CENTRO UNIVERSITÁRIO DE PATOS DE MINAS
DISCIPLINA: DESENVOLVIMENTO WEB II
EDUARDO HENRIQUE SILVA

COLEÇÕES

SUMÁRIO

1 COLLECTIONS

Embora possamos utilizar um *array* para armazenar um grupo de elementos do mesmo tipo, ele não permite uma alocação dinâmica dos elementos, temos que fixar o tamanho que não pode ser alterado uma vez já definido. Em Java, as estruturas de dados de alocação dinâmica são encontradas em uma arquitetura unificada chamada "*Collection Framework*".

Uma *Collection* é simplesmente um objeto que armazena uma coleção (grupo) de objetos. Cada item em uma coleção é chamando de elemento. O *Collection Framework* fornece interfaces unificadas para o armazenamento, recuperação e manipulação dos elementos. Ele permite que os programadores programem para interfaces invés da implementação.

A interface *Collection* define comportamentos comuns esperado de uma coleção, assim como, adicionar e remover elementos. A Figura 1 apresenta a hierarquia de interfaces do *Collection Framework*.

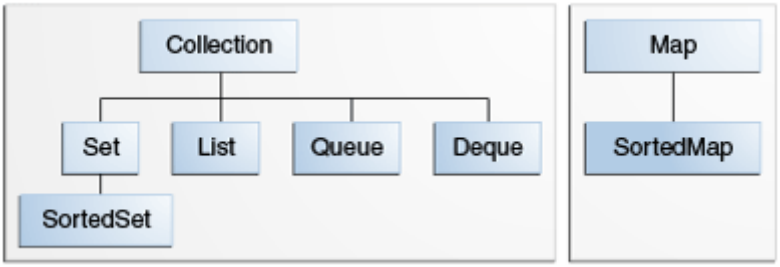

Figura 1 - Hierarquia de interfaces

O *Collection Framework* também fornece implementações para as interfaces. Iremos falar mais sobre as implementações de cada interface posteriormente.

O código abaixo apresenta uma *collection*.

```
Collection colecao = new ArrayList();// Implementação ArrayList

colecao.add("Curso Java");
colecao.add(1);
```

Nesse exemplo criamos uma coleção e adicionamos um elemento. Podemos adicionar elementos dinamicamente. Nesse exemplo temos um problema, veja que adicionamos um "*Integer*" e uma "*String*", em uma coleção deveríamos ter apenas objetos do

mesmo tipo, para facilitar a manipulação dos elementos e impedir que objetos de tipos diferentes sejam inseridos em nossa coleção deixando-a mais segura. Como no ato da criação do objeto não definimos o tipo dos elementos, então ele assume que poderá receber qualquer objeto "*add(Object)*" por ser a superclasse de todas as classe. Para definir o tipo dos elementos utilizaremos o *Generics*. Uma Collection poder conter apenas objetos e não tipos primitivos como "*int*" ou "*float*". Se você tentar inserir um "*int*" ele fará uma conversão implicitamente para a classe *Interger*.

1.1 GENERICS

O *Generics* permite você passar informações do tipo, utilizando o "<Tipo>", na referência do objeto. O compilador ao compilar pode realizar todas as checagem de tipos e garantir a segurança do tipo em tempo de execução.

O código abaixo apresenta a criação de uma coleção com Generics, nesse exemplo o compilador irá permitir que apenas "String" sejam armazenadas na coleção.

```
Collection<String> colecao = new ArrayList<String>();
colecao.add("Curso Java");
```

No Java 7 podemos passar o tipo apenas na referência do objeto ficando assim:

```
Collection<String> colecao = new ArrayList<>();
```

1.2 PERCORRENDO UMA COLEÇÃO

1.2.1 Interface *Iterator*

Usamos o termo "Iterar" para dizer que iremos percorrer os elementos de uma coleção através de uma estrutura de repetição.

A interface *Iterator* fornece métodos para iterar sobre qualquer *Collection*. O *Iterator* permite remover elementos da coleção durante a iteração.

Veja alguns métodos que podemos usar:

- *boolean hasNext():* Retorna "*true*" se ele tiver mais elementos;
- *E next():* Retorna o próximo elemento do tipo genérico E;
- *void remove():* Remove o último elemento retornado pelo *Iterator*.

O código abaixo apresenta um exemplo do *Iterator*.

```
Collection<String> colecao = new ArrayList<String>();
colecao.add("Curso Java");

Iterator<String> iterator = colecao.iterator();//Criando o Iterator

while(iterator.hasNext()){ // Loop para iterar até houver elementos
	String item = iterator.next(); //Acessando o elemento da coleção
	System.out.print(item);
}
```

1.2.2 Laço "*for*"

Quando percorremos uma *array*, geralmente utilizamos a estrutura de repetição "*for*". Em algumas coleções podemos usar essa estrutura de repetição como "*List*", porém em uma estrutura "Set" não é possível.

```
List<String> colecao = new ArrayList<String>();
colecao.add("Curso Java");

for(int i =0; i < colecao.size(); i++){ // Loop para iterar os elementos
	String item =colecao.get(i); //Acessando o elemento da coleção através de
sue índice
	System.out.print(item);
}
```

1.2.3 Laço "*for*" aprimorado

Além do uso do *Iterator* e do "*for*", podemos usar um novo e melhorado "*for*", que você pode utilizar para percorrer todos elementos de uma coleção ou *array*

Veja a sintaxe para o uso do "*for*" aprimorado.

```
for(Tipo item : coleção){
	//Seu código
}
```

A variável “item” do *loop* armazenará cada elemento da coleção, em cada iteração do *loop*.

Veja um exemplo do uso do “*for*” aprimorado.

```
List<String> colecao = new ArrayList<String>();
colecao.add("Curso Java");

for(String item : colecao){ // Loop para iterar os elementos
	System.out.print(item);//Acessando a variável item criada no for
}
```

1.3 A INTERFACE LIST E SUAS IMPLEMENTAÇÕES

List é uma coleção ordenada (ordenada pelo índice) que pode conter elementos duplicados e nulos. Você pode realizar operações como (recuperar, inserir, iterar) utilizando o índice (posição do elemento).

1.3.1 ArrayList

ArrayList é uma implementação de *List* que permite uma rápida iteração e um rápido acesso aleatório aos elementos da coleção através do seu índice. Exemplo:

```
List<String> colecao = new ArrayList<String>();
```

1.3.2 Vector

Vector é uma implementação de *List* basicamente igual ao *ArrayList* porém seus métodos são sincronizados para segurança em uso com *Threads*. Métodos sincronizados geralmente possui um custo maior de desempenho. Quando não precisar de segurança na utilização de coleções com *thread* opte pelo *ArrayList*. Exempo:

```
List<String> colecao = new Vector<String>();
```

1.3.3 LinkedList

LinkedList é uma implementação de *List* bem parecida com o *ArrayList*, exceto porque ela é uma lista duplamente encadeada. Esse encadeamento fornece novos métodos para inserir e remover no inicio ou no fim da coleção. A iteração de uma *LinkedList* é mais lenta do que do *ArrayList*, porém, a inserção e a remoção é mais rápida.

```
List<String> colecao = new LinkedList<String>();
```

1.4 A INTERFACE SET E SUAS IMPLEMENTAÇÕES

Set não permite elementos repetidos e permite apenas um elemento *null*. Um bom amido da interface *Set* é o método "*equals*", que determina se dois objetos são iguais. Se dois objetos iguais são adicionados na coleção, é mantido apenas um. Para que isso funcione é sempre bom implementar os métodos "*equals*" e "*hashCode*" dos elementos que serão adicionados na coleção, pois, se não implementar será utilizado os métodos da classe "*Object*" e ele pode não garantir a unicidade de um objeto.

1.4.1 HashSet

HashSet é uma implementação de *Set*, em que seus dados não são ordenados e nem classificados. Ele usa o "*hashCode*" do objeto que está sendo inserido, assim é sempre bom implementar o "*hashCode*" em seus objetos para melhorar o desempenho de acesso. Assim, você usa essa implementação quando não quiser elementos duplicados e quando não é necessário saber a ordem do elemento durante a iteração.

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

1.4.2 LinkedHashSet

LinkedHashSet é uma implementação de Set, ele é uma versão do *HashSet* ordenada e duplamente encadeada. Você utilizará o *LinkedHashSet* quando desejar saber a ordem do elemento durante a iteração. Enquanto você percorre um *HashSet* a ordem dos elementos é imprevisível, enquanto o *LinkedHashSet* deixa você percorrer os elementos na ordem em que eles foram inseridos.

```
Set<String> colecao = new LinkedHashSet<String>();
```

1.4.3 TreeSet

TreeSet é uma implementação de *Set*, que permite a classificação dos elementos. Ele garante que os elementos serão classificados em ordem ascendente, de acordo com a ordem natural. Você também pode criar um *TreeSet* com um construtor que permite você definir sua própria regra de classificação dos elementos usando as classes "*Comparable*" ou "*Comparator*".

```
Set<String> colecao = new TreeSet<String>();
colecao.add("Web");
colecao.add("Curso Java");

for(String item : colecao){
	System.out.print(item);
}
```

A saída do código acima é "Curso Java" e "Web", pois o *TreeSet* realizou a classificação por ordem alfabética, devido aos elementos serem uma "*String*". Números e cadeias de caracteres a ordenação é feita pela ordem natura. Se seus elementos não forem números ou cadeias de caracteres você deve definir a regra de ordenação pelo "*Comparable*" ou "*Comparator*". Veja o exemplo do uso do "*Comparator*" para ordenar os elementos de acordo com o tamanho da cadeia de caracteres.

Nesse exemplo a saída será "Web" e "Curso Java"

```
Set<String> colecao = new TreeSet<String>(new Comparator<String>() {
	@Override
	public int compare(String o1, String o2) {

		return o1.length() - o2.length();
	}
});
```

1.5 A INTERFACE MAP E SUAS IMPLEMENTAÇÕES

Map é uma coleção de pares com chave-valor. Em um *Map* não é possível ter chaves repetidas, mas os valores podem repetir. Para garantir a unicidade de uma chave você deve implementar o método "*equals*". Podemos pesquisar pelos valores de um *Map* usando a chave. Veja o exemplo de um Map que armazena o CPF e o nome de uma pessoa. Lembre-se a chave deve ser única, então nossa chave será o CPF.

```
Map<Long,String> pessoas;
```

No *Map*, primeiramente passamos o tipo da chave e depois o tipo do valor. Por exemplo, "*Long"* para CPF e "*String"* para o nome da pessoa.

A interface *Map* define os métodos:

- *Object get(Object chave):* Retorna o valor associado a chave;
- *Object put(Object chave, Object valor):* Adiciona o valor na respectiva chave, se a chave já tiver sida adicionada então apenas o valor é alterado;
- *boolean containsKey(Object chave):* Retorna "*true*" se a chave contém no *Map*.

1.5.1 HashMap

HashMap é uma implementação de *Map*, onde os elementos não são ordenados e nem classificados. Você o utilizará quando você iterar a coleção e não precisar saber a ordem

do elemento. Para uma melhor performance você deve implementar o "*hashCode*" de sua chave. Ele permite apenas uma chave nula e múltiplos valores nulos.

```
Map<Long,String> pessoas = new HashMap<Long,String>();
```

1.5.2 HashTable

HashTable é uma implementação de *Map*, é basicamente igual ao *HashMap*, porém, os seus métodos são sincronizados (igual ao *Vector*). Outra diferença é que o *HashTable* não permite nenhum valor nulo.

```
Map<Long,String> pessoas = new Hashtable<Long,String>();
```

1.5.3 LinkedHashMap

O *LinkedHashMap* é uma implementação de *Map*, que mantém os elementos na ordem de inserção. Em comparação com o *HashMap*, ele é mais lento na inserção e remoção, porém mais rápido na iteração.

```
Map<Long,String> pessoas = new LinkedHashMap<Long,String>();
```

1.5.4 TreeMap

O *TreeMap* é uma implementação de *Map*, igual ao *TreeSet*, o *TreeMap* é uma coleção classificada pela chave natural. Caso queira definir a regra de classificação poderá usar o "*Comparable*" e o "*Comparator*"

```
Map<Long,String> pessoas = new TreeMap<Long,String>();
```

1.6 PERCORRENDO UM MAP

A iteração de um *Map* é um pouco diferente, pois, devemos acessar os valores pela chave. Veja o exemplo de iteração:

```
Map<Long,String> pessoas = new HashMap<Long,String>();
pessoas.put(12312312312L, "Paulo");
pessoas.put(91291291291L, "João");

for (Long chave : pessoas.keySet()) {
	String valor = pessoas.get(chave);
	System.out.println(chave + " - "+valor);
}
```

Utilizamos o método "*keySet*()" para retornar um conjunto (*Set*) com todas as chaves do *Map*. Através da iteração de chave, recuperamos o valor através do método "*get*(chave)".

REFERÊNCIAS

FURGERI, Sergio. **Java 6 Ensino Didático**: Desenvolvendo e implementando aplicações. São Paulo: Érica, 2008.

HOCK-CHUAN, Chua. **Programming notes**. Disponível em: <http://www.ntu.edu.sg/home/ehchua/programming/index.html>. Acesso em: 04 jan. 2014.

SIERRA, Katherine; BATES, Bert. **SCJP Sun Certified Programmer**. Mcgraw-hill, 2008.

ORACLE. **The Java Tutorials**. Disponível em: < http://docs.oracle.com/javase/tutorial/>. Acesso em: 04 jan. 2014.