Wykład
Wielowątkowość Android i Java
/
komunikacja między wątkami

Programowanie Urządzeń Mobilnych

Dr inż. Damian Raczyński

Wielowątkowość

Klasy `Looper`, `Handler` oraz `HandlerThread` wspierają programowanie asynchroniczne.

Wątek główny w Android składa się z obiektu `Looper'a` i `Hanlder'ów` - zrozumienie mechanizmów związanych z klasami pozwala na stworzenie interaktywnej apliakcji.

`MessageQueue`
Jest strukturą (kolejką) przechowującą listę obiektów klasy `Message` lub `Runnable`.

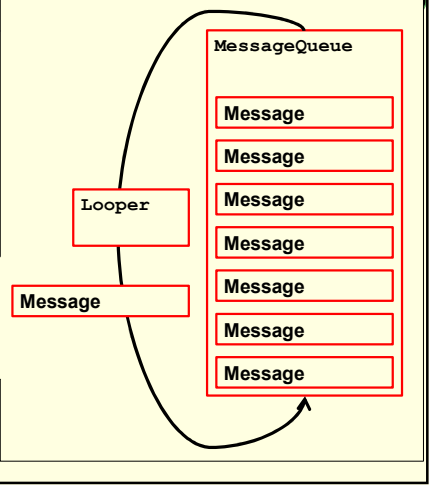
`Handler`
Dodaje do kolejki elementy z wykorzystaniem `Looper'a` i wykonuje/pobiera je, gdy elementy opuszczają kolejkę.
Element jest przypisany wyłącznie do jednego `Looper'a`.

`Looper`
Przechodzi przez kolejkę i wysyła zawarte w niej elementy do odpowiedniego handler'a (szereguje elementy w kolejce).

Wielowątkow

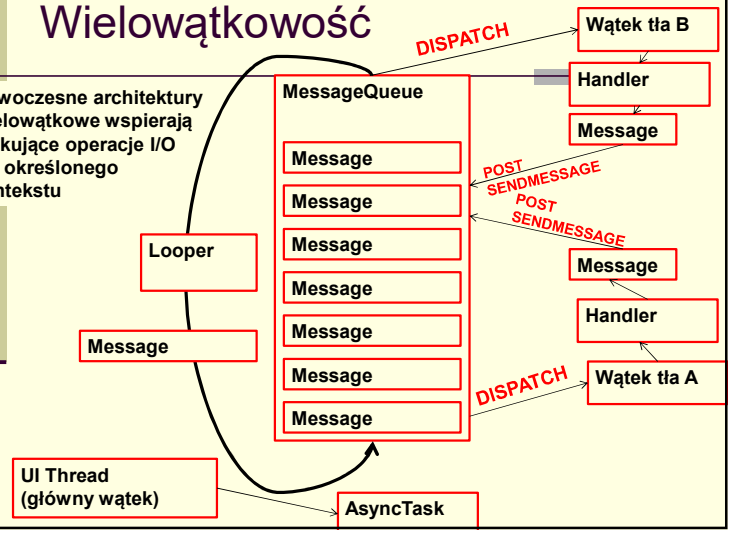
Upraszcza strukturę programu - umożliwia wykonanie operacji blokujących

- **architektura jednozadaniowa nie może wykonywać operacji blokujących (wykonanie blokady „zacina" system)**
- **To komplikuje pisanie aplikacji (rozłączanie przebiegu programu np. w przypadku operacji plikowej)**



Wielowątkowość

Nowoczesne architektury wielowątkowe wspierają blokujące operacje I/O dla określonego kontekstu



Prosty przykład:

```
private Bitmap bitmap;
final ImageView iview=...
final Button button=...
button.setOnClickListener(new OnClickListener(){
        public void onClick(View v){
                new Thread(new Runnable(){
                        public void run(){
                                bitmap=downloadImage(URI);
                                iview.post(new Runnable(){
                                        public void run()
                                        {
                                        iview.setImageBitmap(bitmap);
                                        }
                                });
                        }
                }).start();
```

developer.android.com/guide/components/processes-and-threads.html#WorkerThreads

Prosty przykład:

```
private Bitmap bitmap;
final ImageView iview=...
final Button button=...
button.setOnClickListener(new OnClickListener(){
        public void onClick(View v){
                new Thread(new Runnable(){
                        public void run(){
                                bitmap=downloadImage(URI);
                                iview.post(new Runnable(){
                                        public void run()
                                        {
                                        iview.setImageBitmap(bitmap);
                                        }
                                });
                        }
                }).start();
```

ściągnięcie obrazka

Uruchomienie nowego wątku

developer.android.com/guide/components/processes-and-threads.html#WorkerThreads

Wielowątkowość

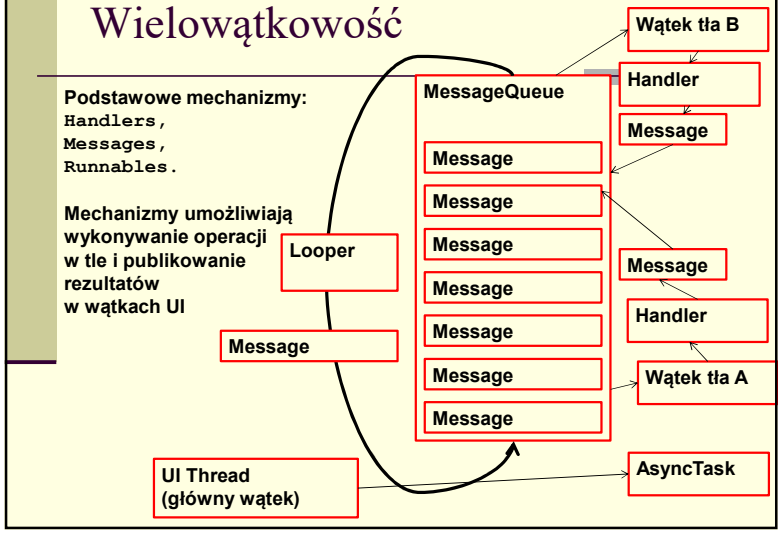
Android posiada kilka ograniczeń związanych z wątkami:

- **Wyjątek generowany w przypadku, gdy UI aplikacji nie odpowiada w krótkim czasie.**
- **Wątki nie związane z UI nie mogą mieć dostępu do widgetów dopóki nie będą bezpieczne (thread-safe).**
- **Android wspiera szereg rozwiązań do przetwarzania długich operacji w wątkach tła (umożliwia komunikację z wątkami UI).**

Wielowątkowość

Podstawowe mechanizmy:
`Handlers,`
`Messages,`
`Runnables.`

Mechanizmy umożliwiają wykonywanie operacji w tle i publikowanie rezultatów w wątkach UI



Wielowątkowość

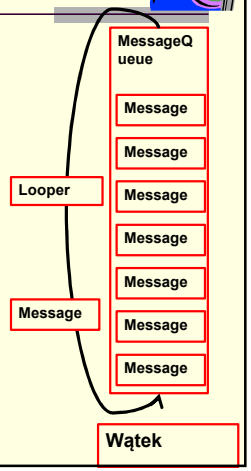
Klasa `Looper`:

`Looper` wykonuje operacje na skojarzonej kolejce komunikatów.

Wysyła wiadomości/obiekty `Runnable` do odpowiednich `handlerów` (pole `target`) w celu ich przetworzenia.

DOZWOLONY JEST TYLKO JEDEN LOOPER NA WĄTEK (ograniczenie osiągnięte przez użycie specyficznego sposobu przechowywania `ThreadLocal`)

Wątek UI posiada obiekt `Looper`, ale obiekty te mogą również zostać użyte przez wątki nie należące do UI.



Wielowątkowość

Metoda `Looper.loop()`:
- działa w pętli oczekując na komunikaty (`Message`),
- określa właściwy element, który zajmuje się obsługą danego typu komunikatu (`dispatch`)
- wywołuje element przetwarzający komunikat (`Handler`).

```
public class Looper{
        ...
        final MessageQueue mQueue;
        public static void loop(){
                ...
                for(;;){
                        Message msg= mQueue.next();
                        ...
                        msg.target.dispatchMessage(msg);
                        ...
                }
        ...
```

Wielowątkowość

Metoda `Looper.loop()`:
- działa w pętli oczekując na komunikaty (`Message`),
- określa właściwy element, który zajmuje się obsługą danego typu komunikatu (`dispatch`)
- wywołuje element przetwarzający komunikat (`Handler`).

```
public class Looper{
        ...
        final MessageQueue mQueue;
        public static void loop(){
                ...
                for(;;){
                        Message msg= mQueue.next();
                        ...
                        msg.target.dispatchMessage(msg);
                        ...
                }
        ...
}
```

Metoda blokująca – czeka na następną wiadomość.

Wielowątkowość

Metoda `Looper.loop()`:
- działa w pętli oczekując na komunikaty (`Message`),
- określa właściwy element, który zajmuje się obsługą danego typu komunikatu (`dispatch`)
- wywołuje element przetwarzający komunikat (`Handler`).

```
public class Looper{
        ...
        final MessageQueue mQueue;
        public static void loop(){
                ...
                for(;;){
                        Message msg= mQueue.next();
                        ...
                        msg.target.dispatchMessage(msg);
                        ...
                }
        ...
}
```

Przekazanie wiadomości do właściwego Handler'a

Wielowątkowość

Domyślnie wątki nie mają pętli komunikatów powiązanych z nimi

```
public class Thread implements Runnable{
        public static Thread currentThread(){
                ...
        }

        public final void join(){
                ...
        }

        public void interrupt(){
                ...
        }

        public synchronized void start(){
                ...
        }
}
```

Wielowątkowość

W celu stworzenia pętli komunikatów należy:
wywołać metodę `Looper.prepare()` w metodzie `run()`,
Stworzyć `Handler'a` przetwarzającego komunikaty,
Wywołać metodę `loop()` obsługującą `dispatch`'ing komunikatów.

```
class LooperThread extends Thread
{
        public Handler mHandler;
        public void run(){
                Looper.prepare();
                mHandler=new Handler(){
                        public void handleMessage(Message msg){
                                //przetwarzanie wiadomości
                        }
                }
                Looper.loop();
        }
}
```

plik LooperThread:

```
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
public class LooperThread extends Thread {
    public Handler mHandler;
    public void run(){
        Looper.prepare();
        mHandler=new Handler(){
            public void handleMessage(Message msg){
                Log.d("Wyklad ", msg.getData().getString("msg"));
            }
        };
        Looper.loop();
    }
}
```

Aktywność główna:

```
public class MainActivity extends AppCompatActivity {
    LooperThread lt;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button b = (Button) findViewById(R.id.button);
        lt=new LooperThread();
        lt.start();
        b.setOnClickListener(new View.OnClickListener(){

            @Override
            public void onClick(View view) {
                Message m = new Message();
                Bundle wiadomosc= new Bundle();
                wiadomosc.putString("msg", "wiadomosc");
                m.setData(wiadomosc);
                lt.mHandler.sendMessage( m);
            }
        });
    }
}
```

Aktywność główna:

```
public class MainActivity extends AppCompatActivity {
    LooperThread lt;
    @Override
```

```
12-06 11:26:45.822 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
12-06 11:26:48.391 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
12-06 11:26:48.740 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
12-06 11:26:49.043 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
12-06 11:26:49.347 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
12-06 11:26:49.633 3341-3551/pl.nysa.pwsz.wyklad D/Wyklad: wiadomosc
```

```
            @Override
            public void onClick(View view) {
                Message m = new Message();
                Bundle wiadomosc= new Bundle();
                wiadomosc.putString("msg", "wiadomosc");
                m.setData(wiadomosc);
                lt.mHandler.sendMessage( m);
            }
        });
    }
}
```

HandlerThread

Klasa `HandlerThread` jest klasą pomocniczą Android dla wątków, które automatycznie zawierają `Looper'a`

```
class HandlerThread extends Thread{
        Looper mLooper;
        ...
        public void run(){
                Looper.prepare();
                synchronized(this){
                        mLooper = Looper.myLooper();
                        ...
                }
                ...
                onLooperPrepared();
                Looper.loop();
                ...
        }
        protected void onLooperPrepared(){
        }
}
```

HandlerThread

Klasa `HandlerThread` jest klasą pomocniczą Android dla wątków, które automatycznie zawierają `Looper'a`

```
class HandlerThread extends Thread{
        Looper mLooper;
        ...
        public void run(){
                Looper.prepare();
                synchronized(this){
                        mLooper = Looper.myLooper();
                        ...
                }
                ...
                onLooperPrepared();
                Looper.loop();
                ...
        }
        protected void onLooperPrepared(){
        }
}
```

Metoda umożliwia klasom pochodnym stworzenie Hanlder'a

HandlerThread

```
public class LooperThread extends HandlerThread {
    public Handler mHandler;
    public LooperThread(String name) {
        super(name);
    }
    @Override
    protected void onLooperPrepared(){
        mHandler=new Handler(){
            public void handleMessage(Message msg){
                Log.d("Wyklad ", msg.getData().getString("msg"));
            }
        };
    }
}
```

Handler

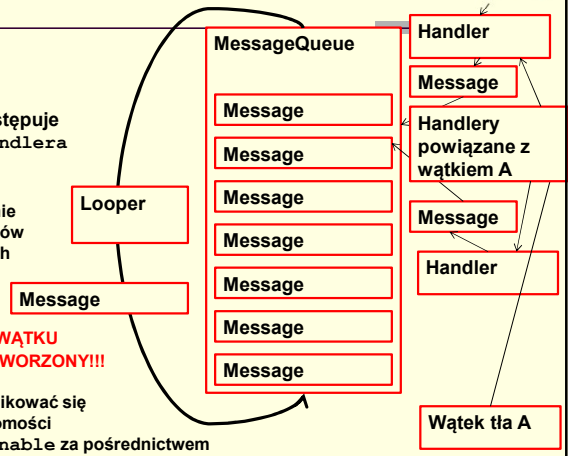
Klasa `Handler`

Większość interakcji z wiadomościami następuje z wykorzystaniem `Handlera`

`Handler` umożliwia wysyłanie i przetwarzanie wiadomości oraz obiektów `Runnable` powiązanych z kolejką komunikatów wątku

HANDLER NALEŻY DO WĄTKU W KTÓRYM ZOSTAŁ STWORZONY!!!

Inne wątki mogą komunikować się poprzez wymianę wiadomości lub obiektów klasy `Runnable` za pośrednictwem Handler'a lub Hanlderów należących do wątku.



Handler

Klasa `Handler`

Każdy obiekt `Handler` jest powiązany z pojedynczym wątkiem i jego kolejką komunikatów (`MessageQueue`).

Podczas tworzenia nowego `Handlera`, jest on przydzielany do `Looper'a` należącego do wątku, w którym `Handler` jest tworzony.

Jeden wątek może mieć wiele `Handlerów` (w przeciwieństwie do `Loopera`)

```
public class Handler{
        ...
        public void handleMessage(Message msg){
                ...
        }
}
```

Klasy potomne muszą nadpisywać metodę w celu przetwarzania wiadomości

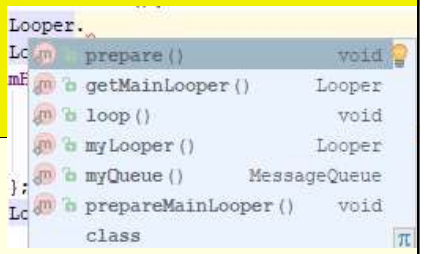
Handler

Klasa Handler

Konstruktor Handler'a upewnia się, że obiekt posiada zainicjalizowanego Looper'a

```
Konstruktor Handler'a:
public Handler(){
        mLooper = Looper.myLooper();
        if(mLooper==null) throw new RuntimeException(„nie można
        stworzyć hanldera w wątku, który nie wywołał
        Looper.prepare()”);

        mQueue = mLooper.mQueue;
        ...
```

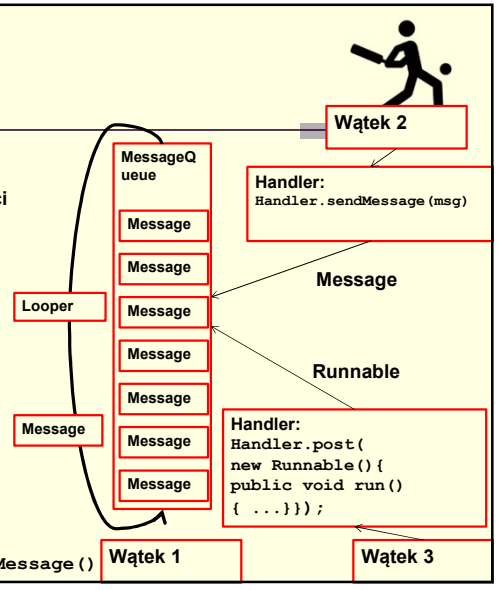


Handler

Możliwości Handlera:

Wysyłanie wiadomości i obiektów `Runnable` do wątku

Implementuje bezpieczny sposób przetwarzania komunikatów



Handler

Metody `Handler'a` powiązane z obiektami `Runnable`:

`boolean post(Runnable r)` – dodaje obiekt `Runnable` do `MessageQueue`

`boolean postAtTime(Runnable r, long uptimeMillis)` – dodaje `Runnable` do `MessageQueue`. Uruchamia w określonym czasie (określonym przez `SystemClock.uptimeMillis()`)

`boolean postDelayed(Runnable r, long delayMillis)` – dodaje `Runnable` do `MessageQueue`. Uruchamia po upływie określonego czasu.

Handler

Metody `Handler'a` powiązane z obiektami `Message`:

`boolean sendMessage(Message msg)` – umieszcza wiadomość w kolejce natychmiast

`boolean sendMessageAtFrontOfQueue(message msg)` – umieszcza wiadomość w kolejce natychmiast na jej początku

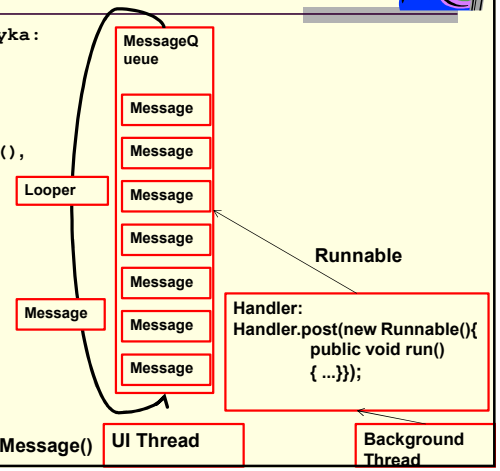
`boolean sendMessageAtTime(Message msg, long uptimeMillis)` – umieszcza wiadomość w kolejce o określonym czasie.

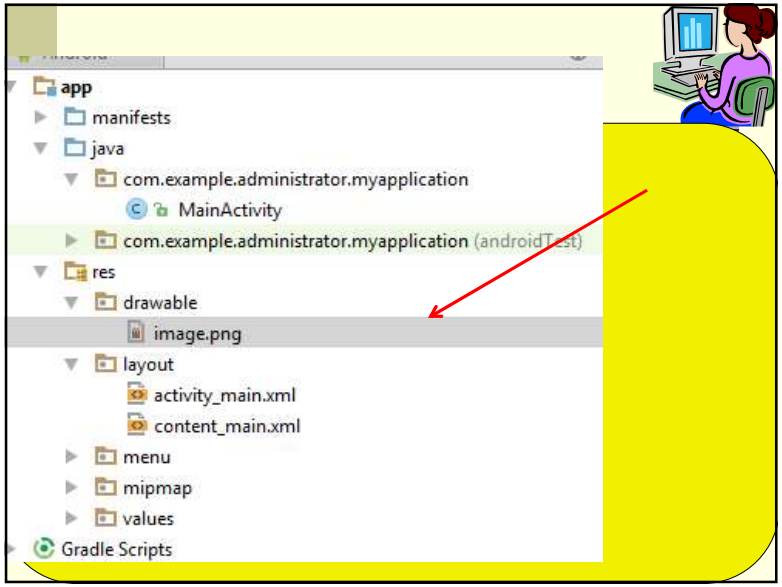
`boolean sendMessageDelayed(Message msg, long delayMillis)` – umieszcza wiadomość w kolejce po upływie określonego czasu.

Handler

`Runnables` - metodyka:
programowanie:

1. Stworzyć obiekt `Runnable`,
2. stworzyć metodę `run()`,
3. przekazać obiekt do `Handler'a`
4. W wątku adresata `Looper` wywoła metodę `run()` w kontekście wątku adresata.





Handler

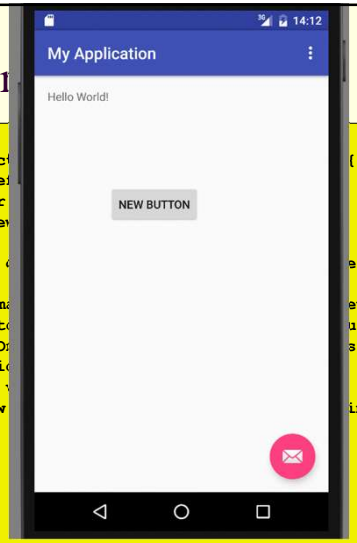
```
public class LoadImage implements Runnable {
        int resId;
        LoadImage(int resId){this.resId=resId;}
        public void run(){
            final Bitmap tmp =
BitmapFactory.decodeResource(getResources(), resId);
            h.post(new Runnable(){
                public void run(){
                    iview.setImageBitmap(tmp);
                }
            });
        }
    }
```

Handler

```
public class MainActivity extends AppCompatActivity {
    ... //Klasa zdefiniowana na poprzednim slajdzie
    private Handler h = new Handler();
    public ImageView iview;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        iview = (ImageView)findViewById(R.id.imageView);
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new Thread(new LoadImage(R.drawable.image)).start();

            }
        });
    }
```

Handler



Handler

Messages - metodyka programowanie:

1. Stworzyć klasę dziedziczącą po `Handler`
2. Stworzyć `handleMessage`
3. Stworzyć obiekt Wiadomości
4. Ustawić zawartość wiadomości
5. `Handler.obtainMessage()`
   `Message.obtain()`
   ...

`handleMessage()`

Looper

MessageQueue

Message

Message

Message

Message

Message

Message

Message

Message

Parametry wiadomości zawierają (wiele metod przeciążonych) mogą zawierać:
- `int arg1, arg2`
- `int what`
- `Object obj`
- `Bundle data`

Message

Handler:
Handler h = new Handler(){
public void handleMessage(Message msg){...}
}

UI Thread

Background Thread

Handler

Messages - metodyka:
programowanie:

1. - Stworzyć klasę dziedziczącą po Handler
2. -
3. - Wi
4. - wia
5. - Handler.obtainMessage()
   - Message.obtain()
   - ...

Metody umożliwiają utworzenie obiektu wiadomości

MessageQueue

Message
Message
Message
Message
Message
Message
Message

Parametry wiadomości zawierają (wiele metod przeciążonych) mogą zawierać:
- int arg1, arg2
- int what
- Object obj
- Bundle data

Message

Handler:
Handler h = new Handler(){
public void handleMessage(Message msg){...}
}

handleMessage() | UI Thread | Background Thread

Handler

6. W kolejnym kroku Looper wywoła handleMessage() w wątku UI

MessageQueue

Looper

Message

Message
Message
Message
Message
Message
Message
Message
Message

Parametry wiadomości zawierają (wiele metod przeciążonych) mogą zawierać:
- int arg1, arg2
- int what
- Object obj
- Bundle data

Message

Handler:
Handler h = new Handler(){
public void handleMessage(Message msg){...}
}

handleMessage() | UI Thread | Background Thread

Handler

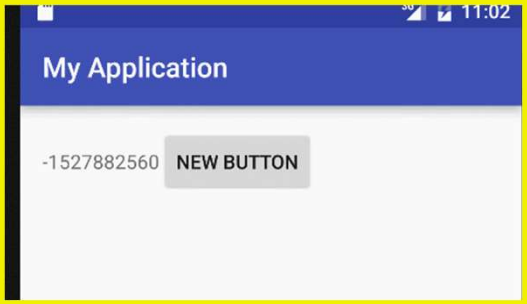
```
public class MainActivity extends AppCompatActivity {
private Handler h = new Handler(){
  public void handleMessage(Message msg)
  {
    switch(msg.what)
    {
      case 1: {
        TextView t = (TextView) findViewById(R.id.textView1);
        t.setText(String.valueOf((Integer) msg.obj));
        break;
      }
    }
  }
};
```

Handler

```
public class SendRandom implements Runnable {
        public void run(){
            while(true) {
                Random r = new Random();
                Integer i = r.nextInt();
                Message msg = h.obtainMessage(1, i);
                h.sendMessage(msg);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

Handler

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        new Thread(new SendRandom()).start();
    }
```



Handler

Ustawienia czasowe

Handler

```
public class LooperThread extends HandlerThread {
    public Handler mHandler;

    public LooperThread(String name) {
        super(name);
    }
    protected void onLooperPrepared(){
        mHandler=new Handler(){
            public void handleMessage(Message msg){
                Log.d("Wyklad ", SystemClock.uptimeMillis()+"
"+msg.getData().getString("msg"));
            }
        };
    }
}
```

```
public class MainActivity extends AppCompatActivity {
    LooperThread lt;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        lt=new LooperThread("Messages");
        lt.start();
        Button b= (Button) findViewById(R.id.button);
        b.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view) {
                Log.d("Wyklad", "KLIK");
                Message m = new Message();
                Bundle wiadomosc= new Bundle();
                wiadomosc.putString("msg", "wiadomosc 1");
                m.setData(wiadomosc);
 lt.mHandler.sendMessageAtTime(m, SystemClock.uptimeMillis()+2000);
                m=new Message();
                wiadomosc=new Bundle();
                wiadomosc.putString("msg", "wiadomosc 2");
                m.setData(wiadomosc);
                lt.mHandler.sendMessageDelayed(m, 3000);
            }
        });
    }
}
```

```
public class MainActivity extends AppCompatActivity {
    LooperThread lt;
    @Override
```

Terminal 0: Messages

```
                m.setData(wiadomosc);
                lt.mHandler.sendMessageDelayed(m, 3000);
            }
        });
    }
```

Handler

Inny przykład

Handler

```
public class LoadImage implements Runnable {
        int resId;
        int operacja;
        LoadImage(int resId, int operacja){
                this.resId=resId;
                this.operacja=operacja;
        }
        public void run(){
            if(operacja==1) {
                Message msg = h.obtainMessage(1, ImageView.VISIBLE);
                h.sendMessage(msg);
            } else if(operacja==2){
                Message msg = h.obtainMessage(2, resId);
                h.sendMessage(msg);
            }
        }
    }
```

Handler

```
public class LoadImage2 implements Runnable {
        int bitm;
        LoadImage2(int bitm){this.bitm=bitm; }
        public void run(){
            tmp = BitmapFactory.decodeResource(getResources(), bitm);
            tmp = tmp.createScaledBitmap(tmp, 100,100, false);
            czy=true;
        }
    }
```

Handler

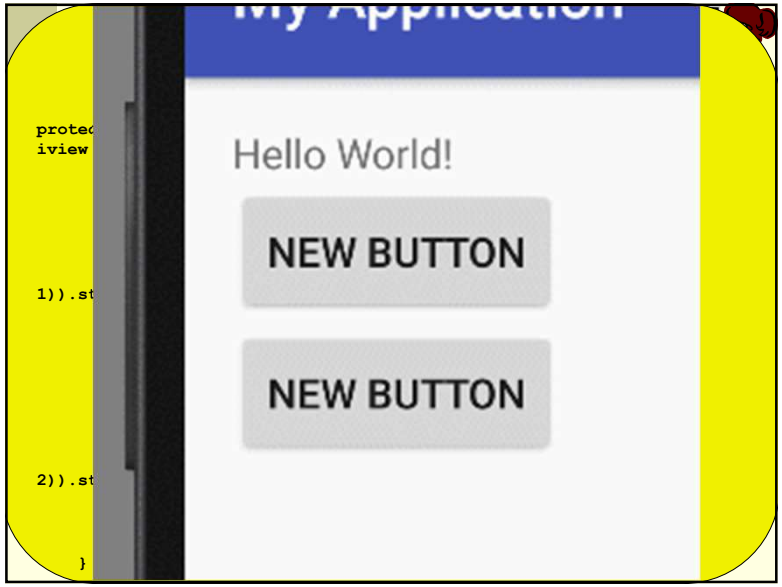
```
private Handler h = new Handler(){
     public void handleMessage(Message msg)
     {
          switch(msg.what)
          {
              case 1:{
                  iview.setVisibility(ImageView.VISIBLE); break;
              }
              case 2:{
            new Thread( new LoadImage2((Integer) msg.obj)).start();
                  while(!czy);
                  iview.setImageBitmap(tmp);
                  break;
              }
          }
     }
   };
```

```
        public ImageView iview;
        Bitmap tmp;
        boolean czy=false;

protected void onCreate(Bundle savedInstanceState) {
iview = (ImageView)findViewById(R.id.imageView2);
        //iview.setVisibility(ImageView.INVISIBLE);
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new Thread(new LoadImage(R.drawable.image,
1)).start();

            }
        });
        Button button2 = (Button) findViewById(R.id.button2);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                new Thread(new LoadImage(R.drawable.image,
2)).start();

            }
        });
    }
```



AsyncTask

AsyncTask – klasa upraszcza tworzenie długotrwałych zadań, które muszą komunikować się z interfejsem użytkownika.

```
class klasa extends AsyncTask<Typ1, Typ2, Typ3>
{
        protected Typ3 doInBackground(Typ1... typ1){
                ...
        }
        protected void onProgressUpdate(Typ2... typ2){
                ...
        }
        protected void onPostExecute(Typ3 typ3){
                ...
        }
}

        ...
new klasa().execute(typ1_a, typ1_b, typ1_c);
```

AsyncTask

Typy uogólnione `AsyncTask<Typ1, Typ2, Typ3>`:

1. `Typ1` – typ parametrów przekazywany zadaniu przy wywołaniu,

2. `Typ2` – typ jednostki postępu publikowany w trakcie wykonywania zadania w tle

3. `Typ3` – typ wyniku zadania tła

Metody:

`onPreExecute()` – wywoływana na wątku UI przed wykonaniem zadania tła.

`doInBackground(Typ1 ...)` – wywoływana w tle – typowo do zadań długotrwałych. Metoda może używać `publishProgress` w celu opublikowania postępu. Przekazana wartość jest używana przez `onProgressUpdate` w wątku UI.

`onProgressUpdate( Typ2...)` – wywoływana na wątku UI po wywołaniu `publishProgress`.

`onPostExecute( Typ3 )` – wywoływana na wątku UI po zakończeniu obliczeń w tle.

AsyncTask

`AsyncTask` można zakończyć wywołując metodę:

`cancel(boolean)`

Wywołanie metody spowoduje, że metoda `isCancelled()` zwróci `true`.

Po wywołaniu metody wykonywana jest `onCancelled(Object)` zamiast `onPostExecute`

```
public class MainActivity extends AppCompatActivity {
class myAsync extends AsyncTask<URL, Integer, String>{
  public Context c;
  ProgressBar postep;
  TextView out;
  protected void onPreExecute(){
      postep = new ProgressBar(c, null,
    android.R.attr.progressBarStyleHorizontal);
      postep.setMax(99);
      postep.setMinimumWidth(500);
      postep.setMinimumHeight(50);
      Button p1 = new Button(c);
      p1.setMinimumWidth(200);
      p1.setMinimumHeight(100);
      p1.setText("PRZERWIJ");
      p1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
          cancel(true);
        }
      });
      out = new TextView(c);
      LinearLayout l = (LinearLayout) findViewById(R.id.lin);
      l.addView(postep);
      l.addView(p1);
      l.addView(out);
}
```

```
protected String doInBackground(URL... urls){
  String result="";
  int postep=0;
  for(URL x: urls){
    if(isCancelled()) return null;
    try {
      BufferedReader in = new BufferedReader(new
InputStreamReader(x.openStream()));
      String add;
      while((add=in.readLine())!=null){
        result+="\n"+add;
      }
      in.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
    postep++;
    publishProgress(postep);
  }
  return result;
}
```

AsyncTask

```
@Override
protected void onProgressUpdate(Integer... i){
  postep.setProgress(i[0]*33);
}

protected void onPostExecute(String tekst){
  out.setText(tekst);
}

protected void onCancelled(String x){
  out.setText("Niestety");
}

}
```

AsyncTask

```
@Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       myAsync x = new myAsync();
       x.c=this;
       try {
           x.execute(new URL("http://www.pwsz.nysa.pl"), new
URL("http://www.rolniknysa.pl"), new URL("http://www.tlen.pl"));
       } catch (MalformedURLException e) {
           e.printStackTrace();
       }

   }
}
```