

Java Standard Edition (JSE)

13. Collections framework



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com



Agenda

- Revisão de dificuldades com Arrays;
- Trabalhando com Listas - List;
- Uso de Generics;
- Interfaces e coleções;
- Ordenação com Collections.sort;
- Exercícios;
- Trabalhando com conjuntos – Set
- Percorrendo coleções;
- Trabalhando com mapas - Map



Uso de Arrays

- O uso de Arrays em java gera alguns problemas:
- Precisamos definir um tamanho máximo
- Não podemos redefinir esse tamanho;
- Acesso a elementos baseado em índice;
- Precisamos implementar “algo” para sabermos quantas posições estão livres;
- Problemas para uso com BD;



Nova forma de armazenar

- Buscando resolver esses e outros problemas, a sun criou um novo conjunto de classes na versão 1.2;
- A esse conjunto, chamamos **Collection Frameworks**, ficando armazenado no pacote `java.util`;
- A nova API Collection é robusta e possui diversas classes que representam estruturas de dados avançadas



Listas: `java.util.List`

- Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre eles;
- Ela resolve os problemas de Arrays, como: busca, remoção e tamanho;
- A interface `java.util.List` define os comportamentos que uma classe deve implementar, para ser uma lista;



Implementações de List

- A implementação mais utilizada de List é a classe `ArrayList`;
- `ArrayList` é mais rápida na consulta de elementos que sua concorrente `LinkedList`;
- `ArrayList` **NÃO** é um array, apesar de possuir um array interno de controle, devidamente encapsulado;



Criação de Listas

- Podemos utilizar a sintaxe direta:
 - `ArrayList lista = new ArrayList();`
- Mas é sempre preferível usar a definição mais abstrata possível:
 - `List lista = new ArrayList();`
- Para adicionarmos elementos a uma lista, chamamos o método `add(Object)`
 - `Lista.add("João");`



O método List.add(Object)

- O método `add(Object)` recebe um objeto e o inclui no final da lista;
- A sobrecarga `add(int, Object)` recebe, também, um inteiro informando a posição em que o objeto deve ser inserido;
- Em nenhum momento informamos o tamanho da lista;

Trabalhando com objetos

- Uma vez que List(e toda API Collection) recebe Object, podemos manipular qualquer subclasse de Object;

```
public static void main(String[] args) {  
    ContaBancaria c1 = new ContaBancaria(100);  
    ContaBancaria c2 = new ContaBancaria(200);  
    List contas = new ArrayList();  
    1 contas.add(c1);  
    contas.add(c2);  
    System.out.println("Tamanho da lista: " + 2 contas.size());  
}
```

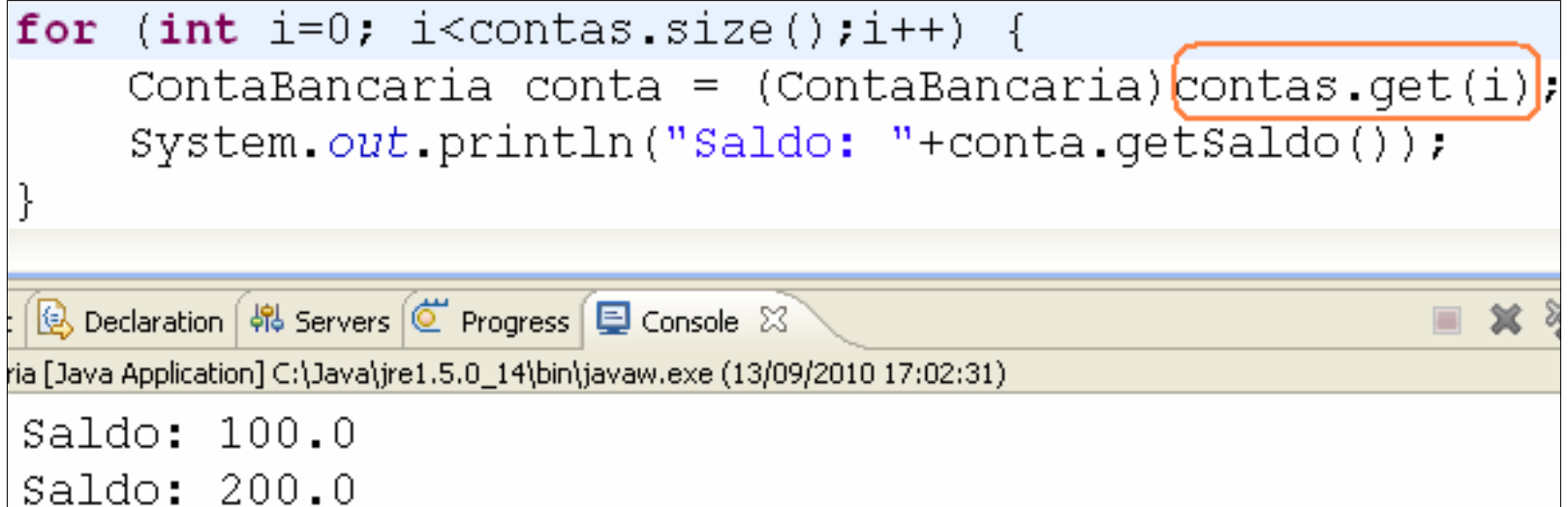
3 Tamanho da lista: 2



Acesso ao saldo das contas

- O método `List.get(int i)` devolve um `Object` armazenado na posição `i`;

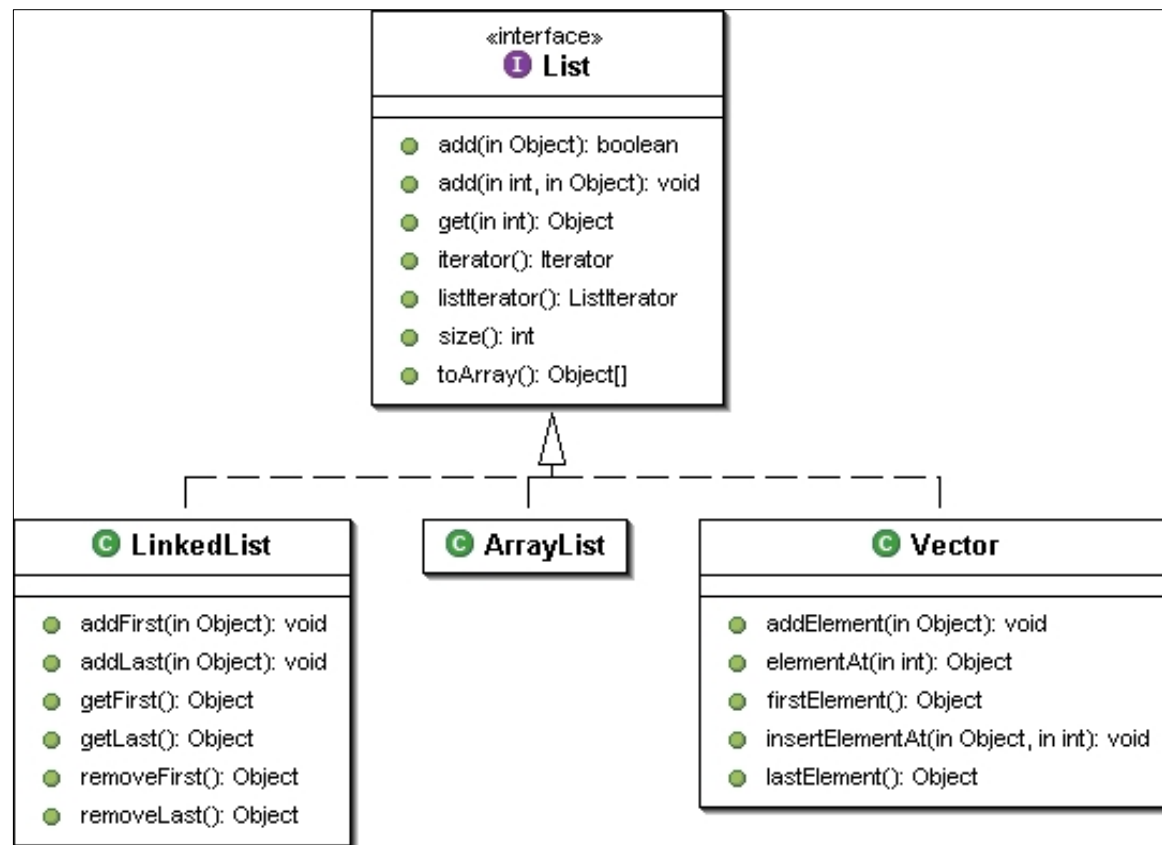
```
for (int i=0; i<contas.size();i++) {  
    ContaBancaria conta = (ContaBancaria)contas.get(i);  
    System.out.println("Saldo: "+conta.getSaldo());  
}
```



ria [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (13/09/2010 17:02:31)

Saldo: 100.0
Saldo: 200.0

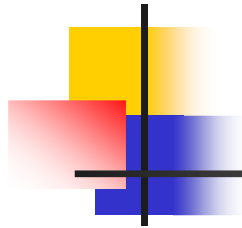
Outras implementações de List





A implementação LinkedList

- Também é muito utilizada;
- Métodos adicionais para obter e remover o primeiro e o último elemento
- Funcionamento interno diferente de ArrayList, o que pode ter impacto direto na performance da aplicação;



A implementação Vector

- Outra implementação tradicional;
- Desde a versão 1.0 do java;
- Foi adaptada para uso com Collections, passando a incluir novos métodos;
- Mais lenta que ArrayList quando não há acesso simultâneo aos dados;



Delimitando o escopo de List

- Em List, podemos incluir qualquer Object;
- Com isso, podemos misturar os objetos:

```
1 ContaBancaria c1 = new ContaBancaria(100);  
2 List lista = new ArrayList();  
  lista.add(c1);  
3 String texto = "Curso de JSE";  
  lista.add(texto);
```



Itens a ponderar

- Como fica o momento de recuperar os objetos da lista?
- Como fica o Casting desses objetos?
- É pouco comum precisarmos de uma lista com objetos de tipos diferentes;
- No java 1.5, surgiu o recurso de Generics, que nos permite restringir as listas a um determinado tipo;



Uso de Generics

- O <parâmetro> indica o tipo de objeto;
- Só podemos adicionar <ContaBancaria>
- Não precisamos mais de Casting;

```
public static void main(String[] args) {  
    1 List<ContaBancaria> lista = new ArrayList<ContaBancaria>();  
    ContaBancaria c1 = new ContaBancaria(100); 2  
    lista.add(c1);  
  
    3 String texto = "Curso de JSE";  
    lista.add(texto); //Não compila mais  
  
    4 for (int i=0; i<lista.size();i++) {  
        ContaBancaria conta = lista.get(i); //Sem Casting  
        System.out.println("Saldo: "+conta.getSaldo());  
    }  
}
```


Referenciar interfaces ou implementações?

- É comum encontrarmos referências à interface `List`, ao invés de uma de suas implementações, como `ArrayList`;
- Imaginemos o método que devolve uma coleção de contas:

```
public 1 ArrayList<ContaBancaria> listarContas() {  
    2 ArrayList<ContaBancaria> lista = new 3 ArrayList<ContaBancaria>();  
    //Implemente aqui o código para carregar a lista  
    4 return lista;  
}
```



Itens a ponderar

- Por que precisamos retornar uma referência a um `ArrayList`?
- No dia em que precisarmos devolver um `LinkedList`, ao invés de um `ArrayList`, será necessário alterar as classe que usam o método `listarContas()`;
- Como seria possível melhorar o código anterior?



Assinatura de métodos

- Em OO, é uma boa prática deixar a assinatura de método o mais genérica possível;
- Com isso, podemos mudar sua lógica, sem afetar outras classes;
- As classes que chamam `listarContas()` não sabem qual implementação estamos usando: `ArrayList` ou `LinkedList`;



Uso do polimorfismo – mudança de implementação

- Usando a interface, podemos mudar a implementação, de forma transparente:

```
public List<ContaBancaria> listarContas(){  
    List<ContaBancaria> lista = new ArrayList<ContaBancaria>();  
    //Implemente aqui o código para carregar a lista  
    return lista;  
}
```

- Ou:

```
public List<ContaBancaria> listarContas(){  
    List<ContaBancaria> lista = new LinkedList<ContaBancaria>();  
    //Implemente aqui o código para carregar a lista  
    return lista;  
}
```



Ordenação de coleções

- É comum a necessidade de armazenar objetos em estruturas como List;
- Mas outro fato comum, é a necessidade de **ordenação** desses objetos;
- A literatura nos oferece alguns algoritmos para resolver esse problema;
- E o java te oferece um método para ordenação das suas **Coleções**;

Ordenação de coleções

```
public static void main(String[] args) {  
    List<String> lista = new ArrayList<String>();  
    1 lista.add("Bianca");  
    lista.add("David");  
    lista.add("André");  
    //Chamada à sobreescrita ArrayList.toString()  
    System.out.println("Lista Inicial : "+lista);  
    2 //Ordenando a lista  
    Collections.sort(lista);  
    //Impressao do resultado  
    System.out.println("Lista Ordenada: "+lista);  
}
```

Javadoc Declaration Servers Progress Console X
BtaBancaria [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (14/09/2010 09:31:21)

```
3 Lista Inicial : [Bianca, David, André]  
  Lista Ordenada: [André, Bianca, David]
```



O que aconteceu?

- No **item 1**, carregamos nossa lista com objetos do tipo String;
- Em seguida, ocorre sua impressão, chamando **ArrayList.toString()**;
- No **item 2**, invocamos o método estático **Collections.sort()** e imprimimos a lista;
- No **item 3**, é exibido o resultado gerado no console;



Além das Strings

- No exemplo anterior, pudemos verificar que a lista foi impressa em ordem alfabética;
- Mas em um List, podemos colocar qualquer tipo de objeto;
- Como fazer para ordenar uma lista de Contas bancárias?
- Precisamos definir o **Critério de Ordenação**;



Critério de ordenação

- Trabalhando com ordenação, precisamos de uma forma de determinar como a lista ficará ordenada;
- Como fazer para ordenar uma lista de Contas bancárias?
- O `sort()` precisa **saber como comparar** objetos ContaBancaria, a fim de determinar a ordem da lista;



Comparação de objetos

- O `sort()` precisa que todos os objetos sejam **comparáveis**;
- Cada objeto ContaCorrente deve oferecer um método que o compara a outra conta;
- Com base em um método de comparação, o método **sort()** pode ordenar a lista pra voce;



Comparação de objetos

- Como o método `sort()` terá certeza de que um objeto é **comparável**?
- Pra isso, usaremos, novamente, o contrato de uma Interface;
- Os objetos da coleção devem implementar a interface `java.lang.Comparable`;
- Dessa forma, `sort()` terá garantida a comparação entre os objetos;



Java.lang.Comparable

- Comparable define o método
 - `int compareTo(Object);`
- Esse método deve retornar:
 - **zero**, se os objetos forem iguais;
 - um número **negativo**, se este objeto for **menor** que o objeto recebido;
 - e um número **positivo**, se este objeto for **maior** que o objeto recebido.

Tornando uma classe comparável

```
public class ContaBancaria implements Comparable<ContaBancaria>{  
    private String numero;  
    private String agencia;  
    private double saldo;  
  
    public int compareTo(ContaBancaria conta) {  
        if(this.saldo < conta.saldo){  
            return -1;  
        }else if(this.saldo > conta.saldo){  
            return 1;  
        }  
        // Se this.saldo == conta.saldo  
        return 0;  
    }  
  
    //mais código aqui...
```



Entendendo o código

- No item 1, atualizamos a declaração da classe `ContaBancaria`, a fim de informar que ela deve implementar a interface `Comparable`;
- No item 2, realizamos a implementação do método `compareTo()` definido pela interface;
- Com isso, nossa classe se tornou `Comparável`;



Exercício 01

- Sobrescreva o método toString() que a classe ContaBancaria herda de Object:

```
//Sobrescrita do método toString()  
public String toString() {  
    return ""+saldo;  
}
```

- Por que precisamos da concatenação?




Exercício 02

- Atualize a classe ContaBancaria, a fim de que se torne comparável:

```
public class ContaBancaria implements Comparable<ContaBancaria>{  
    private String numero;  
    private String agencia;  
    private double saldo;  
  
    public int compareTo(ContaBancaria conta) {  
        if(this.saldo < conta.saldo){  
            return -1;  
        }else if(this.saldo > conta.saldo){  
            return 1;  
        }  
        // Se this.saldo == conta.saldo  
        return 0;  
    }  
  
    //mais código aqui...
```


Exercício 03 – Ordenação



```
public static void main(String[] args) {  
    List<ContaBancaria> listaConta =  
        new ArrayList<ContaBancaria>();  
    ContaBancaria c1 = new ContaBancaria(500);  
    listaConta.add(c1);  
1   c1 = new ContaBancaria(100);  
    listaConta.add(c1);  
    c1 = new ContaBancaria(400);  
    listaConta.add(c1);  
2   System.out.println("Lista inicial");  
    System.out.println(listaConta);  
3   Collections.sort(listaConta);  
    System.out.println("Lista Ordenada");  
    System.out.println(listaConta);  
}
```

Javadoc Declaration Servers Progress Console

steOrdenacao [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (14/09/2010 10:37:54)

```
4   Lista inicial  
    [500.0, 100.0, 400.0]  
    Lista Ordenada  
    [100.0, 400.0, 500.0]
```



Explicando o código

- 01 – criação dos objetos ContaBancaria e inclusão na lista de contas;
- 02 – impressão da lista inicial;
- 03 – ordenação e impressão da nova lista;
- 04 – resultado das impressões;
- Como a JVM sabe que deve imprimir o saldo das contas?



O que mais?

- O critério de ordenação é definido pelo programador;
- O método `sort()` saberá ordenar a lista;
- Por que a ordenação funcionou quando usamos uma lista de Strings?
- Resp: A classe **String** implementa **Comparable** e seu método **compareTo()**
- O mesmo acontece com Integer, Double, Date, BigDecimal etc;



Vários critérios de ordenação

- E se tivermos a necessidade de ordenar as contas pelo número? Ou agência?
- Não queremos mexer na ordenação por saldo. E agora?
- O java possui outra interface, que nos permite criar vários critérios de ordenação;
- A interface é **Comparator**;



A interface Comparator

- Possui o método `int compare(obj, obj);`
- Vejamos uma implementação possível:

```
class NumeroContaComparator implements Comparator<ContaBancaria>{  
    public int compare(ContaBancaria conta1, ContaBancaria conta2) {  
        //Invocando o método String.compareTo(obj)  
        return conta1.getNumero().compareTo(conta2.getNumero());  
    }  
}
```

- Como a classe **String** é um **Comparable**, podemos delegar delegar essa atividade para seu método `compareTo()`;

Troca de critério de ordenação

```
public static void main(String[] args) {  
    List<ContaBancaria> listaConta =  
        new ArrayList<ContaBancaria>();  
    1 listaConta.add(new ContaBancaria("tres"));  
    listaConta.add(new ContaBancaria("_um"));  
    listaConta.add(new ContaBancaria("dois"));  
    System.out.println("Lista inicial");  
    System.out.println(listaConta);  
    2 Collections.sort(listaConta, new NumeroContaComparator());  
    System.out.println("Lista Ordenada");  
    System.out.println(listaConta);  
}
```

Javadoc Declaration Servers Progress Console
teOrdenacao [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (14/09/2010 14:32:04)

```
Lista inicial  
[tres, _um, dois]  
3 Lista Ordenada  
[_um, dois, tres]
```



Exercício 04

- Crie a classe Animal:

```
public class Animal {  
    private int codigo;  
    private String nome;  
    1 public Animal(int codigo, String nome) {  
        this.codigo = codigo;  
        this.nome = nome;  
    }  
    2 public String toString() {  
        return codigo+" - "+nome;  
    }  
    public int getCodigo() {  
    public void setCodigo(int codigo) {  
    public String getNome() {  
    public void setNome(String nome) {  
}
```



Exercício 05

- Torne a classe **Animal** comparável, implementando a interface **Comparable**:

```
public class Animal implements Comparable<Animal>{  
    //declaração de atributos aqui...  
    public int compareTo(Animal animal) {  
        if(this.codigo > animal.codigo){  
            return 1;  
        }else if(this.codigo < animal.codigo){  
            return -1;  
        } else{  
            return 0;  
        }  
    }  
    //Outros métodos aqui...
```


Exercício 06

- Teste a ordenação de Animal por código:

```
public class TesteAnimalComparable {  
    public static void main(String[] args) {  
        List<Animal> lista = new ArrayList<Animal>();  
        lista.add(new Animal(3, "cavalo"));  
        lista.add(new Animal(1, "peixe"));  
        lista.add(new Animal(2, "leao"));  
        System.out.println("Lista inicial\n"+lista);  
        Collections.sort(lista);  
        System.out.println("Lista final\n"+lista);  
    }  
}
```

Items @ Javadoc Declaration Servers Progress Console

ted> TesteAnimalComparable [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (14/09/2010 16:39:57)

```
Lista inicial  
[3 - cavalo, 1 - peixe, 2 - leao]  
Lista final  
[1 - peixe, 2 - leao, 3 - cavalo]
```



Exercício 07

- Agora, vamos criar um novo critério de comparação para ordenação da lista;
- Crie a classe para implementação da comparação por nome de Animal:

```
class NomeAnimalComparator implements Comparator<Animal>{  
  
    public int compare(Animal an1, Animal an2) {  
        //Delegando atividade para String.compareTo()  
        return an1.getNome().compareTo(an2.getNome());  
    }  
}
```

Exercício 08

- Teste a ordenação de Animal por nome:

```
public static void main(String[] args) {  
    List<Animal> lista = new ArrayList<Animal>();  
    lista.add(new Animal(3, "cavalo"));  
    lista.add(new Animal(1, "peixe"));  
    lista.add(new Animal(2, "leao"));  
    System.out.println("Lista inicial\n"+lista);  
    Collections.sort(lista, new NomeAnimalComparator());  
    System.out.println("Lista final\n"+lista);  
}
```

Javadoc Declaration Servers Progress Console

NomeAnimalComparable [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (14/09/2010 16:51:54)

```
Lista inicial  
[3 - cavalo, 1 - peixe, 2 - leao]  
Lista final  
[3 - cavalo, 2 - leao, 1 - peixe]
```

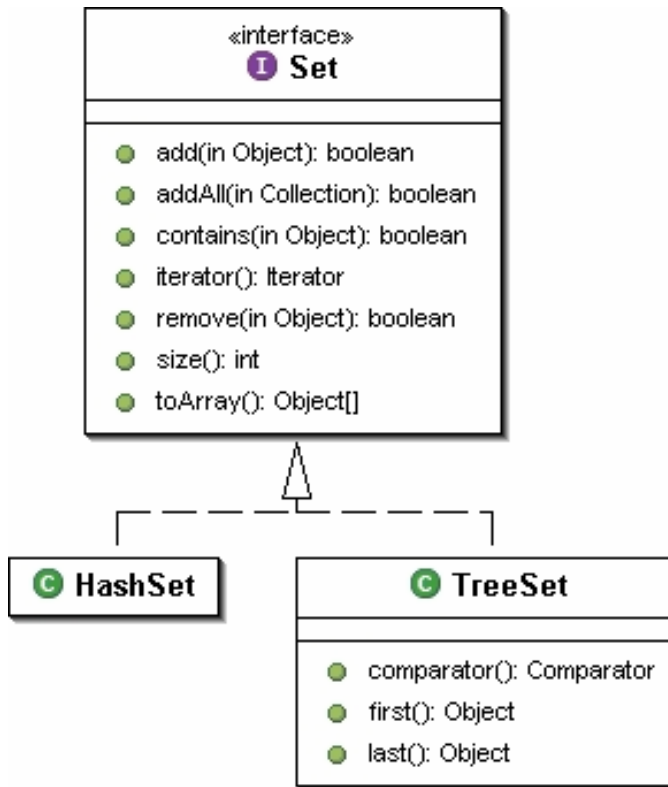


Trabalhando com Conjuntos

- Um conjunto(`java.util.Set`) é uma coleção que **NÃO** permite elementos duplicados;
- A ordem em que os elementos estão armazenados pode ser diferente da ordem de inclusão;
- Quem define esse comportamento é uma implementação da interface **Set**

A interface Java.util.Set

■ Implementações de Set:



Características de Set

- Tem como principais implementações:
 - HashSet, LinkedHashSet e TreeSet

```
public static void main(String[] args) {  
    Set<String> conjunto = new HashSet<String>();  
    1 conjunto.add("mamao");  
      conjunto.add("banana");  
      conjunto.add("uva");  
      //Não será adicionado  
    2 conjunto.add("mamao");  
      System.out.println(conjunto);  
}
```

Javadoc Declaration Servers Progress Console

teSet [Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (15/09/2010 08:53:15)

3 [mamao, uva, banana]



Explicando o código

- Item 01 – ocorre a criação de uma referência a **Set**, usando a implementação **HashSet** e a inclusão de objetos **String**;
- Item 02 – tentamos inserir novamente **mamão**. Neste ponto, o método `add()` devolve **false**;
- Item 03 – Mostra a saída impressa, onde aparece apenas uma vez **mamão**;



Considerações sobre Set

- Não armazena a ordem;
- Não aceita elementos repetidos;
- Não trabalha com índices, como `get(i)`;
- Mais rápido que uma List, quando usado para pesquisa;
- LinkedHashSet matem a ordem de inserção;
- TreeSet permite que a ordem seja definida(Comparable ou Comparator);



Java.util.Collection

- A interface **Collection** é a base para o trabalho com coleções em Java;
- Métodos definidos pela interface:

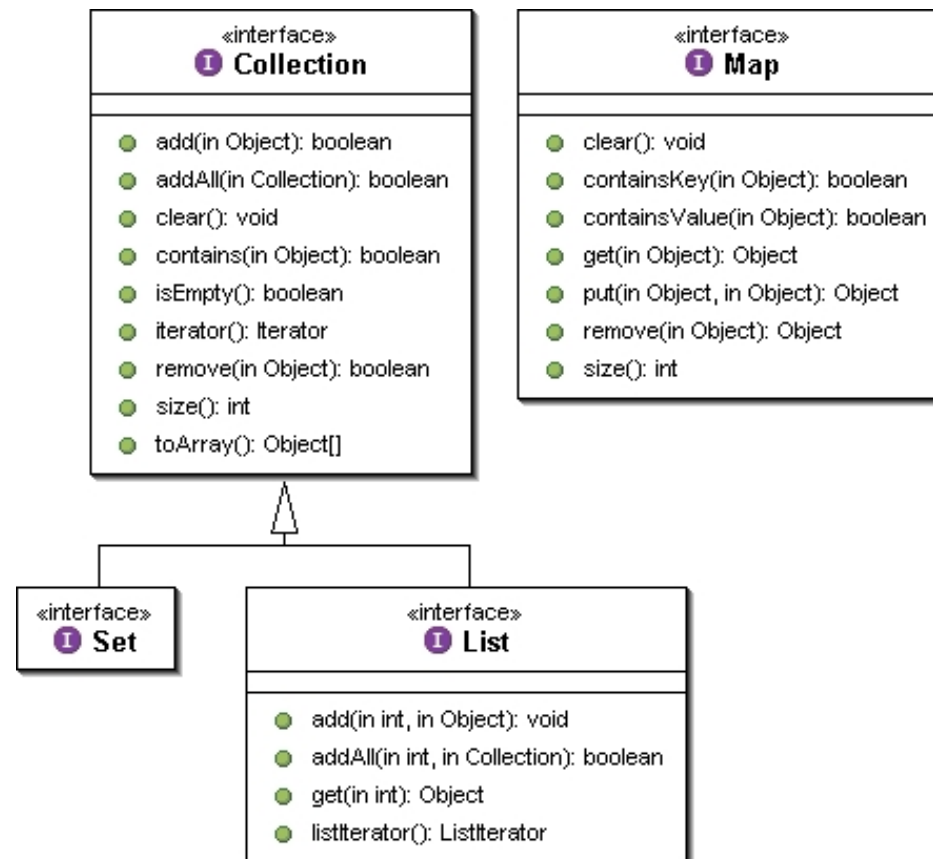
boolean add(Object)	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna true ou false indicando se a adição foi efetuada com sucesso.
boolean remove(Object)	Remove determinado elemento da coleção. Se ele não existia, retorna false.
int size()	Retorna a quantidade de elementos existentes na coleção.
boolean contains(Object)	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método equals() do objeto, e não através do operador ==.
Iterator iterator()	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

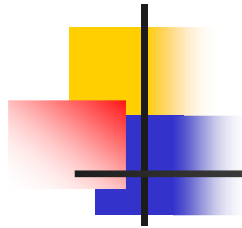


Características de Collection

- Uma coleção pode implementar diretamente a interface **Collection**;
- No geral, implementamos uma das duas subinterfaces mais famosas: **Set** e **List**;
- Set define um conjunto de elementos únicos;
- List permite objetos duplicados e guarda a ordem de inserção;

Visão geral do framework





Acesso a objetos da coleção

- Como percorrer os elementos de uma coleção?
- Se for uma lista, podemos utilizar um laço for, invocando o método `get(int)` para cada elemento;
- Mas e se a coleção não permitir indexação?
- Set não possui `get(int)`, por exemplo

Percorrendo a coleção

- Podemos usar “foreach” do Java 5 para percorrer qualquer Collection sem nos preocupar se é um List ou Set;

```
Set<String> conjunto = new HashSet<String>();  
conjunto.add("mamao");  
conjunto.add("banana");  
conjunto.add("uva");  
for (String fruta : conjunto) {  
    System.out.println(fruta);  
}
```

Declaration Servers Progress Console

Java Application] C:\Java\jre1.5.0_14\bin\javaw.exe (15/09/2010 13:27:22)

```
mamao  
uva  
banana
```



java.util.Iterator

- Antes do Java 5, as iterações em coleções eram baseadas em um **Iterator**
- Toda coleção fornece acesso a um iterator, um objeto que implementa a **interface Iterator**, que conhece internamente a coleção e dá acesso a todos os seus elementos
- O **foreach** é uma capa para o iterator;



Uso do Iterator

- No código abaixo, percorremos uma coleção usando um Iterator;

```
Set<String> conjunto = new HashSet<String>();  
conjunto.add("mamao");  
conjunto.add("banana");  
conjunto.add("uva");  
Iterator<String> it = conjunto.iterator(); 1  
  
while (it.hasNext()) { 2  
    3 //Sem casting? Por que?  
    String fruta = it.next();  
    System.out.println(fruta);  
}
```



Entendendo o código

- **Item 1** – Declaramos um iterator, para `Strings(generics)`, e o atribuímos ao objeto `Iterator` da coleção;
- **Item 2** – Indica que o laço deve existir enquanto houver elemento no iterator;
- **Item 3** – `it.next()` devolve uma referência a uma posição do `Iterator`;



Trabalhando com Mapas

- Em java, um mapa é composto por uma série de associações entre um **objeto chave** e um **objeto valor**;
- Ou seja, o mapa nos permite mapear uma **chave** a um **valor**;
- A API Java oferece a representação de mapas a partir da interface **java.util.Map**;



java.util.Map

- Método `put(chave, valor)` recebe um objeto chave e outro objeto valor para uma nova associação:
 - `meuMapa.put("Nome", "Joao");`
- Para saber o valor de uma associação, informamos o objeto chave:
 - `Object resultado = meuMapa.get("Nome");`

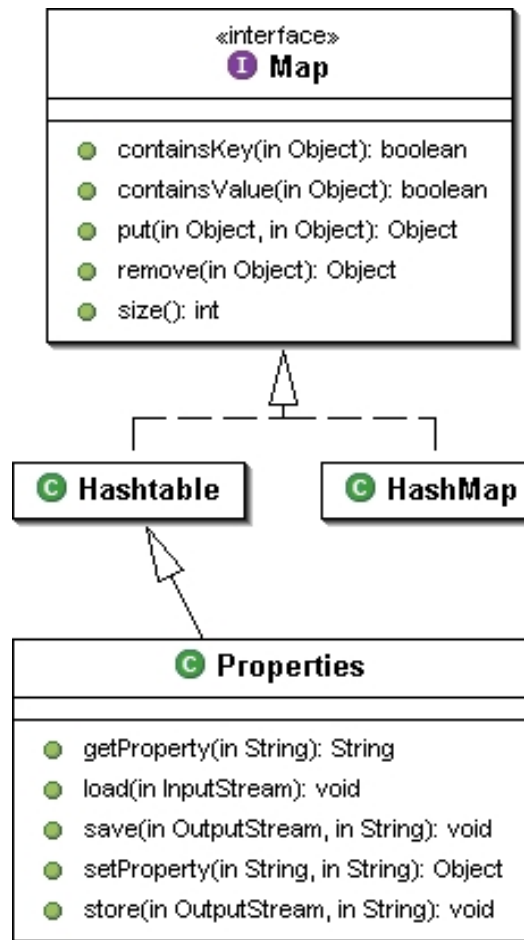


Uso de mapas

- O código a seguir, apresenta um exemplo de uso de mapas em Java:

```
ContaBancaria c1 = new ContaBancaria(500);  
ContaBancaria c2 = new ContaBancaria(250);  
1 //Criação do mapa  
Map mapaDeContas = new HashMap();  
  
2 //Inclusão de chave e valor  
mapaDeContas.put("João", c1);  
mapaDeContas.put("Maria", c2);  
  
3 //Acesso à conta de João  
Object objConta = mapaDeContas.get("João");  
ContaBancaria conta = (ContaBancaria)objConta;  
System.out.println(conta.getSaldo());
```

Implementações de Map





Acesso a mapas com generics

- Com uso do generics, não precisamos nos preocupar com o casting dos objetos:

```
ContaBancaria c1 = new ContaBancaria(500);
ContaBancaria c2 = new ContaBancaria(250);
//Criação do mapa
Map<String, ContaBancaria> mapaDeContas =
    new HashMap<String, ContaBancaria>();
//Inclusão de chave e valor
mapaDeContas.put("João", c1);
mapaDeContas.put("Maria", c2);
//Acesso à conta de João
ContaBancaria conta = mapaDeContas.get("João");
System.out.println(conta.getSaldo());
```



java.util.Properties

- Implementação de Map para o mapeamento entre Strings, usada em configuração de aplicações:

```
//Definição das propriedades
Properties config = new Properties();
config.setProperty("login", "scott");
config.setProperty("password", "tiger");
config.setProperty("url", "jdbc:mysql://localhost/teste");

// Acesso às propriedades salvas
String login = config.getProperty("database.login");
String password = config.getProperty("database.password");
String url = config.getProperty("database.url");
DriverManager.getConnection(url, login, password);
}
```



Bibliografia

- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra
- (Avançado) Effective Java Programming Language Guide, de Josh Bloch



Referências WEB

- SUN: www.java.sun.com

Fóruns e listas:

- Javaranch: www.javaranch.com
- GUJ: www.guj.com.br

Apostilas:

- Argonavis: www.argonavis.com.br
- Caelum: www.caelum.com.br

Java Standard Edition (JSE)

13. Collections framework



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com