Java Básico

Apostila de Java

FURG

Fabio Aiub Sperotto

This book is for sale at http://leanpub.com/apostilajava

This version was published on 2013-02-21

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012-2013 Fabio Aiub Sperotto

Conteúdo

Sobre Java **1**

Capítulo 1 **2**
Revisão de Orientação a Objetos 2

Capítulo 2 **4**
Características da Linguagem 4
Fundamentos da Linguagem 4

Capítulo 3 **8**
Hora de Programar 8
Exercícios 12

Capítulo 4 **14**
Desenvolvendo Interfaces Gráficas 14

Referências **20**

Sobre Java

A Sun Mycrosystems estava observando o desenvolvimento dos microprocessadores e a popularização de computadores pessoais quando em 1991 criou uma pesquisa chamada Green para averiguar a situação do mercado a fim de tomar algumas decisões empresariais. Esta pesquisa resultou em uma linguagem de programação, baseada em C e em C++ que seria chamada de Oak pelo seu criador James Gosling [Deitel et al.,2003]. Oak significa 'carvalho', James deu este nome em homenagem a uma árvore que dava de frente para a janela do seu escritório. Sendo este um exemplo famoso da grande criatividade dos programadores em dar nome as variáveis que utilizam.

Voltando a história, mais tarde verificou-se que já existia uma linguagem de programção chamada Oak (viram a criatividade?). O rebatizado para o que conhecemos por Java é um mistério. Para evitar confusões trocaram para "Greentalk". Não se sabe ao certo como os envolvidos chegaram a Java, mas o nome foi sugerido após conversas entre os membros do projeto Green. Algumas fontes relatam que os desenvolvedores adoravam café e conheceram, em uma cafeteria, um café oriundo da ilha de Java, que fica na Indonésia. Assim, programação + cafeina resultou em uma sugestão que foi aprovada no projeto Green.

Após alguns problemas de mercado, inclusive a perda de contrato sobre tecnologias para uma empresa rival, a Sun viu um grande despertar no desenvolvimento tecnológico na explosão da World Wide Web e seu ecossistema de aplicativos. Em 1995 a Sun, então, anunciou formalmente em uma grande conferência o Java. Devido a época, o anúncio chamou bastante atenção comercial sobre Java e a partir daí, o projeto decolou.

Hoje em dia, segundo a Sun, Java é responsável por atrair 9 milhões de desenvolvedores e Java está por todos os segmentos do mercado. De aplicações para computadores, para dispositivos portáteis; De sistemas para a Internet até grandes datacenters; De games até supercomputadores científicos. Do software ao hardware Java mantém seu status de grande linguagem de programação.

Capítulo 1

Revisão de Orientação a Objetos

Como esta apostila se destina a iniciantes em Java e tem como pré-requisito compreender pelo menos algoritmos, se faz necessária uma revisão nos conceitos sobre orientações a objetos em linguagem de programação. Java é orientando a objeto. Portanto para esta linguagem é necessário compreender sobre objetos, classes, herança, interface. Os detalhes de implementação serão disponibilizados nos próximos capítulos.

Objetos

Objeto é o principal conceito na orientação a objetos. Esse conceito é inspirado em qualquer coisa que possamos visualizar no mundo real. Muitos exemplos podem ser vistos sem esforço, como a mesa que está utilizando, computador, lâmpadas, etc. Se você pensar bem sobre estes objetos, duas características primordiais são encontradas: estado e comportamento.

Um celular comum pode ter alguns estados como *ligado*, *desligado*, *volume atual*. Este mesmo celular pode ter comportamentos como *aumentar volume*, *atender ligação*, *desligar ligação*. Pensando em programação, estados são as variáveis do celular. Os comportamentos são os métodos (funções) que compõem o celular. Os métodos são utilizados para operar um determinado estado do objeto e é a principal forma de realizar comunicação objeto a objeto. Esconder estados internos do objeto e forçar que todas as interações a serem realizadas sejam por via de métodos, é o famoso conceito de *encapsulamento de dados*.

Classes

Podemos visualizar que vários objetos no mundo real são do mesmo tipo. Celulares podem ser de várias marcas ou tipos mas suas características básicas (estados e métodos) serão sempre as mesmas. Ainda que existam celulares de vários modelos podemos constituir uma espécie de estrutura que codifique o objeto Celular e que podemos utilizar para todos os modelos. Essa estrutura chama-se Classe e podemos instanciar um objeto celular dessa estrutura, representando em linguagem de programação, um celular em específico. Nos próximos capítulos vamos desenvolver exemplos práticos.

Herança

Continuando a observar o mundo real, verificamos que existem vários tipos de celulares que compartilham as mesmas características básicas. Mas também existem equipamentos que possuem características adicionais que os diferenciam no mesmo grupo. Existem celulares comuns que efetuam ligações e enviam mensagens, mas existem outros chamados Smartphones que além de tirar fotos, podem se conectar a uma rede *wireless*, utilizar aplicativos modernos, editar documentos e até acessar TV.

Estes comportamentos novos de conectar a rede *wireless* e editar documentos podem não constar em todos os equipamentos de celulares, mas *aumentar volume*, *atender ligação* e *desligar ligação* são comportamentos compartilhados entre todos. Programação orientada a objetos permite herdar estados e comportamentos comuns de uma classe para outras classes.

Então, podemos ter uma super classe chamada **Celular** com estados de ligado, desligado, volume atual e comportamentos como aumentar volume, atender ligação. Outras classes, como *Smartphone* podem herdar o funcionamento básico de *Celular* e adicionar em suas respectivas classes novos estados e comportamentos, tornando-as subclasses de Celular. Cada classe só pode ter uma única super classe. Uma super classe pode ter um número ilimitado de subclasses. Outras definições como classes pai/mãe e classes filhos também podem ser utilizadas nesse contexto.

Interface

Uma interface realiza uma espécie de contrato dos métodos para uma determinada classe. Uma espécie de promessa de quais métodos uma classe precisa ter, mas a lógica, como esses métodos devem ser construídos, fica a cargo de quem implementa essa classe. O botão de ligar do celular é um exemplo no mundo real de uma interface mediando você com o equipamento eletrônico atrás do botão *power*.

Sua utilização serve para formalizar quais métodos uma classe precisa implementar, que um objeto precisa suportar. Especificando as assinaturas de métodos e funcionalidades que um objeto possuirá.

A classe que implementar uma interface só conseguirá compilar com sucesso se todos os métodos da interface estiverem implementadas na classe.

Pacotes e API

Pacotes é uma forma de organização de um conjunto de classes. Assim como nos organizamos no computador criando pastas ou diretórios para separar arquivos, pacotes tem o mesmo objetivo. Uma aplicação pode conter vários pacotes que agrupam um conjunto de classes segundo algum critério do programador.

API é a sigla para *Application Programming Interfaces* (interfaces de programas aplicativos). As APIs são bibliotecas de programação disponibilizadas por outros programadores. Existem uma quantidade grande de APIs e antes de implementar algo ou resolver um problema, sempre é bom pesquisar se a linguagem de programação possui um recurso[1] ou se alguém já não criou um recurso e o disponibilizou. Alguns autores afirmam que existem dois grandes mundos em Java: primeiro aprender a linguagem em si para criar as próprias classes e depois aprender a utilizar o ecossistema de API para a programação.

[1] http://docs.oracle.com/javase/7/docs/api/index.html

Capítulo 2

Características da Linguagem

Java Virtual Machine - JVM

Java é uma linguagemn de programação de alto nível, isto significa que precisa de um tradutor para que o computador compreenda as instruções do programador. Quem possui o papel de tradução é o compilador. O compilador gera um executável do programa e este programa pode ser executado em um determinado tipo de computador, se quiser que o programa rode em outros tipos de máquinas, outros compiladores deverão ser utilizados para criar o executável necessáro.

Uma alternativa a isso é o interpretador. Sua diferença ao compilador é que o interpretador funciona como uma espécie de processador que interpreta linha a linha as instruções oriundas da linguagem de programação. Um benefício é que interpretando um programa este pode ser executado em várias máquinas diferentes, permitindo que você possa usar uma linguagem de programação feita para um tipo de máquina, em outras.

Java utiliza uma combinação do compilador e do interpretador [Eck, 2007]. Programas em Java são compilados para linguagem de máquina, mas não como outros linguagens de programação. Essa máquina para o qual sera compilado é uma espécie de "máquina virtual", chamada portanto de **Java Virtual Machine**. Ao ser compilado para o JVM existe uma linguagem de máquina chamada Java bytecode. Esse Java bytecode funcionará como um interpretador. Assim temos que um programa Java pode ser executado em qualquer máquina, desde computadores até celulares. Sendo necessário que cada tipo de máquina possua um interpretador Java bytecode para a sua arquitetura.

Fundamentos da Linguagem

Esta seção apresenta as principais características da linguagem de programação, palavras reservadas, tipos de dados, entre outros. Todas as informações foram retiradas de pesquisas nas apostilas de estudos de exame de certificação de Java 2 que não constam nas referências.

Palavras Reservadas

Toda linguagem de programação possui palavras reservadas. Essas palavras se destinam ao uso restrito pelo compilador que precisa delas para determinar o que fazer com o código. Logo as palavras da relação a seguir não podem ser utilizadas pelo programador em nomes de classes, funções, variáveis, etc.

abstract	boolean	break	byte
case	catch	char	class
const	continue	default	do
double	else	extends	final
finally	float	for	goto
if	implements	import	instanceof
int	interface	long	native
new	package	private	protected
public	return	short	static

strictfp	super	switch	synchronized
this	throw	throws	transient
try	void	volatile	while
assert			

Modificadores de Classe, Método e Atributo

Muitos dos modificadores ou moderadores de acesso descritos a seguir podem ser semelhantes ou iguais entre classes, métodos e atributos, entretanto vamos dividir esses grupos para rápida fixação.

Classes por padrão (*default*) só podem ser acessadas por outras classes no mesmo pacote, para outros casos é necessário conhecer quais modificadores devem ser utilizados. Os modificadores de acesso para classes são:

- **public**: classes podem ser utilizadas por objetos de fora do pacote, é utilizada por todos.
- **abstract**: a classe não pdoe ter objetos instanciados. Para que serve então? Para definir que uma classe não pode ainda ter objetos, por talvez não estar totalmente definidia, entretanto, subclasses (classes filhas) podem herdar métodos de classes abstratas e definir o comportamento dos mesmos.
- **final**: a classe não pode ter subclasses.

Os modificadores de acesso para métodos são:

- **public**: o método pode ser acessado por qualquer classe em qualquer pacote, respeitando os níveis de visibilidade entre as classes.
- **protected**: o método pode ser acessado por subclasses da classe ou por classes no mesmo pacote.
- **private**: torna uma variável ou função acessível somente dentro de sua própria classe.
- **abstract**: não implementa funcionalidade, a obrigação de implementação fica com uma classe não abstrata que herdar esse método. Lembre que se um método é abstrato, sua classe também é!
- **final**: o método não pode ser sobrescrito, sobreposto.
- **static**: método pode ser acessado diretamente pela sua classe, sem a necessidade de instanciar um objeto (Classe.método).
- **native**: indica que um método é escrito em uma linguagem de plataforma-dependente como C ou C++.
- **synchronized**: indica que um método só pode ser acessado por uma única *thread* por vez.

Os modificadores de acesso para atributos são:

- **public**: o atributo pode ser acessado por qualquer um, igual a um método público.
- **protected**: o atributo pode ser acessado por subclasses da classe ou por classes no mesmo pacote.
- **private**: torna um atributo acessível somente dentro de sua própria classe.
- **final**: indica que o atributo guarda um valor fixo que não pode ser alterado, é utilizado para definir constantes.

- **static**: o atributo definido como static compartilha seu valor por todos os objetos da classe, se o valor desse atributo é alterado, isto é refletido para todos os objetos da mesma classe desse atributo static.

Outros modificadores são:

- **implements**: é usado para indicar a interface que a classe implementará.
- **interface**: palavra usada para especificar uma interface.
- **new**: usado para instanciar um objeto pela invocação do construtor.
- **strictfp**: usada na frente de um método ou classe para indicar que números de ponto flutuante vão seguir as regras de FP-strict em todas as expressões.
- **transient**: previne que campos nunca sejam serializados. Campos transientes são sempre ignorados quando objetos são serializados.
- **volatile**: indica que uma variável pode mudar de sincronização devido ao seu uso em *threads*.

Manipulando Erros

- **catch**: declara um bloco de código para manusear uma exceção.
- **finally**: bloco de código geralmente seguido por uma instrução *try-catch* que executa as linhas de instrução não importando o manuseio feito com relação as exceções.
- **throw**: usado para passar a exceção para o método que está chamando *throw*.
- **throws**: indica que o método vai passar uma exceção ao método que está chamando esta instrução.
- **try**: bloco de código que será tentado a execução mas, pode causar uma exceção.
- **assert**: avalia uma expressão de condicação para verificar a suposição do programador.

Controle de Pacote

Palavras chave para controle de pacotes.

- **import**: instrução para importar pacotes ou classes no código.
- **package**: especifica para qual pacote de classes o arquivo faz parte.

Palavras chave para Variáveis e Métodos

As seguintes palavras chaves são um tipo especial de referência as variáveis.

- **super**: referência a variável imediatamente da super classe.
- **this**: referência a uma variável ou método da instância (objeto) corrente.

No caso de métodos ainda existe a palavra chave **void** determinando que uma função não tem valor de retorno.

Tipos Primitivos

- **boolean**: valor indiciando verdadeiro (true) ou falso (false).
- **byte**: inteiro de 8 bits.
- **char**: um único caracter Unicode.
- **double**: número ponto-flutuante em 64 bits.
- **float**: número ponto-flutuante em 32 bits.
- **int**: número inteiro em 32 bits.
- **long**: número inteiro em 64 bits.
- **short**: número inteiro em 16 bits.

Outros Tipos

- **String**: objeto que representa uma sequência de caracteres.
- **array[]**: array de primitivos. Também podem ser criados arrays n-dimensionais acrescentando um conjunto de colchetes para cada dimensão, existe também os aspectos de declaração e a construção de *arrays*, todos os exemplos no código abaixo.

```
1 //declaração de array de números inteiros:
2 int[] conjuntoNumeros;
3
4 //declaração de array de números inteiros:
5 int[][] conjuntoNumeros;
6
7 //declaração e construção de um array
8 int[] conjuntoNumeros;
9 conjuntoNumeros = new int[5];
```

Apesar da utilidade de *arrays* se recomenda o uso de outras classes de *Collection*, como o ArrayList[a] por exemplo, por possuirem maiores benefícios na manipulação dos dados.

[a] http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

Se utilizar a palavra chave Class, também é possível criar um array com objetos de uma classe e de suas subclasses na mesma estrutura de dados. Exemplo: *Celular [] meuCelulares = {new Celular(), new Smartphone(), new Celular()}*. Assim temos um array com três objetos, dois objetos de celular e um de smartphone.

Preparado para programar?

Capítulo 3

Hora de Programar

Vimos no capítulo 1 alguns conceitos sobre orientação a objetos, sendo que dois grandes elementos que devemos visualizar em objetos no mundo real são os estados e comportamentos. Vimos no capítulo 2 as características essenciais da linguagem Java. Agora precisamos colocar isso em prática em uma aplicação simples, ao invés de celulares, vamos pensar em rádios![2]

Da Interface para as Classes

Vamos planejar nossa codificação. Imagine um rádio comum, quais são os estados que um rádio pode ter? Ligado, desligado, volume atual. E os comportamentos? Ligar, desligar, aumentar volume, trocar de estação. Então, como estamos planejando, vamos criar uma interface com a planta do que uma classe rádio deveria ter.

```
1 public interface Radio{
2
3    public void ligar();
4    public void desligar();
5    public void aumentarVolume();
6    public void diminuirVolume();
7    public void mudarFrequencia();
8 }
```

Veja que não está incluído os estados do rádio. Atributos devem ser inseridos na classe que irá implementar os comportamentos que modelamos na interface acima. Os únicos atributos que devem ser informados na interface são constantes definidas, que são aquelas variáveis que utilizam os modificadores *static* e *final*.

É importante empreender um bom tempo na fase de modelagem do sistema. A modelagem pode envolver a captação de requisitos funcionais e não funcionas que a aplicação irá ter, desenho do banco de dados e também do próprio código. Com um bom tempo gasto na modelagem é possível perceber futuras armadilhas e delinear bem a construção de classes e seus relacionamentos. Neste caso vamos nos limitar a interface acima, através dela podemos codificar uma classe chamada *RadioAMFM*.

[2] Os exemplos de códigos aqui são para apresentação dos conceitos, talvez você, como programador, possa ter outras concepções ou planejar o código de forma diferente (padrões de projeto, não serão apresentados nesta apostila).

```
1  public class RadioAMFM implements Radio {
2
3      public int volumeAtual = 50;
4      private int ligado;
5      public double faixaFrequencia[] = {100.5, 100.8, 205.8, 357.54};
6      public double frequenciaAtual = this.faixaFrequencia[0];
7
8      public RadioAMFM(){
9
10     }
11
12     public void ligar(){
13         this.setLigado(1);
14     }
15
16     public void desligar(){
17         this.setLigado(0);
18     }
19
20     public void aumentarVolume() {
21
22     }
23
24     public void diminuirVolume() {
25
26     }
27 }
```

Erro

Todos os métodos da interface precisam ser implementados.

A classe *RadioAMFM* implementa as definições da interface *Radio* (`public class RadioAMFM implements Radio`). Outros métodos também foram inseridos para dar suporte a lógica. Alguns atributos (a partir de agora vamos utilizar atributos ao invés de estado) além de serem implementados receberam valores logo na sua declaração para usarmos na aplicação.

Veja que existe um atributo `ligado` que tem como modificador `private`. Atributo privado somente pode ser modificado de dentro da classe, assim, os métodos `ligar()` e `desligar()` disponibilizam meios para avisar se o rádio está ligado ou não.

Continuando a observar, outros objetos podem fazer parte dessa aplicação sobre rádios. A interface *Radio* que planejamos contém comportamentos de um rádio simples, mas também existem os *cd players*. Assim precisamos planejar como vamos inserir um esquema de *cd player* na nossa aplicação. Uma exemplo de interface pode ser a seguinte:

```
1  public interface CDPlayer {
2
3      public void inserirCD();
4      public void ejetarCD();
5      public void mudarFaixa(int faixaCDAtual);
6
7  }
```

A interface *CDPlayer* possui três assinaturas de métodos que denotam alguns dos comportamentos básicos de um tocador de CD: inserir cd, ejetar cd e mudar a faixa. Perceba que nesta interface, no método `mudarFaixa`, tem um atributo sobre a faixa de música corrente, que já foi definido como obrigatório, neste exemplo, para a implementação.

Bem, agora temos duas interfaces, uma para um radio simples e outra para um tocador de cd. Também já iniciamos a implementação de um rádio simples chamado *RadioAMFM*. E agora? Um rádio também pode ter um tocador de cd, não acha? Podemos dizer que um determinado aparelho pode ser um RadioAMFM com tocador de cd.

Para a nossa aplicação vemos que esse aparelho rádio + cd poderá herdar as características do *RadioAMFM* (além de implementar novas funcionalidades para tocar cd. Assim podemos ter a seguinte classe ***RadioCDPlayer***:

```
1  public class RadioCDPlayer extends RadioAMFM implements CDPlayer {
2
3      public boolean cdinserido = false;
4
5      public RadioCDPlayer() {
6          super();
7      }
8
9      public void inserirCD() {
10
11 }
12
13    public void ejetarCD() {
14
15    }
16
17    public void mudarFaixa(int faixaCDAtual) {
18
19    }
20
21    private int getQuantFaixas(){
22
23    }
24 }
```

Veja na linha 1. A classe ***RadioCDPlayer*** estende a classe *RadioAMFM*, ou seja, torna-se classe filha herdando as funcionalidades/características da classe *RadioAMFM*. Também implementa o que planejamos na interface *CDPlayer*.

Instanciando Objetos, Utilizando Métodos

Na última seção vimos como projetar nossa aplicação e criamos as classes, que nada mais são como moldes, que modelam os objetos do mundo real (rádios, *cd players*, etc). Precisamos também saber como instanciar essas classes, trazer para um plano físico o que podemos criar a partir de um molde.

Em programação orientada a objetos isso quer dizer que, ainda no plano virtual, precisamos instanciar uma classe para usa-la. A instância de uma classe se chama objeto. Um objeto é instanciado através do operador ***new*** em conjunto com o nome de sua classe. Veja um exemplo:

```
1 RadioAMFM radio = new RadioAMFM();
```

Após a instanciação, temos um objeto *radio* pronto para uso. Todos os métodos implementados na classe RadioAMFM ficam disponíveis para este objeto. É possível ter vários objetos rádios instanciados, compartilhando os mesmos métodos definidos, mas cada qual, em uma parte diferente da memória. Assim podemos ter vários objetos de uma mesma classe, com comportamentos diferentes, ao mesmo tempo.

Para acessar um determinado método de um objeto, é necessário utilizar uma anotação que pode variar de linguagem para linguagem. Veja o exemplo abaixo:

```
1 RadioAMFM radio = new RadioAMFM();
2 radio.aumentarVolume();
```

A anotação em Java para acessar os métodos é no formato `objeto.nomeMétodo()`. Neste caso estamos dizendo ao objeto *radio* para aumentar o volume.

Métodos públicos podem ser chamados em objetos fora de sua classe de criação. Os métodos privados só podem ser chamados de dentro de suas classes.

Executando a Aplicação

Após a implementação das classes devemos utilizá-las como mostrado na seção anterior. Para executar programas em Java precisamos de um método chamado `main()` (principal). Quando este método é inserido em alguma classe, quer dizer que esta classe possui um programa a ser executado. O conjunto de classes que criamos anteriormente não possui esse método pois são classes que guardam informações base para a aplicação.

Para usarmos esse método vamos criar uma nova classe, chamada de *Main* só para sabermos que é a principal de execução (pode usar qualquer outro nome):

```
1  public class Main {
2
3      public Main() {
4
5   }
6
7   public static void main(String[] args) {
8
9          RadioAMFM radio = new RadioAMFM();
10
11  }
12 }
```

Tudo o que for escrito dentro deste método `main()` será executado. Veja que já temos uma instanciação de um objeto na linha 9. Ao compilar o programa (no Eclipse é o atalho ctrl + F11), nada será mostrado no console a não ser que erros existam no projeto.

Agora sim, experimente chamar um método do objeto e veja qual é o efeito no console da aplicação, o método `main()` atualizado com um exemplo fica assim:

```
1  public static void main(String[] args) {
2
3      RadioAMFM radio = new RadioAMFM();
4      radio.setLigado(1);
5  }
```

Talvez números inteiros não sejam a melhor solução para verificação de componentes ligados ou desligados, dependendo do projeto é mais interessante utilizar os booleanos *false* (falso) ou *true* (verdadeiro). Pense a respeito, qual seria a melhor forma de descrever estados de desligado e ligado sem dar margem para outros estados?

Exercícios

Os exercícios a seguir são relacionados com as classes deste capítulo e códigos do repositório.

1. Implemente todos os métodos da classe *RadioAMFM*, sempre quando possível, inclua mensagens na tela para verificar o andamento de sua lógica (`System.out.println()`). Em seguida na classe *Main* instancie e use o objeto de *RadioAMFM*, testando suas implementações.
2. Utilize os atributos de frequência da classe *RadioAMFM* e implemente um ou mais métodos para dar suporte ao rádio para mudar de estação ou frequência. O atributo *faixaFrequencia* pode ser um conjunto de valores de frequência e sempre que o rádio mudar de estação o

atributo *frequenciaAtual* deverá ser atualizado (experimente iniciar *frequenciaAtual* com um valor do conjunto de frequências).

3. Implemente todos os métodos da classe *RadioCDPlayer*. Pense da mesma forma para testar como foi feito no exercício 1.
4. Defina o atributo volumeAtual da classe RadioAMFM como privado e implemente os métodos *getters* e *setters* para manipular o atributo.
5. Crie o método `public void display(String info)` na classe *RadioAMFM* e o método `public String nomeMusica()` na classe *RadioCdPlayer*. Inclua nesses dois métodos a lógica necessária para que no método `main()` o método `display()` possa mostrar na tela a faixa que está sendo tocada pelo cd através do método `nomeMusica()`.
6. Insira uma forma de verificar se existe inicialmente um cd inserido antes de mudar de faixa, no método que altera a faixa de músicas da classe *RadioCdPlayer*.
7. Pense em outros equipamentos do mundo real comparado a sistemas de rádios. Rádio comunicadores como WalkieTalk, por exemplo. Implemente a classe *WalkieTalk* e verifique o que pode reutilizar de métodos das outras classes. Também codifique novos métodos para satisfazer os objetos *walkietalks*.

Capítulo 4

Desenvolvendo Interfaces Gráficas

Dependendo da aplicação desenvolvida, as vezes é fundamental fornecer uma interface de fácil uso para que usuários comuns possam utilizar a sua aplicação desenvolvida sem ter a necessidade de compilar o código-fonte. Também pode ser que seja necessário entregar somente o executável da aplicação, sem o código-fonte. Para isto, existem na maioria das linguages de programação, componentes de interface de usuário gráfica. Neste capítulo será introduzido o desenvolvimento de interfaces gráficas com Swing.

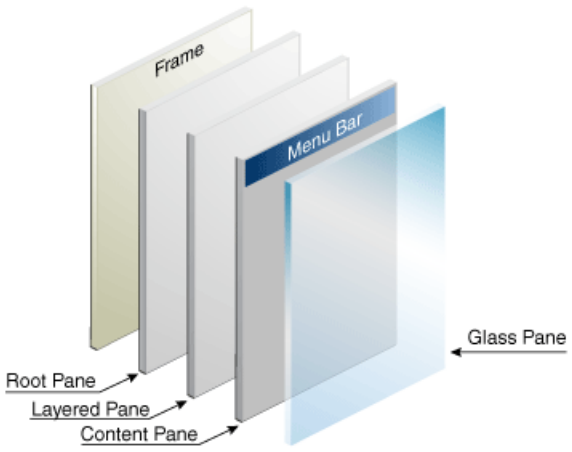
Swing

A API Swing possui um conjunto de pacotes com componentes para a criação de telas interessantes ao usuário final. Abaixo segue a relação deles na API:

javax.accessibility	javax.swing.plaf	javax.swing.text
javax.swing	javax.swing.plaf.basic	javax.swing.text.html
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html.parser
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.rtf
javax.swing.event	javax.swing.plaf.synth	javax.swing.tree
javax.swing.filechooser	javax.swing.table	javax.swing.undo

Níveis de painéis

Antes de analisarmos alguns componentes, é necessário conhecer a hierarquia de painéis que uma interface de usuário em Java possui. Conhecer estes painéis pode ser desnecessário em um primeiro momento, mas em aplicações mais complexas que utilizam a manipulação de vários eventos diferentes, assim como, a inclusão de várias telas em uma janela, sempre é importante conhecer as características arquiteturais envolvidas.



Root Pane

Conforme a figura acima, podemos visualizar uma estrutura de painéis, que segue do *Frame* até o *Glass Pane*. Sobre os principais painéis temos que:

- *Frame*: É o painel de mais alto nível. Um *Frame* designa uma janela principal, com título e bordas, com uma dimensão e recebe os outros painéis e componentes.
- *Layered pane*: é um contêiner da qual pode posicionar componentes uns sobre os outros. Opcionalmente também fornece a inclusão na janela de um *menu bar* (Menu de ferramentas).
- *Content pane*: painel onde ficam todos os componentes visíveis na tela, fora o *menu bar*.
- *Glass pane*: é neste painel que eventos para modificar aspecto de componentes já existentes na tela é realizado. Pode, por exemplo, mostrar uma imagem em cima dos componentes do *content pane* ou utilizar o *glass pane* para interceptar quando um *mouse* passar por cima de uma lista de itens, desativando o evento clique do *mouse*.

Alguns componentes e como utiliza-los serão descritos a seguir. Recomenda-se a leitura das referências, principalmente dos tutoriais de Java da Oracle que detalham e mostram executáveis de exemplos dos vários componentes fornecidos pelo Swing.

JFrame

Para iniciar a construção de uma interface gráfica, crie uma classe e importe a API (javax.swing). Em seguida, faça com que a classe seja estenção da classe JFrame. A classe JFrame possui as funcionalidades para criar nossa janela principal que vamos chamar de *frame* mesmo:

```
1 public class TesteSwing extends JFrame {
2     JFrame frame = new JFrame()
3     frame.setTitle("Olá mundo");
4 }
```

No fragemtno acima, instanciamos o *frame* e colocamos na janela o título “Olá mundo”.

Dica Oracle

Para fazer uma janela ser dependente de outra, trazendo uma informação temporária como um erro ou notificação ao usuário, use caixas de diálogo (dialog[a]) ao invés de *frame*. Caso queira fazer uma janela aparecer dentro de outra use uma internal frame[b].

[a] http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html
[b] http://docs.oracle.com/javase/tutorial/uiswing/components/internalframe.html

Adicionando Componentes na Tela

Seguindo o exemplo, precisamos inserir alguns componentes no frame. Uma listagem completa de como utilizar todos os componentes forneceidos pelo Swing pode ser acessado em How to Use Various Components[3].

Continuando com o exemplo vamos inserir um *label* “etiqueta” com um botão e uma caixa de entrada de texto:

[3] http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html

```
JLabel etiqueta = new JLabel("Etiqueta de alguma coisa:");
JButton botao = new JButton("Botão");
JTextField campoTexto = new JTextField(20); //20 = tamanho do campo
```

Com os componentes instanciados, é necessário adicioná-los ao *frame*. Isto é feito, como vimos na seção anterior, em outro painel. Assim precisamos instanciar um *JPanel* e adicionar os componentes neste painel:

```
JPanel janela = new JPanel();
janela.add(etiqueta);
janela.add(botao);
janela.add(campoTexto);
```

Agora que nós adicionamos os componente a *JPanel* janela, precisamos adicionar essa camada ao frame:

```
frame.getContentPane().add(janela);
```

Visualizando a Janela

Agora com a nossa interface pronta, podemos ajustar o tamanho da janela e precisamos ativar sua visualização. Caso a visualização não seja ativada, após compilar a aplicação, nada será apresentado na tela.

```
frame.setSize(300, 400);
frame.setVisible(true);
```

Caso não queira ajustar o tamanho manualmente, mas queira que o *frame* se ajuste automaticamente conforme a disposição dos componentes no JPanel, use o método `frame.pack()`. A classe completa pode ser verificada abaixo, compile e veja como ficou.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class TesteSwing extends JFrame {

public TesteSwing2(){
    super();
}

public void criarJanela(){

    JFrame frame = new JFrame();
    frame.setTitle("olá mundo");
    JLabel etiqueta = new JLabel("Etiqueta de alguma coisa:");
```

```
18      JButton botao = new JButton("Botão");
19      JTextField campoTexto = new JTextField(20); //tamanho do campo pra 20 c\
20  aracteres
21      JPanel janela = new JPanel();
22      janela.add(etiqueta);
23      janela.add(botao);
24      janela.add(campoTexto);
25      frame.getContentPane().add(janela);
26      frame.setSize(300, 400);
27      frame.setVisible(true);
28  }
29
30
31  public static void main(String[] args) {
32
33      TesteSwing sistema = new TesteSwing();
34      sistema.criarJanela();
35
36      }
37  }
```

A classe TesteSwing acima precisa ser melhor organizada, discuta com seus colegas como poderiam organizar. O método `criarJanela()` deveria se preocupar somente com a inicialização da janela? Será que a declaração e a instanciação dos componentes no escopo da classe é uma boa solução? O que mais?

Manipulando eventos nos componentes

Para que se possa tratar uma ação executada pelo usuário, como o evento de clicar em um botão na tela, é necessário que a classe do nosso exemplo implemente ou estenda as funcionalidades da classe *ActionListener*. Esta classe possui métodos de escuta para os componentes, ou seja, métodos para que possamos reconhecer uma ação em algum componente e lidar com ela. Após a importação de *ActionListener*, precisamos adicionar uma escuta em um componente, para este exemplo, adicione logo após a criação do botão a seguinte codificação:

```
1  botao.addActionListener(this);
```

Agora é necessário criar o método `ActionPerformed()` na classe para que possamos manipular a ação do botão, desta forma temos o seguinte método:

```
public void actionPerformed(ActionEvent evento){

    Object acao = evento.getSource();

    if(acao == botao){
        campoTexto.setText("JAVA NOOB!");

    }
}
```

Para cada `evento`, será verificado qual a origem deste evento, se a ação `acao` for de um botão então será incluído um texto no componente `campoTexto`. Abaixo segue a classe completa.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class TesteSwing extends JFrame implements ActionListener{
    JFrame frame;
    JLabel etiqueta;
    JButton botao;
    JTextField campoTexto;
    JPanel janela;

    public TesteSwing2(){
        super();
   }

   public void criarJanela(){

        frame = new JFrame();
        frame.setTitle("olá mundo");
        etiqueta = new JLabel("Etiqueta de alguma coisa:");
        botao = new JButton("Botão");
        campoTexto = new JTextField(20); //tamanho do campo

        JPanel janela = new JPanel();
        janela.add(etiqueta);

        janela.add(botao);
        botao.addActionListener(this);

        janela.add(campoTexto);
```

```
35
36          frame.getContentPane().add(janela);
37          frame.setSize(300, 400);
38          frame.setVisible(true);
39      }
40
41      public void actionPerformed(ActionEvent evento){
42
43          Object acao = evento.getSource();
44
45          if(acao == botao){
46              campoTexto.setText("JAVA NOOB!");
47
48          }
49
50      }
51
52      public static void main(String[] args) {
53
54          TesteSwing sistema = new TesteSwing();
55          sistema.criarJanela();
56
57      }
58  }
```

Cuidado com as diferenças desta última classe TesteSwing com a anterior!

Existem vários outros componentes prontos para utilizarmos, siga para as referências no final da apostila e não deixe de conferir os links distribuídos pelo texto. Vimos desde os conceitos fundamentais de orientação a objeto até interfaces de usuário gráficas. A partir de agora solte sua imaginação para melhorar os exemplos e praticar sua programação. Não foi apresentado aqui padrões de projeto e nem testes unitários. Existe um mundo Java gigantesco lá fora. Bons estudos!

Referências

- Deitel, H. M. e Deitel, P. J. Java, Como Programar. Tradução de Carlos Arthur, 4 ed. Bookman, Porto Alegre, 2003.
- Eck, D. J. Introduction to Programming Using Java. Department of Mathematics and Computer Science, Hobart and William Smith Colleges, Geneva, NY. Disponível gratuitamente via web[4].
- Tutoriais Oracle para Orientação a Objeto em Java[5].
- Tutoriais Oracle para Componentes Swing[6].

[4] http://math.hws.edu/javanotes

[5] http://docs.oracle.com/javase/tutorial/java/index.html

[6] http://docs.oracle.com/javase/tutorial/uiswing/components/index.html