

Лекция #B-09

(часть 2)

*Многопоточность и GUI. Handler.*

Содержание:

1. [Постановка задачи](#_bookmark0).
2. [Основные сведения о технологии Handler](#_bookmark1).
3. [Отправка данных в Handler](#_bookmark2).
4. [Создание правильной модели Handler’ов](#_bookmark3).

# 

# Постановка задачи.

Развитие любой технологии обязательно влечет за собой усложнение принципов ее работы и, как

следствие – большие затраты труда тех людей, которые взаимодействуют с ней. Не исключением стала и технология многопоточности, которая имеет богатую историю на персональных компьютерах и уже

сейчас занимает новые и новые просторы в мобильных устройствах.

Сегодня любое мобильное устройство может управлять несколькими, а то и несколькими десятками потоков одновременно. Цена таких успехов – результат работы программиста. При работе с графическим интерфейсом жизненно важно координировать информацию, которая обрабатывается в потоках с ее отображением на экране. Однако технически эта главная задача не всегда решается просто.

Как было показано в предыдущей лекции, в *Android* не существует прямого доступа к графическому интерфейсу, как то было, например, во фреймворке *Swing*. Потенциальная склонность к написанию

"тяжелых", "продолжительных" по времени алгоритмов в главном GUI-потоке часто приводит к

негодованию пользователей, что напрямую сказывается на общественном мнении о той или иной технологии.

Однако с техническим усложнением разработчики пытаются создать более легкие механизмы управления, которые скрывают многие ненужные для работы программиста процессы. В многопоточности платформы *Android* так же были введены два механизма, которые позволяют как оптимизировать техническую сторону работы кода, так и логическую – распределение логики

приложения. Они пришли на смену очень неудобному и неэффективному механизму, рассмотренному в предыдущей лекции – *runOnUiThread()*. Одна из технологий, которая может использоваться в

относительно непродолжительных параллельных вычислениях, называется "*Handler*" и рассматривается в этой лекции, вторая – “*AsyncTask*” будет рассмотрена в следующей.

В качестве практики будет рассмотрена следующая задача "Взлом пароля": пользователь вводит в текстовое поле пароль - строку любой длительности с учетом латинских букв, цифр и служебных

символов (рис. 1). Нажав на кнопку, приложение начинает взламывать пароль – подбирать символы. Этот процесс будет мгновенным, так что придется его растянуть – использовать задержку потока. При

правильном подборе нового символа процесс должен сообщить об этом пользователю – вывести время работы процесса после нахождения каждого нового символа в пароле.

Характерным графическим элементом здесь будет постоянно вращающийся *ProgressBar*, который будет свидетельствовать о том, что GUI-поток не зависает при работе фонового процесса.

Этот же проект будет использоваться и в двух следующих темах, посвященной *AsyncTask* и коду универсального движка, поэтому необходимо добавить возможность выбора фонового механизма – группу радиокнопок и чекбокс для запуска кода движка вместо конкретно кодов технологий.



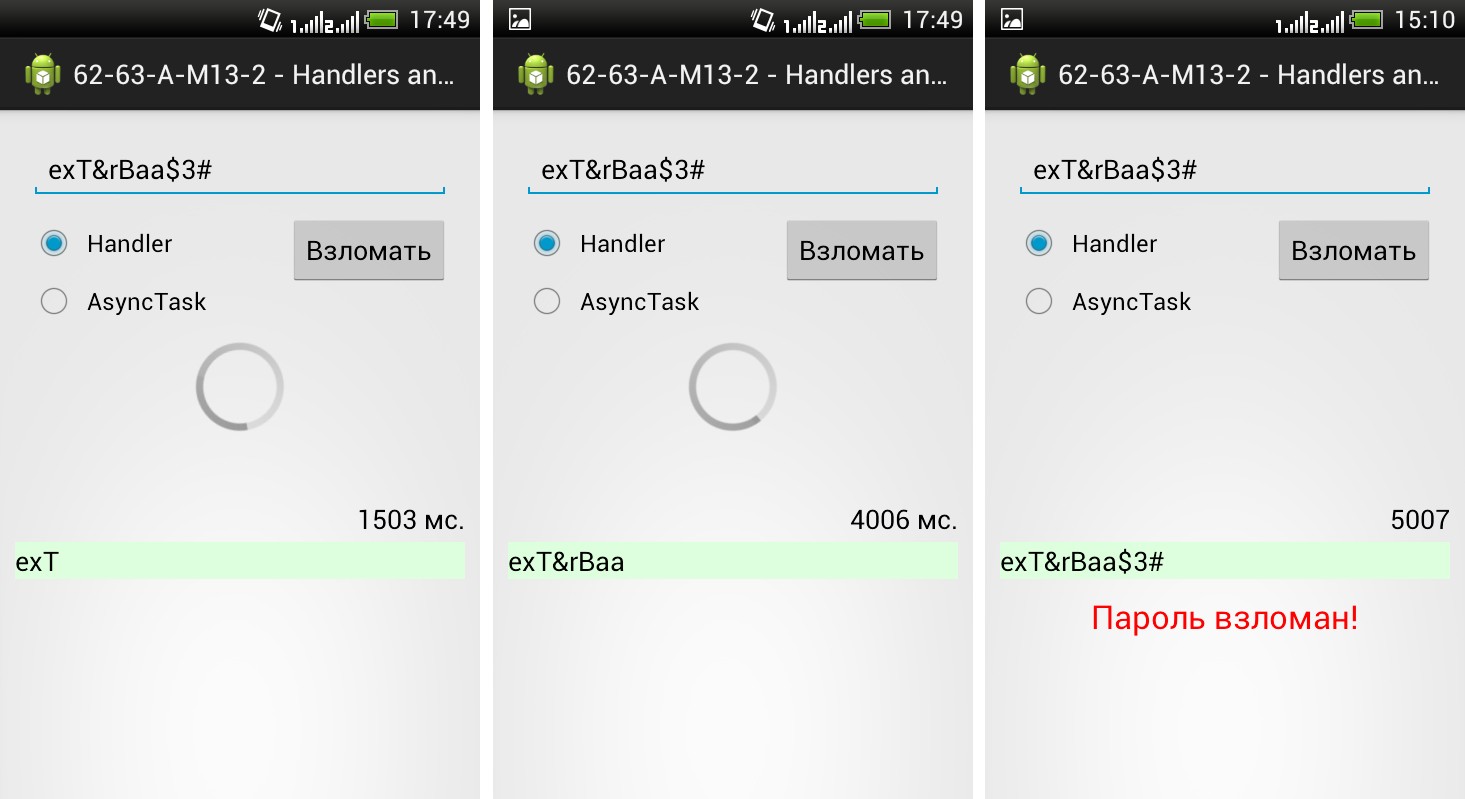


Рис. 1. Процесс обновляет экран В этом проекте используется следующая компоновка: **activity\_main.xml**

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<RelativeLayout xmlns:android=*"*[*http://schemas.android.com/apk/res/android*](http://schemas.android.com/apk/res/android)*"* android:layout\_width=*"match\_parent"* android:layout\_height=*"match\_parent"* android:gravity=*"center\_horizontal"*

android:layout\_margin=*"10dp"*>

<EditText

android:id=*"@+id/etNeedToHack"* android:layout\_width=*"match\_parent"* android:layout\_height=*"wrap\_content"* android:layout\_alignParentLeft=*"true"* android:layout\_alignParentTop=*"true"* android:layout\_margin=*"10dp"* android:ems=*"10"* />

<RadioGroup

android:id=*"@+id/rgTask1"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_alignLeft=*"@+id/etNeedToHack"* android:layout\_below=*"@+id/etNeedToHack"* >

<RadioButton

android:id=*"@+id/rbHandler"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:checked=*"true"* android:text=*"Handler"* />

<RadioButton

android:id=*"@+id/rbAsyncTask"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:text=*"AsyncTask"* />

</RadioGroup>

<Button

android:id=*"@+id/bStartHack"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_alignRight=*"@+id/etNeedToHack"* android:layout\_alignTop=*"@+id/rgTask1"* android:onClick=*"onClick"* android:text=*"Взломать"* />

<CheckBox

android:id=*"@+id/cbUseEngine"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_above=*"@+id/pb"* android:layout\_centerHorizontal=*"true"* android:text=*"Ипользовать движок"* />

<ProgressBar

android:id=*"@+id/pb"* style=*"?android:attr/progressBarStyleLarge"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_centerVertical=*"true"* android:layout\_toLeftOf=*"@+id/bStartHack"* android:visibility=*"invisible"* />

<TextView

android:id=*"@+id/tvHackResult"* android:layout\_width=*"match\_parent"* android:layout\_height=*"wrap\_content"* android:layout\_alignParentLeft=*"true"* android:layout\_below=*"@+id/pb"* android:layout\_marginTop=*"65dp"* android:background=*"#ddffdd"* android:text=*"Medium Text"*

android:textAppearance=*"?android:attr/textAppearanceMedium"* />

<TextView

android:id=*"@+id/tvHackTime"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_alignBottom=*"@+id/tvHackResult"* android:layout\_alignParentRight=*"true"* android:layout\_marginBottom=*"28dp"* android:text=*"Medium Text"*

android:textAppearance=*"?android:attr/textAppearanceMedium"* />

<TextView

android:id=*"@+id/tvHackEndLabel"* android:layout\_width=*"wrap\_content"* android:layout\_height=*"wrap\_content"* android:layout\_below=*"@+id/pb"* android:layout\_centerHorizontal=*"true"* android:layout\_marginTop=*"100dp"* android:text=*"Пароль взломан!"*

android:textAppearance=*"?android:attr/textAppearanceLarge"* android:textColor=*"#ff0000"* android:visibility=*"invisible"* />

</RelativeLayout>

Прежде, чем знакомиться с Handler’ом, необходимо создать сам фоновый процесс и удостовериться, что он работает – отображать результаты его работы в консоль:



**MainActivity.java**

public class MainActivity extends Activity { EditText etNeedToHack;

Button bStartHack; TextView tvHackResult; TextView tvHackTime; TextView tvHackEndLabel; RadioGroup rg;

CheckBox cbUseEngine; ProgressBar pb;

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.*activity\_main*);

etNeedToHack = (EditText) findViewById(R.id.*etNeedToHack*); bStartHack = (Button) findViewById(R.id.*bStartHack*);

rg = (RadioGroup) findViewById(R.id.*rgTask1*); cbUseEngine = (CheckBox) findViewById(R.id.*cbUseEngine*); pb = (ProgressBar) findViewById(R.id.*pb*);

tvHackResult = (TextView) findViewById(R.id.*tvHackResult*); tvHackTime = (TextView) findViewById(R.id.*tvHackTime*); tvHackEndLabel = (TextView) findViewById(R.id.*tvHackEndLabel*);

}

public void onClick(View v){

final String needToHackStr = etNeedToHack.getText().toString(); pb.setVisibility(View.*VISIBLE*);

switch (rg.getCheckedRadioButtonId()) {

case R.id.*rbHandler*: // Используется Handler Runnable runnable = new Runnable() {

@Override

public void run() {

int indexHacking = 0;

StringBuilder sbHackResult = new StringBuilder(); long timeStart = System.*currentTimeMillis*();

// проходимся по каждому символу в строке while(indexHacking < needToHackStr.length()){

// проходимся по всей UNICODE-таблице

// и сравниваем каждый символ в ней

// с выбранным из строки

for(char c = '\u0000'; c < 'z'; c++){

if(c == needToHackStr.charAt(indexHacking)){ Thread.*sleep*(500);

Log.*e*("PassHack", " >> Подобрано : " +

sbHackResult.append(c) +

" за " + (System.*currentTimeMillis*() - timeStart) +

" мс.");

indexHacking++;

break;

}

}

}



Log.*e*("PassHack", " >> ПАРОЛЬ ВЗЛОМАН!!! : " +

sbHackResult + " за " +

(System.*currentTimeMillis*() - timeStart) + " мс.");

}

};

Thread thread = new Thread(runnable); thread.start();

break;

case R.id.*rbAsyncTask*:// Используется AsyncTask

break;

}

}

}

# 

# Основные сведения о технологии *Handler*.

На самом деле *Handler* способен выполнить две частично связанные межу собой задачи:

* 1. Более эффективно обращаться c GUI-потоком (а точнее - передавать данные в него);
  2. Выполнять процессы, отложенные во времени.

Необходимо понять самый важный принцип *Handler’ов* – он очень похож на принцип *Intent’ов*, которые могли передавать конкретные данные (числа, символы, строки, массивы) из одной активности в другую. То же самое могут делать и *Handler’ы* – передавать конкретные данные (а, что более важно – и

ссылки на объекты) из одного потока в другой. Именно здесь и кроется главное отличие от метода *runOnUiThread*() : этот метод в буквальном понимании "вклинивает" код, описанный в параллельном потоке (и зачастую в другом классе) в основной контекст приложения – основной GUI-поток.

Использование же *Handler’а* позволяет получать обновленные данные в GUI-потоке и тут же ими

обновлять графический интерфейс. Таким образом, *Handler* легче всего себе представить как туннель между потоками, по которому в одном направлении будут двигаться данные.

С точки зрения передаваемых данных Handler являет собой некий пул информации, а точнее очередь типа *Queue*. Передаются данные в *Handler* в виде "сообщений" – инкапсулирующих объектов, которые и составляют очередь сообщений "внутри туннеля". Соответственно, если поток информации из

параллельного потока в GUI-поток будет очень частым, очередь может нарастать и по завершению отправки в параллельном потоке, в GUI-потоке понадобится некоторое для того, чтобы обработать те сообщения, которые еще находятся в очереди.

Существует также возможность передать сообщение в *Handler* без помещения в очередь, а разместить его в самой ее голове на выходе.

Поскольку главной задачей *Handler’a* является прием информации, необходимо обработать эту

задачу. Обновляемые данные должны попадать в GUI, сам объект *Handler’a* необходимо создать в классе, имеющий к нему доступ – в самой активности. И поскольку доступ к нему должен быть как из GUI- методов активности (типа *onCreate()*, *onClick()* – для приема сообщений), так и из методов

параллельного потока (для отправки сообщений), ссылку на объект необходимо разместить в поле активности. Чтобы иметь возможность получать сообщения, необходимо иметь либо реальный класс, либо анонимный, наследуемый от *Handler*.

Следует быть внимательным при использовании типа данных *Handler* - в Android-платформе их два, что вносит дополнительную путаницу. Необходимо использовать именно класс *Android*-платформы ***android.os.Handler***, вместо класса Java-платформы *java.util.logging.Handler*, который используется для

задач логирования и журналирования.

Для того, чтобы было легче разобраться с механизмом работы *Handler*, выберем второй вариант – анонимный класс на основе класса *android.os.Handler*: Однако, как будет показано далее, все-же более правильной конструкцией будет являться выделение класса-наследника от типа *Handler*.



public class MainActivity extends Activity { EditText etNeedToHack;

. . . . . . . .

**Handler h;**

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.*activity\_main*);

etNeedToHack = (EditText) findViewById(R.id.*etNeedToHack*);

. . . . . . . .

**h = new Handler(){**

**@Override**

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

**// TODO Auto-generated method stub super.handleMessage(msg);**

**}**

**};**

}

. . . . . . . .

}

В созданном анонимном классе необходимо переопределить метод ***handleMessage(Message msg)***, который будет работать по принципу слушателя событий – срабатывать всякий раз, как из параллельного потока будут отправлены данные через ***Handler h***.

Этот объект создан в GUI-потоке, а значит и все его методы будут работать в нем – соответственно данные, попадая в метод ***handleMessage(Message msg)***, могут быть установлены в соответствующие

графические элементы *без ошибок* типа.

android.view.ViewRootImpl$CalledFromWrongThreadException:

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

# 

# Отправка данных в Handler.

Создав возможность получения данных в главном графическом потоке (GUI), следует рассмотреть их отправку.

Существует несколько возможностей ее выполнения.

* 1. Отправка данных в Handler. Тип Message

Чтобы отправить сообщение в главный поток из параллельного, следует обратиться к Handler – переменной и вызвать ее следующие методы:

## boolean sendMessage(Message msg)

* простейший метод передачи данных в GUI-поток. Устанавливает посланное сообщение в конец очереди сообщений в Handler’е;

## boolean sendMessageAtFrontOfQueue(Message msg)

* устанавливает посланное сообщение в начало очереди сообщений в Handler’е, тем самым ускоряя его доставку в *handleMessage()*;

## boolean sendMessageAtTime(Message msg, long uptimeMillis)

* возможность отправить сообщение в определенные время и дату, хранимую в миллисекундах;

## boolean sendMessageDelayed(Message msg, long delayMillis)

* возможность отправить сообщение с задержкой по времени на ***delayMillis***

миллисекунд;

## boolean sendEmptyMessage(int what)

* отправка пустого сообщения с уникальным значением ***what*** либо возможность передать полезные числовые данные;

## boolean sendEmptyMessageAtTime(int what, long uptimeMillis)

* отправка пустого сообщения в определенное время и дату;

## boolean sendEmptyMessageDelayed(int what, long delayMillis)

* отправка пустого сообщения с задержкой по времени;

Первые 4 метода отправляют объект типа *android.os.Message* – это и есть сообщения. Этот объект довольно прост. Чтобы занести в него данные, необходимо обратиться к его открытым полям. Однако

существует и другой, более распространенный подход, который используется в различных частях Android- платформы и является рекомендуемым – передача блока информации в виде объекта типа *Bundle*, с

помощью метода *setData()*:



**Message msg = new Message();**

**msg.arg1;** - поле типа int – можно передать какие-либо числовые данные;

**msg.arg2;** - поле типа int – можно передать какие-либо числовые данные;

**msg.obj;** - поле типа *Object – позволяет передать ссылку на объект ЛЮБОГО типа*;

**msg.what;** - поле типа int – существует для возможности установить

передаваемому сообщению уникальный номер, по которому его можно найти. Если такая функция не нужна, может использоваться как поле для передачи полезных числовых данных;

**msg.setData(Bundle data)** - метод, позволяющий передать объект типа *Bundle* - набор

разнотипных информационных данных;

Последние три метода можно использовать либо в случае, если необходимо передать только лишь

число (хотя в исходном коде этот метод все равно создает объект Message и присваивает ему полученное значение ***when***), либо вообще ничего не нужно передавать, а отправка сообщения будет своего рода

сигналом для GUI-потока.

На данном этапе есть весь инструментарий для того, чтобы изменить рассматриваемую задачу и выполнить поставленную задачу – отображать взламывание пароля на экране.

Для этого вместо вывода логов на экран, необходимо создать сообщение и занести в него данные.

Через *Handler* в GUI будут передаваться текст и число. Таким образом, произойдет перенаправление выводящей информации – вместо консоли данные пойдут на экран, не нарушая тем самым самого механизма вычислений.

Выполнить передачу можно двумя способами – 1) через поля и 2) через объект типа *Bundle*:

// Log.e("PassHack", " >> Подобрано : " + sbHackResult.append(c) +

// " за " + (System.currentTimeMillis() - timeStart) + " мс."); Message msg = new Message();

// 1-й способ

msg.obj = sbHackResult.append(c).toString();

msg.arg1 = (int) (System.*currentTimeMillis*() - timeStart);

// 2-й способ

Bundle data = new Bundle();

data.putString("text", sbHackResult.append(c).toString()); data.putLong("time", (System.*currentTimeMillis*() - timeStart)); msg.setData(data)

h.sendMessage(msg);

Как видно из примера, второй способ более надежен – лишает программиста сразу 3-х проблем :

* 1. отпадает необходимости приведения типов для строк;
  2. отпадает проблема урезания чисел типа long до типа *int* – потенциальной проблемы потери данных;
  3. поле *msg.arg1* может быть использовано для других целей.

Также существует еще один лог, который нужно заменить – лог об окончании процесса и

необходимости проявить на экране *TextView* со значением "Пароль взломан!", который до этого будет невидим и спрятать крутящийся *ProgressBar*.



Поскольку этот тип сообщения функционально отличается от предыдущих – есть возможность отличать сообщения, устанавливая характерные константы, например в переменную *msg.what*. А при получении сообщений в *handleMessage()* по этому параметру отслеживать действия.

Теперь осталось только получить сообщения в методе *handleMessage()* и обновить графический интерфейс приложения пришедшими данными. Приведенный ниже код является полным решением поставленной задачи.

public class MainActivity extends Activity {

**public static final int *MESSAGE\_HACK\_IN\_PROGRESS* = 0; public static final int *MESSAGE\_HACK\_END* = 1;**

EditText etNeedToHack;

. . . . . . . .

Handler h;

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.*activity\_main*);

etNeedToHack = (EditText) findViewById(R.id.*etNeedToHack*);

. . . . . . . .

h = new Handler(){

@Override

public void handleMessage(Message msg) {

// TODO Auto-generated method stub super.handleMessage(msg);

// 1-й способ

**tvHackResult.setText((String)msg.obj); tvHackTime.setText("" + msg.arg1 + " мс.");**

// 2-й способ

**Bundle b = msg.getData(); tvHackResult.setText(b.getString("text")); tvHackTime.setText("" + b.getLong("time") + " мс.");**

**if(msg.what == *MESSAGE\_HACK\_END*){ tvHackEndLabel.setVisibility(View.*VISIBLE*); pb.setVisibility(View.*INVISIBLE*);**

**}**

}

};

}

public void onClick(View v){

final String needToHackStr = etNeedToHack.getText().toString(); pb.setVisibility(View.*VISIBLE*);

switch (rg.getCheckedRadioButtonId()) {

case R.id.*rb0*: // Используется Handler

Runnable runnable = new Runnable() {

@Override

public void run() {

int indexHacking = 0;

StringBuilder sbHackResult = new StringBuilder(); long timeStart = System.*currentTimeMillis*();

// проходимся по каждому символу в строке while(indexHacking < needToHackStr.length()){

// проходимся по всей UNICODE-таблице

// и сравниваем каждый символ в ней

// с выбранным из строки

for(char c = '\u0000'; c < 'z'; c++){

if(c == needToHackStr.charAt(indexHacking)){ Thread.*sleep*(500);

Message msg = new Message();

// 1-й способ

msg.obj = sbHackResult.append(c).toString(); msg.arg1 =

(int) (System.*currentTimeMillis*() - timeStart);

// 2-й способ

Bundle data = new Bundle(); data.putString("text",

sbHackResult.append(c).toString()); data.putLong("time",

(System.*currentTimeMillis*() - timeStart)); msg.setData(data);

msg.what = *MESSAGE\_HACK\_IN\_PROGRESS*;

h.sendMessage(msg); indexHacking++;

break;

}

}

}

Message msg = new Message();

// 1-й способ

msg.obj = sbHackResult.toString();

msg.arg1 = (int) (System.*currentTimeMillis*() - timeStart);

// 2-й способ

Bundle data = new Bundle(); data.putString("text", sbHackResult.toString());

data.putLong("time", (System.*currentTimeMillis*() - timeStart)); msg.setData(data);

msg.what = *MESSAGE\_HACK\_END*; h.sendMessage(msg);

}

};



Thread thread = new Thread(runnable); thread.start();

break;

case R.id.*rb1*: // Используется AsyncTask break;

}

}

}

* 1. Возможность работы *Handler’а* по принципу *runOnUiThread()*

Наряду с возможностью передачи информационных данных (чисел и строк) всего лишь для

обновления состояния экрана, с помощью *Handler’ов* существует возможность передать в графический поток сам *Runnable*-процесс точно так же, как это делает *runOnUiThread()*, только более удобно. Для этого у *Handler*-объектов существует набор методов, похожий на рассмотренный выше для сообщений:

## boolean post(Runnable r)

***boolean postAtFrontOfQueue(Runnable r)***

***boolean postAtTime(Runnable r, long uptimeMillis) boolean postDelayed(Runnable r, long delayMillis)***

Точно так же, как и объекты типа *Message*, отправляемые в *handler* объекты-процессы типа

*Runnable* будут выстраиваться в очередь.

Однако, по рассмотренным выше причинам, использовать методы *post()* для обновления

состояния GUI-интерфейса нежелательно. Их хорошо использовать при передаче процессов из одного неGUI-потока в другой неGUI-поток.

# 

# Создание правильной модели Handler’ов.

Существует одна проблема технического характера, которая может повлечь за собой утечку

свободной оперативной памяти и, как следствие – возможные непредсказуемые действия внутри

программного обеспечения устройства. Об этой проблеме свидетельствует предупреждение в любой среде разработки (рис. 2)

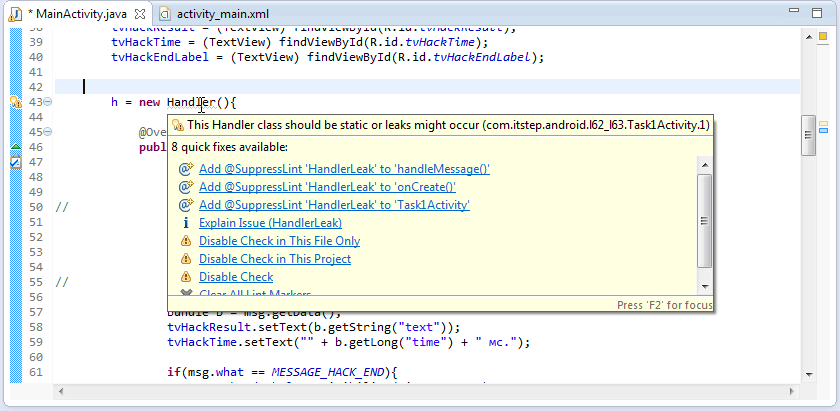


Рис. 2 Предупреждение о возможной утечке памяти при использовании анонимного класса Причиной тому являются два факта:

1. Все системные сообщения платформы находятся в одной глобальной очереди, в нее попадают и те сообщения для *Handler’ов*, которые являются отложенными по времени;
2. Любой объект внутреннего (нестатического) или анонимного класса *хранит неявную ссылку* на объект внешнего класса.

В случае отложенных сообщений, посланное в Handler сообщение передается в глобальную очередь и ждет своего времени. Если же закрыть приложение, что обычно приводит к очищению памяти от

занимаемых объектов класса этой активности (*MainActivity*, например), сообщения, уже находящиеся в стеке, удалены не будут. В данном случае очистить память не удастся – сообщение помещено в

глобальную очередь сообщений и это сообщение *Message* имеет ссылку на объект *Handler*, который его послал, а он, в свою очередь – на объект активности *MainActivity*, где он и размещается. Имея такую

последовательность, объект класса активности не сможет быть удален – такого рода ссылки являются "сильными".

В такой ситуации на помощь приходит механизм "слабых ссылок" *WeakReference*<>. Этот механизм позволяет при необходимости оборвать связи, которые помечены как "слабые" и удалить требуемый

объект. Таким образом, используя слабую ссылку на активность *MainActivity*, в момент, когда система решит, что закрытую активность нужно удалить из памяти, она увидит, что существующая ссылка

Handler’а на активность является слабой и позволит удалить объект активности из памяти. Дождавшись



своего часа, сообщение не будет обработано, поскольку не будет где и в скором времени так же будет освобождено из памяти.

Более подробно можно прочитать в этих статьях: [http://forum.startandroid.ru/viewtopic.php?f=30&t=1870](http://forum.startandroid.ru/viewtopic.php?f=30&amp;t=1870) <http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html>

С точки зрения архитектуры, чтобы реализовать свойства *WeakReference*<>, необходимо создать класс-наследник *Handler* и создать *явную ссылку* на активность, где этот *Handler* создается и

используется и пометить ее как слабую. Этот класс должен быть либо статическим внутренним, либо внешним. Будет выбран первый вариант:

public class MainActivity extends Activity {

. . . . . . . .

Handler h;

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.*activity\_main*);

. . . . . . . .

/\*

h = new Handler(){

@Override

public void handleMessage(Message msg) {

// TODO Auto-generated method stub super.handleMessage(msg);

// . . . . . .

}

};

\*/

h = new ValidHandler(this);

}

public void onClick(View v){

. . . . . . . .

}

**static class ValidHandler extends Handler { WeakReference<MainActivity> wrActivity;**

**public ValidHandler(MainActivity activity) {**

**wrActivity = new WeakReference<MainActivity>(activity);**

**}**

**@Override**

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

**// TODO Auto-generated method stub super.handleMessage(msg);**

**}**

**}**

}



Таким образом, объект *WeakReference*<> является всего лишь оболочкой того, объекта, который в себя принимает, но выполняет свои задачи внутри, невидимо для программиста.

Также, создав отдельный класс, события, которые будут приходит из параллельного потока, будут появляться в методе этого *handleMessage()* класса. Но теперь возникает вопрос: как же получить доступ к View-объектам, которые необходимо обновлять – TextView отображения результата взламывания и отображения времени (для случая, если класс *ValidHandler* будет внешним по отношению к *MainActivity*)? Поскольку объект класса *ValidHandler* будет иметь поле, связанное с активностью. То и получить ссылку на нее тоже не составит труда - метод ***get()*** объекта типа *WeakReference* вернет ссылку на хранимый им объект:

static class ValidHandler extends Handler {

WeakReference<MainActivity> wrActivity; public ValidHandler(MainActivity activity) {

wrActivity = new WeakReference<MainActivity>(activity);

}

@Override

public void handleMessage(Message msg) {

// TODO Auto-generated method stub super.handleMessage(msg);

**MainActivity activity = wrActivity.get();**

if (activity != null) {

}

}

}

Теперь, получив ссылку на активность, в которой размещены все необходимые View-компоненты и для случая если они открыты (***public*** или ***protected***) можно обращаться к ним как к полям полученной

переменной *activity*:

static class ValidHandler extends Handler {

WeakReference< MainActivity > wrActivity; public ValidHandler(MainActivity activity) {

wrActivity = new WeakReference< MainActivity >(activity);

}

@Override

public void handleMessage(Message msg) {

// TODO Auto-generated method stub super.handleMessage(msg);

MainActivity activity = wrActivity.get(); if (activity != null) {

**// 1-й способ**

**activity.tvHackResult.setText((String)msg.obj); activity.tvHackTime.setText("" + msg.arg1 + " мс.");**

**// 2-й способ**

**Bundle b = msg.getData(); activity.tvHackResult.setText(b.getString("text")); activity.tvHackTime.setText("" + b.getLong("time") + " мс.");**

**if(msg.what == *MESSAGE\_HACK\_END*){ activity.tvHackEndLabel.setVisibility(View.*VISIBLE*); activity.pb.setVisibility(View.*INVISIBLE*);**

**}**



}

}

}

Следует запомнить схему классов при работе *Handler’ами*:

class MyActivity extends Activity{

View guiField1; View guiField2; View guiField3;

Handler h;

@Override

protected void onCreate(Bundle savedInstanceState) {

guiField1 = ...; guiField2 = ...;

. guiField3 = ...;

h = new MyHandler(this);

}

static class MyHandler extends Handler { WeakReference<MyActivity> wrActivity;

public MyHandler(MyActivity activity) {

wrActivity = new WeakReference<MyActivity>(activity);

}

@Override

public void handleMessage(Message msg) {

// TODO Auto-generated method stub super.handleMessage(msg);

MyActivity activity = wrActivity.get(); if (activity != null) {

activity.guiField1 ...; activity.guiField2 ...; activity.guiField3 ...;

}

}

}

}