

13.Coleções Parte 2 - Maps e Classes Utilitárias

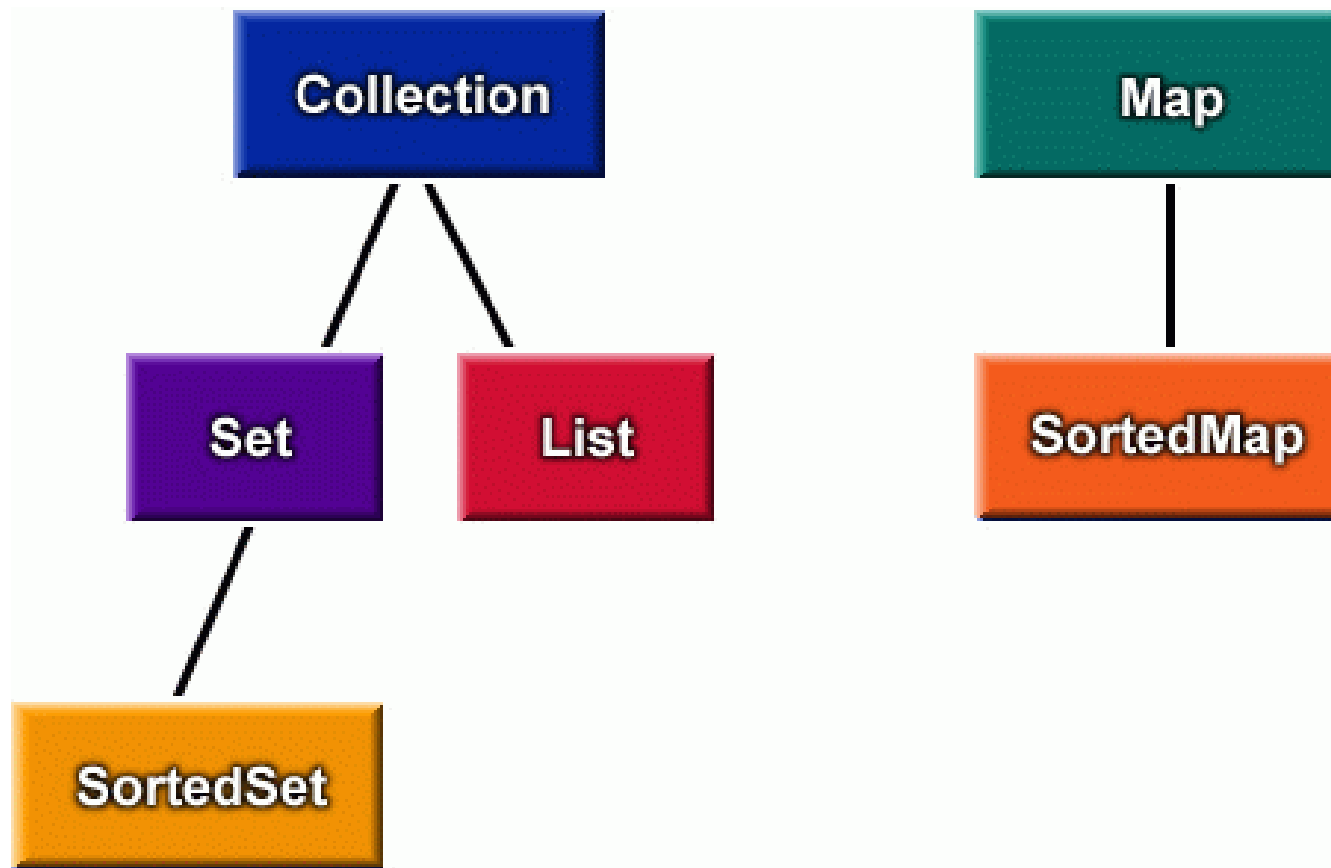
Prof. Alexandre Krohn



Roteiro

- Mapas (Maps)
- Classes Utilitárias
 - Collections
 - Arrays
 - Comparator / Comparable
 - Iterators

Retomando...



Diferença entre coleções e mapas

- **Coleções**


- Você pode adicionar, remover, pesquisar itens isolados na coleção

- **Mapas**

- As operações de coleções estão disponíveis, mas funcionam com um par de valores-chave em vez de um elemento isolado

Diferença entre coleções e mapas

- A interface **Collection** é um grupo de objetos, com elementos duplicados permitidos
- **Set** estende a **Collection**, mas proíbe duplicações
- **List** estende **Collection** e permite itens duplicados e indexação por posição
- **Map** não estende nem **Set** nem **List** nem **Collection**



Mapas (Maps)

- **Um mapa é um conjunto de pares**
 - Cada par representa um "mapeamento" unidirecional de um objeto (a chave) para outro (o valor)
- O uso típico de um mapa é fornecer acesso aos valores armazenados por chave
- Alguns **exemplos**
 - Um mapa de IDs de alunos para registros do banco de dados
 - Um dicionário (palavras mapeadas para significados)
 - A conversão da base 2 para a base 10
 - Nomes para números de telefone
 - Siglas de estados da federação



A interface Map

- Não implementa a interface Collection
- Mapeia chaves para valores (Vetor associativo)
- Assim como conjuntos, não permite que sejam inseridas chaves duplicadas (substituem valor anterior)
- Funciona como listas e vetores, mas usa uma chave ao invés de um índice para acesso
- Principais Implementações
 - HashMap, TreeMap and Hashtable

Mapas : definição

- Um Mapa é definido como um par:
- **Map** $\langle K, V \rangle$, onde
 - K é a **chave** de pesquisa
 - V é o **valor** armazenado
- K e V podem ser objetos
- K precisa implementar ***equals()*** e ***hashCode()***

Exemplos de Uso

- Colocando elementos no mapa:

```
Map<String,String> estados = new HashMap<>();
```

```
estados.put("RS", "Rio Grande do Sul");
```

```
estados.put("SC", "Santa Catarina");
```

```
estados.put("PR", "Paraná");
```

```
estados.put("SP", "São Paulo");
```

Exemplos de Uso

- Buscando dados do mapa:

```
System.out.println(estados.get("RS"));
```

```
System.out.println(estados.get("SC"));
```

```
String nomeDoEstado = estados.get("PR");
```

```
System.out.println(nomeDoEstado);
```

Exemplos de Uso

- Mapas podem conter objetos:

```
Map<String,Pessoa> alunos = new TreeMap<>();
```

```
alunos.put("1", new Pessoa(1,"Chaves"));  
alunos.put("2", new Pessoa(2,"Chiquinha"));  
alunos.put("3", new Pessoa(3,"Inhonho"));
```

```
Pessoa q = new Pessoa(4,"Quico");
```

```
alunos.put("quico", q);
```



Métodos de Map

- **Nenhum método da interface *Collection***
- **Básicos** : put, get, remove, containsKey, containsValue, size, isEmpty
- **Operações em lote** : putAll, clear
- **Visualização** : keySet, values, entrySet



Métodos Básicos

- `V put(K key, V value);` // associa o valor **V** a chave **K** nesse mapa
- `V get(Object key);` // obtém o objeto relacionado à chave **key**
- `boolean remove(Object key);` // remove do mapa o par associado à chave **key**
- `boolean containsKey(Object key);` // retorna **true** se o mapa contiver a chave **key**
- `boolean containsValue(Object value);` // retorna **true** se o mapa contiver o valor **value**
- `int size();` // retorna a quantidade de elementos presentes no mapa
- `boolean isEmpty();` // retorna **true** se o mapa estiver vazio

Métodos para Operações em Lote

- `void putAll(Map <K,V>);` // copia todos os valores do mapa informado no parâmetro para o mapa atual
- `void clear();` // elimina todos os pares armazenados no mapa (zera o mapa)



Métodos para Visualização

- `Set<K> keySet()`; // retorna o conjunto de chaves armazenados no mapa
- `Collection<V> values()`; // retorna a coleção de valores armazenados no mapa
- `Set<Map.Entry<K,V>> entrySet()`; // retorna o conjunto dos mapeamentos armazenados no mapa

Exemplos de Uso

- Listando todos os elementos em um mapa de Strings

```
Set<String> siglas = estados.keySet();  
for (String sigla : siglas) {  
    System.out.println(sigla + " -> " + estados.get(sigla));  
}
```



Exemplos de Uso

- Listando todos os elementos em um mapa de Objetos

```
Set<String> keys = alunos.keySet();  
for (String chave : keys) {  
    System.out.println(chave + " -> " + alunos.get(chave).getNome());  
}
```

Classes que implementam Map

- ***HashMap*** : Implementa um mapa simples
- ***Hashtable*** : também implementa um mapa simples, porém é mais antiga, sincronizada, e com performance pior
- ***TreeMap***: Implementa um mapa ordenado pelas suas chaves



Roteiro

- Mapas (Maps)
- **Classes Utilitárias**
 - Collections
 - Arrays
 - Comparator / Comparable
 - Iterator



Classes Utilitárias

- O framework Collections oferece uma série de classes para facilitar o uso de listas, conjuntos e mapas
- Essas classes implementam algoritmos comumente aplicados à coleções

Classes Utilitárias : Exemplos

```
String[] vetor = {"Phil", "Mary", "Betty", "Bob"};  
List<String> nomes = Arrays.asList(vetor);  
Collections.sort(nomes);  
System.out.println("Lista Ordenada: " + nomes);  
int pos = Collections.binarySearch(nomes, "Bob");  
System.out.println("Bob está na posição " + pos);  
Collections.shuffle(nomes);  
System.out.println("Bagunçada: " + nomes);
```

Executando :

```
Lista Ordenada: [Betty, Bob, Mary, Phil]  
Bob está na posição 1  
Bagunçada: [Phil, Betty, Mary, Bob]
```



Roteiro

- Mapas (Maps)
- Classes Utilitárias
 - Collections
 - Arrays
 - Comparator
 - Iterator



Collections

- ***Collections*** (no plural) é uma classe utilitária que realiza diversas operações sobre coleções:

Collections : Métodos

- `<T> T max(Collection<T>);` // retorna o valor máximo da coleção
- `<T> T min(Collection<T>);` // retorna o valor mínimo da coleção
- `void reverse(List<T> list);` // inverte a ordem de uma lista
- `void rotate(List<T> list, int distance);` // gira uma lista em direção ao fim "distance" posições
- `void shuffle(List<T> list);` // embaralha uma lista
- `void sort(List<T> list);` // ordena uma lista de acordo com a **ordem natural** dos elementos
- `void sort(List<T> list, Comparator<? super T> c);` // ordena uma lista **de acordo com o comparador** informado

Há mais métodos, aqui estão apenas os principais



Roteiro

- Mapas (Maps)
- Classes Utilitárias
 - Collections
 - **Arrays**
 - Comparator / Comparable
 - Iterator



Arrays

- A classe Arrays possui uma série de métodos para facilitar a utilização de vetores

Arrays : Métodos

- Em relação às coleções, o método mais importante da classe Arrays é:
 - `<T> List<T> asList(T[] a);` // retorna uma lista com os mesmos objetos contidos no vetor

Há mais métodos com operações específicas de vetores



Roteiro

- Mapas (Maps)
- Classes Utilitárias
 - Collections
 - Arrays
 - Comparator / Comparable
 - Iterator

Comparator / Comparable

- O ponto central dos **algoritmos de ordenação** é a definição do **critério** que diz que um objeto é maior ou menor do que outro

```
para i de 1 ate 5 faca
  para j de (i+1) ate 5 faca
    se a[i]>a[j] entao
      x<-a[i]
      a[i]<-a[j]
      a[j]<-x
    fimse
  fimpara
fimpara
```

Implementando comparações

- Há duas maneiras de “ensinar” à objetos sua ordem:
 - Fazendo o objeto implementar a interface **Comparable**
 - Utilizando um objeto que implemente a classe **Comparator**



Comparable X Comparator

- ***Comparable*** : Critério **natural** de ordenação dos objetos
- Algumas classes básicas, como Strings, Integer, Double, Float, já implementam essa interface:
 - **Números** são ordenados em **ordem crescente de valor**
 - **Strings** são ordenadas em **ordem alfabética**



Comparable

- Implementa-se o método ***compareTo***(Object o)
- Seleciona-se um atributo que defina o critério de ordenação implementa-se o seguinte algoritmo:
 - Retorne 0 se os valores forem iguais
 - Retorne -1 (ou valor negativo) se o objeto informado tiver o valor menor
 - Retorne 1 (ou valor positivo) se o objeto informado tiver o valor maior

Comparable : Exemplo

- Ordenar pessoa por código:

```
public class Pessoa implements Comparable{  
  
    private int codigo;  
    private String nome;  
  
    @Override  
    public int compareTo(Object o) {  
        Pessoa outra = (Pessoa) o;  
        return this.codigo - outra.codigo;  
    }  
}
```



```
Collections.sort(pessoas);
```

Comparable : Exemplo

- Ordenar pessoa por nome:

```
public class Pessoa implements Comparable{  
  
    private int codigo;  
    private String nome;  
  
    @Override  
    public int compareTo(Object o) {  
        Pessoa outra = (Pessoa) o;  
        return this.nome.compareTo(outra.nome);  
    }  
}
```



```
Collections.sort(pessoas);
```



Comparator

- ***Comparator*** : define critérios alternativos para ordenação, permitindo que uma mesma classe possa ser ordenada por mais do que um atributo.



Comparator

- Para criar um comparador, cria-se uma classe que implementa a interface ***Comparator*** e implementa-se o método ***compare(...,....)***

Comparator : Exemplo

- Comparando pessoas por nome:

```
import java.util.Comparator;

public class PessoaNomeComparator implements Comparator<Pessoa> {

    @Override
    public int compare(Pessoa o1, Pessoa o2) {
        return o1.getNome().compareTo(o2.getNome());
    }
}
```



```
Collections.sort(pessoas, new PessoaNomeComparator());
```

Comparator : Exemplo

- Comparando pessoas por código:

```
import java.util.Comparator;

public class PessoaCodigoComparator implements Comparator<Pessoa> {

    @Override
    public int compare(Pessoa o1, Pessoa o2) {
        return o1.getCodigo() - o2.getCodigo();
    }
}
```



```
Collections.sort(pessoas, new PessoaCodigoComparator());
```

Comparação por mais do que um atributo

```
import java.util.Comparator;

public class PessoaNomeCodigoComparator implements Comparator<Pessoa> {

    @Override
    public int compare(Pessoa o1, Pessoa o2) {

        // Testa a ordem dos nomes
        int compNome = o1.getNome().compareTo(o2.getNome());

        if(compNome!=0) {
            return compNome;
        }

        // Se os nomes são iguais, então compara por código
        int compCodigo = o1.getCodigo() - o2.getCodigo();
        return compCodigo;
    }
}

Collections.sort(pessoas, new PessoaNomeCodigoComparator());
```



Roteiro

- Mapas (Maps)
- Classes Utilitárias
 - Collections
 - Arrays
 - Comparator / Comparable
 - **Iterator**



Iterator

- Podemos percorrer uma lista usando um laço for, mas ao fazer isso **não é possível excluir elementos.**



Iterator

- Iterator fornece um meio de percorrer a lista
- Permite a remoção de elementos, se desejado
- Pode avançar e/ou retroceder na lista



Iterator : Métodos

- boolean **hasNext**(); // retorna **true** se há um objeto após o objeto atual
- E **next**(); // retorna o próximo objeto da lista
- int **nextIndex**(); // retorna o índice do próximo objeto da lista
- boolean **hasPrevious**(); // retorna **true** se há um objeto antes do objeto atual
- E **previous**(); // retorna o objeto anterior da lista
- int **previousIndex**(); // retorna o índice do objeto anterior da lista
- void **remove**(); // remove o elemento da lista na posição atual do *Iterator*

Iterator : Exemplo

```
// Exibe todas as pessoas, mas elimina a pessoa com
// código igual a 3
Iterator<Pessoa> it = pessoas.iterator();
while(it.hasNext()) {
    Pessoa atual = it.next();
    if(atual.getCodigo()==3) {
        it.remove();
    }
    System.out.println(atual);
}
```




Dúvidas?





Atividades

- Execute as atividades presentes no documento

13.Lista.de.Exercícios.POO.pdf

Próximos passos



- Arquivos e Serialização