Е. В. СЕНЬКО

Программирование приложений для мобильных устройств под управлением Android

ЧАСТЬ 2

Евгений Сенько

Программирование приложений для мобильных устройств под управлением Android. Часть 2

«Издательские решения»

Сенько Е. В.

Программирование приложений для мобильных устройств под управлением Android. Часть 2 / Е. В. Сенько — «Издательские решения»,

ISBN 978-5-44-856607-3

Книга посвящена разработке программ для мобильных устройств под управлением операционной системы Android. Рассматривается создание приложений с использованием системных компонентов и служб Android. Приведены базовые данные о структуре приложений, об основных классах и их методах, сопровождаемые примерами кода.

ISBN 978-5-44-856607-3

© Сенько Е. В.

© Издательские решения

Содержание

Notifications – Уведомления 6
Потоки 19
Работа с сетью 36
Broadcast Receivers – приемники широковещательных сообщений 55
Конец ознакомительного фрагмента. 59

Программирование приложений для мобильных устройств под управлением Android Часть 2

Евгений Владимирович Сенько

© Евгений Владимирович Сенько, 2022

ISBN 978-5-4485-6607-3
Создано в интеллектуальной издательской системе Ridero

Это вторая часть книги, здесь рассмотрены: уведомления – Notifications, потоки и асинхронное выполнение задач – Threads & AsyncTask, работа с сетью, приемники широковещательных сообщений – Broadcast Receivers, оповещения – Alarms, графика и анимация, управление тачем и жестами, управление мультимедией – музыкой, видео и встроенной камерой, работа с датчиками, определение местоположения и привязка к картам, управление данными, а также классы ContentProvider и Service.

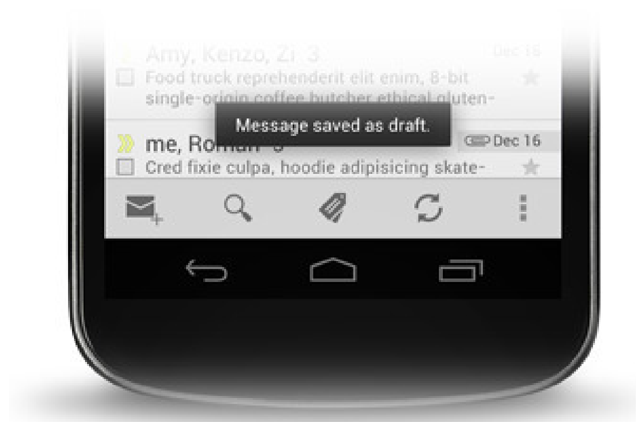
Часть 1 содержит шесть глав, описывающих основные принципы создания приложений, пользовательский интерфейс, полномочия приложений, а так же базовые классы: Activity, Intent, Fragment.

Книга предназначена для программистов, владеющих языком программирования Java и желающих освоить написание приложений, работающих под ОС Android. Книга является переводом общедоступных бесплатных англоязычных интернет ресурсов.

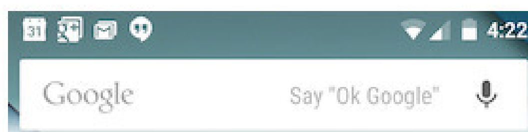
Notifications – Уведомления

Уведомление — это сообщение, которое может быть выведено на экран за пределами обычного пользовательского интерфейса приложения. Например, предположим, что у вас есть приложение, которое может загрузить электронную книгу из Интернета. Во время загрузки пользователь может продолжить использовать приложение или даже выйти из него в то время, как книга загружается. И тогда вы, вероятно, захотите сообщить пользователю об окончании загрузки. Чтобы сделать это, вы должны выяснить, когда загрузка закончится, а затем вывести на экран сообщение об этом.

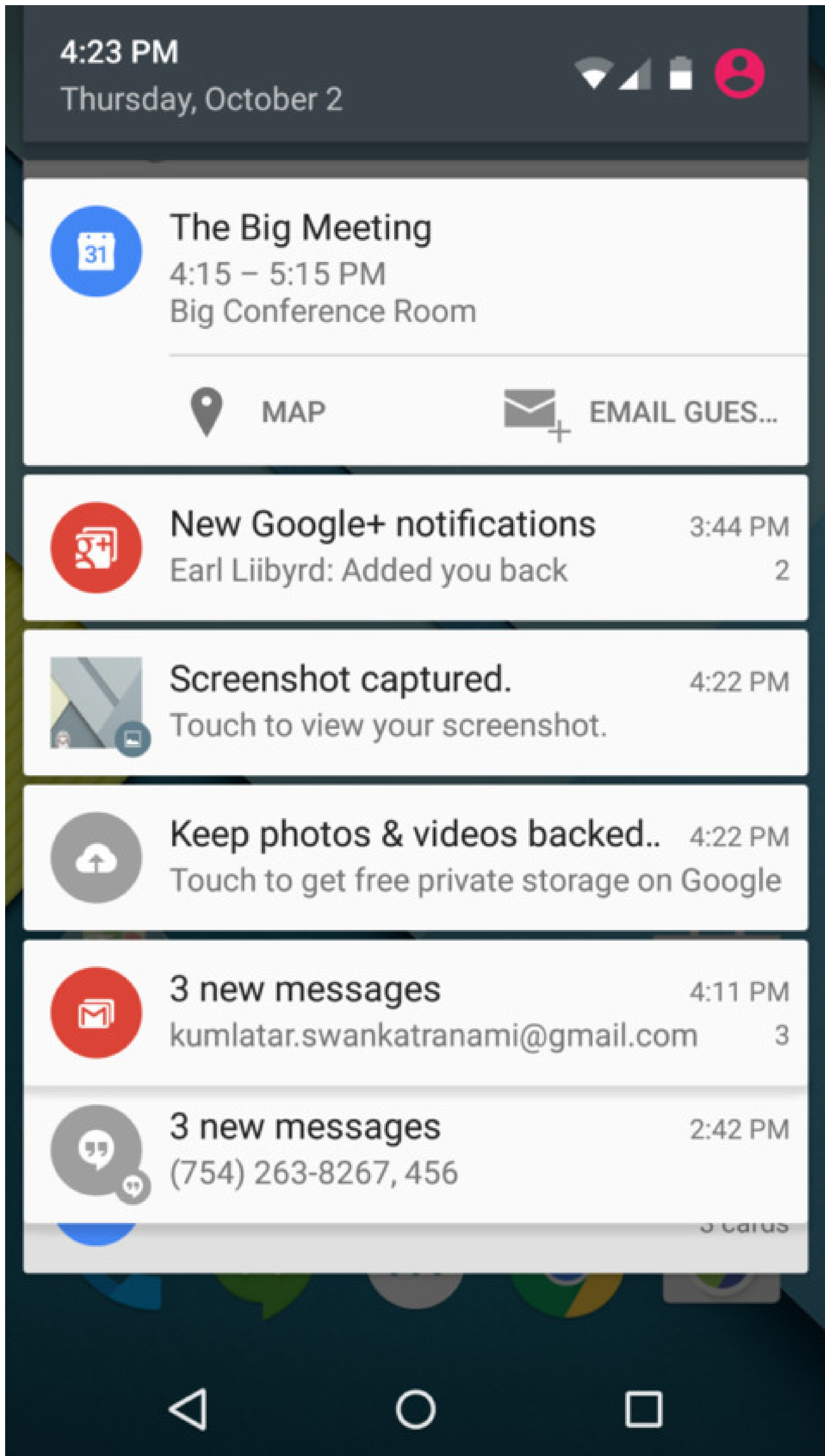
Далее рассмотрим два различных вида пользовательских уведомлений, которые поддерживает Android. Первый вид сообщения – тост (toast). Тост обеспечивает простую обратную связь от операции в небольшом всплывающем окне. Это окно занимает небольшое пространство, минимально необходимое для сообщения, а текущая Activity остается видимой и интерактивной.



Второй вид – уведомления в строке состояния (status bar) или области уведомлений. Когда приложение сообщает системе о необходимости выдать уведомление, оно сначала отображается в виде значка в области уведомлений в верхней части экрана слева.



Чтобы просмотреть подробные сведения об уведомлении, пользователь открывает панель уведомлений, «вытягивая» ее сверху.


4:23 PM
Thursday, October 2
The Big Meeting
4:15 – 5:15 PM
Big Conference Room
MAP
EMAIL GUES...
New Google+ notifications
3:44 PM
Earl Liibyrd: Added you back
2
Screenshot captured.
4:22 PM
Touch to view your screenshot.
Keep photos & videos backed..
4:22 PM
Touch to get free private storage on Google
3 new messages
4:11 PM
kumlatar.swankatranami@gmail.com
3
3 new messages
2:42 PM
(754) 263-8267, 456

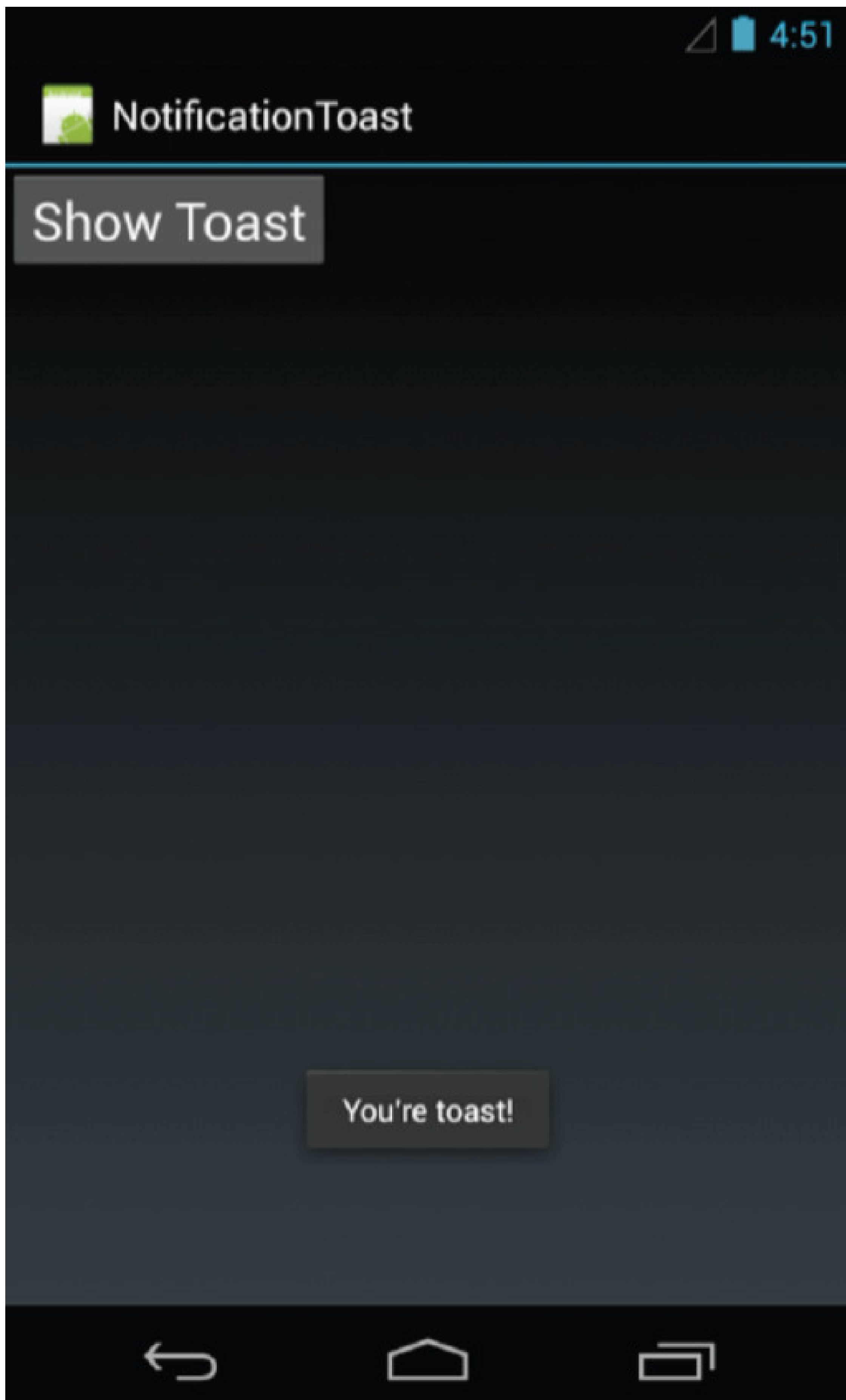
Строкой состояния и панелью уведомлений управляет система, а пользователь может их просматривать в любое время. Такие уведомления обычно используются, чтобы объявить о непредсказуемых событиях и сделать это способом, который не прерывает текущую деятельность пользователя независимо от того, что он делает в этот момент. Например, о входящей смс. Появление значка в области уведомлений говорит о том, что сообщение прибыло. Но это делается ненавязчивым способом – сообщение здесь, и вы свободны прочесть его тогда, когда это удобно для вас.

Итак, Android обеспечивает несколько различных видов пользовательских уведомлений. Эти уведомления – сообщения пользователю, которые отправляются по мере необходимости, и появляются они вне пользовательского интерфейса приложения. То есть и текст смс, который вы редактируете, и кнопка, которую вы нажмете, чтобы отправить сообщение, всегда видимы, пока вы сочиняете сообщение.

Тост-сообщения являются кратковременными сообщениями, которые разворачиваются на дисплее, например, чтобы дать пользователю знать об успешном завершении операции. Тосты автоматически постепенно появляются и постепенно исчезают. Их задача – предоставить информацию пользователю. Но они не предназначены для того, чтобы собирать информацию для передачи её обратно в приложение.

Для создания тоста, используется метод makeText класса Toast. Этот метод имеет два параметра: текст, который вы хотите вывести на экран, и количество времени, в течение которого текст должен быть видимым. После того, как вы создали тост, вы можете вывести его на экран, вызвав метод Toast.show ().

Теперь давайте посмотрим на пример приложения, которое использует тост-сообщения. Это приложение выводит на экран единственную кнопку, подписанную «Show Toast». Если нажать эту кнопку, то в нижней части экрана вы увидите маленькое раскрывающееся окно, которое говорит: «You’re toast!».


4:51
NotificationToast
Show Toast
You're toast!

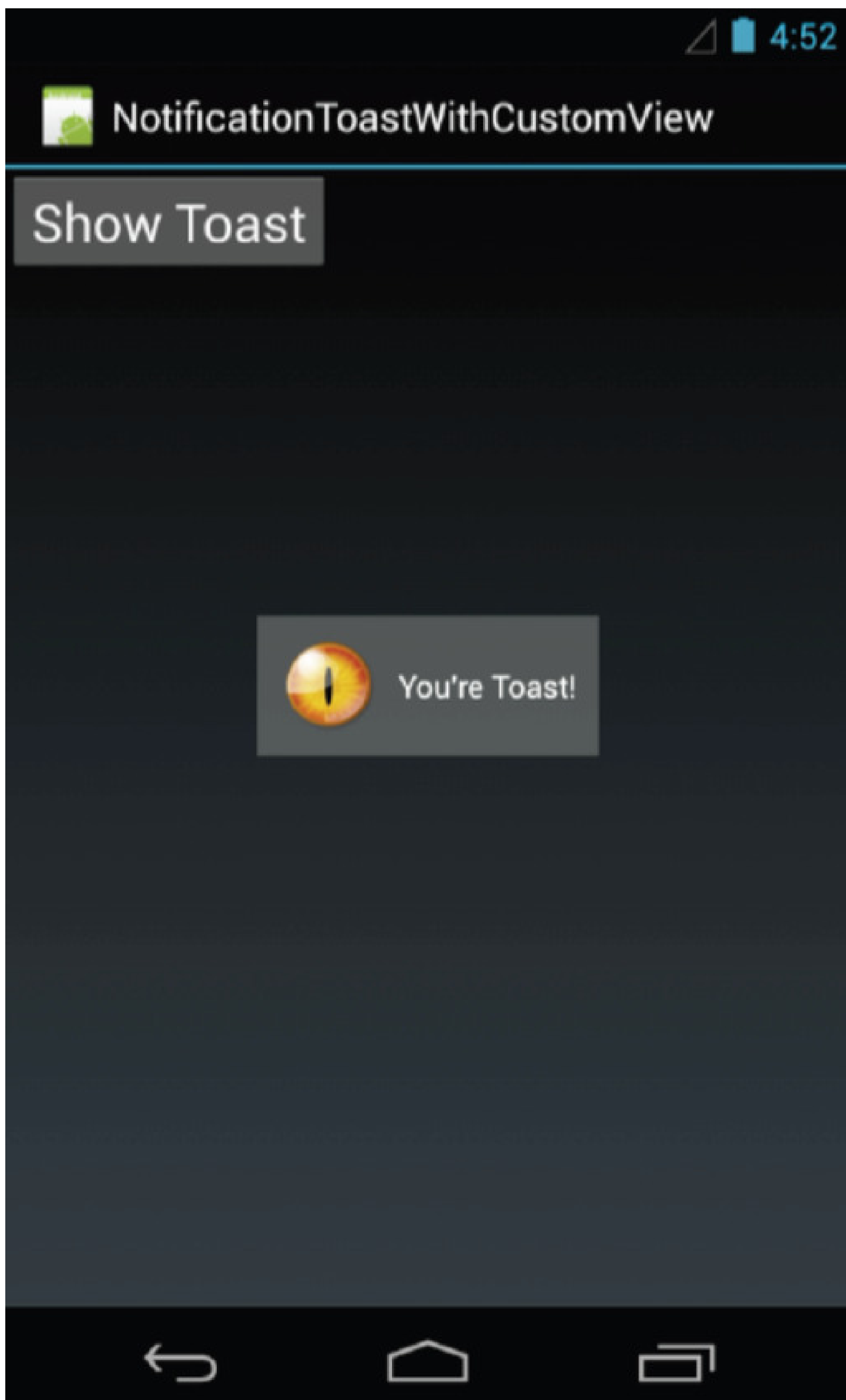
Теперь, если открыть исходный код приложения, мы увидим, как это реализовано.

```
@Override
public void onClick(View v) {
    Toast.makeText(getApplicationContext(), "You're toast!", Toast.LENGTH_LONG).show();
}
```

Здесь вы видите OnClickListener для кнопки «Show Toast», а внутри – вызов метода makeText, передающий текст и константу Toast. LENGTH_LONG, которая делает текст видимым в течение приблизительно трех с половиной секунд. И в конце строки происходит вызов метода show (), который и выводит на экран тост.

Если вам не нравится стандартный вид тоста, вы можете создать кастом вью (пользовательское вью) для своего тоста. Например, вы можете создать кастом лейаут в XML и применить его, а затем присоединить созданную вью к тост-сообщению, вызвав метод setView.

Давайте посмотрим пример. Он во многом выглядит как предыдущий, но с той разницей, что здесь на экран выводится тост, созданный из пользовательской кастом вью.


4:52
NotificationToastWithCustomView
Show Toast
You're Toast!

Теперь давайте откроем основную Activity этого приложения и посмотрим, как был создан этот тост. Видно, что когда нажимается кнопка «Show Toast», код сначала создает новый объект тост-сообщения.

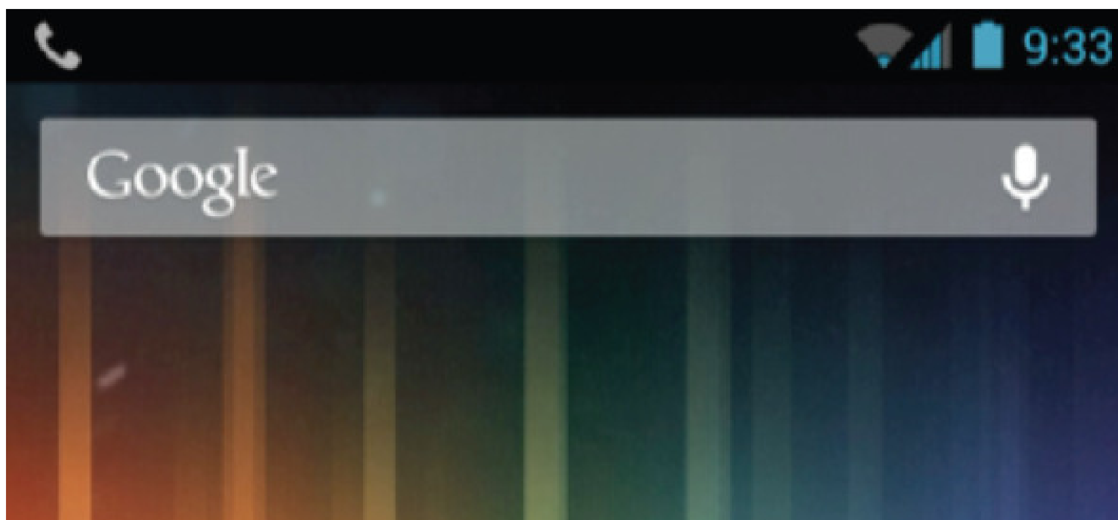
```
@Override
public void onClick(View v) {

    Toast toast = new Toast(getApplicationContext());

    toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
    toast.setDuration(Toast.LENGTH_LONG);

    toast.setView(getLayoutInflater().inflate(R.layout.custom_toast,null));

    toast.show();
}
```

Следующие две строки устанавливают расположение тоста на экране и определяют отрезок времени, в течение которого сообщение будет видимым. Далее идет вызов setView, в котором первый параметр – результат применения XML-лейаута, который находится в пользовательском файле custom_toast. xml. В этом файле находится relativeLayout, содержащий два дочерних элемента. Первый – вью изображения, которое вы видите в тосте рядом с текстом. Второй – текстовое вью, которое выводит на экран текст «You're toast!».

Затем, возвращаясь назад в основную Activity, в заключительной строке мы видим метод show (), который выводит созданный тост на экран.

Другой вид пользовательского уведомления, который мы рассмотрим – это уведомления, которые появляются в области уведомлений (status bar). Приложения и сама система Android могут использовать эту область, чтобы сообщить пользователю о возникновении различных событий. Область уведомлений также предоставляет такой элемент пользовательского интерфейса, как панель (drawer), которую пользователь может открыть, вытянув из области уведомлений сверху. И если её открыть, вы увидите дополнительную информацию о различных уведомлениях, которые были помещены в область уведомлений.

Давайте рассмотрим пример того, как эти уведомления используются в Android. Откроем приложение Телефон и наберем телефонный номер. Телефон начнет вызов и соединит с вызываемым абонентом. Теперь, если посреди этого телефонного вызова, понадобится получить некоторую информацию из интернета, можно нажать кнопку «Домой», чтобы вернуться к домашнему экрану и оттуда открыть приложение браузера. В верхнем левом углу экрана устройства, появится новый значок, это – уведомление.

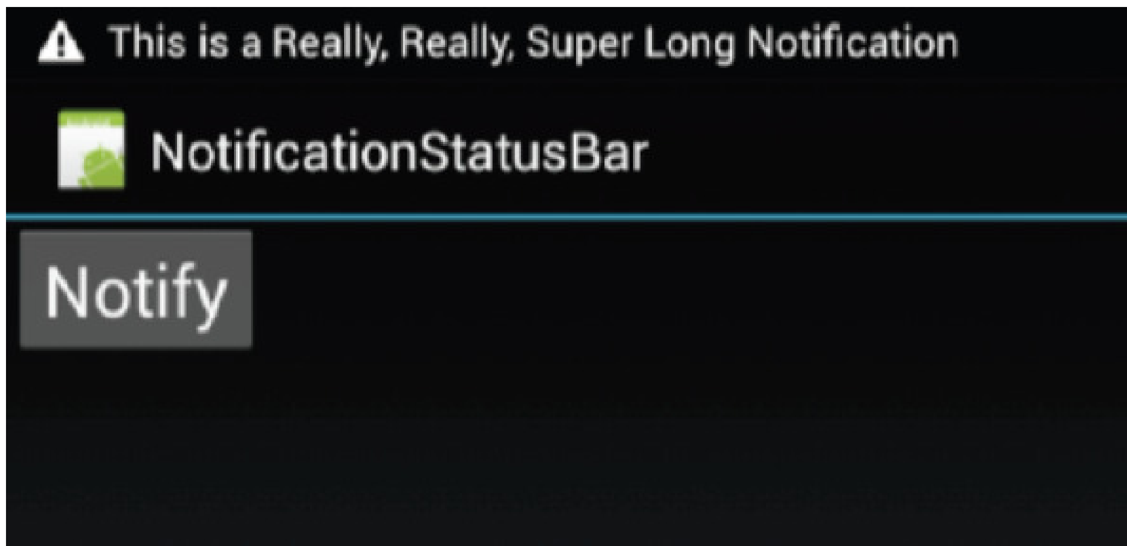


Когда пользователь отлучился из телефонного приложения, Android создал объект уведомления и поместил его в область уведомлений. Это уведомление служит напоминанием, что телефонное соединение все еще активно, а также служит способом быстро вернуться к этому телефонному вызову. Но в это время пользователь запустил браузер, где перешёл на www.google.com и выполняет поиск. Затем вооруженный информацией, в которой нуждался, он хочет вернуться к телефонному вызову. Тогда он вытягивает панель из области уведомлений, чтобы увидеть вьюшку, которая содержит некоторую информацию о вызове. Это позволяет ему либо вернуться к разговору, либо повесить трубку. Если он хочет продолжить разговор, то он кликнет по области уведомлений, и это переведёт телефонное приложение в рабочее состояние, вернет его на передний план и позволит пользователю продолжать разговор.

Если ваше приложение должно отправлять уведомления, вам придется предусмотреть несколько вещей. Во-первых, само уведомление, у которого должны быть, по крайней мере, заголовок, текст содержимого, и маленький значок – иконка. Когда уведомление будет отправлено, оно в конечном счете появится в области уведомлений, где эта иконка и будет выведена на экран. Кроме того, вы можете установить текст тикера уведомления, тогда этот текст будет также выведен на экран вместе с иконкой в области уведомлений. Наконец, если пользователь открывает панель уведомлений, должна быть вьюшка, которую пользователь и будет видеть в раскрытой панели. Далее вы должны будете определить какое-либо действие, которое произойдет, когда пользователь кликнет по уведомлению, вытянув панель уведомлений.

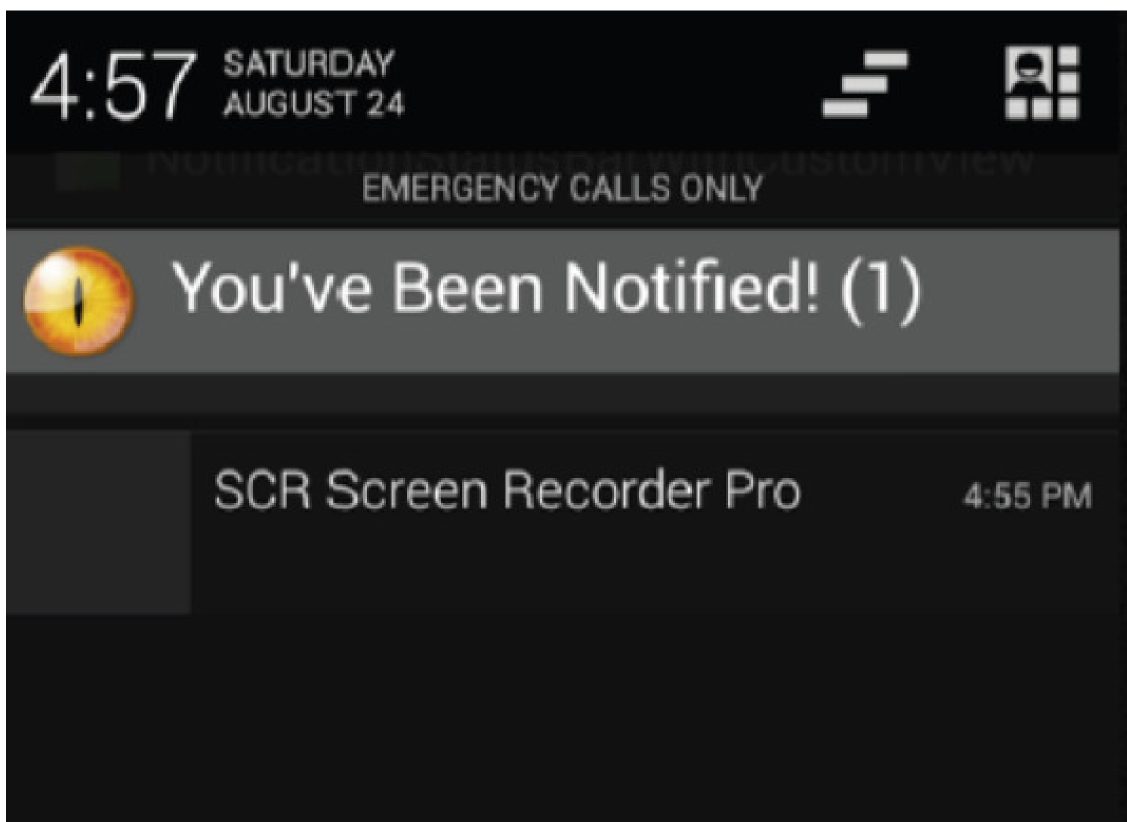
Теперь, когда вы создали уведомление, в какой-то момент вы захотите отправить его, затем обновить, отменить или что-то еще. Этими операциями управляет системная служба Android, называемая Менеджером уведомлений.

Давайте разберем два приложения, которые отправляют уведомления, и рассмотрим их исходный код, чтобы увидеть, как все это реализовано. Запустим приложение NotificationStatusBar. Пользовательский интерфейс содержит на экране единственную кнопку – «Notify». Если нажать эту кнопку, будет создано и отправлено уведомление и, в конечном счете, оно появится в панели уведомлений наверху экрана телефона.



Итак, при нажатии на кнопку прозвучал звук, присоединенный к уведомлению и в верхней части экрана появилась бегущая строка с текстом уведомления. По окончанию прокручивания бегущей строки текст пропадает, а на его месте остается иконка уведомления.

А теперь откроем панель уведомлений. Вы видите, что вьюшка уведомления показывает иконку, текст заголовка уведомления, подробный текст уведомления, который показывает номер один в круглых скобках, указывая, что кнопка «Notify» была нажата один раз. И имеется также метка времени. Если закрыть панель уведомлений и нажать кнопку «Notify» еще раз, то текст уведомления будет обновлен, чтобы показать, что кнопка опять была нажата.



Далее, если кликнуть по самому уведомлению, вы увидите, что запустилась новая Activity, напечатав слова: «Got the Intent». Смысл в том, что вы можете присоединить Intent к вьюшке в панели уведомлений, чтобы переключить пользователя на приложение (или Activity), которое должно обработать то действие, о котором говорилось в уведомлении.

Второе приложение об уведомлениях в панели уведомлений с использованием кастомных вью делает то же самое, что и предыдущий пример, только показывает пользовательскую вьюшку, когда открыта панель уведомлений. Здесь также есть изображение глаза и слова, о том, что вы были уведомлены с числом в круглых скобках. И наконец, если кликнуть по этой вью, то будет запущено новое Activity, выводящее на экран слова «Got the Intent».

Теперь давайте рассмотрим код приложения с кастомной вью. Откроем основную Activity приложения.

```
// Notification ID to allow for future updates
private static final int MY_NOTIFICATION_ID = 1;

// Notification Count
private int mNotificationCount;

// Notification Text Elements
private final CharSequence tickerText = "This is a Really, Really, Super Long Notification Message!";
private final CharSequence contentTitle = "Notification";
private final CharSequence contentText = "You've Been Notified!";

// Notification Action Elements
private Intent mNotificationIntent;
private PendingIntent mContentIntent;

// Notification Sound and Vibration on Arrival
private Uri soundURI = Uri
        .parse("android.resource://course.examples.Notification.StatusBar/"
            + R.raw.alarm_rooster);
private long[] mVibratePattern = { 0, 200, 200, 300 };
```

Начиная сверху, код создает ID для уведомления, которое будет отправлено. Это позволит менеджеру уведомлений обновлять это уведомление, после того, как оно было отправлено. И затем, имеются некоторые переменные, которые содержат текстовые элементы уведомления, включая его текст тиккера (бегущей строки), заголовок и содержание. После этого, код делает некоторые установки, которые используются, чтобы проиграть звук и включать вибрацию, когда прибывает уведомление.

Затем, код создает пользовательскую кастом вью, которая будет выведена на экран в панели уведомлений.

```
RemoteViews mContentView = new RemoteViews(
        "course.examples.Notification.StatusBarWithCustomView",
        R.layout.custom_notification);
```

Лейаут этой вью находится в файле custom_notification. xml. Эта вью представляет собой linear layout с двумя дочерними вью. Первый – изображение глаза. А другой – текстовая вью, которая выводит на экран текст «You’ve been notified».

Далее в основной Activity идет метод onCreate. Здесь код создает Intent, названный mNotificationIntent, который явно активирует sub-Activity уведомления.

```
mNotificationIntent = new Intent(getApplicationContext(),
        NotificationSubActivity.class);
mContentIntent = PendingIntent.getActivity(getApplicationContext(), 0,
        mNotificationIntent, Intent.FLAG_ACTIVITY_NEW_TASK);
```

Следующая строка кода это неописанный прежде объект – PendingIntent, который основан на известном mNotificationIntent, созданный на предыдущей строке. PendingIntent описывает интент (Intent) и действие, которое надо с ним выполнить, позволяет стороннему приложению выполнять определенный код вашего приложения с правами, которые определены для вашего же приложения. То есть позволяет стороннему приложению, в которое его передали, запустить хранящийся внутри него интент, от имени того приложения (и теми же с полномочиями), передавшего этот PendingIntent.

Идем дальше, «слушатель» кнопки «Notify» сначала обновляет текст содержимого, который указывает число нажатий кнопки.

```
final Button button = (Button) findViewById(R.id.button1);
button.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {

        // Define the Notification's expanded message and Intent:

        mContentView.setTextViewText(R.id.text, contentText + " ("
                + ++mNotificationCount + ")");
```

Затем он создает само уведомление, используя класс notification. builder. Код создает новый объект notification. Builder, устанавливает текст тиккера (бегущей строки), устанавливает маленькую иконку и затем устанавливает параметр auto cancel равным true. Это укажет системе отменить уведомление, когда пользователь кликнет по вьюшке в панели уведомлений. Затем устанавливается интент, тот самый PendingIntent, который определяет какое действие произвести, когда пользователь кликнет по вьюшке в панели уведомлений. Далее устанавливаются звуки и образцы вибрации, которые должны быть воспроизведены, когда прибывает уведомление. И наконец, устанавливается пользовательская вью, которая должна быть выведена на экран, когда пользователь раскрывает панель уведомлений.

```
// Build the Notification

Notification.Builder notificationBuilder = new Notification.Builder(
        getApplicationContext())
    .setTicker(tickerText)
    .setSmallIcon(android.R.drawable.stat_sys_warning)
    .setAutoCancel(true)
    .setContentIntent(mContentIntent)
    .setSound(soundURI)
    .setVibrate(mVibratePattern)
    .setContent(mContentView);
```

Теперь, когда уведомление установлено, код получает ссылку на менеджера уведомлений вызовом getSystemService, передавая ID службы уведомлений. И наконец, код вызывает метод notify менеджера уведомлений, передавая ID уведомления в качестве параметра.

```
// Pass the Notification to the NotificationManager:
NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(MY_NOTIFICATION_ID,
        notificationBuilder.build());
```

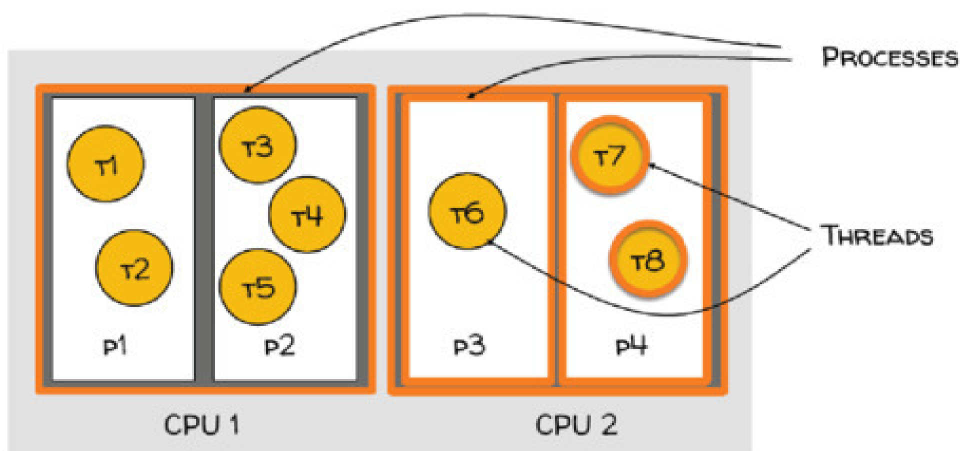
А метод build непосредственно и генерирует тот самый объект уведомления.

Потоки

Мобильные устройства, как и компьютеры, сегодня все чаще содержат по несколько вычислительных ядер. Это означает, что несколько программ могут работать на этих устройствах одновременно. И это – мощная вещь, потому что она позволяет производить больше вычислений за меньшее время. Но это также сделает ваши программы намного более сложными, что может привести к ошибкам и проблемам производительности, если вы, как разработчик, не будете внимательны и осторожны.

Для одновременно работающих программ и параллельно производимых вычислений используются Потоки – Threads. Концептуально поток – одно из многих возможных вычислений, работающих одновременно в операционной системе. С точки зрения реализации каждый поток имеет свой собственный счетчик команд и стек времени исполнения, но совместно использует «кучу» (структура данных, с помощью которой реализована динамически распределяемая память приложения) и области статического ЗУ с другими работающими потоками.

Диаграмма, изображающая эти понятия.



Здесь показано гипотетическое вычислительное устройство. У этого устройства есть два процессора, CPU 1 и CPU 2. Каждый из этих процессоров может выполнять инструкции приложений, работающих на устройстве. Далее на CPU 2 показаны два выполняющихся процесса – p3 и p4. Один из вариантов – рассматривать процессы как автономные среды исполнения. У них есть ресурсы, такие как память, открытые файлы, сетевые соединения и другие вещи, которыми они управляют и разделяют с другими процессами на устройстве. И в одном из этих процессов, p4, показаны два работающих потока – т7 и т8. Каждый из этих потоков – последовательно выполняющийся ряд инструкций со своим собственным стеком вызовов. Но так как они работают в одном и том же процессе, каждый из них может получить доступ к совместно используемым ресурсам этого процесса, включая статические переменные.

В Java потоки представлены объектом типа thread в пакете java.lang. Потоки в Java реализуют интерфейс runnable. Это означает, что у них должен быть открытый (public) метод называющийся run, который не получает параметров и у него нет возвращаемого значения. Среди методов потоки имеют метод start для запуска потока и метод sleep для того, чтобы временно

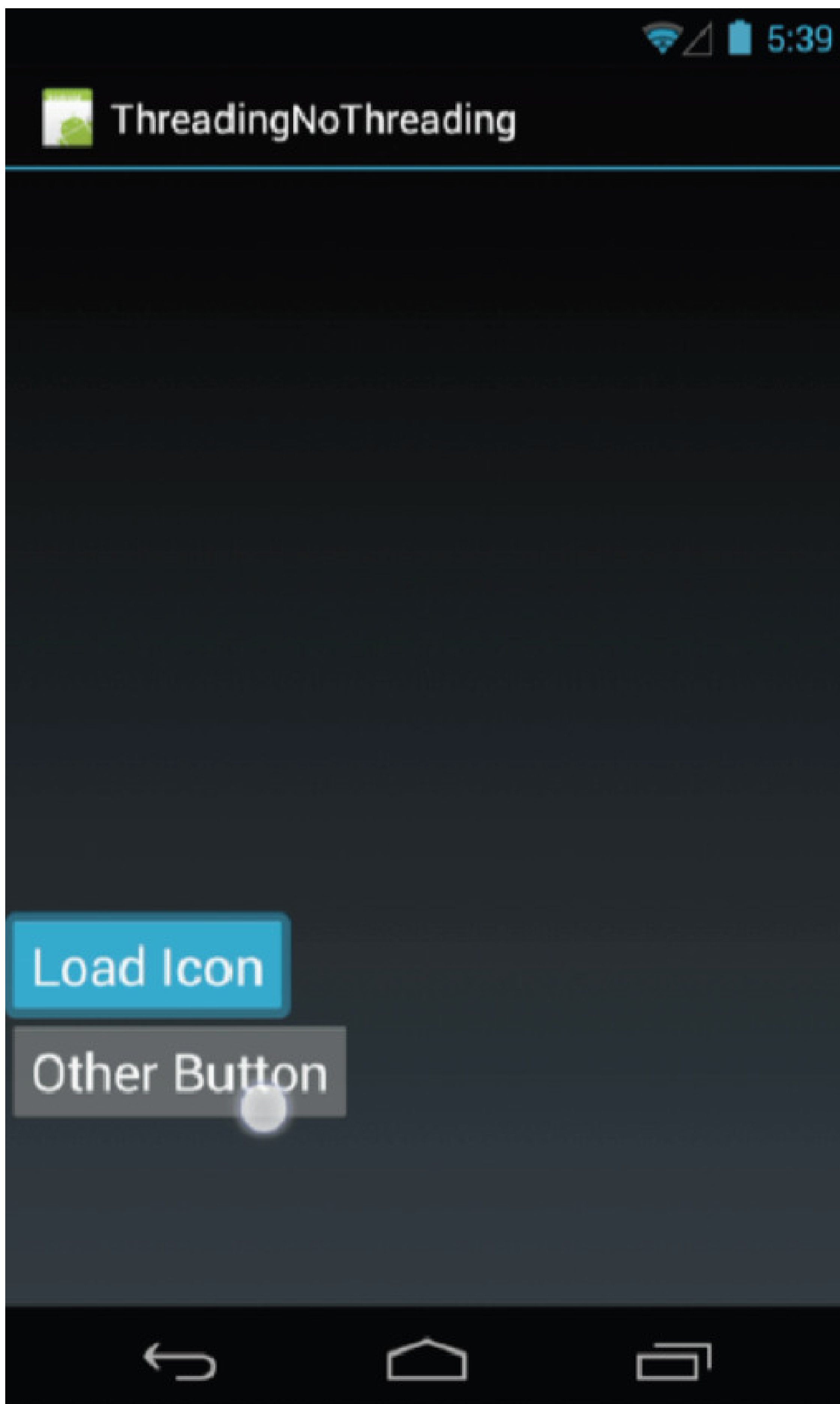
приостановить выполнение потока. Метод wait заставляет поток приостановиться и ждать пока другой поток не разбудит его методом notify.

Чтобы использовать поток, во-первых, необходимо создать поток. Например, при помощи команды new. Далее, потоки автоматически не запускаются, чтобы запустить поток необходимо вызвать метод start. И, в конечном счете происходит вызов метода run, который будет выполняться до тех пор, пока не будет остановлен.

Работающее приложение дает команду new, чтобы создать объект – новый поток. Далее работа приложения продолжается, и когда-то позже произойдет вызов метода запуска потока start. При этом произойдет возврат к приложению, но параллельно начнет выполняться код в методе потока run. И, поскольку программа продолжается, то теперь одновременно выполняются два потока. Таким образом, проделав это многократно, можно создать и выполнять столько потоков, сколько необходимо.

Рассмотрим приложение, в котором поточная обработка была бы полезна, ThreadingNoThreading. Приложение выводит на экран простой пользовательский интерфейс с двумя кнопками. Первая кнопка маркирована «LoadIcon». Когда пользователь кликает по этой кнопке, приложение открывает и читает файл, содержащий изображение, и затем показывает его на дисплее. Идея здесь состоит в том, что эта работа может занимать значительное количество времени. В данном примере количество времени несколько преувеличено для наглядности. Некоторые операции занимают относительно большое количество времени. И разработчик должен это понимать и правильно манипулировать этим.

Вторая кнопка маркирована «Other Button». Когда пользователь кликает по этой кнопке, раскрывается тост-сообщение, выводя на экран некоторый текст. А здесь идея в том, что, если вы видите текст, значит кнопка работает. Соответственно, если вы не можете нажать кнопку или не видите текст, значит что-то работает неправильно.


5:39
ThreadingNoThreading
Load Icon
Other Button

В идеале пользователь должен быть в состоянии нажать любую из кнопок в любое время и получить реакцию, система просто должна работать. Давайте запустим версию этого приложения, которое не использует поточной обработки. Нажмем кнопку «Other Button», и на дисплее появляется обещанное сообщение.

А теперь поступим иначе. Сначала нажмем кнопку «Load Icon», которая запустит трудоемкую работу чтения битового массива из файла и вывода изображения на экран. И сразу же после нажатия кнопки «Load Icon», не дожидаясь появления картинки, нажмем кнопку «Other Button». И что же тогда произойдет? Кажется кнопка «Other Button» не работает. Почему? Потому что при попытке нажать «Other Button», Android все еще загружал картинку после нажатия кнопки «Load Icon» и это препятствовало тому, чтобы вторая кнопка сработала.

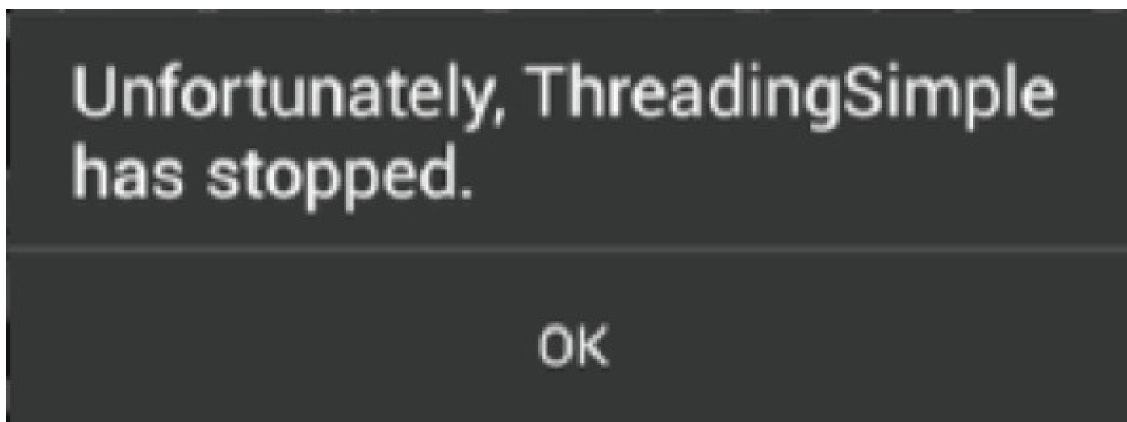
У этой проблемы есть одно на вид очевидное, но, в конечном счете, неправильное решение – перейти к «слушателю» кнопки «Load Icon» и просто создать новый поток, который будет загружать битовый массив и затем выведет его на экран. Пример такой реализации – приложение Threadingsimple, вот его код.

```
public class SimpleThreadingExample extends Activity {

    private static final String TAG = "SimpleThreadingExample";

    private Bitmap mBitmap;
    private ImageView mIView;
    private int mDelay = 5000;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mIView = (ImageView) findViewById(R.id.imageView);

        final Button loadButton = (Button) findViewById(R.id.loadButton);
        loadButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                loadIcon();
            }
        });

        final Button otherButton = (Button) findViewById(R.id.otherButton);
        otherButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(SimpleThreadingExample.this, "I'm Working",
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
```

«Слушатель» кнопки «Load Icon» вызывает метод loadIcon (), который показан чуть ниже. Этот код создает новый поток, который занимает время, загружая битовый массив. И затем пытается установить изображение в image view, который является частью лейаута.

```
private void loadIcon() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(mDelay);
            } catch (InterruptedException e) {
                Log.e(TAG, e.toString());
            }
            mBitmap = BitmapFactory.decodeResource(getResources(),
                    R.drawable.painter);

            // This doesn't work in Android
            mIView.setImageBitmap(mBitmap);
        }
    }).start();
}
```

Теперь, если запустить приложение и нажать кнопку «Load Icon», а затем сразу нажать «Other Button», видно, что есть реакция на «Other Button» и её нажатие не блокируется. Это хорошо, есть прогресс. Однако появилась большая проблема, через несколько секунд приложение рухнуло.



Если исследовать журнал logcat, мы увидим, что есть сообщение о том, что только тот поток, в котором создана иерархия вьюшек, может манипулировать своими вьюшками.

```
course.examples.T… AndroidRuntime FATAL EXCEPTION: Thread-71
course.examples.T… AndroidRuntime android.view.ViewRootImpl$CalledFromWrongThreadException: Only the
                                  original thread that created a view hierarchy can touch its views.
```

Это означает, что новый поток, который мы создали, чтобы загрузить битовый массив, может сделать эту работу, но он не сможет сделать последний шаг и вывести получившееся изображение на дисплей.

И так, какой же поток на самом деле создал иерархию вьюшек этого приложения? У всех Android приложений есть основной поток, который также называют UI thread – потоком пользовательского интерфейса. Все компоненты приложения, которые работают в одном и том же процессе, запущенном по умолчанию, используют тот же поток – UI thread. Все методы жизненного цикла: OnCreate, OnStart и т.д., работают в основном потоке. И, кроме того, сам инструментарий пользовательского интерфейса не ориентирован на многопотоковое исполнение. Все это означает, что, если вы заблокируете UI-поток любой продолжительной работой, это воспрепятствует реакции приложения на другие манипуляции пользователя. Следовательно, долго выполняющиеся операции должны быть помещены в фоновые потоки. Однако, вместе с этим, мы не сможем получить доступ к инструментарию пользовательского интерфейса от потока, не принадлежащего UI tread. Таким образом, мы должны делать длительную работу в фоновом потоке, но когда эта работа сделана, мы должны сделать обновление пользовательского интерфейса в потоке UI.

И Android на самом деле дает нам набор средств сделать это. В частности Android обеспечивает несколько методов, которые гарантированно будут работать в потоке пользовательского интерфейса. Два из них – это метод post класса View и runOnUiThread класса Activity. Оба этих метода получают параметр Runnable, содержащий код, который, например, обновляет экран в наших недавних примерах. Таким образом, если мы используем эти методы, мы могли бы загрузить битовый массив в фоновом потоке, и когда эта операция завершится, мы использовали бы один из этих методов, чтобы выполнить Runnable, который и выведет изображение на экран.

Давайте рассмотрим, как это реализуется в коде. Откроем основную Activity этого приложения и перейдем прямо к методу loadIcon, который вызывается, когда пользователь нажимает кнопку «Load Icon». Как и в предыдущем примере, этот код создает новый поток и затем загружает битовый массив. Но после загрузки данных, теперь происходит вызов view.post, запускающий новый Runnable, код которого и вызывает метод setImageBitmap, чтобы установить загруженное изображение в соответствующий imageView.

```
private void loadIcon() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(mDelay);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            mBitmap = BitmapFactory.decodeResource(getResources(),
                    R.drawable.painter);
            mImageView.post(new Runnable() {
                @Override
                public void run() {

                    mImageView.setImageBitmap(mBitmap);
                }
            });
        }
    }).start();
}
```

Следующий класс, обеспечивающий поточную обработку, это класс AsyncTask. Он обеспечивает правильное и простое использование потока пользовательского интерфейса. Этот

класс служит общей основой для управления задачами, которые, как в наших предыдущих примерах, включают операции, которые должны выполняться в фоновом потоке, а их результат публикуется в UI потоке. Основой использования AsyncTask является то, что работа разделена между фоновым потоком и потоком UI. Фоновый поток выполняет длительную операцию и может дополнительно сообщать о ее прогрессе. Поток пользовательского интерфейса, с другой стороны, ответственен за начальную настройку длительно выполняемой операции, за публикацию промежуточной информации о прогрессе и за завершение операции после того, как фоновый поток сделал свою работу.

AsyncTask – универсальный класс. Он определяется тремя основными параметрами: params, progress, и result. Params – тип параметров, которые отправляются в AsyncTask при выполнении. Progress – тип любых единиц прогресса, публикуемого в процессе выполнения фоновой операции. Result – тип результата фонового вычисления.

Во время выполнения AsyncTask проходит через четыре шага. Во-первых, onPreExecute () – метод вызывается в потоке UI до начала работы фонового потока.

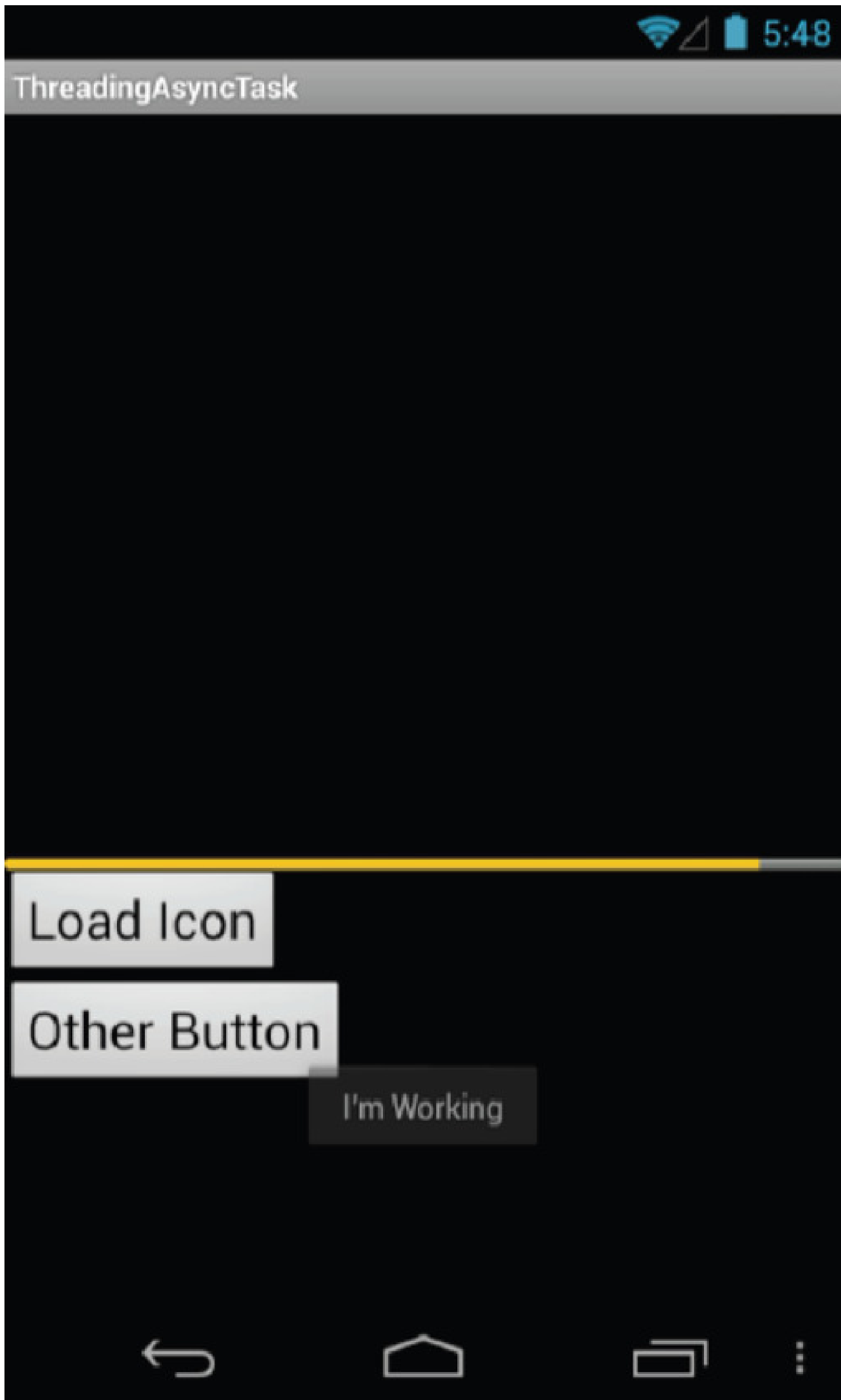
Сразу же после него запускается doInBackground (Params…) и выполняет вычисления в фоновом режиме. Сюда же передаются параметры. И этот же метод возвращает в последний шаг результат типа Result.

Во время работы doInBackground может дополнительно вызвать метод publishProgress (Progress…) с перечнем переменных, который обеспечит некоторую индикацию прогресса длительного процесса. А переданные переменные используются в UI потоке при помощи метода onProgressUpdate (Progress…), который запускается в потоке пользовательского интерфейса после вызова publishProgress.

И последний шаг, onPostExecute (Result) вызывается в UI потоке после окончания фоновых вычислений с результатом, возвращенным как его параметр.

Давайте рассмотрим версию нашего приложения загрузки картинки, реализованного с AsyncTask. Если мы запустим приложение, то увидим, что оно выглядит подобно предыдущим примерам, но здесь добавлен новый элемент интерфейса – индикатор выполнения, который показывает, какое количество работы по загрузке изображения уже выполнено.

После нажатия кнопки «Load Icon» появится маленький индикатор выполнения и начнет медленно заполняться. После нажатия второй кнопки раскроется знакомый текст – кнопка «Other Button» работает. И наконец, на экране появится изображение.


5:48
ThreadingAsyncTask
Load Icon
Other Button
I'm Working

А теперь рассмотрим исходный код этого приложения. Откроем основную Activity.

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

    mImageView = (ImageView) findViewById(R.id.imageView);;
    mProgressBar = (ProgressBar) findViewById(R.id.progressBar);

    final Button button = (Button) findViewById(R.id.loadButton);
    button.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            new LoadIconTask().execute(R.drawable.painter);
        }
    });

    final Button otherButton = (Button) findViewById(R.id.otherButton);
    otherButton.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(AsyncTaskActivity.this, "I'm Working",
                    Toast.LENGTH_SHORT).show();
        }
    });

}
```

«Слушатель» кнопки «Load Button» создает новый экземпляр LoadIconTask и сразу же происходит вызов execute с передачей в качестве параметра id ресурса изображения.

Далее рассмотрим класс LoadIconTask более подробно. LoadIconTask – это AsyncTask с параметрами: params – типа целое, progress – типа целое и result – типа битовый массив.

```
class LoadIconTask extends AsyncTask<Integer, Integer, Bitmap> {


    @Override
    protected void onPreExecute() {
        mProgressBar.setVisibility(ProgressBar.VISIBLE);
    }

    @Override
    protected Bitmap doInBackground(Integer... resId) {
        Bitmap tmp = BitmapFactory.decodeResource(getResources(), resId[0]);
        // simulating long-running operation
        for (int i = 1; i < 11; i++) {
            sleep();
            publishProgress(i * 10);
        }
        return tmp;
    }
```

Первый метод, который мы рассмотрим, это onPreExecute. Этот метод выполняется в UI потоке и его назначение – сделать индикатор выполнения видимым на дисплее.

Следующий метод – doInBackground. Этот метод получает в качестве параметра целое число – id ресурса для битового массива, который был передан в метод LoadiconTask. execute. doInBackground выполняет работу загрузки битового массива. И пока он делает это, то периодически вызывает publishProgress, передавая параметр, представляющий собой процент

загрузки, которая была проделана до этого момента. Этот пример является немного надуманным с целью упрощения понимания.

Следующий метод – onProgressUpdate. Этот метод выполняется в потоке пользовательского интерфейса, получает параметр, который был передан в PublishProgress, и затем устанавливает индикатор выполнения для отображения процента выполненной работы.

И наконец, последний метод – onPostExecute. Этот метод так же работает в UI потоке и получает только что загруженный массив (Bitmap) в качестве параметра.

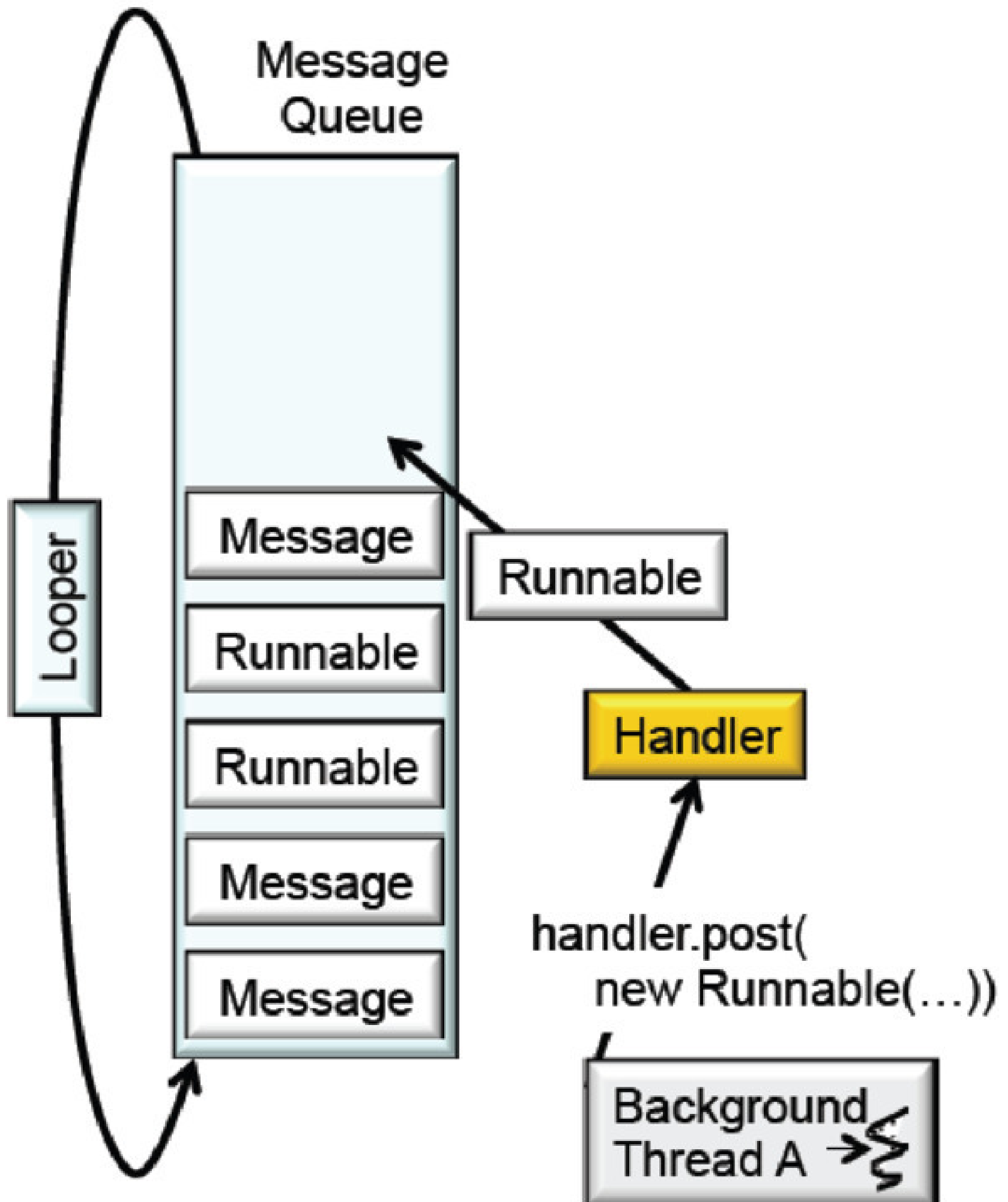
```
    @Override
    protected void onProgressUpdate(Integer... values) {
        mProgressBar.setProgress(values[0]);
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        mProgressBar.setVisibility(ProgressBar.INVISIBLE);
        mImageView.setImageBitmap(result);
    }

    private void sleep() {
        try {
            Thread.sleep(mDelay);
        } catch (InterruptedException e) {
            Log.e(TAG, e.toString());
        }
    }
}
```

Сначала onPostExecute делает индикатор выполнения невидимым, так как он больше не нужен и затем выводит загруженное изображение на экран.

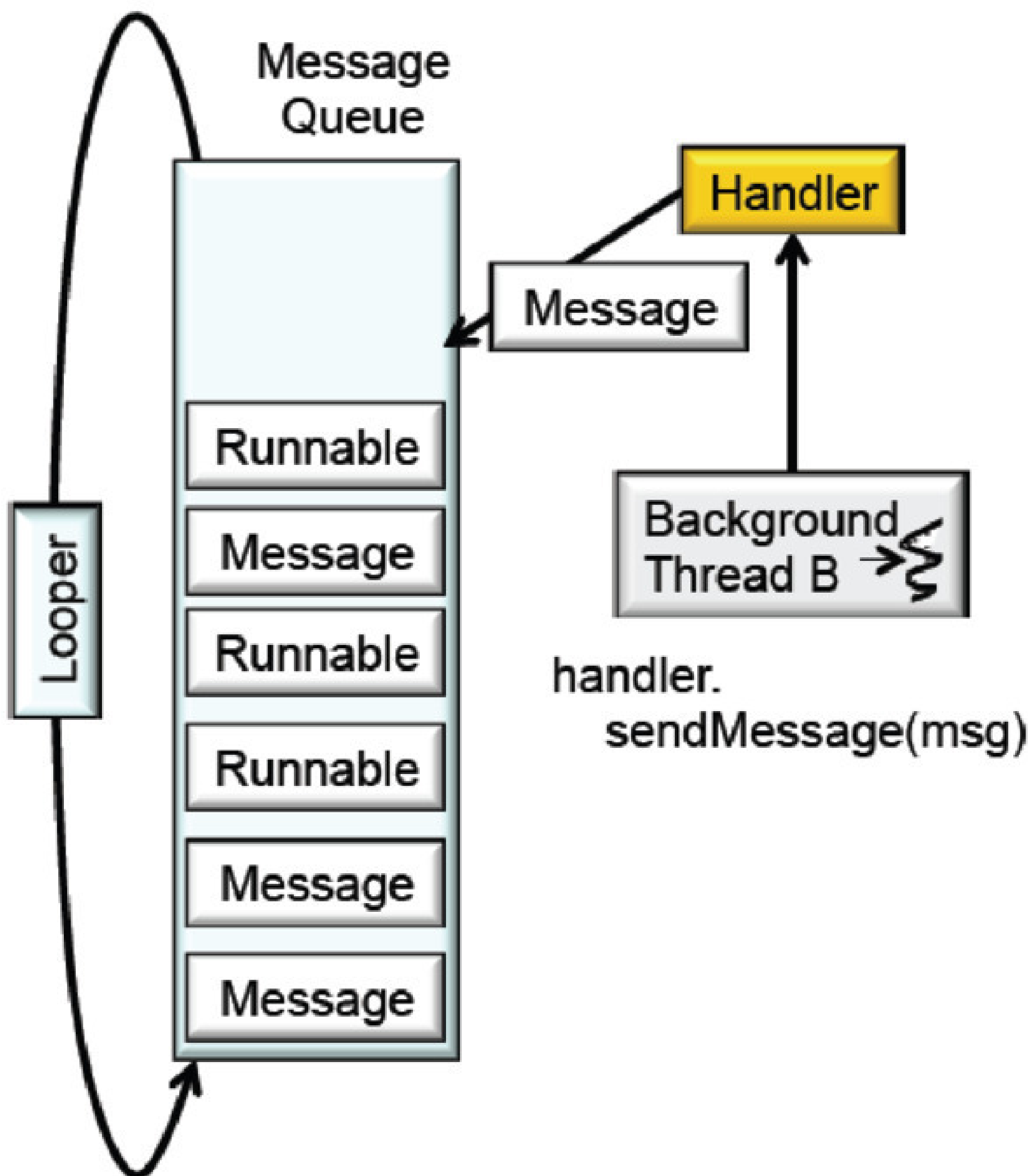
Класс Handler – это обработчик, позволяющий отправлять и обрабатывать сообщения и объекты Runnable, ассоциированные с очередью сообщений потока. Каждый его экземпляр связан с отдельным потоком и его очередью сообщений. Как и AsyncTask, класс Handler предназначен для взаимодействия между двумя потоками, но он более гибок, так как может работать с любыми двумя потоками, а не только между фоновым и потоком пользовательского интерфейса. Каждый поток может передать работу другому потоку путем отправки сообщения или объекта Runnable обработчику, который с ним ассоциирован.

Мы используем объекты Runnable, когда отправитель (поток-заказчик) точно знает какую работу и как необходимо выполнить, но выполнить её надо в потоке обработчика. С другой стороны, Сообщение – это класс, который может содержать данные, такие как код сообщения, произвольный объект данных или целочисленные аргументы. И мы будем использовать сообщения, когда поток-отправитель указывает какую операцию необходимо выполнить в другом потоке, а как её выполнить определит handler.

В Android каждый поток связан с очередью сообщений – messageQueue и петлителем – Looper. Очередь сообщений – структура данных, которая содержит сообщения и объекты runnable. Петлитель забирает эти сообщения и объекты runnable из очереди сообщений и диспетчеризирует их в зависимости от обстоятельств.



Эта диаграмма изображает поток *Tread A*, который создает Runnable внутри метода post (опубликовать) и использует объект-обработчик – Handler, чтобы отправить его в поток этого обработчика. Runnable помещается в очередь сообщений потока, связанного с обработчиком. И тоже самое происходит с сообщениями.



А эта диаграмма изображает поток *Thread B*, который создает сообщение и использует метод обработчика sendMessage, чтобы отправить это сообщение в поток обработчика. Сообщение помещается в очередь сообщений, связанную с этим обработчиком.

В то время, когда выполняются все вышеописанные операции, объект-петлитель просто «сидит» ожидая работы, которая появится в очереди сообщений. И когда эта работа появляется, петлитель реагирует одним из двух способов в зависимости от вида работы, которая только что появилась. Если эта работа – сообщение, петлитель обрабатывает это сообщение, вызывая метод обработчика handleMessage и передавая в нем само сообщение. Если же эта работа будет Runnable, то петлитель просто вызывает его метод run.

Существует несколько методов, которые позволяют запланировать выполнение работы в разное время. Например, можно использовать метод postAtTime, чтобы добавить runnable в очередь сообщений и выполнить его в определенное вами время. Метод postDelayed позволяет добавить runnable в очередь сообщений и выполнить его после указанной задержки.

Если вы хотите отправлять сообщения, сначала необходимо сообщение создать. Один из способов это сделать – использовать метод обработчика obtainMessage, который выдаст вам сообщение уже с установленным обработчиком. Можно также использовать метод obtain класса message.

Что касается runnable, то есть несколько методов, которые можно использовать, чтобы отправить сообщение. Метод sendMessage уже упоминался. Есть также его версия, которая позволяет поместить сообщение вперед в очереди сообщений, чтобы оно было выполнено как можно скорее. Есть метод sendMessageAtTime, чтобы поставить сообщение в очередь согласно требуемому времени. Есть также метод sendMessageDelayed, который поставит сообщение в очередь в текущее время плюс указанная задержка.

Давайте рассмотрим исходный код некоторых версий нашего рабочего примера, в котором были использованы обработчики. Вот приложение Threading handler runnable. Откроем основную Activity этого приложения. Во-первых, вы видите, что этот код создает новый обработчик. Этот обработчик создается в основном потоке пользовательского интерфейса. Таким образом, объекты runnable, которые получает этот обработчик, будут выполняться в UI потоке.

```
public class HandlerRunnableActivity extends Activity {

    private ImageView mImageView;
    private ProgressBar mProgressBar;
    private Bitmap mBitmap;
    private int mDelay = 500;
    private final Handler handler = new Handler();
```

Далее вы видите «слушателя» кнопки «Load Icon». Когда пользователь нажимает кнопку «Load Icon», этот код создает и запускает новый поток, метод run которой определен runnable-объектом LoadIconTask.

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

    mImageView = (ImageView) findViewById(R.id.imageView);
    mProgressBar = (ProgressBar) findViewById(R.id.progressBar);

    final Button button = (Button) findViewById(R.id.loadButton);
    button.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            new Thread(new LoadIconTask(R.drawable.painter)).start();
        }
    });
```

Метод run класса LoadIconTask начинается, создавая новый объект Runnable, который при исполнении сделает видимой шкалу прогресса.

```
private class LoadIconTask implements Runnable {
    int resId;

    LoadIconTask(int resId) {
        this.resId = resId;
    }

    public void run() {

        handler.post(new Runnable() {
            @Override
            public void run() {
                mProgressBar.setVisibility(ProgressBar.VISIBLE);
            }
        });

        mBitmap = BitmapFactory.decodeResource(getResources(), resId);
```

Далее происходит загрузка битового массива. И во время этой загрузки, периодически публикуется её прогресс методом setProgress другого runnable.

```
// Simulating long-running operation

for (int i = 1; i < 11; i++) {
    sleep();
    final int step = i;
    handler.post(new Runnable() {
        @Override
        public void run() {
            mProgressBar.setProgress(step * 10);
        }
    });
}
```

Далее создается новый Runnable, который устанавливает только что загруженный битовый массив на экран. И заканчивается все, созданием последнего Runnable, который делает шкалу прогресса невидимой.

```
            handler.post(new Runnable() {
                @Override
                public void run() {
                    mImageView.setImageBitmap(mBitmap);
                }
            });

            handler.post(new Runnable() {
                @Override
                public void run() {
                    mProgressBar.setVisibility(ProgressBar.INVISIBLE);
                }
            });
        }
    }
```

А теперь рассмотрим вторую версию этого приложения, которое отправляет сообщения вместо runnable – ThreadingHandlerMessages. Откроем основную Activity этого приложения. Во-первых, этот код создает новый обработчик. И как в предыдущем примере, этот обработчик будет создаваться основным UI потоком. И соответственно работа, которую выполняет этот обработчик, будет выполняться в UI потоке. У этого обработчика есть метод handleMessage, в котором он реализует различные виды работ. Этот метод вначале проверяет код сообщения, содержащийся в этом сообщении, и затем выполняет действие, определенное для этого кода сообщения.

```
@Override
public void handleMessage(Message msg) {
    HandlerMessagesActivity parent = mParent.get();
    if (null != parent) {
        switch (msg.what) {
        case SET_PROGRESS_BAR_VISIBILITY: {
            parent.getProgressBar().setVisibility((Integer) msg.obj);
            break;
        }
        case PROGRESS_UPDATE: {
            parent.getProgressBar().setProgress((Integer) msg.obj);
            break;
        }
        case SET_BITMAP: {
            parent.getImageView().setImageBitmap((Bitmap) msg.obj);
            break;
        }
        }
    }
}
```

Например, если код – set_progress_bar_visibility, то будет установлено состояние видимости шкалы прогресса. Если код – progress_update, то будет установлено значение прогресса на шкале прогресса. Если код – set_bitmap, то этот код выводит битовый массив на дисплей.

Теперь перейдем к «слушателю» кнопки «Load Icon». Как и в предыдущих примерах, когда пользователь нажимает кнопку, этот код, создает и запускает новый поток, чей метод run определен runnable-объектом LoadIconTask.

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

    mImageView = (ImageView) findViewById(R.id.imageView);
    mProgressBar = (ProgressBar) findViewById(R.id.progressBar);

    final Button button = (Button) findViewById(R.id.loadButton);
    button.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            new Thread(new LoadIconTask(R.drawable.painter, handler))
                    .start();
        }
    });

    final Button otherButton = (Button) findViewById(R.id.otherButton);
    otherButton.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(HandlerMessagesActivity.this, "I'm Working",
                    Toast.LENGTH_SHORT).show();
        }
    });

}
```

Выполнение этого метода начинается при получении сообщения с кодом set_progress_bar_visibility с параметром, указывающим, что шкала прогресса должна стать видимой. Тогда это сообщение отправляется в обработчик, который и сделает шкалу прогресса видимой.

Далее происходит загрузка битового массива. И одновременно с этим, периодически обновляется прогресс, при помощи сообщения с кодом progress_update и с параметром, который указывает процент выполненной работы. Это будет результатом вызова обработчиком метода setProgress объекта progressBar.

```
private class LoadIconTask implements Runnable {
    private final int resId;
    private final Handler handler;

    LoadIconTask(int resId, Handler handler) {
        this.resId = resId;
        this.handler = handler;
    }

    public void run() {

        Message msg = handler.obtainMessage(SET_PROGRESS_BAR_VISIBILITY,
                ProgressBar.VISIBLE);
        handler.sendMessage(msg);

        final Bitmap tmp = BitmapFactory.decodeResource(getResources(),
                resId);

        for (int i = 1; i < 11; i++) {
            sleep();
            msg = handler.obtainMessage(PROGRESS_UPDATE, i * 10);
            handler.sendMessage(msg);
        }

        msg = handler.obtainMessage(SET_BITMAP, tmp);
        handler.sendMessage(msg);

        msg = handler.obtainMessage(SET_PROGRESS_BAR_VISIBILITY,
                ProgressBar.INVISIBLE);
        handler.sendMessage(msg);
    }

    private void sleep() {
        try {
            Thread.sleep(mDelay);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Затем принимается и отправляется сообщение, чтобы вывести только что загруженное изображение на дисплей. И наконец, отправляется последнее сообщение, чтобы сделать шкалу прогресса невидимой.

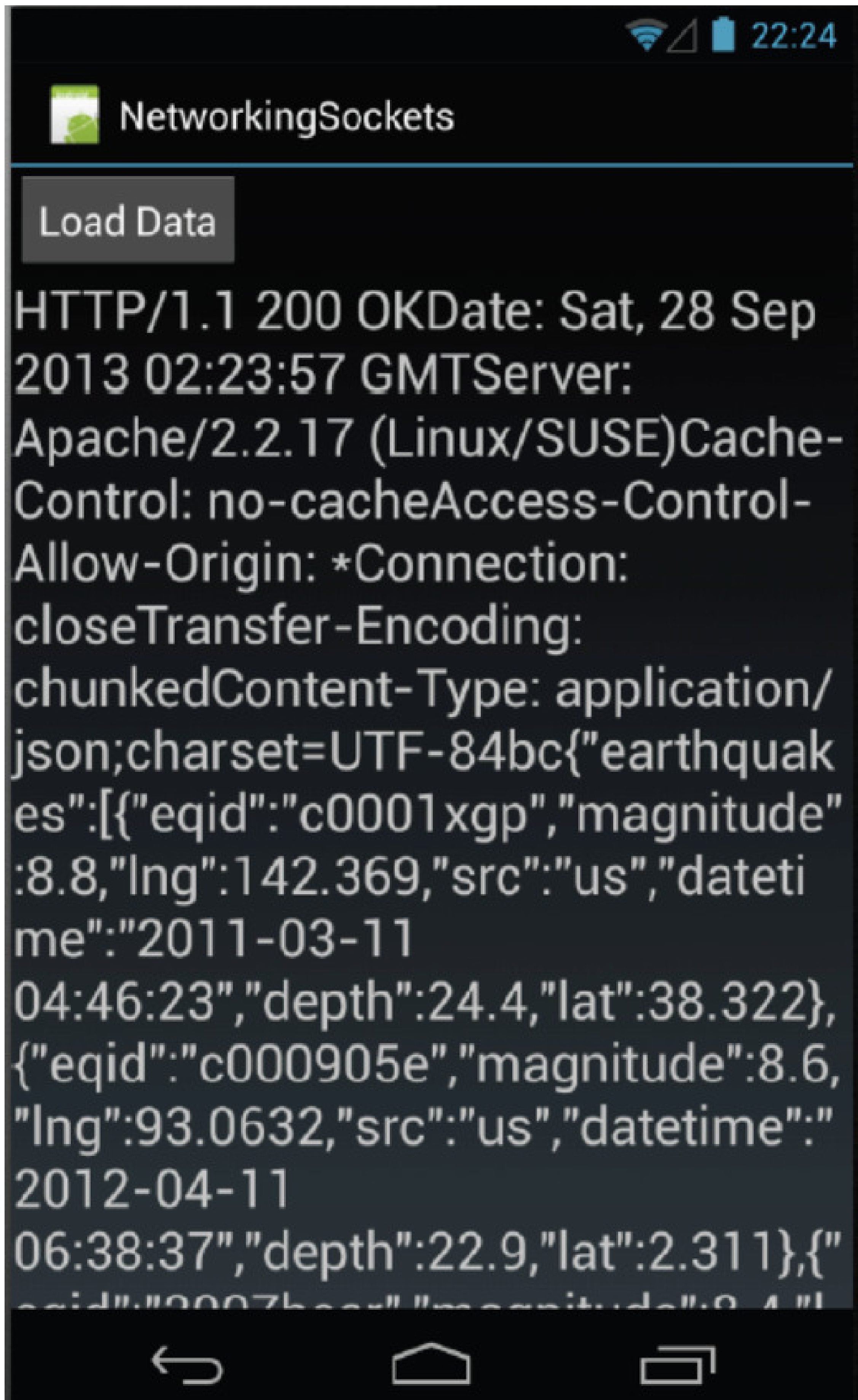
Работа с сетью

Одной из определяющих характеристик современных мобильных устройств является то, что они могут обеспечить нас связью и сетевым подключением, то есть он-лайн, не привязывая нас к определенному местоположению. Смартфоны и планшеты комбинируют мощные процессоры с быстрым сетевым соединением по WiFi и сотовым сетям. Мобильные приложения должны уметь использовать эти сетевые возможности, чтобы обеспечить нас данными и предоставить доступ к службам. Android включает в себя поддержку множества сетевых классов при помощи пакетов Java.net, org. appache и android.net.

В текущей главе мы рассмотрим некоторые из этих классов, используя каждый из них для реализации одного и того же примера приложения. Это приложение взаимодействует с интернет-службой для получения информации о землетрясениях, произошедших в определенном географическом регионе. Чтобы заставить это приложение работать, код должен создать http-запрос, отправить его на сервер, получить результаты и затем отобразить эти результаты.

Для реализации этого, мы будем использовать три класса, это класс сокетов (Socket), класс HttpURLConnection и AndroidHttpClient.

Если запустить приложение, использующее класс socket, первоначально мы увидим одну кнопку с надписью «Load Data». И, если нажать эту кнопку, приложение отправит запрос HTTP GET на внешний сервер, и этот сервер ответит сложным текстом, содержащим запрошенные данные о землетрясениях.


22:24
NetworkingSockets
Load Data
HTTP/1.1 200 OKDate: Sat, 28 Sep 2013 02:23:57 GMTServer: Apache/2.2.17 (Linux/SUSE)Cache-Control: no-cacheAccess-Control-Allow-Origin: *Connection: closeTransfer-Encoding: chunkedContent-Type: application/json;charset=UTF-84bc{"earthquakes":[{"eqid":"c0001xgp","magnitude":8.8,"lng":142.369,"src":"us","datetime":"2011-03-11 04:46:23","depth":24.4,"lat":38.322},{"eqid":"c000905e","magnitude":8.6,"lng":93.0632,"src":"us","datetime":"2012-04-11 06:38:37","depth":22.9,"lat":2.311},{"

Давайте рассмотрим исходный код, чтобы узнать, что нужно для получения этих данных. Откроем основную Activity этого приложения, и здесь мы видим «слушателя» кнопки загрузки данных. Когда эта кнопка нажата, приложение создает и затем выполняет AsyncTask с именем HttpGetTask.

```
public class NetworkingSocketsActivity extends Activity {
    TextView mTextView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mTextView = (TextView) findViewById(R.id.textView1);

        final Button loadButton = (Button) findViewById(R.id.button1);
        loadButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                new HttpGetTask().execute();

            }
        });
    }
```

Класс HttpGetTask сначала объявляет некоторые переменные, которые используются при создании запроса HttpGet.

```
private class HttpGetTask extends AsyncTask<Void, Void, String> {

    private static final String HOST = "api.geonames.org";

    // Get your own user name at http://www.geonames.org/login
    private static final String USER_NAME = "user";
```

```
private static final String HTTP_GET_COMMAND =
```

```
"GET /earthquakesJSON?north=44.1&south=-9.9&east=-22.4&west=55.2&username="
```

```
        + USER_NAME
        + " HTTP/1.1"
        + "\n"
        + "Host: "
        + HOST
        + "\n"
        + "Connection: close" + "\n\n";

private static final String TAG = "HttpGet";
```

Первым вызывается метод doInBackground. Этот метод создает новый сокет, который будет соединен с хостом api.geonames.org по стандартному http-порту: 80.

```
@Override
protected String doInBackground(Void... params) {
    Socket socket = null;
    String data = "";

    try {
        socket = new Socket(HOST, 80);
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(
                socket.getOutputStream()), true);
        pw.println(HTTP_GET_COMMAND);

        data = readStream(socket.getInputStream());

    } catch (UnknownHostException exception) {
        exception.printStackTrace();
    } catch (IOException exception) {
        exception.printStackTrace();
    } finally {
        if (null != socket)
            try {
                socket.close();
            } catch (IOException e) {
                Log.e(TAG, "IOException");
            }
    }
    return data;
}
```

Затем код получает исходящий поток сокета, записывает в него http_get_command, и эта строка будет отправлена хосту, который интерпретирует его как запрос HTTP GET, а затем ответит, отправив обратно соответствующие данные ответа. Далее код получает входящий поток сокета, передавая его методу readStream, который считывает данные ответа из входящего потока сокета, и возвращает ответ в виде строки.

Затем эта строка передается методу onPostExecute, который выполняется в главном потоке, и который отображает ответ в текстовой вью.

```
@Override
protected void onPostExecute(String result) {
    mTextView.setText(result);
}

private String readStream(InputStream in) {
    BufferedReader reader = null;
    StringBuffer data = new StringBuffer();
    try {
        reader = new BufferedReader(new InputStreamReader(in));
        String line = "";
        while ((line = reader.readLine()) != null) {
            data.append(line);
        }


    } catch (IOException e) {
        Log.e(TAG, "IOException");
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                Log.e(TAG, "IOException");
            }
        }
    }
    return data.toString();
}
```

Вернемся к приложению. Текст ответа включает в себя не только данные землетрясения, но и заголовки ответа HTTP. Но вы, возможно, не хотели бы показывать здесь этот избыточный текст, а хотели бы вывести на экран только данные землетрясения. Поэтому вы должны будете проанализировать ответ и извлечь только те данные, которые хотели бы вывести на экран. Кроме того, здесь отсутствует код обработки ошибок, который сделал бы это приложение более надежным. Использование сокетов – это компромиссный вариант, сходный с программированием низкого уровня. Вы можете записать в сокет все, что хотите, но взамен вы должны будете обрабатывать все многочисленные детали создания HTTP-запросов, прописать всю обработку ошибок и всю обработку HTTP-ответов.

В следующей реализации используется класс HttpURLConnection. Этот класс предоставляет интерфейс более высокого уровня, который обрабатывает больше деталей сетевого соединения, чем класс сокетов, но он также имеет менее гибкий API, чем другой вариант – класс Android HTTP client. Также необходимо отметить, что команда Android больше не работает над клиентом Android HTTP, отложив улучшения этого класса на будущее.

Итак, давайте рассмотрим пример приложения, реализованного на этот раз с классом HttpURLConnection. Приложение NetworkingURL внешне выглядит и работает также как

и приложение из предыдущего примера, но после нажатия кнопки «Load Data» в текстовом выводе отсутствуют Http заголовки, они были удалены.

Давайте рассмотрим исходный код и посмотрим, как это работает. Откроем основную Activity этого приложения, и здесь мы увидим «слушателя» кнопки загрузки данных. Как и ранее, при нажатии этой кнопки приложение создаст и выполнит AsynchTask с именем httpGetTask.

```
public class NetworkingURLActivity extends Activity {
    private TextView mTextView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);
        mTextView = (TextView) findViewById(R.id.textView1);

        final Button loadButton = (Button) findViewById(R.id.button1);
        loadButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                new HttpGetTask().execute();
            }
        });
    }
```

Давайте рассмотрим этот класс. Когда в HttpGetTask срабатывает метод execute, вызывается метод doInBackground. Этот метод начинается с создания нового объекта URL и передачи строки URL-адреса для требуемой службы в качестве параметра.

```
@Override
protected String doInBackground(Void... params) {
    String data = "";
    HttpURLConnection httpUrlConnection = null;

    try {
        httpUrlConnection = (HttpURLConnection) new URL(URL)
                .openConnection();

        InputStream in = new BufferedInputStream(
                httpUrlConnection.getInputStream());

        data = readStream(in);
```

```
    } catch (MalformedURLException exception) {
        Log.e(TAG, "MalformedURLException");
    } catch (IOException exception) {
        Log.e(TAG, "IOException");
    } finally {
        if (null != httpUrlConnection)
            httpUrlConnection.disconnect();
    }
    return data;
}
```

Затем код вызывает метод openConnection для объекта URL, который возвращает Http соединение. Этот объект затем сохраняется в переменной httpUrlConnection.

Код продолжается, получая поток входящих данных (input stream) Http-соединения и передавая его в метод readStream. А метод readStream считывает данные ответа из сокета входящего потока, а затем возвращает ответ в виде строки. Однако на этот раз httpUrlConnection лишает заголовков http-ответ и обрабатывает проверку ошибок.

Далее эта строка передается методу onPostExecute, который выводит на экран ответ в текстовую вью.

```
@Override
protected void onPostExecute(String result) {
    mTextView.setText(result);
}

private String readStream(InputStream in) {
    BufferedReader reader = null;
    StringBuffer data = new StringBuffer("");
    try {
        reader = new BufferedReader(new InputStreamReader(in));
        String line = "";
        while ((line = reader.readLine()) != null) {
            data.append(line);
        }
    } catch (IOException e) {
        Log.e(TAG, "IOException");
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    return data.toString();
}
```

Третий класс – AndroidHTTPClient. Этот класс является реализацией DefaultHttpClient проекта Apache. Он позволяет многое настроить. В частности, класс разбивает транзакцию HTTP на объект запроса и объект ответа. Это дает возможность создавать подклассы, которые настраивают обработку запросов и их ответов.

Пример приложения выглядит так же, поэтому перейдем прямо к коду и посмотрим на реализацию в приложении NetworkingAndroidHttpClient.

```java
public class NetworkingAndroidHttpClientActivity extends Activity {
    private TextView mTextView = null;

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

        mTextView = (TextView) findViewById(R.id.textView1);

        final Button loadButton = (Button) findViewById(R.id.button1);
        loadButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                new HttpGetTask().execute();
            }
        });
    }
```

Откроем основную Activity этого приложения и перейдем к классу HttpGetTask. Этот класс начинается с создания нового объекта AndroidHttpClient, вызывая метод newInstance.

```java
private class HttpGetTask extends AsyncTask<Void, Void, String> {

    // Get your own user name at http://www.geonames.org/login
    private static final String USER_NAME = "USeR";

    private static final String URL =
"http://api.geonames.org/earthquakesJSON?north=44.1&south=-9.9&east=-22.4&west=55.2&username="
            + USER_NAME;
```

```java
AndroidHttpClient mClient = AndroidHttpClient.newInstance("");
```

Далее вызывается метод doInBackground, код создает объект HttpGet, передавая в запросе URL-строку.

Затем создается объект ResponseHandler – обработчик ответа. В этом случае для запроса HttpGet обработчик ответа имеет тип BasicResponseHandler, который возвращает тело ответа.

```
@Override
protected String doInBackground(Void... params) {

    HttpGet request = new HttpGet(URL);
    ResponseHandler<String> responseHandler = new BasicResponseHandler();

    try {

        return mClient.execute(request, responseHandler);

    } catch (ClientProtocolException exception) {
        exception.printStackTrace();
    } catch (IOException exception) {
        exception.printStackTrace();
    }
    return null;
}
```

И, наконец, запрос и обработчик ответа передаются в метод execute, который отправляет запрос, получает ответ, передавая его через обработчик ответа. Результат всего этого затем передается в onPostExecute, который выводит ответ в текстовую вью.

```
    @Override
    protected void onPostExecute(String result) {

        if (null != mClient)
            mClient.close();

        mTextView.setText(result);

    }
}
```

До сих пор примеры приложения запрашивали данные, а затем просто отображали эти данные в текстовой вью в том виде, в котором получили. Но эти данные имеют сложный формат и неудобны для восприятия человеку, поэтому нуждаются в дополнительной машинной обработке.

По сути, это все более популярный способ передачи текстовых данных через интернет, и многие веб-сервисы сейчас предоставляют данные в таких форматах. В частности, два формата, о которых мы будем говорить, это JavaScript Object Notation – JSON и Extensible Markup Language – XML. Рассмотрим каждый из них по отдельности.

Первый формат, это JavaScript Object Notation – JSON. Этот формат предназначен для небольших объемов и напоминает структуры данных, встречающиеся в традиционных языках программирования. Данные JSON упаковываются в два типа структур данных. Один – карты, которые являются наборами пар ключ – значение, и два – упорядоченные списки.

Теперь вернемся к примеру приложения, которое обращается к веб-службе за информацией о землетрясениях. Ответ, который вернулся, фактически был отформатирован в JSON.

```
http://api.geonames.org/earthquakesJSON?
north=44.1&south=-9.9&east=-22.4&west=55.
2&username=demo
```

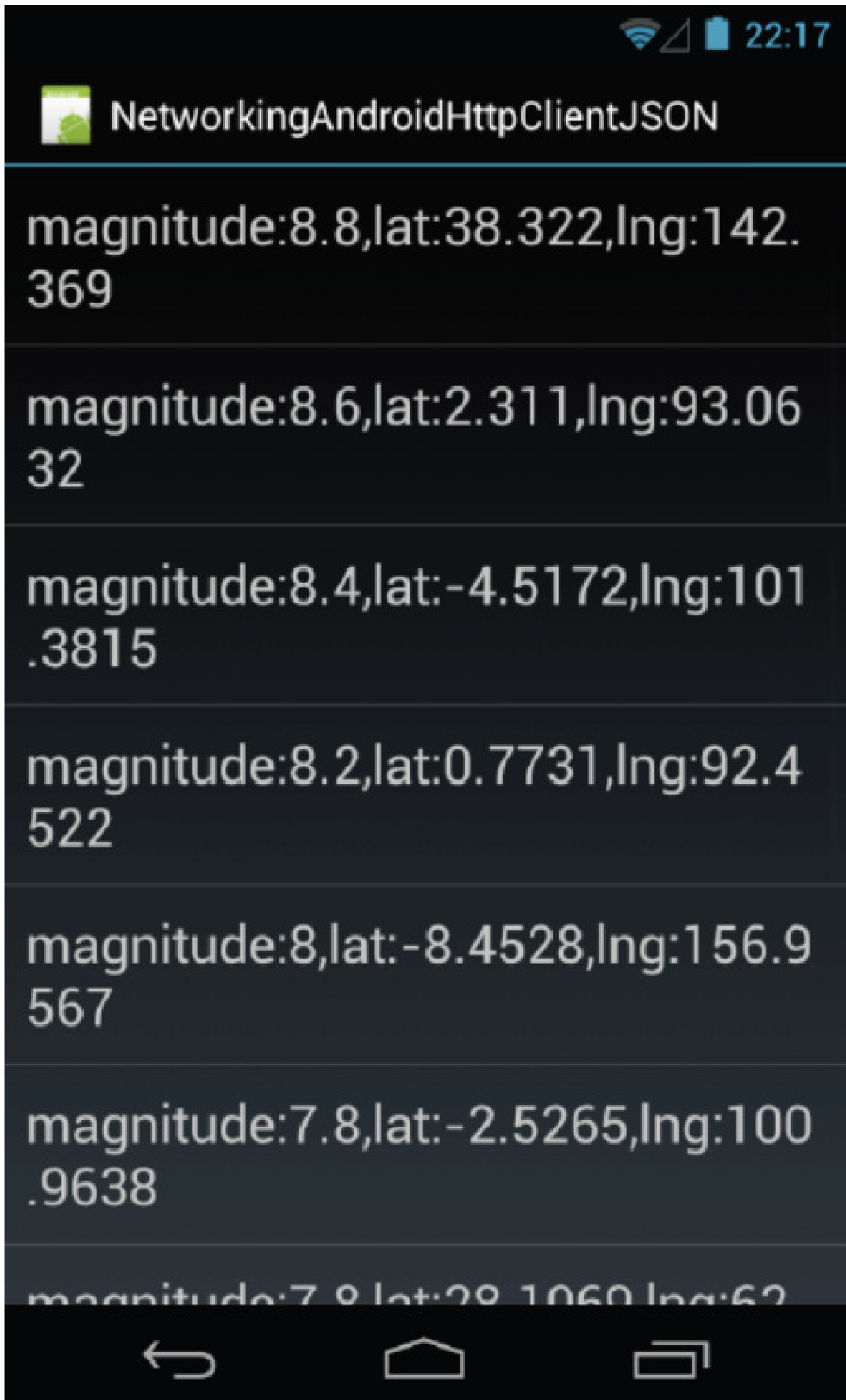
Вот эти данные, давайте разобьем их на части.


```
{"earthquakes": [
 {"eqid":"c0001xgp","magnitude":8.8,"lng":142.369,
  "src":"us", "datetime":"2011-03-11 04:46:23","depth":
   24.4,"lat":38.322}
 {"eqid":"2007hear","magnitude":8.4,"lng":101.3815,
  "src":"us","datetime":"2007-09-12 09:10:26","depth":
  30,"lat":-4.5172},
 ...
 {"eqid":"2010xkbv","magnitude":7.5,"lng":91.9379,
  "src":"us","datetime":"2010-06-12 17:26:50","depth":
  35,"lat":7.7477}
 ]
}
```


Во-первых, данные содержат один объект JSON, и этот объект является картой, которая имеет одну пару ключ – значение. Ключ называется «earthquakes», а значение – это упорядоченный список. Этот список имеет несколько объектов, и каждый из этих объектов сам является картой, содержащей пары ключ – значение.

Например. Есть ключ под названием «eqid» и его значением является id землетрясения. Далее идет ключ «magnitude» с некоторым числовым значением. Затем ключ «lng», его значение – это долгота, на которой произошло землетрясение. Кроме того, есть множество других ключей, и все вместе эти значения являются данными для одного землетрясения.

Давайте посмотрим на пример приложения, которое получает эти данные из интернета, а затем обрабатывает их так, чтобы создать более читабельный для пользователя вид. Запустим приложение NetworkingAndroidHttpClientJSON. Как и ранее, это приложение первоначально отображает единственную кнопку с пометкой «Load Data» и, как и ранее, при нажатии этой кнопки приложение отправляет HTTP-запрос на внешний сервер, и этот сервер будет отвечать сложным текстом, содержащим запрошенные данные о землетрясениях. Однако на этот раз данные будут обработаны и представлены в виде списка.


22:17
NetworkingAndroidHttpClientJSON
magnitude:8.8,lat:38.322,lng:142.369
magnitude:8.6,lat:2.311,lng:93.0632
magnitude:8.4,lat:-4.5172,lng:101.3815
magnitude:8.2,lat:0.7731,lng:92.4522
magnitude:8,lat:-8.4528,lng:156.9567
magnitude:7.8,lat:-2.5265,lng:100.9638

Давайте рассмотрим исходный код, чтобы увидеть, как это работает. Сразу переходим к классу HttpGetTask.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    new HttpGetTask().execute();
}
```

```
private class HttpGetTask extends AsyncTask<Void, Void, List<String>> {

    // Get your own user name at http://www.geonames.org/login
    private static final String USER_NAME = "USeR";

    private static final String URL =
"http://api.geonames.org/earthquakesJSON?north=44.1&south=-9.9&east=-22.4&west=55.2&username="
            + USER_NAME;
```

```
    AndroidHttpClient mClient = AndroidHttpClient.newInstance("");
```

```
@Override
protected List<String> doInBackground(Void... params) {
    HttpGet request = new HttpGet(URL);
    JSONResponseHandler responseHandler = new JSONResponseHandler();
    try {
        return mClient.execute(request, responseHandler);
    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

```
    @Override
    protected void onPostExecute(List<String> result) {
        if (null != mClient)
            mClient.close();
        setListAdapter(new ArrayAdapter<String>(
                NetworkingAndroidHttpClientJSONActivity.this,
                R.layout.list_item, result));
    }
}
```

Здесь метод doInBackground аналогичен тому, что мы видели раньше, но на этот раз он использует класс JSONResponseHandler для обработки ответа. Ключевым методом в этом классе является метод handleResponse. Этот метод начинается с передачи необработанного ответа через базовый обработчик ответа – BasicResponseHandler, который просто возвращает тело ответа без заголовков ответа HTTP.

```
private class JSONResponseHandler implements ResponseHandler<List<String>> {

    private static final String LONGITUDE_TAG = "lng";
    private static final String LATITUDE_TAG = "lat";
    private static final String MAGNITUDE_TAG = "magnitude";
    private static final String EARTHQUAKE_TAG = "earthquakes";

    @Override
    public List<String> handleResponse(HttpResponse response)
            throws ClientProtocolException, IOException {
        List<String> result = new ArrayList<String>();
        String JSONResponse = new BasicResponseHandler()
                .handleResponse(response);
```

Затем код использует JSONTokener для разбора JSON-ответа в объект Java, чтобы затем возвратить этот объекта верхнего уровня, который в данном случае является картой.

```
try {

    // Get top-level JSON Object - a Map
    JSONObject responseObject = (JSONObject) new JSONTokener(
            JSONResponse).nextValue();
```

Затем код извлекает значение, связанное с ключом землетрясения. В этом случае это упорядоченный список.

```
// Extract value of "earthquakes" key -- a List
JSONArray earthquakes = responseObject
        .getJSONArray(EARTHQUAKE_TAG);
```

Затем код перебирает список землетрясений. И для каждого элемента этого списка он получает данные, связанные с одним землетрясением, и эти данные сохранены в картах.

```
// Iterate over earthquakes list
for (int idx = 0; idx < earthquakes.length(); idx++) {

    // Get single earthquake data - a Map
    JSONObject earthquake = (JSONObject) earthquakes.get(idx);
```

Затем код суммирует различные части данных землетрясения, преобразуя их в одну строку и добавляя эту строку в список с именем result.

```
                // Summarize earthquake data as a string and add it to
                // result
                result.add(MAGNITUDE_TAG + ":"
                        + earthquake.get(MAGNITUDE_TAG) + ","
                        + LATITUDE_TAG + ":"
                        + earthquake.getString(LATITUDE_TAG) + ","
                        + LONGITUDE_TAG + ":"
                        + earthquake.get(LONGITUDE_TAG));
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

И затем, наконец, результат возвращается обратно вызывающему методу.

Второй формат данных, который мы рассмотрим – это Extensible Markup Language – XML. XML – это язык разметки для создания XML-документов. XML-документы содержат разметку и контент. Разметка кодирует описание структуры хранения в документе и логической структуры при помощи тегов и атрибутов. Контент – это все остальное. И, в частности, контент содержит данные ответа, когда XML используется для кодирования ответа HTTP.

Теперь вернемся к примеру приложения. Если мы зададим немного другой URL, то веб-сервис вернет данные землетрясения в формате XML, а не в формате JSON. Итак, вот эти данные.

```
<geonames>
 <earthquake>
  <src>us</src>

  <eqid>c0001xgp</eqid>
  <datetime>2011-03-11 04:46:23</datetime>
  <lat>38.322</lat>

  <lng>142.369</lng>
  <magnitude>8.8</magnitude>
  <depth>24.4</depth>
```

```
  </earthquake>
...
</geonames>
```

Вначале есть элемент – тег, называемый geonames. В этот элемент вложен ряд элементов землетрясения и каждый из элементов землетрясения содержит другие элементы, которые обеспечивают данные для одного землетрясения.

Подобно тому, что мы видели в формате JSON, есть элемент eqid, его значение является идентификатором землетрясения. Есть также элемент lng, его значение – долгота, на которой землетрясение произошло, и точно так же, как в примере JSON, есть и множество других элементов.

Таким образом, если приложение получает XML-данные из интернета, ему нужно будет разобрать XML-документ, чтобы создать список для вывода на экран. Для разбора XML-документов Android предоставляет несколько различных типов парсеров XML.

Парсер DOM – Document Object Model (объектный). Парсеры DOM читают весь XML-документ и преобразуют его в структуру объектной модели документа – дерево, а затем приложение обрабатывает эту древовидную структуру. Этот парсер требует больше памяти, но позволяет приложению делать многопроходную обработку документа.

SAX – Simple API for XML (событийные) парсеры читают XML-документ как поток. И когда они сталкиваются с новым тегом в документе, они производят возврат в приложение, которое и обрабатывает информацию в этом теге. Эти парсеры используют меньше памяти, чем DOM-парсеры, но они ограничены выполнением обработки за один проход документа.

Pull-парсеры, так же как и SAX-парсеры, читают документ как поток, но используют подход, основанный на итераторах, где приложение, а не парсер, решает, когда следует переходить к следующему шагу синтаксического анализа. Pull-парсеры также используют меньше памяти, чем DOM, но они в дополнение дают приложению больший контроль над процессом синтаксического анализа, чем SAX-парсеры.

Пример приложения выглядит точно так же, как тот, который мы рассмотрели при разборе ответов JSON. Поэтому перейдем к исходному коду этого приложения.

Рассмотрим сразу класс HTTPGetTask. Метод doInBackground похож на тот, что мы видели ранее. Но теперь он использует класс XML response handler для обработки ответа. Ключевым методом в этом классе является метод handleResponse, он начинается с создания объекта PullParser.

```
class XMLResponseHandler implements ResponseHandler<List<String>> {

    private static final String MAGNITUDE_TAG = "magnitude";
    private static final String LONGITUDE_TAG = "lng";
    private static final String LATITUDE_TAG = "lat";
    private String mLat, mLng, mMag;
    private boolean mIsParsingLat, mIsParsingLng, mIsParsingMag;
    private final List<String> mResults = new ArrayList<String>();

    @Override
    public List<String> handleResponse(HttpResponse response)
            throws ClientProtocolException, IOException {
        try {
```

```
// Create the Pull Parser
XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
XmlPullParser xpp = factory.newPullParser();

// Set the Parser's input to be the XML document in the HTTP Response
xpp.setInput(new InputStreamReader(response.getEntity()
        .getContent()));

// Get the first Parser event and start iterating over the XML document
int eventType = xpp.getEventType();
```

Затем код устанавливает вход парсера в качестве XML-документа, который был возвращен телом HTTP-ответа. После этого код получает первое событие парсера и затем начинает перебирать XML-документ.

```
        while (eventType != XmlPullParser.END_DOCUMENT) {

            if (eventType == XmlPullParser.START_TAG) {
                startTag(xpp.getName());
            } else if (eventType == XmlPullParser.END_TAG) {
                endTag(xpp.getName());
            } else if (eventType == XmlPullParser.TEXT) {
                text(xpp.getText());
            }
            eventType = xpp.next();
        }
        return mResults;
    } catch (XmlPullParserException e) {
    }
    return null;
}
```

Внутри цикла while есть 3 события, наличие которых этот код проверяет: стартовый XML-тег, конечный XML-тег и содержимое элемента. После определения какое событие наступило, происходит вызов соответствующего метода.

Вызывается метод startTag, в качестве параметра получая элемент, который начинается. Этот метод идентифицирует, является ли полученный элемент данных тем, который необходимо сохранить, и если это так, он сохраняет его, задавая значения определенным переменным.

```
public void startTag(String localName) {
    if (localName.equals(LATITUDE_TAG)) {
        mIsParsingLat = true;
    } else if (localName.equals(LONGITUDE_TAG)) {
        mIsParsingLng = true;
    } else if (localName.equals(MAGNITUDE_TAG)) {
        mIsParsingMag = true;
    }
}
```

Метод endTag получает в качестве параметра заканчивающий элемент. И этот метод так же определяет, является ли полученный элемент данных тем, который должен быть сохранен, и если это так, он его сохраняет. Кроме того, если это конечный тег землетрясения, тогда в список результатов добавляется результирующая строка для этой части данных землетрясения.

```
public void endTag(String localName) {
    if (localName.equals(LATITUDE_TAG)) {
        mIsParsingLat = false;
    } else if (localName.equals(LONGITUDE_TAG)) {
        mIsParsingLng = false;
    } else if (localName.equals(MAGNITUDE_TAG)) {
        mIsParsingMag = false;
    } else if (localName.equals("earthquake")) {
        mResults.add(MAGNITUDE_TAG + ":" + mMag + "," + LATITUDE_TAG + ":"
                + mLat + "," + LONGITUDE_TAG + ":" + mLng);
        mLat = null;
        mLng = null;
        mMag = null;
    }
}
```

Когда событие является контентом, вызывается метод text. В качестве параметра передается содержимое элемента. Этот метод определяет, какой тег в настоящее время обрабатывается, и сохраняет содержимое для последующего использования.

```
public void text(String text) {
    if (mIsParsingLat) {
        mLat = text.trim();
    } else if (mIsParsingLng) {
        mLng = text.trim();
    } else if (mIsParsingMag) {
        mMag = text.trim();
    }
}
```

После завершения метода doInBackground в основной Activity, как и в предыдущих примерах, вызывается метод onPostExecute с результатом, переданным в качестве параметра.

```
@Override
protected void onPostExecute(List<String> result) {
    if (null != mClient)
        mClient.close();
    setListAdapter(new ArrayAdapter<String>(
            NetworkingAndroidHttpClientXMLActivity.this,
            R.layout.list_item, result));
}
```

Этот метод создает и устанавливает адаптер списка для вывода на экран, передавая в качестве параметра результирующий список, который был вычислен при работе метода handleResponse.

И последнее. Чтобы ваше приложение могло работать с интернетом, ему необходимо предоставить соответствующее разрешение:

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

Broadcast Receivers – приемники широковещательных сообщений

BroadcastReceiver – базовый класс для кода, который получает и обрабатывает рассылки (сообщения, упакованные в Intent), отправленные методом sendBroadcast (Intent) для компонентов, целью которых является ожидание определенных событий, чтобы принимать эти события и реагировать на них. И способ, которым все это работает, состоит в том, что отдельные широковещательные приемники регистрируются для получения конкретных событий, в которых они заинтересованы.

Например, в Android есть широковещательный приемник, задачей которого является прослушивание входящих SMS-сообщений. Затем где-то в еще, некий компонент совершает какое-то действие, о котором он хочет сообщить вещательным приемникам, например отправить SMS. Тогда он создает интент, представляющий это событие, и передает этот интент, как будто в радиоэфир. Когда Android получит широковещательный интент, содержащий SMS-сообщение, он проверяет какие службы (или приложения) зарегистрированы в системе на его получение, тогда в них происходит вызов метода onReceive, где интент присутствует в качестве одного из параметров.

Итак, первое – широковещательный приемник должен быть зарегистрирован для получения конкретных интентов. Во-вторых, какой-либо компонент генерирует интент и передает его в систему. В-третьих, Android доставляет этот интент получателям широковещательных сообщений, которые зарегистрированы для его получения. И четвертое – в приемниках затем происходит вызов их метода onReceive, в котором они и обрабатывают входящее событие.

Теперь поговорим о каждом из этих шагов по порядку. Для регистрации широковещательного приемника у разработчиков есть два варианта. Первый, они могут статически зарегистрировать широковещательный приемник, помещая информацию о нем в файл AndroidManifest. xml приложения, которому принадлежит широковещательный приемник. И второй, они могут регистрировать широковещательный приемник динамически, вызывая определенные методы во время выполнения программы.

Чтобы зарегистрировать широковещательный приемник статически, необходимо добавить в манифест тег «receiver», и затем внутри этого тега необходимо поместить по крайней мере один тег интент-фильтра. По содержимому интент-фильтра Android и определяет, соответствует ли полученный интент этому широковещательному приемнику. Формат тега выглядит примерно так.

```
<receiver
    android:enabled=["true" | "false"]
    android:exported=["true" | "false"]
    android:icon="drawable resource"
    android:label="string resource"
    android:name="string"
    android:permission="string"
    android:process="string" >
  . . .
</receiver>
```

Вначале стоит ключевое слово receiver, а затем добавляются некоторые из следующих атрибутов.

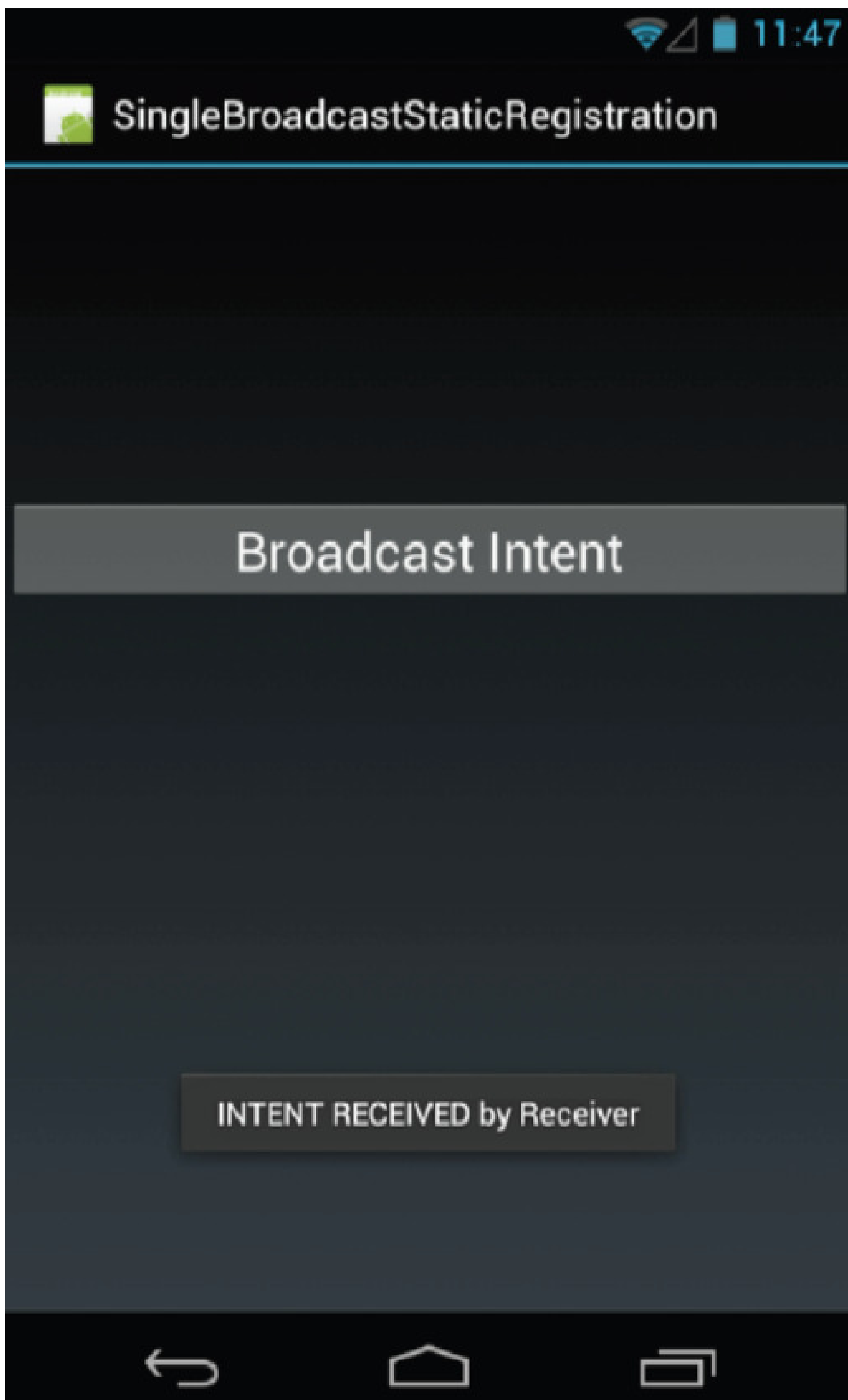
android: enabled позволяет включать или отключать определенный приемник.

android: exported, если установлено значение true, этот приемник может принимать широковещательные передачи от других приложений, в то время как если он установлен в значение false, то только те интенты, которые передаются другими компонентами внутри приложения.

android: name дает имя класса, реализующего этот приемник.

android: permission определяет строку разрешения, которое отправитель интента должен иметь, чтобы этот приемник получил интент от них. Как уже говорилось, необходимо указать хотя бы один тег интент-фильтра, которые были рассмотрены в главе Permissions. Теги этого интент-фильтра так же могут указывать на действия (action), данные (data) и категории (categories).

Если регистрировать приемник статически, эта информация будет считана и обработана при установке приложения и во время каждой загрузки системы. Рассмотрим приложение, которое статически регистрирует один широковещательный приемник, который получает пользовательский интент, называемый show toast intent. Это приложение отображает одну кнопку с надписью «Broadcast Intent». Нажатие этой кнопки вызывает отправку интента, а затем его прием широковещательным приемником, который выводит на экран тост-сообщение.


11:47
SingleBroadcastStaticRegistration
Broadcast Intent
INTENT RECEIVED by Receiver

Теперь откроем основную Activity. Этот код сначала определяет строку – действие интента, которая будет использоваться для идентификации этого интента.

```
package course.examples.BroadcastReceiver.singleBroadcastStaticRegistration;

import android.app.Activity;

public class SimpleBroadcast extends Activity {

    private static final String CUSTOM_INTENT = "course.examples.BroadcastReceiver.show_toast";
```

Далее идет «слушатель» кнопки, который вызывает метод sendBroadcast, передавая в интенте строку разрешения.

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

    Button button = (Button) findViewById(R.id.button);
    button.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {

            sendBroadcast(new Intent(CUSTOM_INTENT),
                    android.Manifest.permission.VIBRATE);
        }
    });
}
```

Этот интент будет сопоставлен с зарегистрированными в системе интент-фильтрами. Строка разрешения custom_intent указывает, что этот интент может быть доставлен только тем приемникам широковещательной передачи, у которых есть конкретно это разрешение.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «ЛитРес».

Прочитайте эту книгу целиком, купив полную легальную версию на ЛитРес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.