

Programação Orientada por Objetos

Coleções

Prof. José Cordeiro,

Prof. Cédric Grueau,

Prof. Laercio Júnior

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2019/2020

- ❑ Sessão 1: Coleções e Listas
- ❑ Sessão 2: Conjuntos
- ❑ Sessão 3: Mapas
- ❑ Sessão 4: Exemplo Escola, Algoritmos



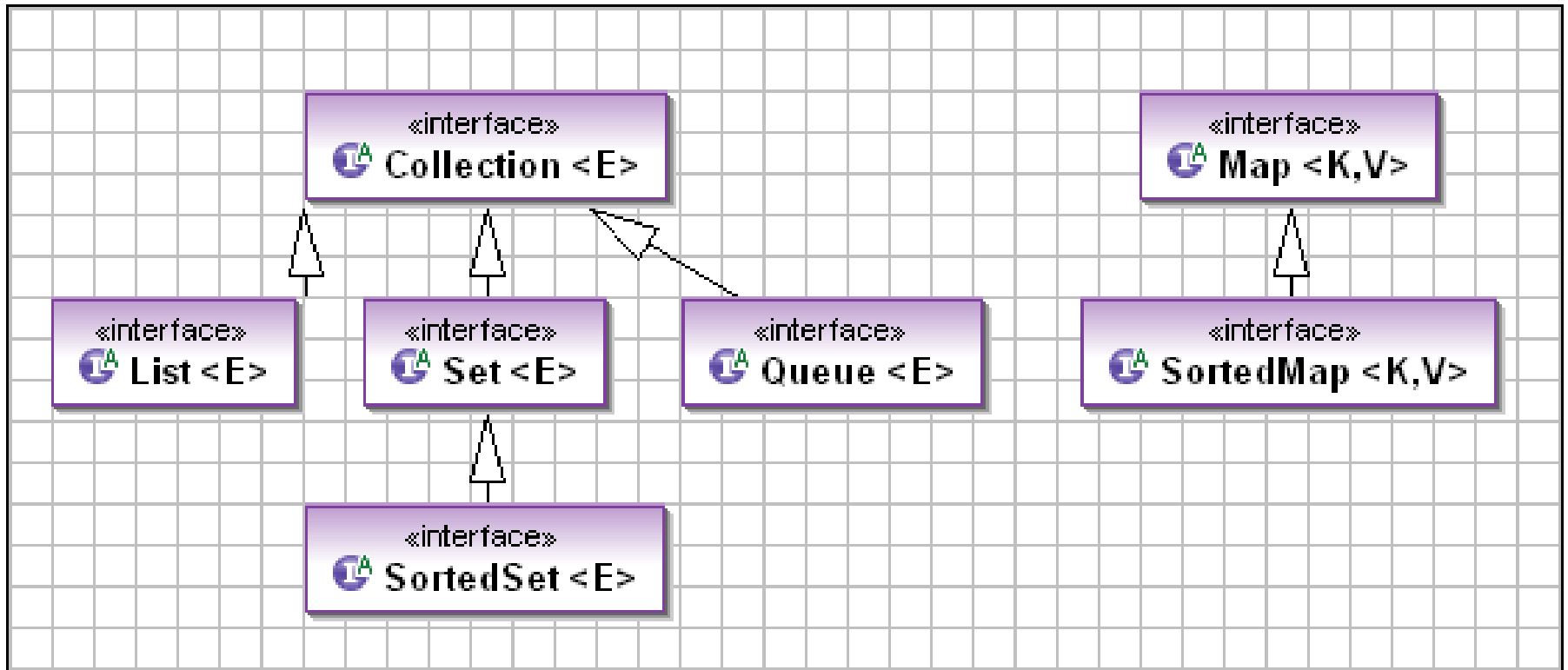
Módulo 6 – Coleções

SESSÃO 1 – COLEÇÕES E LISTAS

Coleções e a Java Collections Framework

- ❑ **Coleções:** permitem agrupar vários elementos
- ❑ **Java Collections Framework (JCF):**
 - É uma arquitetura unificada que inclui **interfaces**, **classes** (abstratas e concretas) e **algoritmos** (implementados por métodos).
 - A JCF inclui quatro tipos principais de coleções:
 - ❑ Conjuntos (**Set**): Coleção de elementos sem ordem e sem elementos repetidos
 - ❑ Listas (**List**): Coleção de elementos ordenados e com possíveis repetições
 - ❑ Mapas (**Map**): Coleção de pares chave-valor, sem repetição da chave
 - ❑ Filas (**Queue**) : Sequências de elementos com diferentes critérios de inserção e remoção

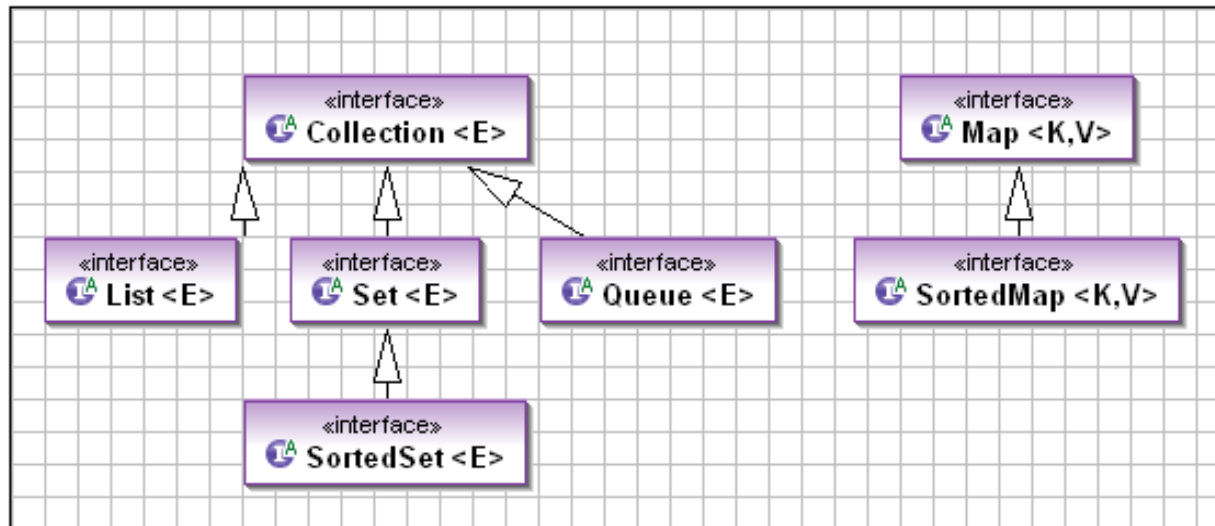
Java Collections Framework — Interfaces Genéricas



- As coleções da JCF são **definidas através de interfaces**. Neste caso cada uma das interfaces estabelece os métodos que um determinado tipo de coleção deve ter.

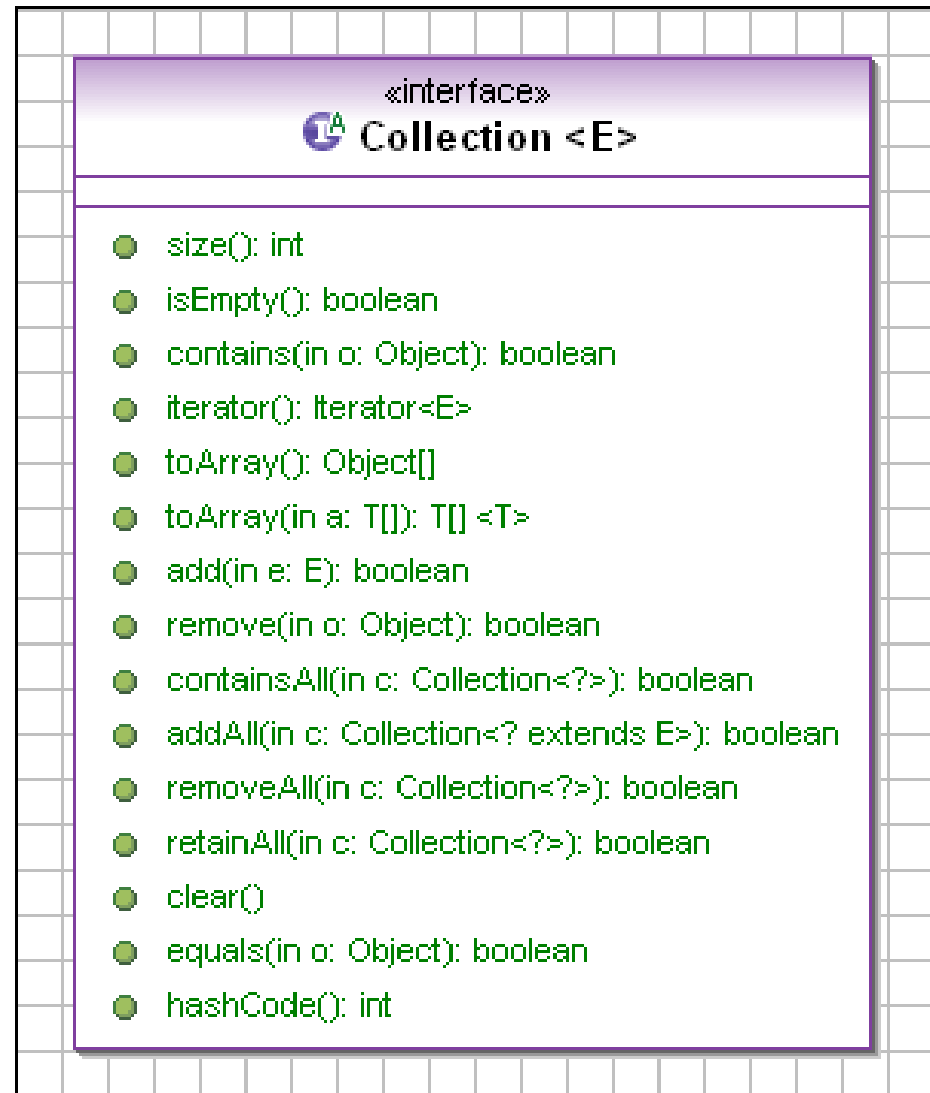
Java Collections Framework — Interfaces Genéricas

- A JCF define **interfaces genéricas** permitindo que se possa utilizar nas coleções um determinado tipo de dados escolhido pelo programador.
- Exemplos:
 - **public interface Collection<E> ...**
 - Coleção de elementos do tipo 'E'lement
 - **public interface Map<K,V> ...**
 - Mapa de elementos do tipo 'V'alue com chave do tipo 'K'ey.



Coleções — Interface Collection<E>

- **Collection<E>** é a interface base de grande parte das coleções.
- Significa que os métodos desta interface serão implementados por todas as coleções.



Coleções — Interface List<E>

□ **List<E>** para coleções com elementos em sequência.

- Os elementos nas listas estão ordenados
- As listas podem ter elementos duplicados.
- O cliente de uma Lista tem, normalmente, controlo sobre a posição onde um elemento é inserido.
- O acesso a um elemento é feito por um índice (referência de posição).



Coleções – ArrayList<E>

- **ArrayList<E>** – é a **implementação** da interface **List<E>** mais comum e que possui um bom desempenho especialmente no acesso aos dados e nas operações de iteração.
 - A implementação interna utiliza *arrays*.
 - **ArrayList<Person> persons = new ArrayList<>();**
 - Declara e cria um **ArrayList** chamado **persons** para armazenar objetos da classe **Person**
- A **JCF** define as coleções através de interfaces e disponibiliza várias classes que implementam essas coleções.
 - Neste caso a classe genérica **ArrayList<E>** é uma das implementações disponíveis na JCF para a interface **List<E>**

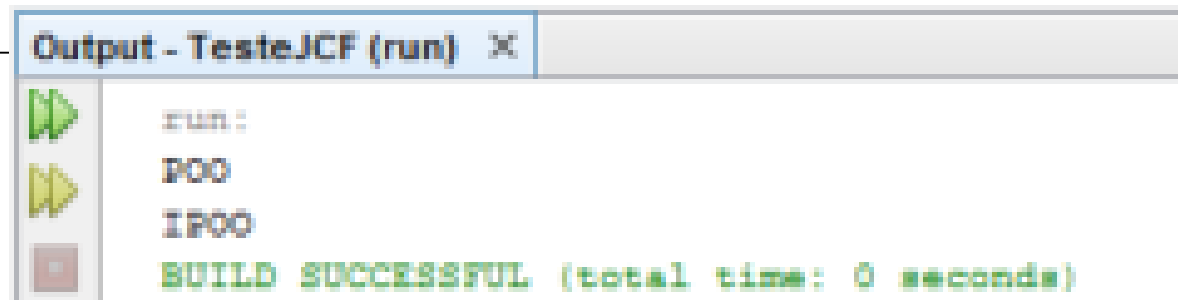
Exemplo — Utilização de listas



Coleções — ArrayList<E>

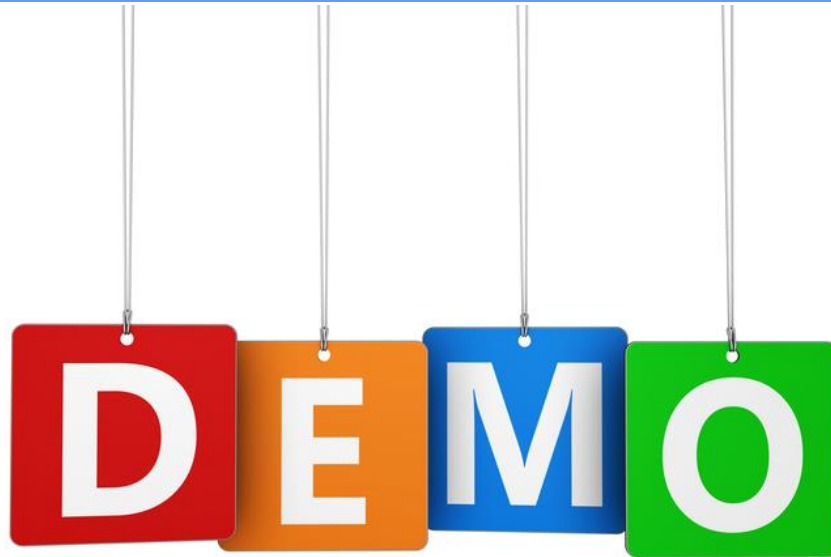
❑ Exemplo de utilização:

```
public static void main(String[] args) {  
    ArrayList<String> list = new ArrayList<>();  
    list.add("IPOO");  
    list.add("POO");  
    list.add("POO");  
    list.add("IPOO");  
    for(int i = 0; i < list.size(); ++i) {  
        if (list.get(i).equals("POO")) {  
            list.remove(i);  
        }  
    }  
    list.remove("IPOO");  
    for(int i = 0; i < list.size(); ++i) {  
        System.out.println(list.get(i));  
    }  
}
```



- **LinkedList<E>** – É outra **implementação** da interface **List<E>**. Nas operações de inserção e remoção de elementos pode oferecer melhor desempenho do que a anterior.
- **LinkedList<Person> persons = new LinkedList<>();**
 - Declara e cria uma **LinkedList** chamada **persons** para armazenar objetos da classe **Person**
- Internamente esta implementação não utiliza **arrays**, para não "desperdiçar" espaço, mas regista, para cada elemento qual o próximo elemento e qual o elemento anterior, sendo então fácil percorrer a lista e inserir e/ou remover elementos.

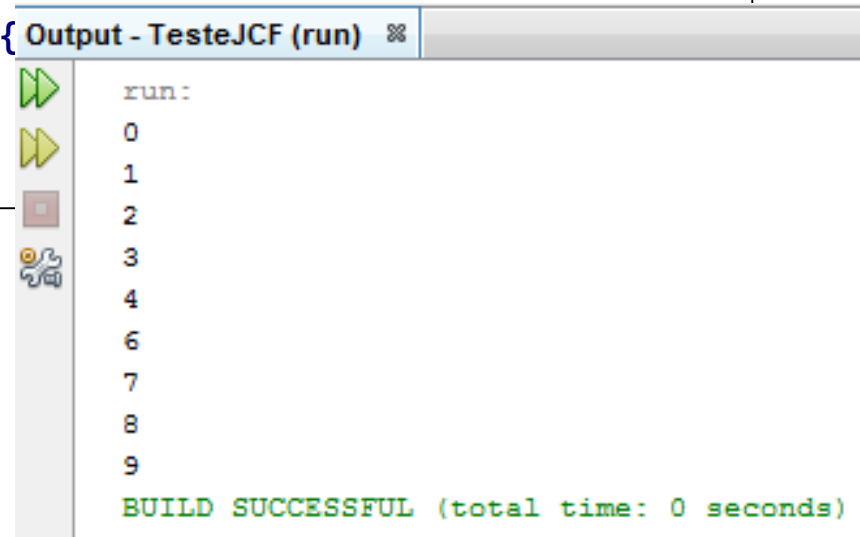
Exemplo — Utilização de listas



Coleções — LinkedList<E>

❑ Exemplo de utilização:

```
public static void main(String[] args) {  
    LinkedList<Integer> list = new LinkedList<>();  
    for (int val = 0; val < 10; val++) {  
        list.add(new Integer(val));  
    }  
    for (int i = 0; i < list.size(); i++) {  
        if (list.get(i).intValue() == 5) {  
            list.remove(i);  
        }  
    }  
    for (int i = 0; i < list.size(); i++) {  
        System.out.println(list.get(i));  
    }  
}
```

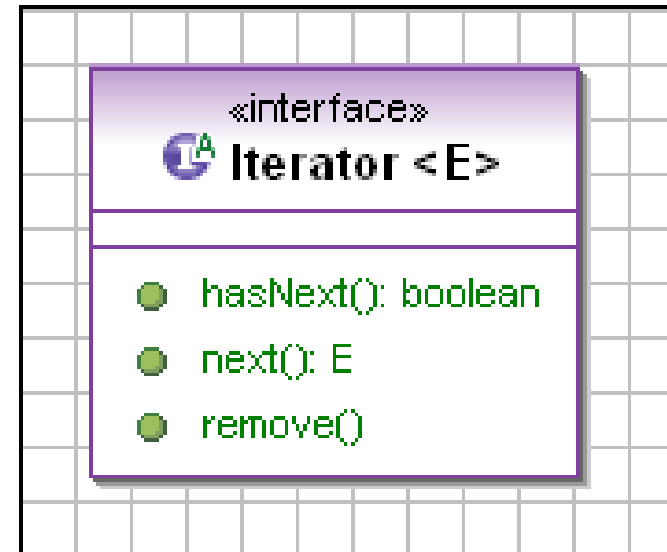


```
run:  
0  
1  
2  
3  
4  
6  
7  
8  
9  
  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Coleções – Interface `Iterator<E>`

- A Interface **`Iterator<E>`** define os métodos essenciais para iterar (percorrer) uma coleção:

- **`hasNext()`** – Determinar se a coleção tem ou não um elemento seguinte.
- **`next()`** – Devolve o elemento seguinte da iteração
- **`remove()`** – Remove o último elemento iterado.



- Exemplo:

```
Iterator<Person> s1 = persons.iterator();
```

- Neste exemplo **`s1`** vai guardar um objeto de uma classe que desconhecemos **`Iterator<E>`**. O objeto é obtido através da chamada ao método **`iterator()`** que existe em todas as coleções (está declarado na interface **`Collection<E>`**)
- Através de **`s1`** vai ser possível “iterar” (percorrer) a lista de pessoas.
- O método **`remove()`** vai permitir remover o elemento obtido com **`next()`** e é a única forma segura de alterar uma coleção durante uma iteração.

Coleções — Interface `Iterator<E>`

□ Exemplo:

Definimos **stringList** a partir da interface **List<E>**, assim podemos mais tarde escolher outra implementação

Vamos preencher a lista num método separado


Iteração dos elementos da lista

```
public static void main(String[] args) {  
    List<String> stringList = new ArrayList<>();  
  
    System.out.println("Com while:");  
    stringList = fillList(stringList);  
    Iterator<String> s1 = stringList.iterator();  
    while (s1.hasNext()) {  
        System.out.println(s1.next());  
        s1.remove();  
    }  
  
    System.out.println("\nCom do-while:");  
    stringList = fillList(stringList);  
    Iterator<String> s2 = stringList.iterator();  
    if (s2.hasNext()) {  
        do {  
            System.out.println(s2.next());  
            s2.remove();  
        } while (s2.hasNext());  
    }  
    // Continua...
```


Coleções — Interface Iterator<E>

```
System.out.println("\nCom for:");
stringList = fillList(stringList);
Iterator<String> s3 = stringList.iterator();
for (int i = 0; i < stringList.size(); i++) {
    if (s3.hasNext()) {
        System.out.println(s3.next());
        s3.remove();
    }
} // Fim do main

public static List<String> fillList(List<String> list) {
    list.add("IPOO");
    list.add("POO");
    list.add("PV");
    list.add("POO");
    list.add("POO");
    return list;
}
```

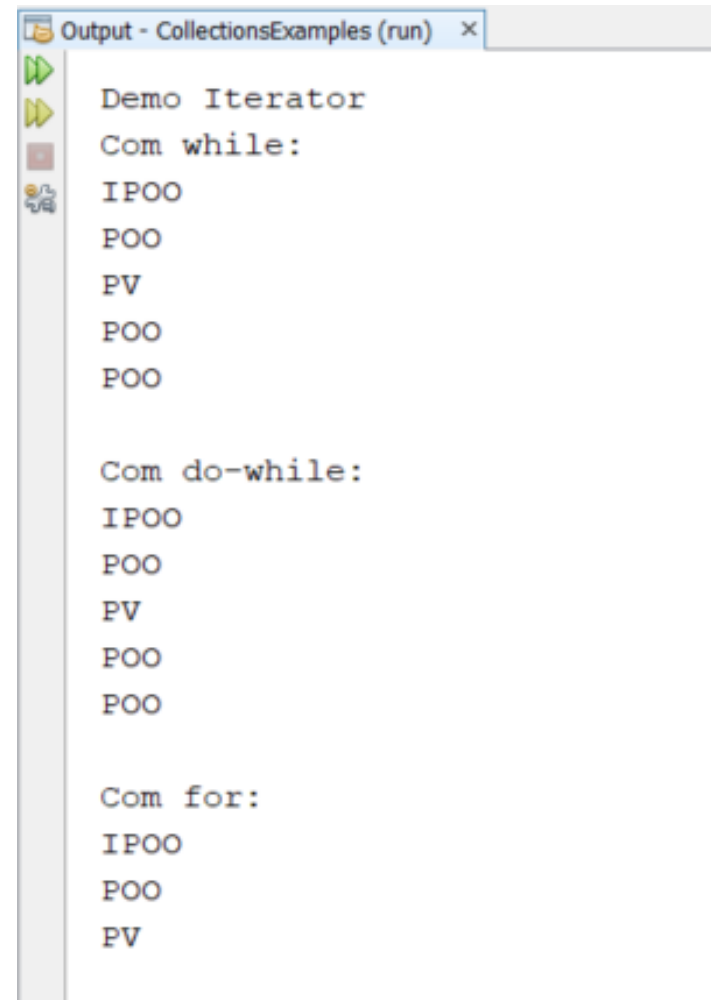


O valor de retorno e o parâmetro de entrada são do tipo **List<E>** !!!

Coleções – Interface Iterator<E>

```
public static void main(String[] args) {
    List<String> listaStrings = new ArrayList<>();

    System.out.println("Com while:");
    listaStrings = preencherLista(listaStrings);
    Iterator<String> s1 = listaStrings.iterator();
    while (s1.hasNext()) {
        System.out.println(s1.next());
        s1.remove();
    }
    System.out.println("Com do-while:");
    listaStrings = preencherLista(listaStrings);
    Iterator<String> s2 = listaStrings.iterator();
    if (s2.hasNext()) {
        do {
            System.out.println(s2.next());
            s2.remove();
        } while (s2.hasNext());
    }
    System.out.println("Com for:");
    listaStrings = preencherLista(listaStrings);
    Iterator<String> s3 = listaStrings.iterator();
    for (int i = 0; i < listaStrings.size(); i++) {
        if (s3.hasNext()) {
            System.out.println(s3.next());
            s3.remove();
        }
    }
}
```



```
Output - CollectionsExamples (run) x
Demo Iterator
Com while:
IPOO
POO
POO
PV
POO
POO

Com do-while:
IPOO
POO
POO
PV
POO
POO

Com for:
IPOO
POO
POO
PV
```

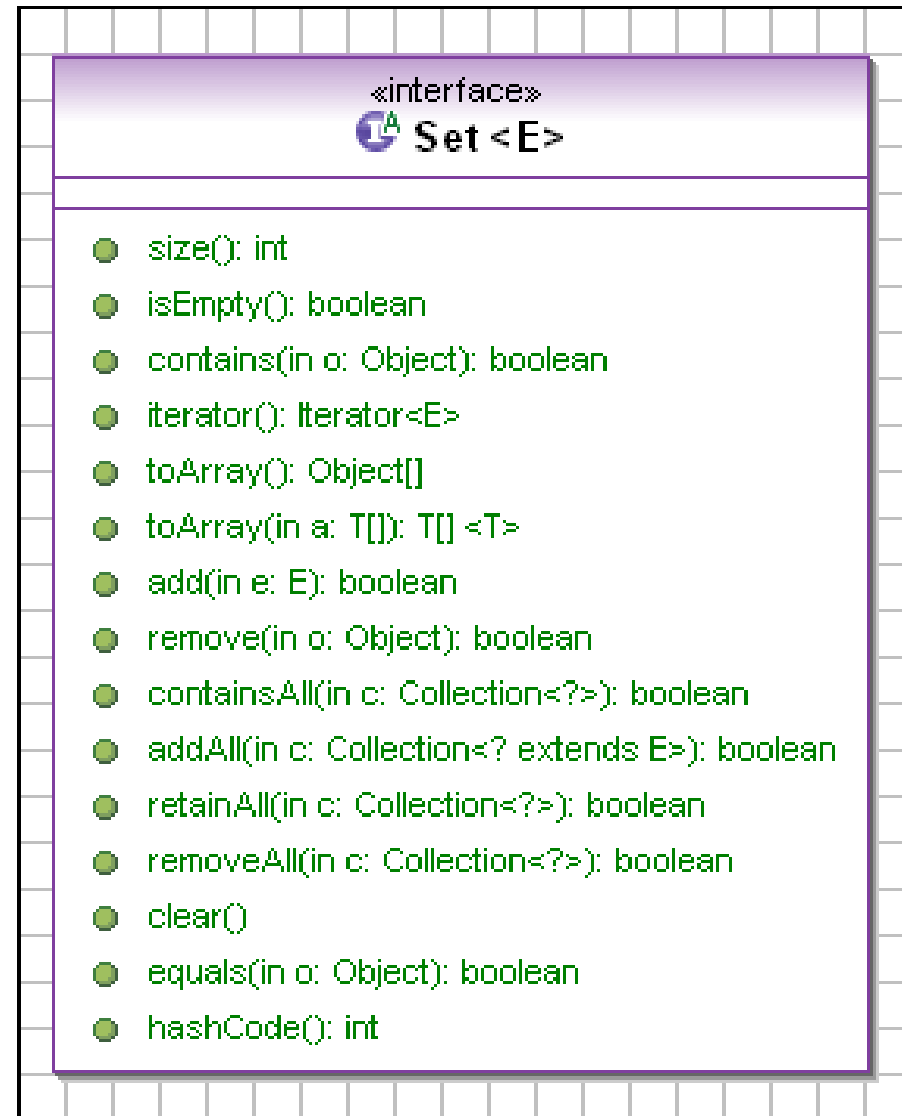


Módulo 6 – Coleções

SESSÃO 2 – CONJUNTOS

Coleções — Interface Set<E>

- **Set<E>** - para conjuntos de elementos sem duplicações
 - Representa a noção matemática de conjunto.
 - Não existe ordenação de qualquer tipo sobre os elementos adicionados.



Coleções — Classes que implementam Set<E>

- **HashSet<E>** – armazena os elementos numa **hash table**.
 - É a implementação com melhor desempenho mas não garante nada quanto à ordem de iteração.
 - **HashSet<Person> persons = new HashSet<>();**
 - Declara e cria um **HashSet** chamado **persons** para armazenar objetos da classe **Person**.
- Outras implementações de conjuntos (que mantêm a ordem de iteração):
 - **TreeSet<E>**
 - **LinkedHashSet<E>**

Utilização de Sets

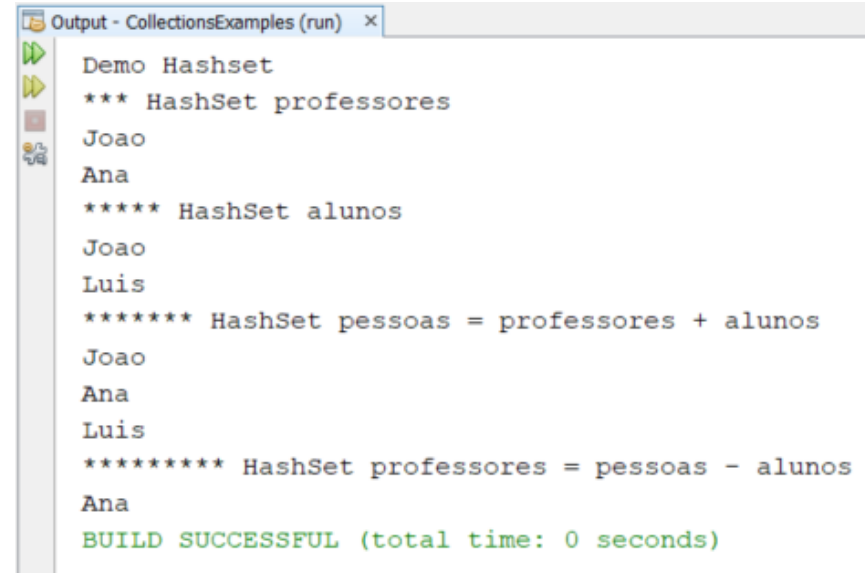
- ❑ A interface **Set** não associa a cada elemento uma posição dentro da coleção, como acontece na interface **List**.
- ❑ Assim a melhor forma para percorrer os elementos de um conjunto é através do ciclo **for-each** ou de um **iterator**
- ❑ Tal como com as outras coleções devemos declarar a variável com a interface **Set** e depois escolher a implementação desejada (**HashSet**, **TreeSet**, **LinkedHashSet**). Assim minimiza-se o “Acoplamento de Subclasses”:

```
Set<String> set = new HashSet<>();
```

- Desta forma podemos optar por diferentes implementações com alterações mínimas.

Coleções - Classe HashSet<E>

```
public static void main(String[] args) {  
    System.out.println("*** HashSet professores");  
    Set<String> teachers = new HashSet<>();  
    teachers.add("Ana");  
    teachers.add("Joao");  
    for (String s : teachers) {  
        System.out.println(s);  
    }  
    Set<String> students = new HashSet<>();  
    students.add("Joao");  
    students.add("Luis");  
    System.out.println("***** HashSet alunos");  
    for (String s : students) {  
        System.out.println(s);  
    }  
    Set<String> persons = new HashSet<>(teachers);  
    persons.addAll(students);  
    System.out.println("***** HashSet pessoas = professores + alunos");  
    for (String s : persons) {  
        System.out.println(s);  
    }  
    teachers.removeAll(students);  
    System.out.println("***** HashSet professores = pessoas - alunos");  
    for (String s : teachers) {  
        System.out.println(s);  
    }  
}
```



```
Output - CollectionsExamples (run) x  
Demo Hashset  
*** HashSet professores  
Joao  
Ana  
***** HashSet alunos  
Joao  
Luis  
***** HashSet pessoas = professores + alunos  
Joao  
Ana  
Luis  
***** HashSet professores = pessoas - alunos  
Ana  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Unicidade dos Elementos

- Num conjunto para se garantir que os elementos são únicos e não existem duplicados tem de se redefinir os métodos **equals** e **hashCode**
 - Redefinir apenas um deles não é suficiente
 - O método **boolean equals(Object obj)** definido na classe **Object** devolve um valor lógico que indica se um objeto é igual a outro passado como argumento.
 - O método **int hashCode()** é definido na classe **Object** e tenta devolver, para todos os objetos, um valor que o identifique univocamente.

Redefinir o método equals

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Person other = (Person) obj;
    if (!Objects.equals(this.name, other.name)) {
        return false;
    }
    return true;
}
```


Garante que os objetos
são da mesma classe

Cast para a classe onde se
está a colocar o método
(neste caso **Person**)

Verificação da igualdade
dos dois objetos

Redefinir o método hashCode

```
@Override  
public int hashCode() {  
    return name.hashCode();  
}
```



Neste exemplo duas pessoas eram *iguais* se tivessem o mesmo nome pelo que poderíamos usar o **hashCode da String** (o name):

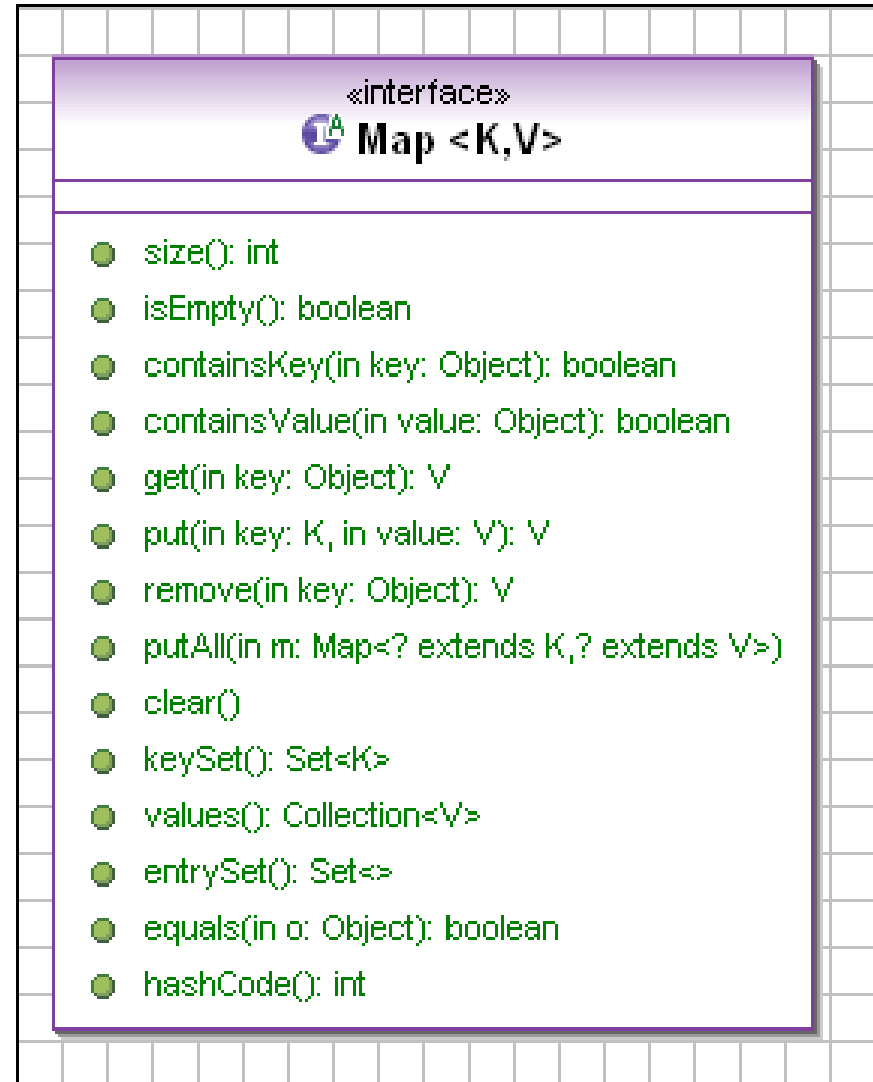


Módulo 6 – Coleções

SESSÃO 3 – MAPAS

Coleções - Interface Map<K,V>

- **Map<K,V>** - para associações entre dois elementos, onde um representa a chave (*Key*) e o outro o seu valor (*Value*).
 - As chaves são únicas (não há chaves repetidas!)
 - Cada chave refere um único elemento.
 - Os valores podem ser repetidos desde que pertençam a chaves diferentes.



Coleções — Classes que implementam Map<K,V>

- **HashMap<K,V>** - armazena pares de elementos associados
 - **HashMap<Integer, Person> persons = new HashMap<>();**
 - Cria um objeto **HashMap** chamado **persons** em que as chaves são do tipo inteiro e os valores são objetos da classe **Person**.
 - O seu comportamento e desempenho são semelhantes ao conjunto análogo **HashSet**: também não temos garantia quanto à ordem de iteração e usa uma **hash table** na sua implementação (recorrendo aos métodos **equals** e **hashCode** para determinar a unicidade).
- Outras implementações de mapas (que mantêm a ordem de iteração):
 - **TreeMap<K,V>**
 - **LinkedHashMap<K,V>**

Utilização de Maps

- ❑ A interface **Map**, tal como a interface **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

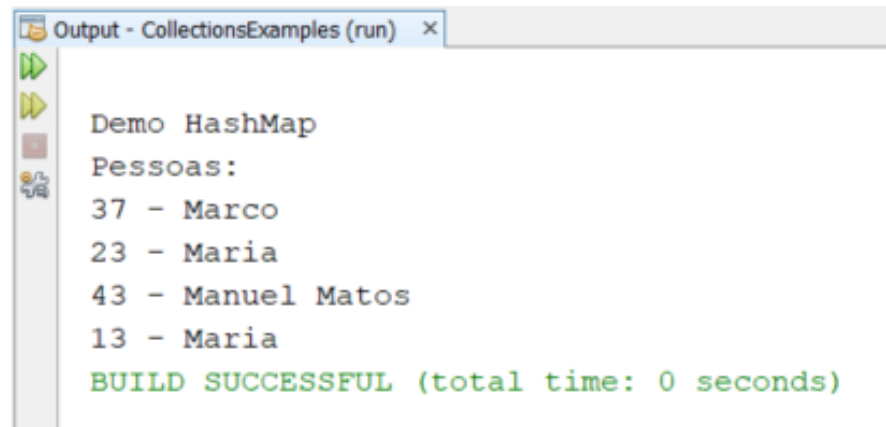
- Supondo que se cria um mapa que associa números (**Integer**) a nomes de pessoas (**String**):
 - **Integer** – porque as classes genéricas não podem trabalhar com tipos primitivos – **int**. É preciso utilizar as classes equivalentes)

```
public static void main(String[] args) {  
    Map<Integer, String> namesMap = new HashMap<>();  
    namesMap.put(13, "Maria");  
    namesMap.put(43, "Manuel");  
    namesMap.put(37, "Marco");  
    namesMap.put(23, "Maria"); //Valor repetido  
    namesMap.put(43, "Manuel Matos"); // Chave repetida,  
    // substitui valor anterior  
    printMap(namesMap); //Mostrar no ecrã    }
```

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** podemos aceder a todos os elementos:

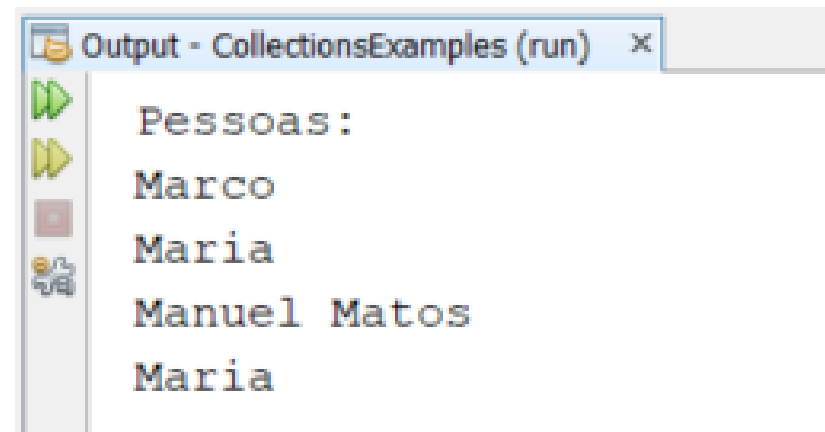
```
public static void printMap(Map<Integer, String> map) {  
    System.out.println("Pessoas:");  
    for (Integer i : map.keySet()) {  
        System.out.println("" + i + " - " + map.get(i));  
    }  
}
```



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** podemos aceder a todos os elementos:

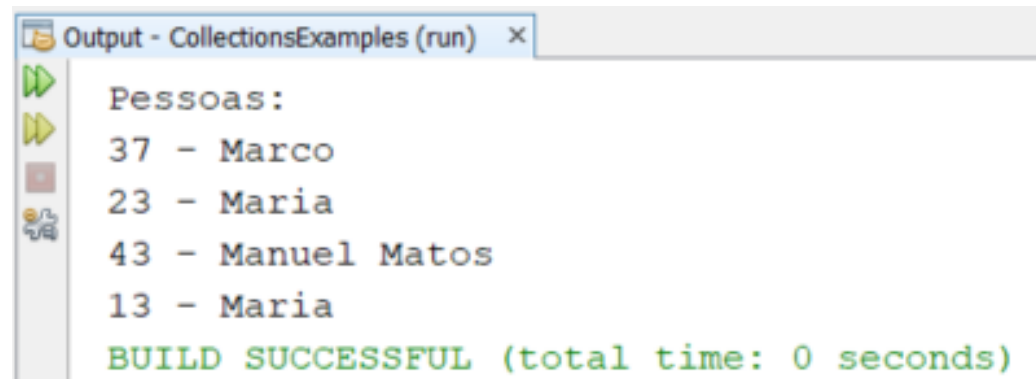
```
public static void printMap(Map<Integer, String> map) {  
    System.out.println("Pessoas:");  
    for (String name : map.values()) {  
        System.out.println(name);  
    }  
}
```



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** podemos aceder a todos os elementos:

```
public static void printMap(Map<Integer, String> map) {  
    System.out.println("Pessoas:");  
    for (Map.Entry pair : map.entrySet()) {  
        System.out.println("" + pair.getKey() + " - " + pair.getValue());  
    }  
}
```



```
Output - CollectionsExamples (run) x  
Pessoas:  
37 - Marco  
23 - Maria  
43 - Manuel Matos  
13 - Maria  
BUILD SUCCESSFUL (total time: 0 seconds)
```



Módulo 6 – Coleções

SESSÃO 4 – EXEMPLO ESCOLA, ALGORITMOS

Exemplo Escola

- ❑ Criar um sistema para registrar as notas de alunos.
- ❑ Serão necessárias as seguintes classes:
 - **Student** – informação de um aluno (número e nome)
 - ❑ O número é único, identifica o aluno e deve ser gerado automaticamente
 - **SchoolClass** - informação da turma
 - **Grade** - onde se associam os alunos às notas



□ Classe **Student**

```
public class Student {  
  
    private static int nextNumber = 1;  
    private int number;  
    private String name;  
  
    public Student(String name) {  
        this.number = Student.nextNumber++;  
        this.name = name;  
    }  
  
    // Continua...
```

□ Classe **Student** – métodos **seletores** e **modificadores**

```
public int getNumber() {  
    return number;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
@Override  
public String toString() {  
    return number + " - " + name;  
}
```

Exemplo Escola

❑ Classe **Student** – métodos **equals** e **hashCode**

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

Usa o **HashCode** da classe **Integer**

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

O número é usado
para distinguir
dois alunos

Exemplo Escola

❑ Classe **SchoolClass**

```
public class SchoolClass {  
  
    private Map<Integer, Student> students;  
    private String name;  
  
    public SchoolClass(String name) {  
        this.name = name;  
        students = new HashMap<>();  
    }  
  
    // Continua...
```

**Para ser mais simples de
aceder aos alunos através do
seu número usamos uma
associação entre o número
(chave) e o aluno (valor)**

Cuidado!!!
Temos de manter o
número de aluno da
chave idêntico ao que
está guardado no objeto
aluno associado

❑ Classe **SchoolClass** – Métodos (1/2)

```
public String getName() {  
    return name;  
}
```

```
public void setName(String name) {  
    this.name = name;  
}
```

```
public void add(Student student) {  
    students.put(student.getNumber(), student);  
}
```

```
public Student get(int number) {  
    return students.get(number);  
}
```

Exemplo Escola

❑ Classe **SchoolClass** – Métodos (2/2)

```
public Student remove(int number) {  
    return students.remove(number);  
}
```

```
public boolean isEnrolled(int number) {  
    return students.containsKey(number);  
}
```

```
@Override  
public String toString() {  
    String list = name + ":";  
    for (Student student : students.values()) {  
        list += "\n" + student;  
    }  
    return list;  
}
```

Exemplo Escola

□ Classe **Grade**

```
public class Grade extends HashMap<Student, Integer> {  
  
    @Override  
    public String toString() {  
  
        String grades = "Notas:";  
        for (Student student : keySet ()) {  
            grades += "\n" + student + ": " + get(student);  
        }  
        return grades;  
    }  
}
```

**Neste caso consideramos a
avaliação como uma
associação entre aluno e nota**

Cuidado!!!
Devemos ter em atenção se
não será necessário redefinir
algum dos métodos herdados.

Exemplo Escola

❑ Programa Principal

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

E se
quiséssemos
ordenar os
alunos pelo
nome?



```
BlueJ: BlueJ: Janela de Termi...  
Opções  
Turma da LEI:  
1 - Rita  
2 - Manuel  
3 - Anibal  
4 - Maria  
Notas:  
1 - Rita: 12  
2 - Manuel: 14  
3 - Anibal: 11  
4 - Maria: 17
```

Coleções e a Java Collections Framework

□ **Java Collections Framework (JCF):**

- É uma arquitetura unificada que inclui **interfaces**, **classes** (abstratas e concretas) e **algoritmos** (implementados por métodos).

- Como foi dito anteriormente a JCF contém as **interfaces** que especificam as coleções e **classes** (concretas) que implementam essas coleções.
- Além disso ainda tem classes abstratas que implementam parcialmente coleções e **algoritmos**.
- Os **algoritmos** permitem várias operações sobre as coleções como por exemplo: a ordenação, a inversão da ordem dos elementos, a mistura aleatória dos elementos, etc.

Java Collections Framework — Algoritmos

- Os **algoritmos** da JCF são fornecidos como métodos estáticos da classe **Collections** e a maioria aplica-se especificamente a *listas*, alguns deles são:
 - **sort** – ordena uma lista por um critério
 - **shuffle** – baralha os elementos da lista aleatoriamente
 - **reverse** – reverte a ordem dos elementos na lista
 - **rotate** – roda todos os elementos da lista numa distância especificada
 - **swap** – troca os elementos em posições especificadas de uma lista
 - **replaceAll** – troca todas as ocorrências de um valor por um outro valor especificado
 - **fill** – atribui a todos os elementos da lista um valor especificado
 - **copy** – copia uma lista para outra
 - **binarySearch** – procura um elemento numa lista com o algoritmo de procura binária
- Exemplo de evocação de um algoritmo:
 - `Collections.rotate(names, 4);`
 - Passa os 4 últimos elementos de **names** para o princípio da lista **names** pela mesma ordem em que estavam no final da lista.

Exemplo Escola - Ordenação dos elementos de uma coleção

- ❑ Poderia ser interessante apresentar as notas dos alunos ordenando-os alfabeticamente pelo seu nome
- ❑ Como foi referido, 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** comparando 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<T>

- A 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.

Exemplo Escola - Ordenação pelo nome dos Alunos

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

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

- Obriga a que seja implementado o método **compareTo**
 - Neste caso não é verificado se o **student** 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(Student student) {  
    return name.compareTo(student.name);  
}
```

Exemplo Escola - Ordenação na apresentação das notas

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

```
@Override
```

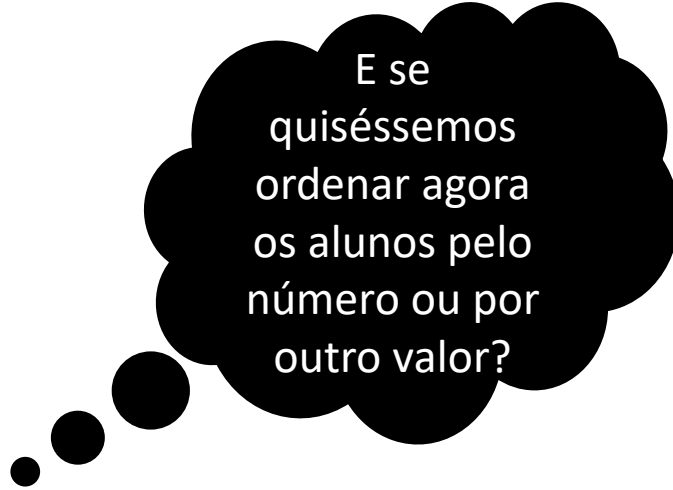
```
public String toString() {  
    List<Student> students = new ArrayList<>(keySet());  
    Collections.sort(students);  
  
    String grades = "Notas:";  
    for (Student student : students) {  
        grades += "\n" + student + ": " + get(student);  
    }  
    return grades;  
}
```

Exemplo Escola - Pauta Ordenada

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



```
Blue! Blue! Janela de Termi...  
Opções  
Turma da LEI:  
1 - Rita  
2 - Manuel  
3 - Anibal  
4 - Maria  
Notas:  
3 - Anibal: 11  
2 - Manuel: 14  
4 - Maria: 17  
1 - Rita: 12
```



E se quiséssemos ordenar agora os alunos pelo número ou por outro valor?

Interface Comparator<T>

- ❑ A solução que utiliza a interface **Comparable** leva a que a ordenação dos objetos duma classe seja feita apenas da forma que é definida através do método **compareTo**

- ❑ Outra solução é utilizar a interface **Comparator**:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- ❑ O método **compare** funciona da mesma forma que o método **compareTo** com a diferença que recebe como argumento dois objetos da mesma classe
 - Neste caso pode-se criar uma classe separada para cada tipo de comparação que se quer fazer e depois será possível usá-la no algoritmo de ordenação da classe **Collections**.

Exemplo Escola - Ordenação pelo número dos Alunos

- Para os alunos poderem ser ordenados pelo número é preciso criar uma classe que implementa a interface **Comparator<Student>** fornecendo o método de comparação **compare** que compara dois alunos pelos seus números.

```
public class StudentNumberComparator implements Comparator<Student> {  
  
    @Override  
    public int compare(Student student1, Student student2) {  
        return student1.getNumber() - student2.getNumber();  
    }  
}
```

Exemplo Escola - Ordenação na apresentação das notas

- Para produzir uma pauta ordenada pelo número de alunos modificando o **toString** de **Grade** é preciso depois de recolher os alunos numa lista e usar a ordenação com a classe **StudentNumberComparator** criada:

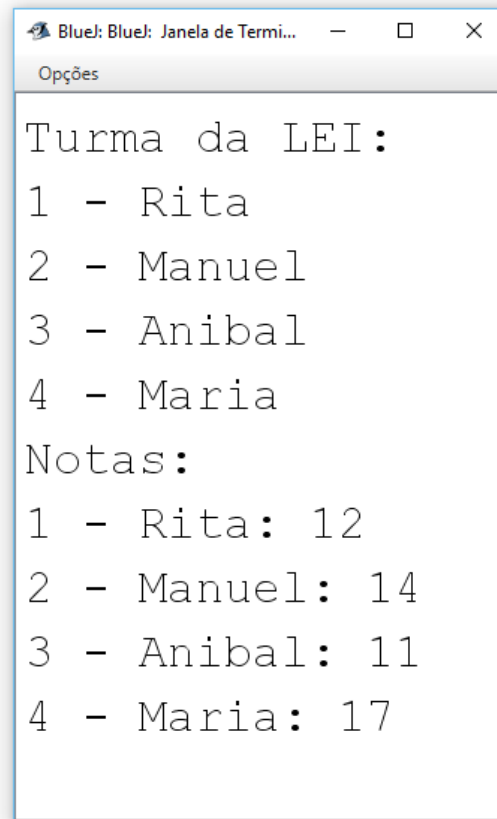
```
@Override
public String toString() {
    List<Student> students = new ArrayList<>(keySet());
    Collections.sort(students, new StudentNumberComparator());

    String grades = "Notas:";
    for (Student student : students) {
        grades += "\n" + student + ": " + get(student);
    }
    return grades;
}
```

Neste caso o método **sort** recebe o objeto comparador como segundo parâmetro

Exemplo Escola - Pauta Ordenada

- As notas são apresentadas com os alunos ordenados pelo número:



```
BlueJ: BlueJ: Janela de Termi...
Opções

Turma da LEI:
1 - Rita
2 - Manuel
3 - Anibal
4 - Maria
Notas:
1 - Rita: 12
2 - Manuel: 14
3 - Anibal: 11
4 - Maria: 17
```

- ❑ **Coleção** – agregado de elementos de um mesmo tipo
- ❑ **JCF** – arquitetura que inclui: Interfaces, Classes Abstratas e Concretas, Algoritmos
 - Interfaces genéricas – **Collection<E>** - a partir das quais se implementam classes de coleção específicas para guardar objetos de um dado tipo **<E>**
 - Os algoritmos da classe **Collections** manipulam listas com grande eficiência.
- ❑ **Listas**
 - Classe **ArrayList<E>** e **LinkedList<E>**
- ❑ **Iteração e Alterações** em Coleções
 - Ciclo **for-each** (type safe)
 - Não modificar a coleção durante a iteração ... a não ser que se use o ...
 - **Iterator<E>** – a única forma segura de iterar e alterar uma coleção em simultâneo

- ❑ **Conjuntos**
 - **HashSet** – implementação baseada em **hash table**
 - **LinkedHashSet** e **TreeSet** – implementações que garantem a ordem de iteração
- ❑ Unicidade de objetos através do uso dos métodos **equals** e **hashCode**
- ❑ **Mapas**
 - **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)
- ❑ A interface **Comparator<T>**, através do seu método **compare**, pode ser utilizada por uma classe onde se define a comparação de dois objetos por uma determinada forma.
- ❑ **Algoritmos**
 - O uso do método **Collection.sort** permite a ordenação dos elementos