ViewHolder

Contextualização

O padrão ViewHolder surgiu junto com o elemento ListView, usado para fazer uma listagem de itens na aplicação. Esse elemento era amplamente usado no desenvolvimento, mas tinha alguns problemas. O primeiro e talvez mais crítico era que uma listagem muito grande muitas vezes se tornava pesada e apresentava a sensação de que o aplicativo está travando.

O motivo disso que é para cada linha da ListView, eram buscados todos os elementos que essa lista continha através do método *findViewById()* que era exatamente o que tornava a performance ruim, pois esse método é custoso para a aplicação.

Para melhorar a performance o foco foi no método que era mais custoso, *findViewById().* Sua finalidade era bem simples, buscar elementos da interface para serem manipulados com valores em texto, datas, imagens e assim por diante de acordo com cada item da listagem.

Contudo, notou-se que o objetivo da ListView poderia ser atingido mantendo somente uma referência aos elementos da listagem uma vez já buscados ao invés de buscar repetidas vezes o mesmo elemento.

Dessa maneira, surgiram alguns padrões de código que tentavam minimizar esse custo, o que traz ao conceito ViewHolder.

Conceito

Observe o código abaixo:

```
EditText editReal = (EditText) this.findViewById(R.id.edit_real);
TextView textDollar = (TextView) this.findViewById(R.id.text_dollar);
TextView textEuro = (TextView) this.findViewById(R.id.text_euro);
Button buttonCalculate = (Button) this.findViewById(R.id.button_calculate);
```

Imagine a execução deste código diversas vezes durante uma listagem de 100 itens, por exemplo. Usando um exemplo real:

```
int i;
for (i = 0; i < 10; i++) {

    EditText editReal = (EditText) this.findViewById(R.id.edit_real);
    TextView textDollar = (TextView) this.findViewById(R.id.text_dollar);
    TextView textEuro = (TextView) this.findViewById(R.id.text_euro);
    Button buttonCalculate = (Button) this.findViewById(R.id.button_calculate);

    editReal.setText(R.string.teste);
    textDollar.setText(R.string.teste);
    textEuro.setText(R.string.teste);
    buttonCalculate.setText(R.string.teste);

}
```

Muitas execuções e requisições para processar, correto? Nesse trecho percebemos a execução do *findViewById()* diversas vezes e poderia ocorrer mais vezes caso existissem mais itens.

Observe o trecho abaixo com uma abordagem diferente:

```
private ViewHolder mViewHolder = new ViewHolder();

private static class ViewHolder {
    private EditText editReal;
    private TextView textDollar;
    private TextView textEuro;
    private Button buttonCalculate;
}

private void bindData() {
    this.mViewHolder.editReal = (EditText) this.findViewById(R.id.edit_real);
    this.mViewHolder.textDollar = (TextView) this.findViewById(R.id.text_dollar);
    this.mViewHolder.textEuro = (TextView) this.findViewById(R.id.text_euro);
    this.mViewHolder.buttonCalculate = (Button) this.findViewById(R.id.button_calculate);

    int i;
    for (i = 0 ; i < 10; i++) {
        this.mViewHolder.editReal.setText(R.string.teste);
        this.mViewHolder.textDollar.setText(R.string.teste);
        this.mViewHolder.textEuro.setText(R.string.teste);
        this.mViewHolder.buttonCalculate.setText(R.string.teste);
    }
}
```

O que se tem agora é a criação de uma classe que armazena os elementos de interface para que não sejam buscados todas as vezes que forem manipulados. Existe então a reusabilidade de uma referência que torna o trecho mais performático, uma vez que não temos mais a execução do *findViewById().*

Nota: *A execução real da ListView é diferente do código apresentado, porém esse exemplo é usado para contextualizar o que acontece de maneira simplificada. Caso deseje visualizar a implementação real, consulte as fontes no final do documento.*

Uso durante o curso

Durante o curso são criadas várias Activities e Layouts. Cada um desses Layouts possuem elementos de interface que são manipulados. Por que então não trazer o conceito de ViewHolder para a Activity?

Observe o código abaixo:

```
private static class ViewHolder {
    private EditText editReal;
    private TextView textDollar;
    private TextView textEuro;
    private Button buttonCalculate;
}

// Contém os elementos da interface.
// Faz o carregamento uma única vez e pode ser usado a qualquer momento dentro da classe.
private ViewHolder mViewHolder = new ViewHolder();

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

    // Insere o layout na activity
    setContentView(R.layout.activity_main);

    // Busca os elementos da interface
    this.mViewHolder.editReal = (EditText) this.findViewById(R.id.edit_real);
    this.mViewHolder.textDollar = (TextView) this.findViewById(R.id.text_dollar);
    this.mViewHolder.textEuro = (TextView) this.findViewById(R.id.text_euro);
    this.mViewHolder.buttonCalculate = (Button) this.findViewById(R.id.button_calculate);

}
```

Perceba a presença da classe ViewHolder e como ela agrega os elementos de interface que são manipulados. A partir do momento que o método onCreate() é chamado e sua execução é completa, qualquer manipulação dos elementos pode ser feita usando a referência à classe ViewHolder e dessa maneira não há mais a necessidade de buscar o elemento de interface, uma vez que já está instanciado.

É importante notar também que a ViewHolder pode não ser usada no código acima, deixando somente a referência aos elementos de interface como atributos da classe e isso funcionaria perfeitamente, porém por questões de legibilidade de código, organização dos

elementos de interface de maneira agrupada e manter um padrão no curso, o padrão ViewHolder é utilizado nas Activities onde elementos são manipulados.

Fontes

---

1. https://developer.android.com/training/improving-layouts/smooth-scrolling.html, disponível em 13/09/2017.
2. http://blog.alura.com.br/utilizando-o-padrao-viewholder/, disponível em 13/09/2017.