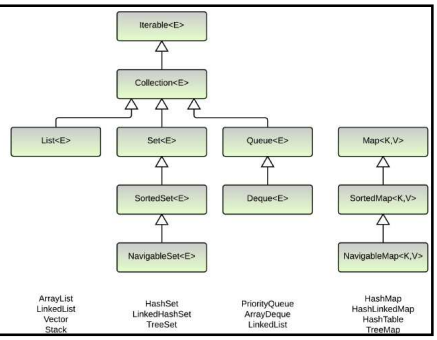
Programação para Web I
Aula 2

Profa. Silvia Bertagnolli

Coleções

Hierarquia de Interfaces



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

Exercícios

Exercícios

1 - Crie uma coleção com números do tipo Double

1.1 Ordene os números da coleção criada no item 1 usando a classe Collections

1.2 Modifique a ordem dos elementos da coleção criada no item 1 usando um método da classe Collections

1.3 Leia um número e verifique se ele está na coleção criada no item 1

Exercícios

2 - Usando a classe String adicionar 5 palavras na lista, ordenar a lista; após imprimir essa lista

3 - Usando a classe Pessoa da aula anterior fazer:

3.1 Adicionar 4 pessoas em uma lista, usar os construtores sem e com parâmetros

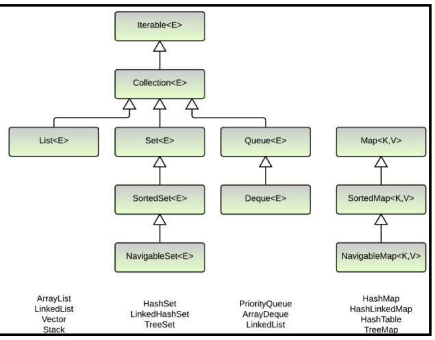
3.2 Imprimir o primeiro e o último objetos

3.3 Ordenar a lista usando o nome

3.4 Imprimir a lista ordenada

Coleções

Hierarquia de Interfaces



Mapas

Interface Map

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

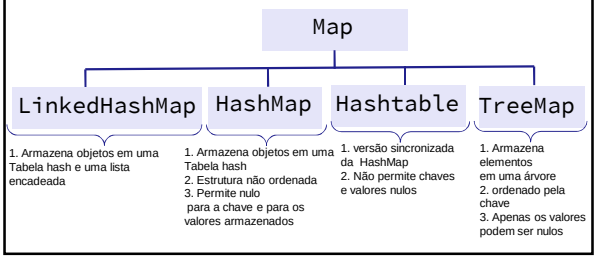
Não contém chaves duplicadas e mapeia chaves K para valores V

Uma chave (única e exclusiva) é 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 das Classes de Mapas



Interface Map Principais Métodos

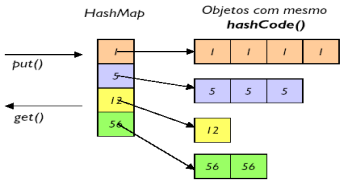
public Object get(Object key) – Retorna o objeto identificado pela chave

public Object put(Object value, Object key) – Coloca 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



Fonte: ROCHA, 2001

HashMap: exemplo

```
HashMap<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

```
Hashtable<String, Integer> itens = new Hashtable<>();
itens.put("cod1", new Integer(1));
   //...
Enumeration<Integer> e= itens.elements();
while(e.hasMoreElements()){
      Integer i = (Integer) e.nextElement();
      System.out.println(i);
}
```

TreeMap

É um mapa ordenado pela ordem natural dos elementos

Permite definir as próprias regras de comparação 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?

```
LinkedHashMap<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?

```
LinkedHashMap<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?

```
LinkedHashMap<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 + " - valor: "+ map.get(chave));
}
```

Exercícios

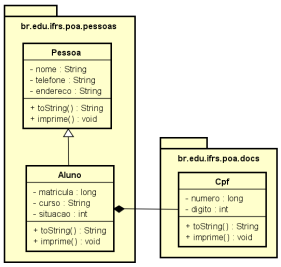
Exercícios

9 Usando as classes ao lado faça

9.1 Adicionar 4 cpfs em um TreeMap usando os construtores sem e com parâmetros – a chave é o número do Cpf e o valor é um objeto Cpf

9.2 Ordenar o mapa usando o número e o digito do Cpf

9.3 Imprimir o mapa ordenado



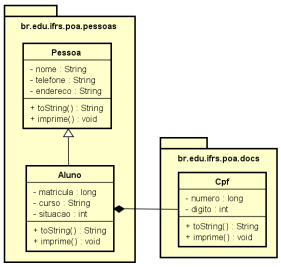
Exercícios

10 Usando as classes ao lado faça

10.1 Adicionar em um TreeMap alunos usando os construtores sem e com parâmetros, onde a chave é o Cpf e o valor um objeto aluno

10.2 Ordenar o mapa usando o Cpf

10.3 Imprimir o mapa ordenado



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 para esse tipo de coleção

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{
   //...
   @Override
   public int compareTo(Cpf o) {
     if(getNumero() == o.getNumero())
          return 0;
     else if(getNumero() < o.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 digito do Cpf forem iguais

Ordenando Coleções: Interface Comparable

Obs.:

a coleção HashMap usa o método equals() para determinar se dois objetos são iguais, enquanto a coleção TreeSet usa o compareTo()

Se esses métodos não são coerentes, cada tipo de coleção pode apresentar um comportamento diferente

Recomendação: manter coerentes os métodos equals(), hashcode() e compareTo()

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

Exercícios

Exercícios

Refaça o Exercício 3 de modo que o critério de ordenação seja o nome da Pessoa

Refaça o Exercício 10 de modo que o critério de ordenação seja o número do Cpf

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

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 o) {
     if(getNumero() == o.getNumero() )
        return 0;
     else if(getNumero() < o.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<>(c);
        conjunto.add(new Cpf(...));
     }
}
```

Obs.: Crie a classe ComparatorCpf!!!

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: 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;
  }
  public P getPrimeiro(){return primeiro;}
  public S getSegundo(){return segundo;}
  public void setPrimeiro(P p){primeiro= p;}
  public void setSegundo(S s){segundo = s;}
  public String toString(){ return primeiro + " " + segundo;}
}
```

Classe Genérica - com dois parâmetros

Parâmetros do construtor são genéricos

Tipo de retorno do método é genérico

Genéricos: usando classe genérica

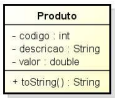
```
public class Teste {
   public static void main(String[] args) {
         Par<Integer, String> funcionario = new Par<>();
         funcionario.setPrimeiro(1);
         funcionario.setSegundo("Fulano");
         System.out.println(funcionario.toString());
         funcionario = new Par(2, "Beltrano");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

Exercícios

Exercícios

12. Use a classe Par definida anteriormente para armazenar:
- O nome e a nota de um aluno (<String, Double>)
- As coordenadas x e y (<Float, Float>)

Crie a classe Produto abaixo:



Exercícios

Crie uma classe genérica Codigo para representar os códigos dos Produtos
   códigos são compostos por duas partes que indicam o setor onde o produto é fabricado (setor pode ser String ou número); e números que são um código sequencial dentro do setor
   Exemplos válidos: **IMM**120; **IMM**121 - **111**120; **111**121)

Modifique a classe Produto para que o seu código seja criado usando a classe Codigo acima

Agora, monte a classe de Testes que deve criar objetos do tipo Produto com a modificação solicitada acima

Exercícios

Crie a interface genérica Lista como descrito abaixo:
- A lista irá conter elementos
- A lista está definida no pacote br.edu.ifrs.progweb2.util;
- Defina os métodos abaixo usando genéricos onde for possível:

```
public void adicionar(Object obj)
public boolean remover(int i)
public String listar()
public int totalizar()
public void removerTodos()
public Object pegarElemento(int)
public void removerElemento(Object obj)
```

Exercícios

Agora, crie a classe **MinhaLista** que implementa a interface **Lista** genérica definida previamente

Essa classe deve declarar um atributo da classe LinkedList para armazenar objetos na classe **MinhaLista** e implemente todos os métodos da interface **Lista**

Monte uma classe de testes para armazenar objetos da classe Produto (que tem composição com a classe Codigo)

Exercícios

Crie a interface genérica **Mapa** como descrito abaixo:
- O mapa irá conter elementos compostos por chave e valor
- O mapa está definido no pacote br.edu.ifrs.progweb2.util;
- Defina os métodos abaixo usando o conceito de genéricos onde for possível:

public Objec get(Object obj) retorna o valor de uma chave especifica
public boolean isEmpty() retorna true se o mapa está vazio
public Set keySet() retorna um conjunto com todas as chaves
public int size() retorna o nnúmero de objetos no mapa
public void put(Object chave, Object valor) - adiciona um objeto no mapa usando a chave e o valor
public void remove (Object obj) - remove um objeto usando a sua chave
public List values() - retorna uma lista com os valores armazenados no mapa

Exercícios

Agora, crie a classe **MeuMapa** que implementa a interface **Mapa** genérica definida previamente

Essa classe deve declarar um atributo da classe LinkedList para armazenar objetos na classe MeuMapa e implemente todos os métodos da interface Mapa

Monte uma classe de testes para armazenar objetos da classe String como chave e Double como valor