DR1

Disciplina Regular 1 Fundamentos do Desenvolvimento Java

Graduação em Engenharia de Software - 2020

DR1

Etapa 7 Aula 1

Estruturas de Dados

DR1

Competências Trabalhadas Nesta Etapa

- Implementar o acesso a dados com Java
  - Compreender a arquitetura JDBC.
  - Criar bancos de dados, tabelas e relacionamentos com o MySQL Workbench.
  - Executar comandos SQL de inserção, atualização, exclusão e seleção.
  - Compreender a hierarquia de coleções.
  - Manipular coleções para tratar o resultado de uma consulta ao banco de dados.

No Moodle esse conteúdo se refere à etapa 9

DR1

Coleções

Biblioteca de Coleções

DR1

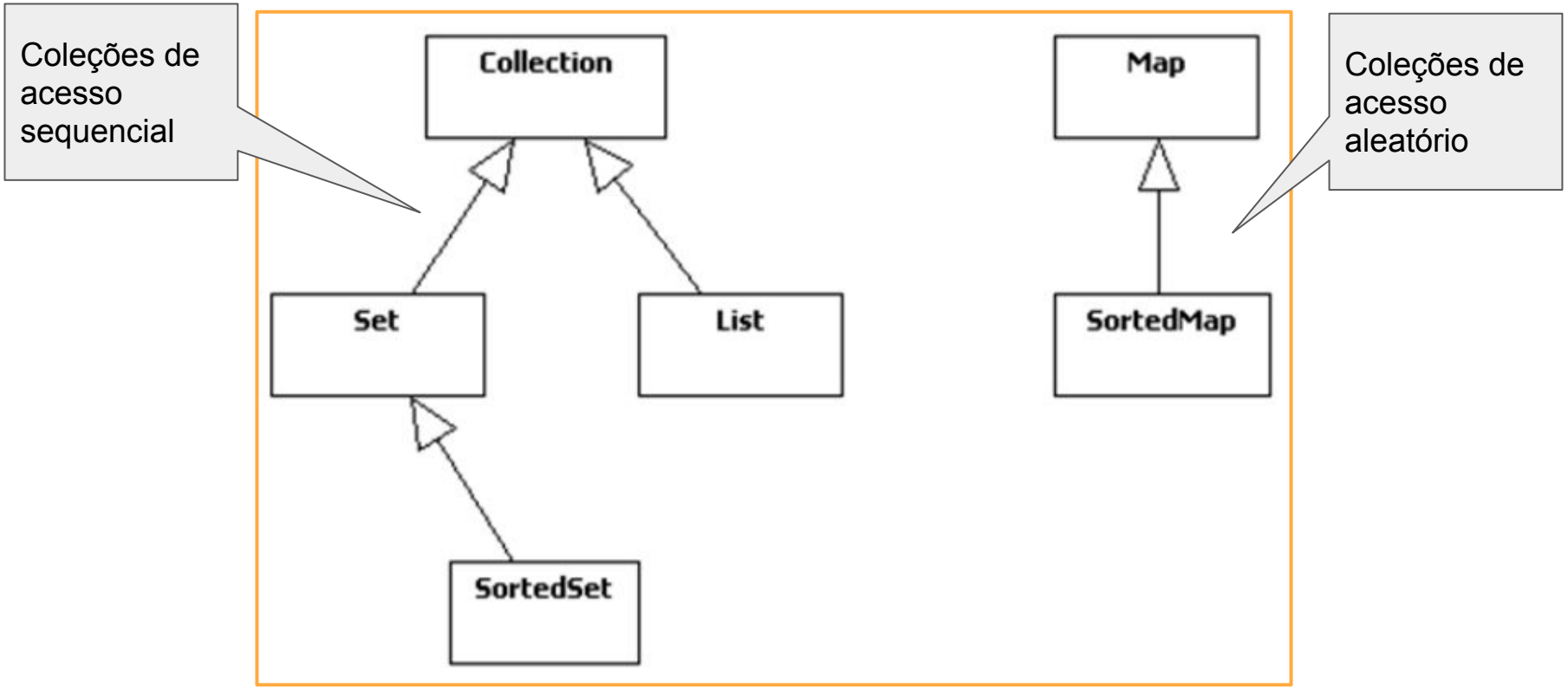
- Coleção é um objeto que contém múltiplos elementos.
- Também é chamada de container.
- Nas versões anteriores ao Java 2, existiam alguns recursos para agrupar objetos:
  - Array, Vector, Hashtable, Enumeration
- Apesar de já existirem estes recursos, eles não estavam integrados.

Biblioteca de Coleções

DR1

- Projeto unificado para manipulação de coleções (estruturas de dados):
  - **Interface**: representação abstrata de uma coleção, permitindo que outras seja usadas independente dos detalhes de sua implementação.
  - **Implementação**: referente às *interfaces*, são as estruturas de dados reutilizáveis.
  - **Algoritmos**: métodos que executam tarefas úteis, como classificação e busca. Permitem a reutilização da funcionalidade.

Interfaces


Coleções de acesso sequencial
Collection
Map
Coleções de acesso aleatório
Set
List
SortedMap
SortedSet

Interfaces

DR1

- **Collection** é a *interface* base da hierarquia, a mais genérica de todas.
- Não existe classe que implemente diretamente esta *interface*.
- Todas as classes implementam interfaces que herdam desta.
- Utilizada principalmente para passar coleções quando for necessária a máxima generalização.

Interfaces

DR1

- **Set** é um tipo de Collection que não pode ter elementos duplicados.

- Herda todos os métodos de Collection e não possui nenhum método adicional.

- Duas classes Java de coleções implementam esta *Interface*: **HashSet** e **TreeSet.**

- Usa-se a primeira quando o desempenho nas leituras for importante e a segunda para manutenção de dados (inserção e alteração).

Interfaces

DR1

- **List** é um tipo de Collection que contém elementos indexados, também conhecida como sequência.

- Pode conter elementos duplicados.

- Possui alguns métodos adicionais com relação à Collection: acesso posicional, busca, iteração e operações em sublistas.

- É de longe a *interface* mais utilizada nas aplicações.

Classes

DR1

- **ArrayList** representa um vetor dinâmico.
- Esta é a implementação mais eficiente de uma lista.
- **Não deve ser compartilhada por vários threads**.
- Caso seja necessária uma estrutura de dados compartilhada por vários threads, deve-se utilizar a classe **Vector**.

Classes

DR1

- **HashMap** representa um conjunto de elementos que mapeia chaves para valores.

- Essas classes são implementações da *interface* **Map**.

Conceitos Básicos sobre Generics

DR1

- Todas as coleções devem ser declaradas e inicializadas com a indicação do tipo de objeto que será armazenado nelas.

- Para isso, colocamos uma *tag* na declaração da coleção assim como no construtor da classe.

```
ArrayList<Aluno> alunos = new ArrayList<>();
HashMap<Integer, Aluno> = new HashMap<>();
```

- Essa instrução traz vários benefícios:
  - Segurança de tipos bem definidos;
  - Alocação otimizada de memória;
  - Uso do For Each

Conceitos Básicos sobre Generics

DR1

- É possível implementar iterações a fim de obter os elementos de uma coleção através da estrutura de controle **for**.

```
ArrayList<Aluno> alunos = new ArrayList<Aluno>();

...

for(Aluno aluno : alunos) {
    System.out.println(aluno.getNome());
}
```

- A cada iteração cada elemento da lista será atribuído à variável de controle.

Ordenação

```java
// A sample Java program to sort an array of integers
// using Arrays.sort(). It by default sorts in
// ascending order
import java.util.Arrays;

public class SortExample
{
    public static void main(String[] args)
    {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

        Arrays.sort(arr);

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Ordenação

```java
// A sample Java program to sort a subarray
// in descending order using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
    public static void main(String[] args)
    {
        // Note that we have Integer here instead of
        // int[] as Collections.reverseOrder doesn't
        // work for primitive types.
        Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Ordenação

```java
// Java program to demonstrate working of Collections.sort()
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);
    }
}
```

DR1

Ordenação

```java
// Java program to demonstrate working of Collections.sort()
// to descending order.
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al, Collections.reverseOrder());

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);
    }
}
```

Comparator

```
// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                               String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                           " " + this.address;
    }
}
```

```
class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}
```

```
class Sortbyname implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}
```

DR1

```
// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Comparable X Comparator

- **Comparable** é implementada **dentro** da própria classe que contém os dados a serem aramazenados. Só pode ter 1 critério.

- **Comparator** permite a criação de classes “comparadoras” sem que se mexa na classe onde estão os dados. Pode ter n critérios.

DR1

Pergunta para Pesquisar

- Qual das três opções é mais “eficiente” para ordenar dados?
  - ORDER BY → ordenar no banco de dados.
  - Arrays / Collections sort → ordenar no back-end.
  - JavaScript / JQuery → ordenar no front-end.

- **Responder e entregar junto com o TP4 no dia 23/3**.