

# ORIENTAÇÃO A OBJETOS

## AULA 9

### Classes Anônimas, Internas e Coleções

Vandor Roberto Vilardi Rissoli



# APRESENTAÇÃO

- Classes Internas
- Classes Anônimas
- Coleções
- Sobreposição do *equals*
- Referências



# Classes Internas

Uma classe ainda pode ser definida dentro de outra classe, sendo somente uma delas definida como pública em seu arquivo físico de implementação, como já abordado anteriormente. Este tipo de implementação é interessante em algumas situações:

- Objetos de classes internas podem acessar diretamente os componentes de sua classe criadora, mesmo sendo estes privados (qualificador *private*)
- Invisibilidade das classes internas para outras classes do mesmo pacote
- Emprego de classes internas anônimas na definição de processamentos que usem retornos para acionamentos em tempo de execução
- Uso conveniente em programas dirigidos por eventos

# Classes Internas

A implementação deste tipo de classe fornece as mesmas mais privilégios de acesso, apesar de serem convertidas para classes normais, com nomes diferentes (\$) na interpretação em Java.

```
public class Aluno extends Pessoa implements Frequencia, Avaliacao
{
    : // componentes e programação da classe “externa”

    private class Notas implements Atividades
    {
        : // componentes e programação da classe interna
    }
}
```

→ Como outra classe qualquer, a interna também pode ser definida com seus elementos padrões (qualificador, interface, e todos aqueles usados para definir uma classe).

# Classes Internas Anônimas

A linguagem Java também permite criar classes localmente em **um único método**, sendo esta possibilidade empregada na criação de tipos que serão usados, geralmente, uma única vez no programa.

Esta possibilidade identifica um recurso de implementação muito vantajoso sobre o acesso as classes locais, tornando-as totalmente ocultas ao “mundo exterior”, inclusive para sua classe criadora.

- Somente seu método criador sabe da existência desta classe, podendo só ele utilizá-la
- Como será criado só um objeto desta classe, não é necessária nem a definição de seu nome (anônima)
- É possível retornar um objeto dentro deste método que não tenha nome, funcionando adequadamente

# Classes Internas Anônimas

Estas classes anônimas não podem possuir métodos construtores, sendo seus parâmetros de construção fornecidos ao construtor da superclasse.

Especificamente, para as situações onde uma classe interna implemente um interface, ela não pode possuir parâmetros de construção, sendo seus parênteses sempre necessários.

Um cuidado na identificação da construção de um objeto de uma classe ou de uma classe interna anônima, que estende essa classe do objeto, é averiguada pela constatação do que vem logo após os parênteses que fecham os parâmetros de construção do objeto, onde a abertura de chaves indica uma definição de classe interna anônima.

# Classes Internas Anônimas

A simples necessidade de ocultar uma classe dentro de outra pode ser conseguida pela implementação de uma classe interna. No entanto, para não permitir que o objeto desta classe interna faça referência a sua classe externa a mesma deverá ser definida com *static*.

```
/** Síntese
 *   Objetivo: Encontrar maior e menor valor do vetor
 *   Entrada:  sem entrada (geração aleatória)
 *   Saída:    maior e menor valor
 */
import java.text.DecimalFormat;
public class ProcuraValores {
    public static void main(String[] args) {
        DecimalFormat mostra= new DecimalFormat("0.000");
        final int MAX = 100;    // tamanho do vetor
        double [] vetor = new double[MAX];
        for(int aux=0; aux<vetor.length; aux++)
            vetor[aux]=10 * Math.random(); // gera aleatório
    }
}
```

# Classes Internas Anônimas

```
// continuação do exemplo anterior
```

```
    BuscaValores.mostra(vetor);  
    BuscaValores.Par valores;  
    valores = BuscaValores.maxmin(vetor);  
    System.out.println("\tMÁXIMO = " +  
                        mostra.format(valores.getMaximo()));  
    System.out.println("\tMÍNIMO = " +  
                        mostra.format(valores.getMinimo()));  
}  
}          // final da classe ProcuraValores
```

```
// Nova classe no mesmo arquivo físico
```

```
class BuscaValores
```

```
{
```

```
    public static class Par          // define classe interna
```

```
{
```

```
    private double maximo;
```

```
    private double minimo;
```



# Classes Internas Anônimas

// continuação do exemplo anterior

```
public Par (double max, double min) { // construtor
    maximo = max;
    minimo = min;
}
public double getMaximo() {
    return maximo;
}
public double getMinimo() {
    return minimo;
}
} // final da classe interna Par

public static void mostra(double [] vetor) {
    System.out.println("*MOSTRA VALORES ALEATÓRIOS*");
    for(int aux=0; aux<vetor.length; aux++)
        System.out.println("  Vetor[" + aux + "]= " +
                                vetor[aux]);
    System.out.println("\n\nANÁLISE");
}
```

# Classes Internas Anônimas

// continuação do exemplo anterior

```
public static Par maxmin(double [] vetor) {  
    if(vetor.length == 0)  
        return new Par(0,0);  
    else {  
        double max = vetor[0];  
        double min = vetor[0];  
        for(int aux=1; aux<vetor.length; aux++) {  
            if(max < vetor[aux])  
                max = vetor[aux];  
            if(min > vetor[aux])  
                min = vetor[aux];  
        }  
        return new Par(max,min);  
    }  
}
```

} // final da **classe externa BuscaValores**



# Classes Internas Anônimas

Observe um pequeno exemplo com uma janela Swing.

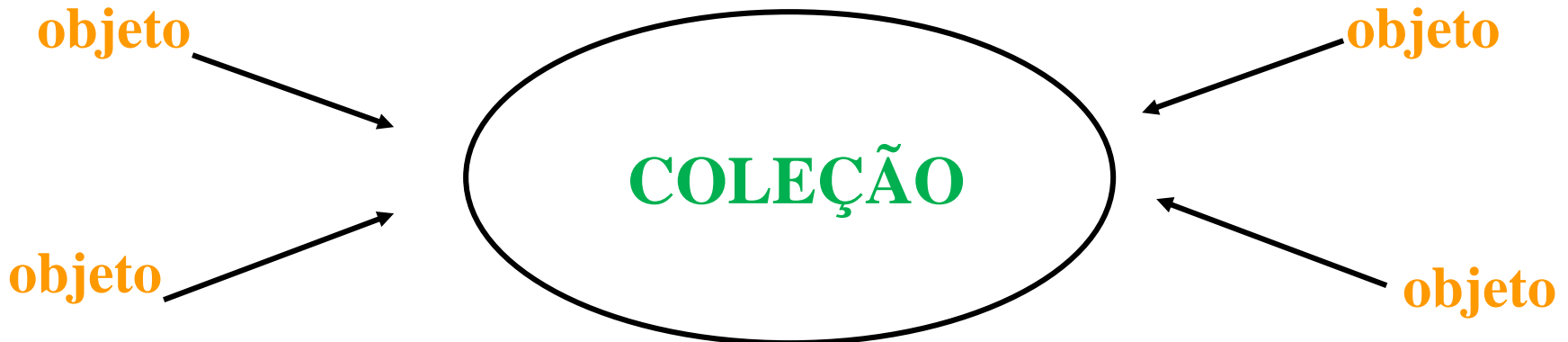
```
/** Síntese
 *   Objetivo: Apresentar uma janela gráfica Swing
 *   Entrada:  sem entrada
 *   Saída:    janela gráfica
 */
import java.awt.event.*;
import javax.swing.JFrame;
public class ExemploAnonima extends JFrame {
    public static void main(String[] args) {
        JFrame janela = new ExemploAnonima();
        janela.setBounds(100,100,250,250);
        janela.setTitle("Exemplo");
        janela.setVisible(true);
        janela.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent evJanela) {
                System.exit(0); //método que encerra o programa
            }
        }); // termina a classe interna anônima
    }
} // termina a classe
```

# Coleções

A manipulação adequada de coleções de dados em Java foi implementada por meio do *framework* Collections, que envolve um conjunto de classes e interfaces.

## Collection

- Disponível no pacote **java.util**
- Interface raiz de outras interfaces que permitem a manipulação de diversas estruturas de dados compostas
- Define as operações básicas sobre as coleções e disponibiliza vários métodos de alto nível



# Coleções

## List

- Disponível no pacote **java.util**
- Esta interface (List) estende Collection
- O comportamento para inserção de elementos em uma lista deve ser realizado no final da lista (método **add**), respeitando as características fundamentais da lista quando este método é sobrescrito de Collection
- Métodos **add** e **addAll** de Collection são sobrecarregados e possuem índice para indicar a posição da inserção
- Operações sobre lista envolvem **acesso sequencial** aos elementos anteriores àquele que está na posição especificada pelo índice
- Custo de acesso aumenta proporcionalmente ao aumento do valor de seu índice, enquanto nos Arrays o custo é independente do valor de seu índice

# Coleções


## Métodos Importantes no Uso da List

- **add(int, object)**: inseri o objeto indicado na posição informada pelo objeto alvo e desloca os elementos desta posição e as posições maiores para à direita
- **boolean addAll(int, collection)**: inseri toda coleção a partir da posição indicada, deslocando para direita os elementos desta posição e superiores. Retorna **true** se lista do objeto alvo for alterada e **false** se não for modificada
- **Object get(int)**: retorna o objeto da lista na posição indicada
- **int indexOf(object)**: retorna primeira posição que ocorre o object na lista ou **-1** se ele não existir nesta lista
- **Object remove(int)**: remove elemento da lista que esta na posição indicada e desloca os elementos das posições subseqüentes para estas posições anteriores
- **set(int, object)**: guarda objeto na posição indicada
- **List subList(int, int)**: retorna elementos que estão entre início e fim (**-1**) indicados

# Coleções

```
/** Síntese
 *   Objetivo: mostrar meses impares em alguma ordem
 *   Entrada:  nenhuma (atribuição)
 *   Saída:    meses em algumas ordenações
 */
import java.util.Arrays;
import java.util.List;
import java.util.Collections;
public class ColecaoMeses {
    public static void main(String[] args) {
        String [] meses = {"Janeiro", "Março", "Maio",
                           "Julho", "Setembro", "Novembro"};

        List lista;
        // Ordem alfabética crescente pelo nome do mês
        Arrays.sort(meses); // método de alto nível
        // Transforma um Array em Lista
        lista = Arrays.asList(meses);
        System.out.println("\nMESES EM ORDEM ALFABÉTICA");
        // Mostra a Lista em Ordem
        for(int aux=0; aux < lista.size();aux++)
            System.out.print(lista.get(aux) + "\t");
    }
}
```



poderia ser **java.util.\***;

# Coleções

```
// continuação do exemplo anterior

// Inverte os dados da lista
Collections.reverse(lista);
System.out.println("\n\nMOSTRA LISTA INVERTIDA");
// Imprime a Lista Invertida
for(int aux=0; aux < lista.size();aux++)
    System.out.print(lista.get(aux) + "\t");
}
```

- O método **size** obtém o tamanho da lista, enquanto que **get** recupera um elemento específico desta lista
- Vários métodos de alto nível estão disponíveis na manipulação de estrutura de dados e coleções
  - **sort**; *binarySearch*, **reverse**, **shuffle**, *fill*, **copy**, min, max, entre outros métodos interessantes
- O método **asList(array)** transforma um Array em List, podendo ser manipulado de várias formas (vários métodos disponíveis para um List e não no Array)



# Coleções

Dentre as classes que implementam a interface List serão estudadas somente a LinkedList e ArrayList.

## LinkedList

- Implementação de List que usa internamente lista encadeada, sendo em Java implementa como uma lista **duplamente encadeada** (cada elemento possui referência ao elemento seguinte e ao anterior)
- Acréscimo de novos elementos acontece com a inserção de novos nós entre os nós adjacentes, sendo necessário primeiro a localização desta posição
- Recomenda-se esta implementação quando a maioria das modificações acontece no início ou final da lista
- Percorrer a lista é feito de maneira seqüencial com uso do objeto iterator e não aleatoriamente, respeitando índices diretamente
- Exemplo de FIFO (*first in first out* - **fila** em E.D.A.)

# Coleções

## Métodos Importantes no Uso da LinkedList

- **LinkedList**: cria uma lista vazia
  - **LinkedList(collection)**: cria uma lista contendo os elementos da collection
  - **addFirst(object)**: inseri object como primeiro elemento
  - **addLast(object)**: inseri object como último elemento
  - **Object getFirst()**: retorna o primeiro elemento da lista
  - **Object getLast()**: retorna o último elemento da lista
  - **Object removeFirst()**: remove e retorna o primeiro elemento da lista
  - **Object removeLast()**: remove e retorna o último elemento da lista
- **toString**: retorna uma representação da lista dos elementos da estrutura de dados representada pelo objeto alvo, estando na ordem em que seriam obtidos pela interface iterator (**next**).

# Coleções

## Iterator

- Disponível no pacote **java.util**
- Consistem em duas interfaces definidas em **Collection** que percorrem a estrutura de dados desejada
- O objeto **Iterator** criado é posicionado no início da estrutura de dados, retornando seu primeiro elemento quando método **next** for acionado

## ListIterator

- Interface estendida de **Iterator** que adiciona recursos específicos para coleções do tipo **List**
- Percorre uma lista nos dois sentidos, permitido pela indexação dos elementos da **List**
- Classes que implementam a interface **List** definem também o método **listIterator**, sendo seu retorno usado como retorno do **Iterator**

# Coleções

## Métodos Importantes no Uso da Iterator

- **Iterator**: cria objeto para percorrer uma estrutura de dados
- **Object next()**: retorna o próximo elemento da estrutura de dados que está sendo percorrida pelo objeto alvo, onde a exceção **NoSuchElementException** indica seu final
- **boolean hasNext()**: retorna **true** se existir próximo elemento ou **false** se não existir
- **remove**: remove elemento da estrutura de dados retornado pelo último **next**, onde a exceção **IllegalStateException** ocorre se o elemento não existir (não foi executado o **next**)



# Coleções

## Métodos Importantes no Uso da ListIterator

- **ListIterator**: cria objeto para percorrer uma lista
- **Object previous()**: retorna elemento anterior ao elemento corrente, gerando exceção **NoSuchElementException** se não existir elemento anterior
- **boolean hasPrevious()**: retorna **true** se existir um elemento anterior ao elemento corrente e **false** em caso contrário
- **int previousIndex()**: retorna o índice do elemento anterior ao corrente e **-1** caso não exista elemento anterior
- **int nextIndex()**: retorna o índice do elemento posterior ao elemento corrente ou o tamanho da lista se este elemento não existir
- **set(object)**: altera o elemento corrente para o **object** indicado na posição corrente, retornado pelo último **next** ou **previous** executado

# Repetição Alternativa para Coleções

A partir da versão 5 (Java 1.5) foi agregado a linguagem Java uma variação da instrução **for** para trabalhar com coleções de objetos.

Esta instrução é conhecida **for each** e percorre uma coleção de objetos por meio de uma definição mais concisa e elegante.

```
for(<tipo> <identificador> : <expressão>)
```

<tipo> - tipo de dado da coleção (classe)

<identificador> - nome do tipo a ser iterado

<expressão> - deve ser uma instancia de classe capaz de implementar **Iterable**

Exemplo que mostrará todos os carros armazenados em uma lista de carros cadastrados:

```
for(Carro auto : lista)  
    System.out.println(auto);
```

# Coleções

```
/** Síntese
 *   Objetivo: Mostrar lista de nomes em ordem
 *   Entrada:  Nenhuma (só atribuições)
 *   Saída:    Nomes em ordem alfabética crescente
 */
import java.util.*;
public class ColecaoStrings {
    public static void main(String[] args) {
        String nome1 = new String("Edson Arantes");
        String nome2 = new String("Luiza Brunet");
        String nome3 = new String("Diego Maradona");
        // Lista de quaisquer objetos
        List lista = new ArrayList();
        lista.add(nome1);
        lista.add(nome2);
        lista.add(nome3);
        Collections.sort(lista);
        for(String str : lista)
            System.out.println(str);
    }
}
```

# Exercício de Fixação

- 1) Elabore um programa orientado a objeto que possua uma superclasse abstrata chamada *Datas* que guardará dia, mês e ano válidos a partir de 1901 (valor definido em uma constante *ANO*). Crie sua subclasse chamada *Aniversario* que armazenará um nome completo que não pode ser nulo e nem possuir menos que 5 caracteres. Este valor também estará definido em outra constante (*MINIMO*).

Seu programa permitirá que o usuário cadastre quantos aniversariantes ele desejar, podendo a qualquer momento fazer Novo cadastro, Mostrar todos cadastro por ordem do cadastro realizado, Mostrar somente os nomes cadastrados em ordem alfabética por meio de uma lista em Java (*List*), Apresentar os nomes cadastrados misturado (ver na API) pela *Collections* ou ainda Sair do programa. Todas estas consultas de dados só poderão ser feitas pela *for-each* ou usando o *Iterator* ou *ListIterator*.

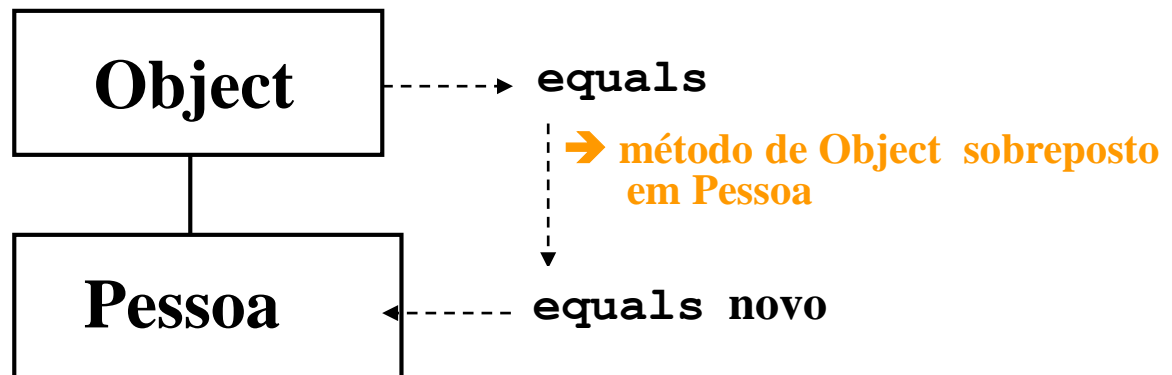


# Sobreposição de Método do Object

O método **equals** é definido em **Object** e compara os endereços de memória dos 2 objetos envolvidos.

No entanto, a classe **String** sobrepõe este método para comparar os conteúdos que estão armazenados nestas duas respectivas strings que serão comparadas por meio do **equals** retornando verdadeiro (**true**) se elas forem idênticas ou falso (**false**) se forem diferentes.

Normalmente, existe a necessidade de se comparar objetos de uma mesma classe a fim de se apurar se tal objeto já está armazenado ou não. Isso pode ocorrer em POO, por meio da sobreposição do método **equals**, similar a **toString**.



# PACOTE e Sobreposição do *equals*

```
/* Síntese
 * Objetivo: cadastrar um conjunto de pessoas
 * Entrada: quantidade, nome e sexo de cada pessoa
 * Saída:   listar todas pessoas cadastradas
 */

package principal;

import java.util.ArrayList;
import javax.swing.JOptionPane;
import visao.Visao;

public class Main {
    public static void main(String[] args) {
        ArrayList pessoas = new ArrayList();
        int qtde = 0;
        qtde = Visao.obtemQuantidade("Digite a quantidade"+
                                     " de Pessoas que serão cadastradas?");
        Visao.cadastraPessoa(pessoas, qtde);
        Visao.mostrarCadastros(pessoas);
    }
}
```

# PACOTE e Sobreposição do *equals*

```
/*Síntese
 * Conteúdo: nome, sexo
 *      - getName(), getSexo(), setName(String)
 *      - setSexo(int), toString(Pessoa), equals(Object)
 */
package dados;
public class Pessoa {
    // atributos
    private String nome;
    private char sexo;
    // métodos
    public Pessoa (String nome, char sexo) {
        this.setName(nome);
        this.setSexo(sexo);
    }
    public String getName() {
        return nome;
    }
    public void setName(String nome) {
        this.nome = nome;
    }
    public char getSexo() {
        return sexo;
    }
}
```

# PACOTE e Sobreposição do *equals*

```
// continuação do exemplo anterior
public void setSexo(char sexo) {
    this.sexo = sexo;
}
// Sobreposições
public String toString() {
    return (this.getNome() + "\t" +
        (this.getSexo() == 'F' ? "Feminino" :
        (this.getSexo() == 'M' ? "Masculino" :
        "Inválido")));
}
public boolean equals(Object obj) {
    Pessoa novaPessoa = (Pessoa) obj;
    if (novaPessoa.nome.equals(this.getNome()))
        return true;
    else
        return false;
}
}
```

---

**//**

```
/* Síntese
 * Conteúdo:
 * - obterQuantidade(String), mostrarCadastros(ArrayList)
 * - cadastraPessoa(ArrayList, int)
 */
```

# PACOTE e Sobreposição do *equals*

// continuação do exemplo anterior

```
package visao;
import java.util.ArrayList;
import javax.swing.JOptionPane;
import dados.*;
import validacao.*;
public class Visao {
    public static int obtemQuantidade(String mensagem) {
        String num;
        do {
            num = JOptionPane.showInputDialog(null,
                mensagem, "Cadastro", JOptionPane.QUESTION_MESSAGE);
        } while (!Validacao.isValidaQtde(num));
        return(Integer.parseInt(num));
    }
    public static void cadastraPessoa(ArrayList pessoas,
                                       int maximo) {
        String nome, sexo;
        for(int aux=0; aux < maximo; aux++) {
            do {
                nome = JOptionPane.showInputDialog(null,
                    "Informe o nome da pessoa:", "Cadastro",
                    JOptionPane.QUESTION_MESSAGE);
            } while (!Validacao.isValidaNome(nome));
        }
    }
}
```

# PACOTE e Sobreposição do *equals*

// continuação do exemplo anterior

```
do {
    sexo = JOptionPane.showInputDialog(null,
        "Informe o sexo da pessoa:", "Cadastro",
        JOptionPane.QUESTION_MESSAGE);
} while(!Validacao.isValidaSexo(sexo));
Pessoa pes = new Pessoa(nome,
    sexo.toUpperCase().charAt(0));
if(aux > 0) {
    if(!Validacao.isValidaPessoa(pessoas, pes, aux)) {
        JOptionPane.showMessageDialog(null,
            "Esta Pessoa já está cadastrada. " +
            "Cadastre um outro", "Erro",
            JOptionPane.ERROR_MESSAGE);
        aux--;
        System.out.println("Pessoa não cadastrada.");
    }
    else {
        pessoas.add(pes);
        System.out.println("Pessoa " + (aux+1) +
            " cadastrada com sucesso.");
    }
}
else {
```

# PACOTE e Sobreposição do *equals*

// continuação do exemplo anterior

```
        pessoas.add(pes);
        System.out.println("Pessoa " + (aux+1) +
                           " cadastrada com sucesso.");
    }
}

public static void mostrarCadastros(ArrayList pessoas) {
    System.out.println("\n\n\n\nNome\t\tSexo");
    System.out.println("=====\t=====");
    for(int aux = 0; aux < pessoas.size(); aux++)
        System.out.println(pessoas.get(aux));
}
}
```

//

/\*Síntese

\* Conteúdo:

- \* - isValidaQtde(String), isValidaNome(String),
- \* - isValidaSexo(String),
- \* - isValidaPessoa(ArrayList, Pessoa, int)

\*/

**package validacao;**

import java.util.ArrayList;

import javax.swing.JOptionPane;

# PACOTE e Sobreposição do *equals*

// continuação do exemplo anterior

```
import dados.*;
public class Validacao {
    public static boolean isValidaQtde(String num) {
        int qtde = 0;
        try {
            qtde = Integer.parseInt(num);
        } catch (NumberFormatException exc) {
            JOptionPane.showMessageDialog(null,
                "Valor incorreto, pois deve ser inteiro e " +
                "maior que zero.", "Erro",
                    JOptionPane.ERROR_MESSAGE);
            return false;
        }
        if(qtde <= 0) {
            JOptionPane.showMessageDialog(null,
                "Valor inválido, pois deve ser maior que zero.",
                "Erro", JOptionPane.ERROR_MESSAGE);
            return false;
        }
        else
            return true;
    }
}
```



# PACOTE e Sobreposição do *equals*

// continuação do exemplo anterior

```
public static boolean isValidaNome(String nome) {
    return((nome != null) && (!nome.isEmpty())) ? true : false;
}
public static boolean isValidaSexo(String sexo) {
    if((sexo != null) && (!sexo.isEmpty())) {
        if(sexo.toUpperCase().charAt(0) == 'F' ||
            sexo.toUpperCase().charAt(0) == 'M')
            return true;
        else
            return false;
    }
    else
        return false;
}
public static boolean isValidaPessoa(ArrayList pessoas,
                                     Pessoa pes, int qtde) {
    for(int cont = 0; cont < qtde; cont++) {
        Pessoa pp = (Pessoa) pessoas.get(cont);
        if(pes.equals(pessoas.get(cont)))
            return false;
    }
    return true;
}
}
```

# Exercício de Fixação

2) Faça um programa orientado a objeto que consiga armazenar quantos cadastros de pessoas o usuário desejar, coletando de cada uma o sexo, nome completo e idade em anos. Nenhum destes dados pode ser nulo ou menor e igual a zero. Este cadastro só poderá acontecer se for confirmando que não existe tal pessoa no cadastro. Seria considerada mesma pessoa somente quando o nome completo e a idade forem iguais e o cadastro não será mais permitido de ser realizado. Toda vez que uma igualdade (duplicidade) for detectada deverá ser registrado na console que houve uma duplicidade e todos os dados armazenados pelo registro da duplicidade deverão ser apresentados em uma linha, sendo na linha abaixo mostrado os dados que se estava tentando realizar um novo cadastro. Este processo de cadastro será interrompido e os dados informados até o momento não poderão ser armazenados no *ArrayList* que os estava guardando, mas o processo de cadastro ainda continuará sendo realizado se o usuário desejar. Quando este (usuário) não quiser mais fazer cadastro todos os registros já efetuados serão mostrados na console com um salto de 10 linhas de qualquer outra informação que possa existir na mesma. Todas estas consultas de dados só poderão ser feitas pela *for-each*.

# Referência de Criação e Apoio ao Estudo

## Material para Consulta e Apoio ao Conteúdo

- HORSTMANN, C. S., CORNELL, G., Core Java2 , volume 1, Makron Books, 2001.
  - Capítulo 5
- FURGERI, S., Java 2: Ensino Didático: Desenvolvendo e Implementando Aplicações, São Paulo: Érica, 2002.
  - Capítulo 7
- Universidade Brasília (UnB FGA)
  - <https://cae.ucb.br/conteudo/unbfga>  
(escolha da disciplina **Orientação a Objetos** no menu superior)

