Programação Orientada por Objectos

Introdução à Coleções - Maps

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática

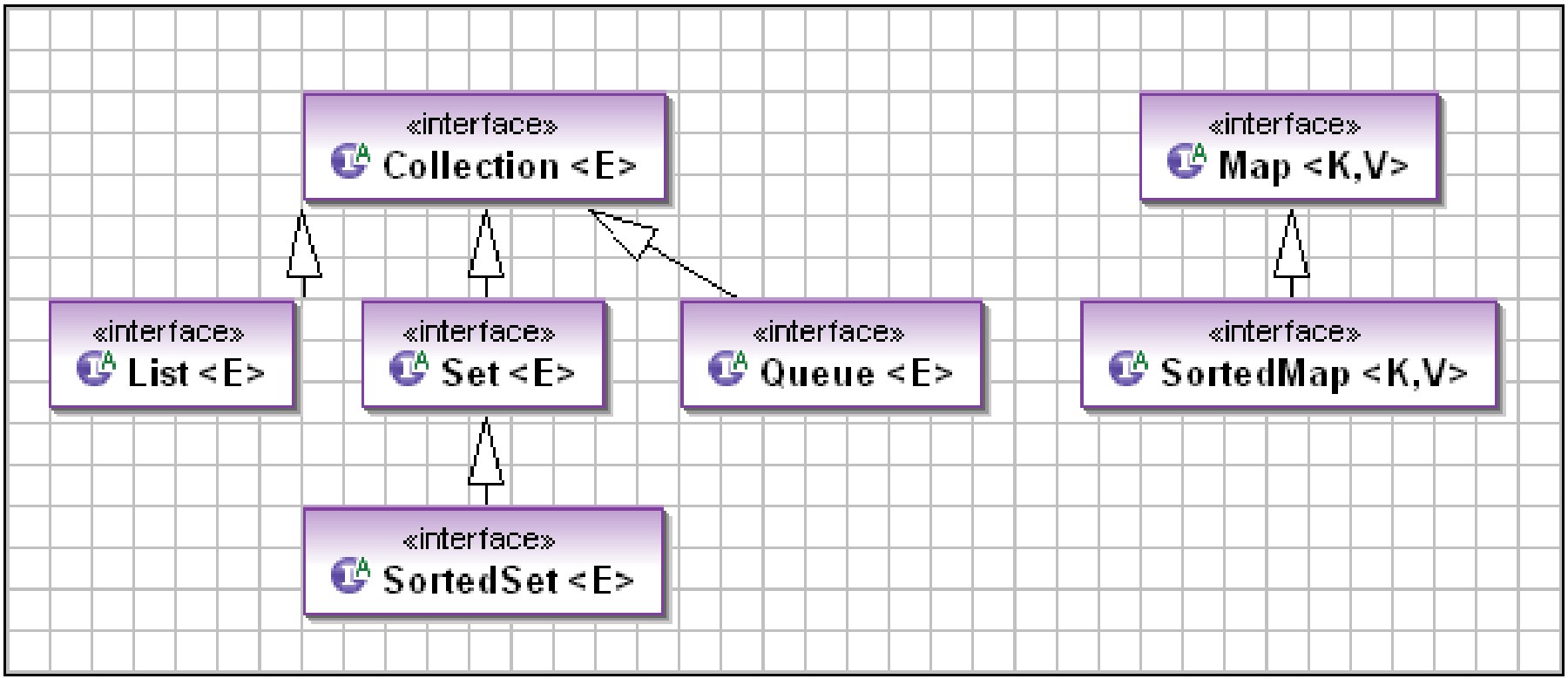
Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2014/2015

Sumário

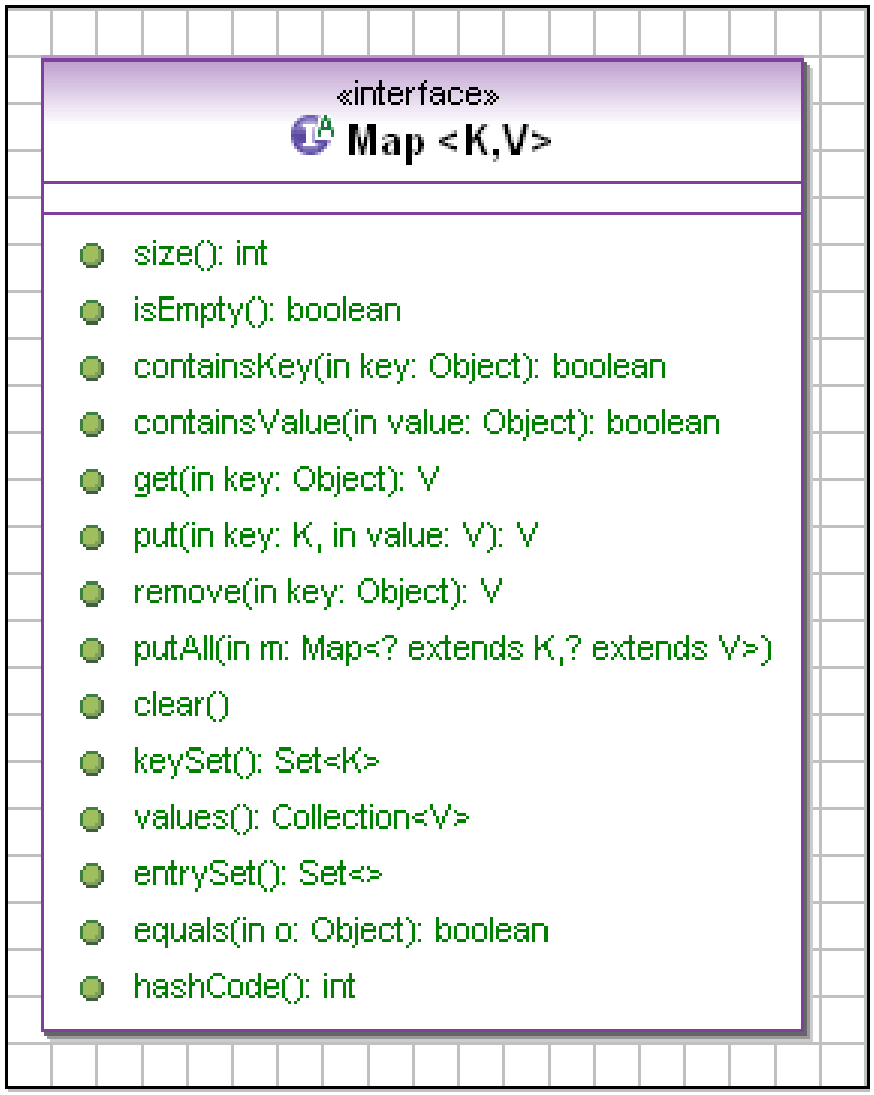
- Mapas
  - Interface **`Map`**
    - Classe **`HashMap<K,V>`**
    - Classe **`TreeMap<K,V>`**
    - Classe **`LinkedHashMap<K,V>`**
- Acesso aos elementos dos mapas
- Exemplo de manipulação de mapas
- Ordenação da informação
- Interface **`Comparable<T>`**

Java Collections Framework – Interfaces Genéricas



Coleções - Interface Map

- Interface que impõe uma relação entre uma chave (*Key*) e um elemento (*Value*).
  - As chaves são únicas (não há chaves repetidas!)
  - Cada chave refere um único elemento.
  - Note que pode haver valores repetidos, mas com diferentes chaves.



Coleções – Classes Map

- Interface **`Map`**
  - **`HashMap<K,V>`**
    - A única a ser estudada detalhadamente
    - **`HashMap<Integer, Pessoa> pessoas = new HashMap<>();`**
      - Cria um objeto **`HashMap`** chamado **`pessoas`** em que as chaves são do tipo inteiro e os valores são objetos da classe **`Pessoa`**.
    - O seu comportamento e desempenho é em tudo semelhantes ao conjunto análogo **`HashSet`**: também não temos garantia quanto à ordem de iteração e usa **`hash table`** na sua implementação (recorrendo ao **`equals`** e **`hashCode`** para determinar a unicidade).
  - Outras implementações de mapas (que mantêm a ordem de iteração):
    - **`TreeMap`**
    - **`LinkedHashMap`**

Utilização de Maps

- A interface **`Map`**, tal como a **`Set`**, não associa a cada elemento uma posição dentro da coleção, como acontece na interface **`List`**.
- Assim a melhor forma para percorrer os elementos de um mapa é através do ciclo **`For-Each`** ou de um **`iterator`**.
- No entanto como no mapa temos uma associação chave/valor podemos percorrer os seus elementos de várias formas distintas:
  - Através do acesso às diversas chaves
  - Através do acesso exclusivo aos valores
  - Através do acesso aos pares chave/valor
- Devemos manter a estratégia de declararmos as variáveis utilizando a interface **`Map`** e definir os seus valores através de uma implementação concreta (ex.: **`HashMap`**)

Aceder aos elementos de um Mapa

- Supondo que se cria um mapa que associa números (**`Integer`** – as classes genéricas não podem trabalhar com tipos primitivos – **`int`**: é preciso utilizar as respetivas classes equivalentes) a nomes de pessoas (**`String`**):

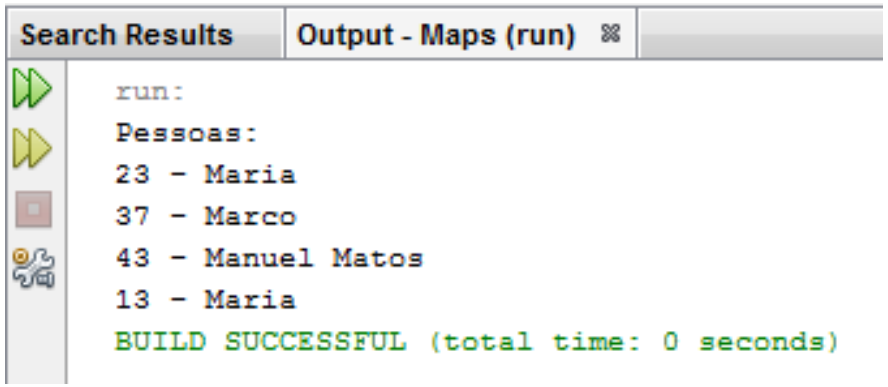
```
public static void main(String[] args) {
  Map<Integer, String> mapaNomes = new HashMap<>();
  mapaNomes.put(13, "Maria");
  mapaNomes.put(43, "Manuel");
  mapaNomes.put(37, "Marco");
  mapaNomes.put(23, "Maria"); //Valor repetido
  mapaNomes.put(43, "Manuel Matos"); //Chave repetida

  printMap(mapaNomes);
}
```

Aceder aos elementos através da chave

- A interface **`Map`** tem o método **`keySet()`** que devolve um **`Set<K>`** com todas as chaves. Assim, recorrendo ao **`For-Each`** ou a um **`iterator`** (aplicados ao conjunto resultante) podemos aceder a todos os elementos:

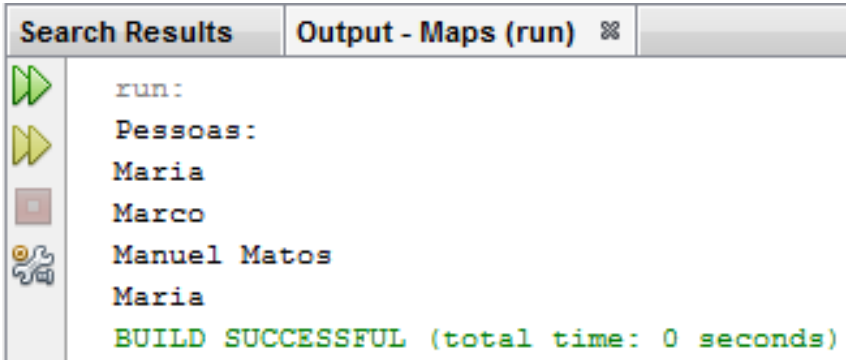
```
public static void printMap(Map<Integer, String> mapa) {
  System.out.println("Pessoas:");
  for (Integer i : mapa.keySet()) {
     System.out.format("%d - %s\n", i, mapa.get(i));
  }
}
```



Aceder aos elementos através dos valores

- A interface **`Map`** tem o método **`values()`** que devolve uma **`Collection<V>`** com todos os valores (sem as chaves). Assim, recorrendo ao **`For-Each`** ou a um **`iterator`** (aplicados à coleção resultante) podemos aceder a todos os elementos:

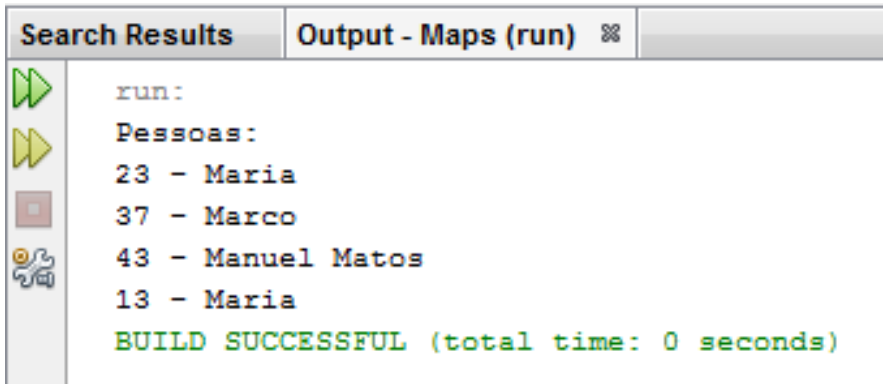
```
public static void printMap(Map<Integer, String> mapa) {
  System.out.println("Pessoas:");
  for (String nome : mapa.values()) {
     System.out.println(nome);
  }
}
```



Aceder aos elementos através dos pares chave/valor

- A interface **`Map`** tem o método **`entrySet()`** que devolve um **`Set<Map.Entry<K,V>>`** com todas os pares chave/valor (implementados através da classe **`Map.Entry`**). Assim, recorrendo ao **`For-Each`** ou a um **`iterator`** (aplicados ao conjunto resultante) podemos aceder a todos os elementos:

```
public static void printMap(Map<Integer, String> mapa) {
  System.out.println("Pessoas:");
  for (Map.Entry par : mapa.entrySet()) {
     System.out.format("%d - %s\n",par.getKey(),par.getValue());
  }
}
```



Exemplo do uso de Mapas

- Criar um sistema para registar as notas de alunos.
- Serão necessárias as seguintes classes:
  - **`Aluno`** que representa a informação de um aluno
  - **`Turma`** que conterá a informação relativa a todos os alunos
  - **`Avaliacao`** onde se associam os alunos às notas

Classe Aluno

- Para simplificar, um aluno será representado por um número (atributo **`numero`**) e o seu nome (atributo **`nome`**);
- O número é gerido automaticamente pelo sistema (através do atributo, de classe, **`proximoNumero`**), começando em 1;
- Desta forma, apenas será necessário indicar o nome do aluno no construtor;
- É possível aceder ao número (**`getNumero`**) e nome (**`getNome`**) de um aluno. Sendo apenas possível alterar o seu nome (**`setNome`**);
- O que garante a unicidade do aluno é o seu número. Assim o **`hashCode`** e o **`equals`** apenas analisarão o atributo **`numero`**;
- Um **`Aluno`** será representado pelo seu número e nome, separados por um hífen (**`toString`**).
- Por simplificação, não estão a ser feitas validações dos valores dos argumentos.

Classe Aluno

```
public class Aluno {

    private static int proximoNumero = 1;
    private int numero;
    private String nome;

    public Aluno(String nome) {
        this.numero = Aluno.proximoNumero++;
        this.nome = nome;
    }
```

Classe Aluno

```
public int getNumero() {
    return numero;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}
```

Classe Aluno

```
@Override
public int hashCode() {
    //return numero;
    Integer numero = new Integer(this.numero);
    return numero.hashCode();
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    return this.numero == ((Aluno)obj).numero;
}
```

Classe Aluno

```
    @Override
    public String toString() {
        return numero + " - " + nome;
    }

}
```

Classe Turma

- Uma turma poderia ser implementada através de um conjunto (**`Set`**) de alunos, pois não pode haver alunos repetidos;
- No entanto para aceder aos alunos através do seu número será mais eficiente criar uma associação número/aluno (**`Map`**): o acesso será imediato em oposição à necessidade de percorrer os elementos do conjunto para encontrar o aluno com o número indicado;
- É preciso especial cuidado na criação desta associação número/aluno para não se criar uma inconsistência entre o número utilizado na associação e o número real do aluno: o problema está minimizado por não se poder alterar o número ao aluno e como a associação será privada à classe, temos o controlo total sobre a sua criação;
- Assim, a classe **`Turma`** terá um atributo (**`alunos`** - **`private`**) que fará a associação número/aluno e um atributo, fornecido no construtor, que representará o nome da turma (**`nome`**);

Classe Turma

- O acesso aos alunos é feito através do seu número:
  - **`void add(Aluno aluno)`** – acrescenta um aluno, associando-o ao seu número;
  - **`Aluno get(int numero)`** – devolve o aluno associado ao número (ou **`null`** caso ele não exista na turma);
  - **`Aluno remove(int numero)`** – remove o aluno com o número;
  - **`boolean inscrito(int numero)`** – verifica se o aluno com o número está na turma;
- Uma turma será representada (**`toString`**) pelo seu nome seguido da listagem dos alunos, em linhas separadas;
- Por simplificação, não estão a ser feitas validações dos valores dos argumentos.

Classe Turma

```
public class Turma {
    private Map<Integer, Aluno> alunos;
    private String nome;

    public Turma(String nome) {
        this.nome = nome;
        alunos = new HashMap<>();
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
```

Classe Turma

```
public void add(Aluno aluno) {
    alunos.put(aluno.getNumero(), aluno);
}

public Aluno get(int numero) {
    return alunos.get(numero);
}

public Aluno remove(int numero) {
    return alunos.remove(numero);
}

public boolean inscrito(int numero) {
    return alunos.containsKey(numero);
}
```

Classe Turma

```
    @Override
    public String toString() {
        String resultado = nome + ":";
        for (Aluno aluno : alunos.values()) {
            resultado += "\n" + aluno;
        }
        return resultado;
    }
}
```

Classe Avaliacao

- Uma avaliação são simples associações entre alunos e notas;
- No caso mais simples, não fazendo qualquer tipo de validação, uma nota será um número inteiro (**`Integer`** para poder ser utilizado numa classe genérica);
- Assim sendo, a avaliação será implementada através de um **`HashMap<Aluno, Integer>`**, sendo uma simples extensão desta classe;
- Ao introduzir uma nota de um aluno já existente, a nova nota substituirá a anterior (não está implementada a regra de substituir só se for a nota for maior que a anterior. Para tal, teríamos que redefinir o método **`put`**);
- Para tornar compreensível a visualização de todas as notas atribuídas, será redefinido o método **`toString`** que apresentará cada nota à frente do respetivo aluno, uma por linha.

Classe Avaliacao

```
public class Avaliacao extends HashMap<Aluno, Integer> {

    @Override
    public String toString() {
        String resultado = "Notas:";
        for (Aluno aluno : keySet()) {
            resultado += "\n" + aluno + ": " + get(aluno);
        }
        return resultado;
    }
}
```

Programa Principal

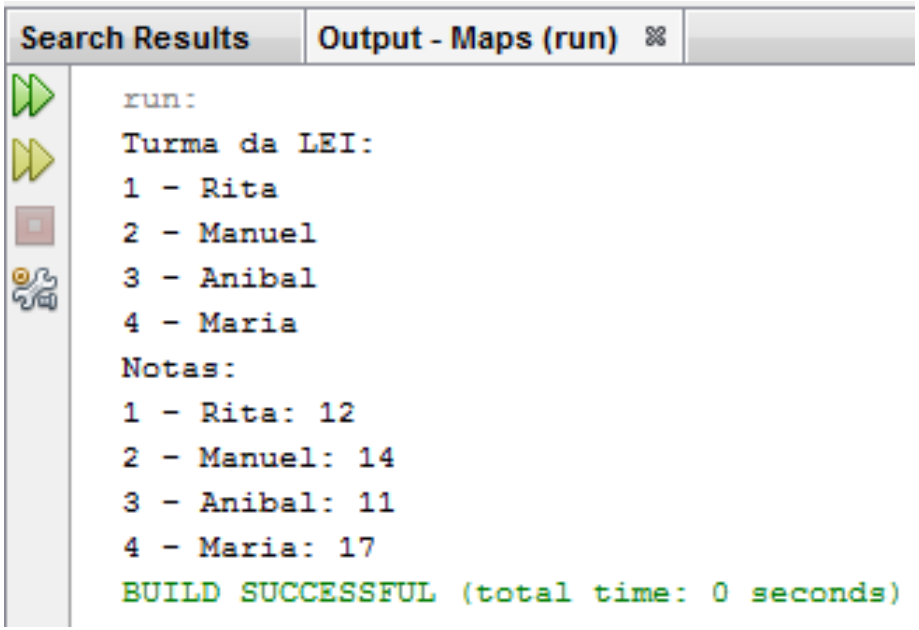
- Com estas três classes (**`Aluno`**, **`Turma`** e **`Avaliacao`**) será possível simular o funcionamento de uma turma.
- Para simplificar é criada uma turma com quatro alunos:
  - Rita;
  - Manuel;
  - Anibal;
  - Maria;
- E são lançadas as respetivas notas:
  - Rita com 12;
  - Manuel com 14;
  - Anibal com 8 mas mais tarde repetiu a avaliação e ficou com 11;
  - Maria com 17.

Programa Principal

```
public static void main(String[] args) {
   Turma turma = new Turma("Turma da LEI");
   turma.add(new Aluno("Rita"));   //Fica com nº 1
   turma.add(new Aluno("Manuel")); //Fica com nº 2
   turma.add(new Aluno("Anibal")); //Fica com nº 3
   turma.add(new Aluno("Maria"));  //Fica com nº 4
   System.out.println(turma);
   Avaliacao notas = new Avaliacao();
   notas.put(turma.get(1),12); //Rita
   notas.put(turma.get(2),14); //Manuel
   notas.put(turma.get(3),8);  //Anibal
   notas.put(turma.get(4),17); //Maria
   notas.put(turma.get(3),11); //Recurso do Anibal
   System.out.println(notas);
}
```

Programa Principal

- Por mero acaso, os valores aparecem ordenados por número (os **`HashMap`** não garantem nenhuma ordem na iteração dos seus valores):



Ordenação dos elementos de uma coleção

- Poderia ser interessante apresentar as notas dos alunos ordenando-os alfabeticamente pelo seu nome
- Como foi referido, no início do estudo das coleções, existem diversos métodos, de classe, que permitem a manipulação dos elementos de uma coleção. Para ordenar será necessário utilizar o método **`Collections.sort`**
- A ordenação é feita comparado os elementos da coleção: dados dois elementos é preciso saber se um é "menor", "igual" ou "maior" que o outro
- Existem então três valores possíveis nessa comparação.
- A forma mais simples de a fazer é recorrer a um método que devolva um valor negativo em caso de "menor", um valor positivo em caso de "maior" e zero em caso de igualdade.

Interface Comparable

- Esta necessidade de comparar elementos é tão importante e comum que o Java disponibiliza a interface:

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- A interface **`Comparable`** apenas obriga à implementação do método **`compareTo`** que recebe um elemento do mesmo tipo e devolve um valor inteiro positivo, negativo ou zero, consoante o elemento a comparar seja maior, menor ou igual ao elemento fornecido.
- Todas as classes que pretendem ordenar os seus objetos devem implementar esta interface, indicando no método **`compareTo`** o algoritmo de comparação.

Ordenação por nome dos Alunos

- Para os alunos poderem ser ordenados por nome é preciso que a classe **`Aluno`** implemente a interface **`Comparable<Aluno>`**:

```
public class Aluno implements Comparable<Aluno> {
    ...
}
```

- E que seja implementado o método **`compareTo`** (não é verificado se o **`aluno`** é diferente de **`null`** e é utilizada a chamada ao método **`compareTo`** da classe **`String`**, que implementa a interface **`Comparable<String>`**):

```
@Override
public int compareTo(Aluno aluno) {
    return nome.compareTo(aluno.nome);
}
```

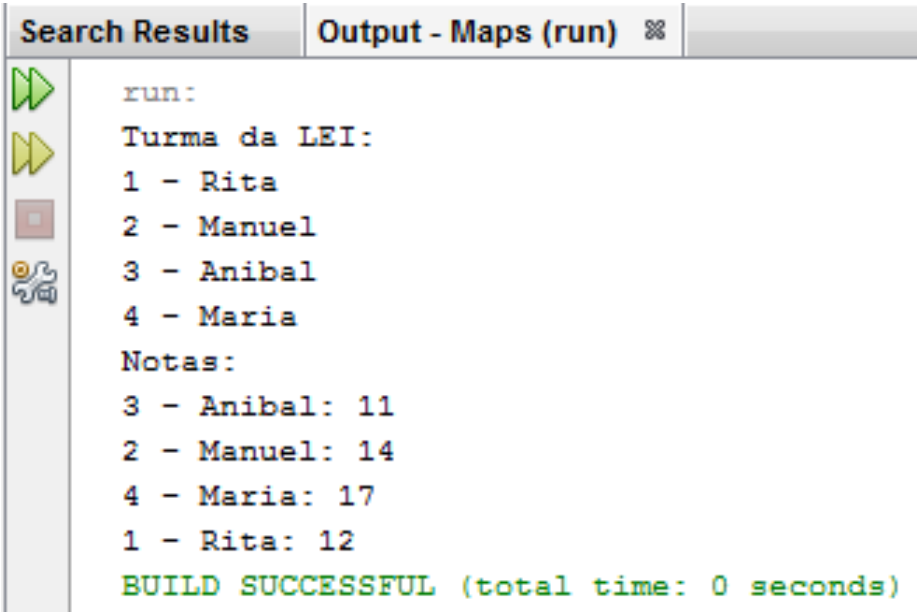
Ordenação na apresentação das notas

- Para produzir uma pauta ordenada (será necessário modificar o **`toString`** de **`Avaliacao`**) é preciso recolher os alunos numa lista e ordená-la (por nome), utilizando-a, de seguida, para obtenção das notas:

```
@Override
public String toString() {
  List<Aluno> alunos = new ArrayList<>(keySet());
  Collections.sort(alunos);
  String resultado = "Notas:";
  for (Aluno aluno : alunos) {
     resultado += "\n" + aluno + ": " + get(aluno);
  }
  return resultado;
}
```

Pauta Ordenada

- As notas são apresentadas com os alunos ordenados por nome:



Resumindo

- **`Map`** – interface que representa associações chave/valor. Não permitindo chaves repetidas (podem haver valores repetidos, desde que associados a chaves distintas)
- **`HashMap`** – implementação baseada em **`hash table`**
- **`LinkedHashMap`** e **`TreeMap`** – implementações que garantem a ordem de iteração
- Utilizar **`For-Each`** ou **`iterator`** para percorrer os elementos, através das:
  - Chaves – **`keySet()`**
  - Valores – **`values()`**
  - Pares chave/valor – **`entrySet()`**
- A interface **`Comparable<T>`**, através do seu método **`compareTo`**, permite a comparação de objetos (devolvendo <0, >0 ou =0, consoante os valores sejam menores, maiores ou iguais)
- O uso do método **`Collection.sort`** permite a ordenação dos elementos

Leitura Complementar

- Capítulo 8
  - Páginas 253 a 344

- http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html