이용해서 메시지 큐에
메시지를 추가할 수 있다.

□ MainActivity.java

- ▣ countThread의 run() 메소드만 수정

```
// 3. 10초 동안 1씩 카운트하는 스레드 생성 및 시작
// =====
Thread countThread = new Thread("Count Thread") {
    public void run() {
        for (int i = 0 ; i < 10 ; i++) {
            mCount ++;

            // 1) 실행 코드가 담긴 Runnable 객체를 하나 생성한다.
            // -----
            Runnable callback = new Runnable() {
                @Override
                public void run() {
                    // 현재까지 카운트 된 수를 텍스트뷰에 출력한다.
                    // -----
                    Log.i("superdroid", "Count : " + mCount);
                    mCountTextView.setText("Count : " + mCount);
                    // -----
                }
            };
            // -----

            // 2) 메시지 큐에 담을 메시지 하나를 생성한다. 생성시
            //     Runnable 객체를 생성자로 전달한다.
            // -----
            Message message = Message.obtain(mHandler, callback);
            // -----

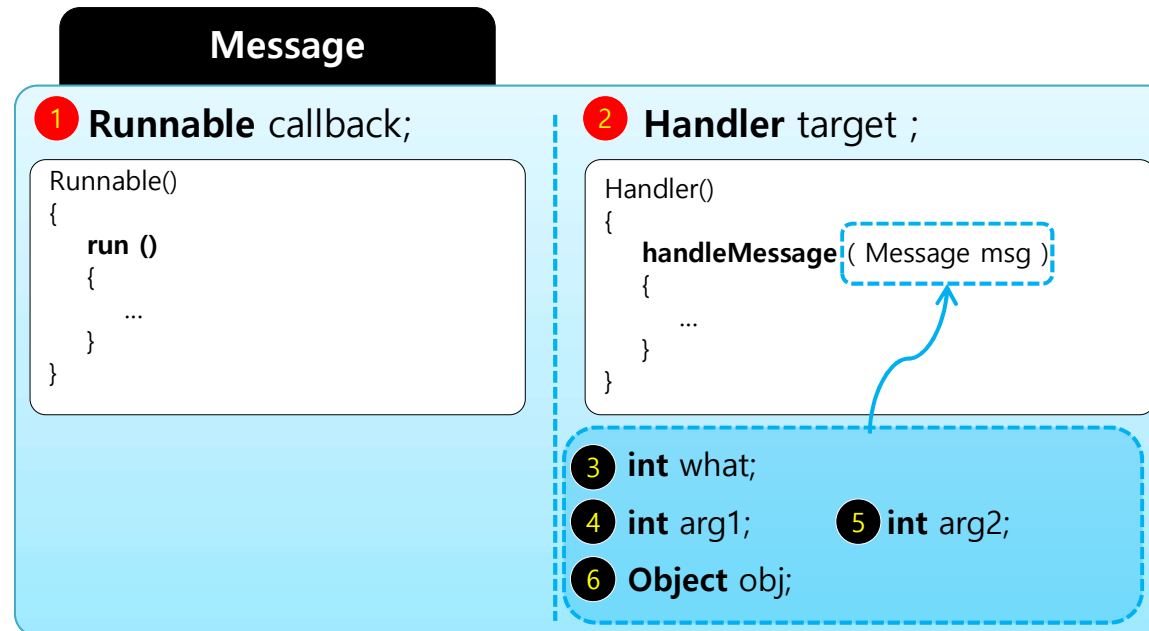
            // 3) 핸들러를 통해 메시지를 메시지 큐에 보낸다.
            // -----
            mHandler.sendMessage(message);
            // -----

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};
```

안드로이드 메인 스레드 구조 - 핸들러

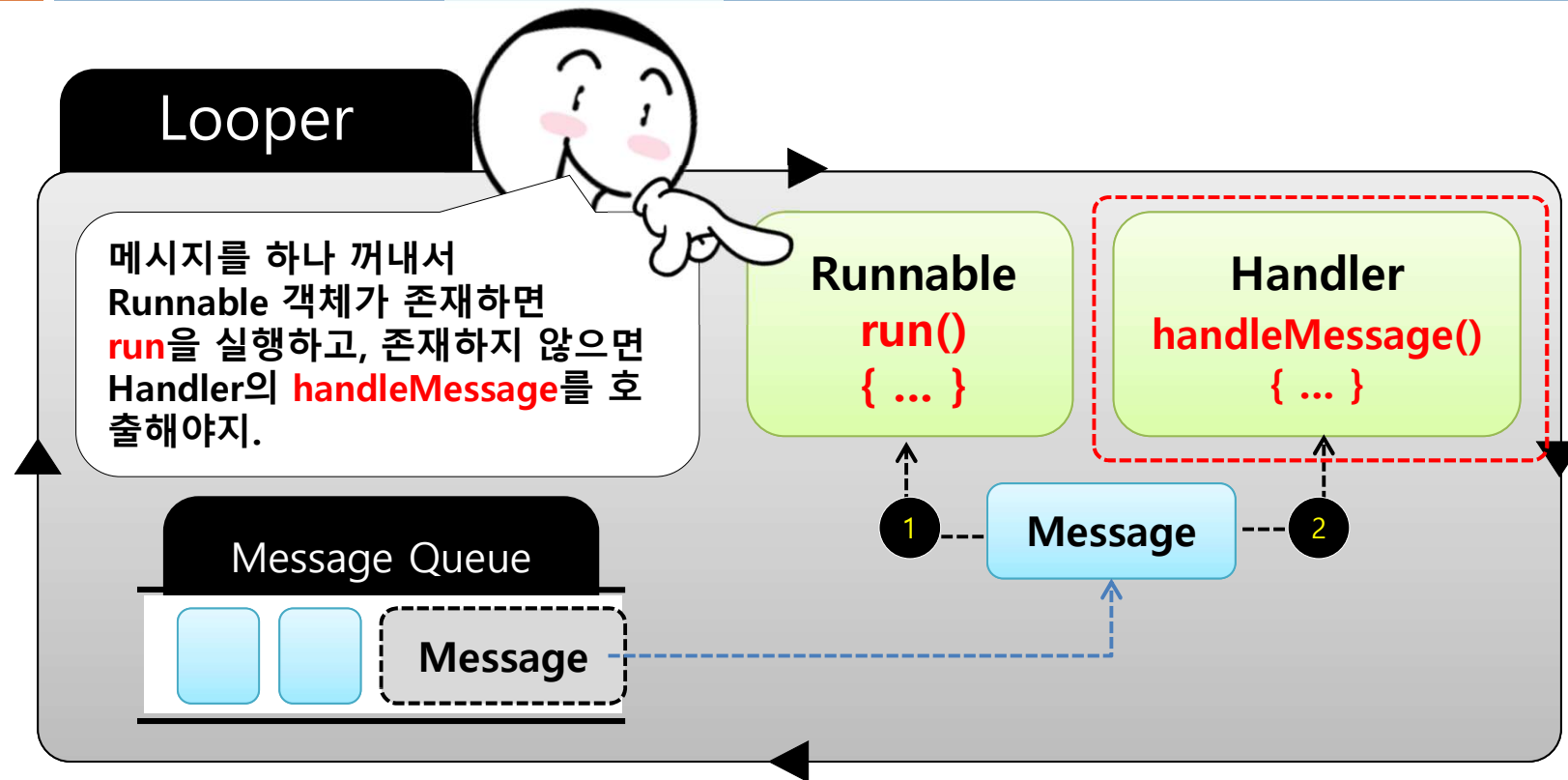
37

■ 메시지에 할 일을 담는 두 가지 방법



안드로이드 메인 스레드 구조 - 핸들러

38



우리는 Message 객체에 Runnable 객체를 추가하던지 혹은 Handler의 handleMessage 재정의
함수를 구현해주면 된다.

Handler의 handleMessage 재정의 함수에 대해서 살펴보자.

안드로이드 메인 스레드 구조 – 핸들러

39

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {
```

```
    int mCount = 0;
```

```
    TextView mCountTextView = null;
```

```
    static final private int MESSAGE_DRAW_CURRENT_COUNT = 1;
```

```
    // 메시지큐에 메시지를 추가하기 위한 핸들러를 생성한다.
```

```
    // =====
```

```
    Handler mHandler = new Handler() {
```

```
        // 메시지 큐는 핸들러에 존재하는 handleMessage 함수를 호출해준다.
```

```
        // -----
```

```
        public void handleMessage(Message msg) {
```

```
            switch (msg.what) {
```

```
                case MESSAGE_DRAW_CURRENT_COUNT: {
```

```
                    int currentCount = msg.arg1;
```

```
                    TextView countTextView = (TextView)msg.obj;
```

```
                    countTextView.setText("Count : " + currentCount);
```

```
                    break;
```

```
                }
```

```
            }
```

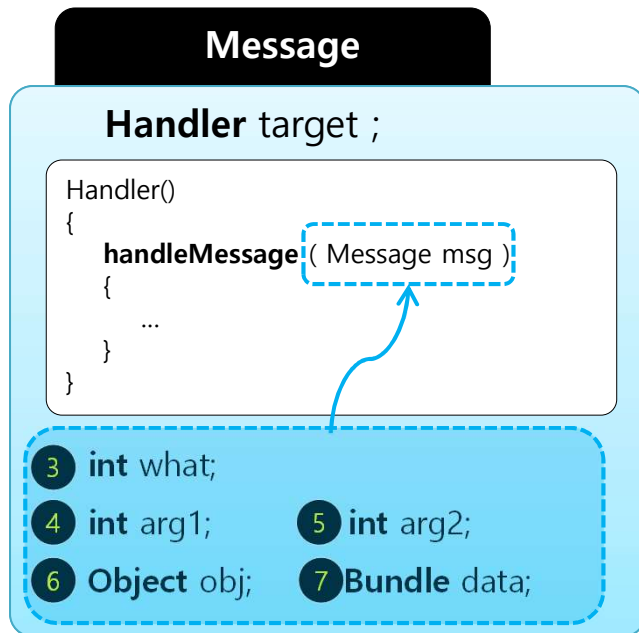
```
        }
```

```
        // -----
```

```
    };
```

```
    // =====
```

□ MainActivity.java



```

// 3. 10초 동안 1씩 카운트하는 스레드 생성 및 시작
// =====
Thread countThread = new Thread("Count Thread") {
    public void run() {
        for (int i = 0 ; i < 10 ; i++) {
            mCount++;

            // 1) 메시지 큐에 답을 메시지 하나를 생성한다.
            // -----
            Message message = Message.obtain(mHandler);
            // -----

            // 2) 핸들러의 handleMessage로 전달한 값들을 설정한다.
            // -----
            message.what = MESSAGE_DRAW_CURRENT_COUNT;
            message.arg1 = mCount;
            message.obj = mCountTextView;
            // -----

            // 3) 핸들러를 통해 메시지를 메시지 큐에 보낸다.
            // -----
            mHandler.sendMessage(message);
            // -----

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};

```


안드로이드 메인 스레드 구조 – 핸들러

41

■ Runnable? handleMessage? 무엇을 써야 하나?

참고로 Runnable 객체는 간단한 실행 코드를 추가하는 목적으로 사용되며,

Handler의 handleMessage 구현은 메시지의 what 멤버변수로 구분하여 여러 가지 목적의 실행 코드 한곳에서 모아 처리할 때 사용한다.

만일 handleMessage가 없다면 처리하려는 실행 코드마다 Runnable 객체를 생성하여 전달해야 할 것이다. 또한 handleMessage 함수의 경우 메시지 객체를 통해 다양한 자료형을 전달할 수 있기 때문에 더 유연하다.

안드로이드 메인 스레드 구조 - 핸들러

42

■ 스케줄링이 가능한 메시지

□ MainActivity.java

- ▣ 기존 코드의 mHandler.sendMessage(message);를 아래 코드로 대체

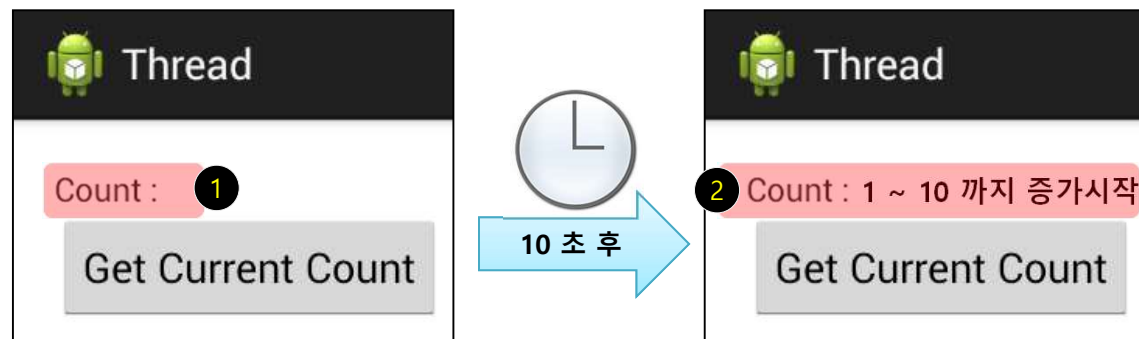
// 3) 핸들러를 통해 메시지를 메시지 큐에 보낸다.

// 메시지는 10초 지연이 설정 되었다.

// -----

mHandler.sendMessageDelayed(message, 10000);

// -----



안드로이드 메인 스레드 구조 - 핸들러

■ 메시지를 생성하는 함수 - 메시지 생성은 메시지의 `obtain` 함수를 통해 가능하다.

- **Message.obtain()**

빈 메시지 객체를 얻어온다.

- **Message.obtain(Message orig)**

인자로 전달한 orig 메시지 객체를 복사한 새로운 메시지 객체를 얻어온다.

- **Message.obtain(Handler h)**

인자로 전달한 핸들러 객체가 설정된 메시지 객체를 얻어온다.

- **Message.obtain(Handler h, Runnable callback)**

인자로 전달한 핸들러, Runnable 객체가 설정된 메시지 객체를 얻어온다.

- **Message.obtain(Handler h, int what)**

인자로 전달한 핸들러, what 객체가 설정된 메시지 객체를 얻어온다.

- **Message.obtain(Handler h, int what, Object obj)**

인자로 전달한 핸들러, what, obj 객체가 설정된 메시지 객체를 얻어온다.

- **Message.obtain(Handler h, int what, int arg1, int arg2)**

인자로 전달한 핸들러, what, arg1, arg2 객체가 설정된 메시지 객체를 얻어온다.

- 45 • **Message.obtain(Handler h, int what, int arg1, int arg2, Object obj)**

인자로 전달한 핸들러, what, arg1, arg2, obj 객체가 설정된 메시지 객체를 얻어온다.

안드로이드 메인 스레드 구조 – 핸들러

44

■ 메시지를 생성하는 것은 핸들러를 통해서도 가능하다.

하지만 내부적으로는 모두 Message 클래스를 사용하기 때문에 같은 방법이다.

- **mHandler.obtainMessage()**

빈 메시지 객체를 얻어온다.

- **mHandler.obtainMessage(int what)**

인자로 전달한 what 객체가 설정된 메시지 객체를 얻어온다.

- **mHandler.obtainMessage(int what, Object obj)**

인자로 전달한 what, obj 객체가 설정된 메시지 객체를 얻어온다.

- **mHandler.obtainMessage(int what, int arg1, int arg2)**

인자로 전달한 what, arg1, arg2 객체가 설정된 메시지 객체를 얻어온다.

- **mHandler.obtainMessage(int what, int arg1, int arg2, Object obj)**

인자로 전달한 what, arg1, arg2, obj 객체가 설정된 메시지 객체를 얻어온다.

안드로이드 메인 스레드 구조 – 핸들러

45

■ 메시지 큐에 메시지를 추가 및 삭제하는 핸들러 함수

핸들러의 **post**로 시작하는 함수: **Runnable** 객체를 이용하는 메시지는 별도로 메시지 객체를 생성하지 않아도 된다. 함수 내에서 메시지 객체를 자동으로 생성해 주기 때문이다.

- **mHandler.post(Runnable r)**

큐에 메시지를 추가한다. 여기서 추가되는 메시지에는 인자로 전달한 Runnable 객체가 설정된다.

- **mHandler.postAtFrontOfQueue(Runnable r)**

큐의 가장 앞에 메시지를 추가하여 기존 추가된 메시지들보다 우선으로 처리하게 한다.

- **mHandler.postAtTime(Runnable r, long uptimeMillis)**

시스템이 부팅된 시간을 기준으로 특정 시간에 실행되도록 한다. uptimeMillis 매개변수는 실행될 시간을 밀리세컨드 단위로 설정할 수 있다.

- **mHandler.postAtTime(Runnable r, Object token, long uptimeMillis)**

Runnable 객체를 사용하는 메시지는 **Object형의 값을 사용할 수 없다**. 하지만 여기서 Object형의 값을 메시지에 추가하는 이유는 **단지 추가된 Runnable 객체 메시지를 삭제할 때 구분자로 사용하기 위함이다**. 추가된 메시지를 큐에서 제거하는 함수에서 다시 확인해보겠다.

- **mHandler.postDelayed(Runnable r, long delayMillis)**

현재 시간을 기준으로 지연 시간 후에 실행되도록 한다. delayMillis 매개변수는 지연 실행될 시간을 밀리세컨드 단위로 설정할 수 있다.

핸들러의 **send**로 시작하는 함수:

handleMessage 함수가 재정의된 메시지를 큐에 추가하는 핸들러 함수들을 살펴보자.

- **mHandler.sendMessage(int what)**

이 함수는 내부적으로 메시지 객체를 자동으로 생성해 주기 때문에 별도로 메시지 객체를 생성하지 않고 사용이 가능하다. Empty가 붙은 함수들은 모두 동일한 의미를 가진다.

- **mHandler.sendMessageAtTime(int what, long uptimeMillis)**

인자로 전달한 what 객체가 메시지에 설정되어 큐에 추가된다. 추가된 메시지는 시스템이 부팅된 시간을 기준으로 인자로 전달한 uptimeMillis 시간에 실행되도록 한다.

- **mHandler.sendMessageDelayed(int what, long delayMillis)**

인자로 전달한 what 객체가 메시지에 설정되어 큐에 추가된다. 추가된 메시지는 인자로 전달한 delayMillis 지연 시간 이후에 실행되도록 한다.

- **mHandler.sendMessage(Message msg)**

인자로 전달한 메시지가 큐에 추가된다.

- **mHandler.sendMessageAtFrontOfQueue(Message msg)**

큐의 가장 앞에 메시지를 추가하여 기존 추가된 메시지들보다 우선으로 처리하게 한다.

- **mHandler.sendMessageAtTime(Message msg, long uptimeMillis)**

인자로 전달한 메시지가 큐에 추가된다. 추가된 메시지는 시스템이 부팅된 시간을 기준으로 인자로 전달한 uptimeMillis 시간에 실행되도록 한다.

- **mHandler.sendMessageDelayed(Message msg, long delayMillis)**

인자로 전달한 메시지가 큐에 추가된다. 추가된 메시지는 인자로 전달한 delayMillis 지연 시간 이후에 실행되도록 한다.

안드로이드 메인 스레드 구조 – 기본적으로 핸들러를 가지는 뷰와 액티비티

47

- 지금까지는 큐에 메시지를 추가하기 위해 핸들러를 생성하여 사용했다.
하지만 뷰와 액티비티에는 자체적으로 하나의 핸들러를 포함하고 있다.
그러므로 별도의 핸들러를 생성하지 않고도 뷰나 액티비티를 이용하여 큐에 메시지를 추가할 수 있다.

단, 핸들러의 `handleMessage` 함수를 사용한 메시지 처리는 핸들러 객체를 생성할 때 해당 함수를 재정의해야 하기 때문에 사용할 수 없고 오직 `Runnable` 객체만 가능하다.

- 뷰의 `post`와 `postDelayed` 함수

`public boolean post (Runnable action)`

`public boolean postDelayed (Runnable action, long delayMillis)`

- 액티비티의 `runOnUiThread` 함수

`public final void runOnUiThread (Runnable action)`

안드로이드 메인 스레드 구조 - 기본적으로 핸들러를 가지는 뷰와 액티비티

48

- MainActivity.java
 - ▣ countThread의 run() 메소드만 수정

```
public void run() {  
    for ( int i = 0 ; i < 10 ; i ++ ) {  
        // 1) 카운트 스레드가 1초간 쉬면서 1씩 증가 시킨다.  
        // -----  
        try { Thread.sleep( 1000 ); }  
        catch ( InterruptedException e ) { e.printStackTrace(); }  
  
        mCount ++;  
        // -----  
  
        // 2) 텍스트뷰에 증가되는 수치를 출력하는 Runnable 객체  
        // 메시지를 큐에 추가한다.  
        // -----  
        mCountTextView.post(new Runnable() {  
            public void run() {  
                mCountTextView.setText("Count : " + mCount);  
            }  
        });  
        // -----  
    }  
}
```

작업 스레드에서
특정 뷰를 갱신하는 경우에는
뷰 자체의 핸들러 사용을 권장

안드로이드 메인 스레드 구조 - 기본적으로 핸들러를 가지는 뷰와 액티비티

49

- View 클래스의 아래 함수를 사용하면 뷰의 핸들러를 얻어와서 직접 사용할 수도 있다.

```
public Handler getHandler ()
```

- 뷰의 핸들러 사용 시 주의사항
 - ▣ 뷰의 핸들러는 액티비티의 생명주기 onCreate, onStart, onResume 함수가 모두 호출된 이후에 참조할 수 있다.

안드로이드 메인 스레드 구조 - 기본적으로 핸들러를 가지는 뷰와 액티비티

50

□ MainActivity.java

□ countThread의 run() 메소드만 수정

```
public void run() {  
    for ( int i = 0 ; i < 10 ; i ++ ) {  
        // 1) 카운트 스레드가 1초간 쉬면서 1씩 증가 시킨다.  
        // -----  
        try { Thread.sleep( 1000 ); }  
        catch ( InterruptedException e ) { e.printStackTrace(); }  
  
        mCount ++;  
        // -----  
  
        // 2) 텍스트뷰에 증가되는 수치를 출력하는 Runnable 객체  
        // 메시지를 큐에 추가한다.  
        // -----  
        MainActivity.this.runOnUiThread(new Runnable() {  
            public void run() {  
                mCountTextView.setText("Count : " + mCount);  
            }  
        });  
        // -----  
    }  
}
```

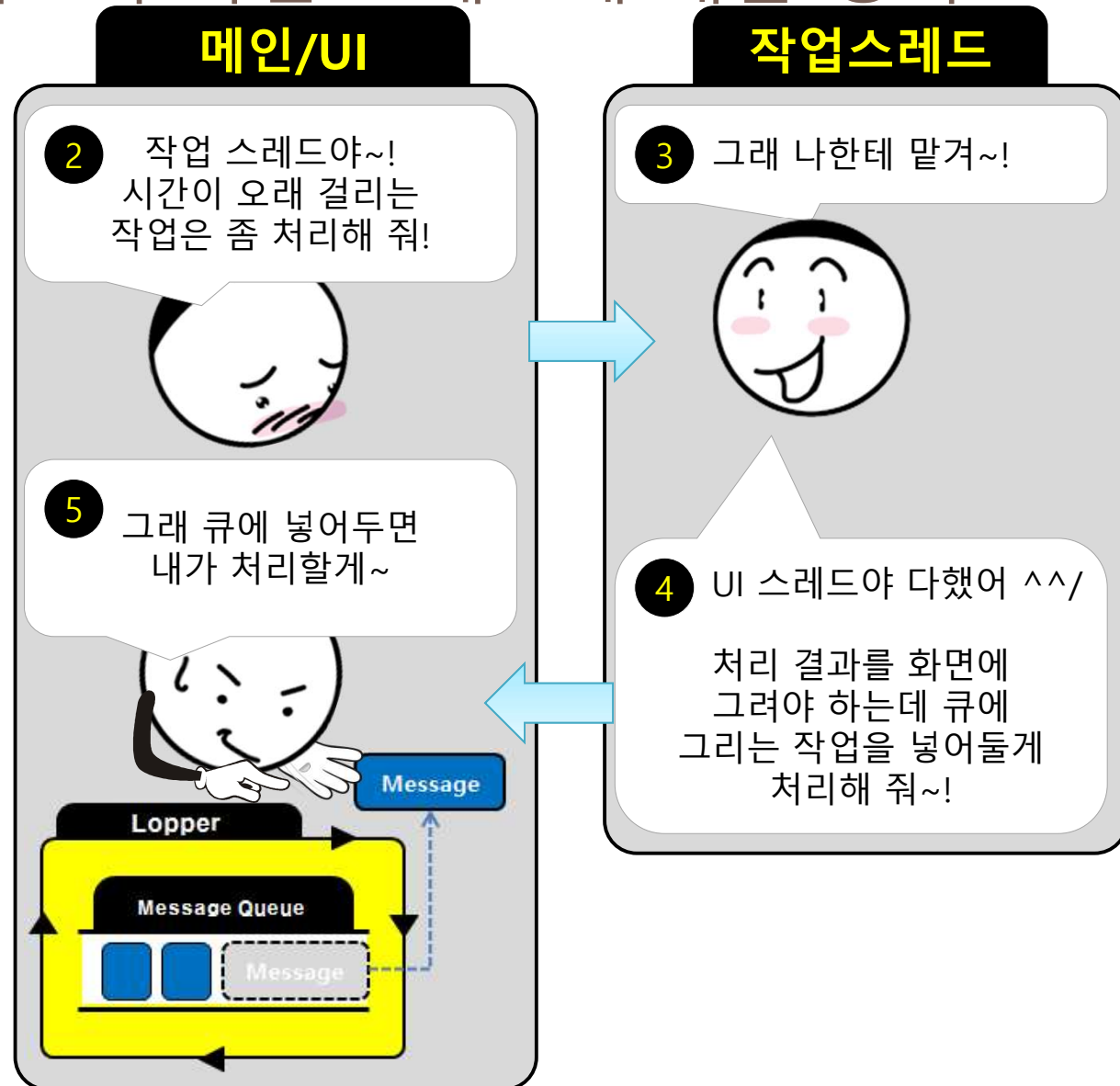
특정 뷰 하나가 아닌 액티비티에
사용되는 여러 가지 뷰들을
동시에 갱신하고자 할 때 사용하
는 것이 직관적이다.

안드로이드 메인 스레드 구조 - 메인 스레드와 작업 스레드에 대한 정리

1 메인 스레드!!!
내가 지켜 보고 있다.
스트릭트 모드, ANR
에 걸리지 마라~!



시스템



편리한 헬퍼 클래스들

52

■ 앱은 메인 스레드와 작업 스레드의 협력으로 원활히 돌아 갈 수 있다.

하지만 이렇게 스레드마다 처리해야 하는 일이 엄격히 구분되는 것은 개발에 있어 부담감이 크다.

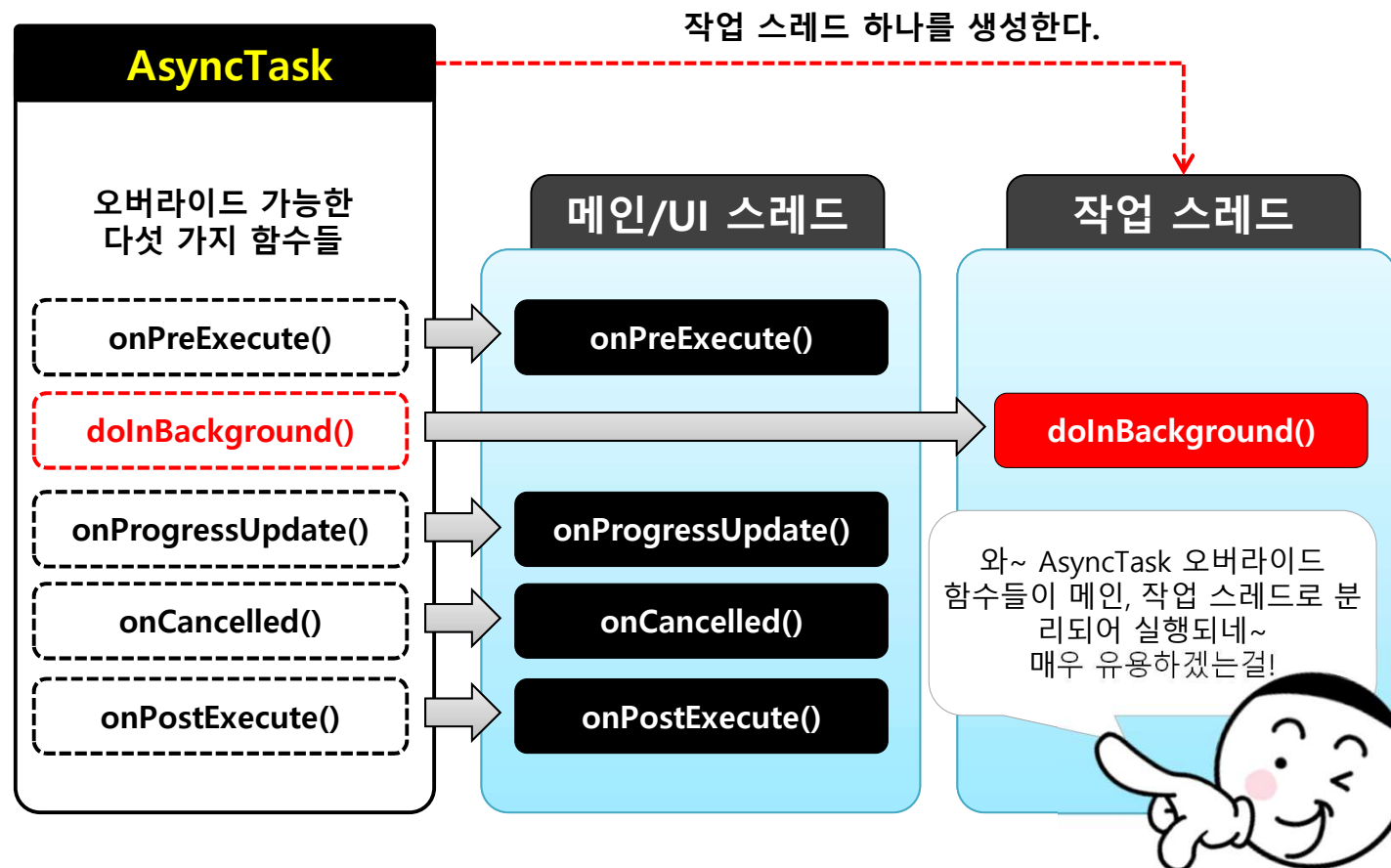
안드로이드에서는 이러한 복잡성을 줄이기 위해 다양한 도우미 클래스를 제공한다.

도우미 클래스는 개발 시간을 단축시키고 소스를 간결하게 정리해줄 뿐만 아니라, 가독성을 높여 유지 보수하기 편하다.

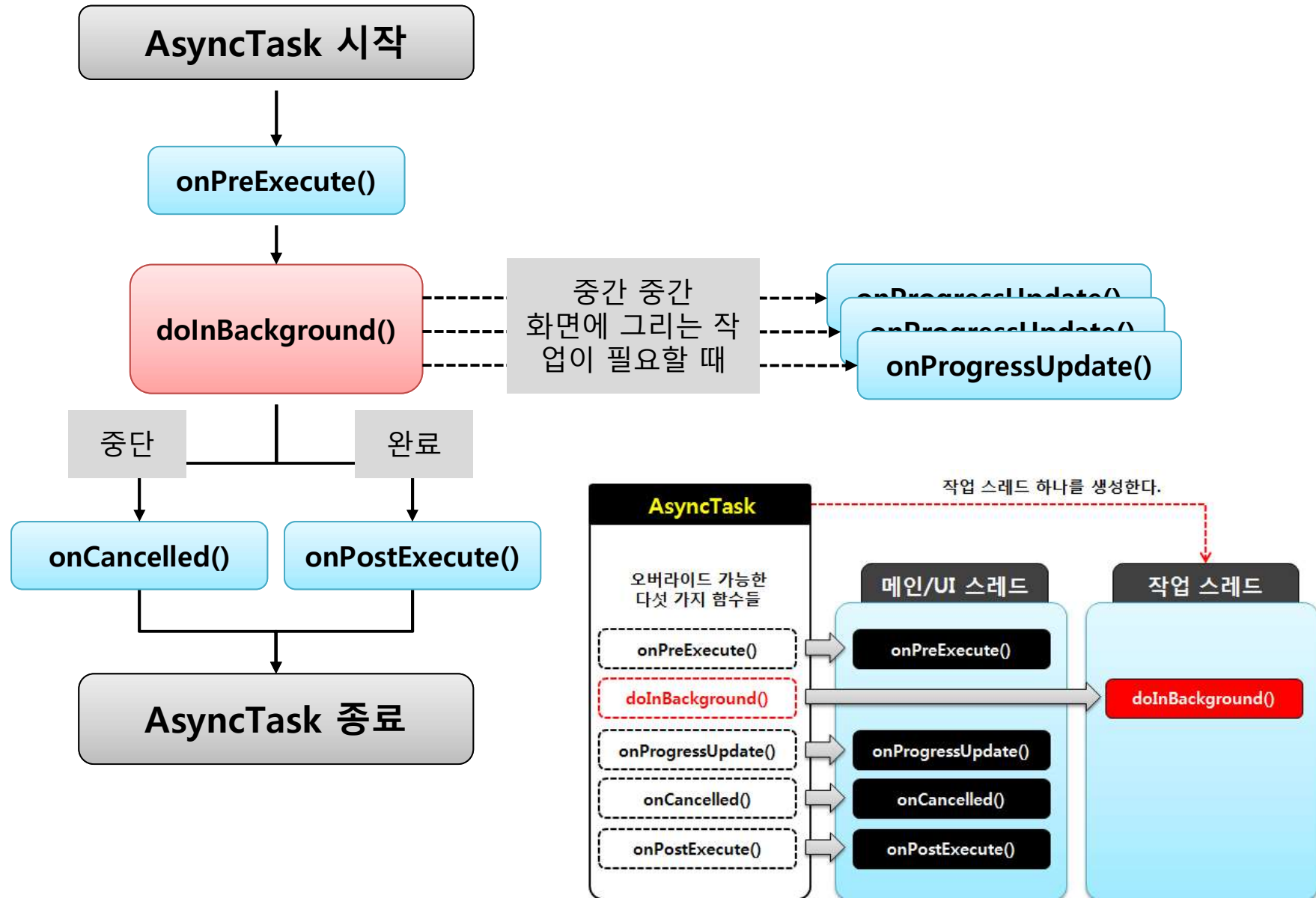
편리한 헬퍼 클래스들 - AsyncTask

53

- AsyncTask는 안드로이드에서 요구하는 메인 스레드와 작업 스레드의 분리 구조를 보다 쉽게 구현하도록 도와주는 추상 클래스다.



편리한 헬퍼 클래스들 - AsyncTask



편리한 헬퍼 클래스들 - AsyncTask

AsyncTask

```
public abstract class AsyncTask  
extends Object
```

```
java.lang.Object
```

```
↳ android.os.AsyncTask<Params, Progress, Result>
```

Protected methods	
abstract Result	<code>doInBackground(Params... params)</code> Override this method to perform a computation on a background thread.
void	<code>onCancelled()</code> Applications should preferably override <code>onCancelled(Object)</code> .
void	<code>onCancelled(Result result)</code> Runs on the UI thread after <code>cancel(boolean)</code> is invoked and <code>doInBackground(Object[])</code> has finished.
void	<code>onPostExecute(Result result)</code> Runs on the UI thread after <code>doInBackground(Params...)</code> .
void	<code>onPreExecute()</code> Runs on the UI thread before <code>doInBackground(Params...)</code> .
void	<code>onProgressUpdate(Progress... values)</code> Runs on the UI thread after <code>publishProgress(Progress...)</code> is invoked.
final void	<code>publishProgress(Progress... values)</code> This method can be invoked from <code>doInBackground(Params...)</code> to publish updates on the UI thread while the background computation is still running.

편리한 헬퍼 클래스들 - AsyncTask

AsyncTask

```
public abstract class AsyncTask  
extends Object
```

```
java.lang.Object
```

```
↳ android.os.AsyncTask<Params, Progress, Result>
```

Public methods	
final boolean	<code>cancel(boolean mayInterruptIfRunning)</code> Attempts to cancel execution of this task.
final <code>AsyncTask<Params, Progress, Result></code>	<code>execute(Params... params)</code> Executes the task with the specified parameters.
final <code>AsyncTask.Status</code>	<code>getStatus()</code> Returns the current status of this task.

AsyncTask.Status

```
public static final enum AsyncTask.Status  
extends Enum<AsyncTask.Status>
```

Enum values	
<code>AsyncTask.Status</code>	FINISHED Indicates that <code>onPostExecute(Result)</code> has finished.
<code>AsyncTask.Status</code>	PENDING Indicates that the task has not been executed yet.
<code>AsyncTask.Status</code>	RUNNING Indicates that the task is running.

편리한 헬퍼 클래스들 - AsyncTask

57

□ activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="20dp" >

    <TextView
        android:id="@+id/download_state_textview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download Cancel"
        android:onClick="onClick"/>

</LinearLayout>
```

편리한 헬퍼 클래스들 - AsyncTask

58

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
  
    TextView          mDownloadStateTextView = null;  
    FileDownloadTask  mFileDownloadTask      = null;  
  
    private class FileDownloadTask extends AsyncTask<String, Integer, Boolean> {  
        @Override  
        protected void onPreExecute() {  
            // 최초 화면에 내려받기 시도를 알리는 텍스트를 출력한다.  
            // =====  
            mDownloadStateTextView.setText("FileDownload...");  
            // =====  
  
            super.onPreExecute();  
        }  
    }  
}
```

편리한 헬퍼 클래스들 - AsyncTask

59

□ MainActivity.java

@Override

protected Boolean doInBackground(String... params) {

int totalCount = params.**length**;

// 전달 받은 파일 Url 개수만큼 반복하면서 파일을 내려받는다.

// =====

for (**int** i = 1 ; i <= totalCount ; i ++) {

// 1. 파일 내려받기 처리 상태를 표시하기 위해 호출

// -----

publishProgress(i, totalCount);

// -----

// 2. 파일을 내려받는 과정이라고 가정한다.

// -----

try { Thread.sleep(1000); }

catch (InterruptedException e) { **return false**; }

// -----

}

// =====

return true;

}

편리한 헬퍼 클래스들 - AsyncTask

60

□ MainActivity.java

@Override

protected void onProgressUpdate(Integer... values) {

int currentCount = values[0];

int totalCount = values[1];

 // 현재 파일 내려받기 상태를 표시한다. 예) Downloading : 3/10

 // =====

mDownloadStateTextView.setText("Downloading : " +
 currentCount + "/" + totalCount);

 // =====

super.onProgressUpdate(values);

}

편리한 헬퍼 클래스들 - AsyncTask

61

□ MainActivity.java

@Override

protected void onCancelled() {

// 화면에 내려받기 취소되었다는 텍스트를 출력한다.

// =====

mDownloadStateTextView.setText(**"Download cancel"**);

// =====

super.onCancelled();

}

@Override

protected void onPostExecute(Boolean result) {

// 화면에 내려받기 성공/실패 여부를 텍스트로 출력한다.

// =====

if (**true** == result)

mDownloadStateTextView.setText(**"Download finish"**);

else

mDownloadStateTextView.setText(**"Download Fail"**);

// =====

super.onPostExecute(result);

}

}

편리한 헬퍼 클래스들 - AsyncTask

62

□ MainActivity.java

@Override

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);
```

```
// 1. 화면에 그려질 레이아웃을 설정하고 액티비티에서 사용될  
// 텍스트뷰를 멤버변수로 참조 시킨다.
```

```
// =====
```

```
mDownloadStateTextView =  
    (TextView)findViewById(R.id.download_state_textview);
```

```
// =====
```

```
// 2. 파일 내려받기 AsyncTask를 생성하고 실행한다. AsyncTask의  
// execute는 AsyncTask를 실행하는 함수이다. 인자 값들을 통해  
// 내려받을 파일 URL을 전달하게 된다. 전달한 인자는 AsyncTask  
// doInBackground 함수가 수신한다.
```

```
// =====
```

```
mFileDownloadTask = new FileDownloadTask();  
mFileDownloadTask.execute("FileUrl_1", "FileUrl_2", "FileUrl_3",  
    "FileUrl_4", "FileUrl_5", "FileUrl_6",  
    "FileUrl_7", "FileUrl_8", "FileUrl_9",  
    "FileUrl_10");
```

```
// =====
```

```
}
```

편리한 헬퍼 클래스들 - AsyncTask

63

□ MainActivity.java

```
public void onClick(View v) {  
    // 만일 파일 내려받기 AsyncTask가 종료 상태가 아니라면  
    // 진행을 취소 시킨다.  
    // =====  
    if (mFileDownloadTask != null &&  
        mFileDownloadTask.getStatus() != AsyncTask.Status.FINISHED) {  
        mFileDownloadTask.cancel(true);  
    }  
    // =====  
}  
}
```

private class FileDownloadTask
 extends AsyncTask< String, Integer, Boolean >
 {

1 시작 2 중간 3 끝

```

    @Override
    protected Boolean doInBackground( String... fileUrls )
    {
        int totalCount = fileUrls.length;

        for ( int i = 1 ; i <= totalCount ; i ++ )
        {
            publishProgress( i, totalCount );
        }

        return true;
    }
  
```

작업 스레드 기준으로
 시작, 중간 끝에
 필요한 값들이다.

```

    @Override
    protected void onProgressUpdate( Integer... downloadInfos )
    {
        int currentCount = downloadInfos[0];
        int totalCount = downloadInfos[1];

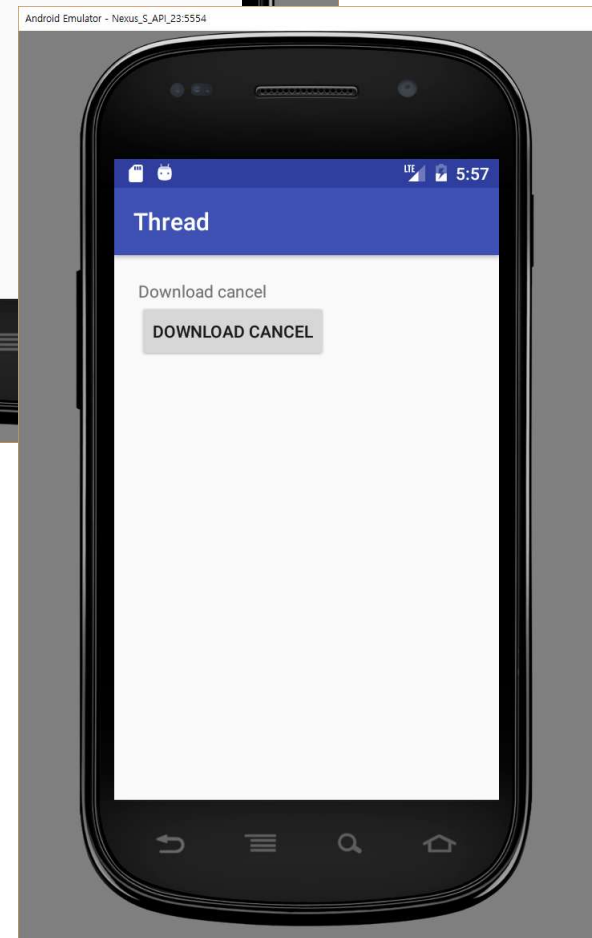
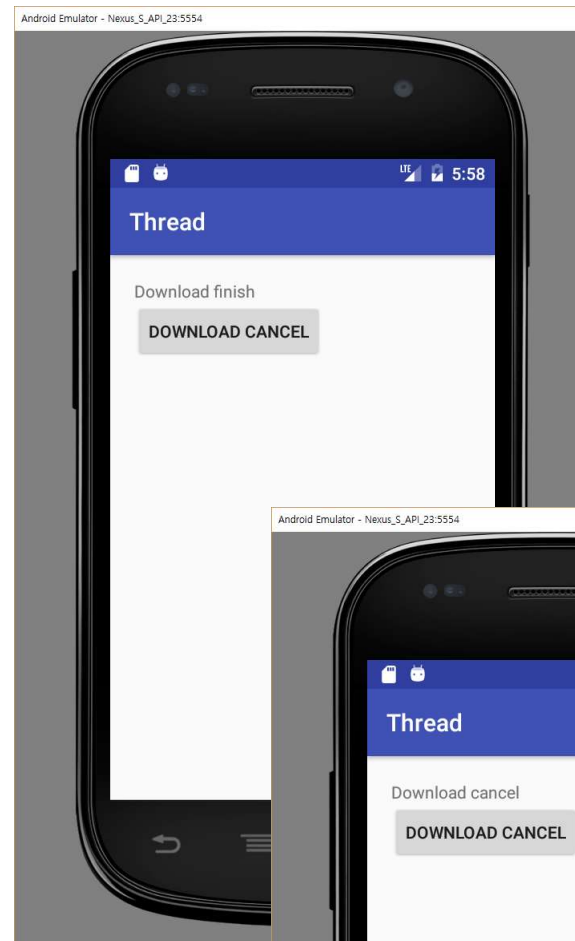
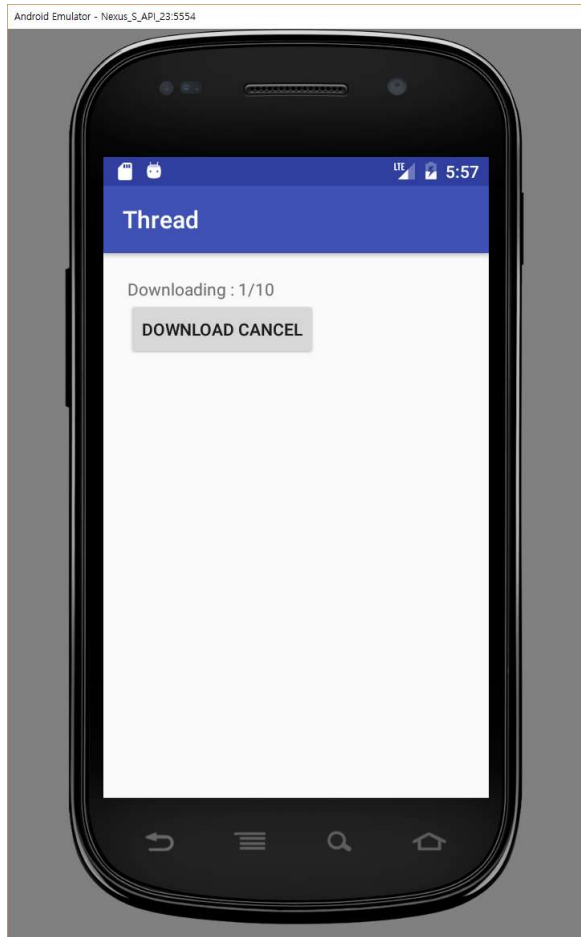
        mDownloadStateTextView.setText( "
            currentCount
        " );
    }
  
```

```

    @Override
    protected void onPostExecute( Boolean result )
    {
        mDownloadStateTextView.setText( "Download finish" );
    }
  
```

```

    @Override
    protected void onCreate( Bundle savedInstanceState )
    {
        mFileDownloadTask = new FileDownloadTask();
        mFileDownloadTask.execute(
            "FileUrl_1", "FileUrl_2", "FileUrl_3",
            "FileUrl_4", "FileUrl_5", "FileUrl_6",
            "FileUrl_7", "FileUrl_8", "FileUrl_9",
            "FileUrl_10" );
    }
  
```

편리한 헬퍼 클래스들 - AsyncTask

66

- 둘 이상의 AsyncTask 객체 실행
 - ▣ API 10 진저브레드까지는 동시에 실행
 - ▣ API 11 허니콤 이후부터는 순서대로 실행

```
public class MainActivity extends AppCompatActivity {  
    ...  
    FileDownloadTask mFileDownloadTask = null;  
    FileDownloadTask mImageFileDownloadTask = null;  
    ...  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        mFileDownloadTask = new FileDownloadTask();  
        mFileDownloadTask.execute("FileUrl_1", "FileUrl_2", "FileUrl_3",  
            "FileUrl_4", "FileUrl_5", "FileUrl_6",  
            "FileUrl_7", "FileUrl_8", "FileUrl_9",  
            "FileUrl_10");  
        mImageFileDownloadTask = new FileDownloadTask();  
        mImageFileDownloadTask.execute("ImageFileUrl_1", "ImageFileUrl_2", "ImageFileUrl_3",  
            "ImageFileUrl_4", "ImageFileUrl_5", "ImageFileUrl_6",  
            "ImageFileUrl_7", "ImageFileUrl_8", "ImageFileUrl_9",  
            "ImageFileUrl_10");  
    }  
    ...  
}
```

편리한 헬퍼 클래스들 - AsyncTask

67

- 둘 이상의 AsyncTask 객체 실행
 - ▣ API 11부터 동시에 실행하려면 다음 함수를 이용
`executeOnExecutor(Executor exec, Params... params)`
 - 첫 번째 매개변수
 - AsyncTask.THREAD_POOL_EXECUTOR: 동시실행
 - AsyncTask.SERIAL_EXECUTOR: 순차실행

```
...
mFileDownloadTask = new FileDownloadTask();
if (Build.VERSION.SDK_INT < 11) {
    mFileDownloadTask.execute("FileUrl_1", "FileUrl_2", "FileUrl_3",
        "FileUrl_4", "FileUrl_5", "FileUrl_6",
        "FileUrl_7", "FileUrl_8", "FileUrl_9",
        "FileUrl_10");
} else {
    mFileDownloadTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
        "FileUrl_1", "FileUrl_2", "FileUrl_3",
        "FileUrl_4", "FileUrl_5", "FileUrl_6",
        "FileUrl_7", "FileUrl_8", "FileUrl_9",
        "FileUrl_10");
}
...
```

편리한 헬퍼 클래스들 - AsyncTask

68

□ 동작 중인 AsyncTask 중단하기

```
public class MainActivity extends AppCompatActivity {
    ...
    private class FileDownloadTask extends AsyncTask<String, Integer, Boolean> {
        ...
        @Override
        protected Boolean doInBackground(String... params) {
            ...
            for (int i = 1; i <= totalCount; i++) {
                ...
                publishProgress(i, totalCount);
                ...
                try { Thread.sleep(1000); }
                catch (InterruptedException e) { return false; }
            }
            return true;
        }
        ...
    }
    ...
    public void onClick(View v) {
        ...
        mFileDownloadTask.cancel(true);
        ...
    }
}
```

편리한 헬퍼 클래스들 - AsyncTask

69

- 동작 중인 AsyncTask 중단하기
 - ▣ AsyncTask의 cancel 함수가 호출되면, doInBackground 함수에서 publishProgress 함수를 호출해도 publishProgress 함수는 onProgressUpdate 함수를 호출하지 않는다.
 - 즉, 작업 스레드는 계속 돌고 있다.
 - ▣ 앞 페이지의 작업 스레드는 어떻게 종료되었을까?
 - AsyncTask의 cancel 함수가 호출되면, 스레드 관련 함수를 사용할 때 InterruptedException 발생에 의해 catch 블록에서 종료되었다.
 - ▣ 그렇다면, 스레드 관련 함수를 사용하지 않으면 중간에 종료되지 않는단 말인가?

sleep(), join(), wait() 메서드가 실행 중(이면 즉시) 혹은 실행 될 때
즉, 스레드가 WAITING or TIMED_WAITING 상태일 때

편리한 헬퍼 클래스들 - AsyncTask

70

□ 동작 중인 AsyncTask 중단하기

```
public class MainActivity extends AppCompatActivity {  
    ...  
    private class FileDownloadTask extends AsyncTask<String, Integer, Boolean> {  
        ...  
        @Override  
        protected Boolean doInBackground(String... params) {  
            ...  
            for (int i = 1; i <= totalCount; i++) {  
                ...  
                publishProgress(i, totalCount);  
                if (isCancelled() == true) return false;  
                try { Thread.sleep(1000); }  
                catch (InterruptedException e) { return false; }  
            }  
            return true;  
        }  
    }  
    ...  
}  
...  
public void onClick(View v) {  
    ...  
    mFileDownloadTask.cancel(true);  
    ...  
}
```

cancel 함수의 true 인자는 InterruptedException
을 발생시키고, false 인자는 발생시키지 않는다.

편리한 헬퍼 클래스들 - CountDownTimer

71

- **CountDownTimer**는 영어 뜻 그대로 카운트 다운하는 클래스다.
- activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="20dp" >
    <TextView
        android:id="@+id/countdown_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="20dp" />
    <Button
        android:id="@+id/start_countdown_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Countdown"
        android:onClick="onClick"/>
    <Button
        android:id="@+id/reset_countdown_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Stop Countdown"
        android:onClick="onClick"/>
</LinearLayout>
```



편리한 헬퍼 클래스들 - CountDownTimer

72

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
    TextView mCountTextView = null;  
    CountDownTimer mCountDownTimer = null;
```


편리한 헬퍼 클래스들 - CountDownTimer

73

□ MainActivity.java

```
private class TestCountDownTimer extends CountDownTimer {
    TestCountDownTimer(long millisInFuture, long countDownInterval) {
        super(millisInFuture, countDownInterval);
    }
    @Override
    public void onTick(long millisUntilFinished) {
        // 매번 틱마다 남은 초를 출력한다.
        // =====
        mCountTextView.setText(millisUntilFinished/1000 + " 초");
        // =====
    }
    @Override
    public void onFinish() {
        // 카운트다운이 완료된 경우 카운트다운의 최종 초를 출력한다.
        // =====
        mCountTextView.setText("0 초");
        // =====
    }
}
```

편리한 헬퍼 클래스들 - CountDownTimer

74

□ MainActivity.java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // 1. 레이아웃을 액티비티에 반영 및 텍스트뷰 객체를 얻어온다.
    // =====
    mCountTextView = (TextView)findViewById(R.id.countdown_text);
    // =====
    // 2. 총 60초 동안 1초씩 카운트다운 객체를 생성한다.
    // =====
    mCountDownTimer = new TestCountDownTimer(60000, 1000);
    // =====
    // 3. 카운트다운 초기 값을 출력한다.
    // =====
    mCountTextView.setText("60 초");
    // =====
}
```

편리한 헬퍼 클래스들 - CountDownTimer

75

□ MainActivity.java

```
public void onClick(View v) {  
    // 버튼 클릭 이벤트 처리  
    // =====  
    switch(v.getId()) {  
        case R.id.start_countdown_btn:  
            // 총 60초 카운트다운을 시작한다.  
            // -----  
            mCountDownTimer.start();  
            // -----  
            break;  
        case R.id.reset_countdown_btn:  
            // 카운트다운을 중단하고 초를 리셋한다.  
            // -----  
            mCountDownTimer.cancel();  
            mCountTextView.setText("60 초");  
            // -----  
            break;  
    }  
    // =====  
}  
}
```

편리한 헬퍼 클래스들 - CountDownTimer

76

- CountDownTimer는 작업 스레드에서 동작하는 것이 아니다.
 - ▣ 메인 스레드의 루퍼 스케줄링을 이용한다.

// handles counting down

```
private Handler mHandler = new Handler() {
```

```
    @Override
```

```
    public void handleMessage(Message msg) {  
        synchronized (CountDownTimer.this) {
```

```
            if (mCancelled) {  
                return;
```

```
            }
```

```
            final long millisLeft = mStopTimeInFuture - SystemClock.elapsedRealtime();
```

```
            if (millisLeft <= 0) {  
                onFinish();
```

```
            } else if (millisLeft < mCountdownInterval) {
```

```
                // no tick, just delay until done
```

```
                sendMessageDelayed(obtainMessage(MSG), millisLeft);
```

```
            } else {
```

```
                long lastTickStart = SystemClock.elapsedRealtime();
```

```
                onTick(millisLeft);
```

```
                // take into account user's onTick taking time to execute
```

```
                long delay = lastTickStart + mCountdownInterval - SystemClock.elapsedRealtime();
```

```
                // special case: user's onTick took more than interval to
```

```
                // complete, skip to next interval
```

```
                while (delay < 0) delay += mCountdownInterval;
```

```
                sendMessageDelayed(obtainMessage(MSG), delay);
```

```
            }
```

```
        }
```

```
    }
```

```
};
```



CountDownTimer 내부의 핸들러 소스 코드

편리한 헬퍼 클래스들 – HandlerThread

77

■ HandlerThread는 메인 스레드의 큐, 루퍼의 구조를 그대로 가진 클래스다.

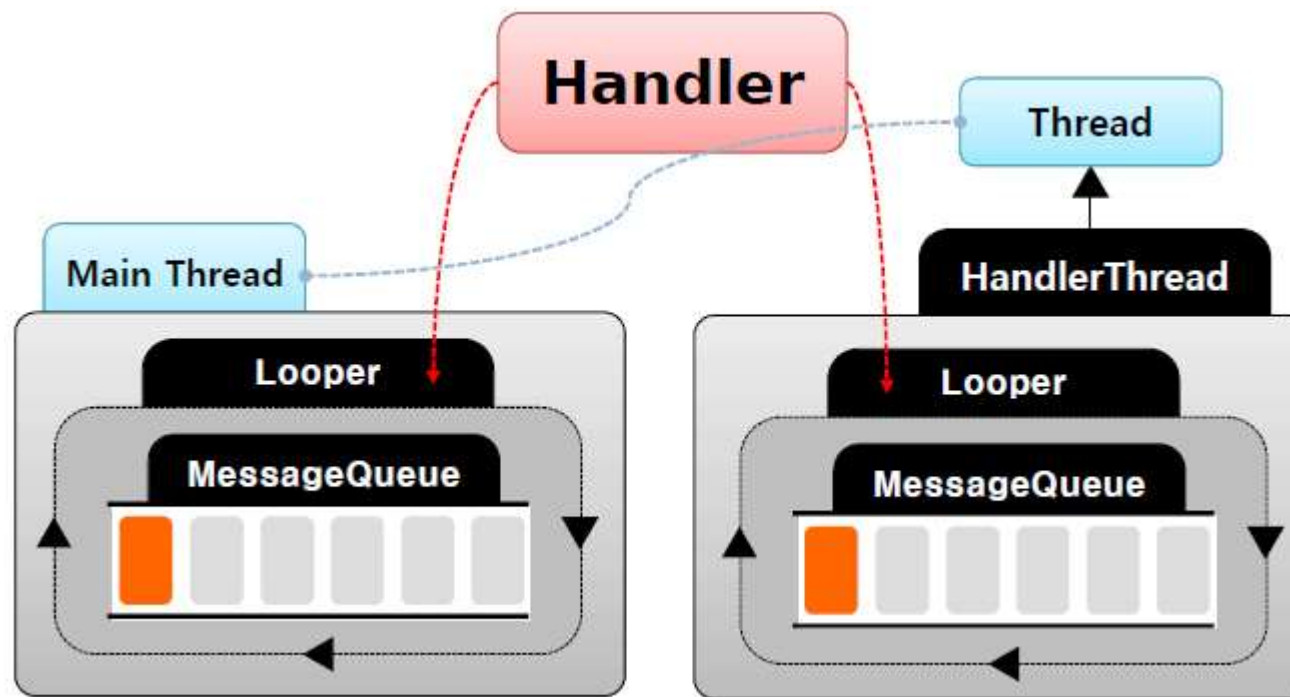
메인 스레드에서 사용되는 루퍼, 메시지 큐 구조는 처리해야 할 작업을 메시지 단위로 추가할 수 있고, 추가된 순으로 순으로 처리된다. 여기서 처리되는 순서를 변경하는 등 스케줄링도 가능하다.

이러한 루퍼, 메시지 큐 구조는 앱을 구현하면서 활용도가 높고 유용하다. 하지만 아쉬운 점은 루퍼가 동작하는 곳이 메인 스레드라는 것이다. 안드로이드에서 메인 스레드는 대부분 화면에 그림을 그리는 용도로 사용하고 있기 때문에 긴 작업을 수행할 수 없다.

이를 위해 안드로이드에서는 작업 스레드에서 동작하는 루퍼와 메시지 큐를 지원하기 위해 HandlerThread라는 클래스를 제공한다.

편리한 헬퍼 클래스들 – HandlerThread

78



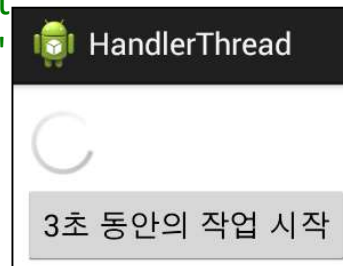
사용법도 매우 간단하다.

편리한 헬퍼 클래스들 - HandlerThread

79

- 3초 동안 긴 작업을 수행하는 예제를 만들어보자.
 - 메인 스레드 루퍼와 메시지 큐를 사용
- activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="10dp">
    <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:id="@+id/working_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="3초 동안의 작업 시작"
        android:onClick="onClick" />
</LinearLayout>
```



● 레이아웃 구조

- LinearLayout
 - ProgressBar
 - working_btn (Button) - "3초 동안의 작업 시작"

편리한 헬퍼 클래스들 – HandlerThread

80

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {
```

```
    Handler mHandler = null;
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);
```

```
        // 메인 스레드의 루퍼를 참조하는 핸들러를 생성한다.
```

```
        // =====
```

```
        mHandler = new Handler();
```

```
        // =====
```

```
    }
```


편리한 헬퍼 클래스들 – HandlerThread

81

□ MainActivity.java

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

```
// 1. 3초 동안 긴 작업을 수행하는 Runnable 객체를 생성한다.
```

```
// =====  
Runnable job = new Runnable() {  
    @Override  
    public void run() {  
        SystemClock.sleep(3000);  
        Toast.makeText(MainActivity.this,  
            "3초 작업 끝",  
            Toast.LENGTH_SHORT).show();  
    }  
};
```

```
};
```

```
// =====
```

```
// 2. 핸들러를 이용하여 메인 스레드의 메시지 큐에 3초간 작업하는 Runnable  
// 객체를 추가한다.
```

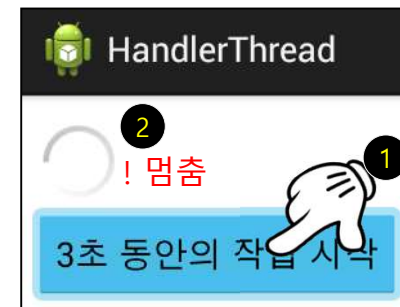
```
// =====
```

```
mHandler.post(job);
```

```
// =====
```

```
}
```

```
}
```



편리한 헬퍼 클래스들 – HandlerThread

82

- 3초 동안 긴 작업을 수행하는 예제를 만들어보자.
 - HandlerThread 루퍼와 메시지 큐를 사용

□ MainActivity.java

```
public class MainActivity extends AppCompatActivity {
```

```
    Handler mHandler = null;  
    HandlerThread mHandlerThread = null;
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);
```

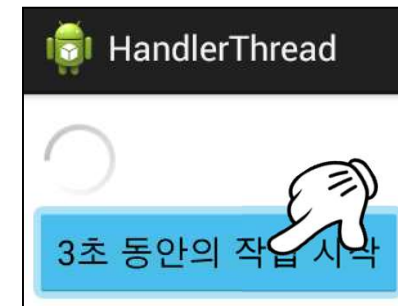
```
        // 1. 루퍼와 메시지큐를 가지고 있는 HandlerThread 객체를 생성하고  
        // HandlerThread가 가진 스레드의 동작을 시작한다.
```

```
        // =====  
        HandlerThread mHandlerThread = new HandlerThread("MyThread");  
        mHandlerThread.start();  
        // =====
```

```
        // 2. HandlerThread의 루퍼를 참조하는 핸들러를 생성한다.
```

```
        // =====  
        mHandler = new Handler(mHandlerThread.getLooper());  
        // =====
```

```
    }
```



```
Handler(Looper looper)  
Use the provided Looper instead of the default one.
```