HELDER GUIMARÃES ARAGÃO

# JAVA E PROGRAMAÇÃO ORIENTADA A OBJETOS:
## UMA ABORDAGEM DIDÁTICA

1ª edição

# HELDER GUIMARÃES ARAGÃO

# JAVA E PROGRAMAÇÃO ORIENTADA A OBJETOS:

## UMA ABORDAGEM DIDÁTICA

1º edição

Salvador

Edição do Autor

2013

Capa: Estúdio9 Design

Revisora: Livia Guimarães Aragão Torres

Diagramação: Flávio Andrade

Java e Programação Orientada a Objetos: Uma Abordagem Didática

Autor: Helder Guimarães Aragão

E-ISBN: 978-85-613558-1-5




Estúdio9
Programação Visual

Rua Oswaldo Cruz, 166. Rio Vermelho. Salvador-Ba.
CEP 41.940-000. Tel: 71 3335.3172
www.estudio9design.com.br

# Sobre o Autor

Helder Guimarães Aragão é Mestre em Sistemas e Computação, Especialista em Componentes Distribuídos e Web e Bacharel em Ciência da Computação. Atualmente, é Gerente da Divisão de Geoprocessamento da EMBASA (Empresa Baiana de Águas e Saneamento), Professor Adjunto do Centro Universitário Estácio da Bahia (Estácio – FIB) e Consultor em Geotecnologias.

O Currículo Lattes do autor está disponível no endereço:

http://buscatextual.cnpq.br/buscatextual/visualizacv.do?id=K4736835J2.

Seu email é: helderaragao@gmail.com

# Agradecimentos

Este livro é um sonho traduzido em palavras. Escrever é uma forma silenciosa e eterna de compartilhar o conhecimento. Este sonho não poderia ser realizado sem o amor das pessoas citadas a seguir. Portanto, agradeço:

• A Deus por ter me dado forças, saúde e fé para a conclusão deste livro. Obrigado meu Deus!

• A todos da minha família e em especial: à minha Mãe, por ser uma verdadeira educadora e por valorizar a minha formação. Ao meu Pai, um típico imigrante digital, por ter me incentivado na área de Computação e ter sido meu primeiro usuário final. À minha irmã e Professora Lívia Aragão, pelo incentivo ao meu trabalho e por ter sido revisora deste livro. Ao meu irmão e Professor Helio Aragão, companheiro dessa fantástica área de Exatas, pela valorização constante ao meu trabalho. À minha sobrinha Ayane, pelos momentos de descontração e descobertas. À minha esposa Lara Aragão, por todo amor, cuidado, carinho, admiração e atenção. Estes sentimentos são fundamentais para a escrita de um livro.

• Aos meus grandes Professores que me deram inspiração para o estudo da Ciência da Computação. Igualmente, agradeço aos meus alunos, que se tornam professores no processo ensino-aprendizagem com suas perguntas e contribuições pertinentes. Finalmente, agradeço aos meus colegas de trabalho da EMBASA e Estácio - FIB pelo compartilhamento de conhecimento e experiência.

# Prefácio

Este livro foi originado a partir das minhas experiências com o ensino da Linguagem de Programação Java e Orientação a Objetos.

O livro **Java e Programação Orientada a Objetos: uma abordagem didática** visa facilitar o aprendizado da linguagem Java com os conceitos básicos da programação orientada a objetos.

A audiência deste livro é, portanto, estudantes de graduação dos cursos de computação e profissionais interessados em aprender a linguagem Java e os conceitos relacionados com a orientação a objetos. Este livro tenta abordar estes assuntos de uma forma didática. Todos os códigos fontes utilizados no livro podem ser encontrados e baixados no site: http://sites.google.com/site/helderaragao.

# 1. Introdução

A complexidade dos sistemas de informação motivou o surgimento de metodologias, que tornassem o processo de construção destes sistemas menos custoso e mais produtivo. Neste contexto, pode-se destacar a metodologia orientada a objetos que tem, basicamente, os seguintes objetivos:

- Aumentar a produtividade do desenvolvimento de software por meio do reuso de códigos existentes;
- Diminuir o custo da construção de software;
- Facilitar a manutenção do código fonte do software.

Embora tenha surgido na década de 60, a orientação a objetos começou a ser amplamente utilizada na década de 90. A partir de então, diversas linguagens de programação foram criadas visando suportar os conceitos relacionados com a programação orientada a objetos. Dentre estas linguagens, destaca-se a linguagem de programação Java.

A linguagem Java se popularizou bastante com a criação de um componente de software denominado de Máquina Virtual Java (*Java Virtual Machine - JVM*), que permite a execução de aplicações desenvolvidas em Java em várias plataformas computacionais. Além disso, a Máquina Virtual Java é gratuita, o que potencializou mais ainda a adoção da linguagem Java tanto no meio acadêmico como empresarial.

Os próximos capítulos deste livro serão dedicados à explicação dos conceitos relacionados com a orientação a objetos e a aplicabilidade destes conceitos utilizando Java. O Capítulo 2 explica a linguagem e a plataforma Java. O Capítulo 3 aborda os fundamentos da linguagem de programação Java. O Capítulo 4 descreve os conceitos da Programação Orientada a Objetos. Por fim, o Capítulo 5 faz as considerações finais.

# 2. A Linguagem e a Plataforma Java

O termo “Java” pode ser utilizado para fazer referência a uma linguagem de programação ou a uma plataforma computacional. Dentre as características da linguagem de programação Java, podem-se destacar:

- **Orientada a objetos:** em Java os conceitos da orientação a objetos podem ser implementados. Todas as variáveis e métodos devem estar associados a uma classe;
- **Sintaxe parecida com a Linguagem de Programação C++:** os comandos da linguagem Java possuem sintaxe semelhante com a linguagem C++;
- **Portável:** o código fonte de uma classe Java pode ser compilado e executado em qualquer computador que utilize um sistema operacional com a Máquina Virtual Java instalada. Esta característica, que será detalhada a seguir, permite que a linguagem Java seja independente de plataforma;
- **Gratuita:** a Máquina Virtual Java ou *Java Virtual Machine* (JVM) pode ser obtida sem custo no site da Oracle. Isto motivou a popularização e a disseminação da linguagem Java;
- **Conjunto de bibliotecas:** a linguagem Java possui uma *Application Programming Interface* (API), isto é, um conjunto de bibliotecas que auxilia as implementações de software. Existem muitas opções de APIs desenvolvidas por diversas comunidades de software livre.

Conforme dito anteriormente, o termo Java pode ser utilizado também para fazer referência a uma plataforma. A plataforma Java é composta pela Máquina Virtual Java, ou *Java Virtual Machine* (JVM), e a *Application Programming Interface* (API).

A plataforma Java pode ser executada em vários dispositivos ou sistemas operacionais. Para isso, a JVM deve estar disponível para o dispositivo ou sistema operacional em que se deseja executar o programa Java (Figura 1).

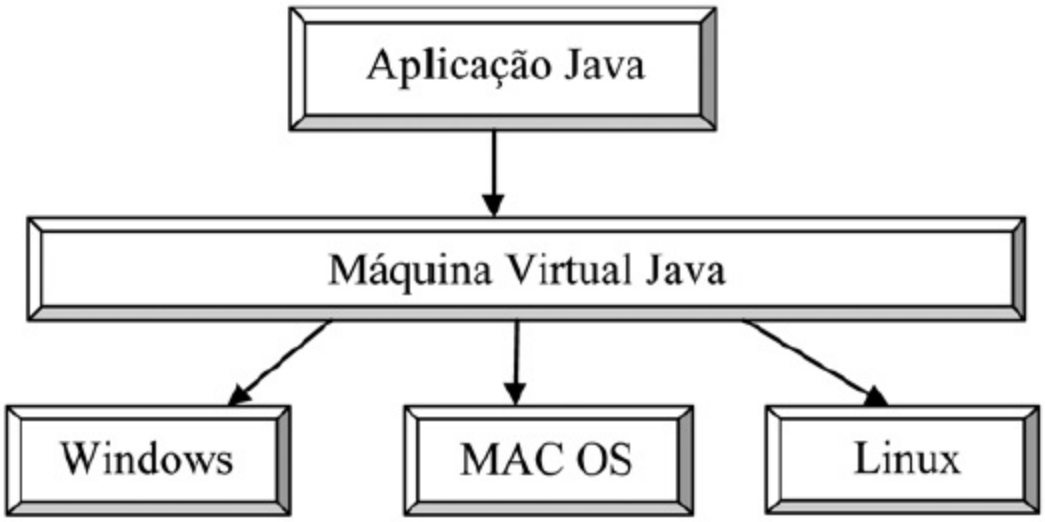


**Figura 1: Execução da aplicação Java em diversos Sistemas Operacionais.**

A plataforma Java pode ser dividida em três diferentes edições:

- *Java Standard Edition* (JSE): esta edição possui a JVM, ferramentas e APIs essenciais para o desenvolvimento de aplicações Java *Desktop*;
- *Java Enterprise Edition* (JEE): esta edição possui a JVM, ferramentas e APIs para o desenvolvimento de aplicações Web e distribuídas;
- *Java Micro Edition* (JME): esta edição possui a JVM, ferramentas e APIs para o desenvolvimento de aplicações para dispositivos móveis, tais como *smartphone* e telefone celular.

O foco deste livro é a plataforma JSE. Portanto, para executar todos os exemplos contidos no livro é necessário ter apenas esta plataforma instalada. A plataforma JSE pode ser baixada no site da Oracle.

As classes implementadas em Java podem ser divididas basicamente em três tipos:

- **Aplicação (*Application*):** refere-se a toda classe que é executada no computador local podendo ou não conter interfaces gráficas;
- **Representação de Modelos:** são as classes utilizadas para representar modelos ou abstrações de dados. Estas classes não podem ser executadas diretamente, mas apenas utilizadas por outras classes;
- **Conjunto de Rotinas:** são as classes que contêm um conjunto de rotinas. Ao invés de representar dados, estas classes funcionam como bibliotecas possuindo métodos que têm algo em comum.

A seguir, no Capítulo 3 deste livro, vamos trabalhar apenas com classes Aplicações (*Application*). As classes para representação de modelos e com implementação de rotinas serão utilizadas no Capítulo 4.

# 3. Fundamentos da Linguagem de Programação Java

## 3.1 Estrutura de um programa Java

Toda aplicação (programa) Java, independente do tipo, é representada por uma classe. A classe do tipo Aplicação é implementada em um arquivo texto com a extensão ".java". A Listagem de Código 3.1 mostra um exemplo de classe deste tipo.

```
//Primeiro Programa Java para exibir uma  mensagem no console

public class Exemplo1 {

	public static void main(String[] args) {

		System.out.println("Programa Java!");

	}

}
```

**Listagem de Código 3.1: Exemplo de uma classe Java Aplicação.**

A classe da Listagem de Código 3.1 quando executada exibe a simples mensagem "Programa Java!" na tela do computador (utilizando como saída o console). As seguintes características podem ser destacadas nesta classe:

- O nome da classe é **Exemplo1**;
- A classe é pública (comando **public**);
- Java é *case sensitive*, isto é, diferencia caracteres maiúsculos de minúsculos;
- As classes, métodos e blocos de código são delimitados por um **abrir** ( **{** ) e **fechar** ( **}** ) de chaves;

- Cada comando deve ser finalizado por um **ponto e vírgula** (**;**);
- Em Java, o comentário de código pode ser feito por // (comentário de uma linha) ou /* */ (comentário com mais de uma linha);
- Método **main**: Todo programa Java para ser executável na plataforma JSE deve possuir o método **main** declarado da seguinte forma: "**public static void main(String[] argumentos)**". Este método indica para a JVM que a classe é executável e, portanto, do tipo Aplicação. Nem toda classe possui este método. Neste capítulo, entretanto, vamos utilizar classes com apenas este método;
- "**System.out.println("Programa Java!");**": nesta linha é utilizada a classe **System** contida na API Java, que possui métodos para exibição de mensagens na tela do computador.

## 3.2 Palavras Reservadas, Tipos de Dados Primitivos e Operadores

A linguagem de programação Java possui um conjunto de palavras reservadas. Uma palavra reservada é aquela que possui um significado especial para o compilador. A lista abaixo possui algumas palavras reservadas da linguagem Java.

- abstract
- double
- if
- super
- boolean
- else
- implements
- this
- catch
- extends
- import
- throw
- char
- final
- int
- void
- class
- float
- interface
- try
- do
- for
- new
- while

A linguagem de programação Java dispõe de tipos primitivos de dados, isto é, tipos que são parte da própria linguagem e não são instâncias de outras

classes. Estes tipos são:

- **boolean:** representa valores lógicos ou booleanos. Uma variável deste tipo pode assumir o valor *true* ou *false*;
- **byte, short, int e long:** são os tipos numéricos que representam valores inteiros. Cada tipo deste utiliza uma determinada quantidade de memória e possui uma faixa de valores. Por exemplo, o **byte** ocupa apenas um byte na memória e pode representar valores entre -128 e 127, enquanto o **long** ocupa oito bytes na memória e pode representar valores entre –9223372036854775808 e 9223372036854775807.
- **float e double:** são os tipos numéricos que representam valores de ponto flutuante. A diferença entre eles está também na memória ocupada e na precisão (simples ou dupla, respectivamente);
- **char:** é o tipo utilizado para representar um caracter. Em Java um caracter deve estar sempre entre aspas simples (‘’).

O tipo **String**, que representa uma cadeia de caracteres, na linguagem Java não é um tipo primitivo. **String** é uma classe, embora o seu uso seja similar a campos de tipos nativos. O conceito de classe será detalhado no próximo capítulo.

Algumas operações básicas podem ser feitas com os tipos numéricos primitivos. Estas operações podem ser: soma (+), subtração (-), divisão (/), multiplicação (*) e extração do resto da divisão (operador %).

A linguagem Java também possui um conjunto de operadores, que podem ser utilizados para efetuar comparações entre valores numéricos. Os operadores são: menor (<), maior (>), menor igual (<=), maior igual (>=), dois iguais (==) e diferente (!=). Vale destacar ainda os operadores lógicos **E** (&&) e **OU** (||).

**Observações importantes:** o operador igual (=) em Java significa atribuição e não comparação. A comparação de **Strings** em Java é feita através do método **equals**.

## 3.3 Declaração de Variáveis

A declaração da variável em Java é da seguinte forma: o tipo do dado vem antes do nome da variável que será daquele tipo. Seguem alguns exemplos:

- String nome; //variável nome do tipo String;
- int idade; //variável idade do tipo inteiro;
- double dinheiroTotal; //variável dinheiroTotal do tipo double (valores reais);
- char sexo; //variável sexo do tipo caracter.

A Listagem de Código 3.2 mostra um exemplo utilizando três variáveis do tipo inteiro (**a**, **b** e **soma**).

```
//Programa Java mostrando declaração de variáveis

public class Exemplo2 {

	public static void main(String[] args) {

		int a = 0;

		int b = 2;

		int soma = a + b;

		System.out.println("a soma foi: "+soma);

	}

}
```

**Listagem de Código 3.2: Exemplo de classe Java utilizando variáveis.**

## 3.4 Entrada e Saída de Dados

A saída dos dados em Java pode ser feita através de uma classe chamada **System**. A Listagem de Código 3.3 mostra um exemplo de código para exibir mensagens no console (sem interface gráfica).

```
//Exemplo 3 mostrando um exemplo de saída em java

public class Exemplo3 {

	public static void main(String[] args) {

		System.out.print("Texto 1 ");

		System.out.println("Texto 2");



	}

}
```

**Listagem de Código 3.3: Exemplificando a saída de dados.**

No código da listagem 3.3, dois métodos para exibir mensagens podem ser destacados: **print** e **println**. O primeiro método exibe a mensagem e não salta linha. Já o segundo método exibirá a mensagem e dará um salto de linha.

A entrada de dados utilizando a linguagem Java pode ser feita de várias maneiras. Neste livro, vamos utilizar uma classe denominada **Keyboard**. Esta classe é bastante utilizada e facilmente encontrada na Web.

A classe **Keyboard** possui vários métodos para leitura dos tipos de dados primitivos e String. A Listagem de Código 3.4 mostra o uso da classe **Keyboard**.

```
public class KeyboardApp {

	public static void main (String[] args) {

		//exibindo uma mensagem

		System.out.println("digite o seu nome");

		//lendo uma variável String com Keyboard

		String nome = Keyboard.readString();

		System.out.println("digite a sua idade");

		int idade = Keyboard.readInt();

		System.out.println("o seu nome é "+nome +" a sua idade é "+idade);

	}

}
```

**Listagem de Código 3.4: Exemplificando a entrada de dados em Java.**

# 3.5 Comandos de Seleção e Repetição

Um dos comandos de seleção da linguagem Java é o **if**. Este comando permite analisar uma expressão lógica e direcionar a execução do programa. Caso a expressão lógica seja verdadeira (*true*), a sequência do **if** será executada. Se a expressão lógica for falsa (*false*) e existir a cláusula **else**, a sequência de comandos dentro do **else** será executada. A Listagem de Código 3.5 mostra um exemplo de uso do comando **if**.

```
public class IFApp {

    public static void main(String[] args) {

        final int mes = 4;

        String nome = "Helder";

        //if sem else

        if (mes == 4) {

            System.out.println("o mês é 4");

        }

        //if com else

        if (nome.equals("Helder") ){

            System.out.println("Nome correto");

        }

        else {

            System.out.println("Nome incorreto");

         }
```

```
        }

    }
```

**Listagem de Código 3.5: Exemplificando o uso do if.**

Existe também em Java o comando de seleção **switch**. Este comando compara o valor de uma determinada variável a uma relação de valores. Comandos podem ser associados a cada valor.

A linguagem de programação Java possui estruturas de controle que permitem repetir um conjunto de instruções. Estes comandos, conhecidos como de repetição, são:

- **while:** este comando executa uma sequência de comandos enquanto a expressão lógica for verdadeira. Neste comando, a expressão lógica é analisada antes de cada execução (Listagem de Código 3.6);
- **do while:** este comando executa uma sequência de comandos enquanto a expressão lógica for verdadeira. Após a execução da sequência, a expressão lógica é analisada. Neste caso, pelo menos uma vez a sequência de comandos será executada (Listagem de Código 3.7);
- **for:** este comando executa uma sequência de comandos enquanto a expressão lógica for verdadeira. Para o uso do comando **for** é necessária a declaração de um contador do tipo inteiro que definirá o fim da execução da repetição (Listagem de Código 3.8).

```
public class WhileApp {

    public static void main( String[] args ) {

        int i =1;

        while (i <=20  ) {

            System.out.println(i);

            i++;
```

```
        }
    }
}
```

**Listagem de Código 3.6: Programa exemplificando o uso do comando while.**

```
public class DoWhileApp{
    int i=1;
    do {
        System.out.println(i);
        i++;
    } while(i!=21);
}
```

**Listagem de Código 3.7: Programa exemplificando o uso do comando do while.**

```
public class ForApp {
    public static void main(String[] args) {
        //comando for e o uso do contador i
        for (int i = 1; i <= 20; i++ ) {
            System.out.println(i);
        }
```

```
        }

}
```

**Listagem de Código 3.8: Programa exemplificando o uso do comando for.**

## 3.6 Vetores (*arrays*)

Vetor é uma estrutura homogênea e estática. Esta estrutura permite guardar múltiplos valores de um mesmo tipo. Cada um dos valores contidos no vetor pode ser acessado através de um índice. O uso do vetor (array) é indicado quando se deseja representar diversos valores do mesmo tipo usando apenas uma variável. A Listagem de Código 3.9 mostra um exemplo de vetor para armazenar notas de alunos.

```
public class ArrayNotasApp {

	public static void main(String[] args) {

		double[] notas = new double[4];

		notas[0] = 2.6;

		notas[1] = 10.0;

		notas[2] = 9.5;

		notas[3] = 10.0;

		System.out.println("exibindo notas");

		for (int i=0; i < notas.length; i++){

			System.out.println(notas[i]);

		}

		}

}
```

**Listagem de Código 3.9: Programa exemplificando o uso de array.**

Observe no código da Listagem de Código 3.9 que o vetor de notas foi definido antes da sua utilização. No momento desta definição, através da notação de colchetes e do comando **new**, deve-se informar o tamanho do vetor, ou seja, quantos dados daquele mesmo tipo o vetor poderá armazenar.

No exemplo do código da Listagem de Código 3.9, o vetor **notas** tem o tamanho definido com 4 (quatro) posições. Em Java, o índice inicial do vetor é sempre 0 (zero). Portanto, os índices do vetor, neste exemplo, variam de 0 a 3, conforme pode ser visto no momento do uso do comando **for** para a exibição das notas.

O exemplo tratado aqui mostrou o vetor com uma dimensão (arrays unidimensionais). Entretanto, em Java, pode-se construir o vetor multidimensional para representação de matrizes. Para isto a definição seria, por exemplo, desta forma: int[][] matriz = new int[4][5].

# 4. Programação Orientada a Objetos

A Programação Orientada a Objetos é um paradigma de desenvolvimento de software, que visa modelar os problemas do mundo real utilizando, basicamente, os conceitos de Classe e Objeto. A Orientação a Objetos surgiu na década de 60, embora a sua popularização tenha acontecido na década de 90. Este paradigma tem os seguintes objetivos:

- Promover a reutilização de código, aumentando a produtividade dos desenvolvedores;
- Facilitar a manutenção do software;
- Organizar e tornar mais legível o código fonte do software.

Os principais conceitos da orientação a objetos estão descritos nas próximas seções deste capítulo.

# 4.1 Classes e Objetos

Uma Classe pode ser definida como a implementação de um tipo abstrato de dado (modelo) formado por propriedades (atributos) e operações (métodos). Podemos definir, por exemplo, uma classe **Notebook** composta pelos seguintes atributos: **codigo** e **tamanho**. Suas operações poderiam ser: **ligar** e **desligar**. A Listagem de Código 4.1 mostra a implementação da classe **Notebook** em Java

```
/*Classe representando um modelo para armazenar dados de um notebook*/

public class Notebook {

    int codigo;

    int tamanho;

    public Notebook(int codigo, int tamanho) {

        this.codigo=codigo;

        this.tamanho=tamanho;

    }


    public void ligar(){

        System.out.println("ligando o notebook");

    }


    public void desligar(){

        System.out.println("desligando o notebook");
```

```
    }

}
```

**Listagem de Código 4.1: Implementação da classe Notebook.**

Pode-se observar que os atributos são os dados que serão armazenados daquela classe (**codigo** e **tamanho**). Os métodos são as operações ou as ações fornecidas por esta classe. Os procedimentos e funções presentes em linguagens estruturadas como C e Pascal, por exemplo, são chamados no paradigma orientado a objetos de métodos de uma classe (void **ligar()** e void **desligar()**).

O conceito de Objeto está relacionado com a definição de classe. Um objeto pode ser considerado uma instância da classe, que possui valores próprios para os atributos definidos. Exemplos de objetos do tipo **Notebook** são mostrados na classe **NotebookApp** da Listagem de Código 4.2.

```
//Classe notebook com dois objetos (note1 e note2)

public class NotebookApp {

    public static void main(String[] args){

    //instanciando o objeto 1

    Notebook note1 = new Notebook(1,14);

    //instanciando o objeto 2

    Notebook note2 = new Notebook(2,17);

      }

}
```

**Listagem de código 4.2: Aplicação instanciando dois objetos.**

Percebe-se no código da listagem 4.2 que o objeto deve ser representado por uma variável que é do tipo da classe. Neste caso, os objetos são **note1** e **note2**. Para instanciar o objeto, a palavra reservada **new** deve ser utilizada. Este comando invoca (chama) um método especial denominado construtor, explicado na próxima seção.

**Observações importantes:** Por convenção, todo nome de classe se inicia com letra maiúscula. Os nomes das variáveis que irão representar os objetos devem iniciar com letra minúscula.

# 4.2 Métodos

Todo Método em Java é definido da seguinte forma: ***visibilidade retorno nome(argumentos)***. No caso da classe **Notebook** da Listagem de Código 4.1, o método chamado **ligar()**, por exemplo, tem a visibilidade **public**, o retorno **void** (nenhum tipo de retorno) e não recebe nenhum argumento (parâmetro). Considere o exemplo deste método: **public float somar(float numero1, float numero2)**. Neste caso, a visibilidade é **public**, o retorno é **float**, o nome do método é **somar** e os seus argumentos são **numero1** e **numero2** do tipo **float**.

Existe um método especial na linguagem de programação Java denominado **Construtor**. Este método permite instanciar ou construir objetos podendo atribuir valores para os atributos. No exemplo da classe **NotebookApp** da Listagem de Código 4.2, o construtor é invocado com o comando **new**. O método construtor, obrigatoriamente, não tem retorno (nem mesmo **void**) e possui o mesmo nome da classe.

A Listagem de Código 4.3 mostra duas classes. A classe **Pessoa** possui o método construtor que recebe os parâmetros (int codigo, String nome) para inicializar os atributos. Na classe **PessoaApp**, dois objetos estão sendo instanciados com os seus valores. Na classe **Pessoa**, observa-se a presença do comando **this**. Este comando faz referência à própria classe.

```
public class Pessoa {

    /*atributos*/

    private int codigo;

    private String nome;

    /*método construtor*/

    public Pessoa(int codigo, String nome) {

        this.codigo=codigo;
```

```
            this.nome=nome;
        }
}


public class PessoaApp {
        public static void main(String[] args){
            //instanciando o objeto 1
            Pessoa pessoa1 = new Pessoa(1,"Helder");
            //instanciando o objeto 2
            Pessoa pessoa2 = new Pessoa(2,"Ana");
          }
}
```

**Listagem de código 4.3: Classes Pessoa e PessoaApp.**

Na classe **Pessoa**, utiliza-se o comando **this** para diferenciar os atributos da classe dos argumentos do método construtor. Nesta classe, o método construtor declara dois argumentos (codigo e nome), que possuem o mesmo nome dos atributos **codigo** e **nome** da classe. O método construtor difere os argumentos do método dos atributos da classe por causa do comando **this**.

# 4.3 Encapsulamento e Mensagens

Segundo as boas práticas da Programação Orientada a Objetos, todo atributo de uma classe deve ser manipulado somente através de métodos acessadores (**get**) ou modificadores (**set**). Este é o conceito de Encapsulamento, que visa garantir uma maior segurança aos atributos da classe. O encapsulamento impede que os dados da classe sejam manipulados diretamente.

Para encapsular os dados, os modificadores de acesso **public**, **protected** ou **private** devem ser utilizados. Estes modificadores podem ser usados também para os métodos definidos em uma classe. O desenvolvedor pode implementar um método interno com o modificador **private**, por exemplo, que só será utilizado dentro da própria classe.

As diferenças entre os modificadores **public**, **protected** e **private** estão explicadas a seguir:

- **public:** atributos ou métodos declarados como **public** em uma classe podem ser acessados pela própria classe, por classes derivadas desta e por outra classe que esteja em qualquer pacote do projeto. Este modificador é o menos restritivo;
- **protected:** atributos ou métodos definidos como **protected** são acessíveis pela própria classe e pelas classes derivadas;
- **private:** atributos ou métodos declarados como **private** só podem ser acessados pela própria classe. É o modificador mais restritivo de todos.

Quando o modificador não é definido, os atributos e métodos podem ser acessados pela própria classe, por classes derivadas e por qualquer outra classe dentro do mesmo pacote. Os conceitos de herança (classes derivadas) e pacote serão vistos com mais detalhes nas Seções 4.5 e 4.7 deste Capítulo.

Considere a classe **Login** mostrada na Listagem de Código 4.4. Esta classe representa o login de usuário com os atributos **nomeUsuario** e **senha**. Perceba que esta classe, além do construtor, possui os métodos acessadores (**get**) e modificadores (**set**). Observe que os atributos da classe estão

definidos como **private** e os métodos como **public**. Portanto, a classe **EncapsulamentoApp**, da Listagem de Código 4.5, não consegue acessar diretamente os atributos da classe **Login**. Este acesso pode ser feito apenas através dos métodos **get** e **set**. A chamada de um método por um objeto é denominada de Mensagem.

```
public class Login {

    /*atributos*/

    private String nomeUsuario;

    private int senha;

    /*método construtor*/

    public Login(String nomeUsuario,int senha) {

        this.nomeUsuario=nomeUsuario;

        this.senha=senha;

    }

    //métodos getters e setters

    public String getNomeUsuario(){

        return nomeUsuario;

    }

    public void setNomeUsuario(String nomeUsuario){

        this.nomeUsuario =nomeUsuario;

    }

    public int getSenha(){
```

```
        return senha;
    }
    public void setSenha(int senha){
        this.senha=senha;
    }
}
```

**Listagem de código 4.4: Classe Login com métodos getters e setters.**

```
public class EncapsulamentoApp {
    public static void main(String[] args){
        //instanciando o objeto 1
        Login login1 = new Login("helder",123456);
        //instanciando o objeto 2
        Login login2 = new Login("adm",654321);
        login1.setSenha(789456);
        System.out.println("login "+login1.getNomeUsuario());
    }
}
```

**Listagem de código 4.5: Classe EncapsulamentoApp utilizando a classe Login.**

# 4.4 Assinatura de Método e Sobrecarga

A Assinatura do Método é composta pelo nome do método mais os seus argumentos (tipos dos parâmetros que o método pode receber). Em Java é possível implementar métodos com o mesmo nome, mas assinaturas diferentes. Este é o conceito de Sobrecarga de Método.

Considere a classe **ContaCorrente** mostrada na Listagem de Código 4.6. Imagine que existam duas formas para abrir uma conta corrente em uma determinada agência bancária:

1. Informando os números da conta e da agência e o valor do limite do cheque especial, no caso de contas especiais;
2. Informando apenas os números da conta e da agência, no caso de conta comum.

Para resolver este caso, pode-se implementar dois métodos construtores com o mesmo nome, mas argumentos distintos. O método construtor da classe **ContaCorrente**, por exemplo, está sobrecarregado.

```
public class ContaCorrente {

    /*atributos*/

    private int agencia;

    private int numero;

    private float limite;

    /*método construtor versao 1*/

    public ContaCorrente(int agencia, int numero, float limite)

    {

        this.agencia=agencia;
```

```
        this.numero=numero;
        this.limite=limite;
}
/*método construtor versao 2*/
public ContaCorrente(int agencia, int numero)
{
        this.agencia=agencia;
        this.numero=numero;
}

//métodos getters e setters
public int getAgencia(){
        return agencia;
}
public void setAgencia(int agencia){
        this.agencia=agencia;
}
public int getNumero(){
        return numero;
}
```

```
    public void setNumero(int numero){
        this.numero=numero;
    }
    public float getLimite(){
        return limite;
    }
    public void setLimite(float limite){
        this.limite =limite;
    }

}
```

**Listagem de código 4.6: Classe ContaCorrente com o conceito de Sobrecarga.**

Perceba que é muito útil na prática o conceito de sobrecarga, pois permite flexibilizar o código. Os argumentos (lista de parâmetros do método) podem ser diferentes em quantidade, tipo ou ordem. O retorno do método não faz parte da assinatura.

# 4.5 Herança

Herança é a capacidade que uma classe tem de herdar atributos e métodos de uma classe já definida chamada de superclasse ou classe pai. Isto permite o reaproveitamento de código, pois o desenvolvedor reutiliza os artibutos e métodos implementados em uma classe que funciona como modelo.

Considere a seguinte situação: precisamos modelar classes que representem clientes de uma determinada empresa. Clientes podem ser de dois tipos: Pessoa Física ou Pessoa Jurídica. Todo cliente possui nome e endereço. Entretanto, cada tipo de cliente possui um dado específico que o identifica: *i)* CPF no caso de cliente Pessoa Física; *ii)* CNPJ no caso de cliente Pessoa Jurídica (Figura 2).

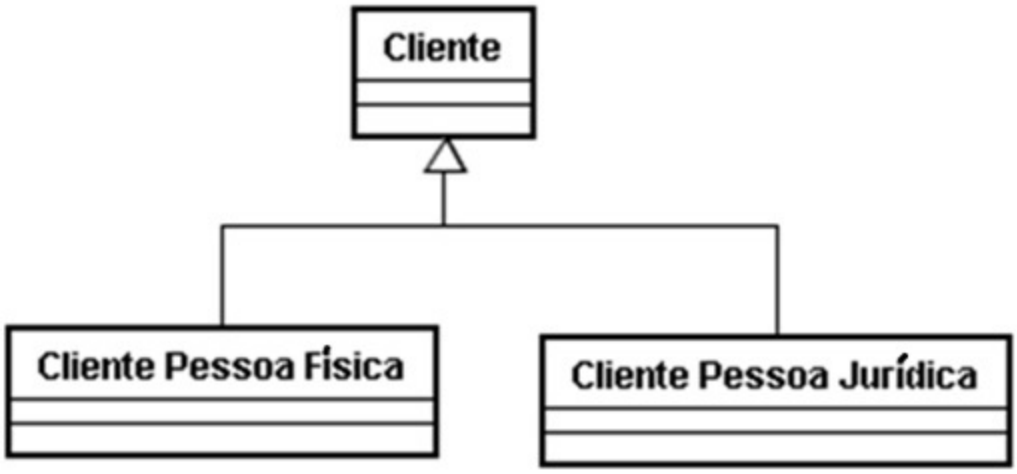


**Figura 2: Classe pai Cliente e subclasses ClientePessoaFisica e ClientePessoaJuridica.**

Uma possível implementação para este caso seria:

- Definir uma classe pai chamada **Cliente**, com os atributos comuns (**nome** e **endereco**) que serão reaproveitados;
- Definir duas subclasses, cada uma representando um tipo de cliente, herdando os atributos e métodos da classe pai (**Cliente**).

A implementação do modelo pode ser vista na Listagem de Código 4.7.

```
/* Definição da classe */
public class Cliente {
	/* Atributos da classe */
	private String nome; //Nome do cliente
	private String endereco; //Endereco do cliente
	/* Construtor */
	public Cliente (String nome, String endereco) {
		this.nome=nome;
		this.endereco=endereco;
	 }
	 public void setNome(String nome) {
		this.nome = nome;
	 }
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}
	public String getNome () {
		return nome;
	}
```

```
	public String getEndereco() {
		return endereco;
	 }
	/* Método que retorna os atributos do cliente */
	public String toString(){
		return(this.nome+ " " + this.endereco);
	}
}
/* SubClasse de Cliente que modela um Cliente Pessoa Juridica */
public class ClientePJ extends Cliente {
	private String cnpj;
	public ClientePJ (String nome, String endereco, String cnpj)
	{
		super(nome,endereco);
		this.cnpj=cnpj;
	}

	 public void setCnpj(String cnpj) {
		this.cnpj = cnpj;
	 }
```

```
    public String getCnpj () {
        return cnpj;
    }
    public String toString () {
        return("Cliente Pessoa Juridica : " + super.toString() + " " +
        this.cnpj);
    }
}

/**
 * SubClasse de Cliente que modela um Cliente Pessoa Fisica
 */
public class ClientePF extends Cliente {
    private String cpf;
    public ClientePF(String nome, String endereco, String cpf)
    {
        super(nome,endereco);
        this.cpf=cpf;
     }
    public void setCpf(String cpf) {
```

```
        this.cpf = cpf;
    }
    public String getCpf() {
        return cpf;
     }
    public String toString () {
        return("Cliente Pessoa Juridica : " + super.toString() +    " " +
        this.cpf);
     }
}


/**
 * Aplicação para demonstrar o uso de Herança
 */
public class ClienteApp {
    public static void main(String[] args) {
        //Declarando uma variável cliente pessoa juridica
        ClientePJ clientePJ;
        //Declarando uma variável cliente pessoa fisica
        ClientePF clientePF;
        //Instanciando um objeto cliente pessoa fisica
```

```
            clientePF = new ClientePF("Jose","rua xxxx", "111.999.999");

            //Instanciando um objeto cliente pessoa juridica

            clientePJ = new ClientePJ("Faculdade X","rua xyz",
            "99.999.999.9999");

            //Imprimindo os dados na tela

            System.out.println(clientePJ.toString());

            System.out.println(clientePF.toString());



        }

}
```

**Listagem de código 4.7: Implementações da classe pai e subclasses.**

Observe que as classes **ClientePF** e **ClientePJ** não necessitam implementar os atributos e os métodos *getters* e *setters* existentes na classe pai **Cliente**. Estes atributos e métodos são herdados. As subclasses implementam apenas os atributos e métodos específicos. Para invocar a classe pai, as subclasses devem utilizar a palavra reservada **super**.

**Observação importante:** Na linguagem Java, uma classe não pode herdar diretamente de duas superclasses.

## 4.5.1 Classes Abstratas e Métodos Abstratos

Classes abstratas são classes em Java que não podem ser instanciadas. Estas classes servem apenas como modelo e devem ser declaradas através da palavra reservada **abstract**.

No caso das classes implementadas na Seção 4.5, por exemplo, seria interessante que a classe pai **Cliente** fosse abstrata. Neste caso, só será

permitido instanciar objetos com tipos das subclasses (**ClientePF** e **ClientePJ**). Desta forma, instâncias com dados incompletos na classe aplicação são evitadas. A Listagem de Código 4.8 mostra a classe **Cliente** com o **abstract**. Observe que esta classe passa a ser abstrata e instâncias do tipo dela não podem ser criadas.

```
/* Definição da classe */

public abstract class Cliente {

    /* Atributos da classe */

    private String nome; //Nome do cliente

    private String endereco; //Endereco do cliente

    /* Construtor */

    public Cliente (String nome, String endereco) {

        this.nome=nome;

        this.endereco=endereco;

    }

    public void setNome(String nome) {

        this.nome = nome;

    }

    public void setEndereco(String endereco) {

        this.endereco = endereco;

    }

    public String getNome () {
```

```
        return nome;
    }
    public String getEndereco() {
        return endereco;
    }

    /* Método que retorna os atributos do cliente */
    public String toString(){
        return(this.nome+ " " + this.endereco);
    }
}
```

**Listagem de código 4.8: Implementação da Classe pai abstrata.**

Em Java, é possível implementar também Métodos abstratos. Um método abstrato é aquele que não possui implementação, mas apenas a sua definição. Isto é muito útil quando se sabe da existência de um método e a sua implementação é ainda desconhecida.

Imagine o caso do método **calcularDesconto()** que irá calcular o desconto de cada cliente. Todos os clientes ganham desconto independente do tipo de cliente. Os valores dos descontos, no entanto, são diferentes e dependem do tipo de cliente. Neste caso, seria interessante definir um método abstrato para o cálculo do desconto do cliente na classe pai (Cliente) e deixar a implementação deste método para as subclasses. Isto pode ser visto na Listagem de Código 4.9.

```
/* Definição da classe */
```

```
public abstract class Cliente {
	/* Atributos da classe */
	private String nome; //Nome do cliente
	private String endereco; //Endereco do cliente

	/* Construtor */
	public Cliente (String nome, String endereco) {
		this.nome=nome;
		this.endereco=endereco;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}
	public String getNome () {
		return nome;
	}
```

```
	public String getEndereco() {
		return endereco;
	}
	/*Método abstrato sem implementação*/
	public abstract int calcularDesconto();
	/* Método que retorna os atributos do cliente */
	public String toString(){
		return(this.nome+ " " + this.endereco);
	 }
}

/**
 * SubClasse de Cliente que modela um Cliente Pessoa Juridica
 */
public class ClientePJ extends Cliente {

	private String cnpj;
	public ClientePJ (String nome, String endereco, String cnpj)
	{
		super(nome,endereco);
```

```
        this.cnpj=cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

    public String getCnpj () {
        return cnpj;
    }

    public int calcularDesconto(){
        return 10;
    }

    public String toString () {
        return("Cliente Pessoa Juridica : " + super.toString() +   " " +
        this.cnpj);
    }
}
```

```
/**
 * SubClasse de Cliente que modela um Cliente Pessoa Fisica
 */
public class ClientePF extends Cliente {
        private String cpf;
       public ClientePF(String nome, String endereco, String cpf)
       {
              super(nome,endereco);
              this.cpf=cpf;
        }
       public void setCpf(String cpf) {
              this.cpf = cpf;
        }
        public String getCpf() {
              return cpf;
        }
        public int calcularDesconto(){
              return 5;
        }
```

```
    public String toString () {
        return("Cliente Pessoa Juridica : " + super.toString() +  " " +
        this.cpf);
    }
}

/**
 * Aplicação para demonstrar o uso de Herança
 */
public class ClienteApp {
    public static void main(String[] args) {
        //Declarando uma variável cliente pessoa juridica
        ClientePJ clientePJ;
        //Declarando uma variável cliente pessoa fisica
        ClientePF clientePF;
        //Instanciando um objeto cliente pessoa fisica
        clientePF = new ClientePF("Jose","rua xxxx", "111.999.999");

        //Instanciando um objeto cliente pessoa juridica
        clientePJ = new ClientePJ("Faculdade X","rua xyz",
        "99.999.999.9999");
```

```
        //Imprimindo na tela os dados
        System.out.println(cliente PJ.calcularDesconto());
        System.out.println(clientePF.calcularDesconto());

    }
}
```

**Listagem de código 4.9: Definindo um método abstrato na classe pai que foi implementado nas subclasses.**

**Observação importante:** A definição de um método abstrato na classe pai (superclasse) impõe a implementação deste método em suas subclasses.

# 4.6 Polimorfismo

Um exemplo de implementação do conceito de Polimorfismo é o método **toString()** das classes da Listagem de Código 4.9 da seção anterior. Este método é responsável por retornar todos os dados dos atributos do objeto. Observe que as assinaturas dos métodos são iguais (**toString()**) nas classes **Cliente**, **ClientePF** e **ClientePJ**. Entretanto, as implementações destes métodos são diferentes. O Polimorfismo caracteriza-se pela criação de métodos com assinaturas iguais, mas implementações distintas. Desta forma, o resultado executado depende do objeto que chamou o método.

Vale destacar que o polimorfismo permite a implementação de soluções legíveis. Este conceito evita a necessidade de criação de métodos com nomes diferentes, que conceitualmente e funcionalmente têm o mesmo objetivo.

# 4.7 Pacotes

Pacote em Java é uma forma lógica de organizar classes que possuem funcionalidades semelhantes. A API da linguagem de programação Java é organizada desta forma. Os pacotes java.awt e javax.swing da API, por exemplo, contêm as classes necessárias para a construção de programas com interfaces gráficas (janelas, botões, etc.). Um pacote é muito útil na prática para evitar nomes duplicados de classes e facilitar a localização destas quando um projeto é complexo.

Para importar classes de um determinado pacote, deve-se usar o comando **import**. Para definir que uma classe pertence a um determinado pacote, utiliza-se a palavra reservada **package**.

A Listagem de Código 4.10 mostra um exemplo de definição de pacote e a importação de classes do pacote javax.swing. O uso do '*' na importação significa que todas as classes daquele pacote serão importadas. Pode-se importar, também, classe por classe.

```
/*definindo o nome do pacote que a classe pertence*/

package calculo;

/*comandos imports para a importação de classes*/

import javax.swing.*;

//importando a classe Color do pacote java.awt

import java.awt.Color;



public class CalculoApp{

    public static  void main(String[] argumentos){
```

```
        float a;

        float b;

        float resultadosoma;

        float resultadomulti;

        a=2;

        b=4;

        resultadosoma = a+b;

        resultadomulti = a*b;

        System.out.println(resultadosoma);

        System.out.println(resultadomulti);

        JOptionPane.showMessageDialog(null,"resultados                    "
        +resultadosoma +" "+resultadomulti);

    }

}
```

**Listagem de código 4.10: Exemplificando o uso de pacotes.**

# 4.8 Tratamento de Exceção

Erros podem ocorrer na execução do programa e não devem ser ignorados pelo desenvolvedor. Ao acontecer um determinado erro, deve-se tratá-lo de alguma forma.

A linguagem de programação Java permite que o desenvolvedor faça o tratamento dos erros através dos comandos **try - catch**. Dentro do bloco do comando **try**, devem ficar todos os comandos que poderão ser executados e são passíveis de erro. Já o bloco do comando **catch** deve conter todos os comandos que só serão executados em caso de algum erro. Existe, ainda, o bloco do comando **finally** que será executado independente se houve erro ou não. A Listagem de Código 4.11 mostra o tratamento de exceção em uma classe Java.

```
public class ExemploExcecaoApp {

    public static void main(String args[])

    {

        try

        {

            int a[] = new int[2];

            int b=0;

            int c=2;

            int resultado = c/b;

        }

        catch (ArrayIndexOutOfBoundsException e)
```

```
            {
                // tratamento da exceção imprime a mensagem
                System.out.println("índice fora de faixa");
            }
            catch (ArithmeticException e)
            {
                // tratamento da exceção imprime a mensagem
                System.out.println("erro de divisão por zero");
            }
            finally
            {
                System.out.println("O finally sempre executa");
            }
        }
}
```

**Listagem de Código 4.11: Uso do try catch.**

Observe que cada bloco **catch** trata um tipo de erro. Portanto, o desenvolvedor tem que prever todos os possíveis erros do seu código e tratá-los. Neste caso, dois erros podem ocorrer: um relacionado com o índice do *array* fora de faixa e o outro associado com alguma operação matemática inválida (por exemplo, divisão por zero).

# 4.9 Interface

O objetivo de uma Interface é definir um modelo de implementação para classes. A interface possui as seguintes características:

- Não pode ser instanciada;
- Todos os métodos são implicitamente **abstract** e **public** e não podem conter implementação;
- Não tem construtor;
- Só pode conter constantes estáticas.

Para uma classe implementar uma interface, deve-se utilizar o comando **implements**. Quando uma determinada classe implementa uma interface, ela é obrigada a implementar todos os métodos definidos na interface. A Listagem de Código 4.12 mostra a implementação e o uso de uma interface denominada **SituacaoAcademica**.

```
/**

 * Interface modelando as operações de verificação da situação Acadêmica de alunos

*/

public interface SituacaoAcademica {

	public abstract String obterSituacaoAcademica();

	public abstract float obterMedia();

}


/*Aluno implementa a interface SituacaoAcademica */
```

```
public class Aluno implements SituacaoAcademica
{

    private String codigo;
    private String situacaoFinal;
    private float media;
    private int faltas;

    /* Construtor */
    public Aluno (String codigo) {
        setCodigo ( codigo );
    }
    /* Métodos para leitura e escrita dos atributos */
    public void setCodigo (String codigo) {
        this.codigo = codigo;
    }
    public String getCodigo () {
        return codigo;
    }
```

```
    public float obterMedia () {
        this.media = 5;
        return media;
    }


    public String obterSituacaoAcademica() {
        if (this.media >7) {
            this.situacaoFinal = "aprovado";
        }
        else {
            this.situacaoFinal = "reprovado";
        }
        return situacaoFinal;
    }

}
```

**Listagem de código 4.12: Implementação de uma interface e o seu uso na classe Aluno.**

O uso de uma interface é interessante quando se deseja padronizar nomes de métodos ou garantir que alguns métodos sejam de fato implementados como foram definidos na interface. A classe **Aluno** da Listagem de Código 4.12 não iria compilar caso não tivesse implementado os métodos **public abstract String obterSituacaoAcademica()** e **public abstract float obterMedia()** como definidos na interface **SituacaoAcademica**.

**Observação:** Uma classe pode implementar mais de uma interface.

# 4.10 Coleções

Em Java, podem-se utilizar classes que representam uma coleção de objetos. Dentre estas classes, destacam-se as listas de objetos. Existe uma interface chamada **List**, que faz a declaração dos métodos utilizados para a manipulação de listas como, por exemplo, o método **add**. Este método permite a inserção de objetos no fim da lista. Além deste método, é importante destacar o método **size** que retorna o tamanho da lista.

Duas classes implementam a interface **List**: **LinkedList** e **ArrayList**. Neste livro, irei focar apenas nesta última classe por ser bastante utilizada como estrutura auxiliar em consultas.

A classe **ArrayList** tem as seguintes características:

- O tamanho é dinâmico, isto é, pode aumentar ou diminuir à medida que objetos são adicionados ou removidos da lista;
- Armazena tipos complexos (objetos).

A Listagem de Código 4.13 mostra a implementação de duas classes com a utilização do **ArrayList**. Na classe **PessoasApp** tem uma instância chamada **pessoas**. Este objeto é do tipo **ArrayList**. Neste **ArrayList**, todos os objetos do tipo **Pessoa** são adicionados através do método **add(Object object)**.

```
public class Pessoa{

    private int idade;

    private String nome;

    public Pessoa(int idade, String nome){

        this.idade=idade;

        this.nome=nome;
```

```
    }
    public int getIdade(){
        return idade;
    }
    public String getNome(){
        return nome;
    }
    public void setNome(String nome){
        this.nome=nome;
    }
    public void setIdade(int idade){
        this.idade=idade;
    }
}

import java.util.ArrayList;
public class PessoasApp{
    public static void main(String[] args){
        ArrayList pessoas = new ArrayList();
        Pessoa p1 = new Pessoa(20,"Helder");
```

```
            Pessoa p2 = new Pessoa(25,"Joana");
            pessoas.add(p1);
            pessoas.add(p2);
        }
}
```

**Listagem de código 4.13: Implementação das classes Pessoa e PessoasApp.**

# 4.11 SWING (Interfaces Gráficas)

Inicialmente, a Sun disponibilizou a biblioteca de classes AWT (***Abstract Window Toolkit***) para o desenvolvimento de interfaces gráficas com Java. Em função de algumas incompatibilidades e *bugs* desta biblioteca, a Sun desenvolveu a biblioteca SWING. Algumas características da AWT foram mantidas e novas funcionalidades foram desenvolvidas na biblioteca SWING.

A biblioteca SWING utiliza os conceitos da orientação a objetos vistos neste livro. Isto facilita bastante o uso e o aprendizado desta biblioteca.

O intuito desta seção é explicar o uso de alguns componentes gráficos, a saber:

- **JFrame:** Classe que representa uma janela gráfica;
- **JOptionPane:** Classe para exibição de caixas de diálogo e mensagens;
- **JButton:** Classe que representa um botão;
- **JTextField:** Classe que representa uma caixa de entrada de texto;
- **JLabel:** Classe que representa um rótulo, comumente utilizado ao lado de campos.

A Listagem de Código 4.14 mostra a classe **Tela1** que herda da classe **JFrame**. Esta herança torna a classe **Tela1** uma janela gráfica. No corpo desta classe, a classe **JOptionPane** está sendo utilizada para exibir uma mensagem (caixa de diálogo).

```
import javax.swing.JFrame;

import javax.swing.JOptionPane;

public class TelaSwing extends JFrame{

    public static void main(String[] args) {

        JOptionPane.showMessageDialog(null, "Exibindo mensagem!");
```

```
    }
}
```

**Listagem de Código 4.14: Exemplo de classe que é um JFrame (Janela Gráfica).**

Perceba que foi utilizado o método estático **showMessageDialog()** da classe **JOptionPane**. Método estático é o tipo de método que pode ser invocado a partir da própria classe.

A classe da Listagem de Código 4.15 mostra outro exemplo de **JFrame** utilizando os componentes **JButton**, **JLabel** e **JTextField**. Esta tela representa a construção de um formulário com os campos “código” e “nome” e um botão “salvar”.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
public class TelaSwing2 extends JFrame {

//método principal que instancia um frame e exibe na tela
    public static void main(String[] args) {

        TelaSwing2 agenda = new TelaSwing2();
        agenda.setSize( 420,250 );
```

```
        agenda.show();
 }
//construtor da aplicação frame
public TelaSwing2() {
        try {
                inicie();
        }
        catch(Exception e) {
                e.printStackTrace();
        }
}

private void inicie() throws Exception {
        this.setLayout(null);
        this.setDefaultCloseOperation(3);
        this.setResizable(false);
        this.setTitle("Agenda");
        JButton jButtonSalvar  = new JButton();
        JLabel jLabelCodigo = new JLabel("Codigo");
        JTextField jTextFieldCodigo = new JTextField();
```

```
            JLabel jLabelNome = new JLabel("Nome");
            JTextField jTextFieldNome = new JTextField();
            //Definindo a posição x e y  - largura e altura do componente
            jLabelCodigo.setBounds(new Rectangle(0,0,50,15));
            jTextFieldCodigo.setBounds(new Rectangle(58,3,99,22));
            jLabelNome.setBounds(new Rectangle(0, 25, 50, 15));
            jTextFieldNome.setBounds(new Rectangle(58,25,99,22));
            jButtonSalvar.setBounds(new Rectangle(0, 55, 99, 27));
            jButtonSalvar.setToolTipText("insere novo item");
            jButtonSalvar.setText("Salvar");
            this.getContentPane().add(jLabelCodigo, null);
            this.getContentPane().add(jTextFieldCodigo, null);
            this.getContentPane().add(jLabelNome, null);
            this.getContentPane().add(jTextFieldNome, null);
            this.getContentPane().add(jButtonSalvar, null);
        }
}
```

**Listagem de Código 4.15: Exemplo de classe JFrame com os componentes JButton, JLabel e JTextField.**

Quando executar a classe da Listagem de Código 4.15, você perceberá que ao clicar no botão, representado pelo objeto **jButtonSalvar**, nada

acontecerá. Isto porque não foi implementado nenhum evento para o botão. O conceito de evento não é abordado neste livro.

# 5. Considerações Finais

Este livro abordou os principais conceitos da programação orientada a objetos. Para facilitar o entendimento destes conceitos, foram implementados alguns exemplos na prática com a tecnologia Java.

Entretanto, existem muitos conceitos relacionados com a plataforma e a linguagem Java não tratados neste livro. Os conceitos não abordados e que o autor recomenda ao leitor estudar são:

- JDBC (*Java Database Connectivity* - API para acesso a banco de dados);
- Eventos e outros componentes da API SWING;
- Sockets;
- Threads;
- Plataformas JME e JEE.

# Bibliografia Recomendada

ORACLE. **ORACLE**. Disponível em: <http://www.oracle.com/technetwork/java/index.html>. Acesso em: 05 de nov. de 2011.

SANTOS, R.. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Elsevier, 2003.

# Exercícios

1) Explique os conceitos de classe e objeto.

2) Cite algumas vantagens no desenvolvimento orientado a objetos.

3) Qual a diferença entre polimorfismo e sobrecarga?

4) Explique o conceito de Herança.

5) Quais as edições da plataforma Java?

6) A Linguagem de programação Java utiliza a metodologia orientada a objetos e permite a implementação de classe e objetos. Utilizando esta linguagem, implemente uma classe Computador com os atributos codigo, marca e modelo e uma classe Aplicação (com o método main) que instancia dois objetos com os seguintes dados: i) 2, “dell”, “NXY000” e ii) 6, “acer”, “YXU200”.

7) Objeto é uma instância de uma classe. Para instanciar uma classe em Java, deve-se utilizar um método especial chamado construtor. Explique quais características deste método e implemente um método construtor para uma classe Casa com os atributos numeroPorta e endereco.

8) Defina uma classe Java para representar pessoas. Esta pessoa possui os seguintes atributos:

• Nome

• Idade

• Altura

• Peso

• Sexo

9) Identifique no código abaixo um exemplo de Objeto (Instância), Método e Classe. Explique cada conceito deste.

```
public class JavaApp {

    public static void main(String[] args) {

        Java javaobj = new Java(“1.7”, “jse”);

        System.out.println(javaobj.getNome());


    }

}
```

10) O programa abaixo pode gerar uma exceção? Se a resposta for positiva, como implementar um tratamento de exceção para que seja exibida uma mensagem em caso de erro?

```
public class Exemplo1App extends Object {

public static void main(String args[])

    {

        int vetor[] = new int[4];

        for (int i=1; i<=5; i++)

            System.out.println(vetor[i]);

    }
```

}

11) Uma API (Application Programming Interface) é um conjunto de classes e componentes que facilitam o desenvolvimento de diversas aplicações em Java. Cite um exemplo de API e as suas funções.

12) Desenvolva um sistema orientado a objetos para controlar reservas de equipamentos. Todo equipamento possui os atributos codigo (representa o identificador do equipamento) e reservado (representa o status do equipamento - se está reservado ou não). Os equipamentos são computador (com os atributos codigo, reservado e capacidade_hd), datashow (com os atributos codigo, reservado e tamanho_cabo) e scanner (com os atributos codigo, reservado e resolucao). Para esta questão utilize todos os conceitos explicados no livro.