PRAP3 – Prof. Adriano - Informática 3º. Ano de A até G

Fragmentos **Part of Android Jetpack.**

Um `Fragment` representa o comportamento ou uma parte da interface do usuário em um `FragmentActivity`. É possível combinar vários fragmentos em uma única atividade para criar uma IU de vários painéis e reutilizar um fragmento em diversas atividades. Você pode imaginar um fragmento como uma seção modular de uma atividade, que tem o próprio ciclo de vida, recebe os próprios eventos de entrada e que pode ser adicionada ou removida durante a execução da atividade (uma espécie de “subatividade” que pode ser reutilizada em diferentes atividades).

Um fragmento deve sempre ser hospedado em uma atividade e o ciclo de vida dele é diretamente impactado pelo ciclo de vida da atividade do host. Por exemplo, quando a atividade é pausada, todos os fragmentos também são e, quando a atividade é destruída, todos os fragmentos também são. No entanto, enquanto uma atividade estiver em execução (estiver no *estado do ciclo de vida* retomado), é possível processar cada fragmento independentemente, como adicioná-los ou removê-los. Ao realizar tal transação com fragmentos, também é possível adicioná-los a uma pilha de retorno que é gerenciada pela atividade — cada entrada da pilha de retorno na atividade é um registro da transação de fragmento que ocorreu. A pilha de retorno permite que o usuário reverta uma transação de fragmento (navegue para trás), pressionando o botão *Voltar*.

Ao adicionar um fragmento como parte do layout da atividade, ele se encontrará em um `ViewGroup` dentro da hierarquia de visualizações, e o fragmento definirá o próprio layout da exibição. É possível inserir um fragmento no layout, declarando-o no arquivo de layout da atividade como um elemento `<fragment>`, ou no código do aplicativo, adicionando-o a um `ViewGroup` atual.

Este documento descreve como compilar o aplicativo para usar fragmentos, incluindo como os fragmentos podem manter o estado ao serem adicionados à pilha de retorno da atividade, como compartilhar eventos com a atividade e com outros fragmentos da atividade, como contribuir para a barra de ação da atividade e muito mais.

Para informações sobre como gerenciar ciclos de vida, inclusive orientação sobre práticas recomendadas, veja os recursos a seguir:

- Como gerenciar componentes cientes do ciclo de vida
- Guia para a arquitetura de aplicativos
- Compatibilidade com tablets e celulares

Filosofia de projeto

O Android introduziu os fragmentos no Android 3.0 (API de nível 11) principalmente para suportar mais projetos de IU flexíveis e dinâmicos em telas grandes, como em tablets. Como a tela de um tablet é muito maior que a de um celular, há mais espaço para combinar e alternar componentes da IU. Os fragmentos permitem tais projetos sem haver a necessidade de gerenciar alterações complexas na hierarquia de visualizações. Ao dividir o layout de uma atividade em fragmentos, é possível modificar a aparência da atividade em ambiente de execução e preservar essas alterações na pilha de retorno que é gerenciada por essa atividade. Agora estão amplamente disponíveis por meio da biblioteca de suporte a fragmentos.

Por exemplo, um aplicativo de notícias pode usar um fragmento para exibir uma lista de artigos na esquerda e outro fragmento para exibir um artigo na direita — ambos os fragmentos aparecem em uma atividade, lado a lado. Cada fragmento tem o próprio conjunto de métodos de callback do ciclo de vida e lida com os próprios eventos de entrada do usuário. Portanto, em vez de usar uma atividade para selecionar um artigo e outra atividade para lê-lo, o usuário pode selecionar um artigo e lê-lo por completo dentro da mesma atividade, como ilustrado no layout do tablet na figura 1.

Você deve projetar cada fragmento como um componente modular e reutilizável da atividade. Ou seja, como cada fragmento define o próprio layout e comportamento com os próprios callbacks do ciclo de vida, é possível incluir um fragmento em várias atividades para poder projetá-lo para reutilização e evitar o tratamento direto de um fragmento em outro fragmento. Isso é especialmente importante porque um fragmento modular permite alterar as combinações de fragmentos para tamanhos de tela diferentes. Ao projetar o aplicativo para ser compatível com tablets e celulares, você poderá reutilizar os fragmentos em diferentes configurações de layout para otimizar a experiência do usuário com base no espaço de tela disponível. Por exemplo, em um celular, talvez seja necessário separar os fragmentos para fornecer uma IU de painel único quando mais de um não se encaixar dentro da mesma atividade.

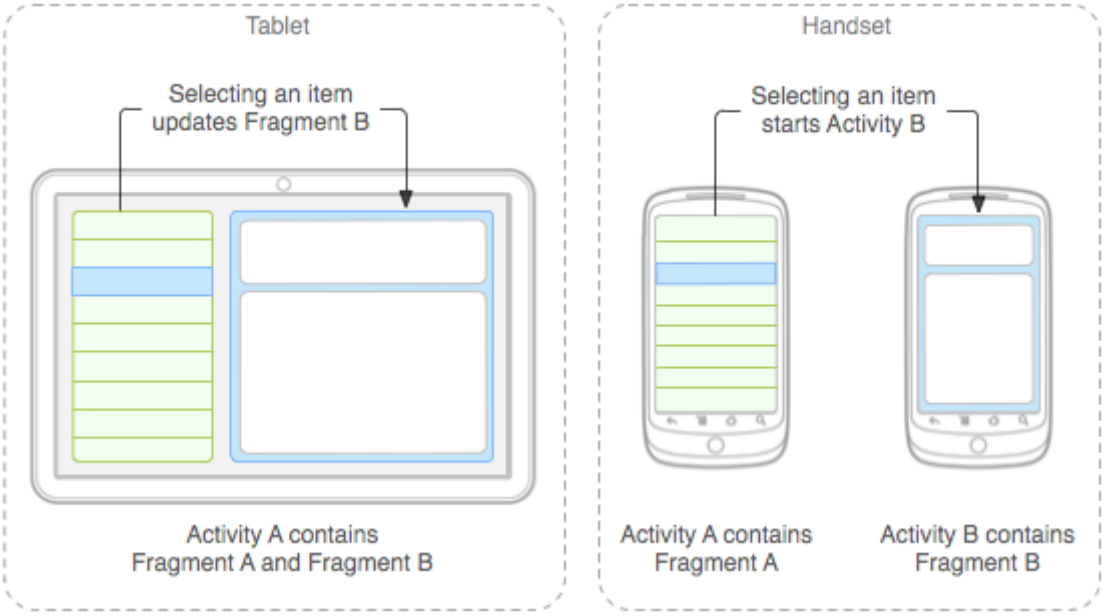

Figura 1. Exemplo de como dois módulos de IU definidos pelos fragmentos podem ser combinados em uma atividade de um projeto para tablet, mas separados em um projeto para celular.

Por exemplo — continuando com o exemplo do aplicativo de notícias —, o aplicativo pode incorporar dois fragmentos na *atividade A* quando executado em um dispositivo do tamanho de um tablet. No entanto, em uma tela de tamanho de celular, não há espaço suficiente para ambos os fragmentos, então a *atividade A* inclui somente o fragmento da lista de artigos e, quando o usuário seleciona um artigo, ele inicia a *atividade B*, que inclui o segundo fragmento para ler o artigo. Portanto, o aplicativo é compatível com tablets e celulares por meio da reutilização dos fragmentos em combinações diferentes, como ilustrado na figura 1.

Para saber mais sobre o projeto de aplicativos com diferentes combinações de fragmentos para configurações de tela diferentes, consulte a Visão geral de compatibilidade de tela.

Criação de um fragmento

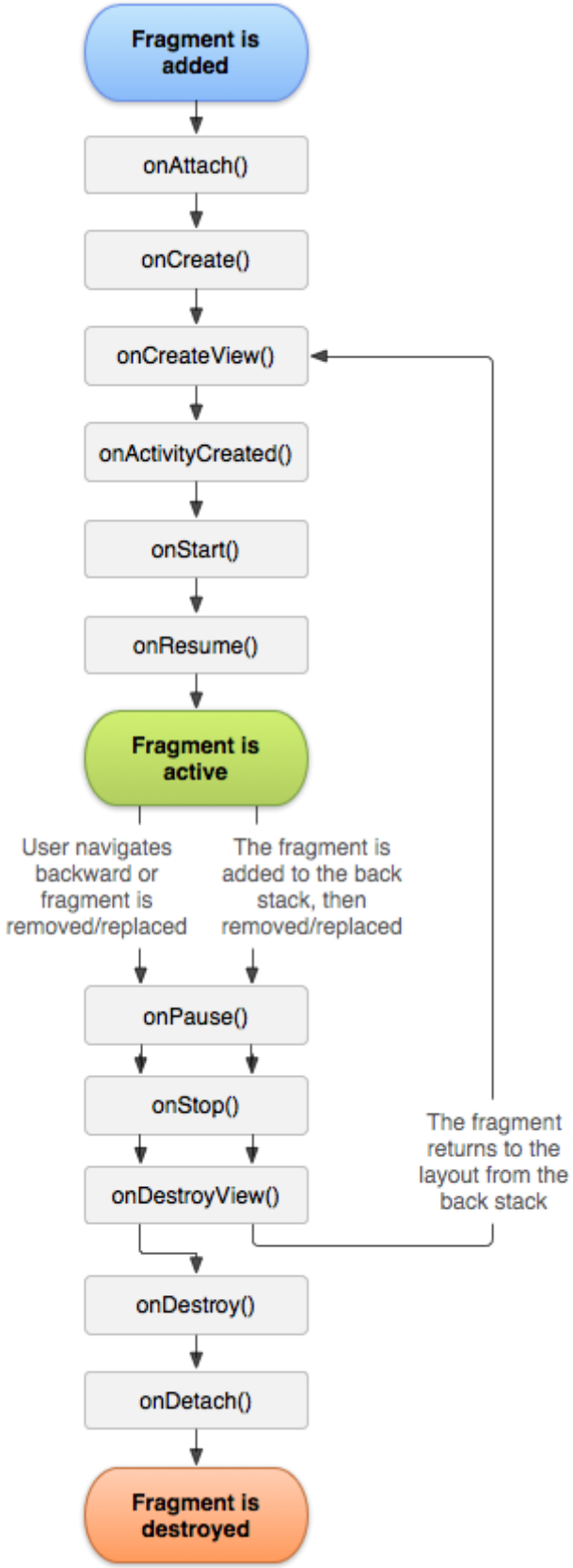

Figura 2. Ciclo de vida de um fragmento (enquanto a atividade está em execução).

Para criar um fragmento, é preciso criar uma subclasse de `Fragment` (ou usar uma subclasse existente dele). A classe `Fragment` tem um código que é muito parecido com o de uma `Activity`. Ele contém métodos de callback semelhantes aos de uma atividade, como `onCreate()`, `onStart()`, `onPause()` e `onStop()`. Na verdade, caso esteja convertendo um aplicativo do Android existente para usar os fragmentos, basta mover o código dos métodos de retorno de chamada da atividade para os respectivos métodos de callback do fragmento.

Geralmente, deve-se implementar pelo menos os seguintes métodos de ciclo de vida:

`onCreate()`

> O sistema o chama ao criar o fragmento. Dentro da implementação, deve-se inicializar os componentes essenciais do fragmento que se deseja reter quando o fragmento é pausado ou interrompido e, em seguida, retomado.

`onCreateView()`

> O sistema chama isso quando é o momento de o fragmento desenhar a interface do usuário pela primeira vez. Para desenhar uma IU para o fragmento, você deve retornar uma `View` deste método, que é a raiz do layout do fragmento. É possível retornar como nulo se o fragmento não fornecer uma IU.

`onPause()`

> O sistema chama esse método como o primeiro indício de que o usuário está saindo do fragmento (embora não seja sempre uma indicação de que o fragmento está sendo destruído). É quando geralmente se deve confirmar qualquer alteração que deva persistir além da sessão do usuário atual (porque o usuário pode não retornar).

A maioria dos aplicativos deve implementar pelo menos três destes métodos para cada fragmento, mas há vários outros métodos de callback que você deve usar para lidar com diversos estágios do ciclo de vida do fragmento. Todos os métodos de callback do ciclo de vida são abordados com mais detalhes na seção Processamento do ciclo de vida dos fragmentos.

O código que implementa as ações do ciclo de vida de um componente dependente precisa ser substituído no próprio componente, e não diretamente nas implementações de callback do fragmento. Veja Como gerenciar componentes cientes do ciclo de vida para saber como tornar seus componentes dependentes cientes do ciclo de vida.

Há também algumas subclasses que você pode querer estender em vez da classe `Fragment` de base:

`DialogFragment`

> Exibe uma caixa de diálogo flutuante. Usar essa classe para criar uma caixa de diálogo é uma boa alternativa para usar métodos auxiliares das caixas de diálogo na classe `Activity`, pois é possível incorporar uma caixa de diálogo de fragmento na pilha de retorno dos fragmentos gerenciados pela atividade, permitindo que o usuário retorne a um fragmento dispensado.

`ListFragment`

> Exibe uma lista de itens gerenciados por um adaptador (como um `SimpleCursorAdapter`), semelhante a `ListActivity`. Ele fornece vários métodos para gerenciar uma visualização de lista, como o callback `onListItemClick()` para lidar com eventos de clique. O método preferido de exibição de uma lista é usar RecyclerView em vez de ListView. Neste caso, você precisaria criar um fragmento que inclui um RecyclerView no layout. Acesse Criar uma lista com RecyclerView para saber como fazer isso.

`PreferenceFragmentCompat`

> Exibe uma hierarquia de objetos `Preference` como uma lista. Isso é usado para criar uma tela de configurações para seu aplicativo.

Adição de uma interface do usuário

Um fragmento é geralmente usado como parte de uma interface do usuário da atividade e contribui para a atividade com o próprio layout.

Para fornecer um layout para um fragmento, você deve implementar o método de callback `onCreateView()`, que o sistema Android chama no momento em que o fragmento precisa desenhar o layout. A implementação desse método deve retornar uma `View`, que é a raiz do layout do fragmento.

Observação: se o fragmento for uma subclasse de **`ListFragment`**, a implementação padrão retornará uma **`ListView`** de **`onCreateView()`** e não será necessário implementá-la.

Para retornar um layout de `onCreateView()`, é possível inflá-lo a partir de um recurso de layout definido no XML. Para ajudar a fazer isso, o `onCreateView()` fornece `um objeto LayoutInflater`.

Por exemplo, a seguir há uma subclasse de `Fragment` que carrega um layout do arquivo `example_fragment.xml`:

KOTLINJAVA

```
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.example_fragment,
container, false);
    }
}
```

Observação: no exemplo acima, **`R.layout.example_fragment`** é uma referência para um recurso de layout chamado **`example_fragment.xml`**, salvo nos recursos do aplicativo. Para obter informações sobre como criar um layout no XML, consulte a documentação Interface do usuário.

O parâmetro `container` passado para `onCreateView()` é o `ViewGroup` pai (do layout da atividade) em que o layout do fragmento será inserido. O parâmetro `savedInstanceState` é um `Bundle` que fornecerá dados sobre a instância anterior do fragmento se o fragmento estiver sendo retomado (a restauração de estado é abordada em mais detalhes na seção Processamento do ciclo de vida dos fragmentos).

O método `inflate()` usa três argumentos:

- O código de recurso do layout que você quer inflar.
- O `ViewGroup` que será pai do layout inflado. Passar o `container` é importante para que o sistema aplique os parâmetros de layout à exibição raiz do layout inflado, especificado pela exibição pai em que está ocorrendo.
- Um booleano que indica se o layout inflado deve ser anexado a `ViewGroup` (o segundo parâmetro) durante a inflação. Nesse caso, isso é falso, pois o sistema já está inserindo o layout inflado no `container` — retornar como verdadeiro criaria um grupo de visualizações redundante no layout final.

Esse é o modo de criar um fragmento que fornece um layout. A seguir, é preciso adicionar o fragmento à atividade.

Adição de um fragmento a uma atividade

Geralmente, um fragmento contribui com a atividade do host com uma parte da IU, que é incorporada como parte da hierarquia de visualizações geral da atividade. Há duas formas de adicionar um fragmento ao layout da atividade:

- **Declarar o fragmento dentro do arquivo de layout da atividade.**

  Neste caso, é possível especificar as propriedades do layout para o fragmento como se fosse uma visualização. Por exemplo, veja o arquivo de layout para uma atividade com dois fragmentos:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
            android:id="@+id/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

  O atributo `android:name` em `<fragment>` especifica a classe `Fragment` a instanciar no layout.

  Quando o sistema cria esse layout de atividade, ele instancia cada fragmento especificado no layout e chama o método `onCreateView()` para cada um para recuperar o layout de cada fragmento. O sistema insere a `View` retornada pelo fragmento diretamente no lugar do elemento `<fragment>`.

  Observação: cada fragmento requer um identificador único que o sistema possa usar para restaurá-lo se a atividade for reiniciada (e que possa ser usado para capturar o fragmento para realizar transações como a remoção). Há duas formas de fornecer um código a um fragmento.

  - Fornecer o atributo **`android:id`** com um código exclusivo.
  - Fornecer o atributo **`android:tag`** com uma string exclusiva.

- **Ou adicionar programaticamente o fragmento a um `ViewGroup` existente.**

A qualquer momento, enquanto a atividade está em execução, é possível adicionar fragmentos ao layout da atividade. Basta especificar um `ViewGroup` no qual posicionar o fragmento.

Para realizar transações de fragmentos na atividade (como adicionar, remover ou substituir um fragmento), você precisa usar APIs de `FragmentTransaction`. É possível adquirir uma instância de `FragmentTransaction` da `FragmentActivity` desta maneira:

KOTLINJAVA

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
```

É possível adicionar um fragmento usando o método `add()`, especificando o fragmento que será adicionado e a visualização em que será inserido. Por exemplo:

KOTLINJAVA

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

O primeiro argumento passado para `add()` é `ViewGroup` em que o fragmento deve ser colocado, especificado pelo código de recurso, e o segundo parâmetro é o fragmento a ser adicionado.

Ao realizar as alterações com `FragmentTransaction`, é necessário chamar `commit()` para que as alterações entrem em vigor.

Gerenciamento de fragmentos

Para gerenciar os fragmentos na atividade, você precisa usar `FragmentManager`. Para adquiri-lo, chame `getSupportFragmentManager()` na atividade.

Algumas das coisas que você pode fazer com `FragmentManager` incluem:

- Adquirir fragmentos existentes na atividade, com `findFragmentById()` (para fragmentos que forneçam uma IU no layout da atividade) ou `findFragmentByTag()` (para fragmentos que forneçam ou não uma IU).
- Retirar os fragmentos da pilha de retorno com `popBackStack()` (simulando um comando de *Voltar* do usuário).
- Registrar uma escuta para as alterações na pilha de retorno com `addOnBackStackChangedListener()`.

Para saber mais sobre esses e outros métodos, consulte a documentação da classe `FragmentManager`.

Como demonstrado na seção anterior, é possível usar o `FragmentManager` para abrir uma `FragmentTransaction`, que permite realizar transações como adicionar e remover fragmentos.

Realização de transações com fragmentos

Um grande recurso fornecido por fragmentos em atividades é a possibilidade de adicionar, remover, substituir e realizar outras ações com eles em resposta à interação do usuário. Cada conjunto de alterações realizadas na atividade é chamado de transação e cada alteração pode ser feita usando APIs em `FragmentTransaction`. Também é possível salvar cada transação em uma pilha de retorno gerenciada pela atividade, permitindo que o usuário navegue inversamente pelas alterações de fragmento (semelhante à navegação inversa pelas atividades).

É possível adquirir uma instância de `FragmentTransaction` do `FragmentManager` da seguinte forma:

KOTLINJAVA

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
```

Cada transação é um conjunto de alterações que você quer realizar ao mesmo tempo. É possível definir todas as alterações desejadas para uma transação usando métodos como `add()`, `remove()` e `replace()`. Em seguida, para aplicar a transação à atividade, chame `commit()`.

Antes de chamar `commit()`, no entanto, você pode querer chamar `addToBackStack()` para adicionar a transação a uma pilha de retorno de transações de fragmentos. A pilha de retorno é gerenciada pela atividade e permite que o usuário retorne ao estado anterior do fragmento ao pressionar o botão *Voltar*.

Por exemplo, veja como você pode substituir um fragmento por outro e preservar o estado anterior da pilha de retorno:

KOTLINJAVA

```
// Create new fragment and transaction
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this
fragment,
// and add the transaction to the back stack
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

Neste exemplo, `newFragment` substitui qualquer fragmento (se houver) que estiver no contêiner do layout identificado pelo código `R.id.fragment_container`. Ao chamar `addToBackStack()`, a transação de substituição é salva na pilha de retorno para que o usuário possa reverter a transação e voltar ao fragmento anterior pressionando o botão *Voltar*.

Em seguida, `FragmentActivity` recupera automaticamente os fragmentos da pilha de retorno por meio de `onBackPressed()`.

Se você adicionar várias alterações à transação — como outra `add()` ou `remove()` — e chamar `addToBackStack()`, todas as alterações aplicadas antes de chamar `commit()` serão adicionadas à pilha de retorno como uma única transação, e o botão *Voltar* reverterá todas elas ao mesmo tempo.

A ordem em que você adiciona as alterações em `FragmentTransaction` não importa, exceto nas situações a seguir:

- Você precise chamar `commit()` por último.
- Você esteja adicionando vários fragmentos ao mesmo contêiner. Nesse caso, a ordem de inclusão determinará a ordem em que eles aparecerão na hierarquia de visualização.

Se você não chamar `addToBackStack()` ao realizar uma transação que remove um fragmento, ele será destruído quando a transação estiver confirmada e o usuário não poderá navegar de volta a ele. No entanto, se você chamar `addToBackStack()` ao remover um fragmento, ele será *interrompido* e retomado posteriormente, caso o usuário navegue de volta a ele.

Dica: para cada transação de fragmento, é possível aplicar uma animação de transição chamando `setTransition()` antes da realização.

Ao chamar `commit()`, a transação não será realizada de imediato. Em vez disso, o parâmetro agenda a execução no thread de IU da atividade (o thread "main", ou principal) assim que possível. Se necessário, no entanto, é possível chamar `executePendingTransactions()` do thread de IU para executar imediatamente as transações enviadas por `commit()`. Essa medida geralmente não é necessária, a não ser que a transação represente uma dependência para jobs em outros threads.

Atenção: é possível realizar uma transação usando `commit()` somente antes da atividade salvar o estado (quando o usuário deixa a atividade). Caso tente efetivar as alterações após esse ponto, uma exceção será lançada. Isso acontece porque o estado após a efetivação pode ser perdido se a atividade precisar ser restaurada. Para situações em que não há problema em perder a confirmação, use `commitAllowingStateLoss()`.

Comunicação com a atividade

Apesar de um `Fragment` ser implementado como um objeto independente de uma `FragmentActivity` e poder ser usado dentro de várias atividades, uma dada instância de um fragmento é diretamente vinculada à atividade que o hospeda.

Especificamente, o fragmento pode acessar a instância `FragmentActivity` com `getActivity()` e realizar facilmente tarefas como encontrar uma visualização no layout da atividade:

KOTLINJAVA

```
View listView = getActivity().findViewById(R.id.list);
```

Do mesmo modo, a atividade pode chamar métodos no fragmento adquirindo uma referência para o `Fragment` no `FragmentManager` usando `findFragmentById()` ou `findFragmentByTag()`. Por exemplo:

KOTLINJAVA

```
ExampleFragment fragment = (ExampleFragment)
getSupportFragmentManager().findFragmentById(R.id.example_fragment)
;
```

Criação de callbacks de evento para a atividade

Em alguns casos, você pode precisar de um fragmento para compartilhar eventos ou dados com a atividade e outros fragmentos hospedados pela atividade. Para compartilhar dados, crie um ViewModel compartilhado, conforme destacado na seção “Compartilhamento de dados entre fragmentos” no guia do ViewModel. Se você precisar propagar eventos que não podem ser tratados com um ViewModel, será possível definir uma interface de callback dentro do fragmento e solicitar que a atividade do host faça a implementação. Quando a atividade recebe um callback por meio da interface, ela pode compartilhar a informação com outros fragmentos no layout, conforme necessário.

Por exemplo, se um aplicativo de notícias em uma atividade tiver dois fragmentos — um para exibir uma lista de artigos (fragmento A) e outro para exibir um artigo (fragmento B) — o fragmento A deve informar à atividade quando um item de lista é selecionado para que ela possa instruir o fragmento B a exibir o artigo. Nesse caso, a interface `OnArticleSelectedListener` é declarada dentro do fragmento A:

KOTLINJAVA

```
public static class FragmentA extends ListFragment {
    ...
    // Container Activity must implement this interface
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }
    ...
}
```

Portanto, a atividade que hospeda o fragmento implementa a interface `OnArticleSelectedListener` e
modifica `onArticleSelected()` para notificar o fragmento B do evento do fragmento A. Para garantir que a atividade do host implemente essa interface, o método de callback `onAttach()` do fragmento A (que o sistema chama ao adicionar o fragmento à atividade)

instanciará `OnArticleSelectedListener` lançando a `Activity`, que é passada para `onAttach()`:

KOTLINJAVA

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener listener;
    ...
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            listener = (OnArticleSelectedListener) context;
        } catch (ClassCastException e) {
            throw new ClassCastException(context.toString() + "
must implement OnArticleSelectedListener");
        }
    }
    ...
}
```

Caso a atividade não tenha implementado a interface, o fragmento lançará `ClassCastException`. Se for bem-sucedida, o membro `mListener` reterá uma referência da implementação da atividade de `OnArticleSelectedListener`, para que o fragmento A possa compartilhar os eventos com a atividade chamando métodos definidos pela interface `OnArticleSelectedListener`. Por exemplo: se o fragmento A for uma extensão de `ListFragment`, sempre que o usuário clicar em um item de lista, o sistema chamará `onListItemClick()` no fragmento, que, em seguida, chamará `onArticleSelected()` para compartilhar o evento com a atividade:

KOTLINJAVA

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener listener;
    ...
    @Override
    public void onListItemClick(ListView l, View v, int position,
long id) {
        // Append the clicked item's row ID with the content
provider Uri
        Uri noteUri =
ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id);
        // Send the event and Uri to the host activity
        listener.onArticleSelected(noteUri);
    }
    ...
}
```

O parâmetro `id` passado para `onListItemClick()` é o código da linha do item clicado que a atividade (ou outro fragmento) usa para resgatar o artigo do `ContentProvider` do aplicativo.

Mais informações sobre como usar o provedor de conteúdo estão disponíveis na documentação Provedores de conteúdo.

Adição de itens à barra de aplicativo

Os fragmentos podem contribuir com itens de menu para o menu de opções da atividade (e consequentemente para a barra de aplicativo) implementando `onCreateOptionsMenu()`. Para que esse método receba chamadas, no entanto, você deve chamar `setHasOptionsMenu()` durante `onCreate()` para indicar que o fragmento gostaria de adicionar itens ao menu de opções. Caso contrário, o fragmento não receberá uma chamada para `onCreateOptionsMenu()`.

Quaisquer itens adicionados ao menu de opções do fragmento são anexados aos itens de menu existentes. O fragmento também recebe callback para `onOptionsItemSelected()` quando um item de menu é selecionado.

Também é possível registrar uma visualização no layout do fragmento para fornecer um menu de contexto chamando `registerForContextMenu()`. Quando o usuário abre o menu de contexto, o fragmento recebe uma chamada para `onCreateContextMenu()`. Quando o usuário seleciona um item, o fragmento recebe uma chamada para `onContextItemSelected()`.

Observação: apesar de o fragmento receber um callback selecionado no item para cada item de menu que adiciona, a atividade é a primeira a receber o respectivo callback quando o usuário seleciona um item de menu. Se a implementação da atividade do retorno de chamada selecionado no item não lida com o item selecionado, o evento é passado para o callback do fragmento. Isso é verdadeiro para o menu de opções e os menus de contexto.

Para saber mais sobre os menus, consulte o guia do desenvolvedor Menus e a aula de treinamento Barra de aplicativo.

Processamento do ciclo de vida dos fragmentos

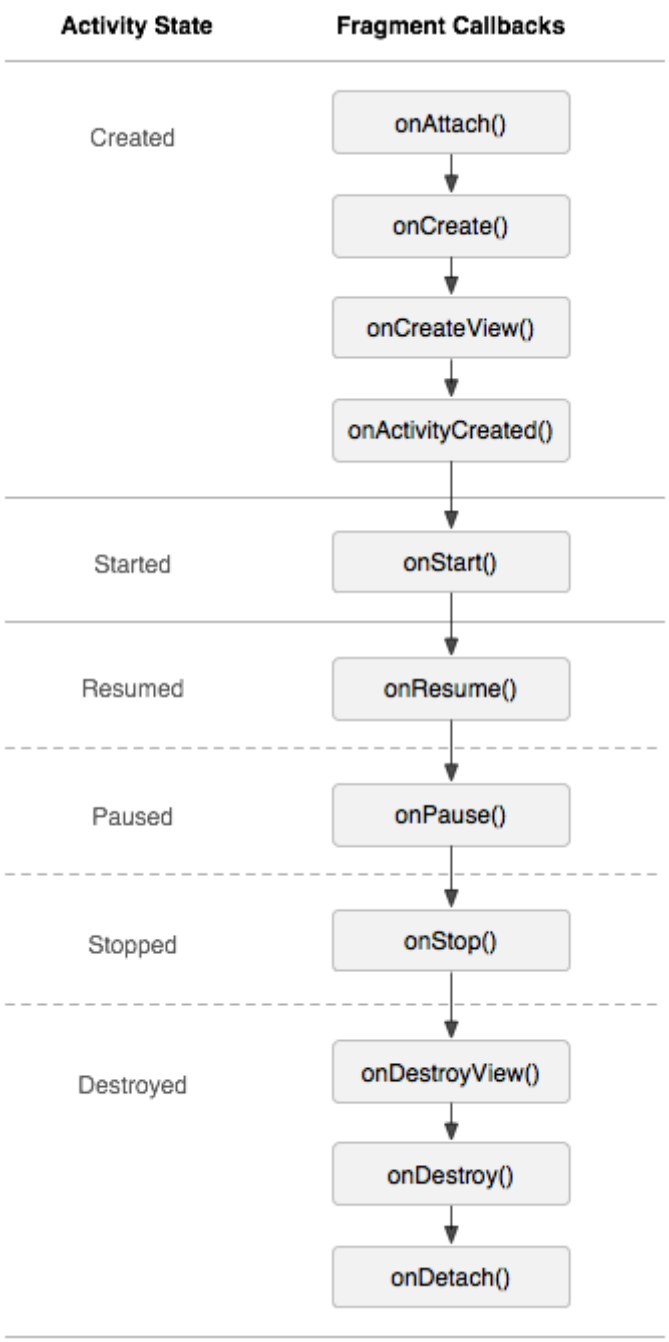

Figura 3. Efeito do ciclo de vida da atividade no ciclo de vida do fragmento.

Gerenciar o ciclo de vida de um fragmento é muito parecido com gerenciar o ciclo de vida de uma atividade. Assim como uma atividade, um fragmento pode existir em três estados:

Resumed

> O fragmento é visível na atividade em execução.

Paused

> Outra atividade está em primeiro plano, mas a atividade em que esse fragmento se encontra ainda está visível (a atividade de primeiro plano é parcialmente transparente ou não cobre a tela inteira).

Stopped

> O fragmento não está visível. A atividade do host foi interrompida ou o fragmento foi removido da atividade, mas adicionado à pilha de retorno. Um fragmento interrompido ainda está ativo (todas as informações do membro e de estado estão retidas no sistema). No entanto, não está mais visível para o usuário e será eliminado se a atividade também for.

Assim como uma atividade, é possível preservar o estado da IU de um fragmento entre alterações de configuração e processar a interrupção usando uma combinação de onSaveInstanceState(Bundle), ViewModel e armazenamento persistente local. Para saber mais sobre a preservação do estado da IU, consulte Como salvar estados de IU.

A diferença mais significativa entre o ciclo de vida de uma atividade e de um fragmento é o armazenamento nas respectivas pilhas de retorno. Por padrão, uma atividade é colocada de volta na pilha de retorno de atividades gerenciada pelo sistema quando interrompida (para que o usuário possa navegar de volta a ela com o botão *Back*, como discutido em Tarefas e pilha de retorno). No entanto, um fragmento é posicionado em uma pilha de retorno gerenciada pela atividade do host somente ao solicitar explicitamente que a instância seja salva chamando `addToBackStack()` durante uma transação que remove o fragmento.

Caso contrário, gerenciar o ciclo de vida do fragmento é muito semelhante a gerenciar o ciclo de vida da atividade. As mesmas práticas se aplicam. Consulte o guia O ciclo de vida da atividade e Como gerenciar componentes cientes do ciclo de vida para saber mais sobre o ciclo de vida da atividade e as práticas para gerenciá-lo.

Atenção: se for necessário um objeto `Context` no `Fragment`, é possível chamar `getContext()`. No entanto, tome cuidado para chamar `getContext()` somente quando o fragmento estiver anexado a uma atividade. Quando o fragmento ainda não tiver sido anexado ou tiver sido desconectado durante o ciclo de vida do fragmento, `getContext()` retornará nulo.

Coordenação com o ciclo de vida da atividade

O ciclo de vida da atividade em que o fragmento se encontra afeta diretamente o ciclo de vida do fragmento, da mesma forma que um callback de cada ciclo de vida da atividade resulta em um callback semelhante de cada fragmento. Por exemplo, quando a atividade receber `onPause()`, cada fragmento na atividade receberá `onPause()`.

Fragmentos têm alguns callbacks extras no ciclo de vida que processam a interação exclusiva com a atividade para realizar ações como criar e destruir a IU do fragmento. Esses métodos de callback adicionais são:

`onAttach()`

> Chamado quando o fragmento tiver sido associado à atividade (`Activity` é passado aqui).

`onCreateView()`

> Chamado para criar a hierarquia de visualizações associada ao fragmento.

`onActivityCreated()`

> Chamado quando o método `onCreate()` da atividade retornou.

`onDestroyView()`

> Chamado quando a hierarquia de visualizações associada ao fragmento estiver sendo removida.

`onDetach()`

> Chamado quando o fragmento estiver sendo desassociado da atividade.

O fluxo do ciclo de vida de um fragmento, conforme ele é afetado pela atividade de host, é ilustrado pela figura 3. Nessa figura, pode-se ver como cada estado sucessivo da atividade determina quais métodos de callback um fragmento pode receber. Por exemplo: quando a atividade recebe o callback `onCreate()`, um fragmento na atividade recebe nada mais do que o callback `onActivityCreated()`.

Quando a atividade atinge o estado retomado, é possível adicionar e remover fragmentos dela livremente. Portanto, somente quando a atividade estiver no estado retomado, o ciclo de vida de um fragmento poderá mudar de forma independente.

No entanto, quando a atividade deixa o estado retomado, o fragmento é novamente forçado a passar pelo ciclo de vida pela atividade.

Exemplo

Para reunir tudo que foi discutido neste documento, abaixo há um exemplo de uma atividade que usa dois fragmentos para criar um layout de dois painéis. A atividade abaixo inclui um fragmento para exibir uma lista de títulos de peças de Shakespeare e outra para exibir um resumo da peça quando selecionada na lista. Ela também determina como fornecer diferentes configurações de fragmentos com base na configuração da tela.

Observação: o código-fonte completo dessa atividade está disponível no app de amostra que mostra o uso de uma classe `FragmentLayout` de exemplo.

A atividade principal se aplica a um layout da maneira normal durante `onCreate()`:

KOTLINJAVA

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.fragment_layout);
}
```

O layout aplicado é `fragment_layout.xml`:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
android:layout_height="match_parent">

    <fragment
class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
            android:id="@+id/titles" android:layout_weight="1"
            android:layout_width="0px"
android:layout_height="match_parent" />

    <FrameLayout android:id="@+id/details"
android:layout_weight="1"
            android:layout_width="0px"
android:layout_height="match_parent"
            android:background="?android:attr/detailsElementBackgro
und" />

</LinearLayout>
```

Usando esse layout, o sistema instancia `TitlesFragment` (que lista os títulos das peças) assim que a atividade carrega o layout, enquanto `FrameLayout` (onde o fragmento para exibir o resumo da peça aparecerá) consome o espaço do lado direito da tela, mas primeiramente permanece vazio. Como você verá abaixo, ele não estará visível até que o usuário selecione um item na lista em que um fragmento esteja posicionado no `FrameLayout`.

No entanto, nem todas as configurações de tela são grandes o suficiente para exibir a lista de peças e o resumo lado a lado. Portanto, o layout acima é usado somente para configuração de tela de paisagem, salvando-o em `res/layout-land/fragment_layout.xml`.

Por isso, quando a tela está na orientação de retrato, o sistema aplica o seguinte layout, que é salvo em `res/layout/fragment_layout.xml`:

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
android:layout_height="match_parent">
    <fragment
class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
            android:id="@+id/titles"
            android:layout_width="match_parent"
android:layout_height="match_parent" />
</FrameLayout>
```

Esse layout inclui somente `TitlesFragment`. Isso significa que, quando o dispositivo está na orientação de retrato, somente a lista de título das peças está disponível. Portanto, quando o usuário clicar em um item da lista nesta configuração, o aplicativo iniciará uma nova atividade para exibir o resumo, em vez de carregar um segundo fragmento.

A seguir, é possível ver como isso é realizado com classes de fragmento. Primeiro, `TitlesFragment`, que exibe a lista de peças de Shakespeare. Esse fragmento estende `ListFragment` e depende dele para processar grande parte do trabalho da visualização de lista.

Enquanto inspeciona esse código, observe que há dois possíveis comportamentos quando um usuário clica em um item de lista: dependendo de qual dos dois layouts está ativo, ele pode criar e exibir um novo fragmento para mostrar os detalhes na mesma atividade (adicionando o fragmento `FrameLayout`) ou iniciar uma nova atividade (em que o fragmento possa ser exibido).

KOTLINJAVA

```
public static class TitlesFragment extends ListFragment {
    boolean dualPane;
    int curCheckPosition = 0;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Populate list with our static array of titles.
        setListAdapter(new ArrayAdapter<String>(getActivity(),
                android.R.layout.simple_list_item_activated_1,
Shakespeare.TITLES));

        // Check to see if we have a frame in which to embed the
details
        // fragment directly in the containing UI.
        View detailsFrame =
getActivity().findViewById(R.id.details);
        dualPane = detailsFrame != null &&
detailsFrame.getVisibility() == View.VISIBLE;

        if (savedInstanceState != null) {
            // Restore last state for checked position.
            curCheckPosition =
savedInstanceState.getInt("curChoice", 0);
        }

        if (dualPane) {
            // In dual-pane mode, the list view highlights the
selected item.
            getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE
);
            // Make sure our UI is in the correct state.
            showDetails(curCheckPosition);
        }
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("curChoice", curCheckPosition);
    }
```

```
    @Override
    public void onListItemClick(ListView l, View v, int position,
long id) {
        showDetails(position);
    }

    /**
     * Helper function to show the details of a selected item,
either by
     * displaying a fragment in-place in the current UI, or
starting a
     * whole new activity in which it is displayed.
     */
    void showDetails(int index) {
        curCheckPosition = index;

        if (dualPane) {
            // We can display everything in-place with fragments,
so update
            // the list to highlight the selected item and show the
data.
            getListView().setItemChecked(index, true);

            // Check what fragment is currently shown, replace if
needed.
            DetailsFragment details = (DetailsFragment)
                    getSupportFragmentManager().findFragmentById(R.
id.details);
            if (details == null || details.getShownIndex() !=
index) {
                // Make new fragment to show this selection.
                details = DetailsFragment.newInstance(index);

                // Execute a transaction, replacing any existing
fragment
                // with this one inside the frame.
                FragmentTransaction ft =
getSupportFragmentManager().beginTransaction();
                if (index == 0) {
                    ft.replace(R.id.details, details);
                } else {
                    ft.replace(R.id.a_item, details);
                }
```

```
        ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
                ft.commit();
            }

        } else {
            // Otherwise we need to launch a new activity to
display
            // the dialog fragment with selected text.
            Intent intent = new Intent();
            intent.setClass(getActivity(), DetailsActivity.class);
            intent.putExtra("index", index);
            startActivity(intent);
        }
    }
}
```

O segundo fragmento, `DetailsFragment` mostra o resumo da peça para o item selecionado na lista de `TitlesFragment`:

KOTLINJAVA

```
public static class DetailsFragment extends Fragment {
    /**
     * Create a new instance of DetailsFragment, initialized to
     * show the text at 'index'.
     */
    public static DetailsFragment newInstance(int index) {
        DetailsFragment f = new DetailsFragment();

        // Supply index input as an argument.
        Bundle args = new Bundle();
        args.putInt("index", index);
        f.setArguments(args);

        return f;
    }

    public int getShownIndex() {
        return getArguments().getInt("index", 0);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
```

```
        if (container == null) {
            // We have different layouts, and in one of them this
            // fragment's containing frame doesn't exist. The
fragment
            // may still be created from its saved state, but there
is
            // no reason to try to create its view hierarchy
because it
            // isn't displayed. Note this isn't needed -- we could
just
            // run the code below, where we would create and return
the
            // view hierarchy; it would just never be used.
            return null;
        }

        ScrollView scroller = new ScrollView(getActivity());
        TextView text = new TextView(getActivity());
        int padding =
(int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
                4,
getActivity().getResources().getDisplayMetrics());
        text.setPadding(padding, padding, padding, padding);
        scroller.addView(text);
        text.setText(Shakespeare.DIALOGUE[getShownIndex()]);
        return scroller;
    }
}
```

Lembre-se da classe `TitlesFragment` que, se o usuário clicar em um item de lista e o layout atual *não* incluir a visualização `R.id.details` (que é o lugar de `DetailsFragment`), o aplicativo iniciará a atividade `DetailsActivity` para mostrar o conteúdo do item.

Veja a seguir a `DetailsActivity`, que simplesmente incorpora `DetailsFragment` para mostrar o resumo da peça selecionada quando a tela está na orientação de retrato:

KOTLINJAVA

```
public static class DetailsActivity extends FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation
                == Configuration.ORIENTATION_LANDSCAPE) {
            // If the screen is now in landscape mode, we can show
the
            // dialog in-line with the list so we don't need this
activity.
            finish();
            return;
        }

        if (savedInstanceState == null) {
            // During initial setup, plug in the details fragment.
            DetailsFragment details = new DetailsFragment();
            details.setArguments(getIntent().getExtras());
            getSupportFragmentManager().beginTransaction().add(andr
oid.R.id.content, details).commit();
        }
    }
}
```

Essa atividade finaliza se a configuração for de paisagem, pois a atividade principal pode tomar o controle e exibir `DetailsFragment` junto com `TitlesFragment`. Isso pode acontecer se o usuário iniciar `DetailsActivity` enquanto estiver na orientação de retrato, mas alternar para orientação de paisagem (o que reinicia a atividade atual).

Recursos adicionais

`Fragment` é usado no app de demonstração Sunflower.