Coleções

POO

Prof. Marcio Delamaro

Definição

- Interface Collection<E>

  E - the type of elements in this collection

- The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements.

Definição

- Interface Collection<E>

  E - the type of elements in this collection

- The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements.
- Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

Definição

- Interface Collection<E>
  E - the type of elements in this collection
- The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements.
- Some collections allow duplicate elements and others do not. Some are ordered and others unordered.
- The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like *Set* and *List*.

Definição

- Interface Collection<E>
  E - the type of elements in this collection
- The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements.
- Some collections allow duplicate elements and others do not. Some are ordered and others unordered.
- The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like *Set* and *List*.
- This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Operações

- https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
- *boolean add(E e)*
  Ensures that this collection contains the specified element
- *boolean remove(Object o)*
  Removes a single instance of the specified element from this collection, if it is present

Operações

- *boolean addAll(Collection<? extends E> c)*
  Adds all of the elements in the specified collection to this collection
- *boolean removeAll(Collection<?> c)*
  Removes all of this collection's elements that are also contained in the specified collection
- *boolean contains(Object o)*
- *boolean isEmpty()*
- *int size()*

Mais simples

- public class Vector<E> extends AbstractList<E> implements List<E>

Mais simples

- public class Vector<E> extends AbstractList<E> implements List<E>
- The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Array x Vector

- ```
  ContaBancaria contas[] = new ContaBancaria[100];
  int nContas = 0;
  ```
- ```
  Vector<ContaBancaria> contas = new
  Vector<ContaBancaria>();
  ```

Array x Vector

- ```
  ContaBancaria contas[] = new ContaBancaria[100];
  int nContas = 0;
  ```
- ```
  Vector<ContaBancaria> contas = new
  Vector<ContaBancaria>();
  ```
- É preciso estabelecer o tipo dos elementos que serão inseridos
- O tipo Vector<> é na verdade um tipo genérico em Java
- Que precisa ser parametrizado para que se crie um tipo real
  - Vector<E>

Array x Vector

- `ContaBancaria contas[] = new ContaBancaria[100];`
  `int nContas = 0;`
- `Vector<ContaBancaria> contas = new Vector<ContaBancaria>();`
- É preciso estabelecer o tipo dos elementos que serão inseridos
- O tipo Vector<> é na verdade um tipo genérico em Java
- Que precisa ser parametrizado para que se crie um tipo real
  - Vector<E>
- Número de contas é dado por `contas.size()`

O que mais muda?

- Adicionar uma conta na lista
- Como era

```
private void add(ContaBancaria cb) {
   contas[nContas++] = cb;
}
```

O que mais muda?

- Adicionar uma conta na lista
- Como era

```
private void add(ContaBancaria cb) {
    contas[nContas++] = cb;
}
```

- Como fica

```
private void add(ContaBancaria cb) {
    contas.add(cb);
}
```

O que mais muda?

- Procurar uma conta
- Como era

```
private ContaBancaria procura(int conta) {
  for (ContaBancaria ctb: contas ) {
    if ( ctb == null ) break;
    if (conta == ctb.getNumConta())
      return ctb;
  }
  return null;
}
```

O que mais muda?

- Procurar uma conta
- Como fica

O que mais muda?

- Procurar uma conta
- Como fica

```
private ContaBancaria procura(int conta) {
  for (ContaBancaria ctb: contas ) {
    if ( ctb == null ) break;
    if (conta == ctb.getNumConta())
      return ctb;
  }
  return null;
}
```

O que mais muda?

- Procurar uma conta
- Como fica

```
private ContaBancaria procura(int conta) {
  for (ContaBancaria ctb: contas ) {

    if (conta == ctb.getNumConta())
      return ctb;
  }
  return null;
}
```

Iteradores

- Vector, assim como outras coleções também implementam a interface Iterable
- Por isso, comando for pode ser usado
- Podemos ter uma forma mais explícita de iteração
- Usando objeto *Iterator*

Exemplo Iteradores

```
private void atualizaPoupança(double tx) {
   Iterator<ContaBancaria> it = contas.iterator();
   while  (it.hasNext())
   {
      ContaBancaria ctb = it.next();
      ctb.atualiza(tx);
   }
}
```

Exemplo Iteradores

```
private void atualizaPoupança(double tx) {
   Iterator<ContaBancaria> it = contas.iterator();
   while  (it.hasNext())
   {
      ContaBancaria ctb = it.next();
      ctb.atualiza(tx);
   }
}
```

Vector retorna um iterador

Exemplo Iteradores

```
private void atualizaPoupança(double tx) {
   Iterator<ContaBancaria> it = contas.iterator();
   while  (it.hasNext())
   {
      ContaBancaria ctb = it.next();
      ctb.atualiza(tx);
   }
}
```

Vector retorna um iterador

Verifica se ainda existem elementos a tratar

Exemplo Iteradores

```
private void atualizaPoupança(double tx) {
   Iterator<ContaBancaria> it = contas.iterator();
   while  (it.hasNext())
   {
      ContaBancaria ctb = it.next();
      ctb.atualiza(tx);
   }
}
```

Vector retorna um iterador

Verifica se ainda existem elementos a tratar

Obtém o próximo elemento

Conjuntos

- public interface Set<E> extends Collection<E>
- Uma interface pode ter uma superinterface

Conjuntos

- public interface Set<E> extends Collection<E>
- Uma interface pode ter uma superinterface
- A collection that contains no duplicate elements.
- More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element.
- As implied by its name, this interface models the mathematical set abstraction.

Operações conjuntos

- *boolean addAll(Collection<? extends E> c)*
  Adds all of the elements in the specified collection to this set if they're not already present. (União)
- *boolean retainAll(Collection<?> c)*
  Retains only the elements in this set that are contained in the specified collection. (Intersecção)
- *boolean removeAll(Collection<?> c)*
  Removes from this set all of its elements that are contained in the specified collection. (Diferença)

Implementações

- Set é uma interface que pode ser implementada de diversas maneiras
- EnumSet, HashSet, LinkedHashSet, TreeSet

Maps

- Interface Map<K,V>

  Type Parameters:

  K - the type of keys maintained by this map

  V - the type of mapped values

Maps

- Interface Map<K,V>

  Type Parameters:

  K - the type of keys maintained by this map

  V - the type of mapped values

- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

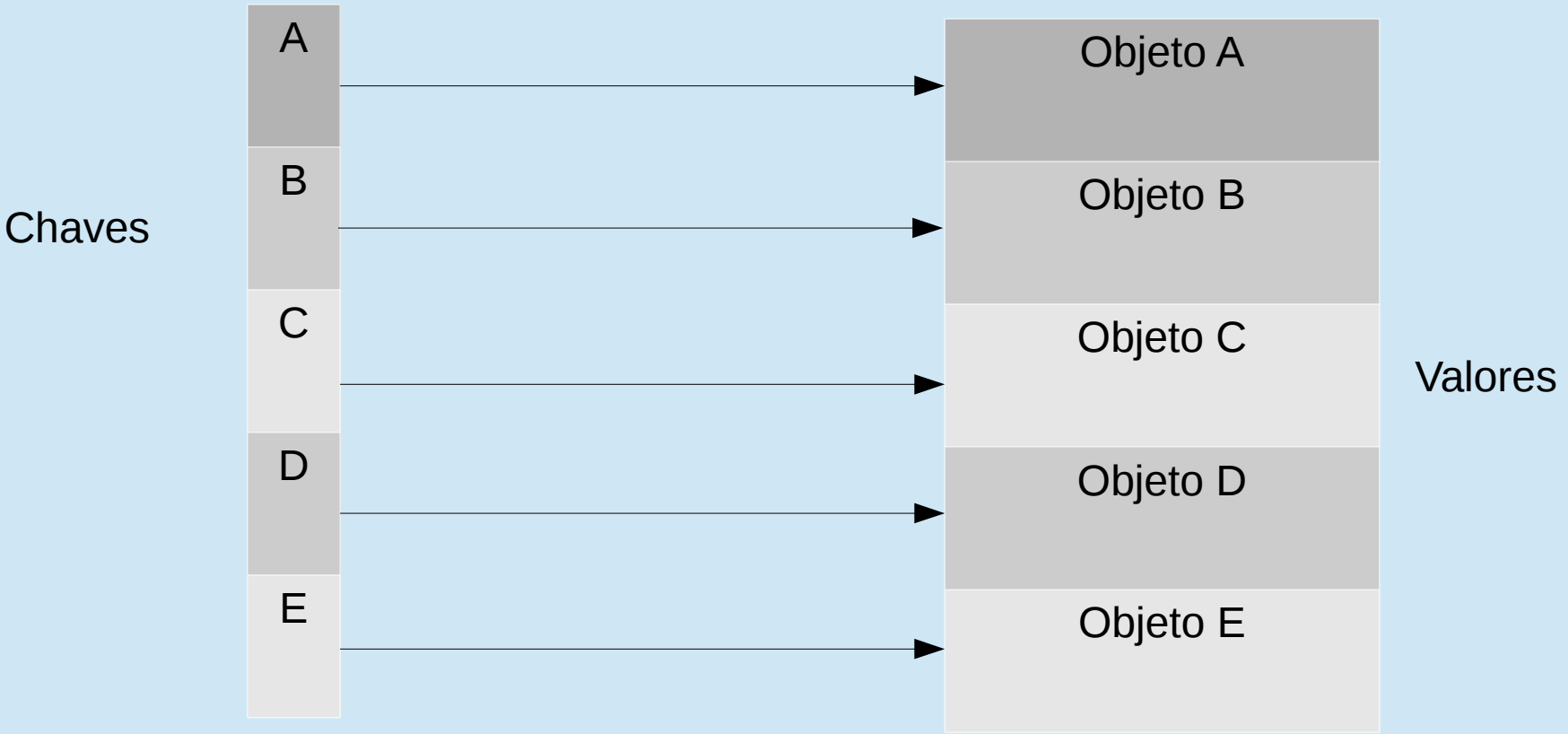
Maps

- Interface Map<K,V>
  Type Parameters:
  K - the type of keys maintained by this map
  V - the type of mapped values
- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
- The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

Pra que serve

- Um Map permite que se crie uma coleção com acesso direto a seus membros
- Acesso baseado numa chave
- NUSP → dados do aluno
- CPF → dados do cidadão
- ISBN → livro

Quer q desenha?



Sistema bancário

- Vamos mais uma vez alterar nossa classe gerenciadora: Contas
- Em vez de usa um vetor, vamos usar um *HashMap*
- Chave é o número da conta
- Assim, não precisamos procurar uma conta para fazer saques ou depósitos

Alterações em Contas

- Mudar a declaração da estruura usada
- ```
  private Vector<ContaBancaria> contas = new Vector<ContaBancaria>();
  private HashMap<Integer,ContaBancaria> contas = new HashMap<Integer,ContaBancaria>();
  ```
- Os elementos dos Maps e Collections precisam ser objetos
- Não podemos ter Vector<int> por exemplo
  ```

Adicionar uma conta

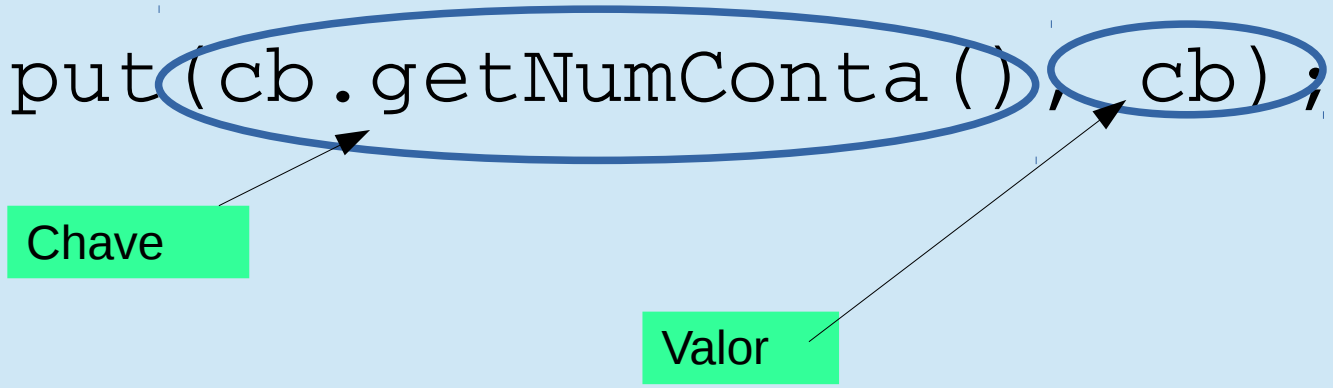
- Agora precisamos ter uma chave e uma conta

```
private void add(ContaBancaria cb) {
    contas.put(cb.getNumConta(), cb);
}
```

Adicionar uma conta

- Agora precisamos ter uma chave e uma conta

```
private void add(ContaBancaria cb) {
    contas.put(cb.getNumConta(), cb);
}
```



Mostrar saldos

- Requer percorrer todos os elementos

```
private void printSaldos() {
      for (ContaBancaria ctb : contas.values()) {
         System.out.println("Numero da conta:" +
ctb.getNumConta());
         System.out.println("Titular: " +
ctb.getNomeCliente());
         System.out.println("Saldo: " + ctb.getSaldo());
         System.out.println();
      }
}
```

Mostrar saldos

- Requer percorrer todos os elementos

```
private void printSaldos() {
      for (ContaBancaria ctb : contas.values()) {
         System.out.println("Numero da conta:" +
ctb.getNumConta());
         System.out.println("Titular: " +
ctb.getNomeCliente());
         System.out.println("Saldo: " + ctb.getSaldo());
         System.out.println();
      }
}
```

Retorna uma Collection que portanto possui um Iterator

Atualizar poupanças

- Percorrer usando Iterator

```
private void atualizaPoupança(double tx) {
      Iterator<ContaBancaria> it = contas.values().iterator();
      while  (it.hasNext())
      {
         ContaBancaria ctb = it.next();
         ctb.atualiza(tx);
      }
}
```

Atualizar poupanças

- Percorrer usando Iterator

```
private void atualizaPoupança(double tx) {
      Iterator<ContaBancaria> it = contas.values().iterator();
      while  (it.hasNext())
      {
         ContaBancaria ctb = it.next();
         ctb.atualiza(tx);
      }
}
```

Procurar uma conta

- Na verdade não precisa procurar uma conta pelo número
- O “vetor” de contas é indexado pelo número

```
private ContaBancaria procura(int conta) {
    return contas.get(conta);
}
```

Procurar uma conta

- Na verdade não precisa procurar uma conta pelo número
- O “vetor” de contas é indexado pelo número

```
private ContaBancaria procura(int conta) {
    return contas.get(conta);
}
```

Retorna null se a conta não existe

Exercício

- Crie uma classe ContaPalavra que tem:
  - um construtor que recebe o nome de um arquivo texto
  - um método criaMapa que cria uma mapa em que as chaves são as palavras do texto e os valores o número de ocorrências
  - um método mostraMapa que vai mostrar o número de ocorrências de cada palavra, em ordem alfabética

Aproveitando

- É possível salvar objetos em um arquivo para recuperá-los mais tarde
- Por exemplo, se quisermos salvar a lista de contas
- Um objeto ObjectOutputStream permite que gravemos objetos inteiros
- Um objeto ObjectInputStream permite que recuperemos objetos inteiros

Gravando objetos

```
FileOutputStream fos = new FileOutputStream("abc");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject("Meu string");
oos.close();
```

Lendo objetos

```
FileOutputStream fos = new FileOutputStream("abc");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject("Meu string");
oos.close();
```

```
FileInputStream fis = new FileInputStream("abc");
ObjectInputStream ois = new ObjectInputStream(fis);
String s = (String) ois.readObject();
ois.close();
```

Voltando ao banco

- No início do programa vamos tentar recuperar as contas da execução passada
- Vamos tentar ler objeto Contas do arquivo “contas.dat”
- No final da execução vamos salvar o objeto Contas no arquivo “contas.dat”

Salvando

```
FileOutputStream fos = new
    FileOutputStream("contas.dat");
ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(ct);
oos.close();
```

Salvando

```
FileOutputStream fos = new
    FileOutputStream("contas.dat");
ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(ct);
oos.close();
```

```
Exception in thread "main" java.io.NotSerializableException: Contas
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at Contas.main(Contas.java:110)
```

Serializable

- Para poder gravar o objeto ele precisa ser serializável
- Implementar interface Serializable
- `public class Contas implements Serializable`

Salvando

```
FileOutputStream fos = new
    FileOutputStream("contas.dat");
ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(ct);
oos.close();
```

Salvando

```
FileOutputStream fos = new
    FileOutputStream("contas.dat");
ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(ct);
oos.close();
```

```
Exception in thread "main" java.io.NotSerializableException: PoupancaOuro
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at java.util.HashMap.internalWriteEntries(HashMap.java:1777)
    at java.util.HashMap.writeObject(HashMap.java:1354)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

Serialização

- Todos os objetos “dentro” de Contas precisam ser serializáveis
- Temos um array de contas bancárias
- Todas as contas bancas devem ser serializáveis
- `public abstract class ContaBancaria implements Serializable`
- Isso faz com que todas as subclasses sejam serializáveis
- Essa interface não tem métodos