

# PROGRAMAÇÃO PARA WEB I

## COLEÇÕES

**Profa. Silvia Bertagnolli**

# COLEÇÃO: O QUE É?

Estrutura de dados que permite armazenar vários objetos (Listas, Filas, Pilhas, Árvores binárias, etc.)

Coleções são usadas para:

- armazenar
- recuperar
- manipular
- pesquisar

} objetos

# COLEÇÕES: VANTAGENS

- Reduz o esforço de programação
- Aumenta a qualidade e o desempenho do código
- Fácil de aprender
- Estruturas de Dados e seus algoritmos já estão prontos

Segundo Deitel com as coleções "você utiliza estruturas de dados existentes, sem se preocupar com a maneira como são implementadas"

# COLEÇÕES PRIMITIVAS

A versão 1.0 do Java tinha suporte para:

- Vector, Stack, Hashtable, Properties, BitSet e Enumeration

Após vários outros tipos de coleções foram introduzidos:

- ArrayList, LinkedList, TreeSet, LinkedHashSet, HashSet, TreeMap, HashMap, LinkedHashMap

# COLEÇÕES JAVA

Collections podem ser:

- **organizadas** coleções serão percorridas na mesma ordem em que os elementos foram inseridos: `LinkedHashSet`, `ArrayList`, `Vector`, `LinkedList`, `LinkedHashMap`
- **ordenadas** possui métodos/regras para ordenação dos elementos: `TreeSet`, `PriorityQueue`, `TreeMap`

# COLEÇÕES JAVA

Listas: organizadas, podem conter elementos duplicados, mantendo a ordem em que foram adicionados e usam índices - implementam `List`

Conjuntos: itens exclusivos, mantém sua própria ideia de ordem, sem pesquisa através de índices - implementam `Set`

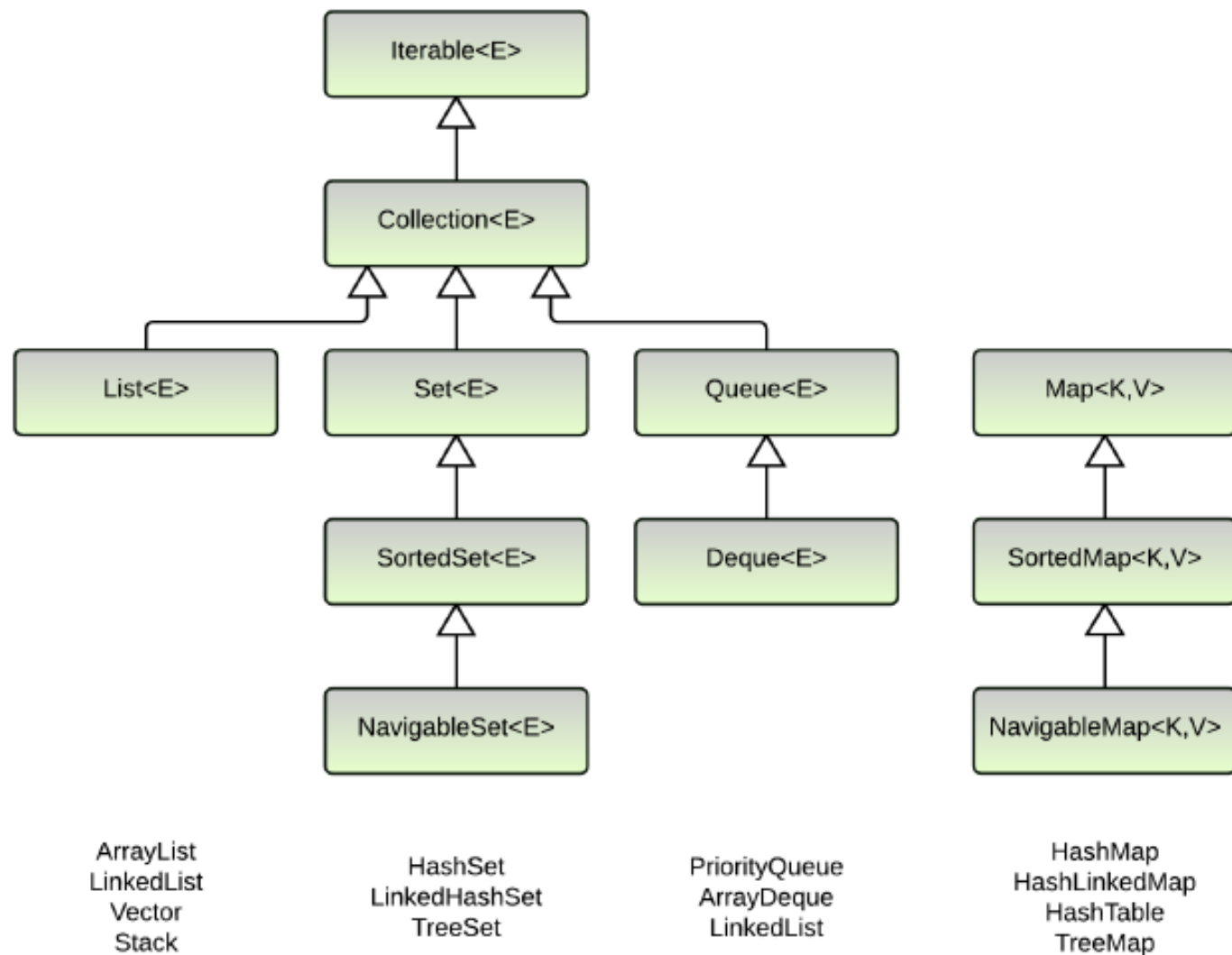
Queue: filas que são usadas para conter elementos antes do processamento (usado com threads)

# COLEÇÕES JAVA

Deque: Do inglês “Double Ended QUEUE”, onde uma queue permite apenas inserções no fim da fila e remoções do seu início, um deque permite que inserções e remoções sejam feitas no início e no fim da fila, ou seja, um objeto que implemente Deque pode ser usado tanto como uma fila FIFO (firstin, firstout), quanto uma fila LIFO (lastin, firstout).

Mapas: usados para armazenar pares de objetos. Possuem itens com uma identificação (chave) exclusiva – implementam Map

# HIERARQUIA DE INTERFACES





# INTERFACE ITERABLE

Seu objetivo é definir que qualquer coleção “filha” possa ser percorrida pelo “for melhorado”

Define métodos que permitem percorrer qualquer tipo de coleção:

- `boolean hasNext()`: retorna `true` se existem mais elementos a serem acessados na coleção vinculada a esse `Iterator`
- `E next()`: retorna o próximo elemento disponível na coleção vinculada a esse `iterator`
- `void remove()`: remove da coleção o último elemento acessado através desse `Iterator`

# INTERFACE COLLECTION

Define os métodos mais gerais, independentes da estrutura e da forma de acesso da coleção – estabelece um padrão de operações básicas para as coleções:

- `size()` – determina o número de elementos armazenados
- `remove()` – remove o elemento informado
- `add()` – adiciona o elemento informado
- `isEmpty()` – verifica se está vazia
- `iterator()` – percorre a coleção
- `contains()` – verifica se um elemento está armazenado

# INTERFACE COLLECTION

## Operações em massa:

- `boolean addAll(Collection<? extends E> c)`: adiciona à coleção todos os elementos da coleção passada como parâmetro
- `void clear()`: esvazia a coleção, mas não elimina da memória os objetos que ela referenciava, a não ser que não haja mais nenhuma outra referência para os mesmos
- `boolean containsAll(Collection<?> c)`: retorna true se a coleção contém todos os elementos da coleção informada como parâmetro

# INTERFACE COLLECTION

## Operações em massa:

- `boolean removeAll(Object o)`: remove da coleção todos os elementos da coleção informada como parâmetro
- `boolean retainAll(Collection<?> c)`: mantém na coleção somente os elementos da coleção informada como parâmetro
- `Object[] toArray()`: converte essa coleção para um array de `Object`.

LISTAS

# INTERFACE LIST

Define coleções que se organizam como arrays de tamanho dinâmico, de forma que cada elemento seja acessível por um índice

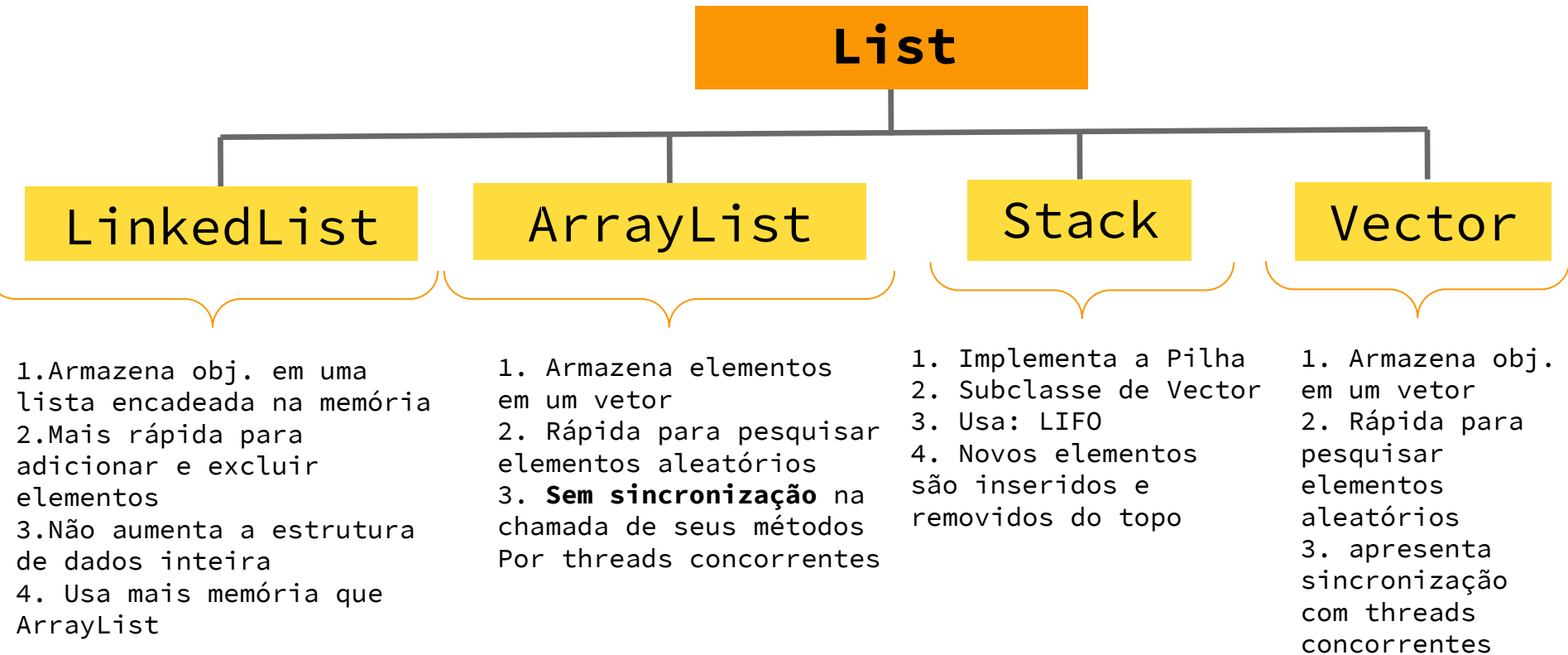
Permite acesso posicional - índices (0 até size-1)

Novos elementos podem ser criados ou removidos em qualquer posição e pode haver elementos duplicados

Nem todas as listas garantem acesso indexado com tempo constante

Listas aceitam a inserção de valores nulos

# HIERARQUIA DA CLASSES DE LISTAS



# ARRAYLIST

- é uma lista que usa um array como estrutura de dados
- tem acesso direto aos seus elementos através do índice e permite adicionar e remover elementos de forma eficiente apenas no fim
- caso seja necessário aumentar o seu tamanho, o custo será alto, já que todo o array será copiado para um novo com maiores dimensões



# ARRAYLIST: EXEMPLO

```
public class Lista1{  
    public static void main(String args[]){  
        List<Integer> lista = new ArrayList<>();  
        lista.add(10);  
        lista.add(20);  
        for(Integer obj: lista) {  
            System.out.println(obj);  
        }  
        System.out.println(lista.indexOf(20));  
    }  
}
```

# LINKEDLIST

- Usa uma lista duplamente encadeada de elementos, onde cada elemento sabe quem é o próximo e quem é o anterior
- Primeiro e último elementos podem ser acessados de maneira direta, mas os restantes terão um custo de acesso (elemento na posição N da lista, temos que passar por todos os elementos de 0 até N-1)
- O maior benefício é que ela pode crescer indefinidamente
- Caso a lista cresça constantemente a melhor opção é a LinkedList

# LINKEDLIST: MÉTODOS (1/2)

## Inclusão:

```
public void add(in index, Object element)
```

```
public void addFirst(Object element)
```

```
public void addLast(Object element)
```

## Recuperação:

```
public Object getFirst()
```

```
public Object getLast()
```

# LINKEDLIST: MÉTODOS (2/2)

Exclusão:

```
public boolean remove(Object element)
```

```
public Object removeFirst()
```

```
public Object removeLast()
```

Os métodos `addFirst()` e `removeFirst()` podem simular uma pilha (LIFO – Last In First Out)

Os métodos `addLast()` e `removeFirst()` podem simular uma fila (FIFO – First In First Out)

# LINKEDLIST: EXEMPLO 2

```
public class Lista2{  
    public static void main(String args[]){  
        LinkedList<Number> lista = new LinkedList<>();  
        lista.add(10);  
        lista.add(20.89);  
        lista.add(30L);  
        lista.add(23.5F);  
        lista.removeFirst();  
        lista.removeLast();  
        for (Number number : numeros) {  
            System.out.println(number);  
        }  
    }  
}
```

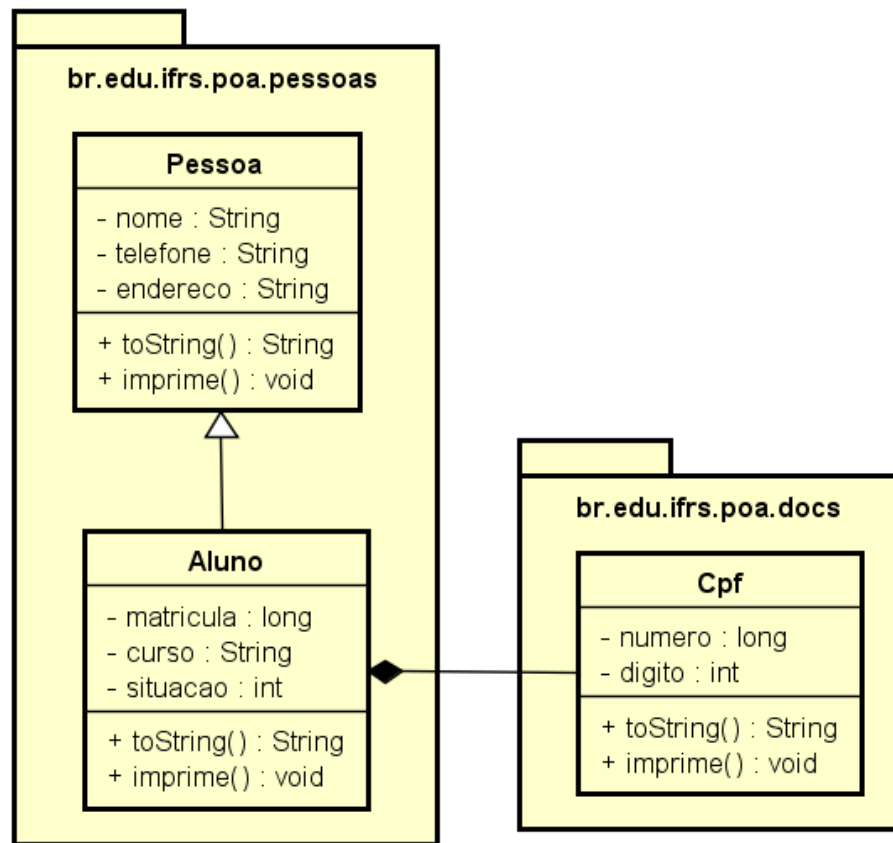
# LINKEDLIST: USO DO GET

Recomenda-se não usar o método `get()`, porque o acesso aos elementos é aleatório o que ocasiona perda de desempenho

```
for(int i=0; i< list.size(); i++)  
    System.out.println(list.get(i));
```

# EXERCÍCIOS

1. Como declarar um ArrayList para objetos do tipo Cpf?
2. Como declarar uma LinkedList para armazenar objetos do tipo Pessoa e Aluno?



# CONJUNTOS



# INTERFACE SET

Os conjuntos não aceitam itens duplicados, permitem a inserção de objetos nulos, não usam índices e são rápidos para inserir e pesquisar elementos

Caso dois objetos sejam iguais, considerando o método equals, apenas um será incluído

Representação para a abstração matemática de “conjuntos”

Suporta as operações de união, intersecção e diferença entre conjuntos

Podem ser vazios, mas não podem ser infinitos

# HIERARQUIA DA CLASSES DE CONJUNTOS

**Set**

```
graph TD; Set[Set] --> HashSet[HashSet]; Set --> TreeSet[TreeSet]; HashSet --> LinkedHashSet[LinkedHashSet];
```

**LinkedHashSet**

1. Armazena objetos em uma tabela hash e lista encadeada
2. Estrutura organizada, mantém a ordem de inserção
3. A melhor implementação para uso geral é a LinkedHashSet, porque é mais flexível que uma HashSet e mais rápida que uma TreeSet

**HashSet**

1. Armazena objetos em uma tabela hash
2. Estrutura não ordenada

**TreeSet**

1. Armazena objetos em uma árvore
2. Estrutura ordenada
3. Tem maior complexidade, porque está ordenada

# INTERFACE SET PRINCIPAIS MÉTODOS

```
public boolean addAll(Collection c)
```

adiciona ao conjunto que o invocar todos os elementos da coleção passada como parâmetro - equivale a operação de **UNIÃO** de conjuntos

```
public boolean retainAll(Collection c)
```

mantém no conjunto que executar o método somente os elementos encontrados no conjunto passado como parâmetro - equivale a operação de **INTERSECÇÃO** de conjuntos

```
public boolean removeAll(Collection c)
```

Remove do conjunto que executar o método todos os objetos iguais encontrados no conjunto passado como parâmetro - equivale a operação de **DIFERENÇA** de conjuntos

# HASHSET

Como utiliza hash é mais rápido para operações de modificação

Conjunto não organizado/classificado e não ordenado

Não garante a ordem de inserção, pois depende do hash

Usar: quando for necessário um conjunto sem duplicatas e sem ordem para iteração

# HASHSET: EXEMPLO

```
public class Conjunto1{  
    public static void main(String args[]){  
        Set<String> conjunto = new HashSet<>();  
        conjunto.add("Dois");  
        conjunto.add("Tres");  
        conjunto.add("Um");  
        conjunto.add("Um");  
        for(String num : conjunto) {  
            System.out.println(num);  
        }  
    }  
}
```

# LINKEDHASHSET

Permite a iteração na ordem em que os elementos foram inseridos

Ordenado pela sequência de inserção

# LINKEDHASHSET: EXEMPLO

```
public class Conjunto2{  
    public static void main(String args[]){  
        LinkedHashSet<String> cidades = new  
                                                    LinkedHashSet<String>();  
        cidades.add("Porto Alegre");  
        cidades.add("Canoas");  
        cidades.add("Alvorada");  
        cidades.add("Viamão");  
        for(String cidade: cidades) {  
            System.out.println(cidade);  
        }  
    }  
}
```

Saída:  
Porto Alegre  
Canoas  
Alvorada  
Viamão

# TREESet

Permite que os elementos fiquem em seqüência ascendente – ordem natural

Mais lenta que HashSet ou LinkedList

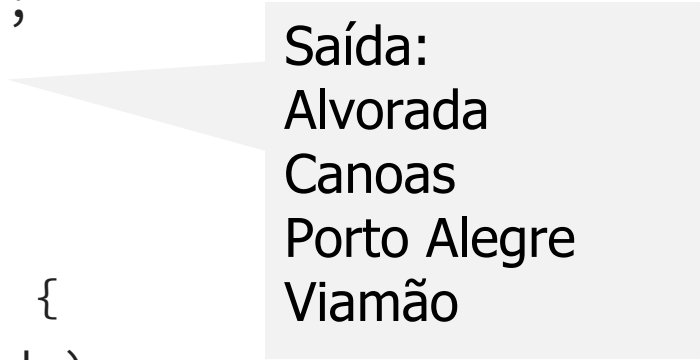
Árvore com  $n$  elementos =  $\log_2 n$  comparações para localizar a posição correta do novo elemento

Permite customizar a ordem dos elementos – definir as regras de ordenação



# TREESET: EXEMPLO

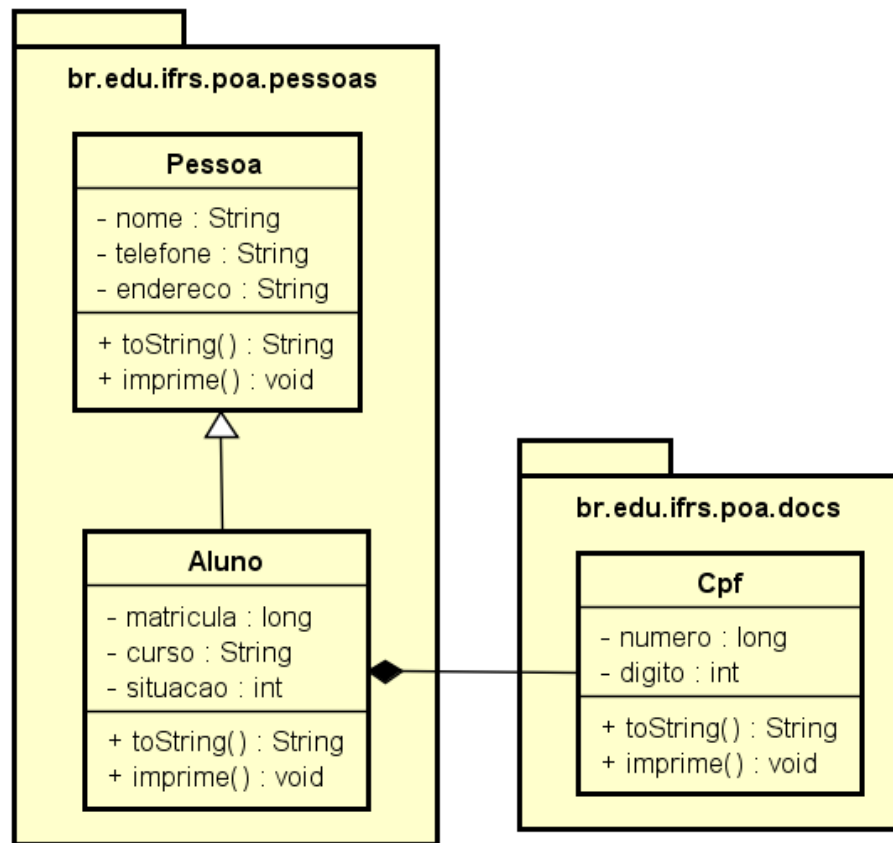
```
public class Conjunto3{  
    public static void main(String args[]){  
        Set<String> cidades = new TreeSet<>();  
        cidades.add("Porto Alegre");  
        cidades.add("Canoas");  
        cidades.add("Alvorada");  
        cidades.add("Viamão");  
        for(String cidade: cidades) {  
            System.out.println(cidade);  
        }  
    }  
}
```



Saída:  
Alvorada  
Canoas  
Porto Alegre  
Viamão

# EXERCÍCIOS

1. Como declarar um TreeSet para objetos do tipo Integer?
2. Crie dois conjuntos e faça a união, diferença ou intersecção deles usando métodos de Set
3. Como declarar um HashSet para armazenar objetos do tipo Aluno?
4. Faça o exercício TesteConjuntos do projeto que está no Moodle



MAPAS

# INTERFACE MAP

A interface Map não descende de Collection, porém faz parte do framework de coleções da linguagem Java

Não permite chaves duplicadas e mapeia chaves K para valores V

Toda chave é única e exclusiva - mapeada para um valor específico - tanto a chave quanto o objeto são valores

A chave é usada para achar um elemento rapidamente

Permite procurar um valor com base na chave, solicitar um conjunto apenas com os valores ou somente com as chaves

# HIERARQUIA DA CLASSES DE LISTAS

## Map

```
graph TD; Map[Map] --- LinkedHashMap[LinkedHashMap]; Map --- HashMap[HashMap]; Map --- Hashtable[Hashtable]; Map --- TreeMap[TreeMap];
```

### LinkedHashMap

1. Armazena objetos em uma tabela hash e uma lista encadeada

### HashMap

1. Armazena objetos em uma tabela hash
2. Estrutura não ordenada
3. Permite nulo para a chave e para os valores armazenados

### Hashtable

1. Versão sincronizada da HashMap
2. Não permite chaves e valores nulos

### TreeMap

1. Armazena elementos em uma árvore
2. Ordenado pela chave
3. Apenas os valores podem ser nulos

# INTERFACE MAP PRINCIPAIS MÉTODOS

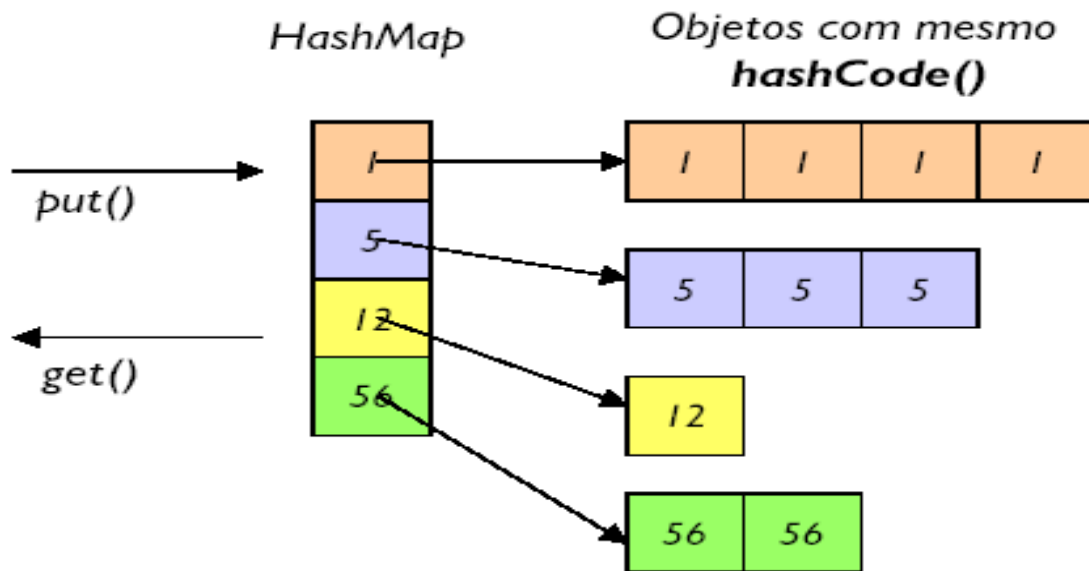
`public Object get(Object key)` - retorna o objeto identificado pela chave

`public Object put(Object value, Object key)` - insere um objeto com uma chave no mapa, retorna null se já existir o mapeamento

`public Set keySet()` - retorna o conjunto de chaves contido no mapeamento

`public Collection values()` - retorna uma coleção com os valores do mapeamento

# HASHMAP: FUNCIONAMENTO



# HASHMAP

```
public class Mapa1{  
    public static void main(String args[]){  
        Map<String, Integer> map = new HashMap<>();  
        map.put("um", new Integer(1));  
        map.put("dois", new Integer(2));  
        map.put("tres", new Integer(3));  
        Iterator it = map.values().iterator();  
        while(it.hasNext())  
            System.out.println((Integer)it.next());  
    }  
}
```



# HASHTABLE

Tabela de hash

possibilita procurar itens armazenados utilizando uma chave associada

=> chave é um objeto

=> uma hash NÃO pode possuir chaves duplicadas

# HASHTABLE: EXEMPLO

```
public static void main(String[] args) {  
    Hashtable<String, Integer> mapTable = new Hashtable<>();  
    mapTable.put("cod1", 1);  
    mapTable.put("cod2", 2);  
    mapTable.put("cod3", 3);  
    Enumeration enumIter = mapTable.elements();  
    while(enumIter.hasMoreElements()){  
        Integer num = (Integer) enumIter.nextElement();  
        System.out.println(num);  
    }  
}
```

# TREEMAP

É um mapa ordenado pela ordem natural dos elementos

Permite definir as próprias regras de ordenação no momento da criação do mapa

# LINKEDHASHMAP

Mantém a ordem de inserção dos elementos

Mantém uma lista encadeada de ponteiros entre as chaves

Mais lenta que HashMap para inserção e remoção

# COMO OBTER SÓ AS CHAVES?

```
public class MapaChaves{  
    public static void main(String args[]){  
        Map<Integer, String> map = new LinkedHashMap<>();  
        map.put(1, "um");  
        map.put(2, "dois");  
        map.put(3, "três");  
        Set<Integer> chaves = map.keySet();  
        for(Integer chave : chaves)  
            System.out.println(chave);  
    }  
}
```

# COMO OBTER SÓ OS VALORES?

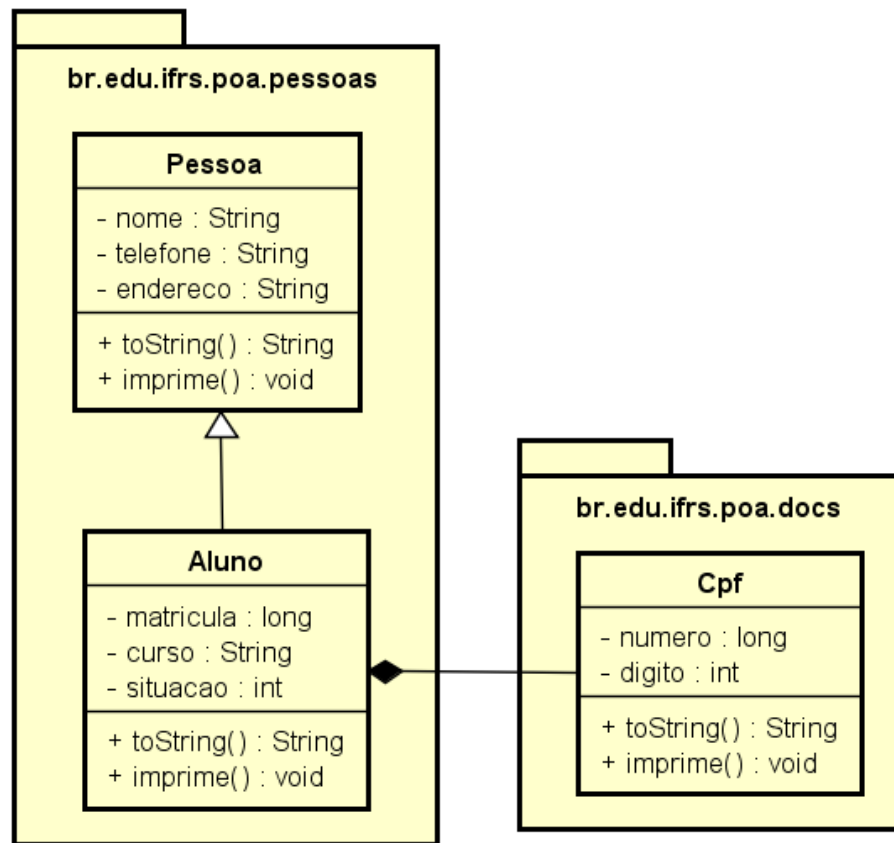
```
public class MapaValores{  
    public static void main(String args[]){  
        Map<Integer, String> map = new LinkedHashMap<>();  
        map.put(1, "um");  
        map.put(2, "dois");  
        map.put(3, "três");  
        Collection<String> valores = map.values();  
        for(String valor : valores)  
            System.out.println(valor);  
    }  
}
```

# COMO OBTER CHAVES E VALORES?

```
public class MapaChaveValores{  
    public static void main(String args[]){  
        Map<Integer, String> map = new LinkedHashMap<>();  
        map.put(1, "um");  
        map.put(2, "dois");  
        map.put(3, "três");  
        Set<Integer> chaves = map.keySet();  
        for(Integer chave : chaves){  
            System.out.println("chave: " + chave +  
                               " - valor: " + map.get(chave));  
        }  
    }  
}
```

# EXERCÍCIOS

1. Como declarar um TreeMap para objetos do tipo String?
2. Crie um HashMap que permite adicionar, excluir e listar objetos do tipo Cpf
3. Como declarar um LinkedHashMap para armazenar objetos do tipo Aluno e Pessoa





# CLASSE COLLECTIONS

# CLASSE COLLECTIONS

Essa classe possui vários métodos estáticos para manipular uma coleção:

- `sort()` - utilizado para ordenar os elementos da coleção
- `binarySearch()` - pesquisa um elemento em toda a coleção usando busca binária
- `shuffle()` - utilizado para embaralhar os elementos contidos na coleção

# CLASSE COLLECTIONS

Essa classe possui vários métodos estáticos para manipular uma coleção:

- `fill()` - usado para preencher a coleção com alguns valores
- `swap()` - usado para trocar a posição de elementos em uma coleção
- `rotate()` - desloca todos os elementos uma determinada quantidade de posições. Com esse método a lista funciona como uma **lista circular**, com isso, os últimos elementos da lista vão para as primeiras posições

O QUE ACONTECE?

# O QUE ACONTECE COM O CÓDIGO ABAIXO?

FUNCIONA?

```
public class Ordenacao1 {  
    public static void main(String[] args) {  
        TreeMap<Integer, Cpf> mapaCpfs = new TreeMap();  
        mapaCpfs.put(1, new Cpf(1, 1));  
        mapaCpfs.put(5, new Cpf(5, 5));  
        mapaCpfs.put(3, new Cpf(3, 3));  
        Set<Integer> chaves = mapaCpfs.keySet();  
        for (Integer chave : chaves) {  
            if(chave != null)  
                System.out.println(chave);  
        }  
    }  
}
```

# O QUE ACONTECE COM O CÓDIGO ABAIXO?

FUNCIONA?

```
public class Ordenacao2 {  
    public static void main(String[] args) {  
        TreeSet<Cpf> conjuntoCpfs = new TreeSet();  
        conjuntoCpfs.add(new Cpf(1, 1));  
        conjuntoCpfs.add(new Cpf(5, 5));  
        conjuntoCpfs.add(new Cpf(3, 3));  
        conjuntoCpfs.add(new Cpf(2, 2));  
        for (Cpf conjuntoCpf : conjuntoCpfs) {  
            if(conjuntoCpf != null)  
                System.out.println(conjuntoCpf.toString());  
        }  
    }  
}
```

# O QUE ACONTECE COM O CÓDIGO ABAIXO?

```
public class Ordenacao2 {  
    public static void main(String[] args) {  
        TreeSet<Cpf> conjuntoCpfs = new TreeSet();  
        conjuntoCpfs.add(new Cpf(1, 1));  
        conjuntoCpfs.add(new Cpf(5, 5));  
        conjuntoCpfs.add(new Cpf(3, 3));  
        conjuntoCpfs.add(new Cpf(2, 2));  
        for (Cpf conjuntoCpf : conjuntoCpfs) {  
            if(conjuntoCpf != null)  
                System.out.println(conjuntoCpf.toString());  
        }  
    }  
}
```

Exceção:  
Cpf cannot be  
cast to class  
java.lang.Com  
parable

# COMO ORDENAR TADS?

Para ordenar Tipos Abstratos de Dados definidos pelo usuário é necessário definir os métodos que permitem ordenar objetos de um determinado tipo

A forma mais simples é definir o `equals()` e o `hashCode()`

Faça isso para a classe `Cpf` do pacote `docs`!



ORDENANDO COLEÇÕES

# ORDENANDO COLEÇÕES: INTERFACE COMPARABLE

Usada para definir a **ordem natural** dos objetos

Possui apenas um método - **`int compareTo(Object o)`**, ele fornece a regra de comparação do próprio objeto com um outro, e determina como é feita a ordenação dos objetos na coleção

As classes Integer, Double e String, já implementam a interface Comparable, logo a ordenação funciona quando os objetos armazenados na coleção são desse tipo de dado

# ORDENANDO COLEÇÕES: INTERFACE COMPARABLE

## `compareTo()`

- retorna **0** (zero) se os dois objetos são iguais
- retorna **-1** se o objeto está “antes” do objeto passado como argumento
- retorna **1** se o objeto está “depois” do que foi passado como argumento

# INTERFACE COMPARABLE: EXEMPLO

```
public class Cpf implements Comparable<Cpf>{  
    //...  
    @Override  
    public int compareTo(Cpf objCpf) {  
        if(getNumero() == objCpf.getNumero())  
            return 0;  
        else if(getNumero() < objCpf.getNumero())  
            return -1;  
        else return 1;  
    }  
}
```

# ORDENANDO COLEÇÕES: INTERFACE COMPARABLE

## Recomendação:

A implementação do método **compareTo()** deve ser coerente com a do método `equals()`. Por exemplo, na classe `Cpf`, o método `equals()` deve retornar verdadeiro quando o número e dígito do `Cpf` forem iguais a de outro objeto do tipo `Cpf`

# ORDENANDO COLEÇÕES: INTERFACE COMPARABLE

Objetos de classes que não implementam **Comparable** não podem ser inseridos numa coleção **TreeSet** ou **TreeMap** sem um comparador específico

Coleções ordenadas que não possuem um comparador, a máquina virtual tenta usar o método **compareTo()** para ordenar os objetos, lançando uma exceção quando não o encontra

FAÇA A CLASSE ALUNO DE MODO QUE A  
ORDENAÇÃO SEJA FEITA PELA MATRÍCULA

COMPARATOR



# ORDENANDO COLEÇÕES: INTERFACE COMPARATOR

Além da ordem natural, também é possível definir regras diferentes de ordenação, neste caso usar a interface **Comparator** do pacote **java.util**

A regra de ordenação é definida no método **int compare (Object o1, Object o2)**

- retorna 0 (zero) se o primeiro objeto passado é igual ao segundo
- retorna -1 se o primeiro objeto está “antes” do segundo
- retorna 1 se o primeiro objeto está “depois” do segundo

# INTERFACE COMPARATOR: EXEMPLO

```
import java.util.*;

public class ComparatorCpf implements Comparator<Cpf> {

    @Override

    public int compare(Cpf obj1, Cpf obj2) {

        return obj1.compareTo(obj2);

    }

}
```

# INTERFACE COMPARATOR: EXEMPLO

```
public class Cpf implements Comparable<Cpf>{  
    //...  
    @Override  
    public int compareTo(Cpf objCpf) {  
        if(getNumero() == objCpf.getNumero() )  
            return 0;  
        else if(getNumero() < objCpf.getNumero())  
            return -1;  
        return 1;  
    }  
}
```

**Coleção vai  
usar o número  
do Cpf para  
ordenar os  
elementos**

# USANDO COMPARATOR

```
public class Teste{  
    main(...){  
        ComparatorCpf comparador = new ComparatorCpf();  
        TreeSet<Cpf> conjunto = new TreeSet(comparador);  
        conjunto.add(new Cpf(5,5));  
        conjunto.add(new Cpf(2,2));  
        System.out.println(conjunto);  
    }  
}
```

**Obs.: Crie a classe ComparatorCpf usando genéricos!!!**

**Coleção vai usar o comparador definido - neste caso é o cpf também**

# COMPARABLE X COMPARATOR

**Comparable** - fazer a ordenação utilizando apenas o critério de ordem natural

**Comparator** - ordenar a coleção por mais de um critério, como por exemplo, por nome, por matrícula

É possível combinar o uso das duas interfaces para permitir a ordenação natural ou a baseada em comparadores

GENÉRICOS

# GENÉRICOS

Incluída a partir da versão J2SE 5.0

Possibilita criar elementos com tipos **parametrizáveis**

Esses tipos são verificados em tempo de compilação

Elimina a necessidade de uso de **cast** - o compilador conhece o tipo do elemento, ele pode verificar se o mesmo está sendo usado corretamente e pode inserir *casts* corretamente

Durante a compilação as variáveis de tipo são **apagadas**, e ocorre uma tradução para código Java tradicional com os tipos e *casts* adequados

# GENÉRICOS

Programação genérica pode ser feita usando: herança ou **variáveis de tipo**

Uma **classe genérica** terá uma ou mais **variáveis de tipo**

O uso de variáveis de tipo torna o código mais seguro e simples de ler

```
public class Arquivo<T>{ ...}
```





# GENÉRICOS: CONVENÇÃO DE NOMES

Nome da variável de tipo	Significado
E	Elemento
K	Chave
V	Valor
T	Tipo genérico
S,U	Tipos Adicionais

# GENÉRICOS: VANTAGENS

Verificação de tipos em tempo de compilação

Eliminação de conversão de tipos (casts)

Programação de códigos genéricos seguros em relação à tipagem e mais legíveis

# GENÉRICOS: ONDE USAR?

Os genéricos podem ser usados em:

- classes, interfaces
- nos tipos de dado dos atributos
- nos tipos de dado dos parâmetros de construtores e métodos
- nos tipos de dado dos retornos dos métodos

# GENÉRICOS: CRIANDO UM TIPO GENÉRICO

```
public class Par <P, S>{  
    private P primeiro;  
    private S segundo;  
    public Par(){}  
    public Par(P p, S s){  
        primeiro = p; segundo = s;  
    }  
}
```

Classe Genérica com dois  
parâmetros: P e S

Parâmetros do  
construtor são genéricos

P - é o tipo de retorno do método é genérico

```
public P getPrimeiro(){return primeiro;}
```

P - é o tipo do parâmetro do método genérico

```
public void setPrimeiro(P p){primeiro= p;}
```

```
...
```

```
}
```

# GENÉRICOS: USANDO CLASSE GENÉRICA

```
public class Teste {  
    public static void main(String[] args) {  
        Par<Integer, String> funcionario = new Par<>();  
        funcionario.setPrimeiro(1);  
        funcionario.setSegundo("Fulano");  
        System.out.println(funcionario.toString());  
        funcionario = new Par(2, "Beltrano");  
        System.out.println(funcionario.toString());  
    }  
}  
//Integer corresponde ao P  
//String corresponde ao S
```