UNIVERSITÀ DI SALERNO

Corso di laurea in Informatica

Mobile Programming

Prof. De Prisco

Prova scritta del

esempio 2

NOME: _____

COGNOME:_____

MATRICOLA: _____

Domande	Punti
1	/10
2	/10
3	/10
4	/10
5	/10
6	/10
7	/10
8	/10
9	/10
10	/10
TOTALE	/100

Il layout di un app può essere definito sia staticamente, tramite un file XML, che programmaticamente tramite istruzioni nel programma. Si discuta dei vantaggi e svantaggi e si faccia un esempio di un caso in cui è possibile usare solo uno dei due e non l'altro, motivando la risposta.

Il layout di un'app Android può essere definito sia staticamente, tramite un file XML, che programmaticamente tramite istruzioni nel codice Java dell'app.

Vantaggi della definizione del layout staticamente, tramite un file XML:

Il layout è più facile da modificare e mantenere. Se vuoi apportare modifiche al layout, puoi farlo modificando direttamente il file XML, senza dover modificare il codice dell'app.
Il layout è più facile da leggere e comprendere. Con il layout definito in un file XML, puoi facilmente vedere come gli elementi del layout sono organizzati e come vengono utilizzati gli attributi per definire le proprietà degli elementi.
Vantaggi della definizione del layout programmaticamente:

Il layout può essere dinamico. Se hai bisogno di creare elementi del layout o modificare le loro proprietà in base ai dati o alle azioni dell'utente, puoi farlo programmaticamente.
Il layout può essere ottimizzato in modo più efficace. Se conosci in anticipo quali elementi del layout verranno utilizzati e in quale ordine, puoi crearli programmaticamente in modo da ottenere un'app più performante rispetto alla creazione di tutti gli elementi del layout staticamente.
Un esempio di un caso in cui è possibile usare solo la definizione del layout programmaticamente è quando hai bisogno di creare elementi del layout dinamicamente in base ai dati o alle azioni dell'utente. Ad esempio, immagina di avere un'app che mostra una lista di prodotti in una ListView. Se vuoi aggiungere un nuovo prodotto alla lista, dovresti creare un nuovo elemento del layout per il prodotto programmaticamente, poiché non sai in anticipo quanti prodotti verranno mostrati nella lista.

Gli svantaggi della definizione del layout staticamente, tramite un file XML:

Il layout può diventare complesso se hai molti elementi o se l'organizzazione degli elementi è complessa. Ciò può rendere difficile la modifica o il debug del layout.
Il layout può essere meno performante se hai molti elementi o se utilizzi elementi di layout anidati (ad esempio, un LinearLayout all'interno di un altro LinearLayout). Ciò può rallentare il rendering del layout sulla schermata.
Gli svantaggi della definizione del layout programmaticamente:

Il layout può diventare difficile da leggere e comprendere se è stato creato utilizzando molte istruzioni nel codice. Ciò può rendere difficile la modifica o il debug del layout.
La creazione di elementi del layout programmaticamente può essere più lenta rispetto alla creazione di elementi del layout staticamente. Ciò può ridurre le prestazioni dell'app se si creano molti elementi del layout dinamicamente.

Un listview prevede on OnItemClickListener che gestisce i click sugli elementi della lista chiamando il metodo onItemClick al quale viene passato un riferimento dell'elemento selezionato. Se usiamo un listview customizzato, in cui ogni elemento della lista è composto da vari sottoelementi (es. una foto, un nome, un numero), il riferimento passato al metodo onItemClick non distingue quale dei sottoelementi è stato selezionato. Come si può fare per reagire in maniera diversa in funzione di quale dei sottoelementi è stato selezionato con il click?

per personalizzare gli elementi di una ListView puoi creare un file di layout personalizzato che descrive l'aspetto di ogni elemento della lista. Inoltre, puoi creare una classe CustomAdapter che estende ArrayAdapter e sovrascrive il metodo getView() per creare gli elementi della lista utilizzando il layout personalizzato.

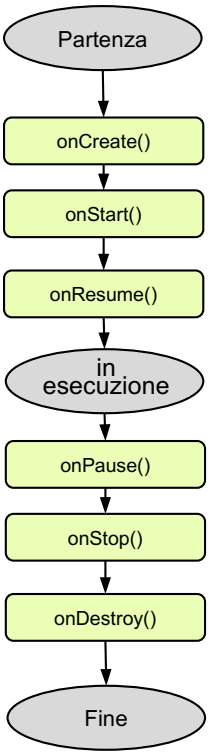
Per gestire il click multiplo sugli elementi della ListView, non puoi più utilizzare il listener del ListView, poiché questo viene chiamato solo per il click sull'intero elemento della lista. Invece, devi impostare listeners ad-hoc per ogni sottoelemento del layout personalizzato. Tuttavia, questo può creare il problema di non sapere più in quale posizione dell'array di dati sei.

Per risolvere questo problema, puoi utilizzare il metodo setTag() e getTag() per associare un valore (ad esempio, la posizione dell'elemento della lista) a un sottoelemento del layout personalizzato. Ad esempio:

```
public class CustomAdapter extends ArrayAdapter<Contatto> {
   private LayoutInflater inflater;

   public CustomAdapter(Context context, int resourceId, List<Contatto> objects) {
         super(context, resourceId, objects);
         inflater = LayoutInflater.from(context);
   }

   @Override
   public View getView(int position, View v, ViewGroup parent) {
                  if (v == null) {
                        Log.d("DEBUG","Inflating view");
                        v = inflater.inflate(R.layout.list_element, null);
                  }

         Contatto c = getItem(position);

                        Log.d("DEBUG","contact c="+c);

         Button nameButton;
         Button telButton;
         ImageButton fotoButton;

         nameButton = (Button) v.findViewById(R.id.elem_lista_nome);
         telButton = (Button) v.findViewById(R.id.elem_lista_telefono);
         fotoButton = (ImageButton) v.findViewById(R.id.elem_lista_foto);

         fotoButton.setImageDrawable(c.getPicture());
         nameButton.setText(c.getName());
         telButton.setText(c.getTel());

         fotoButton.setTag(position);
         nameButton.setTag(position);
         telButton.setTag(position);

         return v;
   }
}
```

```
public void onPictureClick(View v) {
                  Log.d("DEBUG","Picture has been clicked:
position="+v.getTag());
                  int position = Integer.parseInt(v.getTag().toString());
                  Contatto c = customAdapter.getItem(position);
            Toast.makeText(getApplicationContext(),
                  "Click su Foto - posizione n."+position+": "
+c.getName(), Toast.LENGTH_LONG)
                  .show();
      }

      public void onNameClick(View v) {
                  Log.d("DEBUG","Name has been clicked
position="+v.getTag());
                  int position = Integer.parseInt(v.getTag().toString());
                  Contatto c = customAdapter.getItem(position);
            Toast.makeText(getApplicationContext(),
                  "Click su Nome - posizione n."+position+": "
+c.getName(), Toast.LENGTH_LONG)
                  .show();
      }

      public void onTelClick(View v) {
                  Log.d("DEBUG","Tel has been clicked
position="+v.getTag());
                  int position = Integer.parseInt(v.getTag().toString());
                  Contatto c = customAdapter.getItem(position);
            Toast.makeText(getApplicationContext(),
                  "Click su Tel - posizione n."+position+": "
+c.getName(), Toast.LENGTH_LONG)
                  .show();   }
```

Quesito 3

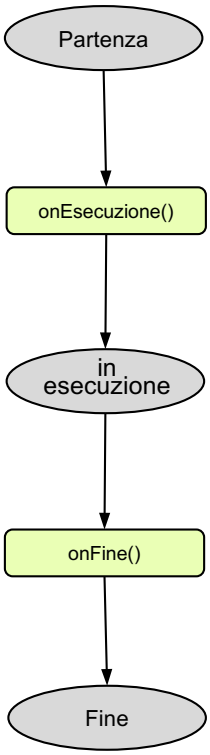


Il ciclo di vita delle activity, riportato schematicamente a sinistra, prevede l'esecuzione in successione di 3 metodi (onCreate, onStart, onResume) per far partire l'esecuzione di un'app.

Pechè? Non sarebbe stato meglio avere un solo metodo, come indicato nella figura a destra, nel quale eseguire tutto ciò che viene fatto nei 3 metodi onCreate, onStart, onResume?

Analogamente per distruggere un app è prevista l'esecuzione di 3 metodi in successione (onPause, onStop, onDestroy). Non sarebbe stato più semplice avere un solo metodo come indicato nella figura a destra?

Motivare la risposta.



Il ciclo di vita delle Activity di Android prevede l'esecuzione di tre metodi (onCreate(), onStart(), onResume()) per far partire l'esecuzione di un'app e tre metodi (onPause(), onStop(), onDestroy()) per distruggere un'app perché ciascuno di questi metodi ha uno scopo specifico.

onCreate() viene chiamato quando l'Activity viene creata per la prima volta. In questo metodo vengono effettuate le operazioni di inizializzazione, come il caricamento del layout, l'inizializzazione dei membri della classe e l'eventuale recupero dello stato salvato precedentemente.

onStart() viene chiamato quando l'Activity diventa visibile all'utente. In questo metodo vengono effettuate le operazioni di avvio, come il caricamento dei dati che verranno visualizzati nell'Activity.

onResume() viene chiamato quando l'Activity diventa l'Activity attiva e riceve l'input dell'utente. In questo metodo vengono effettuate le operazioni di preparazione all'interazione dell'utente, come l'avvio di animazioni o il ripristino dell'aggiornamento dell'interfaccia utente.

Se tutte queste operazioni fossero incluse in un unico metodo, sarebbe più difficile comprendere quale operazione viene eseguita in quale momento e potrebbe essere più difficile gestire eventuali problemi o errori. Inoltre, avere tre metodi separati consente di eseguire operazioni specifiche in momenti specifici del ciclo di vita dell'Activity, come il salvataggio dello stato nell'onPause() o l'arresto di animazioni o richieste di rete nell'onStop().

Inoltre, avere tre metodi separati per l'avvio e la distruzione di un'Activity consente di avere un maggiore controllo su come viene gestito il passaggio tra le Activity all'interno dell'app. Ad esempio, puoi scegliere di salvare lo stato dell'Activity in onPause() solo se l'Activity sta per essere sostituita da un'altra Activity all'interno dell'app, ma non se l'Activity viene chiusa. Ciò può essere utile per ottimizzare le prestazioni dell'app evitando di eseguire operazioni non necessarie quando l'Activity viene chiusa.

Inoltre, avere tre metodi separati per il ciclo di vita delle Activity consente di avere un maggiore controllo su come vengono gestite le risorse dell'app, come la memoria e le risorse di rete. Ad esempio, puoi scegliere di arrestare le animazioni o le richieste di rete nell'onStop() per liberare risorse quando l'Activity non è più visibile all'utente. Ciò può aiutare a garantire che l'app funzioni in modo efficiente e sia sicura dal punto di vista della gestione della memoria.

Si scrivano degli snippet di codice per lanciare da un activity “Principale” un’altra activity, “Secondaria”, passando un valore di tipo intero dall’activity principale a quella secondaria e facendo in modo che l’activity secondaria restituisca un valore di tipo stringa all’activity principale.

Classe principale

```
Intent intent = new Intent(Principale.this, Secondaria.class);
intent.putExtra("key_name", 123);
startActivity(intent);
```

Ecco un esempio di come ricevere il valore di tipo intero passato dall'Activity principale e restituire un valore di tipo stringa all'Activity principale:

```
// ricevi il valore di tipo intero passato dall'Activity principale
Intent intent = getIntent();
int value = intent.getIntExtra("key_name", 0);

// restituisci un valore di tipo stringa all'Activity principale
Intent resultIntent = new Intent();
resultIntent.putExtra("result_key", "valore restituito");
setResult(Activity.RESULT_OK, resultIntent);
finish();
```

Per ottenere il valore di tipo stringa restituito dall'Activity secondaria, puoi usare il metodo onActivityResult() dell'Activity principale:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == Activity.RESULT_OK) {
        String result = data.getStringExtra("result_key");
        // usa il valore restituito dall'Activity secondaria
    }
}
```

Si descriva il meccanismo dei permessi spiegando la differenza fra permessi normali e permessi pericolosi. Si metta in evidenza la gestione dei permessi in gruppi spiegando come vengono gestiti tali gruppi.

I permessi in Android consentono alle app di accedere a risorse protette, come la posizione dell'utente, la fotocamera o i dati delle chiamate. Le app devono richiedere l'autorizzazione dell'utente per accedere a queste risorse protette.

Ci sono due tipi di permessi in Android: permessi normali e permessi pericolosi. I permessi normali sono permessi che hanno un impatto minimo sulla privacy dell'utente o sulla sicurezza del dispositivo. Ad esempio, il permesso per accedere al networking o per accedere al vibratore del dispositivo sono considerati permessi normali.

I permessi pericolosi, al contrario, sono permessi che possono avere un impatto significativo sulla privacy dell'utente o sulla sicurezza del dispositivo. Ad esempio, il permesso per accedere alla posizione dell'utente o per accedere ai dati delle chiamate sono considerati permessi pericolosi.

Quando un'app viene installata su un dispositivo, i permessi normali vengono concessi automaticamente all'app. I permessi pericolosi, invece, devono essere esplicitamente concessi dall'utente durante l'utilizzo dell'app. L'utente può revocare questi permessi in qualsiasi momento nelle impostazioni del dispositivo.

I permessi in Android possono essere organizzati in gruppi. Ad esempio, i permessi per accedere ai dati delle chiamate e ai dati dei contatti appartengono al gruppo dei permessi per l'accesso ai dati dei contatti. Quando un'app richiede un permesso appartenente a un gruppo, l'utente viene avvisato che l'app richiede l'accesso a tutti i permessi del gruppo, non solo al singolo permesso. L'utente può quindi concedere o negare l'accesso a tutti i permessi del gruppo in modo facile e veloce.

Si completi il seguente codice assumendo di avere a disposizione la funzione “partialLoad()” che si occupa di caricare in ogni chiamata un 10% dell’immagine img (quindi dopo dieci chiamate a tale funzione img sarà completa). Si renda visibile la ProgressBar all’inizio del caricamento e invisibile alla fine. Si aggiorni la progress bar ad ogni 10% di caricamento e si mostri un Toast di avviso “Caricamento quasi completato” quando si raggiunge l’80% del caricamento. Si mostri l’immagine nell’imageView alla fine del caricamento.

```
public class ThreadAsyncTaskActivity extends Activity {
    private ImageView imageView;
    private ProgressBar progressBar;
    private Bitmap img;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);
        imageView = (ImageView)findViewById(R.id.imageView);
        progressBar = (ProgressBar)findViewById(R.id.progressBar);

        progressBar.setVisibility(View.VISIBLE);

        new LoadIconTask().execute();
    }

    class LoadIconTask extends AsyncTask<Integer, Integer, Bitmap> {
        private Integer index = 1;
        @Override
        protected void onPreExecute() {
        }
        @Override
        protected Bitmap doInBackground(Integer... ids) {
            while (index <= 10) {
                partialLoad();
                index++;
                publishProgress((index-1) * 10);
            }
            return img;
        }
        @Override
        protected void onProgressUpdate(Integer... values) {
            progressBar.setProgress(values[0]);
            if (values[0] >= 80) {
                Toast.makeText(ThreadAsyncTaskActivity.this, "Caricamento quasi completato",
Toast.LENGTH_SHORT).show();
            }
        }
        @Override
        protected void onPostExecute(Bitmap result) {
            progressBar.setVisibility(View.INVISIBLE);
            imageView.setImageBitmap(result);
        }
    }
}
```

Se due frammenti di un activity devono comunicare è buona prassi di programmazione implementare tale comunicazione non in modo diretto da frammento a frammento ma passando attraverso l'activity che ospita i frammenti (quindi la frammento che vuole inviare la comunicazione lo fa interagendo con l'activity ospitante e poi questa interagisce con il frammento che deve ricevere la comunicazione).

Perché è una buona prassi di programmazione?

Si descriva un modo per implementare la comunicazione fra due frammenti attraverso l'activity ospitante.

È una buona prassi di programmazione implementare la comunicazione fra due frammenti attraverso l'activity ospitante perché questo rende più facile il mantenimento del codice e rende più facile modificare la struttura dell'app in futuro. Se la comunicazione viene implementata direttamente fra i frammenti, ciò potrebbe rendere più difficile modificare la struttura dell'app in futuro, poiché potrebbe essere necessario modificare anche il modo in cui i frammenti si comunicano tra loro.

Per implementare la comunicazione fra due frammenti attraverso l'activity ospitante, puoi creare un'interfaccia all'interno del primo frammento che definisce un metodo per inviare la comunicazione. Quindi, nel metodo onAttach() del primo frammento, puoi impostare un riferimento al primo frammento come un'implementazione dell'interfaccia. Nell'activity ospitante, puoi quindi creare un metodo che prende come argomento la comunicazione da inviare e richiama il metodo dell'interfaccia per inviare la comunicazione al primo frammento. Nel secondo frammento, puoi quindi implementare il metodo dell'interfaccia per ricevere la comunicazione dal primo frammento.

Ecco un esempio di come implementare la comunicazione fra due frammenti utilizzando il primo metodo descritto:

Crea un'interfaccia con un metodo di callback:
Copy code

```
public interface OnFragmentInteractionListener {
    void onFragmentInteraction(String message);
}
```

Fai implementare l'interfaccia all'activity ospitante:
Copy code

```
public class MainActivity extends AppCompatActivity implements FirstFragment.OnFragmentInteractionListener {
    ...
    @Override
    public void onFragmentInteraction(String message) {
        // Gestisci la comunicazione tra i due frammenti qui
    }
    ...
}
```

Nel primo frammento, ottieni un riferimento all'activity ospitante e richiama il metodo di callback quando hai bisogno di comunicare con il secondo frammento:
Copy code

```
public class FirstFragment extends Fragment {
    ...
    private OnFragmentInteractionListener mListener;
    ...
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        if (context instanceof OnFragmentInteractionListener) {
            mListener = (OnFragmentInteractionListener) context;
        } else {
            throw new RuntimeException(context.toString()
                    + " must implement OnFragmentInteractionListener");
        }
    }
    ...
    public void sendMessage(String message) {
        mListener.onFragmentInteraction(message);
    }
    ...
}
```

In questo esempio, l'activity ospitante implementa l'interfaccia OnFragmentInteractionListener e quindi può gestire la comunicazione tra i due frammenti quando viene richiamato il metodo onFragmentInteraction(). Il primo frammento ottiene un riferimento all'activity ospitante e richiama il metodo di callback quando ha bisogno di comunicare con il secondo frammento.

Si fornisca un file XML per un'animazione che prima ruota la view di 180° e poi la sposta orizzontalmente di 200dp.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
  <rotate
     android:fromDegrees="0"
     android:toDegrees="180"
     android:pivotX="50%"
     android:pivotY="50%"
     android:duration="1000"
     android:fillAfter="true"/>
  <translate
     android:fromXDelta="0"
     android:toXDelta="200dp"
     android:duration="1000"
     android:fillAfter="true"/>
</set>
```

Quando si registra il listener di un sensore è possibile selezionare la velocità di campionamento da utilizzare:

– *SENSOR_DELAY_NORMAL (0,2sec)*
– *SENSOR_DELAY_GAME (0,02sec)*
– *SENSOR_DELAY_UI (0,06sec)*
– *SENSOR_FASTEST (0sec)*

Si discuta dei vantaggi e svantaggi di queste varie possibilità e di quale accortezze deve avere il programmatore per un app che utilizza i sensori.

La velocità di campionamento dei sensori su un dispositivo Android può essere impostata utilizzando i costanti SENSOR_DELAY_NORMAL, SENSOR_DELAY_GAME, SENSOR_DELAY_UI e SENSOR_FASTEST. Ecco una spiegazione di ciascuna di queste opzioni:

SENSOR_DELAY_NORMAL: questa opzione imposta una velocità di campionamento di circa 0,2 secondi. È l'opzione di default e dovrebbe essere utilizzata per la maggior parte delle applicazioni che utilizzano i sensori.

SENSOR_DELAY_GAME: questa opzione imposta una velocità di campionamento di circa 0,02 secondi. È più veloce di SENSOR_DELAY_NORMAL e dovrebbe essere utilizzata per le applicazioni che richiedono una maggiore precisione dei sensori, ad esempio giochi o applicazioni che utilizzano la realtà aumentata.

SENSOR_DELAY_UI: questa opzione imposta una velocità di campionamento di circa 0,06 secondi. È più lenta di SENSOR_DELAY_NORMAL e dovrebbe essere utilizzata per le applicazioni che richiedono solo una precisione dei sensori limitata, ad esempio le applicazioni che mostrano solo informazioni di base sullo schermo.

SENSOR_FASTEST: questa opzione imposta la velocità di campionamento più alta possibile, ovvero 0 secondi. Dovrebbe essere utilizzata solo se si ha bisogno dei dati dei sensori il più rapidamente possibile, ad esempio per applicazioni di monitoraggio della salute o per il controllo delle apparecchiature industriali.

Quando si sceglie la velocità di campionamento da utilizzare per un'applicazione, è importante considerare l'impatto sulla durata della batteria del dispositivo. Le opzioni più rapide consumano più energia, quindi dovrebbero essere utilizzate solo se strettamente necessarie. Inoltre, è importante considerare l'impatto sulla precisione dei dati dei sensori. Le opzioni più lente forniscono dati più precisi, ma potrebbero non essere adatte per le applicazioni che richiedono una maggiore precisione.

Che cosa è un Toast customizzato? Si spieghi come implementare un Toast customizzato.

Un Toast è una piccola notifica pop-up che viene visualizzata sullo schermo per un breve periodo di tempo. In Android, è possibile creare Toast personalizzati modificando il layout del Toast utilizzando un file XML. Ecco come implementare un Toast personalizzato:

Crea un layout personalizzato per il Toast utilizzando un file XML. Ad esempio:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/custom_toast_container"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:gravity="center_vertical"
    android:background="#FFFFFF"
    android:padding="8dp">

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher_background" />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#000000"
        android:text="Custom Toast" />

</LinearLayout>
```

Crea una nuova istanza di Toast e imposta il layout personalizzato utilizzando il metodo setView(). Ad esempio:

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.custom_toast, (ViewGroup)
findViewById(R.id.custom_toast_container));

Toast toast = new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

In questo esempio, il layout personalizzato viene caricato utilizzando il metodo inflate() della classe LayoutInflater e viene impostato come visualizzazione del Toast utilizzando il metodo setView(). Si noti che è inoltre possibile impostare la durata del Toast e la posizione sullo schermo utilizzando i metodi setDuration() e setGravity(), rispettivamente.

Prova scritta

Pagina aggiuntiva

Prova scritta

Pagina aggiuntiva

Pagina aggiuntiva