

ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ  
ПРИЛОЖЕНИЙ ПОД ПЛАТФОРМУ

# ANDROID



# Урок № 7

## Многопоточность. Сетевое программирование

## Содержание

<b>1. Многопоточность в Android приложениях.....</b>	<b>4</b>
1.1. Запуск вторичных потоков .....	5
1.2. Класс java.lang.Thread .....	11
1.3. Завершение (прерывание) работы вторичных потоков .....	17
1.4. Взаимодействие вторичных потоков с виджетами. Метод runOnUiThread .....	41
1.5. Приостановка и возобновление работы потоков .....	52
1.6. Вторичные потоки и жизненный цикл Активности .....	66
1.7. Классы Handler, Message .....	71
<b>2. Сетевое программирование для мобильных устройств Android .....</b>	<b>85</b>
2.1. Класс android.net.ConnectivityManager. Выбор сетевого подключения. Получение информации о сетевых подключениях.....	86

2.2. Получение информации о сетевом подключении.	
Класс android.net.NetworkInfo .....	89
2.3. Сетевое подключение.	
Класс android.net.Network .....	105
2.4. Отправка данных в HTTP запросе методом GET .....	120
2.5. Отправка данных в HTTP запросе методом POST ....	125
<b>3. Домашнее задание .....</b>	<b>130</b>
<b>4. Экзаменационное задание .....</b>	<b>132</b>

# 1. Многопоточность в Android приложениях

Современные мобильные устройства представляют из себя карманные персональные компьютеры, которые по своим техническим характеристикам не так уж сильно отстают от настольных персональных компьютеров. Приложения для мобильных устройств могут решать задачи такой же сложности, как и задачи, которые решаются приложениями настольных компьютеров. Причем возможность одним приложением выполнять несколько задач одновременно (то есть быть многозадачным приложением) является ключевой как для приложений настольных компьютеров, так и для приложений мобильных устройств. В данном разделе мы рассмотрим как создаются многозадачные Android приложения.

Наверняка вы уже знаете, что многозадачность приложений обеспечивается операционными системами, в которых эти приложения исполняются. Исполняемое в операционной системе приложение называется процессом. Процесс (*Process*) это экземпляр исполняемого приложения. Сам по себе процесс не выполняет никаких задач. Процесс владеет виртуальным адресным пространством и является контейнером для потоков исполнения. Каждый поток исполнения (*Thread*) предназначен для решения одной задачи. Операционная система обеспечивает одновременное (или почти одновременное) исполнение

ние потоков, позволяя тем самым процессу (исполняемому приложению) одновременно решать несколько задач. При старте процесса операционная система запускает только один поток. Этот поток называется Первичным Потоком. Остальные потоки (их называют Вторичными Потоками) в процессе запускаются при помощи программного кода, который написали разработчики приложения. Как только в процессе не остается ни одного потока исполнения, операционная система уничтожает этот процесс и приложение завершает свою работу. Вот так, вкратце, мы повторили с вами основы системного программирования. Если вы, прочитав этот абзац, не поняли о чем идет речь, мы настоятельно рекомендуем вам вернуться к темам «Системного программирования» из предыдущих изучаемых курсов и только потом рекомендуем вам приступить к изучению тем этого урока.

## 1.1. Запуск вторичных потоков

Механизм запуска вторичных потоков в Android приложениях аналогичен механизму запуска вторичных потоков в Java. Давайте их рассмотрим.

Вторичный поток начинает свою работу со специальной функции, которая называется Главной Функцией вторичного потока. Вторичный поток работает до тех пор, пока не покинет главную функцию. Для того, чтобы создать класс в котором будет главная функция для вторичного потока, необходимо или чтобы этот класс был наследником от класса [java.lang.Thread](#) или чтобы этот класс реализовал интерфейс [java.lang.Runnable](#). Ре-

командуемый вариант — это создание производного от **java.lang.Thread** класса. А вот если создать производный класс не получается по причине того, что создаваемый класс уже наследует другой класс, то в этом случае остается вариант с реализацией интерфейса **java.lang.Runnable** (ведь множественного наследования в Java нет). В интерфейсе **java.lang.Runnable** объявлен всего один метод (см. Листинг 1.1) — это метод **void run()**. Этот метод **run()** и должен быть главной функцией вторичного потока.

### Листинг 1.1. Интерфейс **java.lang.Runnable**

```
public interface Runnable
{
    void run();
}
```

Класс **java.lang.Thread** реализует интерфейс **java.lang.Runnable** (см. Листинг 1.2).

### Листинг 1.2. Реализация классом **java.lang.Thread** интерфейса **java.lang.Runnable**

```
public class Thread implements Runnable
{
    @Override
    public void run()
    {
    }
}
```

Как видно из Листинга 1.2, в классе **java.lang.Thread**, метод **run()** интерфейса **java.lang.Runnable** реализован пустым. Это сделано специально — конкретная реализа-

ция метода **run()** должна быть выполнена в производных классах путем переопределения метода **run()**.

Давайте создадим класс для потока исполнения путем наследования класса **java.lang.Thread**. В рассматриваемом примере класс назовем **FirstThr**. Листинг класса **FirstThr** приведен в Листинге 1.3.

**Листинг 1.3.** Создание класса потока исполнения путем наследования класса **java.lang.Thread**

```
class FirstThr extends Thread
{
    private final static String THR_TAG = "FirstThr";
    /**
     * Message that will output the thread
     *
     * Сообщение, которое будет выводить поток
     */
    private String msg;
    public FirstThr(String msg)
    {
        this.msg = msg;
    }

    @Override
    public void run()
    {
        try
        {
            while (true)
            {
                Thread.sleep(750);
                Log.d(THR_TAG, this.msg + "(" +
                    Thread.currentThread().getName() +
                    ")");
            }
        }
    }
}
```

```
        }
    catch (InterruptedException ie)
    {
        Log.d(THR_TAG, "Поток прерван : " +
                Thread.currentThread().getName());
    }
}
```

Как видно из Листинга 1.3, потоку исполнения на этапе создания объекта (при помощи конструктора) будет передано некоторое строковое сообщение, которое поток исполнения будет периодически (с периодом 750 миллисекунд) выводить в консоль. Запустим поток на основе класса `FirstThr` в методе `onCreate()`. Запуск этого потока показан в Листинге 1.4.

## **Листинг 1.4.** Запуск потока исполнения на основе класса FirstThr

```
public class MainActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
    }
    // ----- Starting The Thread -----
    // ----- Запуск потока -----
    FirstThr FT = new FirstThr("Hello World!");
    FT.start();
}
}
```

Результат исполнения примера из Листинга 1.4 приведен в Листинге 1.5.

**Листинг 1.5.** Консольный вывод во время работы потока исполнения из Листингов 1.3 и 1.4

```
09-04 16:01:24.720 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:01:25.480 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:01:26.240 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:01:27.000 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:01:27.760 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
```

Обратите внимание (см. Листинг 1.3), что при выводе сообщения в консоль при помощи метода `Log.d()`, поток также выводит и свое системное имя (в Листинге 1.5 системное имя выводится в круглых скобках). Это сделано специально, чтобы продемонстрировать следующую особенность — при повороте устройства поток исполнения не только продолжит свою работу, но и будет запущен еще один поток `FirstThr`. Это происходит потому, что при повороте устройства приложение продолжает свою работу, а значит и продолжает работать поток исполнения, запущенный в Листинге 1.4. Однако, при повороте устройства происходит пересоздание Активности приложения (повторите учебный материал «Жизненный цикл Активности» из первого урока данного курса), следовательно, метод `onCreate()` в нашем приложении запуститься еще раз и это приведет к созданию еще одного

экземпляра потока исполнения **FirstThr**. Результат можно увидеть в Листинге 1.6 — выводятся разные системные имена потоков, а значит работает не один поток.

**Листинг 1.6.** Консольный вывод потоков исполнения FirstThr после поворота устройства

```
09-04 16:02:17.160 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:02:17.470 6778-7390/itstep.com.app
    D/FirstThr: Hello World! (Thread-125)
09-04 16:02:17.920 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:02:18.230 6778-7390/itstep.com.app
    D/FirstThr: Hello World! (Thread-125)
09-04 16:02:18.680 6778-6799/itstep.com.app
    D/FirstThr: Hello World! (Thread-123)
09-04 16:02:18.990 6778-7390/itstep.com.app
    D/FirstThr: Hello World! (Thread-125)
```

Необходимо учитывать эту особенность при разработке приложений под мобильные устройства Android. В Листинге 1.7 показано, как можно избавиться от создания ненужных дополнительных экземпляров потоков.

**Листинг 1.7.** Решение, избавляющее от создания ненужных экземпляров потока исполнения при повороте устройства

```
public class MainActivity extends AppCompatActivity
{
    // ----- Class members -----
    private static FirstThr FT = null;

    // ----- Class methods -----
    @Override
    protected void onCreate(Bundle savedInstanceState)
```

```
{  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    ...  
    // ----- Starting The Thread -----  
    // ----- Запуск потока -----  
    if (MainActivity.FT == null)  
    {  
        MainActivity.FT = new FirstThr("Hello World!");  
        MainActivity.FT.start();  
    }  
}
```

Как видно из Листинга 1.7, проблема решается при помощи статического поля (важные фрагменты кода выделены в Листинге 1.7 жирным шрифтом). В методе **onCreate()** происходит проверка значения статического поля на равенство значению **null**. Если поле равно **null** то Активность запускается впервые и необходимо запустить поток, в противном случае, запуск потока производить не нужно так как он уже запущен. Запустите пример из Листинга 1.7 и переверните устройство несколько раз — дополнительных потоков не создаться.

Исходный код примера из данного раздела находится в модуле «app14» среди файлов исходных кодов которые прилагаются к данному уроку.

## 1.2. Класс `java.lang.Thread`

Для дальнейшего изучения темы многопоточности в Android приложениях, давайте рассмотрим наиболее важные методы класса `java.lang.Thread`. Возможно, вы

уже знакомы с классом **java.lang.Thread** из предыдущих курсов, тогда материал этого раздела будет для вас полезным повторением пройденного материала.

Итак, методы класса **java.lang.Thread** которые будут использоваться в этом и в последующих уроках данного курса:

- **static Thread currentThread()** — статический метод, возвращает ссылку на объект **java.lang.Thread** текущего потока исполнения.
- **final String getName()** — возвращает имя потока **java.lang.Thread**. Имя потока полезно использовать на этапе разработки и отладки приложений. По умолчанию, если имя потока не задано, поток автоматически получает имя (Как правило — Thread-номер) от системы. Задать имя потоку можно при помощи метода **setName()**.
- **void interrupt()** — метод предназначен для прерывания работы потока. Вызов этого метода не приведет к прерыванию работы потока, а приведет к генерации Исключительной ситуации **java.lang.Interrupted-Exception** внутри потока исполнения, но только в том случае, если поток исполнения находится внутри вызовов методов **wait()**, **sleep()**, **join()** которые описаны ниже. Также поток может узнать о том, что его прерывают при помощи вызовов метода **interrupted()**, который описан ниже. Вызов метода **interrupt()** не гарантирует прерывание потока, так как прерывание потока зависит от программного кода, который обрабатывает ситуацию прерывания потока.

- **static boolean interrupted()** — статический метод. Проверяет, вызывался ли для потока, в котором вызван этот метод, метод **interrupt()**. Возвращает **true** если метод **interrupt()** вызывался, при этом, повторные вызова метода **interrupted()** будут возвращать **false**, до тех пор, пока вновь не будет вызван метод **interrupt()**.
- **final void join()** — приостанавливает исполнение потока, в котором был вызван метод **join()**, до тех пор пока поток, для которого был вызван метод **join()** не завершит свою работу. Здесь необходимы пояснения и пример программного кода (Листинг 1.8). Предположим, что код из Листинга 1.8 исполняется в первичном потоке. В строке 01 создается объект потока **T**. В строке 02 происходит запуск потока **T**. А в строке 03 первичный поток приостанавливается до тех пор пока поток **T** не завершит свою работу. Первичный поток продолжит свою работу только после завершения работы потока **T**. Метод **join()** генерирует исключительную ситуацию **java.lang.InterruptedException**, которая означает, что текущий поток прерывается при помощи метода **interrupt()**.

#### Листинг 1.8. Код для разъяснения работы метода **join()**

```
01 SomeThreadClass T = new SomeThreadClass();  
02 T.start();  
03 T.join();
```

- **final void join(long millis)** — То же самое, что и предыдущий метод, только в качестве параметра пере-

дается время в миллисекундах, указывающее, сколько времени поток, в котором вызван метод `join()` готов ожидать завершения работы потока, для которого метод `join()` вызывается. Метод `join()` генерирует исключительную ситуацию `java.lang.Interrupted-Exception`, которая означает, что текущий поток прерывается при помощи метода `interrupt()`.

- **final void setDaemon(boolean on)** — если передать в параметр значение `true`, то поток становится фоновым. По умолчанию, все создаваемые потоки не фоновые. Отличие между фоновыми и не фоновыми потоками заключается в том, что если в процессе не останется ни одного не фонового потока, то такой процесс будет уничтожен операционной системой, даже при наличии в процессе фоновых потоков. В Android приложениях этот метод не имеет никакого эффекта — при завершении работы первичного потока процесс уничтожается вне зависимости от наличия других не фоновых потоков.
- **static void sleep(long millis)** — статический метод. Приостанавливает работу потока, в котором вызван этот метод на время в миллисекундах, указанное в параметре `millis`. Метод `sleep()` генерирует исключительную ситуацию `java.lang.InterruptedException`, которая означает, что текущий поток прерывается при помощи метода `interrupt()`.
- **void start()** — запускает поток на исполнение.

Также в классе `java.lang.Thread` объявлены методы, которые предназначены для остановки, приостановки

и возобновления работы потоков. Эти методы отменены и использовать их в разрабатываемых приложениях не получится — будет ошибка компиляции. Эти методы существуют в классе `java.lang.Thread` лишь с целью совместимости очень ранних приложений Java с новыми версиями Java виртуальных машин. Тем не менее, эти методы мы приведем здесь, так как ниже в этом уроке будут рассматриваться механизмы остановки и приостановки/возобновления работы потоков, которые (механизмы) являются заменой соответствующим отмененным методам. Итак, отмененные методы класса `java.lang.Thread` для остановки, приостановки и возобновления работы потоков:

- **final void suspend()** — метод приостанавливает работу потока, для которого он вызывается, на неопределенное время, до тех пор пока не будет вызван метод `resume()`. Метод отменен.
- **final void resume()** — возобновляет работу потока, который был ранее приостановлен при помощи вызова метода `suspend()`. Метод отменен.
- **final void stop()** — останавливает (завершает) работу потока. Метод отменен.

Эти методы отменены из-за своей не безопасности. Одной из причин, которую называют разработчики платформы Java, является то, что использование этих методов может привести к взаимной блокировке потоков (*deadlock*). Мы вернемся к проблеме безопасной остановки потоков в последующих темах этого урока.

И еще несколько методов, которые имеют прямое отношение к потокам и синхронизации потоков и кото-

ые наследуются всеми классами от базового класса `java.lang.Object`:

- **final void wait()** — Приостанавливает работу потока, в котором вызывается метод `wait()` до тех пор пока в другом потоке не будет вызван метод `notify()` или `notifyAll()`. Метод `wait()`, как правило (и это правильно), вызывается для какого-то, специально предназначенного для этого, объекта. Использование метода `wait()` будет показано в этом уроке в разделе «Приостановка и возобновление работы потоков». Метод `wait()` генерирует исключительную ситуацию `java.lang.InterruptedException`, которая означает, что текущий поток прерывается при помощи метода `interrupt()`.
- **final void wait(long millis)** — Делает то же самое что и предыдущий метод, только позволяет задать еще время таймаута в миллисекундах, которое поток сможет ждать до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`. Метод `wait()` генерирует исключительную ситуацию `java.lang.Interrupted-Exception`, которая означает, что текущий поток прерывается при помощи метода `interrupt()`.
- **final void notify()** — Возобновляет работу потока, который ранее вызвал метод `wait()`. Если есть несколько потоков, которые вызвали метод `wait()`, то будет возобновлена работа только одного из этих потоков.
- **final void notifyAll()** — Возобновляет работу всех потоков, которые ранее вызвали метод `wait()`.

⇒ **Примечание.** Методы `wait()`, `notify()`, `notifyAll()` необходимо вызывать только в синхронизируемом контексте, а именно или в методах помеченных ключевым словом `synchronized`, или в блоках `synchronized()`.

Еще раз сообщим. Что примеры использования методов `wait()` и `notify()` будут показаны в разделе «Приостановка и возобновление работы потоков».

### 1.3. Завершение (прерывание) работы вторичных потоков

Завершать или прерывать работу потока имеет смысл в тех случаях, когда поток предназначен для длительной, повторяющейся при помощи цикла, работы. Примером (не единственным) таких потоков могут быть потоки, которые обрабатывают поступающие сообщения. Сообщения могут поступать на всем протяжении работы приложения в произвольное время. Такие потоки выполняют повторяющиеся (циклические) действия — «Получил сообщение — Обработал сообщение — Вернулся к ожиданию следующего сообщения». Такие потоки, как правило, часто находятся в блокирующих исполнение методах `wait()`, `sleep()`, `join()`. Потоки, которые выполняют единичное, непродолжительное действие, обычно не прерывают, так как выполнив свою задачу, такие потоки завершаться самостоятельно.

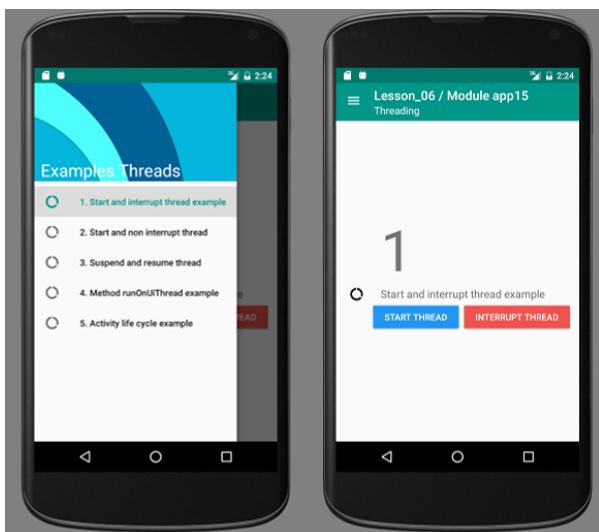
Как сообщалось в предыдущем разделе, прервать работу потока можно при помощи метода `interrupt()` класса `java.lang.Thread`. Однако, вызов метода `interrupt()` будет

эффективен лишь в том случае, когда поток готов к тому, чтобы быть прерванным. Другими словами, эффективен в том случае, когда программист написал программный код потока таким образом, чтобы вызов метода `interrupt()` привел к правильному завершению работы потока. Помните, в предыдущем разделе рассказывалось, что метод `stop()` класса `java.lang.Thread` отменен. Главной причиной такой отмены метода является то, что поток при помощи вызова метода `stop()` может быть завершен в любом месте своего исполнения, включая даже очень критические места исполнения. Например, представьте себе ситуацию, когда поток заблокировал доступ к Базе Данных чтобы монопольно выполнить операцию записи в эту Базу Данных и, не успев записать данные в БД, прерывается при помощи метода `stop()`. Разумеется, прерванный поток в нашем рассматриваемом случае не успел разблокировать доступ к Базе Данных и это привело к блокировке других потоков, которые ждут своей очереди на доступ к этой же Базе Данных. Подобных примеров в программировании достаточно много. Поэтому разработчиками платформы Java принято правильное решение — при прерывании работы потока, этому потоку дается возможность корректно завершить свою работу. В нашем примере с блокировкой Базы Данных, поток, который заблокировал доступ к Базе Данных, получив информацию о том, что его прерывают, должен снять блокировку с Базы Данных и только потом завершить свою работу. Удобно, не так ли?

Поток исполнения узнает о том что его прерывают методом `interrupt()` или с помощью обработки исключи-

тельной ситуации `java.lang.InterruptedException` или при помощи вызова метода `interrupt()`.

Давайте рассмотрим примеры корректного прерывания работы потока. С этой целью создадим модуль «app15» в проекте Android Studio, который прилагается к данному уроку. Сразу сообщим, что в модуле «app15» находится несколько примеров, посвященных вторичным потокам, поэтому приложение из модуля «app15» оснащено навигационным меню при помощи виджета `android.support.design.widget.NavigationView`. Внешний вид приложения из модуля «app15» изображен на Рис. 1.1.



**Рис. 1.1.** Внешний вид рассматриваемого приложения с примерами использования вторичных потоков

Как видно из Рис. 1.1, для рассмотрения работы конкретного примера необходимо выбрать соответствующий этому примеру пункт в навигационном меню. Для

удобства, каждый пример имеет свой порядковый номер, который присутствует в названии класса вторичного потока. Каждый из рассматриваемых примеров является отдельным примером, хотя все эти примеры находятся в одном модуле и взаимодействуют с одной и той же Активностью. Структура файла с макетом внешнего вида Активности /res/layout/activity\_main.xml показана в Листинге 1.9.

**Листинг 1.9.** Структура файла с макетом внешнего вида Активности /res/layout/activity\_main.xml рассматриваемого приложения с примерами на вторичные потоки

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"

    android:id="@+id/drawerLayout"
    tools:context="itstep.com.myapp15.MainActivity">

    <!-- Content layout -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:id="@+id/mainLL">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
```

```
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:background="@color/colorPrimary"
    />

<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/mainFL">

    <!-- Example One -->
    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/exampleOne"
        android:alpha="1"
        android:visibility="visible">
        ...
    </FrameLayout>

    <!-- Example Two -->
    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/exampleTwo"
        android:alpha="0"
        android:visibility="invisible">
        ...
    </FrameLayout>
    ...
</FrameLayout>

</LinearLayout>

<!-- NavigationView -->
<android.support.design.widget.NavigationView
    android:layout_width="wrap_content"
```

```
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/drawer_header"
        app:menu="@menu/drawer_menu"
        android:id="@+id/naviView"
    />

</android.support.v4.widget.DrawerLayout>
```

Как видно из Листинга 1.9, корневым элементом макета Активности является контейнер **android.support.v4.widget.DrawerLayout**, который предоставляет нам возможность использования выдвижного навигационного меню **android.support.design.widget.NavigationView** (см. Рис. 1.1). В макете Активности присутствует панель инструментов, которая представлена виджетом **android.support.v7.widget.Toolbar** (которому назначена иконка навигации и обработчик события клика по ней — см. Листинг 1.10) и контейнер **android.widget.FrameLayout** с идентификатором **mainFL**, который является контейнером для виджетов рассматриваемых примеров. Для каждого из рассматриваемых примеров создан свой контейнер (дочерний по отношению к **mainFL**) и каждому из этих контейнеров присвоен идентификатор, который ассоциирован с порядковым номером примера: **exampleOne**, **exampleTwo**, **exampleThree** и так далее. У всех этих контейнеров, кроме одного, установлен атрибут **android:alpha** в значение 0 (**android:alpha="0"**). При выборе пункта из навигационного меню, контейнер который соответствует выбранному пункту, получает значение

для атрибута `android:alpha="1"`, и становится видимым (отображается), а предыдущий видимый контейнер получает значение атрибута `android:alpha="0"` и становится невидимым (скрывается). Причем переход между отображающимся и скрывающимся контейнерами осуществляется плавно при помощи использования анимации. Это сделано с целью закрепления полученного вами ранее материала и с целью максимально приблизить внешний вид рассматриваемых примеров к внешнему виду реальных приложений с учетом концепции Material Design. Программный код, показывающий переключение между контейнерами для демонстраций примеров, показан в Листинге 1.10.

**Листинг 1.10.** Класс MainActivity рассматриваемого приложения

```
public class MainActivity extends AppCompatActivity
{
    // ----- Class members -----
    /**
     * A Reference to the currently visible container
     *
     * Ссылка на текущий видимый контейнер
     */
    private View curView;

    /**
     * A reference to the container that is used to
     * display Snackbar
     *
     * Ссылка на контейнер, который используется для
     * отображения Snackbar
     */
}
```

```
private LinearLayout mainLL;

/**
 * A Reference to DrawerLayout for displaying or
 * hiding navigation menu
 *
 * Ссылка на DrawerLayout для отображения или
 * скрытия навигационного меню
 */
private DrawerLayout drawerLayout;

// ----- Class methods -----
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

// ----- Toolbar -----
Toolbar toolBar = (Toolbar) this.findViewById(
    R.id.toolbar);
this.setSupportActionBar(toolBar);
toolBar.setTitle(R.string.app_name);
toolBar.setSubtitle("Threading");
toolBar.setTitleTextColor(Color.WHITE);
toolBar.setSubtitleTextColor(Color.WHITE);

// ----- Set Navigation Icon for Toolbar -----
// ----- Устанавливаем иконку навигации
// ----- и обработчик нажатия на нее -----
toolBar.setNavigationIcon(R.drawable.menu);
toolBar.setNavigationOnClickListener(
    new View.OnClickListener()
{
    @Override
    public void onClick(View v)
    {
```

```
        DrawerLayout drawerLayout =
            (DrawerLayout)
            MainActivity.this.findViewById(
                R.id.drawerLayout);
        drawerLayout.openDrawer(Gravity.LEFT);
    }
});  
  
// ----- Initializing object fields -----
// ----- Инициализация полей объекта -----
this.mainLL = (LinearLayout)
    this.findViewById(R.id.mainLL);
this.drawerLayout = (DrawerLayout)
    this.findViewById(R.id.drawerLayout);
this.curView = this.findViewById(
    R.id.exampleOne);  
  
// ----- NavigationView item select listener -----
// ----- Обработчик события выбора пункта
// ----- навигационного меню -----
final NavigationView naviView =
    (NavigationView) this.findViewById(
        R.id.naviView);
naviView.setNavigationItemSelectedListener(
    new NavigationView.
    OnNavigationItemSelectedListener()
{
    @Override
    public boolean
        onNavigationItemSelected(MenuItem item)
    {
        item.setChecked(true);
        MainActivity.this.drawerLayout.
            closeDrawers();  
  
// ----- Hide the previous visible layout -----

```

```
// ----- Прячем предыдущий видимый контейнер -----
ObjectAnimator anim =
    new ObjectAnimator();
anim.setPropertyNames("alpha");
anim.setDuration(300);
anim.setFloatValues(1.0f, 0.0f);
anim.setTarget(MainActivity.this.curView);

anim.addListener(new Animator.
    AnimatorListener()
{
    @Override
    public void onAnimationStart(
        Animator animation)
    {
    }

    @Override
    public void onAnimationEnd(
        Animator animation)
    {
        ((ViewGroup)((ObjectAnimator)
            animation)
            .getTarget())
            .setVisibility(View.
                INVISIBLE);
    }

    @Override
    public void onAnimationCancel(
        Animator animation)
    {
    }

    @Override
    public void onAnimationRepeat(
        Animator animation)
```

```
        {
        }
    });

anim.start();

// ----- Show the current layout -----
// ----- Показываем контейнер который
// ----- соответствует выбранному пункту меню
switch (item.getItemId())
{
    case R.id.navigation_item_1:
        MainActivity.this.curView =
            MainActivity.this.
            findViewById(R.
                id.exampleOne);
        break;

    case R.id.navigation_item_2:
        MainActivity.this.curView =
            MainActivity.this.
            findViewById(
                R.id.exampleTwo);
        break;

    case R.id.navigation_item_3:
        MainActivity.this.curView =
            MainActivity.this.
            findViewById(
                R.id.exampleThree);
        break;
    ...
}

MainActivity.this.curView.
    setVisibility(View.VISIBLE);
ObjectAnimator anim2 =
    new ObjectAnimator();
```

```
        anim2.setPropertyNames("alpha");
        anim2.setDuration(300);
        anim2.setStartDelay(300);
        anim2.setFloatValues(0.0f, 1.0f);
        anim2.setTarget(MainActivity.this.
                curView);
        anim2.start();
        return true;
    }
});
```

Итак, вернемся к первому примеру в котором будем рассматривать запуск и прерывание работы потока. Для демонстрации работы этого примера необходимо выбрать в навигационной панели пункт меню с надписью «1. Start and interrupt thread example» (см. Рис. 1.1). В Контейнере с идентификатором **exampleOne** созданы две кнопки **android.widget.Button** — одна для запуска потока (идентификатор кнопки **R.id.btnStart**), другая для его прерывания (идентификатор **R.id.btnInterrupt**). Класс потока для этого примера называется **ThrExampleOne** и его содержимое показано в Листинге 1.11.

### **Листинг 1.11.** Класс потока ThrExampleOne из первого примера «Запуск и прерывание работы потока»

```
class ThrExampleOne extends Thread
{
    private final static String THR_TAG = "ThrExampleOne";

    /**

```

```
* The message that will output the thread
*
* Сообщение, которое будет выводить поток
*/
private String msg;

public ThrExampleOne(String msg)
{
    this.msg = msg;
}

@Override
public void run()
{
    try
    {
        int cnt = 0;
        while (true)
        {
            Thread.sleep(750);
            cnt++;
            String str = this.msg + " " + cnt + "(" +
                Thread.currentThread().getName() +
                ")";
            Log.d(THR_TAG, str);
        }
    }

    catch (InterruptedException ie)
    {
        Log.d(THR_TAG, "Thread interrupted: " +
            Thread.currentThread().getName());
    }
}
}
```

При клике на кнопку `R.id.btnStart` будет происходить запуск потока `ThrExampleOne`, а при клике на кнопку `R.id.btnInterrupt` будет происходить прерывание работы потока при помощи вызова метода класса `java.lang.Thread`:

```
void interrupt();
```

Чтобы исключить нежелательное создание нескольких потоков `ThrExampleOne` (например после поворота устройства), ссылка на объект `ThrExampleOne` объявлена в виде статического поля класса `MainActivity`, как показано в Листинге 1.7. Другими словами, используется описанное выше решение, избавляющее от создания ненужных экземпляров потока. Обработка событий нажатия на кнопки `R.id.btnStart` и `R.id.btnInterrupt`, а также запуск и прерывание потока `ThrExampleThread` осуществляется в методе `btnExampleOneClick()` класса Активности `MainActivity`. Код метода `btnExampleOneClick()` приведен в Листинге 1.12.

**Листинг 1.12.** Программный код обработчика событий нажатия на кнопку запуска и прерывания работы потока

```
public class MainActivity extends AppCompatActivity
{
    ...
    private TextView tvInfo1;

    private static ThrExampleOne T1 = null;

    // ----- Class methods -----
}
```

```
public void btnExampleOneClick(View v)
{
    if (MainActivity.T1 != null && MainActivity.T1.isAlive() == false)
    {
        MainActivity.T1 = null;
    }

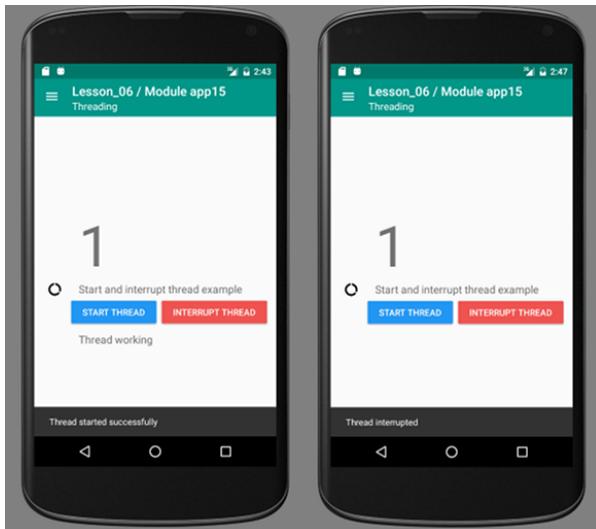
    switch (v.getId())
    {
// ----- Start ThrExampleOne thread -----
        case R.id.btnStart :
        {
            if (MainActivity.T1 == null)
            {
                MainActivity.T1 = new ThrExampleOne(
                    "Android forever!");
                MainActivity.T1.start();

                Snackbar.make(mainLL,
                    "Thread started successfully",
                    Snackbar.LENGTH_LONG).show();
                this.tvInfo1.setText("Thread working");
            }
        }
        else
        {
            Snackbar.make(mainLL,
                "Thread already running",
                Snackbar.LENGTH_INDEFINITE)
                .setAction("OK", new View.
                OnClickListener()
                {
                    @Override
                    public void onClick(View v)
                    {
                    }
                })
        }
    }
}
```

```
        .show();
    }
}
break;

// ----- Interrupt ThrExampleOne thread -----
case R.id.btnInterrupt :
{
    if (MainActivity.T1 != null)
    {
        MainActivity.T1.interrupt();
        Snackbar.make(mainLL,
                      "Thread interrupted",
                      Snackbar.LENGTH_LONG).show();
        this.tvInfol.setText("");
    }
    else
    {
        Snackbar.make(mainLL,
                      "No thread to interrupt",
                      Snackbar.LENGTH_INDEFINITE)
                      .setAction("OK", new View.
                        OnClickListener()
{
            @Override
            public void onClick(View v)
            {
            }
        })
        .show();
    }
}
break;
}
}
```

Внешний вид работы примера из Листинга 1.11 и 1.12 изображен на Рис. 1.2.



**Рис. 1.2.** Внешний вид работы примера из Листинга 1.11 и 1.12

При запуске потока (нажатие на кнопку `R.id.btnStart`) появляется **Snackbar** с сообщением об успешном запуске потока и в текстовом поле `android.widget.TextView` с идентификатором `R.id.tvInfo1` появляется сообщение «Thread working». При остановке работы потока (нажатие на кнопку `R.id.btnInterrupt`) появляется **Snackbar** с сообщением об успешной остановке потока и текстовое поле `R.id.tvInfo1` очищается. Пример защищен от повторного запуска уже запущенного потока или от повторного прерывания уже прерванного потока. При попытке это сделать появится **Snackbar** с сообщением об ошибке (см. Листинг 1.12).

Во время работы потока осуществляется вывод этим потоком сообщений в логи приложения, как показано в Листинге 1.13.

**Листинг 1.13.** Вывод сообщений о работе и прерывании работы потока из Листинга 1.11

```
28135-29334/itstep.com.myapp15
D/ThrExampleOne: Android forever! 83(Thread-217)
28135-29334/itstep.com.myapp15
D/ThrExampleOne: Android forever! 84(Thread-217)
28135-29334/itstep.com.myapp15
D/ThrExampleOne: Android forever! 85(Thread-217)
28135-29334/itstep.com.myapp15
D/ThrExampleOne: Thread interrupted : Thread-217
```

Когда происходит прерывание работы потока при помощи вызова метода **interrupt()** в Листинге 1.12, метод **sleep()** который вызывается в методе **run()** потока **ThrExampleOne** (см. Листинг 1.11), генерирует исключительную ситуацию **java.lang.InterruptedIOException**. Поток **ThrExampleOne** в методе **run()** обрабатывает эту исключительную ситуацию, и завершает свою работу путем выхода из метода **run()**. Напомним, что метод **run()** является главной функцией вторичного потока. Вторичный поток начинает свою работу с метода **run()**, и завершает свою работу при выходе из метода **run()**. Так вот, получив исключительную ситуацию **java.lang.InterruptedIOException**, поток понимает, что ему необходимо как можно быстрее закончить свою работу и в обработчике **catch** этой исключительной ситуации выводит сообщение «**Thread interrupted**» и покидает метод **run()**.

Вторичный поток может, при необходимости, проигнорировать попытку прерывания своей работы и продолжить свою работу (то есть не покинуть метод `run()`). Это делается в случаях, когда работа прерывание работы потока в какой-то момент времени недопустимо (например поток не закончил какую-то важную операцию, которую необходимо завершить до того как поток сможет прервать свою работу). Давайте рассмотрим, как это делается. Для этого создан следующий пример в модуле «app15». Для демонстрации работы этого примера необходимо выбрать в навигационной панели пункт меню с надписью «2. Start and non stop thread example». Внешний вид набора виджетов (контейнер `R.id.exampleTwo`) для демонстрации работы этого примера такой же как и для примера «1. Start and interrupt thread example». В контейнере `R.id.exampleTwo` присутствуют две кнопки. Первая кнопка с надписью «Start thread» (идентификатор `R.id.btnStartExampleTwo`) предназначена для запуска потока. Вторая кнопка с надписью «Interrupt thread» (идентификатор `R.id.btnInterruptExampleTwo`) предназначена для прерывания работы потока. Этим кнопкам назначен обработчик события нажатия `btnExampleTwoClick()`. Содержимое этого метода практически ничем не отличается от метода обработчика события нажатия `btnExampleOneClick()` предыдущего примера (см. Листинг 1.12) и поэтому полностью не показывается в листингах данного урока, но прилагается к уроку в модуле «app15». В данном уроке мы покажем лишь не-

большой фрагмент метода **btnExampleTwoClick()**, который относится к обработке события нажатия на кнопку «Interrupt thread» («Прервать поток»). Этот фрагмент метода **btnExampleTwoClick()** показан в Листинге 1.14.

**Листинг 1.14.** Обработка события нажатия на кнопку «Interrupt thread» («Прервать поток») в методе **btnExampleTwoClick()** класса **MainActivity**.

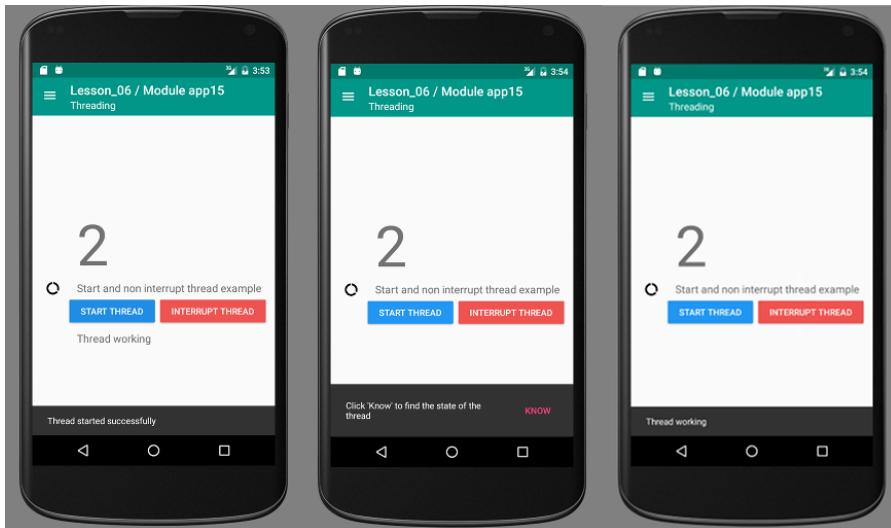
```
public void btnExampleTwoClick(View v)
{
    ...
    switch (v.getId())
    {
        // ----- Interrupt ThrExampleTwo thread -----
        case R.id.btnInterruptExampleTwo:
        {
            if (MainActivity.T2 != null)
            {
                MainActivity.T2.interrupt();
                Snackbar.make(mainLL,
                    "Click 'Know' to find the state of the thread",
                    Snackbar.LENGTH_INDEFINITE)
                    .setAction("Know", new View.OnClickListener()
                {
                    @Override
                    public void onClick(View v)
                    {
                        Snackbar.make(mainLL,
                            ((MainActivity.T2 != null &&
                                MainActivity.T2.isAlive()) ?
                            "Thread working":
                            "Thread interrupted"),
                            Snackbar.LENGTH_LONG).show();
                    }
                });
            }
        }
    }
}
```

```
        }
    })
    .show();
this.tvInfo2.setText("");
}
else
{
    Snackbar.make(mainLL,
        "No thread to interrupt",
        Snackbar.LENGTH_INDEFINITE)
        .setAction("OK",
        new View.OnClickListener()
    {
        @Override

            public void onClick(View v)
        {
        }

    })
    .show();
}
break;
}
}
```

Как видно из Листинга 1.14, после нажатия на кнопку «Interrupt Thread» (идентификатор **R.id.btnInterruptExampleTwo**) появляется Snackbar с предложением нажать на кнопку «Know» («Узнать») чтобы узнать прерван ли поток. Поток оказывается не прерван, как показано на Рис. 1.3. Это происходит благодаря программному коду класса потока **ThrExampleTwo**.



**Рис. 1.3.** Внешний вид работы примера «2. Start and non stop thread example»

Код класса потока потока **ThrExampleTwo** (для примера «2. Start and non stop thread example») показан в Листинге 1.15.

**Листинг 1.15.** Программный код класса потока ThrExampleTwo для демонстрации игнорирования попытки прерывания работы потока

```
/*
 * Class ThrExampleTwo - Thread for Example Two
 */

class ThrExampleTwo extends Thread
{
    private final static String THR_TAG = "ThrExampleTwo";

    /**
     * The message that will output the thread

```

```
*  
* Сообщение, которое будет выводить поток  
*/  
private String msg;  
  
public ThrExampleTwo(String msg)  
{  
    this.msg = msg;  
}  
  
@Override  
public void run()  
{  
    int cnt = 0;  
    while (true)  
    {  
        try  
        {  
            Thread.sleep(750);  
        }  
        catch (InterruptedException ie)  
        {  
            Log.d(THR_TAG,  
                  "attempt to interrupt the  
                  thread (ignored) : " +  
                  Thread.currentThread().getName());  
        }  
  
        cnt++;  
        String str = this.msg + " " + cnt +  
                    "(" + Thread.currentThread().  
                    getName() + ")";  
        Log.d(THR_TAG, str);  
    }  
}
```

Как видно из Листинга 1.15, поток **ThrExampleTwo** не прерывается потому, что обработка исключительной ситуации **java.lang.InterruptedException** осуществляется внутри главного цикла потока. Обработав исключительную ситуацию **java.lang.InterruptedException**, поток не выходит из главного цикла, и следовательно, не выходит из метода **run()**. При этом поток **ThrExampleTwo** выводит в логи приложения сообщение о том, что была предпринята попытка прервать работу потока и поток эту попытку проигнорировал («attempt to interrupt the thread (ignored)»). Сообщения, которые поток **ThrExampleTwo** выводит в логи приложения, показаны в Листинге 1.16.

**Листинг 1.16.** Вывод потоком **ThrExampleTwo** сообщений в логи приложения в том числе и во время попытки прерывания его работы

```
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    Example Two Message 2109(Thread-254)  
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    Example Two Message 2110(Thread-254)  
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    Example Two Message 2111(Thread-254)  
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    attempt to interrupt the thread (ignored) :  
    Thread-254  
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    Example Two Message 2112(Thread-254)  
18789/itstep.com.myapp15 D/ThrExampleTwo:  
    Example Two Message 2113(Thread-254)
```

У вас может возникнуть вопрос — как можно прервать работу потока из Листинга 1.15, если поток **ThrExampleTwo** игнорирует посылаемые ему исключи-

тельные ситуации `java.lang.InterruptedException`? Одно из несложных решений показано в Листинге 1.17.

**Листинг 1.17.** Способ завершения работы потока без использования исключительной ситуации `java.lang.InterruptedException`

```
class SomeThread extends Thread
{
    private boolean isRun = true;

    @Override
    public void run()
    {
        while (this.isRun)
        {
            ...
        }
    }

    public void completeThread()
    {
        this.isRun = false;
    }
}
```

Исходный код примеров из данного раздела находится в модуле «app15» среди файлов исходных кодов которые прилагаются к данному уроку.

## 1.4. Взаимодействие вторичных потоков с виджетами. Метод `runOnUiThread`

В предыдущих примерах, посвященных вторичным потокам, для демонстрации работы потоков осуществлялся вывод сообщений от этих потоков в логи прило-

жения при помощи методов класса `android.util.Log`. Однако, в некоторых случаях было бы удобно осуществлять вывод сообщений не в логи приложения, а в виджеты Активности. Или, речь может идти совсем не о выводе сообщений вторичных потоков, о об их взаимодействии с виджетами Активности. Особеностям взаимодействия вторичных потоков с виджетами Активности и посвящен данный раздел.

Предположим, необходимо написать класс вторичного потока, который будет выводить сообщения в виджет `android.widget.TextView` расположенный в Активности. Для этого случая создан пример «3. Method runOnUiThread example» в модуле «app15». Контейнер в макете Активности для этого примера имеет идентификатор `exampleThree`.

В этом контейнере, аналогично предыдущим примерам, созданы две кнопки: кнопка «Start thread» (идентификатор `R.id.btnStartExampleThree`) и кнопка «Interrupt thread» (идентификатор `R.id.btnInterruptExampleThree`). Обработчиком событий нажатия на эти кнопки является метод `btnExampleThreeClick()`, который аналогичен методу `btnExampleOneClick()` из Листинга 1.12 и поэтому не показан в листингах данного урока (программный код этого метода можно найти в модуле «app15»). Внешний вид примера «3. Method runOnUiThread example» изображен на Рис. 1.4.

Класс вторичного потока для примера «3. Method runOnUiThread example» называется `ThrExampleThree` и код этого класса показан в Листинге 1.18.

**Листинг 1.18.** Класс потока ThrExampleThree с попыткой отображения в виджете android.TextView информации

```
/***
 * Class ThrExampleThree - Thread for Example Three
 */
class ThrExampleThree extends Thread
{
    private final static String THR_TAG = "ThrExampleThree";

    /**
     * The message that will output the thread
     *
     * Сообщение, которое будет выводить поток
     */
    private String msg;

    /**
     * Widget TextView where the thread will output
     * the message
     *
     * Виджет TextView куда поток будет выводить
     * сообщение
     */
    private TextView tvInfo;

    public ThrExampleThree(String msg, TextView tvInfo)
    {
        this.msg = msg;
        this.tvInfo = tvInfo;
    }

    @Override
    public void run()
    {
        try
        {
```

```
int cnt = 0;
while (true)
{
    Thread.sleep(750);
    cnt++;
    String str = this.msg + " " + cnt +
                 "(" + Thread.currentThread().
                 getName() + ")";
    Log.d(THR_TAG, str);
    this.tvInfo.setText(str);
}
}
catch (InterruptedException ie)
{
    Log.d(THR_TAG, "Thread interrupted : " +
        Thread.currentThread().getName());
}
}

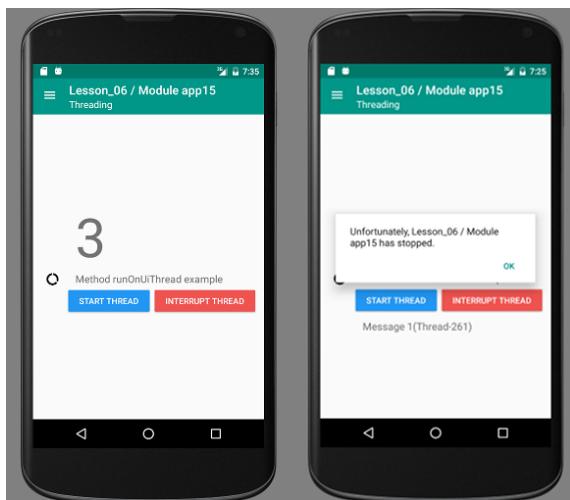
public void setTextView(TextView tvInfo)
{
    this.tvInfo = tvInfo;
}
}
```

Из Листинга 1.18 видно, объект **ThrExampleThree** получает при создании не только текст сообщения (параметр конструктора **msg**), но и ссылку на виджет **android.widget.TextView** (параметр конструктора **tvInfo**) в который поток **ThrExampleThree** должен периодически выводить сообщение **msg**. Вывод сообщения в текстовое поле **tvInfo** выделен в Листинге 1.18 жирным шрифтом. Однако, попытка вывода сообщения в текстовое поле **tvInfo** приводит к исключительной ситуации и остановке работы программы, как показано на Рис. 1.4. Исключительная ситуация, кото-

рая возникает в данном примере, называется **CalledFromWrongThreadException** и текст ее сообщения выглядит так:

```
CalledFromWrongThreadException: Only the original thread  
that created a view hierarchy can touch its views.
```

То есть, эта исключительная ситуация, при попытке взаимодействия вторичного потока с виджетом, который создан в первичном потоке, сообщает нам, что с виджетом может взаимодействовать только тот поток, который создал этот виджет. В нашем случае с текстовым полем **android.widget.TextView**, который создан в первичном потоке, может взаимодействовать (устанавливать текст в этот виджет) только первичный поток.



**Рис. 1.4.** Внешний вид работы примера «3. Method runOnUiThread example» с сообщением об ошибке при попытке взаимодействия вторичного потока ThrExampleThree с текстовым полем android.widget.TextView

Решением данной проблемы является использование специального метода, который объявлен в классе Активности:

```
public final void runOnUiThread (Runnable action);
```

Этот метод запускает метод **run()** из объекта **action** в потоке пользовательского интерфейса (в первичном потоке). Если текущий поток, из которого вызывается метод **runOnUiThread()**, является потоком пользовательского интерфейса, то **action** выполняется (запускается) немедленно. Если текущий поток не является потоком пользовательского интерфейса, **action** отправляется в очередь событий потока пользовательского интерфейса.

Модифицируем код класса потока **ThrExampleThree** из Листинга 1.18 таким образом, чтобы поток при записи текста в виджет **android.widget.TextView** (который находится Активности) использовал метод **runOnUiThread()**. Измененный код класса **ThrExampleThree** показан в Листинге 1.19. Изменения в классе **ThrExampleThree** в Листинге 1.19 выделены жирным шрифтом.

**Листинг 1.19.** Модифицированный код класса потока **ThrExampleThree** для использования метода **runOnUiThread**

```
/**  
 * Class ThrExampleThree - Thread for Example Three  
 */  
class ThrExampleThree extends Thread  
{  
    private final static String THR_TAG = "ThrExampleThree";  
    /**
```

```
* The message that will output the thread
*
* Сообщение, которое будет выводить поток
*/
private String msg;

/***
 * Widget TextView where the thread will output
 * the message
*
* Виджет TextView куда поток будет выводить сообщение
*/
private TextView tvInfo;
private Activity activity;

public ThrExampleThree(String msg,
                      TextView tvInfo, Activity activity)
{
    this.msg = msg;
    this.tvInfo = tvInfo;
    this.activity = activity;
}

@Override
public void run()
{
    try
    {
        int cnt = 0;
        while (true)
        {
            Thread.sleep(750);
            cnt++;
            final String str = this.msg + " " +
                cnt + "(" +
                Thread.currentThread().getName() + ")";
            Log.d(THR_TAG, str);
    }
}
```

```
        this.activity.runOnUiThread(new Runnable()
    {
        @Override
        public void run()
        {
            ThrExampleThree.this.tvInfo.
                setText(str);
        }
    });
}

catch (InterruptedException ie)
{
    Log.d(THR_TAG, "Thread interrupted : " +
        Thread.currentThread().getName());
}
}

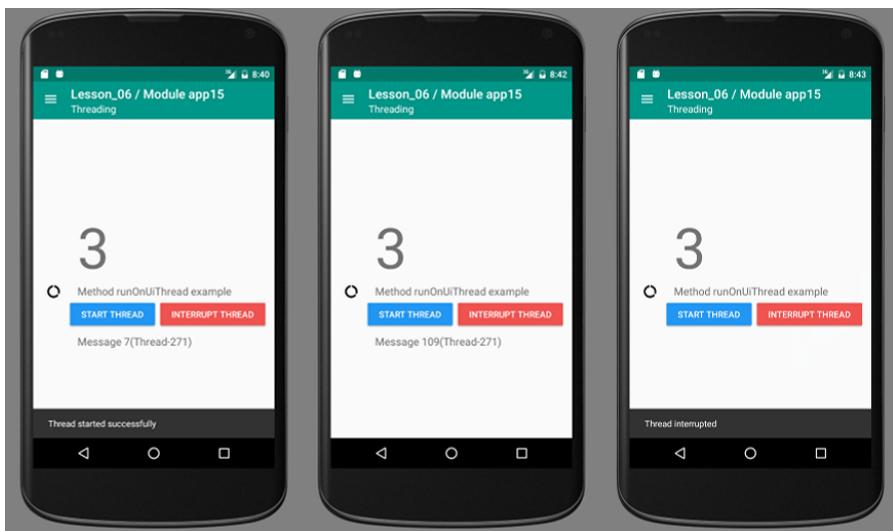
public void setTextView(TextView tvInfo,
                      Activity activity)
{
    this.tvInfo = tvInfo;
    this.activity = activity;
}
```

Как видно из Листинг 1.19, в класс **ThrExampleThree** пришлось добавить ссылку на объект Активности. Это наводит на мысль, что лучше в таком случае использовать класс **ThrExampleThree** как вложенный класс в класс Активности **MainActivity**. В этом случае запуск метода **runOnUiThread** выглядел бы так, как показано в Листинге 1.20 и в классе **ThrExampleThree** не нужно было бы

создавать поле **activity** — так как вложенный класс Java автоматически получает доступ к любым полям и методам объемлющего класса (см. Листинг 1.20).

**Листинг 1.20.** Пример запуска метода runOnUiThread в случае, если бы класс ThrExampleThree был бы вложенным классом в класс Активности MainActivity

```
MainActivity.this.runOnUiThread(new Runnable()
{
    @Override
    public void run()
    {
        ThrExampleThree.this.tvInfo.setText(str);
    }
});
```



**Рис. 1.5.** Внешний вид работы примера «3. Method runOnUiThread example» с модифицированным кодом класса потока ThrExampleThree

Внешний вид работы примера «3. Method runOnUiThread example» с модифицированным кодом класса потока **ThrExampleThree** изображен на Рис. 1.5.

Как видно из Рис. 1.5, вторичный поток **ThrExampleThree** успешно отображает свои сообщения в виджет **android.widget.TextView** принадлежащий первичному потоку.

Также хотим рассказать вам о том, что метод **runOnUiThread()** не является единственным методом, который позволяет взаимодействовать вторичным потокам с виджетами которые принадлежат первичному потоку. В классе **android.view.View** есть методы:

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

которые предоставляют аналогичные возможности, что и метод **runOnUiThread()**. Причем метод **postDelayed()** позволяет еще и указать задержку в миллисекундах перед исполнением метода **run()** объекта **action**.

Далее, вы наверняка обратили внимание, что в коде класса **ThrExampleThree** (см. Листинги 1.18 и 1.19) есть метод **setTextView()**. Этот метод играет важную роль. Дело в том, что после поворота устройства поток **ThrExampleThree** продолжит свое существование, так как создается в единственном экземпляре и ссылка на него объявлена в виде статического поля класса **MainActivity**, как показано в Листинге 1.21. Почему так сделано мы уже объясняли в данном уроке (см. пояснения к Листингу 1.7).

**Листинг 1.21.** Объявление ссылки на поток ThrExampleThree в классе Активности

```
public class MainActivity extends AppCompatActivity
{
    ...
    private static ThrExampleOne T1 = null;
    private static ThrExampleTwo T2 = null;
private static ThrExampleThree T3 = null;
    ...
}
```

Так вот, как мы уже сказали, поток **ThrExampleThree** после поворота устройства продолжит существование, а объекты Активности и принадлежащее ему текстовое поле **android.widget.TextView** (с которым взаимодействует поток) будут пересозданы. Следовательно, после пересоздания этих объектов необходимо передать ссылки на эти объекты потоку **ThrExampleThree** чтобы он корректно продолжил свою работу после поворота устройства. В примере модуля «app15» это делается в методе **onCreate()** класса Активности (см. Листинг 1.22).

**Листинг 1.22.** Установка ссылок на Активность и текстовое поле для потока ThrExampleThree после поворота устройства в методе **onCreate()**

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    ...
    // ----- Set the references for Activity and
    // ----- TextView for ThrExampleThree thread -----
    // ----- Установим ссылку на Activity и TextView
    // ----- для потока ThrExampleThree -----
}
```

```
    if (MainActivity.T3 != null)
    {
        MainActivity.T3.setTextView(this.tvInfo3, this);
    }
}
```

Теперь, с учетом Листинга 1.22, после поворота устройства поток **ThrExampleThree** продолжит корректно взаимодействовать с виджетом **android.widget.TextView** который принадлежит Активности.

Итак, подведем итоги данного раздела:

- Вторичный поток может взаимодействовать с виджетами из первичного потока при помощи метода класса Активности **runOnUiThread()** (или аналогичных ему методов)
- После поворота устройства вторичному потоку нужно передать новые ссылки на виджеты с которыми он взаимодействует.

Исходный код примеров из данного раздела находится в модуле «app15» среди файлов исходных кодов которые прилагаются к данному уроку.

## 1.5. Приостановка и возобновление работы потоков

В предыдущем разделе мы рассмотрели каким образом можно прерывать работу вторичных потоков. Но достаточно часто случаются ситуации, когда поток не нужно прервать навсегда, а нужно всего лишь приостановить его работу на некоторое (неопределенное) время. Как уже рассказывалось в данном уроке, в классе **java.lang.Thread**

были методы `suspend()` и `resume()` которые предназначались для приостановки на неопределенное время и возобновления работы потока соответственно. Эти методы уже отменены. И причиной, по которой эти методы отменены является то, что небезопасно останавливать потоки в любой точке их исполнения (а именно это и делает метод `suspend()`). Поясним это на примере. Предположим, что метод `suspend()` не отменен и что у нас работает поток «Пекарь», задача которого месить тесто, закладывать это тесто в духовой шкаф для выпечки. Причем эта работа повторяется потоком «Пекарь» циклически. Рассмотрим эту работу более подробно и с разделением на шаги исполнения. Для этого мы условно реализовали метод `run()` для потока «Пекарь» (**Baker**). Код условного класса «Пекарь» показан в Листинг 1.23.

#### Листинг 1.23. Код условного класса «Пекарь» (**Baker**)

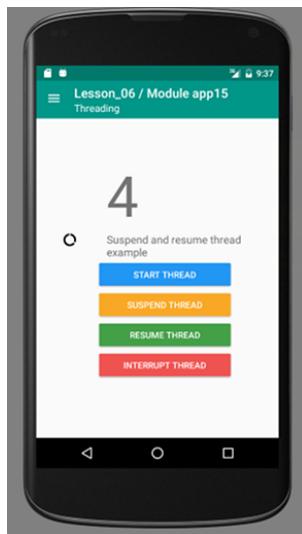
```
class Baker extends Thread
{
    private boolean isRun = true;

    @Override
    public void run()
    {
        while (this.isRun)
        {
            1. Пекарь месит тесто.
            2. Пекарь открывает дверь духового шкафа.
            3. Пекарь открывает вентиль газа духового
               шкафа.
            4. Пекарь зажигает спичку и поджигает газ
               в духовом шкафу.
        }
    }
}
```

```
    5. Пекарь кладет тесто в духовой шкаф.  
    6. Пекарь закрывает дверь духового шкафа.  
    Выпечка началась.  
}  
}  
  
public void stopBakerThread()  
{  
    this.isRun = false;  
}  
}
```

Теперь, с учетом работы потока «Пекарь» из Листинга 1.23, представим себе такую ситуацию: поток «Пекарь» остановлен при помощи метода **suspend()**, между шагами 3 и 4. То есть «Пекарь» уже успел включить газ, но не успел его зажечь потому что был приостановлен. И через неизвестное время — оно может быть очень длительным — поток возобновит свою работу после вызова для него метода **resume()**. Что при этом произойдет — когда кухня уже наполнена газом и «Пекарь» зажигает спичку — вы можете себе представить. Именно поэтому методы **suspend()** и **resume()** в классе **java.lang.Thread** отменены. Что же предлагают нам взамен разработчики Java? Нам предлагают взамен самостоятельно реализовывать механизмы остановки и возобновления работы наших потоков так как именно мы (создатели наших классов потоков) знаем в каком месте наш поток может быть безопасно остановлен. Именно это мы и будем показывать вам в данном разделе урока. Пример данного раздела доступен через пункт навигационного меню «4. Suspend

and resume thread example» («Пример приостановки и запуска работы потока»).



**Рис. 1.6.** Внешний вид примера  
«4. Suspend and resume thread example»

Контейнер для примера «4. Suspend and resume thread example» имеет идентификатор `exampleFour`. Внешний вид набора виджетов, входящий в этот контейнер `exampleFour`, изображен на Рис. 1.6. Класс потока для этого примера называется `ThrExampleFour` (см. Листинг 1.24) и в рассматриваемом примере этот класс будет «Пекарем». Поскольку создание функциональности по остановке и возобновлению работы потока является задачей разработчика, то для случая с классом потока «Пекарь» необходимо определиться, в каких местах работы потока можно будет его безопасно приостанавливать. Опираясь на Листинг 1.23, определим, что безопасными

местами для приостановки работы потока «Пекарь» являются: между первым и вторым шагом, между вторым и третьим, между четвертым и пятым и между шестым и первым. Не безопасными местами являются места: между третьим (включен газ) и четвертым (зажигается спичка) — потому что возможен взрыв, между пятым (кладется тесто в шкаф) и шестым (закрывается дверь духового шкафа) — потому что приведет к порче продукта из-за неравномерного прогрева теста. Перечисленные выше безопасные места и будем использовать для возможности приостановки в них работы потока «Пекарь» (мы вернемся чуть позже к обсуждению приостановки потока при анализе класса **ThrExampleFour** из Листинга 1.24, который является прототипом потока «Пекарь» в рассматриваемом примере).

Как видно из Рис. 1.6, в примере «4. Suspend and resume thread example» есть четыре кнопки: кнопка «Start thread» (идентификатор **R.id.btnStartExampleFour**) которая предназначена для запуска потока **ThrExampleFour**, кнопка «Suspend thread» (идентификатор **R.id.btnSuspendExampleFour**) которая предназначена для приостановки потока **ThrExampleFour**, кнопка «Resume thread» (**R.id.btnResumeExampleFour**) которая предназначена для возобновления работы потока **ThrExampleFour**, кнопка «Interrupt thread» (**R.id.btnInterruptExampleFour**) которая предназначена для прерывания работы потока **ThrExampleFour**. Этим кнопка назначен метод обработчик события **btnExampleFourClick()**, содержимое которого приведено в Листинге 1.25.

**Листинг 1.24.** Содержимое класса потока ThrExampleFour для примера «4. Suspend and resume thread example»

```
public class MainActivity extends AppCompatActivity
{
    // ----- Inner classes -----
    class ThrExampleFour extends Thread
    {
        private final static String THR_TAG =
                "ThrExampleFour";

        /**
         * Text field for displaying information messages
         * from this thread
         * Текстовое поле для вывода информационных
         * сообщений из этого потока
         */
        private TextView tvInfo;

        /**
         * Indicator of a thread suspend
         * Признак приостановки работы потока
         */
        private boolean isPause = false;

        public ThrExampleFour(TextView tvInfo)
        {
            this.tvInfo = tvInfo;
        }

        @Override
        public void run()
        {
            try
            {
                int cnt = 0;
                while(true)
                {
                    cnt++;
                    if (isPause)
                        Thread.sleep(100);
                    else
                        tvInfo.setText("Count = " + cnt);
                }
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

```
// ----- 1. Step -----
    this.putMessage(
        "1. Baker \nkneads the dough. (" +
        cnt + ")");
    Thread.sleep(2000); // The duration of
                      // this step
    this.checkPause(); // May be thread
                      // suspend?

// ----- 2. Step -----
    this.putMessage(
        "2. The baker opens the oven door. (" +
        cnt + ")");
    Thread.sleep(2000); // The duration
                      // of this step
    this.checkPause(); // May be thread
                      // suspend?

// ----- 3. Step -----
    this.putMessage(
        "3. The baker opens the gas valve
          of the oven. (" +
        cnt + ")");
    Thread.sleep(2000); // The duration
                      // of this step
                      // Suspension
                      // is forbidden here

// ----- 4. Step -----
    this.putMessage(
        "4. Baker lights a match and sets fire" +
        "to the gas in the oven. (" + cnt + ")");
    Thread.sleep(2000); // The duration
                      // of this step
    this.checkPause(); // May be thread
                      // suspend?
```

```

// ----- 5. Step -----
        this.putMessage(
            "5. Baker puts the dough in the oven. (" +
            cnt + ")");
        Thread.sleep(2000); // The duration
                            // of this step
                            // Suspension is
                            // forbidden here

// ----- 6. Step -----
        this.putMessage(
            "6. The baker closes the oven door." +
            "Baking started. (" + cnt + ")");
        Thread.sleep(2000); // The duration
                            // of this step
        this.checkPause(); // May be thread
                            // suspend?
    }
}

catch (InterruptedException ie)
{
    this.putMessage("Thread Interrupted : " +
        ie.getMessage());
}
}

/***
 * The method suspends the current thread
 * Метод приостанавливает работу текущего потока
 */
public synchronized void suspendWork()
{
    this.isPause = true;
}

/***
 * The method resumes the current thread
*/

```

```
* Метод возобновляет работу текущего потока
*/
public synchronized void resumeWork()
{
    this.isPause = false;

// ----- Tell the current thread that need to wake up
// ----- Сообщаем текущему потоку что необходимо
// ----- проснуться -----
    this.notify();
}

private synchronized void checkPause() throws
    InterruptedException
{
    while (this.isPause)
    {
        Log.d(THR_TAG, "Thread Suspended");

// ----- Suspend the current thread -----
// ----- Текущий поток останавливается -----
        this.wait();
    }
}

private void putMessage(final String msg)
{
    Log.d(THR_TAG, msg);
    MainActivity.this.runOnUiThread(new Runnable()
    {
        @Override
        public void run()
        {
            ThrExampleFour.this.tvInfo.setText(msg);
        }
    });
}
```

```
public void setTextView(TextView tvInfo)
{
    this.tvInfo = tvInfo;
}
...
}
```

Как видно из Листинга 1.24, класс **ThrExampleFour** является вложенным классом в класс Активности **MainActivity**. Это сделано для удобства взаимодействия потока **ThrExampleFour** с виджетом **android.widget.TextView** в который поток выводит свои сообщения.

Давайте проанализируем код класса **ThrExampleFour**. Для приостановки работы потока в классе создан метод **suspendWork()**, который устанавливает значение специального поля **isPause** в значение **true**. Этот метод предназначен для вызова из другого потока. Для возобновления работы потока в классе **ThrExampleFour** создан метод **resumeWork()**, который также предназначен для вызова из другого потока. В методе **run()** класса **ThrExampleFour** в местах, где можно безопасно приостановить поток, вызывается специальный метод этого же класса **checkPause()**. В этом методе проверяется значение поля **isPause**, и если оно равно значению **true**, то поток **ThrExampleFour** останавливается при помощи вызова метода **wait()**. Поток будет находиться в остановленном состоянии до тех пор, пока из другого потока не будет вызван метод **resumeWork()**. В методе **resumeWork()** устанавливается значение поля **isPause** равным **false** и вызы-

вается метод **notify()** для пробуждения работы потока. Методы **wait()** и **notify()** описывались ранее в этом уроке. Также мы настоятельно рекомендуем вспомнить тему «Синхронизация потоков» из курса Java.

Запуск потока **ThrExampleFour**, прерывание, приостановка и возобновление работы этого потока осуществляется в методе обработчика событий нажатия на кнопки (см. Рис. 1.6) **btnExamplefourClick()**. Содержимое этого метода показано в Листинге 1.25.

**Листинг 1.25.** Метод обработчик событий нажатия на кнопки **btnExampleFourClick()** из примера «4. Suspend and resume thread example»

```
public void btnExampleFourClick(View v)
{
    if (MainActivity.T4 != null && MainActivity.T4.isAlive() == false)
    {
        MainActivity.T4 = null;
    }

    switch (v.getId())
    {
// ----- Start ThrExampleFour thread -----
        case R.id.btnStartExampleFour:
        {
            if (MainActivity.T4 == null)
            {
                MainActivity.T4 =
                    new ThrExampleFour(this.tvInfo4);
                MainActivity.T4.start();
                Snackbar.make(mainLL,
                            "Thread started successfully",
                            Snackbar.LENGTH_LONG).show();
            }
        }
    }
}
```

```
        }
    else
    {
        Snackbar.make(mainLL,
            "Thread already running",
            Snackbar.LENGTH_INDEFINITE)
            .setAction("OK", new View.
            OnClickListener()
            {
                @Override
                public void onClick(View v)
                {
                }
            })
            .show();
    }
}
break;

// ----- Interrupt ThrExampleFour thread -----
case R.id.btnExitExampleFour:
{
    if (MainActivity.T4 != null)
    {
        MainActivity.T4.interrupt();

        Snackbar.make(mainLL, "Thread interrupted",
            Snackbar.LENGTH_LONG).show();
        this.tvInfo1.setText("");
    }
}
else
{
    Snackbar.make(mainLL,
        "No thread to interrupt",
        Snackbar.LENGTH_INDEFINITE)
        .setAction("OK", new View.
        OnClickListener()
```

```
        {
            @Override
            public void onClick(View v)
            {
            }
        })
        .show();
    }
}
break;

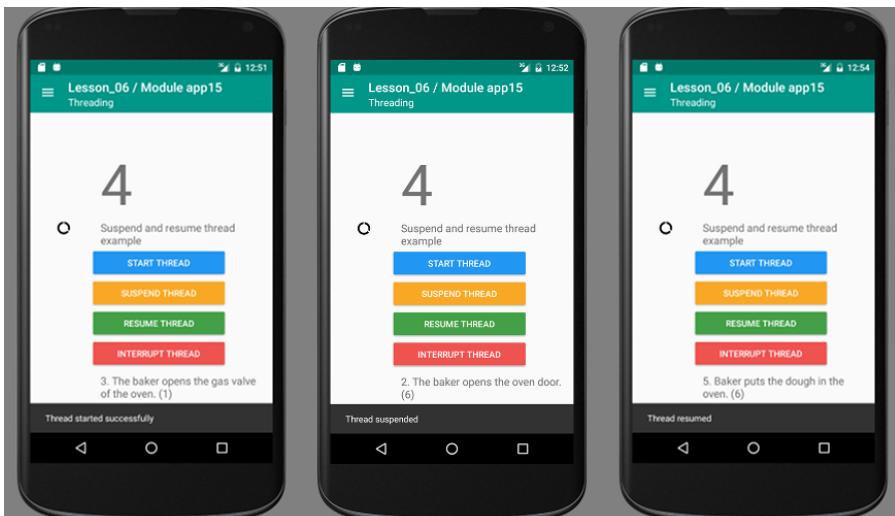
// ----- Suspend ThrExampleFour thread -----
case R.id.btnSuspendExampleFour :
    if (MainActivity.T4 != null)
    {
        MainActivity.T4.suspendWork();
        Snackbar.make(mainLL, "Thread suspended",
                     Snackbar.LENGTH_LONG).show();
    }
    break;

// ----- Resume ThrExampleFour thread -----
case R.id.btnResumeExampleFour :
    if (MainActivity.T4 != null)
    {
        MainActivity.T4.resumeWork();
        Snackbar.make(mainLL, "Thread resumed",
                     Snackbar.LENGTH_LONG).show();
    }
    break;
}
```

Как видно из Листинга 1.25, запуск и прерывание работы потока **ThrExampleFour** такие же как и других потоков из модуля «app15». Интерес для данного примера предо-

ставляет приостановка работы потока. Она осуществляется при обработке события нажатия на кнопку «Suspend thread» (идентификатор `R.id.btnSuspendExampleFour`). Остановка работы потока `ThrExampleFour` осуществляется при помощи вызова метода `suspendWork()` для этого потока (в Листинге 1.25 этот код выделен жирным шрифтом). Возобновление работы потока `ThrExampleFour` осуществляется при нажатии на кнопку «Resume thread» (идентификатор `R.id.btnResumeExampleFour`) при помощи вызова метода `resumeWork()` для этого потока (в Листинге 1.25 этот код выделен жирным шрифтом).

Внешний вид работы примера «4. Suspend and resume thread example» (см. Листинги 1.24 и 1.25) с приостановкой и возобновлением работы потока изображен на Рис. 1.7.



**Рис. 1.7.** Работа примера «4. Suspend and resume thread example». Слева направо: запуск потока, приостановка потока, возобновление работы потока

На Рис. 1.7 когда поток **ThrExampleFour** выводит информационное сообщение о работе «Пекаря», в конце этого сообщения в круглых скобках выводится номер цикла итерации работы потока (значение переменной **cnt**, см. Листинг 1.24)

Попробуйте запустить пример «4. Suspend and resume thread example» и несколько раз приостанавливать / возобновлять работу потока **ThrExampleFour**. Поток будет останавливаться только в безопасных для остановки местах. Программный код потока из Листинга 1.24 можно использовать как образец для остановки и возобновления работы потоков для любых задач.

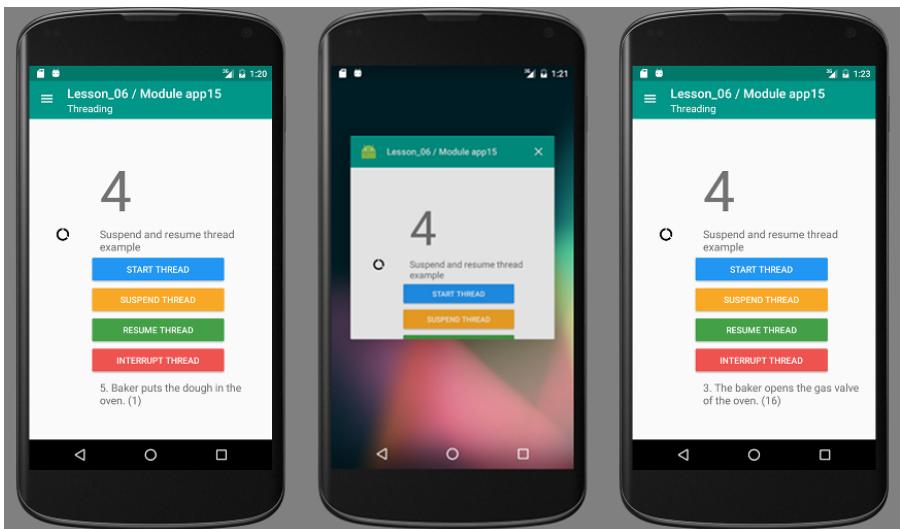
Также добавим, что код метода **run()** потока **ThrExampleFour** в Листинге 1.24 не совсем корректный с точки зрения прерывания потока при помощи метода **interrupt()**. Это сделано специально для того, чтобы вы в Домашнем задании доработали класс потока **ThrExampleFour** таким образом, чтобы он мог не только приостанавливаться в безопасных местах, но и корректно прерываться с учетом безопасных и небезопасных мест исполнения. Подробнее о задании вы сможете почитать в разделе «Домашнее задание» этого урока.

Исходный код примера из данного раздела находится в модуле «app15» среди файлов исходных кодов которые прилагаются к данному уроку.

## 1.6. Вторичные потоки и жизненный цикл Активности

Давайте проделаем следующий эксперимент с примером «4. Suspend and resume thread example» из предыдущего раздела. Вначале мы запустим пример, затем

сразу же уберем Активность на задний план (можно переключиться на другое приложение) и через некоторое время вернемся к Активности из примера «4. Suspend and resume thread example». Такая последовательность действий показана на Рис. 1.8.



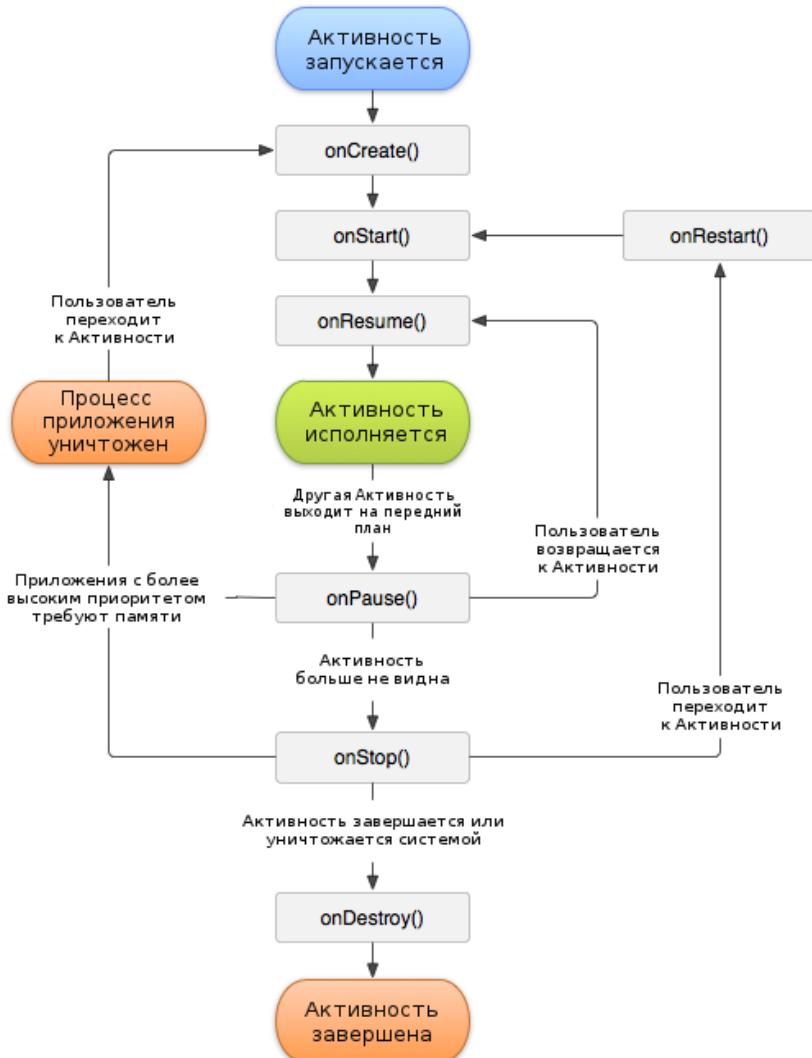
**Рис. 1.8.** Наблюдение за работой потока **ThrExampleFour** при помещении Активности приложения на задний план и последующем возврате Активности на передний план

После возврата Активности приложения на передний план мы увидим, что поток **ThrExampleFour** продолжает свою работу, и, за то время, пока Активность была на заднем плане, поток **ThrExampleTwo** не прекращал своей работы. Это совсем не плохо. Однако, бывают ситуации, когда при помещении Активности на задний план необходимо чтобы вторичные потоки приостановили свою работу. Такие ситуации относятся, например,

к игровым приложениям. Так как, за то время, пока Активность находится на заднем плане, игра может быть закончена проигрышем игрока. Согласитесь, что это не удобно и было бы логично в таких случаях автоматически приостанавливать работу вторичных потоков при уходе пользователя с Активности приложения и автоматически возобновлять работу вторичных потоков при возврате пользователя на Активность. Именно этому и посвящен текущий раздел данного урока.

Как же мы узнаем, когда необходимо приостановить работу вторичных потоков, а когда возобновить? Ответ достаточно прост — нам в этом помогут события жизненного цикла Активности. Мы уже знакомились с вами с событиями, через которые проходит Активность на протяжении своего жизненного цикла. Поэтому в данном уроке мы не будем повторно рассказывать знания из предыдущих уроков — лишь напомним о жизненном цикле Активности диаграммой, которая изображена на Рис. 1.9.

Если посмотреть на диаграмму жизненного цикла Активности из Рис. 1.9, то станет понятно, что для автоматической приостановки работы потока при уходе Активности с переднего плана лучше всего подойдет событие **onPause()**. А для автоматического возобновления работы потока при возврате Активности на передний план лучше всего подойдет событие **onResume()**. Разумеется, класс потока, работу которого необходимо будет приостанавливать и возобновлять, должен обладать соответствующими методами (см. предыдущий раздел текущего урока который посвящен приостановке и возобновлению работы вторичных потоков).



**Рис. 1.9.** Диаграмма событий жизненного цикла Активности

В данном разделе мы будем использовать класс потока из предыдущего раздела **ThrExampleFour**. Этот класс обладает необходимой функциональностью для

приостановки и возобновления своей работы (см. Листинг 1.24). В классе Активности **MainActivity** создадим методы обработчики событий **onResume()** и **onPause()** и напишем в них программный код (см. Листинг 1.26) при помощи которого поток **ThrExampleFour** будет автоматически приостанавливаться и автоматически возобновлять свою работу при уходе Активности с переднего плана и при возврате Активности на передний план соответственно.

**Листинг 1.26.** Автоматическая приостановка работы потока **ThrExampleFour** при уходе Активности с переднего плана в методе **onPause()**, и автоматическое возобновление работы потока **ThrExampleFour** при возврате Активности на передний план в методе **onResume()**

```
public class MainActivity extends AppCompatActivity
{
    ...
    @Override
    public void onResume()
    {
        super.onResume();

        if (MainActivity.T4 != null)
        {
            MainActivity.T4.resumeWork();
        }
    }

    @Override
    public void onPause()
    {
        super.onPause();
```

```

        if (MainActivity.T4 != null)
        {
            MainActivity.T4.suspendWork();
        }
    }
}

```

Как видно из Листинга 1.26, для приостановки работы потока используется знакомый нам (см. Листинг 1.24) метод **suspendWork()** класса **ThrExampleFour**, а для возобновления работы потока используется метод **resumeWork()** этого же класса.

Запустите пример «4. Suspend and resume thread example» из модуля «app15» и повторите последовательность действий, изображенную на Рис. 1.8. Теперь, поток **ThrExampleFour** при уходе Активности с переднего плана, приостанавливает свою работу, а при возврате Активности — возобновляет.

Исходный код примера из данного раздела находится в модуле «app15» среди файлов исходных кодов которые прилагаются к данному уроку.

## 1.7. Классы Handler, Message

В данном разделе будет рассказано о механизме обмена информацией (данными) между разными потоками. Этот механизм использует объекты классов [android.os.Handler](#) и [android.os.Message](#). Оба класса являются классами, производными от класса [java.lang.Object](#).

Класс [android.os.Message](#) предназначен для инкапсулирования (содержания) в себе информации, ко-

торая должна быть передана одним потоком другому потоку. Другими словами, класс **android.os.Message** это посылка (или сообщение) которая передается от одного потока к другому.

Конкретнее, объект **android.os.Message** посылается объекту **android.os.Handler**. Информация, которая размещается в объекте **android.os.Message** может быть объектом, или несколькими объектами, или (в целях экономии памяти) может быть одним или двумя целыми числами.

Класс **android.os.Handler** это получатель сообщений. Он создается в том потоке, который должен получать сообщения от других потоков. Более того, при создании объекта **android.os.Handler**, происходит привязка этого объекта к потоку в котором он создан.

Каждый объект **android.os.Handler** имеет свою собственную очередь сообщений, а так же может принимать не только информацию в виде данных, но и объекты **java.lang.Runnable** для их исполнения в своем потоке.

Давайте познакомимся с полями и методами классов **android.os.Message** и **android.os.Handler**, а затем закрепим полученные знания на примере.

Вначале познакомимся с классом **android.os.Message**, который имеет следующие поля и методы:

- **public int arg1** — поле **arg1** (и **arg2**) является легкой альтернативой использованию объекта (который назначается объекту **android.os.Message** при помощи метода **setData()**), для случая, когда нужно хранить только несколько целых значений.

- **public int arg2** — поле **arg2** (и **arg1**) является легкой альтернативой использованию объекта (который назначается объекту **android.os.Message** при помощи метода **setData()**), для случая, когда нужно хранить только несколько целых значений.
- **public Object obj** — произвольный объект с данными для отправки получателю. назначается объекту **android.os.Message** при помощи метода **setData()**.
- **public int sendingUid** — Необязательное поле, указывающее uid (идентификатор), отправителя сообщения.
- **public int what** — код сообщения, который задается программистом, чтобы объект получатель мог определить, что это за сообщение.
- **Message ()** — конструктор. Но предпочтительным способом получения объекта **android.os.Message** будет являться вызов статического метода **Message.obtain ()**.
- **Message obtain ()** — статический метод. Существует несколько вариантов перегрузки этого метода (см. справочную информацию по классу [android.os.Message](#)). Этот метод возвращает новый экземпляр объекта **android.os.Message** из глобального пула. Позволяет во многих случаях избегать создания новых объектов **android.os.Message**.
- **void setData (Bundle data)** — записывает набор (объект **Bundle**) произвольных значений данных в объект **android.os.Message**.

- **Bundle getData ()** — возвращает набор данных (объект **Bundle**), связанных с этим объектом **android.os.Message**. Если ранее объект **Bundle** не создавался, то метод создаст его.

Как видим, класс **android.os.Message** является обычным хранилищем. Поэтому использование этого класса не вызывает никаких сложностей.

Теперь познакомимся с полями и методами класса **android.os.Handler**:

- **Handler()** — конструктор по умолчанию, создает объект обработчика **android.os.Handler** и связывает этот обработчик с циклом обработки сообщений для текущего потока. Цикл обработки сообщений представлен классом **android.os.Looper**, который не рассматривается в данном уроке.
- **void handleMessage(Message msg)** — метод предназначен для переопределения в производных классах чтобы реализовать функциональность обработки получаемых сообщений (параметр **msg**).
- **boolean post(Runnable r)** — метод добавляет **Runnable r** в очередь сообщений для последующего исполнения.
- **boolean postAtTime(Runnable r, long uptimeMillis)** — метод добавляет **Runnable r** в очередь сообщений для последующего запуска в указанное в параметре **uptimeMillis** время. Время передается в формате **timestamp** в миллисекундах.
- **boolean postDelayed(Runnable r, long delayMillis)** — метод добавляет **Runnable r** в очередь сообщений

для последующего запуска через указанное в параметре `delayMillis` время (в миллисекундах).

- `void removeMessages(int what)` — метод удаляет все ожидающие в очереди сообщения со значением поля `what`.
- `boolean sendMessage(Message msg)` — помещает сообщение `msg` в конец очереди ожидающих сообщений после всех сообщений, время исполнения которых равно текущему времени.
- `boolean sendMessageAtTime(Message msg, long uptimeMillis)` — помещает сообщение `msg` в очередь сообщений после всех ожидающих сообщений, время запуска которых меньше абсолютного времени (в миллисекундах) `uptimeMillis`.
- `boolean sendMessageDelayed(Message msg, long delayMillis)` — то же, что и предыдущий метод, только время `delayMillis` указывается не абсолютное, а относительное.
- `boolean sendEmptyMessage(int what)` — помещает в очередь сообщений пустое сообщение, содержащее только значение `what`.

Теперь мы покажем принцип передачи сообщений `android.os.Message` из одного потока в обработчик `android.os.Handler` другого потока. Предположим, есть класс потока `TestThr`. Этот поток будет отсылать сообщения обработчику `android.os.Handler`. Код потока `TestThr` в таком случае будет иметь вид, который приведен в Листинге 1.27.

**Листинг 1.27.** Механизм отправки сообщения android.os.Message другому потоку в объект android.os.Handler

```
class TestThr extends Thread
{
    private Handler H;

    public TestThr(Handler h)
    {
        this.H = h;
    }

    @Override
    public void run()
    {
        while (true)
        {
            Message msg = new Message();

            // ----- Set data to the msg object -----
            // ----- Заполняем объект msg данными -----
            ...

            // ----- Send the message msg -----
            // ----- Отправляем сообщение msg -----
            this.H.sendMessage(msg);
        }
    }
}
```

Как видно из Листинга 1.27, поток который будет отправлять сообщения в виде объектов **android.os.Message**, должен иметь ссылку на объект **android.os.Handler** которому адресованы сообщения.

Поток, который получает сообщения (то есть владеет объектом **android.os.Handler**) имеет вид, который показан в Листинге 1.28.

**Листинг 1.28.** Код потока, который предназначен для получения и обработки сообщений android.os.Message

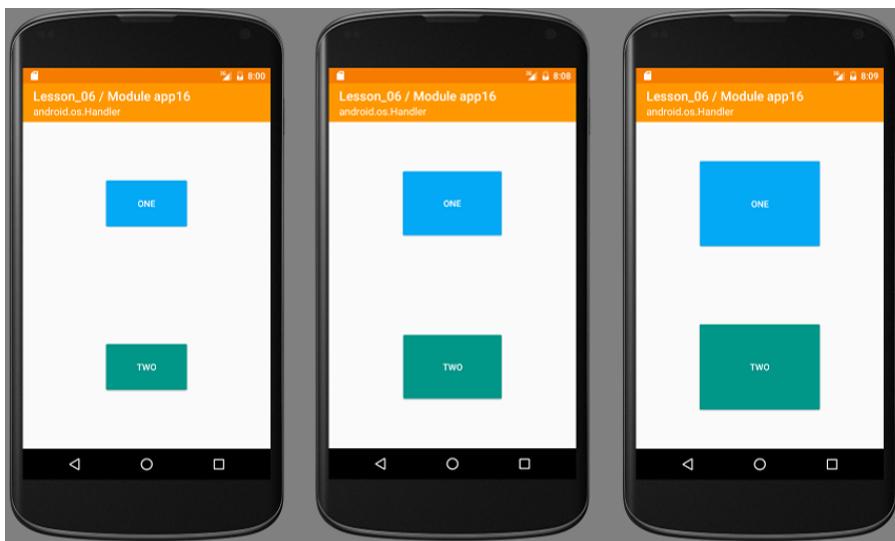
```
Handler H = new Handler()
{
    @Override
    public void handleMessage(Message msg)
    {
        // ----- Handling messages msg -----
        // ----- Обработка сообщений msg -----
        ...
    }
};

TestThr T1 = new TestThr(H);
T1.start();
```

Как видим из Листинга 1.28, поток, который будет получать сообщения в виде объектов **android.os.Message** от других потоков, должен создать производный класс от класса **android.os.Handler** и переопределить в нем метод **handleMessage()**. Затем этот поток должен создать объект этого класса и передать потокам отправителям ссылку на этот объект.

Теперь рассмотрим применение полученных знаний о классах **android.os.Message** и **android.os.Handler** на практическом примере. Для этого примера создан модуль «app16» в проекте Android Studio, который прилагается к данному уроку. Суть примера заключается в следующем. Есть класс вторичного потока (в примере он называется **MyThrResizer**), который с течением времени плавно меняет значения своих локальных переменных **width** и **height**.

и сообщает о новых значениях этих переменных первичному потоку. Первичный поток применяет полученные значения к ширине и высоте своего виджета `android.widget.Button`. В примере будет запущено два потока `MyThrResizer` и на Активности (то есть в первичном потоке) будут два виджета `android.widget.Button`, размеры для которых будут рассчитываться этими потоками. Поэтому, в процессе работы примера размеры виджетов `android.widget.Button` будут плавно меняться. Внешний вид работы примера показан на Рис. 1.10. Сообщения будут передаваться при помощи объектов `android.os.Message` и обрабатываться при помощи объекта `android.os.Handler`.



**Рис. 1.10.** Внешний вид рассматриваемого в данном разделе примера

Как видно из Рис 1.10, в рассматриваемом примере на Активности расположены два виджета `android.widget.`

**Button** с идентификаторами **R.id.btn1** (с надписью на кнопке «One») и **R.id.btn2** (с надписью на кнопке «Two»). Листинг файла ресурсов с макетом внешнего вида Активности (/res/layout/activity\_main.xml) в этом уроке не приводится ввиду своей понятности. Класс **MyThrResizer** является внутренним классом для класса Активности **MainActivity**. Программный код класса **MyThrResizer** показан в Листинге 1.29.

**Листинг 1.29.** Программный код класса MyThrResizer

```
public class MainActivity extends AppCompatActivity
{
    // ----- Inner classes -----
    class MyThrResizer extends Thread
    {
        /**
         * Reference to the message handler object
         * Ссылка на объект обработчик сообщений
         */
        private android.os.Handler H;

        /**
         * Widget ID for which the dimensions will change
         * Идентификатор виджета для которого будут
         * меняться размеры
         */
        private int what;

        public MyThrResizer(android.os.Handler H,
                            int what)
        {
            this.H = H;
            this.what = what;
        }
    }
}
```

```
@Override
public void run()
{
    int width = 200; // initial button width
    int height = 100; // initial button height
    boolean isForward = true; // true -
                                // increase size,
                                // otherwise - decrease
    try
    {
        while (true)
        {
            Thread.sleep(20);

// ----- Changing width and height -----
// ----- Изменение переменных width и height -----
            if (isForward)
            {
                width++; height++;
                if (width > 400) isForward = false;
            }
            else
            {
                width--; height--;
                if (width < 200) isForward = true;
            }

// ----- Sending a message to the primary thread
// ----- about new button sizes -----
// ----- Передача сообщения первичному потоку
// ----- о новых размерах кнопки -----
                Message msg = new Message();
                msg.arg1 = width;
                msg.arg2 = height;
                msg.what = this.what;
                this.H.sendMessage(msg);
            }
        }
    }
```

```
        catch (InterruptedException ie) { }
    }
}
...
}
```

Как видно из Листинга 1.29, объекту потока `MyThr-Resizer` через конструктор передается ссылка на объект `android.os.Handler` (параметр `H`) которому объект `MyThr-Resizer` будет отправлять сообщения с величинами размеров. Также через конструктор объекту `MyThrResizer` передается идентификатор виджета (параметр `what`), для которого будет рассчитываться изменение размера с течением времени.

При передаче сообщения `android.os.Message`, первичному потоку передается такая информация: через поле `what` передается значение идентификатора виджета, размер которого необходимо поменять; через поле `arg1` передается значение новой ширины виджета; через поле `arg2` передается значение новой высоты виджета. Если бы был выбран способ передачи значений через объект `android.os.Bundle`, то создание объекта `android.os.Message` и заполнение его информацией выглядело бы так, как показано в Листинг 1.30.

**Листинг 1.30.** Передача сообщения android.os.Message со значениями, находящимися в объекте android.os.Bundle

```
Bundle B = new Bundle();
B.putInt("btnId", this.what);
B.putInt("width", width);
```

```
B.putString("height", height);
Message msg = new Message();
msg.setData(B);
this.H.sendMessage(msg);
```

Теперь давайте рассмотрим как в первичном потоке создается объект **android.os.Handler** и как в этом объекте происходит обработка сообщений, поступающих от вторичных потоков. Интересующий нас программный код находится в методе **onCreate()** класса Активности **MainActivity** и показан в Листинге 1.31.

**Листинг 1.31.** Программный код класса Активности который относится к созданию объекта **android.os.Handler** и обработке событий этим объектом

```
public class MainActivity extends AppCompatActivity
{
    // ----- Inner classes -----
    class MyThrResizer extends Thread
    {
        ...
    }

    // ----- Class members -----
    private static Handler H;
    private static Activity activity;
    private static MyThrResizer MTR1;
    private static MyThrResizer MTR2;

    // ----- Class methods -----
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
```

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

// ----- Toolbar -----
Toolbar toolBar = (Toolbar) this.findViewById(
    R.id.toolbar);
this.setSupportActionBar(toolBar);
toolBar.setTitle(R.string.app_name);
toolBar.setSubtitle("android.os.Handler");
toolBar.setTitleTextColor(Color.WHITE);
toolBar.setSubtitleTextColor(Color.WHITE);

// ----- Handler -----
MainActivity.activity = this;
if (MainActivity.H == null)
{
    MainActivity.H = new Handler()
    {
        @Override
        public void handleMessage(android.
            os.Message msg)
        {
            Button B = (Button)
                MainActivity.activity.
                    findViewById(msg.what);
            if (B != null)
            {
                B.setWidth (msg.arg1);
                B.setHeight(msg.arg2);
            }
        }
    };
}

if (MainActivity.MTR1 == null)
{
    MainActivity.MTR1 = new
        MyThrResizer(MainActivity.H, R.id.btn1);
```

```
        MainActivity.MTR1.start();
    }
    if (MainActivity.MTR2 == null)
    {
        MainActivity.MTR2 = new
            MyThrResizer(MainActivity.H, R.id.btn2);
        MainActivity.MTR2.start();
    }
}
```

Как видно из Листинга 1.31, в классе **MainActivity** объявлены в виде статических полей две ссылки на объекты потоков **MyThrResizer**: **MTR1** и **MTR2**. То есть, эти потоки создаются один раз при старте приложения и продолжают работать до завершения работы приложения. При повороте устройства эти потоки не пересоздаются и продолжают работать дальше. Поскольку эти потоки во время своего создания получают ссылку на объект **android.os.Handler**, то экземпляр объекта **android.os.Handler** также создается один раз при старте приложения и ссылка на этот объект также объявлена статической. Это позволяет не беспокоится о переинициализации полей вторичных потоков **MyThrResizer** после поворота устройства.

Внешний вид работы примера из Листингов 1.30 и 1.31 изображен на Рис. 1.10.

Исходный код примера из данного раздела находится в модуле «app16» среди файлов исходных кодов которые прилагаются к данному уроку.

## 2. Сетевое программирование для мобильных устройств Android

Мобильные устройства при сетевом взаимодействии, в силу своей специфики, в большинстве случаев выступают в качестве клиентов для архитектуры «Клиент-Сервер». Поэтому, при рассмотрении материала текущего раздела, изложение материала по сетевому программированию для мобильных устройств Android происходит в контексте создания клиентских приложений. Поскольку, в подавляющем большинстве случаев, в качестве сетевых подключений мобильным устройствам доступны «Мобильный Интернет» и «Wi-Fi» (что фактически то же подключение к сети Интернет), то протоколом данных выступает протокол HTTP (или HTTPS). Большинство Android приложений, которые подключаются к сети, используют HTTP для отправки и получения данных.

Чтобы выполнять сетевые операции в Android приложении, необходимо добавить в Манифест приложения следующие разрешения:

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

Перед тем, как приступать к реализации функциональности обмена данными по сети, компания Google рекомендует (<https://developer.android.com/training/basics/network-ops/connecting.html>) убедиться, что данные и информация в разрабатываемом приложении остаются в безопасности при передаче по сети. Для этого необходимо следовать следующим рекомендациям:

- Минимизировать количество конфиденциальных или личных пользовательских данных, которые передаются по сети.
- Отправлять весь сетевой трафик из своего приложения через SSL (*Secure Sockets Layer* — уровень защищённых сокетов). Т.е., использовать протокол HTTPS.
- Продумать создание конфигурации сетевой безопасности, позволяющей приложению доверять настраиваемым ЦС (Центр Сертификации, см. [https://ru.wikipedia.org/wiki/Центр\\_сертификации](https://ru.wikipedia.org/wiki/Центр_сертификации)) или ограничить набор системных ЦС, для безопасной связи.

И еще один важный момент: передача данных по сети в Android приложениях осуществляется во вторичных потоках. Попытка реализовать сетевой обмен данными в первичном потоке приведет к исключительной ситуации.

## **2.1. Класс android.net.ConnectivityManager. Выбор сетевого подключения. Получение информации о сетевых подключениях**

Для получения информации о состоянии подключения к сети или для получения доступа к передаче данных через сеть, предназначен класс [android.](#)

[net.ConnectivityManager](#). Иерархия класса `android.net.ConnectivityManager` очень проста:

```
java.lang.Object
|
+--- android.net.ConnectivityManager
```

Для получения ссылки на объект `android.net.ConnectivityManager` используется метод класса `android.content.Context`:

```
T getSystemService (Class<T> serviceClass);
```

который применяется следующим образом (например в методе `onCreate()`):

```
ConnectivityManager cm = (ConnectivityManager)
    this.getSystemService(Context.CONNECTIVITY_SERVICE);
```

Давайте познакомимся с методами класса `android.net.ConnectivityManager`:

- `void addDefaultNetworkActiveListener(ConnectivityManager.OnNetworkActiveListener l)` — метод добавляет объект обработчик событий, который будет получать уведомления о том, что сеть, которая является сетью по умолчанию в ОС Android, доступна к обмену данными. Готовность сети по умолчанию означает, что настало удобное время для обмена данными по сети.
- `Network getActiveNetwork()` — возвращает объект `android.net.Network` (объект сетевого подклю-

чения), который соответствует текущей активной сети передачи данных, то есть сети, которая будет использоваться для исходящих сетевых соединений. В случае, если произойдет отключение (разрыв соединения) активного сетевого подключения, то возвращенный этим методом объект не может быть более использован. Если в системе нет активного сетевого подключения, то метод вернет **null**. Использование этого метода требует разрешения **ACCESS\_NETWORK\_STATE**, которое должно быть указано в Манифесте приложения.

- **NetworkInfo getActiveNetworkInfo()** — возвращает информацию о текущем активном сетевом подключении. Текущее активное сетевое подключение используется для исходящих сетевых соединений. Если в системе нет активного сетевого подключения, то метод вернет **null**. Использование этого метода требует разрешения **ACCESS\_NETWORK\_STATE**, которое должно быть указано в Манифесте приложения.
- **Network[] getAllNetworks()** — возвращает массив объектов **android.net.Network** (объектов сетевых подключений) которые в настоящее время доступны в системе. Использование этого метода требует разрешения **ACCESS\_NETWORK\_STATE**, которое должно быть указано в Манифесте приложения.
- **NetworkInfo getNetworkInfo(Network network)** — Возвращает объект с информацией о сетевом соединении для конкретного объекта сетевого соеди-

нения **network**. Использование этого метода требует разрешения **ACCESS\_NETWORK\_STATE**, которое должно быть указано в Манифесте приложения.

Как видно из перечисленных выше методов, ключевыми классами, объекты которых возвращают методы класса **android.net.ConnectivityManager** являются классы **android.net.Network** и **android.net.NetworkInfo**. Эти классы рассматриваются в следующих разделах этого урока.

## 2.2. Получение информации о сетевом подключении. Класс **android.net.NetworkInfo**

Класс **android.net.NetworkInfo** содержит информацию о сетевом подключении (сетевом интерфейсе) для конкретного объекта **android.net.Network**. Иерархия класса выглядит так:

```
java.lang.Object  
|  
+--- android.net.NetworkInfo
```

Методы класса **android.net.NetworkInfo**:

- **NetworkInfo.DetailedState getDetailedState()** — возвращает детализированный статус состояния сетевого соединения **android.net.Network**. Перечень **NetworkInfo.DetailedState** содержит такие состояния статуса как **AUTHENTICATING**, **CONNECTED**, **CONNECTING**, **DISCONNECTED**, **FAILED**, **IDLE** и другие. Такая информация является детальной и поэтому требуется далеко не каждому приложению. В подавляющем большинстве случаев для

получения состояния статуса используется метод `getState()`.

- `String getExtraInfo()` — возвращает дополнительную информацию о состоянии сети, если таковая информация была предоставлена низкоуровневыми сетевыми уровнями.
- `String getReason()` — метод возвращает строку с информацией, почему для доступного сетевого соединения `android.net.Network` попытка установить соединение не удалась. Если информация не доступна, метод вернет `null`.
- `NetworkInfo.State getState()` — возвращает текущее состояние сетевого соединения для объекта `android.net.Network`. В отличии от детального состояния `NetworkInfo.DetailedState`, перечень `NetworkInfo.State` ограничивается всего лишь следующими значениями: `CONNECTING`, `CONNECTED`, `DISCONNECTING`, `DISCONNECTED`.
- `int getSubtype()` — возвращает целочисленное значение, описывающее подтип сети. Это значение специфично для подтипа сети. Для получения строкового значения подтипа сети предназначен метод `getSubtypeName()`.
- `String getSubtypeName()` — метод возвращает понятное имя, описывающее подтип сети.
- `int getType()` — возвращает целочисленный тип сети, к которой относится информация в данном объекте `android.net.NetworkInfo`.

- **String getTypeName()** — метод возвращает понятное имя, описывающее тип сети, например «WIFI» или «MOBILE».
- **boolean isAvailable()** — метод возвращает значение, которое является признаком, возможно ли подключение к сети. Сеть недоступна, если постоянное или временное условие предотвращает возможность подключения к этой сети. Например: 1) устройство находится за пределами зоны обслуживания сети, 2) устройство находится в сети, отличной от домашней сети (т. е. роуминге), а роуминг данных отключен, 3) радиоприемник устройства отключен, потому что включен режим самолета.
- **boolean isConnected()** — указывает, существует ли сетевое подключение, и можно ли установить соединение для передачи данных. Рекомендуется всегда вызывать этот метод перед попыткой выполнить передачу данных.

Чтобы получить ссылку на объект **android.net.NetworkInfo** для активного сетевого подключения, необходимо выполнить действия, показанные в Листинге 2.1 или в Листинге 2.2.

#### **Листинг 2.1.** Пример получения объекта android.net.NetworkInfo

```
ConnectivityManager cm = (ConnectivityManager)
    this.getSystemService(Context.
    CONNECTIVITY_SERVICE);
Network N = cm.getActiveNetwork();
NetworkInfo NI = cm.getNetworkInfo(N);
```

**Листинг 2.2.** Пример получения объекта android.net.NetworkInfo для активного сетевого подключения

```
ConnectivityManager cm = (ConnectivityManager)
    this.getSystemService(Context.
        CONNECTIVITY_SERVICE);
NetworkInfo NI = cm.getActiveNetworkInfo();
```

Для получения информации о всех доступных сетевых подключениях создадим приложение, которое находится в модуле «app17» проекта Android Studio который прилагается к данному уроку.

Активность рассматриваемого приложения будет содержать менеджер **android.support.design.widget.TabLayout** и контейнер **android.support.v4.view.ViewPager**. Исходный код макета Активности (файл `/res/layout/activity_main.xml`) показан в Листинге 2.3.

**Листинг 2.3.** Файл макета внешнего вида Активности `/res/layout/activity_main.xml` рассматриваемого примера

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent"

    android:fitsSystemWindows="true"
    android:id="@+id/clMain"
    android:background="#E0E0E0"
    tools:context="itstep.com.myapp17.MainActivity">
```

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        app:popupTheme=
            "@style/ThemeOverlay.AppCompat.Light"
        app:layout_collapseMode="pin"
    />

    <android.support.design.widget.TabLayout
        android:id="@+id/tabLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:tabGravity="fill"
        app:tabMode="fixed" >
    </android.support.design.widget.TabLayout>

</android.support.design.widget.AppBarLayout>

<android.support.v4.view.ViewPager
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior=
        "@string/appbar_scrolling_view_behavior"
    android:id="@+id/vpContent">
    </android.support.v4.view.ViewPager>

</android.support.design.widget.CoordinatorLayout>
```

Информация о каждом доступном сетевом соединении будет отображаться на отдельной странице контейнера **android.support.v4.view.ViewPager**. Внешний

вид каждой страницы представлен макетом, который находится в файле ресурсов /res/layout/fragment\_page.xml. Содержимое этого файла показано в Листинге 2.4.

**Листинг 2.4.** Содержимое файла ресурсов /res/layout/fragment\_page.xml с макетом внешнего вида страницы для контейнера android.support.v4.view.ViewPager

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.NestedScrollView
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#E0E0E0">

    <android.support.v7.widget.CardView
        android:id="@+id/card_view"
        android:layout_width="match_parent"
        android:layout_height="140dp"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginTop="8dp"
        app:contentPadding="8dp"
        app:cardCornerRadius="4dp">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical">

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center_horizontal"
                android:textSize="12sp"
```

```
        android:text="Type Name"
    />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="24sp"
    android:textColor="#404040"
    android:id="@+id/tvTypeName"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:textSize="12sp"
    android:text="Subtype Name"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="16sp"
    android:textColor="#404040"
    android:id="@+id/tvSubtypeName"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:textSize="12sp"
```

```
        android:text="Network state"
    />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="16sp"
    android:textColor="#404040"
    android:id="@+id/tvState"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:textSize="12sp"
    android:text="Extra Info"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="16sp"
    android:textColor="#404040"
    android:id="@+id/tvExtraInfo"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:textSize="12sp"
    android:text="Is available"
/>
```

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:textSize="16sp"  
    android:textColor="#404040"  
    android:id="@+id/tvAvailable"  
/>  
  
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:layout_marginTop="16dp"  
    android:textSize="12sp"  
    android:text="Is connected"  
/>  
  
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:textSize="16sp"  
    android:textColor="#404040"  
    android:id="@+id/tvConnected"  
/>  
</LinearLayout>  
  
</android.support.v7.widget.CardView>  
  
</android.support.v4.widget.NestedScrollView>
```

Как видно из Листинга 2.4, информация на странице будет отображаться внутри виджета **android.support.v7.widget.CardView** и будет содержать информацию о назывании типа сетевого соединения (*TypeName*), назывании

подтипа сетевого соединения (*SubtypeName*), дополнительной информации (*ExtraInfo*), состояния сетевого соединения (*State*), а также о доступности сетевого соединения (*Is available*) и наличия сетевого соединения (*Is connected*). Внешний вид страницы изображен на Рис. 2.1.

Каждая страница из Листинга 2.4 представляется классом Фрагмента **android.support.v4.app.Fragment**. В рассматриваемом примере этот класс называется **MyFragmentPage** (класс располагается в файле *MyFragmentPage.java*). Содержимое этого класса показано в Листинге 2.5.

**Листинг 2.5.** Класс *MyFragmentPage* который представляет страницы для контейнера *android.support.v4.view.ViewPager*

```
public class MyFragmentPage extends Fragment
{
    private NetworkInfo NI;

    public void setNetworkInfo(NetworkInfo NI)
    {
        this.NI = NI;
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container,
                            Bundle savedInstanceState)
    {
        ViewGroup rootView = (ViewGroup) inflater.inflate(
            R.layout.fragment_page, container, false);

        // ----- NetworkInfo Type Name -----
        TextView tvTypeName = (TextView)
            rootView.findViewById(R.id.tvTypeName);
    }
}
```

```
tvTypeName.setText(this.NI.getTypeName());  
  
// ----- NetworkInfo Subtype Name -----  
TextView tvSubtypeName = (TextView)  
    rootView.findViewById(R.id.tvSubtypeName);  
tvSubtypeName.setText(this.NI.getSubtypeName());  
  
// ----- NetworkInfo State -----  
TextView tvState = (TextView)  
    rootView.findViewById(R.id.tvState);  
tvState.setText(this.NI.getState().toString());  
  
// ----- NetworkInfo Extra info -----  
TextView tvExtraInfo = (TextView)  
    rootView.findViewById(R.id.tvExtraInfo);  
tvExtraInfo.setText(this.NI.getExtraInfo());  
  
// ----- NetworkInfo Available -----  
TextView tvIsAvailable = (TextView)  
    rootView.findViewById(R.id.tvAvailable);  
tvIsAvailable.setText((this.  
    NI.isAvailable())?"YES":"NO");  
  
// ----- NetworkInfo Connected -----  
TextView tvIsConnected = (TextView)  
    rootView.findViewById(R.id.tvConnected);  
tvIsConnected.setText((this.  
    NI.isConnected())?"YES":"NO");  
return rootView;  
}  
}
```

Как видно из Листинга 2.5, класс Фрагмента **MyFragmentPage** содержит в качестве источника данных о сетевом подключении ссылку на объект **android.net.NetworkInfo**, которая передается в объект при помощи

метода `setNetworkInfo()`. В методе `onCreateView()` объекта `MyFragmentPage`, происходит создание страницы на основе файла ресурсов `/res/layout/fragment_page.xml` (см. Листинг 2.4) и заполнение этой страницы информацией из объекта `android.net.NetworkInfo`.

Далее, как вы уже знаете, источником для страниц контейнера `android.support.v4.view.ViewPager` выступает специальный Адаптер страниц (класс, производный от класса `android.support.v4.view.PagerAdapter`). В рассматриваемом примере создан класс Адаптера страниц. Этот класс называется `MyFragmentStatePagerAdapter` (класс находится в файле `MainActivity.java`). Содержимое этого класса показано в Листинге 2.6.

**Листинг 2.6.** Класс Адаптера страниц `MyFragmentStatePagerAdapter` для контейнера `android.support.v4.view.ViewPager`

```
class MyFragmentStatePagerAdapter extends
    FragmentStatePagerAdapter
{
    private ArrayList<Fragment> pages =
        new ArrayList<>();

    public MyFragmentStatePagerAdapter(FragmentManager fm)
    {
        super(fm);
    }

    @Override
    public Fragment getItem(int position)
    {
        return this.pages.get(position);
    }
}
```

```
@Override  
public int getCount()  
{  
    return this.pages.size();  
}  
  
public void addPage(Fragment fragment)  
{  
    this.pages.add(fragment);  
}  
}
```

И наконец, получение информации о всех доступных сетевых соединениях и заполнение этой информацией Адаптер страниц происходит в методе **onCreate()** класса **MainActivity**. Содержимое класса **MainActivity** показано в Листинге 2.7.

**Листинг 2.7.** Получение информации о всех доступных сетевых соединениях и заполнение этой информацией контейнер android.support.v4.view.Viewpager

```
public class MainActivity extends AppCompatActivity  
{  
    // ----- Class members -----  
    private ViewPager viewPager;  
    private FragmentStatePagerAdapter adapter;  
  
    // ----- Class methods -----  
    @Override  
    protected void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```

```
// ----- Toolbar -----
Toolbar toolBar = (Toolbar)
    this.findViewById(R.id.toolbar);
this.setSupportActionBar(toolBar);
toolBar.setTitle(R.string.app_name);
toolBar.setSubtitle("android.net.NetworkInfo");
toolBar.setTitleTextColor(Color.rgb(0x82,
    0x77, 0x17));
toolBar.setSubtitleTextColor(Color.rgb(0x82,
    0x77, 0x17));

// ----- ViewPager -----
this.viewPager = (ViewPager)
    this.findViewById(R.id.vpContent);

// ----- TabLayout -----
final TabLayout tabLayout = (TabLayout)
    this.findViewById(R.id.tabLayout);
tabLayout.setOnTabSelectedListener(
    new TabLayout.OnTabSelectedListener()
{
    @Override
    public void onTabSelected(TabLayout.Tab tab)
    {
        MainActivity.this.viewPager.
            setCurrentItem(tab.
                getPosition(), true);
    }

    @Override
    public void onTabUnselected(TabLayout.
        Tab tab)
    {
    }

    @Override
    public void onTabReselected(TabLayout.
        Tab tab)
```

```
        {
    }
}) ;

// ----- PagerAdapter -----
this.adapter = new MyFragmentStatePagerAdapter(
    this.getSupportFragmentManager());

// ----- Retrieving Information about all Network
// ----- objects -----
ConnectivityManager cm = (ConnectivityManager)
    this.getSystemService(Context.
CONNECTIVITY_SERVICE);

Network[] allNetworks = cm.getAllNetworks();
for (Network N : allNetworks)
{
    NetworkInfo NI = cm.getNetworkInfo(N);

// ----- Add a Tab to TabLayout -----
tabLayout.addTab(tabLayout.newTab().
    setText(NI.getTypeName()));

// ----- Add a Page To PagerAdapter -----
MyFragmentPage fragmentPage =
    new MyFragmentPage();
fragmentPage.setNetworkInfo(NI);
((MyFragmentStatePagerAdapter)
    (this.adapter)).addPage(fragmentPage);
}

// ----- Set the PagerAdapter to ViewPager -----
this.viewPager.setAdapter(this.adapter);
}
```

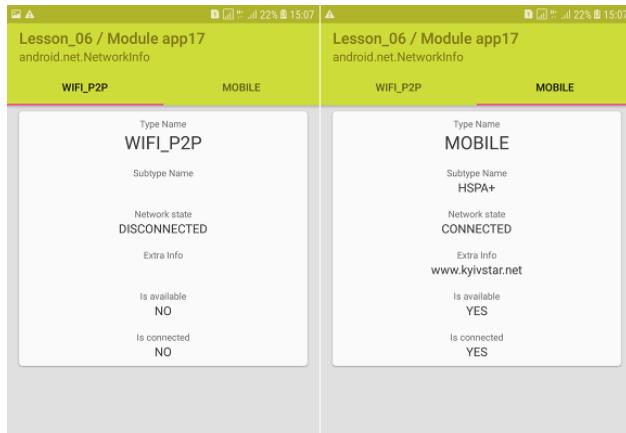
Внешний вид примера из Листингов 2.3–2.7 изображен на Рис. 2.1 и Рис. 2.2.



**Рис. 2.1.** Внешний вид работы примера из Листингов 2.3–2.7 для эмулятора

На Рис. 2.1 изображен внешний вид работы примера из модуля «app17» для случая, когда приложение запущено в эмуляторе. Как видно, в эмуляторе доступно только одно сетевое подключение — «Мобильный Интернет» («Mobile»). Как видно из Рис. 2.1, эмулятор дает только минимальную картину о доступных сетевых соединениях. Поэтому приложение из модуля «app17» было запущено на реальном устройстве (Samsung Galaxy J7, API 24). Внешний вид работы примера на реальном устройстве изображен на Рис. 2.2.

Как видно из Рис. 2.2, на реальном устройстве, кроме сетевого подключения «Mobile», существует еще сетевое подключение «WIFI\_P2P». Информация о сетевом подключении «WiFi» на этом реальном устройстве отсутствует.



**Рис. 2.2.** Внешний вид работы примера из Листингов 2.3–2.7 на реальном устройстве

Отсутствие сетевого подключения «WiFi» объясняется тем, что в настройках мобильного телефона подключение к WiFi было намерено отключено. Вы можете запустить приложение из модуля «app17» на своих реальных устройствах с разными настройками подключения WiFi чтобы понаблюдать за тем, как будет меняться информация в объектах `android.net.NetworkInfo` в случае если сетевое подключение WiFi будет включено.

Исходный код примера из данного раздела находится в модуле «app17» среди файлов исходных кодов которые прилагаются к данному уроку.

## 2.3. Сетевое подключение. Класс `android.net.Network`

Класс `android.net.Network` представляет сетевое подключение, при помощи которого можно передавать и получать данные по сети.

Иерархия класса выглядит следующим образом:

```
java.lang.Object  
|  
+--- android.net.Network
```

В предыдущих разделах данного урока уже рассказывалось, как получать и использовать объекты **android.net.Network** для получения информации о сетевых подключениях. В этом разделе мы рассмотрим, как при помощи объектов **android.net.Network** передавать и получать данные по сети.

При изложении материала данного раздела мы исходим из того, что вы знакомы с курсом сетевого программирования Java, и следовательно, знаете о классах **[java.net.InetAddress](#)**, **[java.net.Socket](#)**, **[java.net.ServerSocket](#)**, **[java.net.URLConnection](#)**. Если вы не знакомы с курсом сетевого программирования Java или подзабыли его, то вам необходимо немедленно восполнить отсутствующие знания и только затем вернуться к изучению материалов данного раздела.

Теперь давайте познакомимся с основными методами класса **android.net.Network**:

- **InetAddress[] getAllByName(String host)** — Возвращает массив объектов **InetAddress**, каждый из которых представляет Интернет Адрес (*IP address*), ассоциированный с указанным именем хоста **host**.
- **InetAddress getByName(String host)** — Возвращает объект **InetAddress** который содержит IP адрес ассоциированный с указанным именем хоста **host**.

- **SocketFactory getSocketFactory()** — Возвращает ссылку на объект **javax.net.SocketFactory** который является «Фабрикой» (см. Паттерн «Abstract Factory») которая создает объекты сокетов **java.net.Socket** для сети, которую представляет данный объект **android.net.Network**. Знакомство с классом **SocketFactory** выходит за рамки данного урока и мы предлагаем вам, при необходимости, самостоятельно ознакомиться с данным классом.
- **URLConnection openConnection(URL url)** — Создает соединение (в виде объекта производного класса от класса **java.net.URLConnection**) в сети, подключение к которой представляет данный объект **android.net.Network**. Далее через это соединение можно отправлять и получать данные по протоколам HTTP или HTTPS.
- **URLConnection openConnection(URL url, Proxy proxy)** — Метод делает то же самое, что и предыдущий метод, только позволяет указать параметры для прокси сервера **proxy**, через который необходимо осуществлять сетевой обмен данными.

Напомним, что для обмена данными по сети необходимо в Манифесте приложения указать соответствующие разрешения:

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

А также напомним, что обмен данными по сети в Android приложениях необходимо выполнять во вторичных потоках.

В общем случае, отправка запроса по протоколу HTTP (методом GET) от устройства к некоторому серверу (например к серверу, который содержит сайт [example.com](http://example.com)) будет состоять из следующих шагов:

1. Получить при помощи объекта `android.net.ConnectivityManager` ссылку на объект `android.net.Network`, который представляет активное сетевое подключение.
2. При помощи объекта `android.net.Network` создать объект [java.net.HttpURLConnection](http://java.net.HttpURLConnection), при помощи которого осуществить подключение к серверу example.com и отправку HTTP запроса на этот сервер.
3. После отправки запроса, получить при помощи метода `getInputStream()` класса `java.net.HttpURLConnection` ссылку на объект байтового потока [java.io.InputStream](http://java.io.InputStream), из которого прочитать полученные от сервера данные.

Перечисленная последовательность шагов показана в Листинге 2.8.

**Листинг 2.8.** Пример использования объектов `android.net.Network` и `java.net.HttpURLConnection` для посылки и отправки данных по сети по протоколу HTTP

```
// ----- Getting a reference to an active Network  
// ----- connection -----  
// ----- Получение ссылки на активное сетевое  
// ----- подключение -----
```

```
ConnectivityManager cm = (ConnectivityManager)
    MainActivity.this.
    getSystemService(Context.
    CONNECTIVITY_SERVICE);

Network N = cm.getActiveNetwork();

try
{
    // ----- Obtaining the HttpURLConnection object
    // ----- and sending the request -----
    // ----- Получение объекта HttpURLConnection
    // ----- и отправка запроса -----
    HttpURLConnection cn = (HttpURLConnection)
        N.openConnection(
            new URL("http://example.com"));
        cn.setDoInput(true);
    cn.connect();

    // ----- Reading response from server -----
    // ----- Чтение полученного от сервера ответа -----
    InputStream IS = cn.getInputStream();
    ByteArrayOutputStream BAOS =
        new ByteArrayOutputStream();
    byte[] b = new byte[1024];

    while (true)
    {
        int cnt = IS.read(b, 0, b.length);
        if (cnt == -1) break;
        BAOS.write(b, 0, cnt);
    }
    byte[] a = BAOS.toByteArray();
    BAOS.reset();

    final String content = new String(a, 0,
        a.length, "UTF8");
}
```

```
>MainActivity.this.runOnUiThread(new Runnable()
{
    @Override
    public void run()
    {
        MainActivity.this.tvContent.setText(content);
    }
});

// ----- Close the connection to the server -----
// ----- Закрываем соединение с сервером -----
cn.disconnect();
}

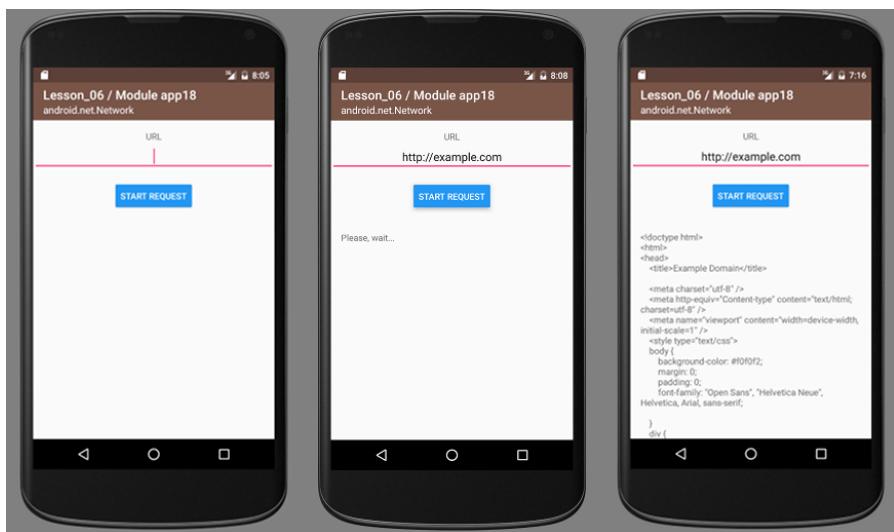
catch (Exception e)
{
    Log.d(TAG, "Error! " + e.getMessage());
}
```

В Листинге 2.8 используется запрос к сайту [example.com](http://example.com). Это специальный сайт, который находится в сети Интернет и предназначен для посылки к нему тестовых запросов.

Теперь настало время рассмотреть пример приложения, которое будет передавать и получать данные по сети. Для этого примера приложения, создадим модуль «app18» в проекте Android Studio, который прилагается к данному уроку.

Внешний вид примера изображен на Рис. 2.3. Суть примера заключается в следующем. На Активности есть текстовое поле **android.widget.EditText** (идентификатор **R.id.edtUrl**) в который пользователь вводит адрес сайта, к которому приложение должно отправить HTTP за-

прос и получить HTTP ответ. На Активности есть кнопка **android.widget.Button** с надписью «Start Request». Этой кнопке назначен обработчик события — метод **btnStartRequestClick()**. При нажатии на эту кнопку, приложение берет адрес сайта из текстового поля **R.id.edtUrl**, отправляет HTTP запрос к этому сайту и получает HTTP ответ от этого сайта. Полученный ответ выводится в текстовое поле **android.widget.TextView** с идентификатором **R.id.tvContent**. Обращаем ваше внимание, что поскольку в рассматриваемом примере данные отправляются к Интернет сайтам, то полученные от этих сайтов ответы приходят в формате HTML, что мы и будем наблюдать в текстовом поле **R.id.tvContent** (см. Рис. 2.3).



**Рис. 2.3.** Внешний вид рассматриваемого в данном разделе примера. Слева внешний вид приложения после запуска, в центре — пользователь ввел адрес сайта и нажал на кнопку «Start request», справа — полученный ответ отобразился в текстовом поле R.id.tvContent

Содержимое класса **MainActivity** рассматриваемого примера показано в Листинге 2.9.

**Листинг 2.9.** Класс MainActivity рассматриваемого в данном разделе примера

```
public class MainActivity extends AppCompatActivity
{
    // ----- Inner classes -----
    class ThrRequest extends Thread
    {
        /**
         * A indicator of a waiting for a user to click
         * a "Start request" button
         * Признак ожидания потоком события нажатия
         * пользователем кнопки "Start request"
         */
        private boolean isWaiting = true;
        private EditText edtUrl;
        private TextView tvContent;
        private Activity activity;

        @Override
        public void run()
        {
            try
            {
                while (true)
                {
                    if (this.edtUrl == null) break;

                    // ----- Waiting the user's click -----
                    this.isWaiting = true;
                    this.waiting();
                    String strUrl = this.edtUrl.getText().toString();
                }
            }
        }
    }
}
```

```
if (strUrl.isEmpty()
    || (!strUrl.startsWith("http://") &&
        !strUrl.startsWith("https://")))
{
    this.showErrorInSnackbar(
        "Url must be not empty and starts with" +
        "'http://' or 'https://'";
    continue;
}

// ----- Getting a reference to an active Network
// ----- connection -----
// ----- Получение ссылки на активное сетевое
// ----- подключение -----
    ConnectivityManager cm =
        (ConnectivityManager)
            this.activity.
                getSystemService(Context.
                    CONNECTIVITY_SERVICE);

    Network N = cm.getActiveNetwork();

    try
    {

// ----- Obtaining the HttpURLConnection object
// ----- and sending the request -----
// ----- Получение объекта HttpURLConnection
// ----- и отправка запроса -----
        HttpURLConnection cn =
            (HttpURLConnection)
                N.openConnection(new
                    URL(strUrl));
        cn.setDoInput(true);
        cn.connect();

// ----- Reading response from server -----
    }
}
```

```
// ----- чтение полученного от сервера ответа -----
InputStream IS = cn.getInputStream();
ByteArrayOutputStream BAOS =
    new ByteArrayOutputStream();
byte[] b = new byte[1024];

while (true)
{
    int cnt = IS.read(b, 0, b.length);
    if (cnt == -1) break;
    BAOS.write(b, 0, cnt);
}
byte[] a = BAOS.toByteArray();
BAOS.reset();

final String content =
    new String(a, 0, a.length, "UTF8");

this.activity.runOnUiThread(new
    Runnable()
{
    @Override
    public void run()
    {
        ThrRequest.this.
            tvContent.setText(content);
    }
});

// ----- Close the connection to the server -----
// ----- Закрываем соединение с сервером -----
cn.disconnect();
}

catch (Exception e)
{
    this.showErrorInSnackbar("Error! " +
        e.getMessage());
}
```

```
        }
    }
}
catch (InterruptedException ie)
{
}
}

/***
 * The method tells the thread that it is
 * necessary to wake up and send an HTTP request
 * Метод сообщает потоку, что необходимо
 * проснуться и послать HTTP запрос
 */
public synchronized void startRequest()
{
    this.isWaiting = false;
    this.notify();
}

/***
 *The method suspends the work of the thread,
 * until the startRequest () method is called
 * Метод приостанавливает работу потока до тех пор,
 * пока не будет вызван метод startRequest()
 */
private synchronized void waiting()
    throws InterruptedException
{
    while (this.isWaiting)
    {
        this.wait();
    }
}

/***
 * The method displays an error message in
 * the Snackbar object
*/
```

```
* Метод показывает сообщение об ошибке
* в объекте Snackbar
* @param err - Error message
*/
private void showErrorInSnackbar(String err)
{
    final String strErr = err;
    this.activity.runOnUiThread(new Runnable()
    {
        @Override
        public void run()
        {
            Snackbar.make(
                ((MainActivity) ThrRequest.this.
                activity).llMain,
                strErr, Snackbar.LENGTH_LONG)
                .setAction("OK", new View.
                OnClickListener()
                {
                    @Override
                    public void onClick(View v)
                    {
                    }
                })
                .show();
        }
    });
}

public void setWidgets(EditText edtUrl,
                      TextView tvContent, Activity activity)
{
    this.edtUrl = edtUrl;
    this.tvContent = tvContent;
    this.activity = activity;
}
```

```
// ----- Class members -----
private LinearLayout llMain;
private EditText edtUrl;
private TextView tvContent;
private static ThrRequest TR;

// ----- Class methods -----
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

// ----- Initializing the object's fields -----
    this.llMain = (LinearLayout)
        this.findViewById(R.id.llMain);
    this.edtUrl = (EditText)
        this.findViewById(R.id.edtUrl);
    this.tvContent = (TextView)
        this.findViewById(R.id.tvContent);

// ----- Starting the Request Thread -----
    if (MainActivity.TR == null)
    {
        MainActivity.TR = new ThrRequest();
        MainActivity.TR.start();
    }
    MainActivity.TR.setWidgets(this.edtUrl,
        this.tvContent, this);

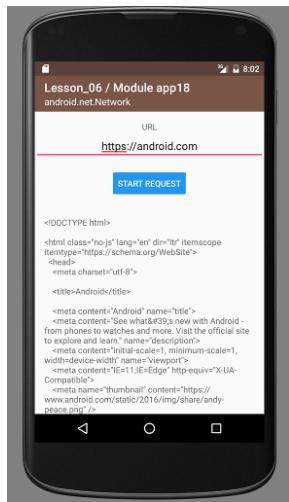
// ----- Toolbar -----
    Toolbar toolBar = (Toolbar)
        this.findViewById(R.id.toolbar);
    this.setSupportActionBar(toolBar);
    toolBar.setTitle("Lesson_06 / Module app18");
    toolBar.setSubtitle("android.net.Network");
```

```
        toolBar.setTitleTextColor(Color.WHITE);
        toolBar.setSubtitleTextColor(Color.WHITE);
    }

    public void btnStartRequestClick (View v)
    {
        MainActivity.this.tvContent.setText("Please,
            wait...");
        MainActivity.TR.startRequest ();
    }
}
```

Сделаем некоторые пояснения к Листингу 2.9. Для обмена сетевыми данными (для посылки HTTP запроса и получения HTTP ответа) предназначен поток, класс которого называется **ThrRequest**. Поток в приложении исполняется в единственном экземпляре, поэтому создается поток один раз при старте приложения и класс Активности ссылается на объект этого потока при помощи статической ссылки (имя статического поля **TR**). Поток **ThrRequest** находится в приостановленном состоянии при помощи вызова метода своего класса **waiting()**. Механизм приостановки потоков рассматривался ранее в этом уроке. Как только пользователь вводит адрес сайта и нажимает на кнопку «Start request», поток **ThrRequest** выводится из состояния ожидания при помощи метода **startRequest()** (см. метод **btnStartRequestClick()** из Листинга 2.9) и выполняет запрос по принципу, который описан в Листинге 2.8. При получении ответа, поток **ThrRequest** при помощи метода класса Активности **runOnUiThread()** записывает содержимое полученного ответа в текстовое поле **R.id.tvContent**.

Также обращаем ваше внимание, что пример из Листинга 2.9 может выполнять не только запросы по протоколу HTTP, но и по протоколу HTTPS (см. Рис. 2.4).



**Рис. 2.4.** Работа примера из Листинга 2.9  
с протоколом HTTPS

Возможность примера из Листинга 2.9 выполнять запросы и получать ответы по протоколу HTTPS объясняется тем, что метод `openConnection()` класса `java.net.URLConnection` возвращает ссылку как на объект `java.net.HttpURLConnection`, так и на объект `javax.net.HttpsURLConnection` в зависимости от типа URL. В примере из Листинга 2.9 для получения результата вызова метода `openConnection()` используется тип `java.net.HttpURLConnection`:

```
HttpURLConnection cn = (HttpURLConnection)
N.openConnection(new URL(strUrl));
```

Но, класс `java.net.HttpURLConnection` является родительским для класса `javax.net.HttpsURLConnection`:

```
java.lang.Object
  |
  +-- java.net.URLConnection
    |
    +-- java.net.HttpURLConnection
      |
      +-- javax.net.ssl.
          HttpsURLConnection
```

Поэтому, пример из Листинга 2.9 работает по протоколу HTTPS через объект родительского класса `java.net.HttpURLConnection` используя полиморфизм (виртуальные методы) объектно-ориентированного программирования!

В рассмотренном в данном разделе примере выполнялся HTTP запрос к серверу без отправки каких-либо данных. В большинстве случаев Android приложениям приходится выполнять отправку данных в запросах HTTP. Механизмы отправки данных методами GET и POST описываются в следующих разделах данного урока.

Исходный код примера из данного раздела находится в модуле «app18» среди файлов исходных кодов которые прилагаются к данному уроку.

## 2.4. Отправка данных в HTTP запросе методом GET

Отправка данных методом GET в HTTP запросах осуществляется путем добавления строки формата из Листинга 2.10 в адресную строку запроса после символа «?» (см. Листинг 2.11).

**Листинг 2.10.** Формат строки с данными, отправляемыми методом GET

```
parameter_name1=value1&parameter_name2=
    value2&parameter_nameN=valueN
```

**Листинг 2.11.** Адресная строка с данными, которые отправляются методом GET

```
http://someSiteAddress.domain/someScript?
    parameter_name1=value1&parameter_name2=value2
```

Символ «?» является разделителем между адресом сайта и данными, которые отправляются методом GET. Сами данные (см. Листинг 2.10) представлены в виде набора пар «ключ-значение». Каждая пара разделяется друг от друга символом «&». В Листинге 2.10 названия ключей следующие: **parameter\_name1**, **parameter\_name2**, **parameter\_nameN**. Значения для этих ключей соответственно следующие: **value1**, **value2**, **valueN**. Чтобы отправить запрос с данными методом GET из Листинга 2.11 необходимо создать соединение, как показано в Листинге 2.12.

**Листинг 2.12.** Создание запроса java.net.HttpURLConnection с отправкой данных методом GET

```
HttpURLConnection cn = (HttpURLConnection)
    .openConnection(
        new URL("http://someSiteAddress.domain/someScript" +
            "?parameter_name1=
            value1&parameter_name2=value2"));
```

Для демонстрации отправки данных в HTTP запросе методом GET необходим сервер, который бы обработал полученные данные и вернул бы ответ нашему приложению. Мы предоставим вам сценарий PHP (см. Листинг 2.13), который вам необходимо будет выложить на локальный компьютер, если у вас на локальном компьютере развернута система для разработки Web-приложений (Apache, PHP, MySQL). Если на вашем компьютере нет такой системы (или вы хотите исполнить пример в условиях, которые близки к реальным), то в таком случае мы рекомендуем вам выложить сценарий PHP из Листинга 2.13 на бесплатный хостинг в сети Интернет. Авторы урока считают, что вы знакомы с курсом «Разработка Web-приложений с использованием PHP/MySQL» и поэтому вам понятны термины «PHP», «бесплатный хостинг» и так далее. Если вы еще не знакомы с курсом «Разработка Web-приложений с использованием PHP/MySQL» то мы рекомендуем обратиться за помощью в исполнении примера из данного раздела к вашему преподавателю.

**Листинг 2.13.** Содержимое PHP сценария для рассматриваемого в данном разделе примера

```
<?php
$firstName = (isset($_REQUEST["firstname"]))
             ?$_REQUEST["firstname"] :"noname";
$lastName = (isset($_REQUEST["lastname"]))
             ?$_REQUEST["lastname"] :"noname";
echo "<h1>Hello, $firstName $lastName!</h1>";
```

Сразу же отметим, что сценарий PHP из Листинга 2.13 предназначен как для получения данных методом

GET, так и для получения данных методом POST. Этот сценарий будет использоваться и в следующем разделе.

Далее, при рассмотрении примера, будем считать, что сценарий из Листинга 2.13 находится в файле с названием testrequest.php на сервере с вымышленным адресом http://someaddress.com. Сценарий из Листинга 2.13 принимает два параметра: **firstname** и **lastname** (имя и фамилию). Получив значения имени и фамилии в параметрах **firstname** и **lastname**, сценарий из Листинга 2.13 выводит приветствие в заголовке первого уровня: «Hello Имя Фамилия!». Чтобы отправить имя и фамилию на сервер http://someaddress.com сценарию testrequest.php необходимо сформировать адресную строку, которая имеет следующий вид:

```
http://someaddress.com/testrequest.  
php?firstname=William&lastname=Blake
```

Программный код отправки такого запроса показан в Листинге 2.14.

#### Листинг 2.14. Отправка данных методом GET.

```
// ----- Getting a reference to an active Network  
// ----- connection -----  
// ----- Получение ссылки на активное сетевое  
// ----- подключение -----  
ConnectivityManager cm = (ConnectivityManager)  
    MainActivity.this.getSystemService(Context.  
    CONNECTIVITY_SERVICE);  
  
Network N = cm.getActiveNetwork();
```

```
try
{
// ----- Obtaining the HttpURLConnection object
// ----- and sending the request -----
// ----- Получение объекта HttpURLConnection
// ----- и отправка запроса -----
    HttpURLConnection cn = (HttpURLConnection)
        N.openConnection(new URL(
            "http://someaddress.com/testrequest.php" +
            "?firstname=William&lastname=Blake"));
    cn.setDoInput(true);
    cn.connect();

// ----- Reading response from server -----
// ----- Чтение полученного от сервера ответа -----
    InputStream IS = cn.getInputStream();
    ByteArrayOutputStream BAOS =
        new ByteArrayOutputStream();
    byte[] b = new byte[1024];
    while (true)
    {
        int cnt = IS.read(b, 0, b.length);
        if (cnt == -1) break;
        BAOS.write(b, 0, cnt);
    }
    byte[] a = BAOS.toByteArray();
    BAOS.reset();
    final String content =
        new String(a, 0, a.length, "UTF8");

MainActivity.this.runOnUiThread(new Runnable()
{
    @Override
    public void run()
    {
        MainActivity.this.tvContent.
            setText(content);
    }
});
```

```
        }
    }) ;

// ----- Close the connection to the server -----
// ----- Закрываем соединение с сервером -----
cn.disconnect();
}

catch (Exception e)
{
    ...
}
```

Программный код из Листинга 2.14 может быть напрямую использован в коде метода **run()** класса **ThrRequest** из Листинга 2.9.

Как видно из Листинга 2.14, получение HTTP ответа для случая отправки данных методом GET ничем не отличается от получения ответов от обычных запросов HTTP.

Исходный код примера, который рассматривается в данном разделе, вы не найдете среди файлов с исходными кодами для этого урока потому, что этот пример вам будет необходимо выполнить самостоятельно в качестве домашнего задания. При самостоятельном исполнении, вам необходимо будет заменить вымышленный адрес <http://someaddress.com> на реальный адрес сервера, который вы будете использовать для реализации данного примера.

## 2.5. Отправка данных в HTTP запросе методом POST

При создании серверных сценариев (например при помощи PHP, NodeJS и т.д.) или при создании программ серверов (например при помощи Java, C++) разработчик

сам выбирает, каким способом (GET или POST) клиентское приложение будет отправлять данные на сервер. Однако, существует твердая рекомендация, что если объем передаваемых данных будет превышать размер 512 байт, то данные необходимо отправлять методом POST.

В отличии от отправки данных методом GET, данные, отправляемые методом POST отправляются в теле HTTP запроса. (К сведению — адресная строка, которая содержит данные GET, находится в заголовке HTTP запроса). Формат данных, передаваемых методом POST, такой же, как и для метода GET (см. Листинг 2.10).

Чтобы сформировать объект **java.net.HttpURLConnection** который отправляет данные методом POST необходимо выполнить действия, показанные в Листинге 2.15.

**Листинг 2.15.** Запись данных, передаваемых методом POST в тело HTTP запроса

```
try
{
    HttpURLConnection cn =
        (HttpURLConnection) N.openConnection(
            new URL("http://someaddress.com"));

    cn.setDoInput(true);
    cn.setDoOutput(true);

    OutputStream out = cn.getOutputStream();
    String str = "key1=value1&key2=value2&keyN=valueN";
    byte[] a = str.getBytes("UTF8");
    out.write(a, 0, a.length);
    ...
}
```

Как видно из Листинга 2.15, для отправки данных методом POST, объекту `java.net.HttpURLConnection` необходимо при помощи вызова метода `setDoOutput(true)` указать, что перед отправкой запроса будет осуществляться запись в тело HTTP запроса. После этого, необходимо при помощи вызова метода `getOutputStream()` получить ссылку на байтовый поток записи, который представляет тело HTTP запроса. Записывая байты в этот поток при помощи метода `write()` мы и запишем данные в тело HTTP запроса.

Для примера данного раздела воспользуемся сценарием PHP из Листинга 2.13. Как и в предыдущем разделе, мы рекомендуем вам самостоятельно исполнить данный пример воспользовавшись локальным Web-сервером или услугами бесплатного хостинга. Полный код формирования запроса с POST данными и получения ответа показан в Листинге 2.16.

### Листинг 2.16. Пример отправки данных методом POST

```
try
{
    HttpURLConnection cn =
        (HttpURLConnection) N.openConnection(
            new URL("http://someaddress.com"));

    cn.setDoInput(true);

    // ----- Writing the POST data into HTTP body -----
    // ----- Запись данных, отправляемых методом POST
    // ----- в тело HTTP запроса -----
    cn.setDoOutput(true);
    OutputStream out = cn.getOutputStream();
```

```
String str = "firstname=William&lastname=Blake";
byte[] a = str.getBytes("UTF8");
out.write(a, 0, a.length);

// ----- Reading response from server -----
// ----- чтение полученного от сервера ответа -----
InputStream IS = cn.getInputStream();
ByteArrayOutputStream BAOS =
    new ByteArrayOutputStream();
byte[] b = new byte[1024];

while (true)
{
    int cnt = IS.read(b, 0, b.length);
    if (cnt == -1) break;
    BAOS.write(b, 0, cnt);
}
byte[] a = BAOS.toByteArray();
BAOS.reset();

final String content =
    new String(a, 0, a.length, "UTF8");

MainActivity.this.runOnUiThread(new Runnable()
{
    @Override
    public void run()
    {
        MainActivity.this.tvContent.
            setText(content);
    }
});

// ----- Close the connection to the server -----
// ----- Закрываем соединение с сервером -----
cn.disconnect();
}

catch (Exception e)
```

```
{  
    ...  
}
```

Программный код из Листинга 2.16 может быть напрямую использован в коде метода `run()` класса `ThrRequest` из Листинга 2.9.

В Листинге 2.16 происходит отправка данных методом POST к серверному сценарию `testrequest.php` (см. Листинг 2.13) который находится на вымышленном сервере с адресом `http://someaddress.com`. Сценарий `testrequest.php` из Листинга 2.13 принимает два параметра: `firstname` и `lastname` (имя и фамилию). Получив значения имени и фамилии в параметрах `firstname` и `lastname`, сценарий из Листинга 2.13 выводит приветствие в заголовке первого уровня: «Hello Имя Фамилия!».

Как видно из Листинга 2.16, получение HTTP ответа для случая отправки данных методом POST ничем не отличается от получения ответов от обычных запросов HTTP и от запросов, которые отправляют данные методом GET.

Исходный код примера, который рассматривается в данном разделе, вы не найдете среди файлов с исходными кодами для этого урока потому, что этот пример вам будет необходимо выполнить самостоятельно в качестве домашнего задания. При самостоятельном исполнении этого примера, вам необходимо будет заменить вымышленный адрес `http://someaddress.com` на реальный адрес сервера, который вы будете использовать для реализации данного примера.

### 3. Домашнее задание

1. Доделать пример с классом потока «Пекарь» (класс **ThrExampleFour** из Листинга 1.24).

Как уже упоминалось в данном уроке, код метода **run()** потока **ThrExampleFour** в Листинге 1.24 не совсем корректный с точки зрения прерывания потока при помощи метода **interrupt()**. Необходимо доработать класс потока **ThrExampleFour** таким образом, чтобы он мог не только приостанавливаться в безопасных местах, но и мог быть корректно прерван при помощи метода **interrupt()** с учетом безопасных и небезопасных мест исполнения. Уточнение: поток прерывается извне только при помощи вызова метода **interrupt()**, и поток должен прерваться в любом безопасном месте. При этом функциональность приостановки и возобновления работы потока в классе **ThrExampleFour** должна остаться.

2. Переделайте код в методе **run()** класса **MyThrResizer** из Листинга 1.29 таким образом, чтобы передача значений **width**, **height**, **what** осуществлялась через объект **android.os.Bundle**.
3. Реализовать отправку данных методом GET из Листинга 2.14. Для этого разместите сценарий из Листинга 2.13 в виде файла с именем **testrequest.php** на Web-сервере. Создайте приложение в котором пользователь введет в текстовые поля имя и фамилию

и нажав на кнопку «Отправить» отправит данные на сервер сценарию testrequest.php. Полученный от сервера ответ необходимо вывести в специальном текстовом поле.

4. Реализовать отправку данных методом POST из Листинга 2.16. Остальные условия те же, что и предыдущей задаче.

## 4. Экзаменационное задание

Курс «Программирование мобильных устройств Android» подходит к завершению и поэтому настало время задать вам экзаменационное задание. Курс еще не завершен, но Экзаменационное задание для этого курса серьезное и интересное и поэтому приступать к нему необходимо уже сейчас.

В качестве Экзаменационного задания необходимо разработать клиент-серверное приложение «Чат». Перед тем как описывать условия задания, сообщим, что скриншоты которые сопровождают это задание, намеренно выполнены не в стиле Material Design. Но, реализация задания должна быть выполнена в концепции Material Design. И, для усложнения задания, вы должны самостоятельно увидеть элементы и виджеты Material Design, которые должны присутствовать в разработанном вами приложении.

Итак, необходимо разработать клиент-серверное приложение «Чат». При помощи этого приложения пользователи Android устройств смогут общаться друг с другом посредством текстовых сообщений. Клиентом в данной системе будет выступать приложение Android которое необходимо разработать. Для сервера необходимо разработать либо набор сценариев PHP, либо Java приложение, либо сценарии NodeJS — на ваш выбор. Приветствуется, если серверная часть системы будет размещена в сети Интернет (на бесплатном хостинге).

Передача сообщений осуществляется от клиента (мобильного устройства Android) на сервер. Для этого у пользователя Android приложения должна быть возможность ввести текст сообщения и нажать на кнопку «Отправить». Сообщения в «Чате» отправляются от всех участников ко всем участникам (т.е. реализовывать функциональность приватных сообщений не требуется).

Получение новых сообщений осуществляется при помощи регулярно посылаемых от клиента специальных запросов. Эти запросы должны отправляться в специальном вторичном потоке. Период отправки таких запросов — 3 секунды. Поток должен получать только новые сообщения, то есть сообщения, которые уже получены от сервера, повторно не должны быть получены.

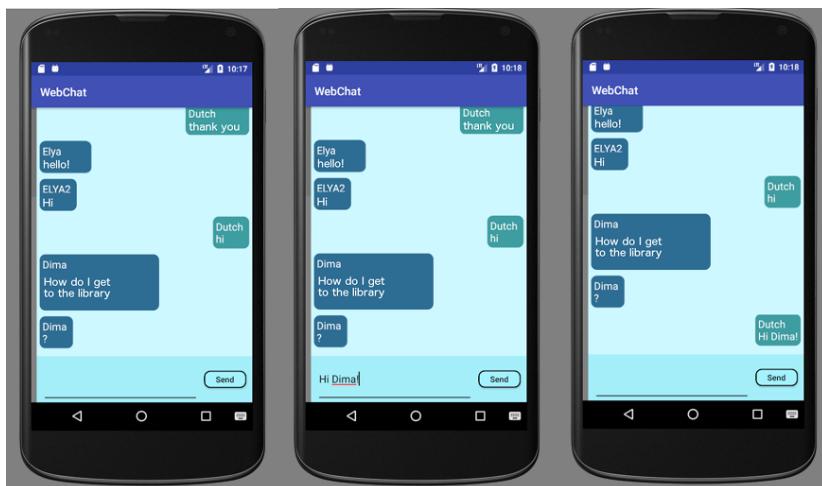


**Рис. 4.1.** Панель ввода логина и пароля (слева) и панель регистрации пользователя чата (справа)

Получение списка пользователей «online» также осуществляется при помощи регулярно посылаемых от клиента специальных запросов. Эти запросы должны отправляться в еще одном специальном вторичном потоке. Период отправки таких запросов — 5 секунд.

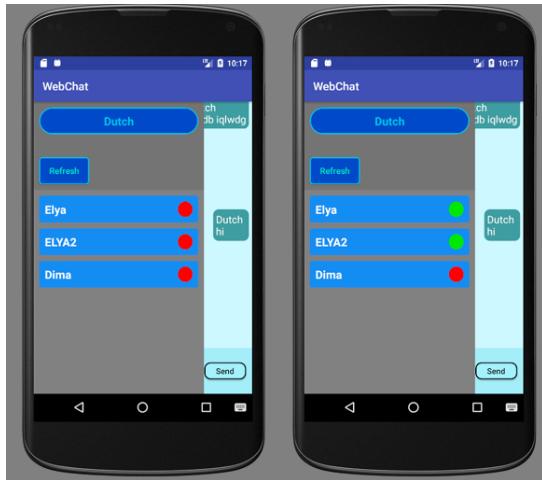
При запуске приложения, пользователь должен ввести свой логин и пароль (см. Рис. 4.1). Если пользователь не зарегистрирован в системе, то у него должна быть возможность зарегистрироваться (см. Рис. 4.1). Зарегистрированные пользователи сохраняются на сервере в БД.

Далее, зайдя в чат, пользователь видит сообщения от других пользователей и может отправлять свои сообщения, как показано на Рис. 4.2.



**Рис. 4.2.** Общий вид сообщений чата и отправка пользователем сообщений

Пользователь чата может видеть список пользователей «online», как показано на Рис. 4.3.



**Рис. 4.3.** Список пользователей «online»

Как видно из Рис. 4.3, в списке пользователей отображаются все пользователи, причем пользователи которые находятся сейчас в чате отмечаются зеленым кружком, а отсутствующие пользователи отображаются красным кружком. Что касается списка пользователей «online», то вы можете, по желанию, отображать только пользователей, находящихся сейчас в чате («online»).

Еще раз сделаем акцент, каким должно быть клиентское Android приложение — оно должно быть сетевым многопоточным приложением в стиле Material Design и с виджетами из библиотек совместимости.

Желаем вам удачи!



## Урок № 7.

# Многопоточность.

## Сетевое программирование

© Бояршинов Юрий

© Компьютерная Академия «Шаг»

[www.itstep.org](http://www.itstep.org)

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.