자바응용SW(앱)개발자양성

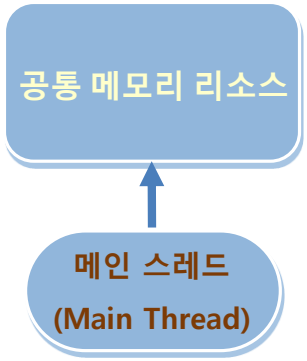
스레드

백제직업전문학교

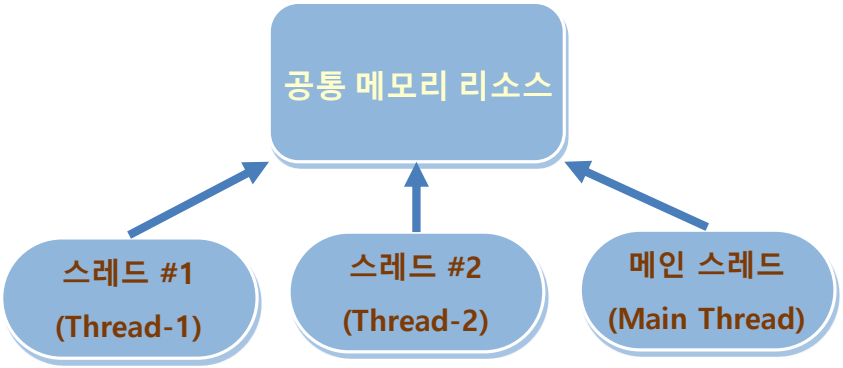
강사 : 김영준

1. 핸들러 사용하기

멀티 스레드



(1) 프로젝트 생성 시



(2) 별도의 스레드 생성 시

[멀티스레드 시스템에서 시스템에서
공통 메모리 리소스 접근]

- **메인 액티비티**
  - 애플리케이션이 실행될 때 하나의 프로세스에서 처리
  - 이벤트를 처리하거나 필요한 메소드를 정의하여 기능을 구현하는 경우에도 동일한 프로세스 내에서 실행
- **문제점**
  - 대기 시간이 길어지는 네트워크 요청 등의 기능을 수행할 때는 화면에 보이는 UI도 멈춤 상태로 있게 됨
- **해결 방안**
  - 하나의 프로세스 안에서 여러 개의 작업이 동시 수행되는 멀티 스레드 방식을 사용
- **멀티 스레드**
  - 같은 프로세스 안에 들어 있으면서 메모리 리소스를 공유하게 되므로 효율적인 처리가 가능
  - 동시에 리소스를 접근할 경우 데드락(DeadLock) 발생

메시지 전송하여 실행하기

- **메인 스레드**
  - 애플리케이션 객체인 액티비티, 브로드캐스트 수신자 등과 새로 만들어지는 윈도우를 관리하기 위한 메시지 큐(Message Queue)를 실행함

- **메시지 큐 : Message Queue**
  - 순차적으로 코드를 수행함

- **핸들러 : Handler**
  - 메시지 큐를 이용해 메인 스레드에서 처리할 메시지를 전달하는 역할을 담당함
  - 특정 메시지가 미래의 어떤 시점에 실행되도록 스케줄링 할 수 있음

메시지 전송하여 실행하기

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

왜 이러한 제약사항이 존재할까?

여러 스레드에서 뷰를 변경하여 화면을 갱신한다면 동기화 문제가 발생될 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 허용하여 그리는 순서를 보장하며,
이를 단일 스레드 GUIGraphical User Interface 모델이라 한다.

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

메시지 전송하여 실행하기

■ 그렇다면 작업 스레드에서 화면을 그리면 어떻게 될까?

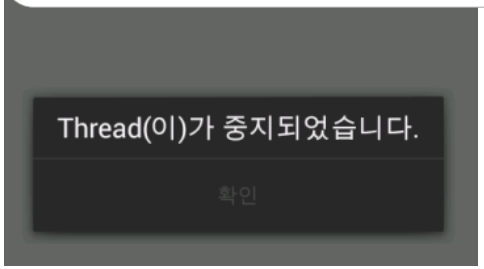
src/ThreadActivity.java

```
public class ThreadActivity extends Activity
{
    ...

    @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 ++;

                    // 현재까지 카운트된 수를 텍스트뷰에 출력한다.
                    mCountTextView.setText( "Count : " + mCount );
                    try { Thread.sleep( 1000 ); }
                    catch ( InterruptedException e ) { e.printStackTrace(); }
                }
            }
        };

        countThread.start();
    }
```

메시지 전송하여 실행하기

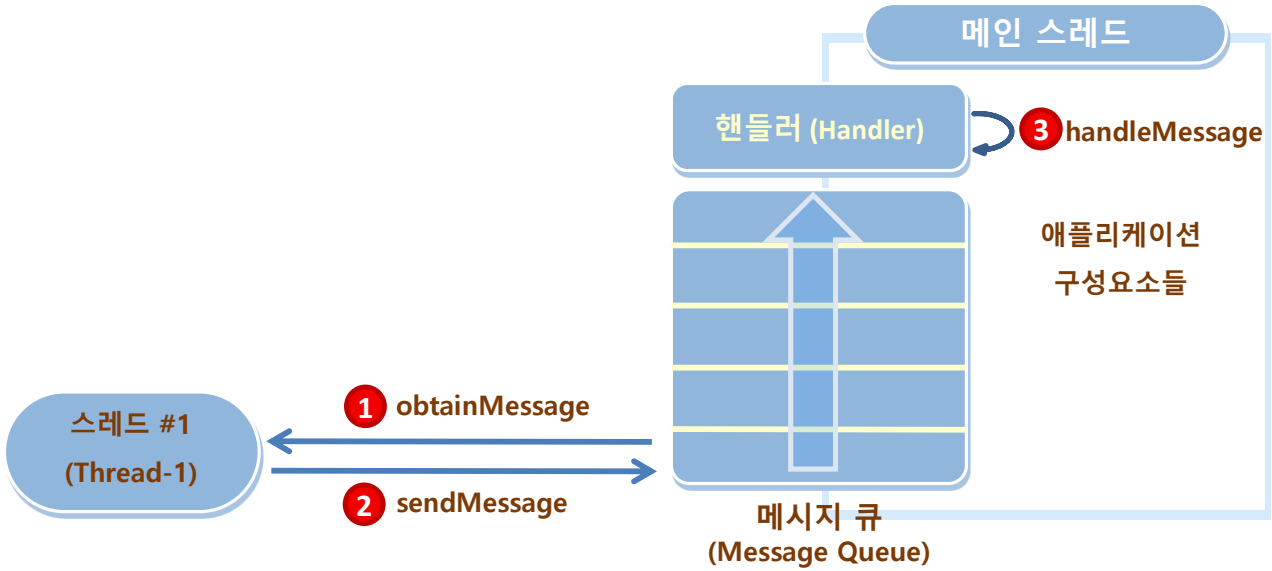
● 로그 출력 결과

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



따라서 작업 스레드에서 그리는 작업이 필요하다면 메인 스레드로 이관해야 한다.

메시지 전송하여 실행하는 구조



[핸들러를 사용할 때 필요한 세 가지 단계]

- **obtainMessage()**
  - 호출의 결과로 메시지 객체를 리턴받게함

- **sendMessage()**
  - 메세지큐에 넣음

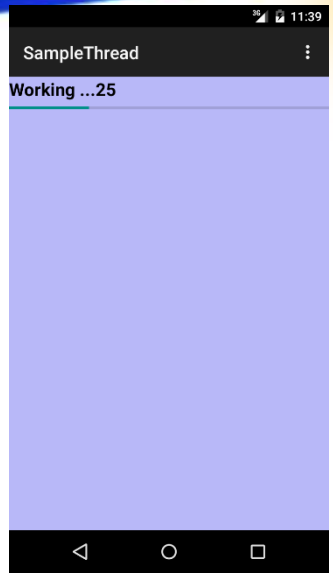
- **handleMessage()**
  - 메소드에 정의된 기능이 수행됨
  - 코드가 수행되는 위치는 새로 만든 스레드가 아닌 메인 스레드가 됨

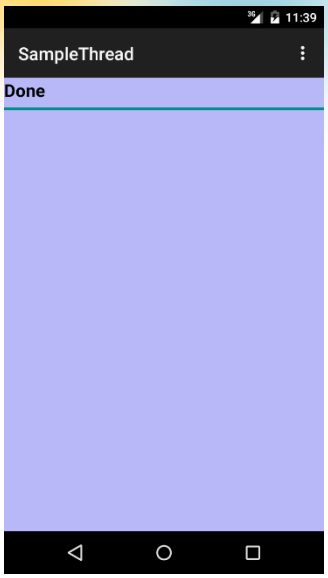
메시지 전송하여 실행하는 구조 (계속)

```
public class MainActivity extends Activity {

 ...


 ProgressHandler handler;

 public void onCreate(Bundle savedInstanceState) {
  ...


  handler = new ProgressHandler();
 }
```

1 새로 정의한 핸들러의 변수 선언

2 액티비티가 만들어질 때 핸들러 객체 생성





- Handler 클래스를 상속한 ProgressHandler 클래스 새로 정의
- Handler 클래스에 들어있는 handleMessage() 메소드를 다시 정의하여 메시지가 메인 스레드에서 수행될 때 필요한 기능을 정의하기 위함

메시지 전송하여 실행하는 구조 (계속)

```
public void onStart() {
 ...

 Thread thread1 = new Thread(new Runnable() {

  public void run() {
    ...


    Message msg = handler.obtainMessage();
    handler.sendMessage(msg);
    ...

  }
 });
 ...

 thread1.start();
}
```

3 액티비티가 시작될 때 스레드를 만들어 시작

4 작업 상태나 결과를 핸들러의 sendMessage() 메소드로 전송

- new 연산자를 이용하여 객체로 만든 후 start() 메소드를 이용해 실행함
- 스레드의 run() 메소드 안에 있는 코드가 스레드 시작 시에 수행됨

메시지 전송하여 실행하는 구조 (계속)

```
public class ProgressHandler extends Handler {

  public void handleMessage(Message msg) {
    ...
  }
 }

}
```

5 Handler 클래스를 상속하여 새로운 핸들러 클래스 정의

6 메소드 안에서 전달된 정보를 이용해 UI 업데이트

- run() 메소드 안에서는 별도의 스레드에서 수행한 작업의 결과가 나왔을 때 핸들러 객체의 obta nMessage()로 메시지 객체 하나를 참조한 후 sendMessage() 메소드를 이용해 메시지 큐에 넣음

메시지 전송하여 실행하는 레이아웃

- 프로그레스바는 원형이 아닌 막대 모양으로 보여주기 위해 style 속성을 "?android:attr/progressBarStyleHorizontal"로 설정

진행 상태를 표시하기 위한 프로그레스바 정의

```
<ProgressBar
    android:id="@+id/progress"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:max="100"
    />
```

Runnable 객체 실행하기

- 핸들러 클래스

 - 메시지 전송 방법 이외에 Runnable 객체를 실행시킬 수 있는 방법을 제공함

- Runnable 객체

 - 새로 만든 Runnable 객체를 핸들러의 post() 메소드를 이용해 전달해 주기만 하면 이 객체에 정의된 run() 메소드 내의 코드들은 메인 스레드에서 실행됨

Runnable 객체 실행하기 – 메인 액티비티 코드

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

  Handler handler;

  ProgressRunnable runnable;

  public void onCreate(Bundle savedInstanceState) {
    ...

    handler = new Handler();
    runnable = new ProgressRunnable();
  }

  public void onStart() {
    ...
```

1 핸들러의 변수 선언

2 새로 정의한 Runnable 객체의 변수 선언

3 새로 정의한 Runnable 객체의 변수 선언

- 일반적으로 사용하는 Handler 클래스를 이용하여 객체를 생성함

Runnable 객체 실행하기 – 메인 액티비티 코드 (계속)

```
public void onStart() {
  ...

  Thread thread1 = new Thread(new Runnable() {
    public void run() {
      ...

      handler.post(runnable);

      ...
    }
  });
  ...
  thread1.start();
}

...
```

4 핸들러의 변수 선언

5 새로 정의한 Runnable 객체의 변수 선언

- ProgressRunnable 클래스는 Runnable 클래스를 상속하여 정의하였으며 run() 메소드 안에는 이전 예제에서 사용했던 handleMessage() 메소드 안의 코드를 그대로 넣어 동일한 기능을 수행함

```
public class ProgressRunnable implements Runnable {
  public void run() {
    ...
  }
}
```

6 새로 정의한 Runnable 객체의 변수 선언

Runnable 객체 실행하기 – 메인 액티비티 코드 (계속)

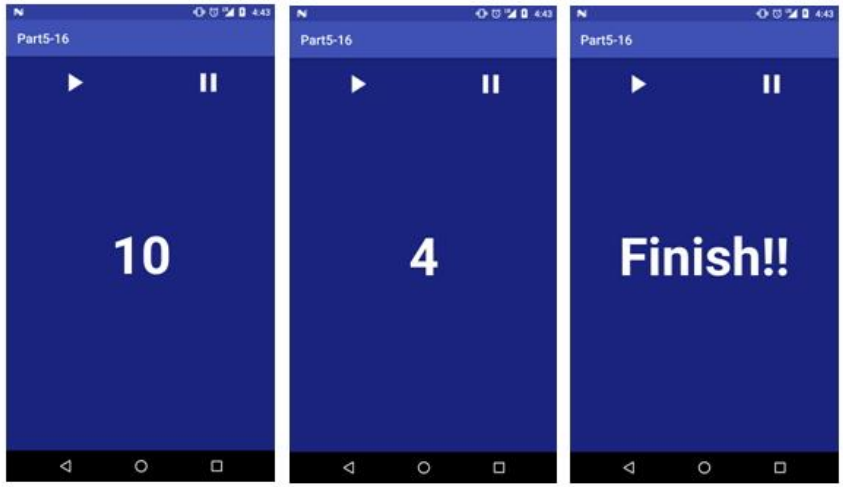
- 프로그레스바의 값이 최대값에 도달하게 되면 텍스트뷰에 표시되는 메시지를 "Done" 이라고 표시하여 더 이상 스레드가 실행되지 않는 상태라는 것을 알 수 있도록 함

```
public class ProgressRunnable implements Runnable {
  public void run() {

    bar.incrementProgressBy(5);

    if (bar.getProgress() == bar.getMax()) {
      textView01.setText("Runnable Done");
    } else {
      textView01.setText("Runnable Working ... "
      + bar.getProgress());
    }
  }
}
```

Thread – Handler 작성하기

Thread – Handler 구조를 테스트

테스트는 모래시계를 가정
간단한 테스트를 위해 시간은 숫자로만 표현
지정된 시간이 지나면 Finish 문자열을 출력하는 단순한 구조

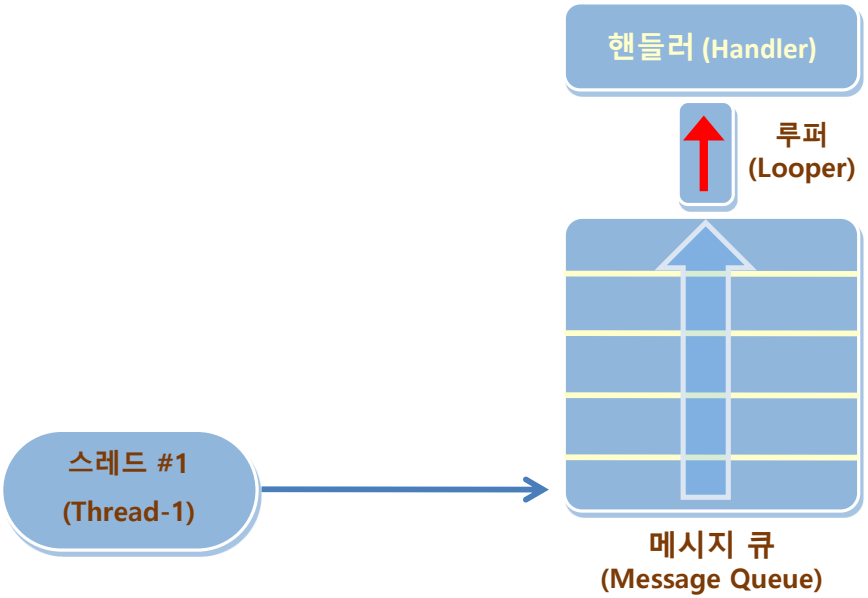


2. 스레드로 메시지 전송하기

스레드로 메시지 전송하기

- **핸들러의 기능**
  - 새로 만든 스레드에서 메인 스레드로 메시지를 전달하는 것임
- **스레드의 작업 결과물을 메시지로 만들어 전달하는 이유**
  - 별도의 스레드에서 메인 스레드가 관리하는 UI 객체에 직접 접근할 수 없기 때문임
- **메인 스레드에서 별도의 스레드로 메시지를 전달하는 방법이 필요한 경우**
  - 일반적으로 변수 선언을 통해 데이터를 전달하는 것이 가장 쉬운 방법임
  - 핸들러가 사용하는 메시지 큐를 이용하여 순차적으로 메시지를 실행하는 방식이 필요한 경우가 발생함
  - 특히 별도의 스레드가 동일한 객체에 접근할 때 다른 스레드들이 동시에 메소드를 호출하는 경우가 있을 수 있으므로 메시지 큐를 이용한 접근 방식에 대해 이해가 필요함

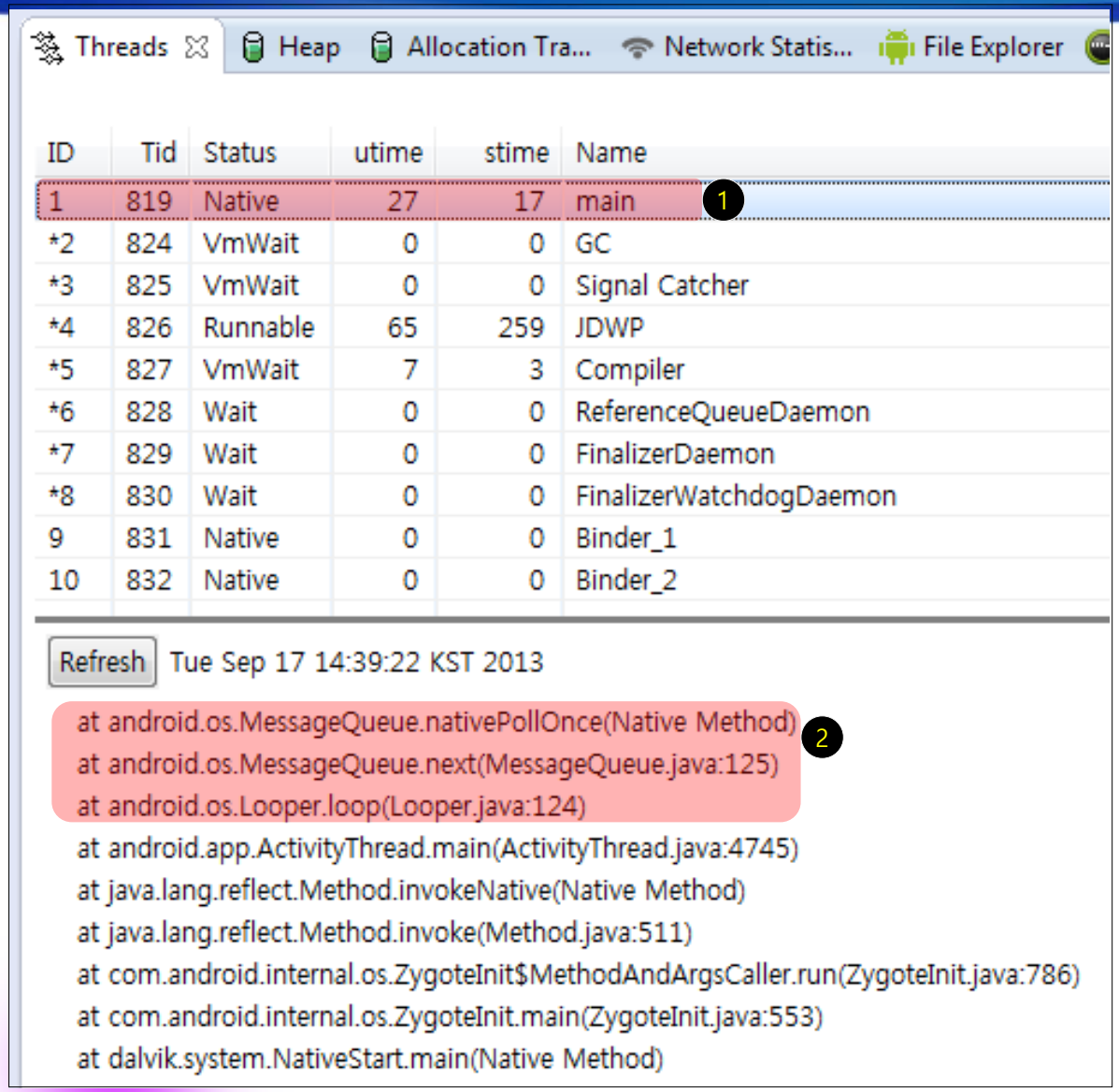
루퍼 이해하기



[루퍼를 이용한 메시지 처리]

- **루퍼**
  - 무한 루프 방식을 이용해 메시지 큐에 들어오는 메시지를 지속적으로 보면서 하나씩 처리함

루퍼 이해하기



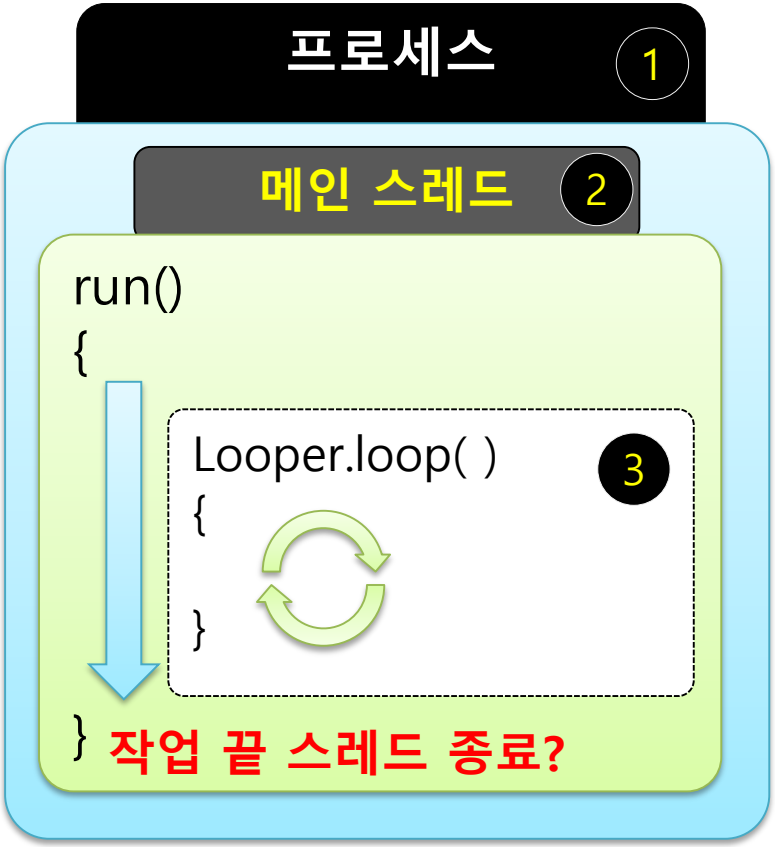
메인 스레드 콜 스텍을 보면

메시지 큐와

루퍼 객체를 계속 사용하고 있다.

루퍼 이해하기

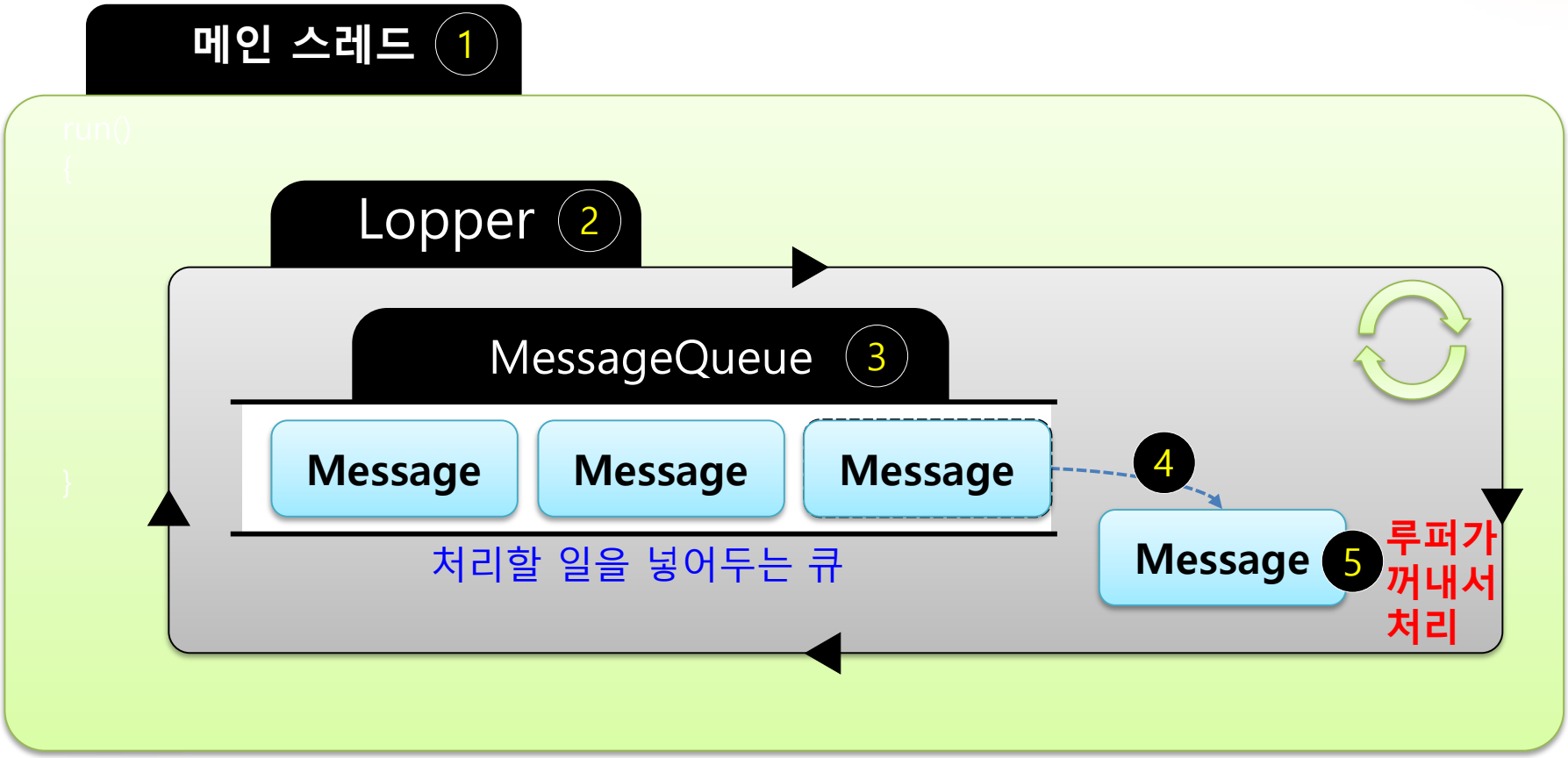
■ 메인 스레드의 루퍼



루퍼는 어떤 작업을 하며 끊임없이 반복하고 있을까?

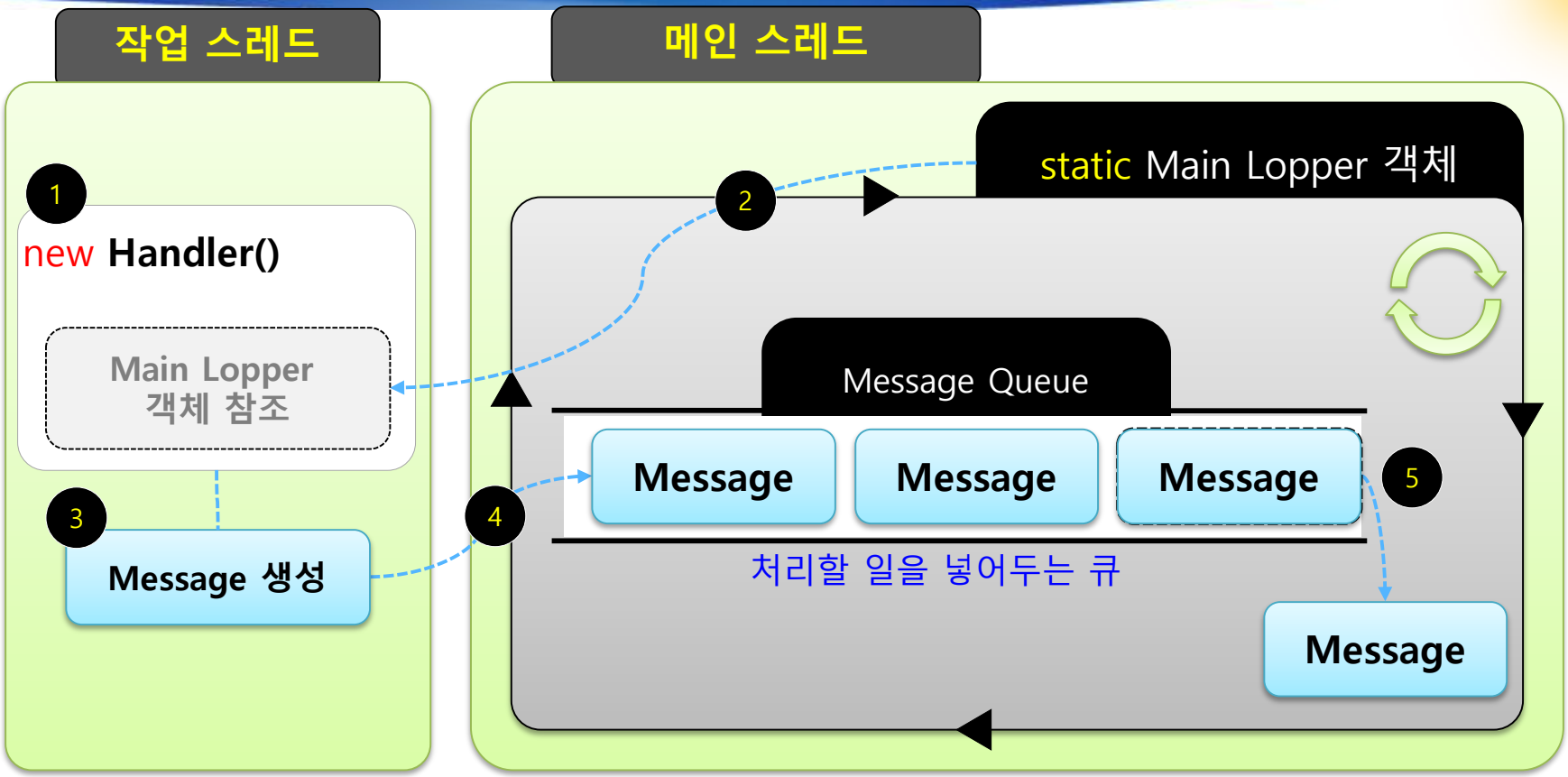
루퍼 이해하기

■ 루퍼와 메시지 큐



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

핸들러



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.

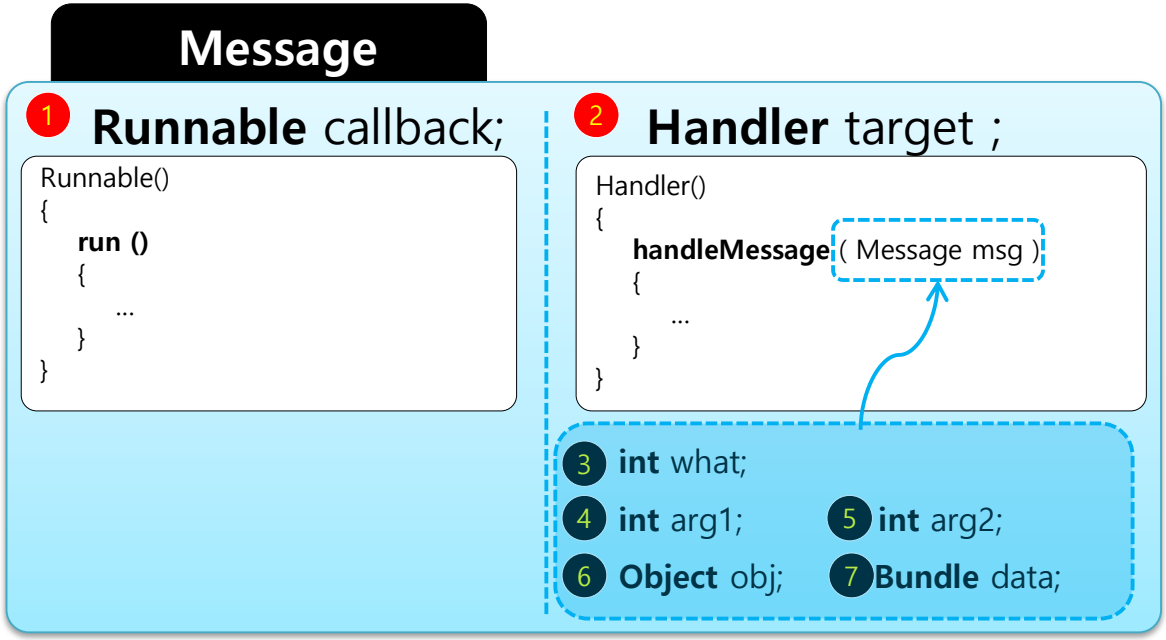
핸들러

src/ThreadActivity.java

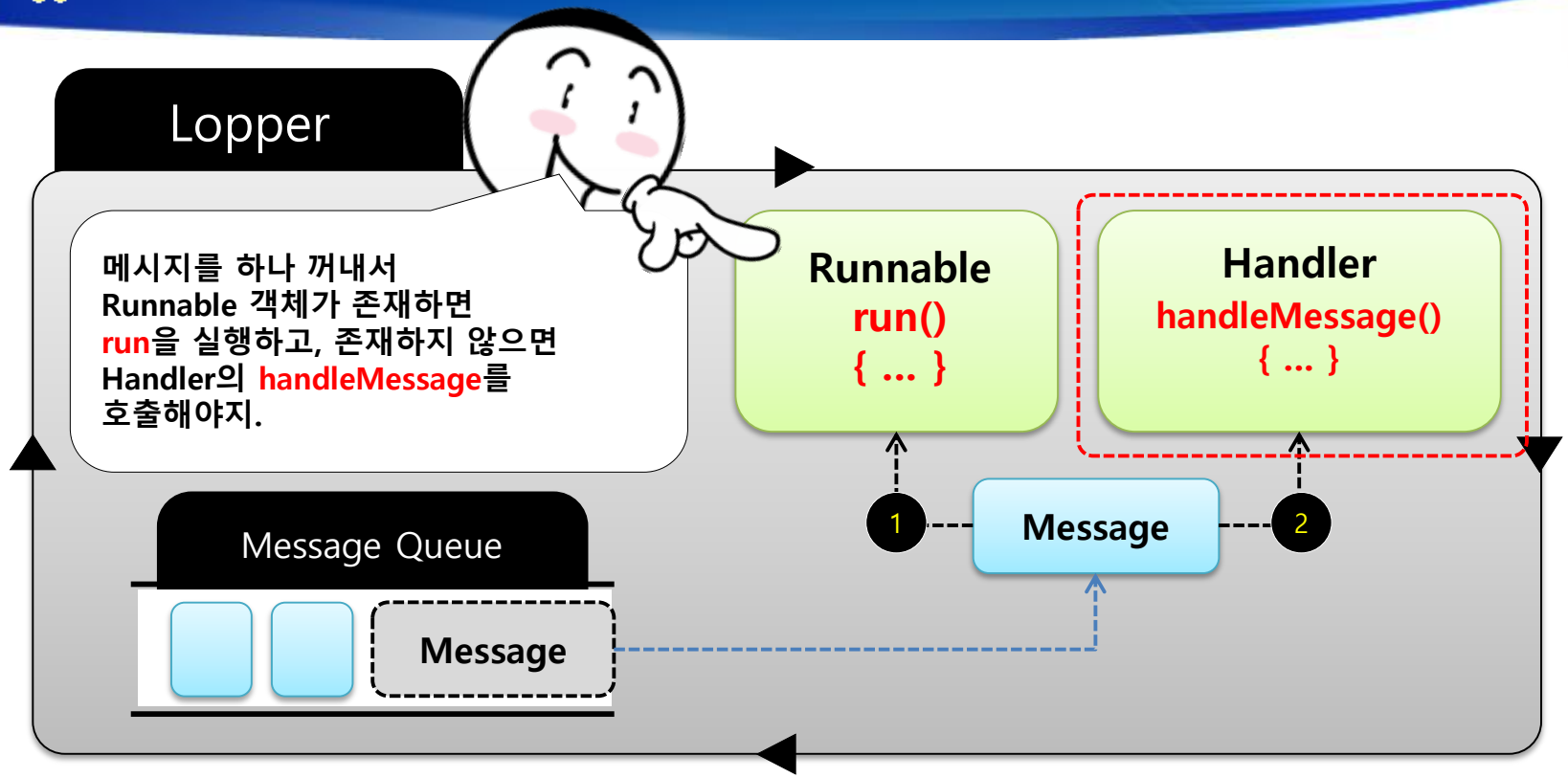
```
public class ThreadActivity extends Activity
{
    ...
    // ① 메시지큐에 메시지를 추가하기 위한 핸들러를 생성한다.
    Handler  mHandler = new Handler();

    protected void onCreate( Bundle savedInstanceState )
    {
        ...
        // ② 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 ++;

                    // ③ 실행 코드가 담긴 Runnable 객체를 하나 생성한다.
                    Runnable callback = new Runnable()
                    {
                        @Override
                        public void run()
                        {
                            // 현재까지 카운트된 수를 텍스트뷰에 출력한다.
                            Log.i("superdroid", "Count : " + mCount );
                            mCountTextView.setText( "Count : " + mCount );
                        }
                    };
```

```
                    // ④ 메시지 큐에 담을 메시지 하나를 생성한다. 생성 시
                    //    Runnable 객체를 생성자로 전달한다.
                    Message message = Message.obtain( mHandler, callback );

                    // ⑤ 핸들러를 통해 메시지를 메시지 큐에 보낸다.
                    mHandler.sendMessage( message );

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

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

    ...
}
```

■ 메시지에 할일을 담는 두 가지 방법



핸들러



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

Handler의 HandleMessage 재정의 함수에 대해서 살펴보자.

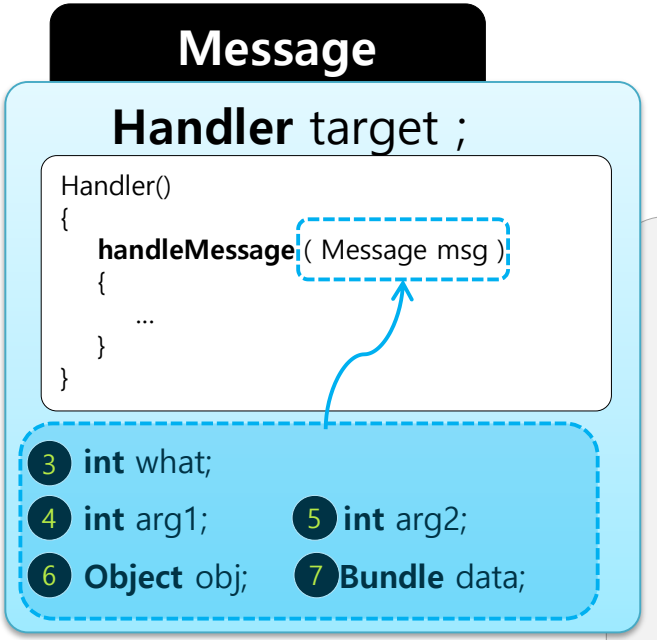
핸들러

src/ThreadActivity.java

```
public class ThreadActivity extends Activity
{
    int      mCount = 0;
    TextView mCountTextView = null;

    static final private int MESSAGE_DRAW_CURRENT_COUNT    = 1;

    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;
                }
            }
        }
        // ------------------------------------------------------------------
    };
```

```
@Override
protected void onCreate( Bundle savedInstanceState )
{
    ...
    // 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 담을 메시지 하나를 생성한다.
                // ------------------------------------------------------
                Message message = Message.obtain( mHandler );
                // ------------------------------------------------------
                // ② 핸들러의 handleMessage로 전달한 값들을 설정한다.
                // ------------------------------------------------------
                // 무엇을 실행하는 메시지 인지 구분하기 위해 구분자 설정
                message.what = MESSAGE_DRAW_CURRENT_COUNT;
                // 메시지가 실행될 때 참조하는 int형 데이터 설정
                message.arg1 = mCount;
                // 메시지가 실행될 때 참조하는 Object형 데이터 설정
                message.obj = mCountTextView;
                // ------------------------------------------------------

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

핸들러



■ **Runnable? Handler 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.

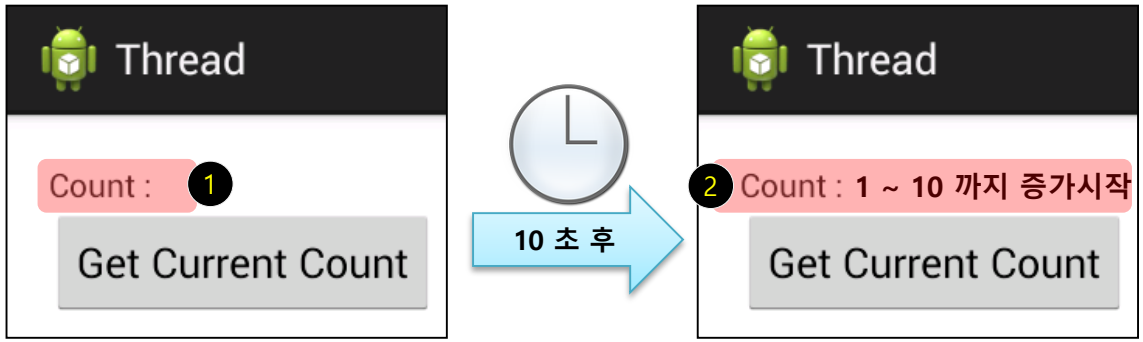
만일 handleMessage가 없다면 처리하려는 실행 코드마다 Runnable 객체를 생성하여 전달해야 할 것이다. 또한 handleMessage 함수의 경우 메시지 객체를 통해 다양한 자료형을 전달할 수 있기 때문에 더 유연하다.

■ 스케줄링이 가능한 메시지

src/ThreadActivity.java

```
public class ThreadActivity extends Activity
{


    @Override
    protected void onCreate( Bundle savedInstanceState )
    {
        ...
       Thread countThread = new Thread("Count Thread")
        {
            public void run()
            {
                for ( int i = 0 ; i < 10 ; i ++ )
                {
                    ...
                    mHandler.sendMessageDelayed( message, 10000 );

                }
            }
        };
```

핸들러



루퍼 스케줄링에 대해서는 핸들러와 메시지 함수를 배워보면서 좀더 살펴본다.

■ 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.

•**Message.*obtain*(Handler h, int what, int arg1, int arg2, Object obj)**
인자로 전달한 핸들러, what, arg1, arg2, obj 객체가 설정된 메시지 객체를 얻어온다.

핸들러

■ 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.

■ 메시지 큐에 메시지를 추가 및 삭제하는 핸들러 함수

핸들러에 post로 시작하는 함수는 모두 Runnable 객체를 이용하는 메시지 추가 함수다.

- **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.

핸들러

핸들러에 send로 시작하는 함수는 모두
Handler handleMessage 재정의 함수를 이용하는 메시지 추가 함수다.

- **mHandler.sendEmptyMessage(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.sendEmptyMessageAtTime(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.sendEmptyMessageDelayed(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.

■ 메시지 큐에 메시지를 추가 및 삭제하는 핸들러 함수

큐에 추가된 Runnable 객체 메시지는 루퍼가 실행하기 전 아래의 함수로 제거할 수 있다.

- **mHandler.removeCallbacks(Runnable r)**
인자로 전달한 Runnable 객체가 포함된 메시지를 큐에서 제거한다.

- **mHandler.removeCallbacks(Runnable r, Object token)**
삭제할 Runnable 객체 메시지 중 추가된 Object 타입의 값을 가진 메시지만 제거할 수 있다.
Runnable 객체 메시지의 Object 타입 값은 메시지를 구분하는 용도로 사용된다.

handleMessage 함수가 재정의된 메시지는 루퍼가 실행하기 전 다음의 함수로 제거할 수 있다.

- **mHandler.removeMessages(int what)**
인자로 전달한 what 값이 설정된 메시지를 큐에서 제거한다.

- **mHandler.removeMessages(int what, Object object)**
인자로 전달한 what 값이 설정되고, object 객체가 포함된 메시지를 큐에서 제거한다.

다음은 Runnable 객체 메시지와 핸들러 메시지 모두 큐에서 제거하는 함수며, 두 가지 타입의 메시지를 모두 제거할 수 있는 유일한 함수이기도 하다.

- **mHandler.removeCallbacksAndMessages(Object token)**
인자로 전달한 object 객체가 포함된 메시지를 큐에서 제거한다.

메인 액티비티 코드 만들기

```
public class MainActivity extends AppCompatActivity {
  TextView textView01, textView02;
  EditText editText01, editText02;

  MainHandler mainHandler;
  ProcessThread thread1;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mainHandler = new MainHandler();


    thread1 = new ProcessThread();

    textView01 = (TextView) findViewById(R.id.textView01);
    textView02 = (TextView) findViewById(R.id.textView02);
    editText01 = (EditText) findViewById(R.id.editText01);
    editText02 = (EditText) findViewById(R.id.editText02);
```

1 새로 정의한 스레드 객체 생성

- 첫 번째 입력상자에 글자를 넣고 버튼을 누르면 입력된 메시지에 " Mike!!!"라는 문자열을 덧붙여 두 번째 입력상자에 보여주게 됨

- 내부적으로는 메인 스레드에서 새로 만든 별도의 스레드로 Message 객체를 전송하고, 별도의 스레드에서는 전달받은 문자열에 다른 문자열을 덧붙여 메인 스레드 쪽으로 다시 전송하는 과정을 거치게 됨

메인 액티비티 코드 만들기 (계속)

```
Button processBtn = (Button) findViewById(R.id.processBtn);
  processBtn.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
      String inStr = editText01.getText().toString();
      Message msgToSend = Message.obtain();
      msgToSend.obj = inStr;


      thread1.handler.sendMessage(msgToSend);
    }
  });

  thread1.start();
}

class ProcessThread extends Thread {

  ProcessHandler handler;

  public ProcessThread() {
    handler = new ProcessHandler();
  }
```

2 버튼을 눌렀을때 스레드 내의 핸들러로 메시지 전송

3 스레드 내에 선언된 핸들어 객체

- ProcessHandler 클래스와 MainHandler 클래스는 각각 새로 만든 스레드와 메인 스레드에서 만들어지고 사용됨
- Message 객체는 Message.obtain() 메소드를 이용해 참조할 수 있으며, 새로 만든 스레드(thread1)의 handler 변수를 이용해 sendMessage() 메소드를 호출하면 메시지 객체가 스레드로 전송됨

메인 액티비티 코드 만들기 (계속)

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

class ProcessHandler extends Handler {
  public void handleMessage(Message msg) {
    Message resultMsg = Message.obtain();
    resultMsg.obj = msg.obj + " Mike!!!";

    mainHandler.sendMessage(resultMsg);
  }
}

class MainHandler extends Handler {
  public void handleMessage(Message msg) {
    String str = (String) msg.obj;

    editText02.setText(str);
  }
 }
}
```

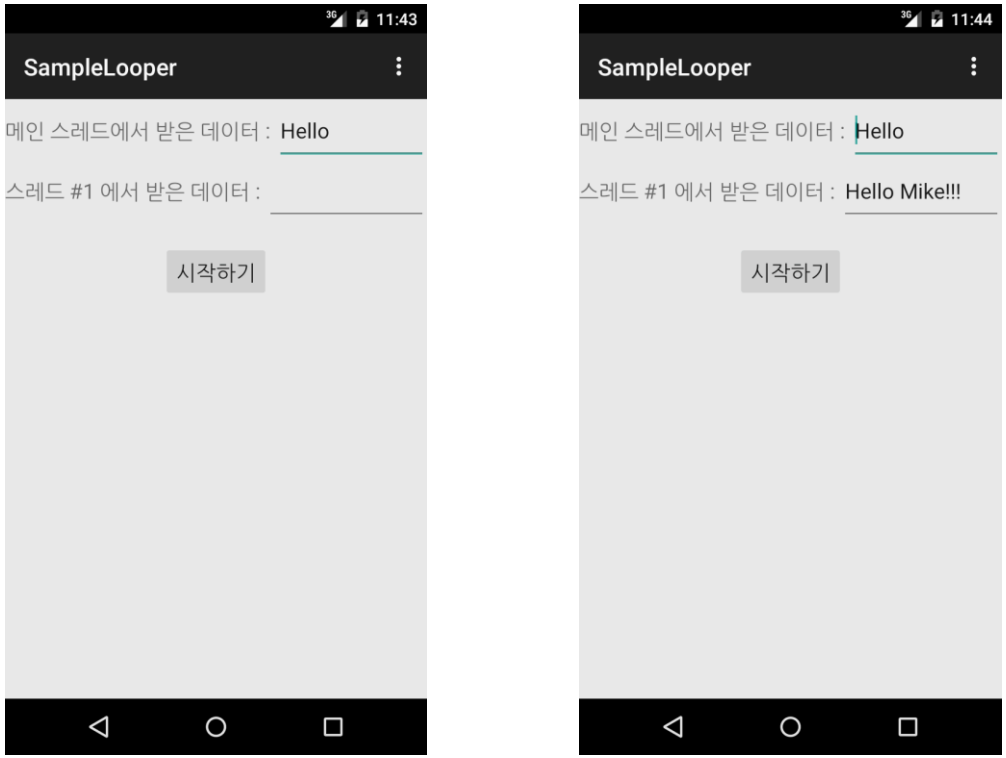
4 스레드의 run() 메소드 안에서 루퍼 실행

5 스레드 내의 핸들러에서 메인 스레드의 핸들러로 메시지 전송

6 메인 스레드의 핸들러 내에서 입력상자의 메시지 표시

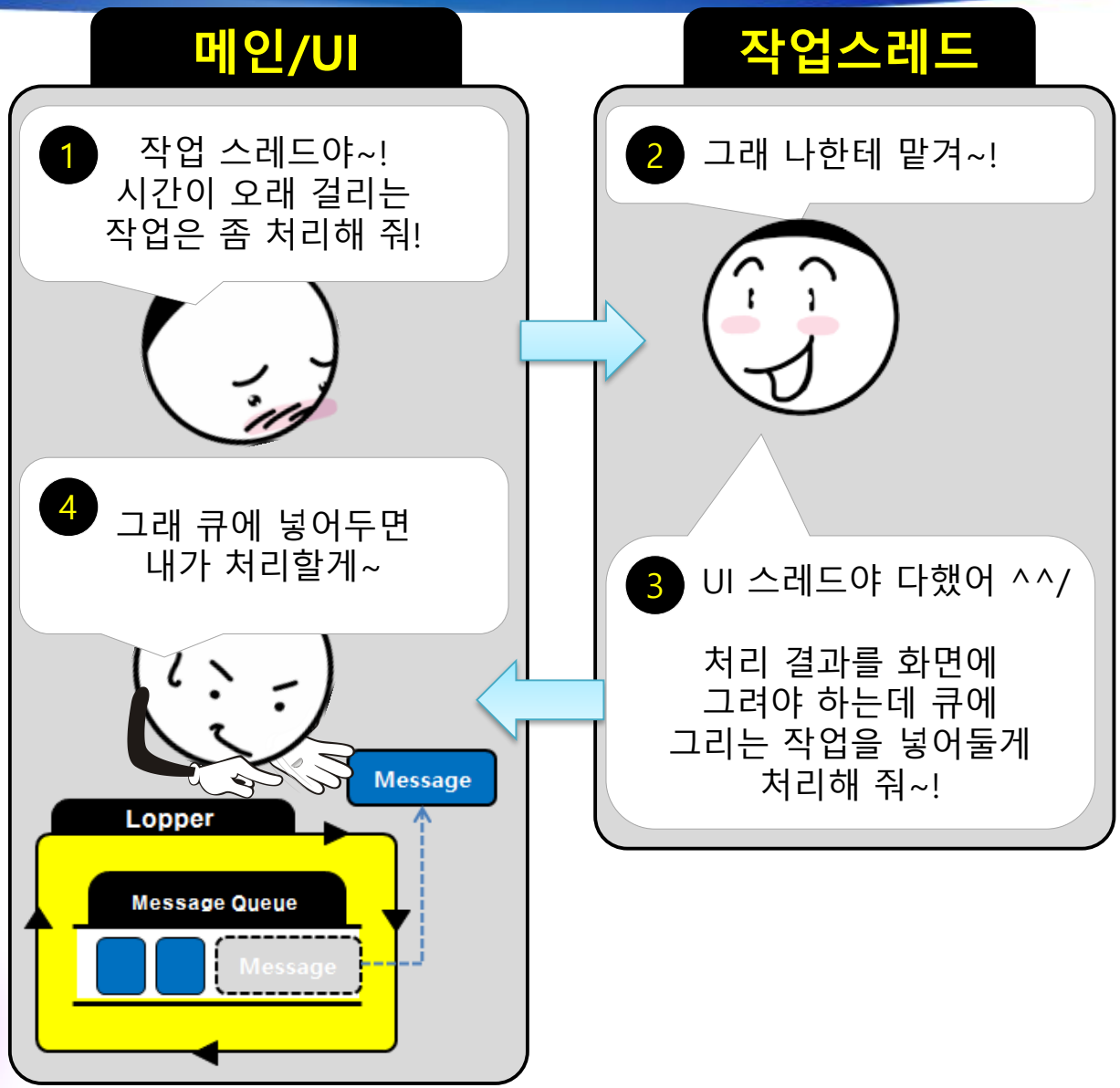
- ProcessHandler 클래스에 정의된 handleMessage() 메소드에서는 전달받은 Message 객체의 obj 변수에 들어있는 문자열을 이용해 새로운 문자열을 만든 후 메인 스레드에서 만들어진 MainHandler 객체를 이용해 sendMessage() 메소드를 호출함

실행 화면

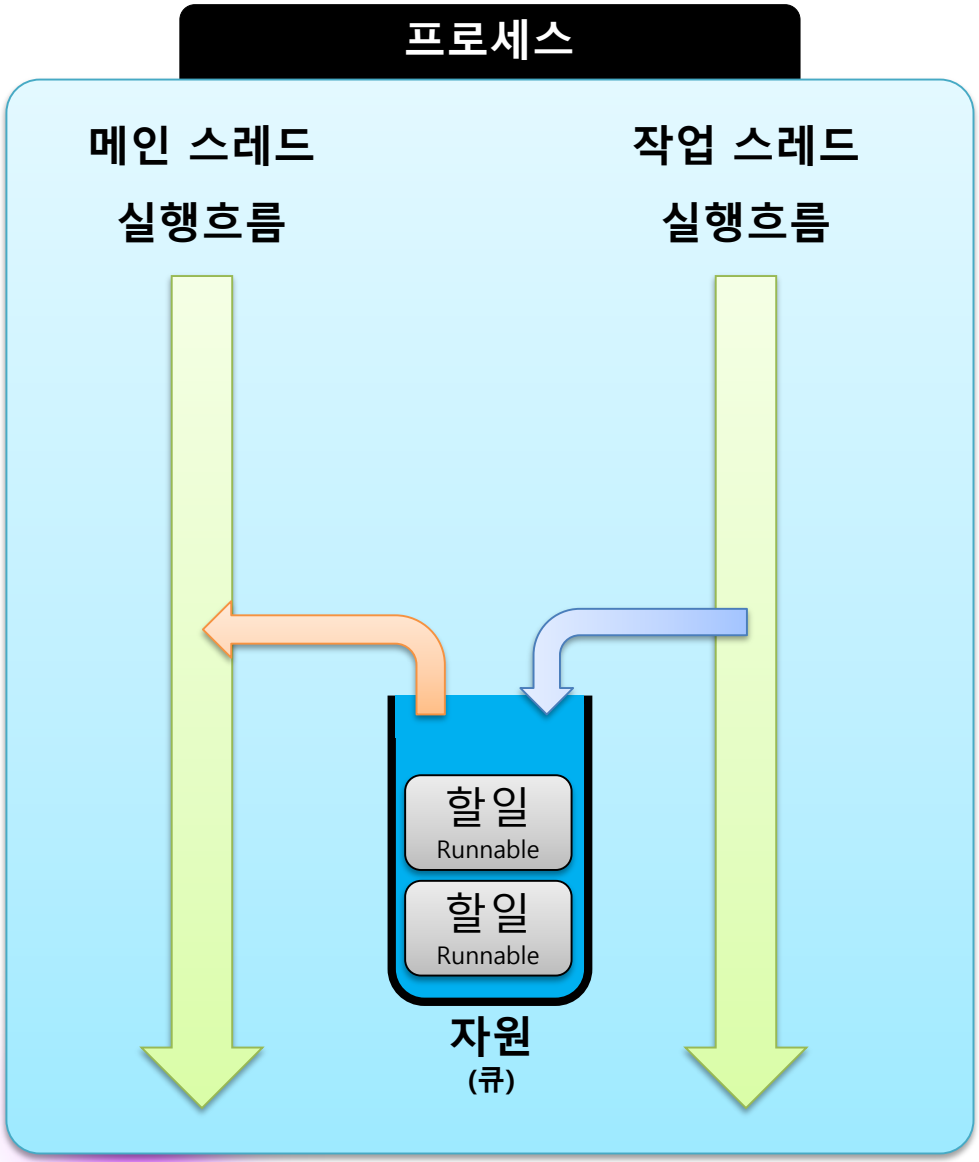


[루퍼를 이용한 스레드 간 문자열 전달]

메인 스레드와 작업 스레드에 대한 정리



메인 스레드와 작업 스레드에 대한 정리



특정 스레드에서 처리해야 할 작업을
다른 스레드의 실행흐름에서 처리하려면
어떻게 해야할까?

자원에 처리해야 할 일들을 넣고
다른 스레드에서 꺼내서 처리해야 한다.
이 방법 이외엔 없다.

따라서 메인 스레드가
큐를 이용하는 것은
당연한 것이다.

뭐든지 당연한 원리라 생각하면
이해하기 쉽고 오래 기억된다.

3. AsyncTask 사용하기

AsyncTask 사용하기

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

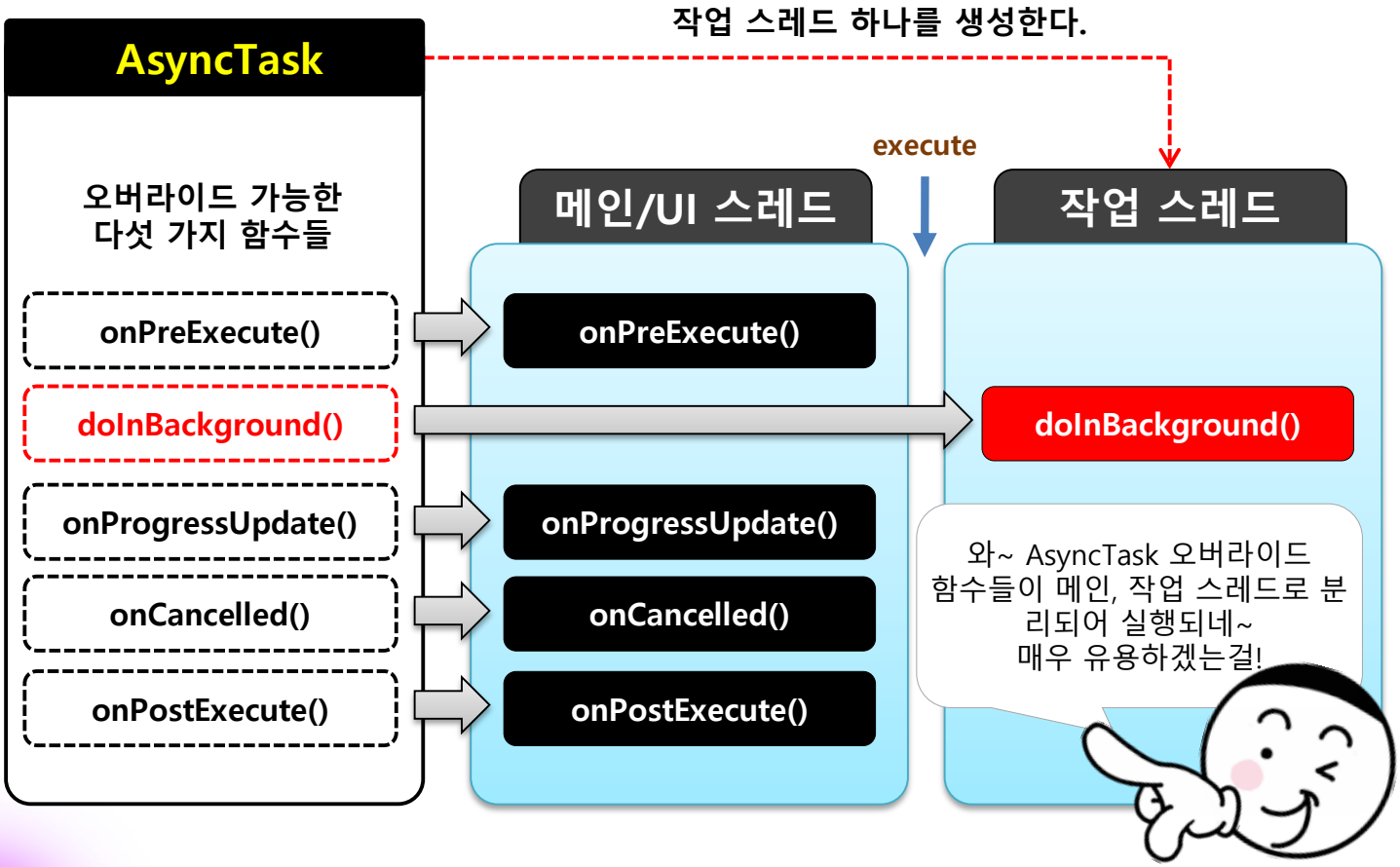
하지만 이렇게 스레드마다 처리해야 하는 일이 엄격히 구분되는 것은 개발에 있어 부담감이 크다.

안드로이드에서는 이러한 복잡성을 줄이기 위해 다양한 도우미 클래스를 제공한다.

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

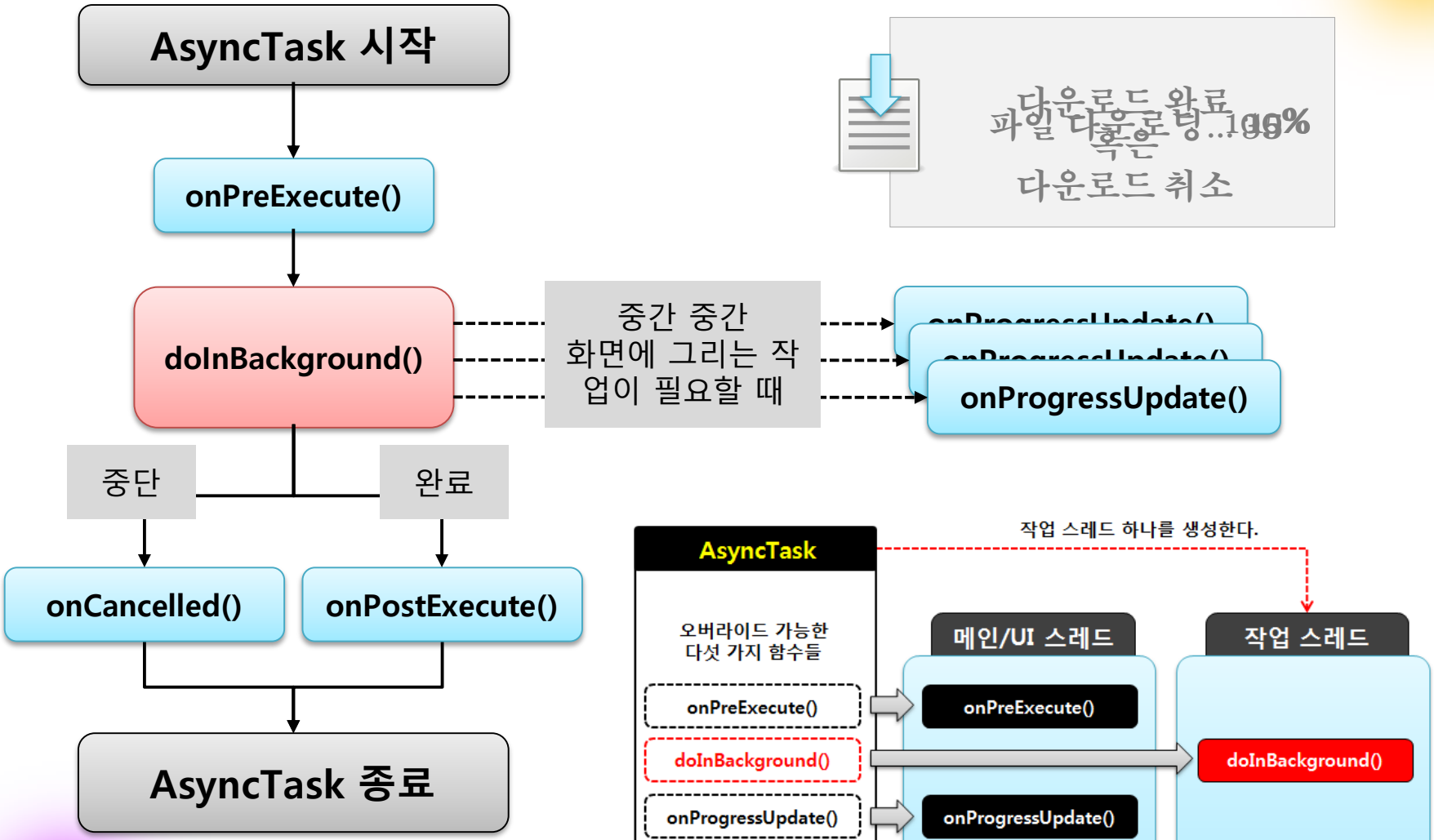
AsyncTask 사용하기

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



- AsyncTask 객체를 만들고 execute() 메소드를 실행하면 이 객체는 정의된 백그라운드 작업을 수행하고 필요한 경우에 그 결과를 메인 스레드에서 실행하므로 UI 객체에 접근하는데 문제가 없게 됨

AsyncTask의 주요 메소드



AsyncTask의 주요 메소드

메소드 이름	설 명
doInBackground	- 새로 만든 스레드에서 백그라운드 작업 수행 - execute() 메소드를 호출할 때 사용된 파라미터를 배열로 전달받음
onPreExecute	- 백그라운드 작업 수행 전 호출 - 메인 스레드에서 실행되며 초기화 작업에 사용
onProgressUpdate	- 백그라운드 작업 진행 상태를 표시하기 위해 호출 - 작업 수행 중간 중간에 UI 객체에 접근하는 경우 사용 - 이 메소드가 호출되도록 하려면 백그라운드 작업 중간에 publishProgress() 메소드를 호출
onPostExecute	- 백그라운드 작업이 끝난 후 호출 - 메인 스레드에서 실행되며 메모리 리소스를 해제하는 등의 작업에 사용 - 백그라운드 작업의 결과는 Result 타입의 파라미터로 전달

AsyncTask의 주요 메소드

```
private class FileDownloadTask
                    extends AsyncTask< String, Integer, Boolean >
{
    @Override
    protected Boolean doInBackground( String... fileUrls )
    {
        int totolCount = fileUrls.length;

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

        return true;
    }

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

        mDownloadStateTextView.setText( "
                                currentCo
    }

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

1 시작 2 중간 3 끝

작업 스레드 기준으로 시작, 중간 끝에 필요한 값들이다.

```
@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");
}
```

Async 메소드 사용하기

- AsyncTask 객체의 cancel() 메소드
  - 작업을 취소함, 이 메소드를 통해 작업을 취소했을 경우에는 onCancelled() 메소드가 호출됨

- AsyncTask 객체의 getStatus() 메소드
  - 작업의 진행 상황을 확인함
  - 메소드를 호출했을 때 리턴되는 AsyncTask.Status 객체는 상태를 표현함
  - 각각의 상태는 PENDING, RUNNING, FINISHED로 구분됨

- PENDING
  - 작업이 아직 시작되지 않았다는 것을 의미함

- RUNNING
  - 실행 중임을 의미함

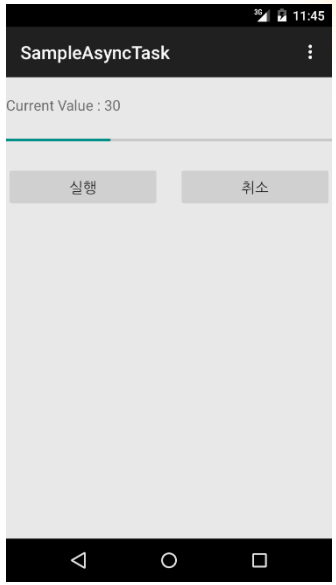
- FINISHED
  - 종료되었음을 의미함

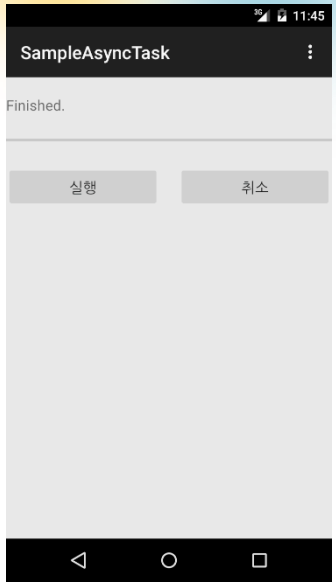
AsyncTask 사용하기 예제

AsyncTask 사용하기 예제

-AsyncTask를 이용해 진행상태 표시

-기존 핸들러 방식을 AsyncTask 방식으로 변경





XML 레이아웃 정의

-프로그레스바를 포함하는 레이아웃 정의

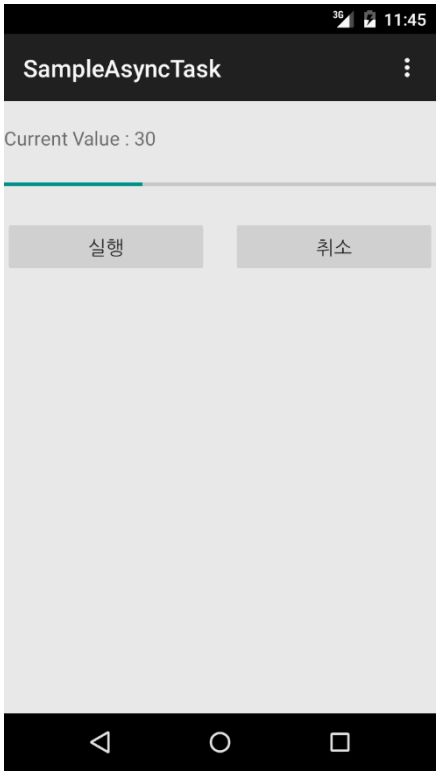
메인 액티비티 코드 작성

-필요한 작업을 위한 메인 액티비티 코드 작성

AsyncTask 클래스 정의

-AsyncTask를 상속하는 새로운 클래스를 내부 클래스로 정의

화면 레이아웃 구성하기



[AsyncTask를 이용해 진행 상태를 표시하는 화면 구성]

- 이 화면의 버튼을 클릭하면 별도의 스레드에서 값을 1씩 증가시키도록 함
- 100밀리 초마다 한 번씩 값을 증가시키므로 프로그레스바의 최대값으로 지정된 100이 될 때까지 10초가 걸리게 됨

메인 액티비티 코드 만들기

- 백그라운드 작업을 수행할 클래스는 BackgroundTask라는 이름의 클래스로 정의하고 AsyncTask 클래스를 상속받음

- onPreExecute() 메소드는 초기화 단계에서 사용되므로 값을 저장하기 위해 메인 액티비티에 정의한 value 변수의 값을 0으로 초기화하고 프로그레스바의 값도 0으로 만들어 줌

- doInBackground() 메소드는 주 작업을 실행하는데 사용되므로 while 구문을 이용해 value의 값을 하나씩 증가시키도록 함

```
Button executeBtn = (Button) findViewById(R.id.executeBtn);
executeBtn.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
      task = new BackgroundTask();
      task.execute(100);
    }
});
```

AsyncTask에서 상속한 클래스 정의

```
Button cancelBtn = (Button) findViewById(R.id.cancelBtn);
  cancelBtn.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {

      task.cancel(true);
    }
  });
}
```

3 [취소] 버튼을 눌렀을 때 cancel() 메소드 실행

```
class BackgroundTask extends AsyncTask<Integer , Integer , Integer> {
  protected void onPreExecute() {
    value = 0;
    progress.setProgress(value);
  }
```

4 AsyncTask를 상속하여 새로운 BackgroundTask 클래스 정의

- 중간 중간 진행 상태를 UI에 업데이트하도록 만들기 위해 publishProgress() 메소드를 호출함
- onProgressUpdate() 메소드는 doInBackground() 메소드 안에서 publishProgress() 메소드가 호출될 때마다 자동으로 호출됨
- 이 안에서는 프로그레스바의 값을 전달된 파라미터의 값(여기서는 value 변수의 값과 동일)으로 설정함

AsyncTask에서 상속한 클래스 정의 (계속)

```
protected Integer doInBackground(Integer ... values) {
    while (isCancelled() == false) {
      value++;
      if (value >= 100) {
        break;
      } else {

        publishProgress(value);
      }

      try {
        Thread.sleep(100);
      } catch (InterruptedException ex) {}
    }

    return value;
  }
```

5 doInBackground() 메소드 내에서 publishProgress() 메소드 호출

- 텍스트뷰에도 파라미터의 값을 설정하여 수치를 볼 수 있도록 함
- onPostExecute() 메소드는 스레드에서 수행되던 작업이 종료되었을 때 호출되므로 프로그레스바의 값을 0으로 설정하고 텍스트뷰에는 "Finished."라는 문자열을 보여줌

AsyncTask에서 상속한 클래스 정의 (계속)

```
    protected void onProgressUpdate(Integer ... values) {
      progress.setProgress(values[0].intValue());
      textView01.setText("Current Value : " + values[0].toString());
    }

    protected void onPostExecute(Integer result) {
      progress.setProgress(0);
      textView01.setText("Finished.");
    }

    protected void onCancelled() {
      progress.setProgress(0);
      textView01.setText("Cancelled.");
    }
  }
}
```

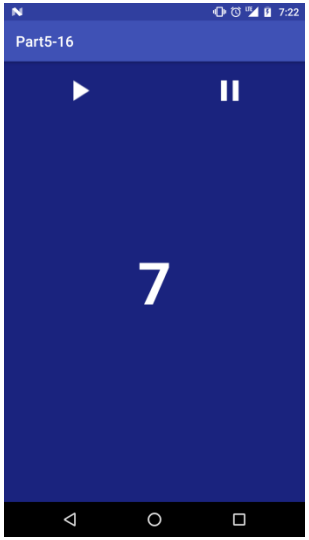
6 onProgressUpdate() 메소드 내에서 프로그레스바와 텍스트뷰 변경

- onCancelled() 메소드는 작업이 취소되었을 때 호출되므로 프로그레스바의 값은 똑같이 0으로 설정한 후 텍스트뷰에 "Cancelled."라는 문자열을 보여줌

AsyncTask작성하기

AsyncTask를 테스트

Thread - Handler 로 작성하기의 Project를 AsyncTask를 이용해 작성하는 테스트



Looper를 이용하는 테스트

스레드를 이용해 이미지 이동시키기

1. activity_main.xml 파일을 열고 화면의 좌측 상단에 이미지뷰를 두 개 추가한 후 각각 강아지 이미지가 보이도록 합니다
2. MainActivity.java 파일을 열고 한 마리 강아지를 터치하면 가로 방향으로 끝까지 갔다가 돌아오도록 코드를 입력합니다
3. 다른 한 마리 강아지를 터치하면 세로 방향으로 끝까지 갔다가 돌아오도록 코드를 입력합니다
4. 사용자가 강아지를 언제 터치하든 상관없이 동작할 수 있도록 두 개의 스레드를 만들어 각 스레드가 이미지를 움직이도록 만듭니다. 이미지뷰의 margin 값을 수정하면 이미지뷰가 보이는 위치를 변경할 수 있습니다

스레드를 이용해 이미지 이동시키기

<출력화면>