Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.2

http://www.dcc.ufrj.br/~fabiom/java

Estado da Aplicação

- Em um computador, aplicações ficam abertas até o usuário fechá-las explicitamente
- No sistema Android, se uma aplicação não está mais em primeiro plano, o sistema pode decidir fechá-la para economizar memória
- Se o usuário volta para a aplicação, é como se ela estivesse sendo iniciada de novo
- Na nossa calculadora, isso implica que perdemos todo o estado atual dela

Salvando o Estado

- Quando o sistema fecha uma aplicação, ele dá uma chance para ela guardar toda a informação necessária para recriar o estado onde estava
- Isso é feito pelo método `onSaveInstanceState` de `Activity`
- Esse método recebe um objeto `Bundle`, onde a informação necessária deve ser armazenada
- Esse objeto é passado depois para o método `onCreate`, quando a aplicação é reiniciada
- Podemos guardar nele informação para recriar o modelo da calculadora

Serialização

- Podemos recriar o estado da calculadora só a partir de dados primitivos, com alguma “mágica”
  - `Class.forName(“Nome”).getConstructor(...).newInstance(...)`
- Uma alternativa é *serializar* todo o modelo, empacotando todos os seus objetos em uma representação que permite depois reconstruí-los
- Para marcar um objeto como serializável, precisamos implementar a interface `Serializable`
- Essa interface não tem métodos; o trabalho todo é feito pelo sistema Java

Externalizable

- Para os objetos de uma classe serem serializáveis, todos os seus campos devem ser também
- Não podemos tornar todo o modelo serializável, pois não podemos serializar o `ObservadorDisplay`
- `Externalizable` é uma subinterface se `Serializable` onde podemos controlar o que é serializado
- `Externalizable` tem dois métodos que permitem serializar e desserializar as partes do objeto que queremos
- No caso do modelo, é tudo exceto o observador

Busca no Twitter

- Vamos fazer outra aplicação simples, para exercitar outras partes do Android
- Nossa aplicação vai fazer uma busca no Twitter, usando a biblioteca Twitter4J
- Ao invés de usar as próprias classes do Twitter4J diretamente como modelo, vamos escondê-las atrás de uma classe `Tweet`
- Temos apenas uma atividade: o usuário entra uma string para buscar, e isso dá uma lista de tweets

ListView

- Para mostrar a lista dos tweets, vamos usar um componente `ListView`
- Uma `ListView` é como um `LinearLayout` vertical que pode facilmente alterado em tempo de execução
- Se o número de componentes na lista é grande demais para exibir todos, a `ListView` pode ser deslizada
- Os componentes podem ser qualquer coisa: conectamos uma `ListView` ao que ela deve exibir com um adaptador derivado de `BaseAdapter`

WebView

- Para mostrar cada tweet, vamos usar uma `WebView`
- Uma `WebView` é como um mini-browser, e pode mostrar HTML qualquer; é assim que vamos mostrar as fotos de perfil dos donos dos tweets
- Conectamos nossa `ListView` às `WebViews` no método `getView` do adaptador
- Podemos sempre criar uma nova `WebView`, mas é mais eficiente reutilizar a já existente, trocando apenas o conteúdo

Tarefas

- Há um problema na nossa busca: uma busca pode demorar para completer
- Enquanto a busca executa, nossa aplicação está travada
- Se demorar tempo demais, o sistema Android vai perguntar pro usuário se ele quer encerra a aplicação
- Tratadores de eventos nunca devem fazer nada que não seja efetivamente instantâneo; qualquer ação mais demorada deve ser delegada para uma *tarefa* que vai executar em segundo plano
- Tarefas são subclasses de `AsyncTask`

AsyncTask<E,I,S>

- Uma tarefa é parametrizada por três tipos:
  - `E` é o tipo dos parâmetros de entrada para a tarefa
  - `I` é o tipo dos resultados intermediários, caso ela reporte progresso
  - `S` é o tipo do resultado da tarefa
- O método `S doInBackGround(E... params)` executa a tarefa
- O método `void onPostExecute(S res)` atualiza a interface com o resultado

Inflando layouts

- Se quisermos um layout mais complexo nas linhas de uma `ListView`, podemos criar objetos layout e adicionar objetos view a ele

- Ou podemos declarar um layout em XML e *inflar* esse layout

- Para isso usamos o *serviço* `LAYOUT_INFLATER_SERVICE`, uma instância de LayoutInflater:

```
LayoutInflater inf =
    (LayoutInflater)lv.getContext().getSystemService(
        Context.LAYOUT_INFLATER_SERVICE);
view = inf.inflate(R.layout.item_lv, null);
```

`Toast` e `ProgressDialog`

- Uma busca pode dar errado: pode não ter resultados, ou pode ter algum problema na rede
- O `Toast` é uma maneira de a aplicação informar ao usuário que alguma coisa aconteceu
- Ele faz aparecer uma mensagem que fica na tela por alguns segundos e depois some
- Para informar pro usuário que uma busca está acontecendo, podemos usar um `ProgressDialog` dentro de nossa `AsyncTask`

Editor de Figuras

- Vamos usar nosso modelo de editor de figuras para fazer uma versão Android dele

- Ao invés de usarmos botões, vamos usar toda a área da aplicação como área de desenho, e usar ações e itens de menu para controlar o modo atual do editor

- Vamos implementar uma view para ser o Canvas do editor, e conectá-lo com componentes Android implementando as outras interfaces do modelo

- Nosso canvas também vai capturar os eventos de clique e arrasto, e passá-los para um controlador que vai traduzi-lo nos métodos do modelo

Canvas

- Para poder desenhar em uma `View`, redefinimos seu método `onDraw`
- Esse método recebe uma instância de `Canvas`
- Para desenhar também precisamos de uma instância de `Paint`, que dá a cor e o estilo de desenho (no nosso caso, queremos que as figuras sejam preenchidas)

```
pen = new Paint();
pen.setARGB(255,0,0,0);
pen.setStyle(Style.FILL);
```

```
@Override
public void onDraw(Canvas c) {
  super.onDraw(c);
  c.drawRGB(255,255,255);
  c.drawRect(10, 10, 100, 100, pen);
}
```

onTouchEvent

- Para capturar toques na nossa área de desenho, redefinimos o método `onTouchEvent`, que recebe uma instância de `MotionEvent`

- Estamos interessados em três partes do evento: a ação, a coordenada x e a coordenada y

- A ação diz se um toque começou (o usuário encostou o dedo na tela), se um toque terminou (o usuário removeu o dedo), ou se ele se moveu (o usuário deslizou o dedo sobre a tela)

- Vamos traduzir isso em eventos aperto, solta e arrasto no nosso controlador

Menu de Opções

- É comum que as atividades de uma aplicação Android tenham um *menu de opções*

- Na nossa aplicação de busca, por exemplo, vamos por alguns termos de busca comuns

- Começamos editando o XML do menu, acrescentando blocos item:

```
<item
    android:id="@+id/busca_java"
    android:orderInCategory="100"
    android:showAsAction="never"
    android:title="Java"/>
```

Ações e exibição

- Se quisermos que uma opção apareça como uma ação na barra de título então mudamos o atributo `showAsAction` para *"ifRoom"* ao invés de *"never"*

- Para exibir o menu, redefinimos o método onCreateOptionsMenu da atividade, e então *inflamos* o menu:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.buscas, menu);
    return true;
}
```

Respondendo a cliques

- Quando o usuário toca em um item do menu, ou em uma ação, o sistema chama o método `onOptionsItemSelected`, basta redefini-lo:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.busca_java:
            busca("java");
            return true;
        case R.id.busca_ufrj:
            busca("ufrj");
            return true;
        default:
            return super.onOptionsItemSelected(item);
        }
    }
}
```

Menus dinâmicos

- E se quisermos mostrar no menu as cinco últimas frases buscadas?

- É fácil guardar uma lista de termos de busca, mas como fazemos o menu mudar junto? Implementando o método `onPrepareOptionsMenu` ao invés de `onCreateOptionsMenu`:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    menu.clear();
    menu.add(0, R.id.busca_java, 1, "Java");
    menu.add(0, R.id.busca_ufrj, 2, "UFRJ");
    return true;
}
```

- Podemos usar quaisquer números no segundo parâmetro

Checked menus

- Para seleção do modo de edição, vamos usar *checked menus* no menu da barra de aplicação

- Esses menus têm uma checkbox ao lado; a checkbox do modo atual aparecerá ticada

```
<item android:id="@+id/modo_mover"
      android:showAsAction="never"
      android:checkable="true"
      android:checked="true"
      android:title="Mover"/>
```

Mudando entre Atividades

- Até agora nossas aplicações Android têm apenas uma atividade, o que simplifica a sua estrutura
- Mas nem sempre é possível fazer uma aplicação assim
- Vamos acrescentar cores ao Editor de Figuras, e fazer a escolha da cor ser feita por uma segunda atividade
- As mudanças no modelo são simples, e precisamos também mudar os métodos de desenho em `Tela`

Intent

- Cada atividade é como uma miniaplicação, então a comunicação entre elas é indireta, através de instâncias de Intent

- Usando um `Intent` podemos mandar uma mensagem para outra atividade na mesma aplicação, ou para alguma outra aplicação instalada

- Na forma mais simples, criamos um `Intent` dando a atividade atual, e *classe* atividade que queremos disparar:

```
Intent icor = new Intent(this, EscolheCor.class);
```

Extras

- Podemos acrescentar dados a um `Intent` que a outra atividade vai poder usar, usando o método `putExtra`

- Esses dados podem ser qualquer objeto serializável

- Isso quer dizer que poderíamos passar o modelo inteiro do editor na nossa mensagem, mas isso seria um desperdício, então passamos apenas a cor corrente

```
icor.putExtra("cor_r", modelo.r);
icor.putExtra("cor_g", modelo.g);
icor.putExtra("cor_b", modelo.b);
```

`startActivityForResult`

- Uma vez que temos todos os dados no `Intent`, disparamos a outra atividade com o método `startActivityForResult` da atividade atual

- Passamos o `Intent`, e um código numérico que serve para identificar a razão de estarmos chamando outra atividade

- Quando a outra atividade quer voltar para quem chamou, ela usa o método `setResult` para dar um código de retorno (outro número), e um `Intent` com dados que ela queira passar de volta, depois chama o método `finish`

- Esse `Intent` pode ser criado com um construtor vazio

onActivityResult

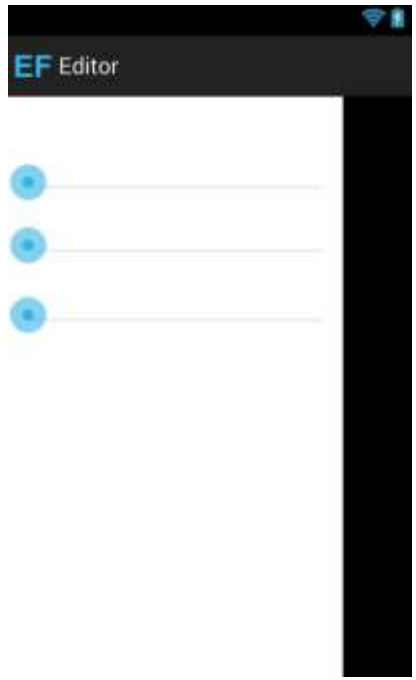
- Quando a atividade que disparamos termina, o método `onActivityResult` na atividade original é chamado

- Esse método recebe o código da razão, o código de resultado, e o `Intent` retornado pela outra atividade

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch(requestCode) {
    case ESCOLHE_COR:
        if(resultCode == RESULT_OK) {
            modelo.r = data.getIntExtra("cor_r", 0);
            modelo.g = data.getIntExtra("cor_g", 0);
            modelo.b = data.getIntExtra("cor_b", 0);
        }
        break;
    }
}
```

Múltiplos layouts

- O layout da nossa atividade de escolha de cor não é o mais adequado quando dispositivo está na orientação “retrato”:



- Podemos criar outro arquivo `escolhe_cor.xml`, em um diretório `layout-port`, com o layout correto

Intents implícitos

- Uma terceira forma de criar um `Intent` é passando apenas uma *ação* que queremos fazer, e deixar o sistema escolher uma aplicação e uma atividade que pode fazer aquela ação

- Essa é a maneira que as aplicações se comunicam umas com as outras no sistema Android sem precisar acoplá-las umas às outras

- Vamos usar esse recurso para implementar outro recurso no editor: enviar o desenho atual como uma imagem

Armazenamento local

- Não podemos enviar uma imagem como extra no Intent, pois uma imagem pode ser grande demais para ter várias cópias dela na memória
- Precisamos primeiro gravá-la em alguma parte permanente do dispositivo
- Vamos gravá-la em um arquivo no *armazenamento local*
- O armazenamento local é uma área privativa da aplicação, e que é apagada quando a aplicação é desinstalada
- Acessamos o armazenamento local pelo métodos `getFilesDir`, ou pelos métodos `openFileInput` e `openFileOutput`

Salvando a Tela

- Usamos um `Bitmap` e outro `Canvas` para salvar o conteúdo da tela, e o método `compress` de `Bitmap` para gravá-lo como um arquivo PNG:

```
public void salvaTela(FileOutputStream arq) {
    Bitmap bm = Bitmap.createBitmap(this.getWidth(),
                                    this.getHeight(),
                                    Config.ARGB_8888);
    Canvas c = new Canvas(bm);
    this.draw(c);
    bm.compress(CompressFormat.PNG, 100, arq);
}
```

- Na atividade, criamos o arquivo com `openFileOutput` e passamos o resultado para `salvaTela`

Enviando

- Agora basta criar o Intent de enviar, e pedir pro sistema começar uma atividade que o responda:

```
Intent ienviar = new Intent(Intent.ACTION_SEND);
ienviar.setType("image/png");
Uri uri = Uri.fromFile(new File(getFilesDir(), "tela.png"));
ienviar.putExtra(Intent.EXTRA_STREAM, uri.toString());
startActivity(Intent.createChooser(ienviar, "Enviar com"));
```

- Só que isso não funciona! O armazenamento local é privado da aplicação, então outras aplicações não conseguem ler o arquivo

- Precisamos criar um *provedor de conteúdo*

Provedor de Conteúdo

- Com um provedor de conteúdo uma aplicação pode compartilhar dados com outra aplicação

- Cada fonte de conteúdo é identificada por uma URI:

```
content://compii.editor/tela.png
```

- A parte de *host* da URI identifica a aplicação, a parte de *caminho* identifica o conteúdo dentro da aplicação

- Conteúdo pode ser dados não estruturados, como um arquivo, ou uma *tabela* que responde a consultas, como uma tabela de um banco de dados

ContentProvider

- Um provedor de conteúdo é uma subclasse de `ContentProvider`

- Ele implementa uma série de métodos, para consultas a tabelas e a arquivos:

```
// Inicialização
public boolean onCreate();
// Tipo MIME do dado na uri
public String getType(Uri uri);
// Consulta linhas da tabela (como SELECT do SQL)
public Cursor query(Uri uri, String[] prj, String sel, String[] args, String sort);
// Insere linhas na tabela (como INSERT do SQL)
public Uri insert(Uri uri, ContentValues values);
// Apaga linhas da tabela (como DELETE do SQL)
public int delete(Uri uri, String sel, String[] args);
// Modifica linhas da tabela (como UPDATE do SQL)
public int update(Uri uri, ContentValues values, String sel, String[] selArgs);
// Abre e retorna um arquivo na uri
public ParcelFileDescriptor openFile(Uri uri, String mode)
   throws FileNotFoundException;
```

Provedor de conteúdo do Editor

- O provedor de conteúdo do editor só retorna um arquivo, então ele só implementa efetivamente openFile:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
          throws FileNotFoundException {
    File arq = new File(getContext().getFilesDir(), uri.getPath());
    return ParcelFileDescriptor.open(arq,
        ParcelFileDescriptor.MODE_READ_ONLY);
}
```

- O provedor também precisa de uma entrada no manifesto da aplicação:

```
<provider
            android:name="compii.editor.ProvedorArq"
            android:authorities="compii.editor"
            android:exported="true" />
```

Enviando

- Agora basta criar o Intent de enviar, e pedir pro sistema começar uma atividade que o responda:

```
Intent ienviar = new Intent(Intent.ACTION_SEND);
ienviar.setType("image/png");
Uri uri = Uri.parse(“content://compii.editor/tela.png");
ienviar.putExtra(Intent.EXTRA_STREAM, uri.toString());
startActivity(Intent.createChooser(ienviar, "Enviar com"));
```

- O emulador não tem muitas opções de aplicação que podem receber uma imagem, então vamos usar a aplicação RecebeImagem, no projeto RecebeImagem.zip