Colecções

Programação III
José Luis Oliveira; Carlos Costa

1

Colecções

- Collection: Interface de JAVA que determina o comportamento que uma colecção deve ter.
- Introduzidas no Java 1.2 com a denominação de "JAVA Collections FrameWork (JCF)".
- São estruturas de dados com propriedades próprias que permitem agregar objectos de determinado tipo.
- Também são conhecidas como "containers"
- Não suporta tipos primitivos (*int*, *float*, *double*,..)
  - Utilizar Wrapper's (*Integer*, *Float*, *Double*, …)

2

Principais Interfaces

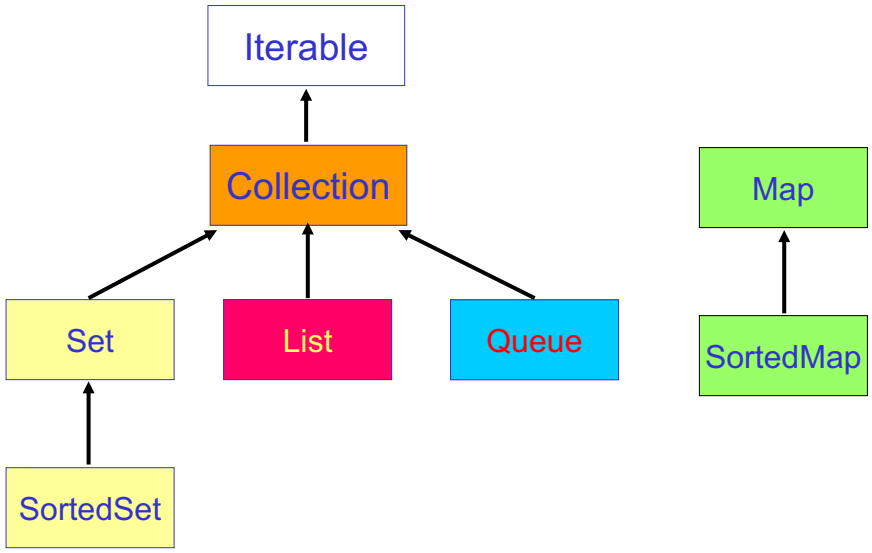
Java Collections Framework (JCF):

- Conjunto de classes, interfaces e algoritmos que representam vários tipos de estruturas de armazenamento de dados.

- Conjunto de 4 Interfaces Principais:
  - **Conjuntos (Set):** sem noção de posição (sem ordem), sem repetição
  - **Listas (List):** sequências com noção de ordem, com repetição
  - **Filas (Queue):** são as filas do tipo First in First Out
  - **Mapas (Map):** estruturas associativas onde os objectos são representados por um par chave-valor. Pares chave-valor (com repetição - MultiMap)

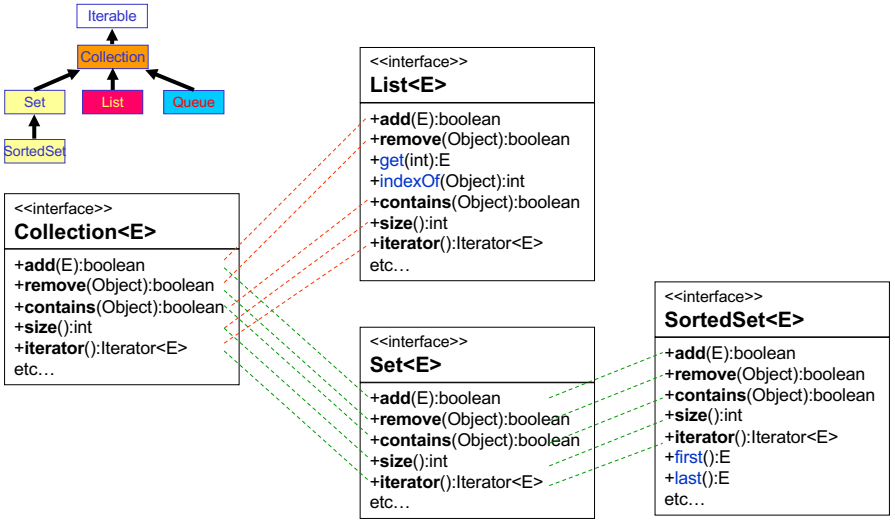
3

Hierarquia de interfaces

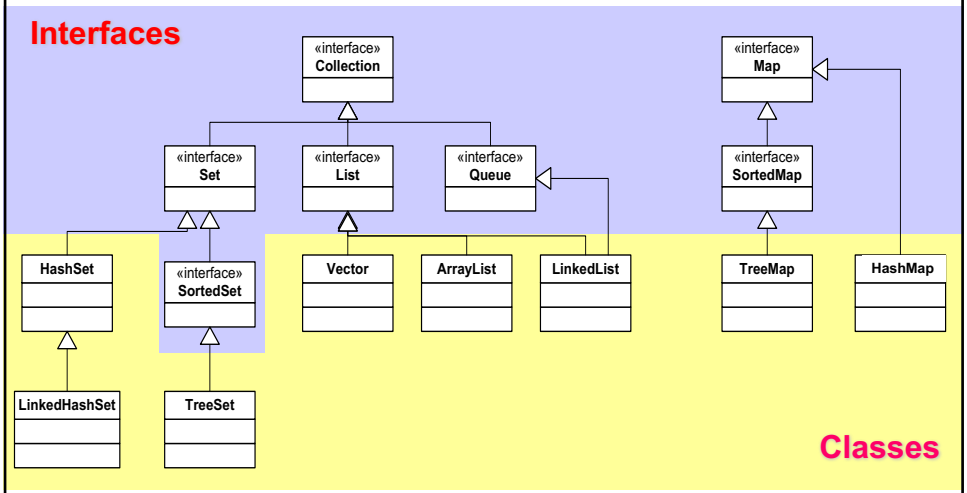


4

Expansão de contratos



Hierarquia de Classes



6

Interfaces e Implementações

Collections					
Interfaces	**Implementações**				
	Hash table	Resizable array	Balanced Tree (sorted)	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

7

Vantagens das Collections

- Vantagem de criar interfaces:
  - Separa-se a especificação da implementação
  - Pode-se substituir uma implementação por outra mais eficiente sem grandes impactos na estrutura existente.

- Exemplo:

```
Collection<String> c = new LinkedList<String>();
c.add("Aveiro");
c.add("Paris");
Iterator<String> i = c.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

8

Genéricos em Collections

Desde o JAVA 5 que as Collections são parametrizáveis

Antes..

```
LinkedList lista =
            new LinkedList();

lista.add(new Data(..));
lista.add(new Pessoa(..));

Iterator i = lista.iterator();

Data d = (Data)i.next();
Pessoa p = (Pessoa)i.next();
```

Agora..

```
LinkedList<Data> lista =
          new LinkedList<Data>();

lista.add(new Data(..));
// lista.add(new Pessoa(..));

Iterator<Data> i =
               lista.iterator();

Data d = i.next();
//Pessoa p = (Pessoa)i.next();
```

Compile-Time Error

Compile-Time Error

9

Interface Collection

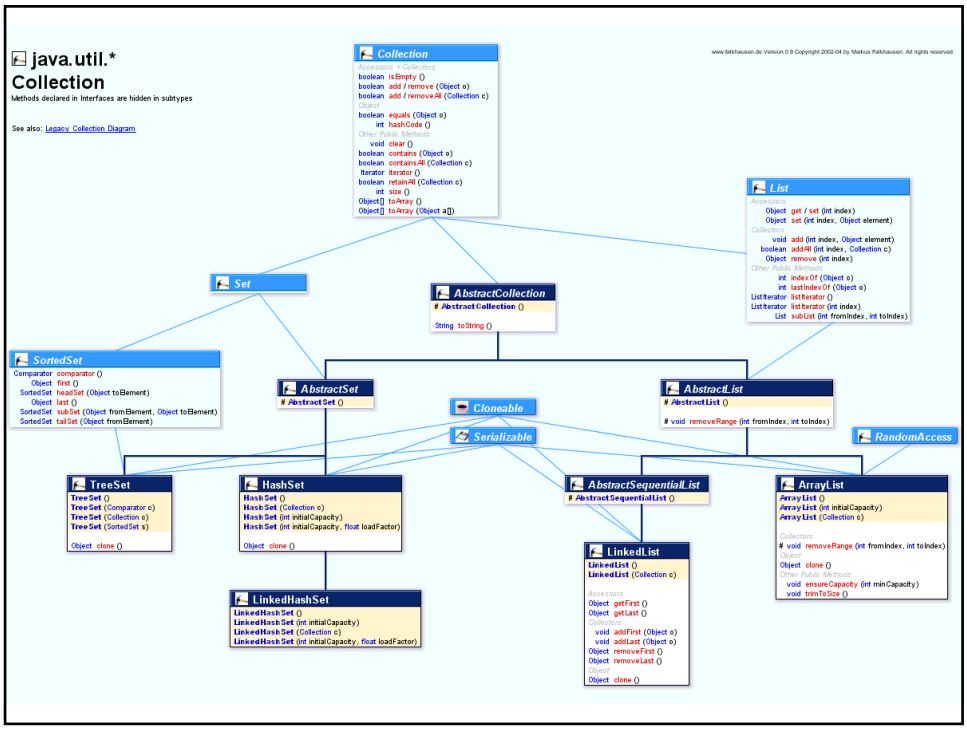
```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                    //optional
    boolean remove(Object element);            //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

10

Interface Iterable

```
public interface Iterable<T> {

    default void forEach(Consumer<? super T> action)
      Performs the given action for each element of the Iterable until
      all elements have been processed or the action throws an
      exception.

   Iterator<T>       iterator()
      Returns an iterator over elements of type T.

   default Spliterator<T> spliterator()
      Creates a Spliterator over the elements described by this
      Iterable.
}
```

11



Set - Conjuntos

- Uma coleção que não pode conter elementos duplicados.
- Contém apenas os métodos definidos na interface Collection
  - Novos contratos nos métodos add, equals e hashCode
- Implementações:
  - HashSet
  - TreeSet
  - ..

13

AbstractSet

```
public abstract class AbstractSet<E>
        extends AbstractCollection<E> implements Set<E>  {

    protected AbstractSet();

    public boolean equals(Object o) {
        if (!(o instanceof Set)) return false;
        return ((Set)o).size()==size() && containsAll((Set)o);
    }

    public int hashCode() {
        int h = 0;
        for( E el : this )
            if ( el != null ) h += el.hashCode();
        return h;
    }
}
```

14

HashSet

- Usa uma tabela de dispersão (Hash Table) para armazenar os elementos.
  - Uma instância de Hashmap
- A inserção de um novo elemento não será efectuada se o equals do elemento a ser inserido com algum elemento do Set retornar true.
  - A implementação da função equals é fundamental.
- Desempenho constante,
  - O(~1) para add, remove, contains e size

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.HashSet<E>
```

15

HashSet

```
import java.util.*;

public class TestHashSet {

   public static void main(String args[]) {
     String[] str = {"Rui", "Manuel", "Rui", "Jose",
                     "Pires", "Eduardo", "Santos"};

     Set<String> s = new HashSet<String>();
       for (String i: str ) {
         if (!s.add(i))
           System.out.println("Nome duplicado: " + i);
       }
       System.out.println(s.size() + " palavras distintas");


       Iterator<String> itr = s.iterator();
       while ( itr.hasNext() )
               System.out.println( itr.next() );
     }
}
```

Nome duplicado: Rui

6 palavras distintas

Manuel

Rui

Jose

Eduardo

Santos

Pires

Ordem!

Porquê ?

Conclusão: sem noção de posição (sem ordem)

16

TreeSet

- A implementação baseada numa estrutura em árvore balanceada.
- Desempenho log(n) para add, remove e contains
- Permite a Ordenação dos Elementos pela:
  - sua "ordem natural". Os objectos inseridos em TreeSet's devem implementar a interface Comparable .
  - ou utilizando um objecto do tipo Comparator no construtor de TreeSet.

... Exemplo detalhado mais adiante

17

TreeSet

```
public class TestTreeSet {

   public static void main(String[] args) {
        Collection<Quadrado> c = new TreeSet<Quadrado>();
        c.add(new Quadrado(3, 4, 5.6)); c.add(new Quadrado(1, 5, 4));
        c.add(new Quadrado(0, 0, 6)); c.add(new Quadrado(4, 6, 7.4));
        System.out.println(c);

        Quadrado q;
        Iterator<Quadrado> itr = c.iterator();
        while (itr.hasNext()) {
                q = itr.next();
                System.out.println(q);
        }
   }
}
```

```
[Quadrado de Centro (1.0,5.0) e de lado 4.0, Quadrado de Centro (3.0,4.0) e de lado 5.6,
Quadrado de Centro (0.0,0.0) e de lado 6.0, Quadrado de Centro (4.0,6.0) e de lado 7.4]
Quadrado de Centro (1.0,5.0) e de lado 4.0
Quadrado de Centro (3.0,4.0) e de lado 5.6
Quadrado de Centro (0.0,0.0) e de lado 6.0
Quadrado de Centro (4.0,6.0) e de lado 7.4
```

Ordem OK

```
import java.util.Comparator;
import java.util.TreeSet;

class MyComp implements Comparator<String> {
  public int compare(String a, String b) {
       return (a.length() > b.length() ? 1: -1);
  }
}

public class Teste {
  public static void main(String args[]) {
    TreeSet<String> ts = new TreeSet<String>(new MyComp());

    ts.add("jgdshj");
    ts.add("hj");
    ts.add("khsdfk jjskfk");
    ts.add("f");
    ts.add("opeiwoj kn kndsjsa");
    ts.add("kndkd");

    for (String element : ts)
      System.out.println(element + " ");
  }
}
```

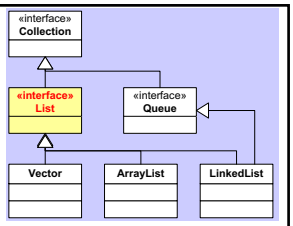
```
(a,b) -> a.length() > b.length() ? 1: -1
```

Using FI

```
f
hj
kndkd
jgdshj
khsdfk jjskfk
opeiwoj kn kndsjsa
```

19

Listas



- Podem conter elementos duplicados.
- Para além das operações herdadas de Collection, a interface lista inclui ainda:
  - Acesso Posicional – manipulação de elementos baseada na sua posição (índice) na lista
  - Pesquisa – de determinado elemento na lista. Retorna a sua posição.
  - ListIterator – estende a semântica do Iterator tirando partido da natureza sequencial da lista.
  - Range-View – execução de operações sobre uma gama de elementos da lista. (list.subList(fromIndex, toIndex).clear();)

20

List Interface

```
public interface List<E> extends Collection<E> {
   // Positional Access
   boolean add(E e)
   void add(int index, E element);            // Optional
   E get(int index);
   E set(int index, E element);               // Optional
   E remove(int index);                       // Optional
   boolean addAll(Collection<? extends E> c); // Optional

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   // Range-view
   List<E> subList(int from, int to);
}
```

```
public interface ListIterator<E>
                    extends Iterator<E> {
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();
   int nextIndex();
   int previousIndex();
   void remove();  //optional
   void set(E e);   //optional
   void add(E e);  //optional
}
```

Listas - Implementações

- ArrayList - Array redimensionável
- LinkedList - Listas Ligadas

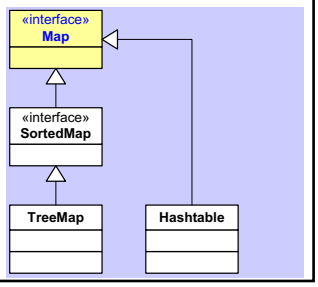
Diferença?

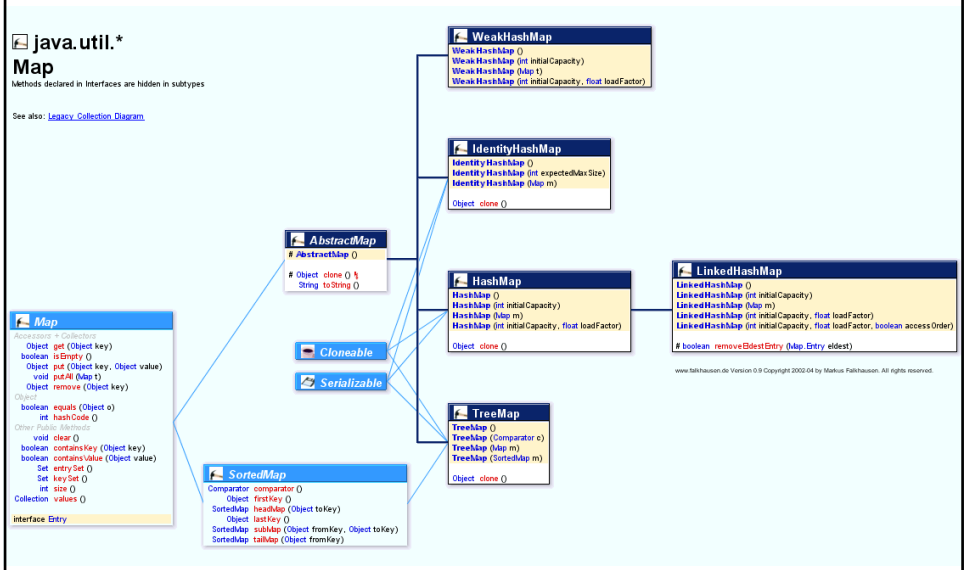
```
public static void main(String args[]) {
   String[] str1 = {"Rui", "Manuel", "Jose", "Pires", "Eduardo", "Santos"};
   String[] str2 = {"Rosa", "Pereira", "Rui", "Vidal", "Hugo", "Maria"};
   List<String> larray = new ArrayList<String>();
   List<String> llist = new LinkedList<String>();

   for (String i: str1 )  larray.add(i);

   for (String i: str2 ) llist.add(i);

   llist.addAll(llist.size()/2, larray);
   ListIterator itr = llist.listIterator();
   while ( itr.hasNext() )
   System.out.println( itr.next() );

    System.out.println("Rui está na posição " +
      llist.indexOf("Rui") + " e " + llist.lastIndexOf("Rui"));

   llist.set(llist.lastIndexOf("Rui"), "Rui2");
   System.out.println(llist.lastIndexOf("Rui"));
}
```

```
Rosa
Pereira
Rui
Rui
Manuel
Jose
Pires
Eduardo
Santos
Vidal
Hugo
Maria
Rui está na posição 2 e 3
2
```

Mapas - Map

- A Interface Map não descende de Collections.
  - Interface Map<K,V>
- Um mapa é um objecto que associa uma chave (K) a um único valor (V)
  - Não contém keys duplicadas
- Também é denominado como dicionário ou memória associativa
- Métodos disponíveis:
  - adicionar: put(Object key, Object value)
  - remover : remove(Object key)
  - obter um objecto: get(Object key)





24

Interface Map<K,V>

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Vistas

25

Vistas

- Mapas não são Collections.
- No entanto, podemos obter vistas dos mapas.
- As vistas são do tipo Collection
- Há três vistas disponíveis:
  - conjunto (set) de chaves
  - colecção de valores
  - conjunto (set) de entradas do tipo par chave/valor

26

Implementações de Map

- HashMap
  - Utiliza uma tabela de dispersão (Hash Table)
  - Não existe ordenação nos pares.
- LinkedHashMap
  - Semelhante ao HashMap, mas preserva a ordem de inserção
- TreeMap
  - Baseado numa árvore balanceada
  - Os pares são ordenados com base na chave - o acesso é O(log N)

27

HashMap

```
public static void main(String[] args) {
    Map<String, Double> mapa = new HashMap<>();
    mapa.put("Rui", 32.4);
    mapa.put("Manuel", 3.2);
    mapa.put("Rita", 5.6);

    System.out.println("O Mapa contém " + mapa.size() + " elementos");
    System.out.println("O Rui está no Mapa? " + mapa.containsKey("Rui"));

    System.out.println("O Rita tem " +  mapa.get("Rita") + "€");
    mapa.put("Rita", mapa.get("Rita") + 3.6);
    System.out.println("O Rita tem " +  mapa.get("Rita") + "€");

    Set<Entry<String, Double>> set = mapa.entrySet();
    Iterator<Entry<String, Double>> i = set.iterator();
    while(i.hasNext()) {
       Entry<String, Double> aux = i.next();
       System.out.println("O " + aux.getKey() + " ganha " + aux.getValue() + "€");
    }
}
```

```
O Mapa contém 3 elementos
O Rui está no Mapa? true
O Rita tem 5.6€
O Rita tem 9.2€
O Manuel ganha 3.2€
O Rui ganha 32.4€
O Rita ganha 9.2€
```

Vista

28

TreeMap

- Mesmas características das descritas para a TreeSet mas adaptadas a pares key/value.
- No exemplo anterior, só necessitamos de subtituir HashMap por TreeMap

```
public static void main(String[] args) {
    Map<String, Double> mapa =  new TreeMap<>();
    mapa.put("Rui", 32.4);
    ...
    }
}
```

- TreeMap oferece a possibilidade de ordenar objectos
  - utilizando a "Ordem Natural" (compareTo) ou um objecto do tipo Comparator

29

Ordenação em Colecções

1. Implementações com ordenação (TreeSet, TreeMap).
2. Utilizando o método static Collections.sort()

Há duas formas de definir uma ordem (key) de objectos:

- **Ordem Natural**
  - Cada Classe ao implementar a interface Comparable.
  - Método: int compareTo(Object o)
- **Utilizando o Comparator**
  - Se um objecto não tem ordem natural e/ou pretendemos definir uma nova ordem arbitrária

```
interface Comparator<T> {
   int compare(T o1, T o2)
   boolean equals(Object obj)
}
```

30

TreeMap Ordenado

```
class StringLenComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        if (s1 == null || s2 == null)
            throw new NullPointerException();
        return s1.length() - s2.length();
    }
}

public class TestTreeMap {
    public static void main(String[] args) {
        Map<String, Double> mapa =
            new TreeMap<>(new StringLenComparator());
        mapa.put("Rui", 32.4);
        ...
        }
    }
}
```

```
O Mapa contém 3 elementos
Rui está no Mapa? True

O Rui ganha 32.4€
O Rita ganha 9.2€
O Manuel ganha 3.2€
```

Ordenação

31

Collections sort()

- Para ordenar uma colecção não ordenada

```
public class TestArrayLinkedListSorted {
   public static void main(String args[]) {
       String[] str1 = {"Rui", "Manuel", "Jose",
                        "Pires", "Eduardo", "Santos"};
        List<String> list = new LinkedList<>();
        list.addAll(Arrays.asList(str1));

        Collections.sort(list, new StringLenComparator());

        ListIterator<String> itr = list.listIterator();
        while (itr.hasNext())
            System.out.println(itr.next());

    }
}
```

```
Rui
Jose
Pires
Manuel
Santos
Eduardo
```

32

Collections sort()

- Outra forma ... Classe Anónima

```
public class TestArrayLinkedListSorted {
   public static void main(String args[]) {
       String[] str1 = {"Rui", "Manuel", "Jose",
                        "Pires", "Eduardo", "Santos"};
        List<String> list = new LinkedList<>();
        list.addAll(Arrays.asList(str1));
        Collections.sort(list, new Comparator<String>()
                @Override
                public int compare(String s1, String s2) {
                    if (s1 == null || s2 == null)
                        throw new NullPointerException();
                return s1.length() - s2.length();
             }
        });
        for (String s: list) // equivalente ao anterior
            System.out.println(s);
    }
}
```

```
Rui
Jose
Pires
Manuel
Santos
Eduardo
```

33

Collections sort()

- Outra forma ainda ... Lambda expressions

```
public class TestArrayLinkedListSorted {
   public static void main(String args[]) {
       String[] str1 = {"Rui", "Manuel", "Jose",
                        "Pires", "Eduardo", "Santos"};
        List<String> list = new LinkedList<>();
        list.addAll(Arrays.asList(str1));
        Collections.sort(list, (s1,s2) -> {
                    if (s1 == null || s2 == null)
                        throw new NullPointerException();
                    return s1.length() - s2.length();}
        });
        for (String s: list) // equivalente ao anterior
            System.out.println(s);
    }
}
```

```
Rui
Jose
Pires
Manuel
Santos
Eduardo
```

34

Collections sort()

- E ainda!... utilizando a Java Stream API*

```
public class TestArrayLinkedListSorted {
   public static void main(String args[]) {
       String[] str1 = {"Rui", "Manuel", "Jose",
                        "Pires", "Eduardo", "Santos"};
        List<String> list = new LinkedList<>();
        list.addAll(Arrays.asList(str1));

 Collections.sort(list, Comparator.comparing(String::length));

        for (String s: list) // equivalente ao anterior
            System.out.println(s);
    }
}
```

Rui
Jose
Pires
Manuel
Santos
Eduardo

Method Reference!

*described in the next slides…

Algoritmos

- A JCF fornece ainda um conjunto de algoritmos que podem ser usados em colecções
  - Métodos estáticos de utilização global
- Exemplos:
  - sort, binarySearch, copy, shuffle, reverse, max, min, etc.
- java.util.Collections
- java.util.Arrays

java.util.Collections

Collections

+binarySearch(list: List, key: Object) : int
+binarySearch(list: List, key: Object, c: Comparator) : int
+copy(src: List, des: List) : void
+enumeration(c: final Collection) : Enumeration
+fill(list: List, o: Object) : void
+max(c: Collection) : Object
+max(c: Collection, c: Comparator) : Object
+min(c: Collection) : Object
+min(c: Collection, c: Comparator) : Object
+nCopies(n: int, o: Object) : List
+reverse(list: List) : void
+reverseOrder() : Comparator
+shuffle(list: List) : void
+shuffle(list: List, rnd: Random) : void
+singleton(o: Object) : Set
+singletonList(o: Object) : List
+singletonMap(key: Object, value: Object) : Map
+sort(list: List) : void
+sort(list: List, c: Comparator) : void
+synchronizedCollection(c: Collection) : Collection
+synchronizedList(list: List) : List
+synchronizedMap(m: Map) : Map
+synchronizedSet(s: Set) : Set
+synchronizedSortedMap(s: SortedMap) : SortedMap
+synchronizedSortedSet(s: SortedSet) : SortedSet
+unmodifiedCollection(c: Collection) : Collection
+unmodifiedList(list: List) : List
+unmodifiedMap(m: Map) : Map
+unmodifiedSet(s: Set) : Set
+unmodifiedSortedMap(s: SortedMap) : SortedMap
+unmodifiedSortedSet(s: SortedSet) : SortedSet

java.util.Arrays

Arrays

+asList(a: Object[]) : List
+binarySearch(a: byte[],key: byte) : int
+binarySearch(a: char[], key: char) : int
+binarySearch(a: double[], key: double) : int
+binarySearch(a,: float[] key: float) : int
+binarySearch(a: int[], key: int) : int
+binarySearch(a: long[], key: long) : int
+binarySearch(a: Object[], key: Object) : int
+binarySearch(a: Object[], key: Object, c: Comparator) : int
+binarySearch(a: short[], key: short) : int
+equals(a: boolean[], a2: boolean[]) : boolean
+equals(a: byte[], a2: byte[]) : boolean
+equals(a: char[], a2: char[]) : boolean
+equals(a: double[], a2: double[]) : boolean
+equals(a: float[], a2: float[]) : boolean
+equals(a: int[], a2: int[]) : boolean
+equals(a: long[], a2: long[]) : boolean
+equals(a: Object[], a2: Object[]) : boolean
+equals(a: short[], a2: short[]) : boolean
+fill(a: boolean[], val: boolean) : void
+fill(a: boolean[], fromIndex: int, toIndex: int, val: boolean) : void

Overloaded fill method for char, byte, short, int, long, float, double, and Object.

+sort(a: byte[]) : void
+sort(a: byte[], fromIndex: int, toIndex: int) : void

Overloaded sort method for char, short, int, long, float, double, and Object.

Exemplo

```
String[] str1 = {"Rui", "Manuel", "Jose",
                 "Pires", "Eduardo", "Santos"};
List<String> list = new ArrayList<>();
list.addAll(Arrays.asList(str1));
Collections.sort(list, new Comparator<String>() {
     @Override
     public int compare(String s1, String s2) {
        if (s1 == null || s2 == null)
            throw new NullPointerException();
        return s1.compareTo(s2);
     }
});

for (String s: list)
   System.out.println(s);

for (int i=0; i<str1.length; i++)
   System.out.println(Collections.binarySearch(list,str1[i]));
```

```
Eduardo
Jose
Manuel
Pires
Rui
Santos
4
2
1
3
0
5
```

39

Collections in Java 8

Stream API

40

Method References

- Treating an existing method as an instance of a Functional Interface
  - Examples

```
class Person {
  private String name;
  public String getName() { return name; }
}

Person[] people = ...;
Comparator<Person> byName =
       Comparator.comparing(Person::getName);
Arrays.sort(people, byName);
```

  - More Examples

```
Consumer<Integer> b1 = System::exit;
Consumer<String[]> b2 = Arrays::sort;
Consumer<String> b3 = MyProgram::main;
Runnable r = MyProgram::main;
```

Method References

- A static method (ClassName::methName)
- An instance method of a particular static object (instanceRef::methName)
- A super method of a particular object (super::methName)
- An instance method of an arbitrary object of a particular type (ClassName::methName)
- A class constructor reference (ClassName::new)
- An array constructor reference (TypeName[]::new)- "Instance method of an arbitrary object" adds an argument of that type which becomes the receiver of the invocation

Traversing Collections

There are three ways to traverse collections:

1. Iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();   //optional
}
```

2. "for-each" e forEach (java 8)

```
for (Object o : collection) // for each
    System.out.println(o);

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.forEach(s -> System.out.println(s)); // forEach
// l.forEach(System.out::println); // forEach
```

3. Aggregate operations (java 8)

Aggregate Operations - Java 8 Streams API

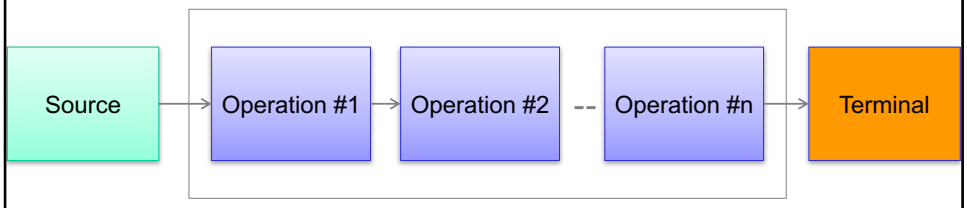
- The preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it.
- Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code.
- **Package java.util.stream**
  - The key abstraction introduced in this package is stream.
  - The classes Stream, IntStream, LongStream, and DoubleStream are streams over objects and the primitive int, long and double types.

java.util.stream

Streams differ from collections in several ways:

- No storage
  - A stream is not a data structure that stores elements; instead, it conveys elements through a pipeline of computational operations.
- Functional in nature
  - An operation on a stream produces a result, but does not modify its source.
- Laziness-seeking
  - Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. Intermediate operations are always lazy.
- Possibly unbounded
  - While collections have a finite size, streams need not.
- Consumable
  - The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Stream Pipeline



(1) Obtain a stream from a source

(2) Perform one or more intermediate operations

(3) Perform one terminal operation

46

java.util.stream - sources

- Streams can be obtained in a number of ways. Streams sources include:
  - From a **Collection** via the stream() and parallelStream() methods;
  - From an **Array** via Arrays.stream(Object[]);
  - From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);
  - The lines of a file can be obtained from BufferedReader.lines();
  - Streams of file paths can be obtained from methods in Files;
  - Streams of random numbers can be obtained from Random.ints();
  - Numerous other stream-bearing methods in the JDK, including BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().

java.util.stream - Intermediate operations

- .filter - excludes all elements that don't match a Predicate
- .map - perform transformation of elements using a Function
- .flatMap - transform each element into zero or more elements by way of another Stream
- .peek - performs some action on each element
- .distinct - excludes all duplicate elements (equals())
- .sorted - ordered elements (Comparator)
- .limit - maximum number of elements
- .substream - range (by index) of elements
- *(and many more -> see java.util.stream.Stream<T>)*

```
List<Person> persons = ...;
Stream<Person> tenPersonsOver18 = persons.stream()
                                .filter(p -> p.getAge() > 18)
                                .limit(10);
```

java.util.stream - Terminating operations

- Reducers
  - reduce(), count(), findAny(), findFirst()
- Collectors
  - collect()
- forEach
- iterators

```
List<Person> persons = ..;
List<Student> students = persons.stream()
                         .filter(p -> p.getAge() > 18)
                         .map(Student::new)
                         .collect(Collectors.toList());
```

Stream.Filter

- Filtering a stream of data is the first natural operation that we would need.
- Stream interface exposes a filter method that takes in a Predicate that allows us to use lambda expression to define the filtering criteria:

```
List<Person> persons = …
Stream<Person> personsOver18 =
  persons.stream().filter(p -> p.getAge() > 18);

// other Filter example with Predicate && Consumer

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.stream().filter(n -> n.length()>3)
          .forEach(System.out::println);
```

Stream.Map

- The map operations allows us to apply a function that takes in a parameter of one type, and returns something else.

```
Stream<Student> map = persons.stream()
        .filter(p -> p.getAge() > 18)
        .map(person -> new Student(person));


// other example with Map && Consumer

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.stream().map(n -> "Nome Pessoa:" + n)
          .forEach(System.out::println);
```

Stream.Reduce

- A *reduction* operation takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation
- For instance, finding the sum or maximum of a set of numbers, or accumulating elements into a list.

```
// example with Map & Reduce
List<Integer> costBeforeTax = Arrays.asList(100, 200, 300,
  400, 500);

double bill = costBeforeTax.stream()
         .map(cost -> (cost*1.23))
         .reduce(0.0,(sum, cost) -> sum + cost));

System.out.println("Total : " + bill);
```

Stream.Collect

- While stream abstraction is continuous by its nature, we can describe the operations on streams but to acquire the final results we have to collect the data somehow.
- The Stream API provides a number of "terminal" operations. The collect() method is one of those terminals that allows us to collect the results of the operations:

```
List<Student> students = persons.stream()
        .filter(p -> p.getAge() > 18)
        .map(Student::new)
        .collect(Collectors.toList());

// other example with Map && Collect
List<String> l = Arrays.asList("Ana", "Ze", "Rui");
List<String> res = l.stream()
        .map(n -> "Nome: " + n)
        .collect(Collectors.toList());
res.forEach(System.out::println);
```

Stream.Parallel and Sequential

- One interesting feature of the new Stream API is that it doesn't require the operations to be either parallel or sequential from beginning till the end.
- It is possible to start consuming the data concurrently, then switch to sequential processing and back at any point in the flow:

```
List<Student> students = persons.stream()
     .parallel()
     .filter(p -> p.getAge() > 18)
  // filtering will be performed concurrently
     .sequential()
     .map(Student::new)
     .collect(Collectors.toCollection(ArrayList::new));
```

Aggregate Operations - examples

- The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

- There are many different ways to collect data with this API. For example, you might want to convert the elements of a Collection to String objects, then join them, separated by commas:

```
String joined = elements.stream()
.map(Object::toString)
.collect(Collectors.joining(", "));
```

- Or perhaps sum the salaries of all employees:

```
int total = employees.stream()
.collect(Collectors.summingInt(Employee::getSalary)));
```

bulk operations

- The Collections framework has always provided a number of so-called "bulk operations" as part of its API.
- These include methods that operate on entire collections, such as containsAll, addAll, removeAll, etc.
- Do not confuse those methods with the aggregate operations that were introduced in JDK 8.
- The key difference between the new aggregate operations and the existing bulk operations (containsAll, addAll, etc.) is that the old versions are all mutative, meaning that they all modify the underlying collection.
- In contrast, the new aggregate operations do not modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

Sumário

- JAVA Collections FrameWork (JCF)
  - Organização e Principais Interfaces
  - Conjuntos (HashSet e TreeSet)
  - Listas (ArrayList e LinkedList)
  - Mapas (HashMap e TreeMap)
  - Operações sobre Colecções
- JAVA Stream API

57