

Лекция #B-09

### (часть 3)

*Многопоточность и GUI. AsyncTask.*

Содержание:

1. [Главные сведения по *AsyncTask*](#_bookmark0).
2. [Применение *AsyncTask* на практике](#_bookmark1).



1. Основные сведения по *AsyncTask*.

## Технология *AsyncTask* является встроенным механизмом платформы Android специально для работы в фоном режиме. Ее можно себе грубо представить как класс-обертку технологии *Handler'ов*. Если последний представлял из себя только канал передачи данных из одного потока в другой, то *AsyncTask* – это сам по себе автоматически создаваемый системой фоновый поток с собственным доступом к GUI-

потоку. Огромным его плюсом является то, что все "настроечные" этапы создания фонового процесса проводятся без участия программиста, который сосредоточен только на логике этого внутреннего

процесса и на механизме обновления GUI, он не тратит свое время на настройку работы этого процесса, как это происходит с *Handler'ами* – выбор анонимного либо отдельного класса, размещение *Handler*-

переменной в классе, отправка и обработка пришедших сообщений, доступ к GUI-объектам ит.д. Также

*AsyncTask* имеет еще несколько преимуществ, рассматриваемых далее в лекции.

Для изучения технологии потребуется проект, рассмотренный в предыдущей лекции, посвященной

*Handler’ам* (рис. 1), но теперь при клике на кнопку "Взломать" блок *switch()* должен пойти по другой ветке.

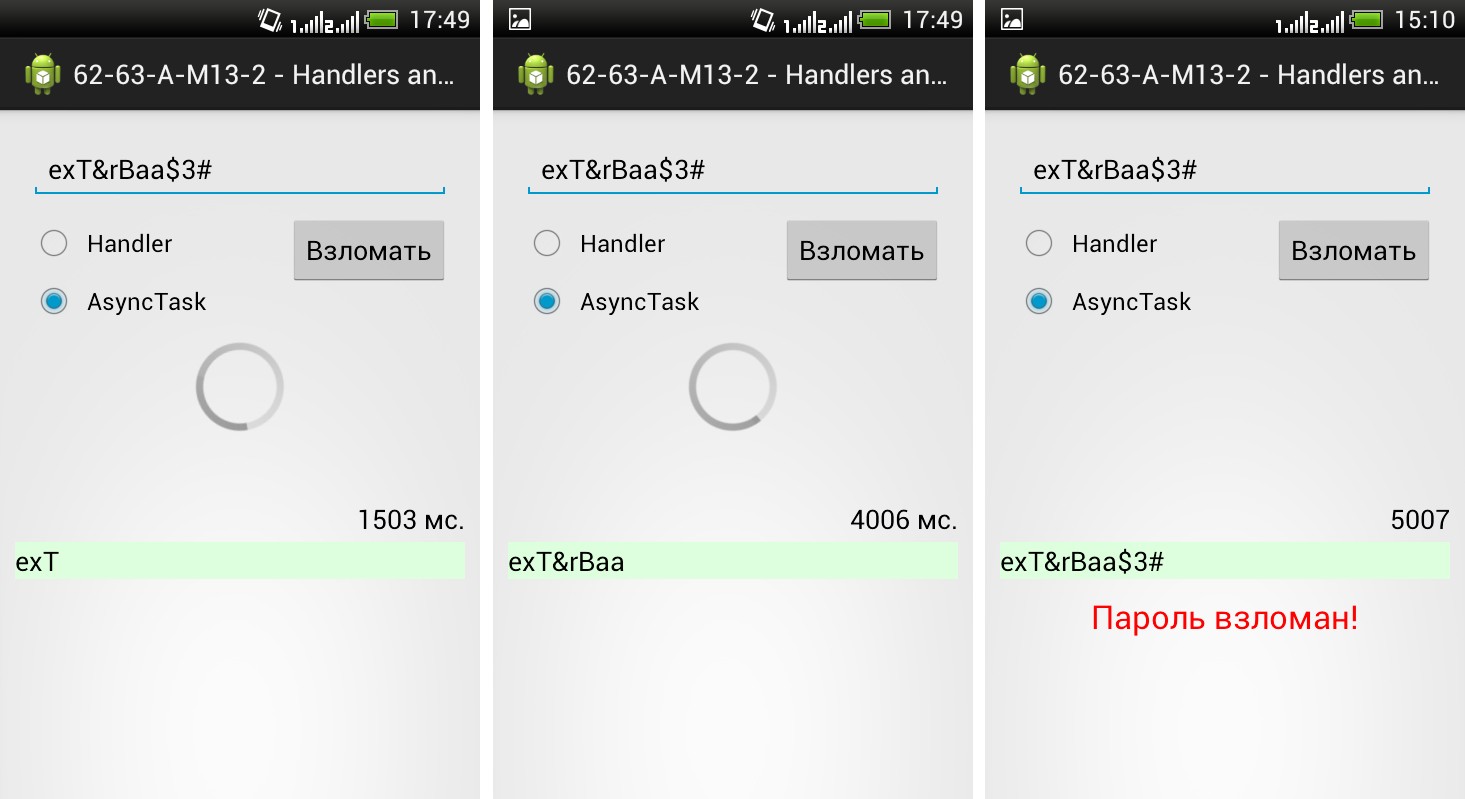


Рис. 1. Процесс обновляет экран.

* 1. Настройка *AsyncTask*.

## *AsyncTask* используется таким же образом, как и активность или правильный *Handler* - необходимо создать кастомный класс, наследуемый от Android-класса *AsyncTask*. Он будет внутренним для рассматриваемой в задаче активности и параллельным классу *ValidHandler*:

public class MainActivity extends Activity {

. . . . .

@Override

protected void onCreate(Bundle savedInstanceState) { . . . } public void onClick(View v) { . . .}

static class ValidHandler extends Handler { . . .}



Написав кусок кода, например, ***private class ValidAsyncTask extends AsyncT*** и нажав автозаполнение Ctrl+пробел – *Android Studio* дополнит строку данными в угловых скобках <> и попросит обязательно переопределить один метод, который называется *doInBackground()*.

**class ValidAsyncTask extends AsyncTask<Params, Progress, Result>{**

**@Override**

**protected Result doInBackground(Params... params) {**

**// TODO Auto-generated method stub return null;**

**}**

**}**

}

### Вначале рассмотрим методы класса *AsyncTask*, а затем разберемся с "угловыми" параметрами.

Метод *doInBackground()* и есть самый важный метод технологии *AsyncTask*. Если провести параллель со способами выполнения фоновых процессов, изученных раннее (*runOnUiThread()* и *Handler*) – *метод doInBackground()* является аналогом метода *run()* нового *Thread*-класса, в котором происходит вся "тяжелая" фоновая работа. Соответственно, *метод doInBackground() не имеет доступа к GUI*.

Если не требуется обновления графики в процессе работы *AsyncTask*, вполне достаточно единственного метода *doInBackground()*.

### Для того, чтобы обновлять графику, необходимо использовать похожие действия, как в случае с *Handler'ом* - отправлять сообщения в графический поток.

Это достигается следующим способом:

* параллельно методу *doInBackground()* переопределить метод *onProgressUpdate()* – метод, который наделен доступом к GUI. В сравнении с *Handler'ом* - аналог метода *handleMessage()*, в который будут приходить данные для обновления и сразу же отображаться в графических элементах на экране;
* для передачи данных в *onProgressUpdate()* из метода *doInBackground()* нужно вызывать метод

*publishProgress()* – аналог метода *sendMessage()* у *Handler'ов.*

### Существует также третий "полуобязательный" , но очень полезный метод, сигнализирующий об окончании работы *doInBackground()* - метод *onPostExecute()* . Также в этот метод можно передать определенное

значение – например, результат фоновых вычислений – значение, которое возвратит метод *onProgressUpdate()* в конце своей работы. Этот метод также имеет доступ к GUI.

В итоге, класс *ValidAsyncTask* для минимальной удовлетворительной работы будет выглядеть так:

class ValidAsyncTask extends AsyncTask<Params, Progress, Result>{

@Override

protected Result doInBackground(Params... params) {

// Что-то происходит, что-то вычисляется ... publishProgress(values);

// Что-то происходит, что-то вычисляется ...

return result;

}

protected void onProgressUpdate(Progress[] values) {

// Доступ к GUI

};

protected void onPostExecute(Result result) {

};

}



* 1. Запуск *AsyncTask*.

## Запускается *AsyncTask* очень просто - необходимо создать объект описанного класса-наследника и вызвать его метод *execute()*:

public void onClick(View v){

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

switch (rg.getCheckedRadioButtonId()) {

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

. . . . . . . .

break;

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

**AsyncTask at = new ValidAsyncTask(); at.execute(...);**

break;

}

}

Главной особенностью *AsyncTask*, о которой нужно помнить всегда, является его ОДНОРАЗОВАЯ работа.

Это означает, что однажды создав ссылку /***at***/ и проинициализировав ее /**new ValidAsyncTask()**/, можно запустить *AsynkTask* для этой ссылки только ОДИН РАЗ. Второй раз запустить эту ссылку уже

нельзя, если необходимо ее запустить второй раз, **для этой ссылки нужно создать новый объект**

**(т.е.опять инициализировать конструктором)**.

* 1. Остановка *AsyncTask*.

## Прерывание работы *AsynсTask* также происходит очень просто с помощью единственного метода *cancel()*. Но теперь необходимо вынести ссылку *AsyncTask* в поле. Также для того, чтобы реализовать для одной кнопки двойной функционал "Запуск/Стоп", необходимо добавить *boolean*-переменную:

public class MainActivity extends Activity {

. . . .

**AsyncTask<Params, Progress, Result> at; boolean isTaskOnNow;**

. . . .

public void onClick(View v){

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

switch (rg.getCheckedRadioButtonId()) {

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

. . . . . . . .

break;



Важно! Вызов метода *cancel****()*** запретит вызов метода *onPostExecute()* и вместо него будут вызваны два

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

if (isTaskOnNow) {

// остановить задачу

**at.cancel(...);**

bStartHack.setText("Взломать"); isTaskOnNow = false;

}

else {

// начать задачу

**at = new ValidAsyncTask(); at.execute(...);**

bStartHack.setText("Стоп"); isTaskOnNow = true;

}

break;

}

}

}

### метода (также переопределяемых параллельно):

@Override

protected void onCancelled() { super.onCancelled();

}

protected void onCancelled(Result result) { super.onCancelled(result);

}

И САМОЕ ВАЖНОЕ - вызов метода *cancel()* ТАКЖЕ ЗАПРЕТИТ СРАБАТЫВАНИЕ

МЕТОДА *onProgressUpdate()*, который и обновляет данные непосредственно. Так что, нажав кнопку стоп - данные прекратят обновляться только на экране, а вот сам поток будет работать до своего конца, если, конечно же, его не оборвать через *return* либо *break*, если в фоновом процессе находится цикл.

Также существует последний метод, который используется относительно редко, но все же имеет функцию подготовки некоторых данных перед работой *doInBackground()*. Это метод *onPreExecute()*.

### Теперь можно полностью получить строение *AsyncTask*:

public class MainActivity extends Activity {

. . . . . . . .

Handler h;

AsyncTask<Params, Progress, Result> at; boolean isTaskOnNow;

@Override

protected void onCreate(Bundle savedInstanceState) { . . . }

public void onClick(View v){

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



* 1. Параметризированные типы *AsyncTask<Params, Progress, Result>*.

switch (rg.getCheckedRadioButtonId()) {

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

. . . . . . . .

break;

case R.id.*rbAsyncTask*: // Используется AsyncTask if (isTaskOnNow) {

// остановить задачу

at.cancel(...);

bStartHack.setText("Взломать"); isTaskOnNow = false;

}

else {

// начать задачу

at = new ValidAsyncTask(); at.execute(...);

bStartHack.setText("Стоп"); isTaskOnNow = true;

}

break;

}

}

static class ValidHandler extends Handler { . . . }

class ValidAsyncTask extends AsyncTask<Params, Progress, Result>{

@Override

protected void onPreExecute() {

}

@Override

protected Result doInBackground(Params... params) { publishProgress(values);

return result;

}

protected void onProgressUpdate(Progress[] values) {

}

protected void onPostExecute(Result result) {

}

@Override

protected void onCancelled() {

}

protected void onCancelled(Result result) {

}

}

}

### Для того, чтобы досконально понять что такое параметризированные типы, следует прочитать главу 14 "Параметризация" книги "Философия Java" Б. Эккеля.

Вкратце - необходимо выяснить для чего вообще применяется *AsyncTask*:



* параллельная работа необходима для обработки некоторых входных данных (*Params*),
* выдачи промежуточных результатов в GUI (*Progress*) и
* получение результатов деятельности *AsyncTask* (*Result*)

### Следует обратить внимание на направления передачи данных:

1. *Запуск*. Метод *execute()* передает данные в метод *doInBackground()* - у них ОБЩИЕ ВХОДЯЩИЕ параметры - типа *Params*;
2. *Обновление*. Метод *pubishProgress()* передает данные в метод *onProgressUpdate()* - у них ОБЩИЕ ВХОДЯЩИЕ параметры - типа *Progress*;
3. *Результат работы*. *doInBackground()* возвращает результат типа *Result*, и это значение является входящим для методов *onPostExecute()* и *onCancelled()*.

*В угловых скобках не может быть примитивов.*

### Например, в случае самого распространенного применения технологии *AsyncTask* - скачивании файла – типизация параметров может быть следующая:

***Params*** - входящие параметры - url, по которому будет загружаться файл (и/или путь его сохранения) - тип ***String***;

***Progress*** - обновляемые параметры - целое число процентов загрузки - тип ***Integer***;

***Result*** - результирующее значение - подсчет переданного количества байт при загрузке - тип ***Long***.

Важно! Если, например, некоторый из параметров не нужен (например, не нужно подсчитывать количество

загруженных байт в конце выполнения процесса) - на его место следует поставить тип ***Void***.



1. Применение *AsyncTask* на практике.

## Какие Generic-типы необходимо установить в сигнатуре класса?

* В фоновый процесс необходимо передать текст – значение пароля, который будет "взламываться". Значит, в качестве *Params* будет указываться *String*;
* На экране во время работы фонового процесса необходимо обновлять два значения – *строку* уже подобранной части пароля и число – *время* подбора последнего символа. К сожалению, в метод *onProgressUpdate()* можно передать несколько параметров, но только лишь одного типа. Поэтому,

## поскольку в Java нет подходящего класса, который мог бы в себе инкапсулировать *String*- и *int*- переменную (аналога *Message* у *Handler'ов*), придется создать его вручную – это будет еще один внутренний класс активности *MainActivity*:

class UpdateMessage{

String hackedPassword; int hackingTime;

public UpdateMessage(String hackedPassword, int hackingTime) { super();

this.hackedPassword = hackedPassword; this.hackingTime = hackingTime;

}

}

Тогда в качестве типа *Progress* будет использован тип *UpdateMessage*;

## В рассматриваемом примере не будет вычисляться какой либо результат работы метода *doInBackground()*, и это значит, что тип *Result* его возвращаемого значения и тип входящего параметра в метод *onPostExecute()* будет *Void*. В конце выполнения процесса взлома не будет

дополнительного сообщения, как это было в случае с *Handler’ом*, а скрыть ProgressBar и отобразить метку " Пароль взломан!" можно будет как раз из метода *onPostExecute()*!

## Теперь активность примет следующий вид:

public class MainActivity extends Activity {

. . . . . . . .

AsyncTask<**String**, **UpdateMessage**, **Void**> at; boolean isTaskOnNow;

@Override

protected void onCreate(Bundle savedInstanceState) { . . . } public void onClick(View v){

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

switch (rg.getCheckedRadioButtonId()) {

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

. . . . . . . . .

break;



### Последнее, что осталось – заполнить тела методов класса *ValidAsyncTask*.

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

if (isTaskOnNow) {

// остановить задачу at.cancel(true);

bStartHack.setText("Взломать"); isTaskOnNow = false;

}

else {

// начать задачу

at = new ValidAsyncTask(); at.execute(**needToHackStr**);

bStartHack.setText("Стоп"); isTaskOnNow = true;

}

break;

}

}

static class ValidHandler extends Handler { . . . }

class ValidAsyncTask extends AsyncTask<**String**, **UpdateMessage**, **Void**>{

@Override

protected void onPreExecute() {

}

@Override

protected **Void** doInBackground(**String**... params) {

. . . . . . . .

**UpdateMessage um = new UpdateMessage(String, int); publishProgress(um)**;

. . . . . . . .

return null;

}

@Override

protected void onProgressUpdate(**UpdateMessage**... values) {

}

protected void onPostExecute(**Void** result) {

}

}

class UpdateMessage{

String hackedPassword; int hackingTime;

public UpdateMessage(String hackedPassword, int hackingTime) { super();

this.hackedPassword = hackedPassword; this.hackingTime = hackingTime;

}

}

}



**>> *doInBackground()***

Поскольку, будет выполняться функционал, идентичный тому, который выполняется в *Handler’е*, можно просто взять содержимое метода *run()* и занести его в метод *doInBackground()* с небольшой корректировкой – вместо классов *Message* использовать *UpdateMessage*, а вместо *sendMessage()* – метод *publishProgress()*. Также в этом методе не будет блока кода, который отправляет последнее сообщение с константой, по которой производится отображение и скрытие графических элементов *TextView tvHackEndLabel* и *ProgressBar pb*.

@Override

protected Void doInBackground(String... params) { String needToHackStr = params[0];

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);

**UpdateMessage um = new UpdateMessage(**

**sbHackResult.append(c).toString(), (int) (System.*currentTimeMillis*() - timeStart)**

**);**

**publishProgress(um);** indexHacking++; break;

}

}

}

return null;

}

# >> onProgressUpdate()

Здесь будут отображаться передаваемые в метод *publishProgress()* данные:

@Override

protected void onProgressUpdate(UpdateMessage... values) {

tvHackResult.setText(values[0].hackedPassword); tvHackTime.setText("" + values[0].hackingTime);

}

# 

# >> onPostExecute ()

Этот метод сигнализирует об окончании фонового процесса и здесь можно разместить скрытие *ProgressBar pb*, отображение метки *TextView tvHackEndLabel*, и переименование кнопки *Button* bStartHack:

protected void onPostExecute(Void result) {

pb.setVisibility(View.*INVISIBLE*); tvHackEndLabel.setVisibility(View.*VISIBLE*); bStartHack.setText("Взломать");

}

На этом решение задачи с помощью *AsyncTask* выполнено. Далее приведен *весь код проекта*,

## посвященный рассматриваемой задаче "Взлома пароля", в которой используется технологии *Handler* и

*AsyncTask*.



public class MainActivity extends Activity {

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

EditText etNeedToHack; Button bStartHack; TextView tvHackResult; TextView tvHackTime; TextView tvHackEndLabel; RadioGroup rg; ProgressBar pb;

Handler h;

AsyncTask<String, UpdateMessage, Void> at; boolean isTaskOnNow;

@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*); pb = (ProgressBar) findViewById(R.id.*pb*);

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

/\*

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);

}

}

};

\*/

h = new ValidHandler(this);

}

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)){

try {

Thread.*sleep*(500);

} catch (InterruptedException e) {}

|  |  |  |
| --- | --- | --- |
|  | | 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.*rbAsyncTask*: // Используется AsyncTask if (isTaskOnNow) {

// остановить задачу at.cancel(true);

}

else {

bStartHack.setText("Взломать"); isTaskOnNow = false;

// начать задачу

at = new ValidAsyncTask(); at.execute(needToHackStr);

bStartHack.setText("Стоп"); isTaskOnNow = true;

}

break;

}

}

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*);

}

}

}

}

class ValidAsyncTask extends AsyncTask<String, UpdateMessage, Void>{

@Override

protected Void doInBackground(String... params) { String needToHackStr = params[0];

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)){

try { Thread.*sleep*(500);

} catch (InterruptedException e) {}

UpdateMessage um = new UpdateMessage(sbHackResult.append(c).toString(),

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

// msg.what = MESSAGE\_HACK\_IN\_PROGRESS; publishProgress(um);

indexHacking++;

break;

}

}

}

return null;

}

@Override

protected void onProgressUpdate(UpdateMessage... values) {

tvHackResult.setText(values[0].hackedPassword); tvHackTime.setText("" + values[0].hackingTime);

}

protected void onPostExecute(Void result) {

pb.setVisibility(View.*INVISIBLE*); tvHackEndLabel.setVisibility(View.*VISIBLE*); bStartHack.setText("Взломать");

}

}

class UpdateMessage{

String hackedPassword; int hackingTime;

public UpdateMessage(String hackedPassword, int hackingTime) { super();

this.hackedPassword = hackedPassword; this.hackingTime = hackingTime;

}

}

}