UA . MASTER MOVILES

MÁSTER UNIVERSITARIO EN DESARROLLO DE SOFTWARE PARA DISPOSITIVOS MÓVILES

PROGRAMACIÓN HIPERMEDIA PARA DISPOSITIVOS MÓVILES

Android – Acceso Web

CONTENIDOS

- Acceso a URLs
- Conexiones asíncronas
- Estado de la red
- Carga *lazy* de imágenes
  - Bloquear la carga cuando se hace *scroll*
  - Referencias débiles

CONSIDERACIONES PREVIAS

- Características del acceso a la red en móviles:
  - Según la cobertura, el acceso puede ser lento.
  - La conexión puede perderse en ciertos momentos.
  - Límite de descarga o tarificación por consumo.

- Recomendaciones:
  - Realizar la conexión en segundo plano.
  - Tener en cuenta el tipo de red a la hora de conectar.
  - Minimizar el tráfico de datos.

PERMISOS DE ACCESO A LA RED

- En Android debemos solicitar permiso para acceder a Internet.
- Añadir etiqueta `uses-permission` a `AndroidManifest.xml`
- Fuera de la sección "`application`" añadimos la etiqueta:

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

- Si se nos olvida hacer esto, nos parecerá que el móvil no tiene conexión a Internet.
  - A partir de la versión 3.0 de Android aparecerá el error:

    `java.lang.SecurityException: Permission denied (missing INTERNET permission?)`

ACCESO A URLS

- Es la forma más común de acceso a la red.
- Una URL localiza de forma única un recurso:

```
URL url = new URL("https://eps.ua.es");
```

- El acceso a URLs se realiza mediante el protocolo HTTP.
- Conectando a la URL podemos obtener su contenido:
  - Por ejemplo, leer el HTML de una página.
- Podemos también enviar datos al servidor usando:
  - Parámetros de la *query*: http://www.domain.es/user.php?nombre=Javi&apellidos=gallego
  - En el bloque de contenido de la petición HTTP

ACCESO A URLS

- En Android se puede utilizar la misma API que en Java SE:

```
TextView tv = (TextView)findViewById(R.id.tvVisor);
tv.setText("Conexión http.\n\n");

try {
  URL url = new URL("https://eps.ua.es");
  HttpURLConnection http = (HttpURLConnection)url.openConnection();

  tv.append("Cabeceras de https://eps.ua.es: \n");
  tv.append("longitud      = "+http.getContentLength()+"\n");
  tv.append("encoding      = "+http.getContentEncoding()+"\n");
  tv.append("tipo          = "+http.getContentType()+"\n");
  tv.append("resp. code    = "+http.getResponseCode()+"\n");
  tv.append("resp. message = "+http.getResponseMessage()+"\n");
  tv.append("content stream= "+http.getContent().toString()+"\n");
} catch (MalformedURLException e) {
} catch (IOException e) {
}
```

ACCESO A URLS

- Es importante comprobar el código de estado devuelto por la petición.
- Normalmente devolverá el código 200 en caso de que la petición sea correcta y un código distinto si hay algún error.
- La lista de todos los códigos de estado la podéis consultar en: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- El código lo podemos comprobar de la forma:

```
URL url = new URL("https://eps.ua.es");
HttpURLConnection http = (HttpURLConnection)url.openConnection();

if( http.getResponseCode() == 200 ) {
     // Petición correcta!
}
// O también podemos hacer...
if( http.getResponseCode() == HttpURLConnection.HTTP_OK ) {
     // Petición correcta!
}
```

ACCESO A URLS. DESCARGAR CONTENIDO

```
public String descargarContendio( String strUrl ) {
    HttpURLConnection http = null;
    String content = null;
    try {
        URL url = new URL( strUrl );
        http = (HttpURLConnection)url.openConnection();

        if( http.getResponseCode() == HttpURLConnection.HTTP_OK ) {
            StringBuilder sb = new StringBuilder();
            BufferedReader reader = new BufferedReader(
                    new InputStreamReader( http.getInputStream() ));
            String line;
            while ((line = reader.readLine()) != null) {
                sb.append(line);
            }
            content = sb.toString();
            reader.close();
        }
    }
    catch(Exception e) {        e.printStackTrace();    }
    finally {
        if( http != null )      http.disconnect();
    }
    return content;
}
```

Leemos todos los datos del *stream* de entrada

Cerrar conexión

ACCESO A URLS. DESCARGAR IMÁGENES

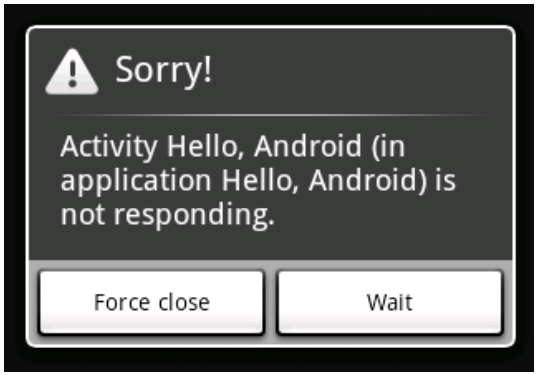
```
public Bitmap descargarImagen(String strUrl) {
    HttpURLConnection http = null;
    Bitmap bitmap = null;

    try {
        URL url = new URL( strUrl );
        http = (HttpURLConnection)url.openConnection();

        if( http.getResponseCode() == HttpURLConnection.HTTP_OK )
            bitmap = BitmapFactory.decodeStream(http.getInputStream());
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    finally {
        if( http != null )
            http.disconnect();
    }
    return bitmap;
}
```

PROBLEMAS DE LAS CONEXIONES SÍNCRONAS

- Las conexiones de red en móviles pueden ser muy lentas.
- Al conectar de forma **síncrona** bloqueamos la aplicación.
- Puede aparecer el diálogo "*Application Not Responding*" (ANR).



- Invita al usuario a "matar" nuestra aplicación.
- A partir de la versión 3.0 de Android: `NetworkOnMainThreadException`
- Debemos evitar que esto ocurra.

CONEXIONES ASÍNCRONAS

- Se realizan mediante un hilo en segundo plano.
- En Android **tenemos que crear nosotros el hilo** en el que ejecutar la conexión.

  → Podremos ayudarnos de la clase *AsyncTask*.
- La aplicación puede seguir funcionando mientras se descargan los datos.
- Debemos evitar el abuso de diálogos de progreso que entorpecen el uso de la aplicación.

CONEXIÓN ASÍNCRONA EN ANDROID

- Podemos crear un hilo al igual que en Java SE.
- **Problema**: No podemos acceder a la UI desde los hilos secundarios.
- **Solución**: Utilizar `runOnUiThread`

```
ImageView imageView = (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
  public void run() {
    final Drawable imagen = descargarImagen("http://...");
    runOnUiThread(new Runnable() {
      public void run() {
        imageView.setDrawable(imagen);
      }
    });
  }
}).start();
```

- Genera un código demasiado complejo.

TAREAS ASÍNCRONAS

- La clase *AsyncTask* nos proporciona una forma elegante de crear tareas asíncronas que actualizan la interfaz:

```
class MiTarea extends AsyncTask<ENTRADA, PROGRESO, SALIDA>
{
    protected SALIDA doInBackground(ENTRADA...)
    {
        publishProgress( PROGRESO );
        return SALIDA;
    }

    protected void onPreExecute() { }
    protected void onProgressUpdate(PROGRESO) { }
    protected void onCancelled() { }
    protected void onPostExecute(SALIDA) { }
}
```

TAREAS ASÍNCRONAS

- Para llamar a una tarea asíncrona se realizaría de la forma:

```
new miTarea().execute( ENTRADA );
```

- Por ejemplo:

```
new miTarea().execute( "url1", "url2", "url3" );
```

- Desde el método "`doInBackground(String... urls)`" podremos:
  - Obtener el número de elementos de entrada: `urls.length;`
  - Acceder a su contenido: `urls[0], urls[1], ...`

TAREAS ASÍNCRONAS, EJEMPLO 1

```
private class DownloadTask extends AsyncTask<String, Void, String>
{
    @Override
    protected String doInBackground(String... urls)
    {
        // Llamada al método de descarga de contenido
        return descargarContendio( urls[0] );
    }

    @Override
    protected void onPreExecute() {}

    @Override
    protected void onCancelled() {}

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

TAREAS ASÍNCRONAS, EJEMPLO 2

```
private class DownloadTask extends AsyncTask<String, Integer,
                                              List<Drawable>>
{
     protected List<Drawable> doInBackground(String... urls) {
          ArrayList<Drawable> imagenes = new ArrayList<Drawable>();
          for(int i=1;i<urls.length; i++) {
               imagenes.add( descargarImagen(urls[0]) );
               publishProgress(i);
          }
          return imagenes;
     }

     protected void onPreExecute() { }
     protected void onProgressUpdate(Integer... values) { }
     protected void onCancelled() { }

     protected void onPostExecute(List<Drawable> result) {
          for(int i=0; i<result.length; i++){
               imageView[i].setDrawable(result.getItemAt(i));
          }
     }
}
```

ESTADO DE LA RED

- Es importante saber si disponemos de conexión a Internet o no.
- Para esto primero tenemos que solicitar permiso en el Manifest:

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

- Y después podremos utilizar `ConectivityManager` de la forma:

```
boolean checkStatus(Context ctx) {
   ConnectivityManager cm = (ConnectivityManager)
                ctx.getSystemService(Context.CONNECTIVITY_SERVICE);
   NetworkInfo i = cm.getActiveNetworkInfo();

   if (i == null || !i.isAvailable() || !i.isConnected())
       return false;  // Inteeerneeer!!

   return true;
}
```

ESTADO DE LA RED

- También podemos consultar el tipo de conexión que usa el dispositivo.
- Algunas operaciones pesadas podemos/deberemos realizarlas solo cuando contemos con Wi-Fi (o al menos preguntar al usuario).

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

NetworkInfo i = cm.getActiveNetworkInfo();

if( i.getType() == ConnectivityManager.TYPE_MOBILE )
    // Conectado por datos
else if( i.getType() == ConnectivityManager.TYPE_WIFI )
    // Conectado a una red wifi
```

CARGA *LAZY* DE IMÁGENES

- Se utiliza habitualmente para descargar una lista o *grid* de imágenes.
- El objetivo es evitar descargar imágenes de forma innecesaria y dar una mejor respuesta al usuario.
- Solo se descargarán las imágenes de las filas que se muestran en pantalla y al hacer *scroll* se irán solicitando las nuevas imágenes.
- Optimizaciones:
  - Solo descargar cuando no se esté haciendo *scroll*.
  - Liberar las imágenes descargadas cuando falte memoria (`SoftReference` en Java).
  - Almacenar las imágenes de forma persistente en BD o caché.

CARGA *LAZY* DE IMÁGENES

- Iniciaremos la carga en el ***adapter*** cuando se solicite cada item:

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
{
    //...
    if( elemento.getImagen()!=null ) {
       ivIcono.setImageBitmap(elemento.getImagen());
    } else {
       // Si la imagen no está descargada...
       ivIcono.setImageResource(R.drawable.icon); // Icono temporal
       descargarImagen(elemento, ivIcono); // Iniciamos la descarga
    }
    return convertView;
}
```

Comprobamos si ya está descargada

- Si la imagen no ha sido descargada todavía, ponemos una imagen temporal e iniciamos la descarga.
- **¡Cuidado!** Si hacemos *scroll* arriba y abajo corremos el riesgo de que se vuelva a solicitar una imagen cuya descarga está en curso.

CONTROL DE DESCARGAS REPETIDAS

- Creamos un mapa con las descargas en curso:

```
Map<Elemento, CargarImagenTask> mImagenesCargando;
```

- Sólo iniciamos la tarea de descarga si no hay ninguna en marcha para el mismo item.
- En el método "`getView`" tendríamos que hacer:

```
// ...
if(elemento.getImagen()!=null) {
    ivIcono.setImageBitmap(elemento.getImagen());
} else {
    ivIcono.setImageResource(R.drawable.icon); // Icono temporal
    // Si la imagen no está descargando...
    if(mImagenesCargando.get(elemento)==null) {
        CargarImagenTask task = new CargarImagenTask();
        mImagenesCargando.put(elemento, task);
        task.execute(elemento, ivIcono);
    }
}
```

CARGA *LAZY* SOLO CUANDO NO SE HACE *SCROLL*

- Para esto tenemos que hacer que el adaptador implemente:

```
AbsListView.OnScrollListener
```

- Y registrarlo como oyente a los eventos de *scroll* de la lista:

```
miListView.setOnScrollListener(adaptador);
```

- Por último, en el *adapter*, tendremos que:
  - Crear una variable para bloquear la descarga cuando se haga *scroll*.
  - Implementar los métodos del `OnScrollListener`.

CARGA *LAZY* SOLO CUANDO NO SE HACE *SCROLL*

```
public class ImagenAdapter extends BaseAdapter implements OnScrollListener {
  private boolean mBusy = false;
  // ...
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
      //...
      // Si la imagen no está cargada y no se está haciendo SCROLL...
      if(mImagenesCargando.get(elemento)==null  && !mBusy) {
          //...
      }
      return convertView;
  }
  public void onScroll(AbsListView v, int f, int c, int t) { }
  public void onScrollStateChanged(AbsListView view, int scrollState) {
    switch(scrollState) {
      case OnScrollListener.SCROLL_STATE_IDLE:
        mBusy = false;
        notifyDataSetChanged();
        break;
      case OnScrollListener.SCROLL_STATE_TOUCH_SCROLL:   mBusy = true;   break;
      case OnScrollListener.SCROLL_STATE_FLING:          mBusy = true;   break;
    }
  }
}
```

REFERENCIAS *DÉBILES* A ELEMENTOS

- Las referencias débiles son interesantes para listados muy largos de elementos que puedan consumir mucha memoria.
- Para esto podemos utilizar la clase "`SoftReference`".
- Esta clase permite a Java liberar la memoria en caso de ser necesario.
- Para el *lazy loading* podremos crear el `Bitmap` como un objeto de este tipo de la forma:

```
public class Elemento {
      // …
      SoftReference<Bitmap> imagen;
      // …
}
```

REFERENCIAS *DÉBILES* A ELEMENTOS

- Para obtener el elemento almacenado en un objeto del tipo "`SoftReference`" tenemos que usar su método "`get`".
- Por ejemplo, ahora en el *adapter* tendríamos que hacer:

```
if(elemento.getImagen().get()!=null) {
   ivIcono.setImageBitmap( elemento.getImagen().get() );
} else {
   //...
}
```

- Y para crear una referencia nueva tenemos que utilizar el constructor de "`SoftReference`" de la forma:

```
protected void onPostExecute(Bitmap result) {
    if(result!=null) {
         this.elemento.setImagen(new SoftReference<>(result));
         this.view.setImageBitmap(result);
    }
}
```

¿PREGUNTAS?