
Java Starter
www.t2ti.com

Apresentação

O Curso Java Starter foi projetado com o objetivo de ajudar àquelas pessoas que têm uma base de lógica de programação e desejam entrar no mercado de trabalho sabendo Java,

A estrutura do curso é formada por módulos em PDF e por mini-cursos em vídeo. O aluno deve baixar esse material e estudá-lo. Deve realizar os exercícios propostos. Todas as dúvidas devem ser enviadas para a lista de discussão que está disponível para inscrição na página do Curso Java Starter no site www.t2ti.com. As dúvidas serão respondidas pelos instrutores Albert Eije, Cláudio de Barros e Miguel Kojiio, além dos demais participantes da lista.

Nosso objetivo é que após o estudo do Curso Java Starter o aluno não tenha dificuldades para acompanhar um curso avançado onde poderá aprender a desenvolver aplicativos para Web, utilizando tecnologias como Servlets e JSP e frameworks como Struts e JSF, além do desenvolvimento para dispositivos móveis.

Albert Eije trabalha com informática desde 1993. Durante esse período já trabalhou com várias linguagens de programação: Clipper, PHP, Delphi, C, Java, etc. Atualmente mantém o site www.alberteije.com.

Cláudio de Barros é Tecnólogo em Processamento de Dados.

Miguel Kojiio é bacharel em Sistemas de Informação, profissional certificado Java (SCJP 1.5).

O curso Java Starter surgiu da idéia dos três amigos que trabalham juntos em uma instituição financeira de grande porte.

Módulo 09

Coleções

O Framework Collections

Dada sua importância, coleções (estruturas de dados) são encontradas em qualquer linguagem de programação e em Java as coleções recebem um tratamento especial, neste sentido o framework "Collections" surgiu com a intenção de formar uma arquitetura unificada para representação e manipulação de coleções.

Mas o que é o framework "Collections"? É o conjunto de implementações (classes e interfaces) oferecidas no pacote java.util que fornecem as principais funcionalidades esperadas quando trabalha-se com conjuntos de elementos (coleções).

Legal, mas o que este framework me permite fazer que eu não conseguiria com os vetores (arrays) convencionais? Nada, na verdade todas as funcionalidades implementadas no framework "Collections" podem ser obtidas através de uma implementação própria, ou seja, você pode até fazer a sua própria Lista, Fila ou Árvore mas sabe que isto já está pronto e acessível neste framework, na prática implementações próprias são difíceis de encontrar – afinal de contas tempo é um recurso valioso e o reuso é um dos pilares da orientação a objetos, lembra-se?.

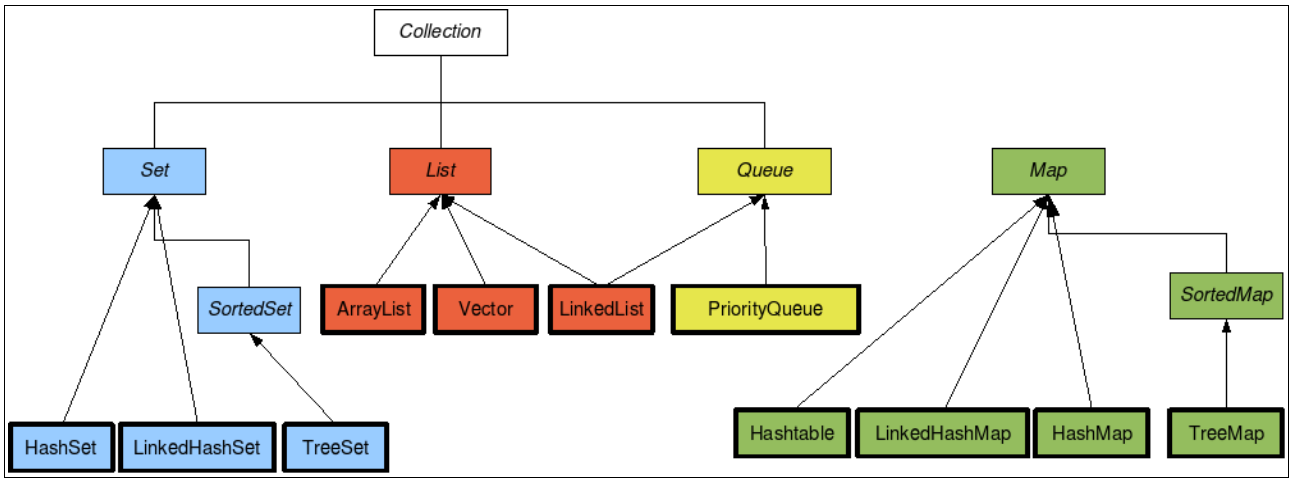
Em síntese, ao utilizarmos as coleções já implementadas no pacote java.util, obtemos os seguintes benefícios:

- ✔ Redução na codificação: Muitas funcionalidades esperadas durante a manipulação de um conjunto de elementos já estão prontas;
- ✔ Melhor desempenho do programa: As implementações das funcionalidades e estruturas de armazenamento foram feitas com o objetivo de fornecerem um desempenho ótimo;
- ✔ Maior qualidade do programa: Respeitando as interfaces podemos

substituir as implementações utilizadas sem prejuízo ao resto do programa;

Existem outros benefícios porém, para o iniciante – nosso caso – o entendimento destes é mais do que suficiente.

Tudo bem, entendi, é importante reutilizar código, tempo é um recurso escasso, as implementações fornecidas são ótimas... Mas o que você quis dizer quando falou da "arquitetura unificada"? A resposta encontra-se na próxima figura:



Nesta figura temos as interfaces (raízes e nós das árvores) que formam o conjunto de interfaces disponíveis. Nas folhas (retângulos com bordas mais espessas) encontramos as classes concretas ou implementações destas interfaces.

Esta arquitetura de interfaces forma o núcleo do Java Framework Collections e todas as classes concretas irão derivar de uma ou mais interfaces.

É interessante notar que apesar da "arquitetura única" as coleções derivadas da interface Map não implementam a interface Collection no entanto elas também fazem parte do pacote de implementações Java para coleções.

A figura apresentada pode ser interpretada da seguinte forma: "Um Set, List ou Queue é um tipo de Collection, cada um com suas particularidades. Já um Map não é do mesmo tipo dos demais mas também manipula coleções de elementos"

As implementações de cada ramo estão divididas da seguinte forma:

- **Set**: HashSet, LinkedHashSet e TreeSet;
- **List**: ArrayList, Vector e LinkedList;
- **Queue**: LinkedList e PriorityQueue;

- **Map**: Hashtable, LinkedHashMap, HashMap e TreeMap.

Muito interessante tudo isto mas, na prática, quando devo utilizar uma ou outra implementação? Qual a diferença entra cada uma destas interfaces?

A seguir serão explicados, simplificadamente, os principais benefícios obtidos com o uso de cada grupo de interfaces e respectivas implementações. A compreensão das funcionalidades de cada uma das interfaces e implementações representa a parte mais importante deste módulo.

Então vamos lá:

- **Set**: Representa uma coleção que não pode conter duplicatas, implementa uma abstração dos conjuntos matemáticos;
- **List**: Representa uma coleção ordenada (ordem de inserção) e que permite duplicatas;
- **Queue**: Parecida com a interface List porém adiciona algumas funcionalidades, é muito utilizada para representar listas de elementos cuja ordem é importante, normalmente elementos em ordem de processamento;
- **Map**: É um objeto que armazena um elemento e o remove através da sua chave, os Maps não aceitam chaves duplicadas.

Nós iniciaremos esta jornada investigando a interface Set porém antes de começarmos precisamos conhecer um pouco dos métodos equals() e hashcode() da classe Object e suas funções.

equals()

Quando nós estamos trabalhando com objetos os operadores de igualdade, == e != , podem ter efeitos desastrosos se não forem corretamente aplicados. Quando estes operadores são utilizados, na verdade, o que está sendo comparado é a referência do objeto, isto é, esta operação irá retornar verdadeiro (true) apenas se as duas referências são do mesmo objeto.

Um exemplo desta situação é o seguinte:

```
System.out.println(new String("Pedro") == new String("Pedro"));//Imprime false
```

No exemplo anterior, caso o código seja executado, obteremos como saída

"false". Isto acontece, pois conforme explicado anteriormente, estamos comparando as referências e não o conteúdo das Strings (que são idênticos). Ou seja cada uma das partes da operação refere-se a um objeto diferente porém, do mesmo tipo String e com o mesmo valor "Pedro".

Para resolver este problema basta substituir o código acima por:

```
System.out.println(new String("Pedro").equals(new String("Pedro")));//Imprime true
```

Neste exemplo o resultado obtido será true, isto acontece pois agora o que está sendo comparado é o conteúdo de cada objeto String, ou seja o valor "Pedro".

O importante é que você compreenda que ao utilizar os operadores de igualdade, sobre objetos, (== ou !=) o que você está verificando é se as duas referências fazem alusão ao mesmo objeto. Ao utilizar o método equals() nós estamos verificando se os dois objetos são significativamente iguais (equivalentes).

A maioria das coisas (objetos) que importam, no contexto da computação, possuem algum identificador único ou alguma forma de identificação única. Ex: Carro – placa/renavam, Pessoa – CPF, Empresa – CNPJ, Fatura – código, Nota fiscal – número/série, Município – Código do IBGE, Produto – código de barras e etc.

Sendo assim ao sobreescrevermos o método equals() devemos observar estes identificadores e implementar o método a fim de realizar os testes de igualdade sobre este(s) atributo(s).

A fim de clarificar o que foi dito acompanhe o exercício a seguir.

Exercício resolvido

Crie uma classe que represente notas fiscais com código, valor e data, e sobreescreva o método equals de forma que duas notas fiscais sejam consideradas iguais apenas se seus códigos forem idênticos.

Resolução:

1. Criar classe NotaFiscal com atributos código, valor e data;
2. Sobreescrever método equals de forma que compare os códigos.

```
import java.util.Calendar;
import java.util.Date;
```

```
public class NotaFiscal {

	private Integer codigo;
	private Date dataEmissao;
	private Double valor;

	//Construtor parametrizado
	public NotaFiscal(Integer codigo, Date dataEmissao, Double valor) {
		super();
		this.codigo = codigo; //atribui o valor do codigo
		this.dataEmissao = dataEmissao; //atribui a data da emissao
		this.valor = valor;//atribui o valor
	}

	//metodo equals
	public boolean equals(Object obj) {
		//Verifica se o objeto é da classe NotaFiscal e verifica se os codigos são iguais
		//Se a resposta for afirmativa para os dois testes retorna verdadeiro (true)
		if((obj instanceof NotaFiscal) &&
			((NotaFiscal)obj).getCodigo().equals(this.getCodigo()))
		{
			return true;
		}else
		{
			return false;
		}
	}

	//retorna o codigo da nota fiscal
	public Integer getCodigo() {
		return codigo;
	}

	public static void main(String[] args) {
		//Criacao da nota fiscal 1
		NotaFiscal nf1= new NotaFiscal(1, Calendar.getInstance().getTime() ,123456.0 );
		//Criacao da nota fiscal 2
		NotaFiscal nf2= new NotaFiscal(2, Calendar.getInstance().getTime() ,555566.0 );
		//Criacao da nota fiscal 3
		NotaFiscal nf3= new NotaFiscal(1, Calendar.getInstance().getTime() ,788955.0 );

		System.out.println("nf1 igual a nf2: "+nf1.equals(nf2));//resulta false
		System.out.println("nf1 igual a nf3: "+nf1.equals(nf3));//resulta verdadeiro
	}
}
```

Resultado da execução desta classe:

```
nf1 igual a nf2: false
nf1 igual a nf3: true
```

hashcode()

Uma vez compreendida a importância do método equals() iniciamos os estudos para descobrir a função do método hashcode(). Códigos de Hash ou Hashcodes são utilizados para aumentar a performance quando manipulamos grandes quantidades de objetos.

Hashcodes* são utilizados para identificar objetos de forma que o acesso a estes se torne mais eficiente. Estes códigos são gerados por uma função de

* Para saber mais visite http://pt.wikipedia.org/wiki/Tabela_de_hashing

espalhamento, o resultado desta função é obtido de acordo com o valor (dado) de entrada.

É importante compreender que entradas diferentes podem produzir tanto hashcodes iguais quanto diferentes, já entradas iguais produzirão sempre o mesmo hashcode, desta forma o método hashCode() deve respeitar o método equals(), ou seja, sempre que dois objetos forem iguais, de acordo com o método equals(), obrigatoriamente o resultado obtido pelo método hashCode() deve ser igual para estes objetos.

Caso o método hashCode() não respeite a implementação do método equals() poderemos ter problemas de desempenho e consistência nas nossas coleções.

O ideal é que a função de espalhamento nunca produza o mesmo código hash para entradas diferentes porém isto nem sempre é possível, nestes casos devemos nos preocupar em criar funções que façam uma boa dispersão de códigos.

Mais a frente iremos utilizar estas funções (hashcode() e equals()) na prática.

Algumas definições

- **Ordenação**: Define que a coleção possui algum tipo de ordem determinada por regra(s).
  Ex: Crescente, decrescente, prioridade e etc.

- **Organização**: Garante que a coleção sempre é percorrida em uma determinada seqüência não aleatória.
  Ex: Ordem de inserção, índice e etc.

Durante a explicação das coleções estas definições ficarão mais claras.

SET

Um Set, ou seja, qualquer implementação desta interface, obrigatoriamente, não aceita duplicatas em seu conjunto de elementos. Estas duplicatas são identificadas através do método equals(), logo, qualquer implementação desta interface (Set) é altamente dependente de uma correta

implementação do método equals().

A implementação do método equals() em desacordo com o universo a ser modelado pode fazer com que elementos diferentes sejam considerados iguais ou elementos iguais sejam tratados como diferentes.

Alguns dos métodos mais utilizados quando manipulamos Sets, estes métodos são comuns a qualquer implementação desta interface:

Método	Descrição
add(Objeto)	Adiciona um elemento ao nosso Set, retorna true se o elemento foi inserido
remove(Objeto)	Remove o elemento da nossa coleção, retorna true se o elemento foi removido
iterator()	Retorna um objeto do tipo Iterator
size()	Retorna um int (inteiro) com a quantidade de elementos da coleção
contains(Objeto)	Retorna true se o elemento já existe dentro do Set
clear()	Elimina todos os elementos do Set

HashSet

Uma das implementações concretas da interface Set. O HashSet caracteriza-se por não aceitar duplicatas, característica derivada do Set, ser uma coleção desordenada e desorganizada, isto é, não há nenhuma garantia quanto a ordem que os elementos serão percorridos.

O HashSet utiliza o hashcode como chave de identificação de cada um dos seus elementos então, quanto mais eficiente for a função de espalhamento mais eficiente será a sua pesquisa e no entanto, se o método hashCode() não respeitar a implementação do método equals() poderemos ter elementos duplicatas no nosso Set.

A seguir será apresentada a utilização do HashSet com uma implementação eficiente comparada a uma ineficiente.

Exercício resolvido

Utilizando a classe NotaFiscal, desenvolvida no E.R. - 1, sobreescreva o método hashcode() duas vezes, uma utilizando o código e outra retornando sempre o mesmo valor (pior função de espalhamento possível – afinal de contas não espalha

nada). Crie um HashSet e armazene 10.000 notas fiscais com códigos variando de 1 a 10.000, retire o elemento com código 5.000. Compare os tempos de cada remoção.

Resolução:

1. Sobreescrever o método hashcode, utilizar o código como função de espalhamento;
2. Criar um HashSet com 100.000 elementos e atribuir os códigos de acordo com a ordem de criação;
3. Remover o elemento de código 5.000 verificar o tempo total;
4. Sobreescrever o método hashcode, utilizar uma função de espalhamento que retorne sempre o mesmo valor;
5. Repetir passos 2 e 3.

```
import java.util.Calendar;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

public class NotaFiscalHashSet {

	private Integer codigo;
	private Date dataEmissao;
	private Double valor;

	//Construtor parametrizado
	public NotaFiscalHashSet(Integer codigo, Date dataEmissao, Double valor) {
		super();
		this.codigo = codigo; //atribui o valor do codigo
		this.dataEmissao = dataEmissao; //atribui a data da emissao
		this.valor = valor;//atribui o valor
	}

	//metodo equals
	public boolean equals(Object obj) {
		//Verifica se o objeto Ã© da classe NotaFiscal e verifica se os codigos sÃ£o iguais
		//Se a resposta for afirmativa para os dois testes retorna verdadeiro (true)
		if((obj instanceof NotaFiscalHashSet) &&
			((NotaFiscalHashSet)obj).getCodigo().equals(this.getCodigo()))
		{
			return true;//
		}else
		{
			return false;
		}
	}

	//retorna o codigo da nota fiscal
	public Integer getCodigo() {
		return codigo;
	}

	//metodo hashcode
	public int hashCode() {
		return codigo;//funcao de espalhamento retorna o valor do codigo
		//return 2;//funcao de espalhamento retorna sempre o mesmo valor
	}
	//Retorna um long que identifica o tempo atual
	public static long getTempoAtual()
	{
		return Calendar.getInstance().getTimeInMillis();
```

```
	}

	public static void main(String[] args) {
		Set hashset = new HashSet();//Criacao do HashSet
		Long inicio = getTempoAtual(); //Inicio da contagem do tempo
		for(int i = 0; i < 10000; i++)//Iteracao sobre os elementos
		{
			//adicao dos elementos
			hashset.add(new NotaFiscalHashSet(i, Calendar.getInstance().getTime(), 0.0));
		}
		//remocao dos elementos
		hashset.remove(new NotaFiscalHashSet(5000, Calendar.getInstance().getTime(), 0.0));
		Long fim = getTempoAtual();//fim da contagem do tempo
		System.out.println("Tempo total: "+(fim-inicio)+" ms");//impressao do tempo total
	}
}
```

Resultado obtido utilizando o hashcode baseado no código:

```
Tempo total : 191 ms
```

Resultado obtido utilizando o hashcode fixo (2):

```
Tempo total: 11279 ms
```

A partir dos resultados temos que, nesta situação, utilizando uma função de espalhamento mais adeqüada o tempo total de pesquisa foi aproximadamente 60 vezes menor do que utilizando a pior função de hashcode possível, espalhando para apenas um código.

LinkedHashSet

É uma versão organizada do HashSet, lembrando, organizada significa que existe algum tipo de seqüência não-aleatória durante a iteração dos elementos, neste caso a ordem de inserção é respeitada. Por ser um Set, o LinkedHashSet não aceita duplicatas e, da mesma forma que o HashSet, é dependente da correta implementação do método equals(). Devemos utilizar o LinkedHashSet ao invés do HashSet quando a ordem de iteração dos elementos é importante.

Já que falamos de iteração vamos apresentar duas formas de iteração de elementos de coleções: utilizando um Iterator; e utilizando o for-each ou enhanced-for.

- **Iterator**: É uma interface que fornece poucos métodos (next(), hasNext() e remove()) porém de muita utilidade quando precisamos percorrer coleções. Todas as implementações do Framework Collections fornecem um Iterator. Os

dois principais métodos serão apresentados a seguir.

Método	Descrição
hasNext()	Retorna true se existe próximo elemento.
next()	Retorna o próximo elemento na iteração.

- **For-Each**: É um for adaptado a coleções e que percorre todos os elementos qualquer de coleção do Framework Collection. A seguir temos um exemplo:

```
Set carrosHashSet = new HashSet();
for(Carros carro : carrosHashSet)
{
	carro.setPlaca(...);
	...
}
```

A estrutura deste for-each poderia ser lida da seguinte forma : "Para cada elemento do tipo Carro existente na minha coleção de Carro (carrosHashSet) faça isto";

Nós próximos exercícios as duas modalidades de iteração serão utilizadas e a observação do uso prático facilitará o entendimento.

Exercício resolvido

Implemente um método que manipule uma coleção HashSet e outra coleção LinkedHashSet, adicione a elas 20 números inteiros (Integer), em ordem crescente, e percorra todos os elementos, observe se a ordem de iteração é a mesma.

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;

public class IntegerLinkedHashSet {

	public static void main(String[] args) {
		Set hashSet = new HashSet();//Criacao do HashSet
		Set linkedHashSet = new LinkedHashSet(); //Criacao do LinkedHashSet

		for(int i = 0; i < 20; i++) //For para preencher as duas colecoes
		{
			hashSet.add(i); //Adicionando elemento i ao hashSet
			linkedHashSet.add(i); //Adicionando elemento i ao linkedHashSet
		}

		Iterator itHashSet = hashSet.iterator(); //obtendo Iterator para HashSet
		//obtendo Iterator para LinkedHashSet
		Iterator itLinkedHashSet = linkedHashSet.iterator();
```

```
                //Enquanto existir proximo elemento nos Iterators
                while(itHashSet.hasNext() && itLinkedHashSet.hasNext())
                {
                        //Imprime (system.out.println) o proximo elemento de cada Iterator
                        System.out.println("HashSet: "+itHashSet.next()+ " LinkedHashSet: "
                                        +itLinkedHashSet.next());
                }
        }
}
```

Resultado encontrado:

```
HashSet: 0 LinkedHashSet: 0
HashSet: 1 LinkedHashSet: 1
HashSet: 2 LinkedHashSet: 2
HashSet: 3 LinkedHashSet: 3
HashSet: 4 LinkedHashSet: 4
HashSet: 5 LinkedHashSet: 5
HashSet: 6 LinkedHashSet: 6
HashSet: 7 LinkedHashSet: 7
HashSet: 8 LinkedHashSet: 8
HashSet: 9 LinkedHashSet: 9
HashSet: 10 LinkedHashSet: 10
HashSet: 11 LinkedHashSet: 11
HashSet: 12 LinkedHashSet: 12
HashSet: 13 LinkedHashSet: 13
HashSet: 14 LinkedHashSet: 14
HashSet: 15 LinkedHashSet: 15
HashSet: 17 LinkedHashSet: 16
HashSet: 16 LinkedHashSet: 17
HashSet: 19 LinkedHashSet: 18
HashSet: 18 LinkedHashSet: 19
```

Perceba que até o elemento 15 o HashSet vinha mantendo a ordem de inserção durante o processo de iteração dos seus elementos, mas como dito anteriormente, esta situação não é garantida e, a partir do elemento seguinte, a ordem foi desrepeitada (linhas em negrito) enquanto isto, no LinkedHashSet, a ordem de inserção foi respeitada como era esperado.

TreeSet

É um Set ordenado e como tal não aceita duplicatas, ou seja existe alguma regra que garante a ordenação, no Treeset os elementos inseridos serão percorridos de acordo com sua ordem natural e de forma ascendente.

Exercício resolvido

Implemente um método que manipule as três modalidades de Set (HashSet, LinkedHashSet e TreeSet), e insira o mesmo conjunto de inteiros em cada uma delas. Percorra todos os elementos de cada uma delas e compare a ordem

encontrada.

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class IntegerLinkedHashSet {

	public static void main(String[] args) {
		Set hashSet = new HashSet();//Criacao do HashSet

		hashSet.add(100);//Adicionando valores aleatoriamente
		hashSet.add(0);
		hashSet.add(23);
		hashSet.add(465);
		hashSet.add(45);
		hashSet.add(21);
		hashSet.add(10);
		hashSet.add(3);
		hashSet.add(12);
		hashSet.add(5);
		hashSet.add(147);
		hashSet.add(29);
		hashSet.add(7);
		hashSet.add(3);
		hashSet.add(12);
		hashSet.add(54);
		//Criacao do LinkedHashSet a partir do hashSet
		Set linkedHashSet = new LinkedHashSet(hashSet);
		Set treeSet = new TreeSet(hashSet);//Criacao do TreeSet a partir do hashSet

		Iterator itHashSet = hashSet.iterator(); //obtendo Iterator para HashSet
		//obtendo Iterator para LinkedHashSet
		Iterator itLinkedHashSet = linkedHashSet.iterator();
		Iterator itTreeSet = treeSet.iterator(); //obtendo Iterator para TreeSet

		//Enquanto existir proximo elemento nos Iterators
		while(itHashSet.hasNext() && itLinkedHashSet.hasNext() && itTreeSet.hasNext())
		{
			//Imprime (system.out.println) o proximo elemento de cada Iterator
			System.out.println("HashSet: "+itHashSet.next()+
					" LinkedHashSet: "+itLinkedHashSet.next()+
					" TreeSet: "+itTreeSet.next());
		}
	}
}
```

Resultado encontrado:

```
HashSet: 0 LinkedHashSet: 0 TreeSet: 0
HashSet: 100 LinkedHashSet: 100 TreeSet: 3
HashSet: 3 LinkedHashSet: 3 TreeSet: 5
HashSet: 5 LinkedHashSet: 5 TreeSet: 7
HashSet: 7 LinkedHashSet: 7 TreeSet: 10
HashSet: 10 LinkedHashSet: 10 TreeSet: 12
HashSet: 12 LinkedHashSet: 12 TreeSet: 21
HashSet: 465 LinkedHashSet: 465 TreeSet: 23
HashSet: 45 LinkedHashSet: 45 TreeSet: 29
HashSet: 21 LinkedHashSet: 21 TreeSet: 45
HashSet: 54 LinkedHashSet: 54 TreeSet: 54
HashSet: 23 LinkedHashSet: 23 TreeSet: 100
HashSet: 147 LinkedHashSet: 147 TreeSet: 147
HashSet: 29 LinkedHashSet: 29 TreeSet: 465
```

Antes de nos avaliarmos a ordem dos resultados, observe que durante a

fase de inclusao dos numeros no HashSet alguns deles se repetem, especificamente o 3 e o 12 (linhas em negrito), porém no resultado estes números aparecem apenas uma vez cada um, isto acontece pois, conforme dito anteriormente, qualquer coleção do tipo Set não permite duplicatas, na verdade, na segunda tentativa de inclusão dos números 3 e 12, eles não foram incluídos.

Em relação a ordem o HashSet, desta vez, manteve a ordem de inserção e obtivemos um resultado igual ao visto no LinkedHashSet, no entanto não temos garantia nenhuma de que este comportamento irá se repetir. Já o TreeSet não respeitou a ordem de inserção mas, ordenou de acordo com a ordem natural de cada um dos valores.

Interessante! Para números a ordem natural é fácil de compreender – afinal de contas 1 é menor do que 2 que é menor do que 3 que é menor... – no entanto, nem sempre trabalhamos com números, muitas vezes criamos nossos próprios tipos (Classes). Nestes casos devemos informar que a nossa Classe (tipo) pode ser comparada. Isto é feito utilizando-se a interface Comparable, este assunto será abordado no mini-curso relativo a Coleções e disponível junto com este material.

LIST

A interface List garante que todas as suas implementações serão organizadas, isto é, a ordem de inserção será mantida. Em implementações da interface List podemos são permitidas duplicatas, isto é, objetos iguais de acordo com o método **equals()**.

Ao contrário do Set, qualquer implementação da interface List mantém seus elementos indexados, ou seja, existe uma preocupação com o posicionamento de cada elemento e esta posição é determinada pelo índice. Devemos utilizar um List quando a ordem de inserção ou a posição na coleção nos interessa.

Alguns métodos importantes quando manipulamos Lists:

Método	Descrição
add(Objeto)	Adiciona um elemento ao List, na última posição
get(índice)	Retorna o elemento da posição do índice
iterator()	Retorna um objeto do tipo Iterator
size()	Retorna um int (inteiro) com a quantidade de elementos da coleção

contains(Objeto)	Retorna true se o elemento já existe dentro do List
clear()	Elimina todos os elementos do Set

ArrayList e Vector:

Um ArrayList pode ser visto como um array (vetor) porém dinâmico, isto é, ele aumenta o espaço disponível a medida que é demandado. Ele é organizado pelo índice, ou seja, temos alguma garantia quanto a ordem que encontraremos os elementos.

Vector é basicamente um ArrayList, no entanto seus métodos são sincronizados o que significa que o acesso por vários processos simultaneamente é coordenado.

Antes de continuarmos falando das implementações da interface List, iremos introduzir uma nova característica da linguagem Java e mostrar trechos de códigos que a utilizam junto com ArrayList e Vector:

- **Generics**: É uma funcionalidade introduzida a partir da versão 1.5 do Java. Em relação as coleções sua principal característica é definir o tipo aceito pela coleção;

Até agora as nossas coleções aceitavam qualquer tipo de elemento, desde que fosse um Object, por exemplo, abaixo temos um ArrayList onde adicionamos um String e um Integer:

```
//Criando ArrayList
List arrayList = new ArrayList();
//Adicionando um String
arrayList.add(new String("Texto"));
//Adicionando um Integer
arrayList.add(new Integer(3));
```

Isto era possível pois nossas coleções aceitam qualquer elemento que seja do tipo Object, e todas as classes criadas em Java automaticamente herdam, e portanto são, da classe Object. No entanto, com o uso de Generics podemos definir de que tipo específico será a nossa coleção. Abaixo temos um exemplo de ArrayList, parecido com o anterior, utilizando Generics:

```
//Criando ArrayList de Integer
List<Integer> arrayList = new ArrayList<Integer>();
//Adicionando Integer
arrayList.add(new Integer(3));
```

```
arrayList.add(new Integer(5));
```

Perceba que a partir de agora este List aceita apenas objetos do tipo Integer utilizando esta notação, declaração <Integer> junto a declaração da Classe/Interface, evito que sejam inseridos indevidamente outros tipos que não sejam Integer na minha coleção.

LinkedList:

Muito similar as duas coleções vistas anteriormente (Vector e ArrayList), a diferença é que todos os elementos são ligados entre si. O desempenho do LinkedList é muito superior aos do ArrayList e Vector quando necessitamos inserir elementos no início da coleção, no entanto se precisarmos obter algum elemento pelo o índice o desempenho é muito inferior.

Exercício resolvido

Implemente um código que compare o tempo de obtenção e inserção, na primeira posição dos Lists, de 25.000 números inteiros para o ArrayList, Vector e LinkedList.

```
import java.util.LinkedList;
import java.util.List;
import java.util.ArrayList;
import java.util.Vector;

public class ArrayListVector {

	public static void main(String[] args)
	{
		Integer quantidadeElementos = 25000;//Quantidade total de elementos
		List<Integer> arrayList = new ArrayList<Integer>();//Criacao do ArrayList de Integer
		for(int i = 0; i < quantidadeElementos; i++)//Inserir elementos dentro do ArrayList
		{
			arrayList.add(i);
		}
		//Criacao do LinkedList de Integer, com os mesmos elementos do ArrayList
		LinkedList<Integer> linkedList = new LinkedList<Integer>(arrayList);
		//Criacao do Vector de Integer, com os mesmos elementos do ArrayList
		List<Integer> vector = new Vector<Integer>(arrayList);

		//inicio da contagem do tempo
		long inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do ArrayList
		for(int i = 0; i < arrayList.size(); i++)
		{
			arrayList.get(i);//Obtendo elementos do ArrayList
		}
		//fim da contagem do tempo
		long fim = System.currentTimeMillis();
		System.out.println("Obter elemento. Tempo arrayList: "+(fim-inicio)+" ms");

		//inicio da contagem do tempo
```

```
		inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do Vector
		for(int i = 0; i < vector.size(); i++)
		{
			vector.get(i);//Obtendo elementos do Vector
		}
		//fim da contagem do tempo
		fim = System.currentTimeMillis();
		System.out.println("Obter elemento. Tempo Vector: "+(fim-inicio)+" ms");

		//inicio da contagem do tempo
		inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do LinkedList
		for(int i = 0; i < linkedList.size(); i++)
		{
			linkedList.get(i);//Obtendo elementos do LinkedList
		}
		//fim do cronomentro
		fim = System.currentTimeMillis();
		System.out.println("Obter elemento. Tempo linkedList: "+(fim-inicio)+" ms");

		//inicio da contagem do tempo
		inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do ArrayList
		for(int i = 0; i < quantidadeElementos; i++)
		{
			arrayList.add(0, i);//Adicionando um elemento no inicio do ArrayList
		}
		//fim da contagem do tempo
		fim = System.currentTimeMillis();
		System.out.println("Inserir elemento no inicio. Tempo arrayList: "+(fim-inicio)
			+" ms");

		//inicio da contagem do tempo
		inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do Vector
		for(int i = 0; i < quantidadeElementos; i++)
		{
			vector.add(0, i);//Adicionando um elemento no inicio do Vector
		}
		//fim da contagem do tempo
		fim = System.currentTimeMillis();
		System.out.println("Inserir elemento no inicio. Tempo Vector: "+(fim-inicio)+" ms");

		//inicio da contagem do tempo
		inicio = System.currentTimeMillis();
		//percorrendo todos os elementos do LinkedList
		for(int i = 0; i < quantidadeElementos; i++)
		{
			linkedList.addFirst(i);//Adicionando elemento no inicio do LinkedList
		}
		//fim da contagem do tempo
		fim = System.currentTimeMillis();
		System.out.println("Inserir elemento no inicio. Tempo linkedList: "+(fim-inicio)
			+" ms");
	}

}
```

Resultado obtido:

```
Obter elemento. Tempo arrayList: 12 ms
Obter elemento. Tempo Vector: 12 ms
Obter elemento. Tempo linkedList: 1561 ms
Inserir elemento no inicio. Tempo arrayList: 1804 ms
Inserir elemento no inicio. Tempo Vector: 1809 ms
Inserir elemento no inicio. Tempo linkedList: 17 ms
```

Considerando que esta é uma experiência apenas ilustrativa (não controlada), os resultados mostram que o ArrayList e o Vector são equivalente nos

quesitos avaliados. Já o LinkedList foi muito inferior quando da obtenção dos elementos e muito superior na inserção dos elementos na primeira posição.

MAP

Um Map, qualquer implementação desta interface, identifica seus elementos através de uma chave, logo, esta coleção aceita duplicatas de valores (elementos) desde que eles possuam chaves diferentes. A principal característica dos Maps são suas chaves que nunca podem se repetir, ou seja, em um Map nunca irão haver dois elementos com a mesma chave. Assim como os elementos armazenados as chaves utilizadas nos Maps também devem ser objetos.

Os principais métodos utilizados quanto manipulamos Maps são:

Método	Descrição
put(chave, objeto)	Adiciona a chave e o elemento
get(chave)	Retorna o elemento de acordo com a chave
iterator()	Retorna um objeto do tipo Iterator
size()	Retorna um int (inteiro) com a quantidade de elementos da coleção
containsKey(chave)	Retorna true se a chave já existe
clear()	Elimina todos os elementos do Set
containsValue(objeto)	Retorna true se o objeto já existir no Map

HashMap e Hashtable

HashMap é um Map desorganizado, isto é, a ordem de iteração dos elementos é desconhecida, e desordenado, ou seja, os elementos não sofrem nenhum tipo de ordenação derivada de alguma regra.

Hashtable, similarmente ao ArrayList e Vector, é a versão sincronizada do HashMap. A maior diferença é que o HashMap aceita nulos tanto para chaves quanto para valores enquanto o Hashtable não aceita nenhum nulo.

A seguir segue trecho de código que exemplifica o uso do HashMap:

```
//Criacao do HashMap
Map notasFiscaisMap = new HashMap();

//Criacao das Notas Fiscais
NotaFiscal nf1 = new NotaFiscal(1, Calendar.getInstance().getTime(), 123.56);
```

```
NotaFiscal nf2 = new NotaFiscal(2, Calendar.getInstance().getTime(), 127.00);
NotaFiscal nf3 = new NotaFiscal(3, Calendar.getInstance().getTime(), 100.25);

//Armazenamento das notas fiscais no HashMap
notasFiscaisMap.put(nf1.getCodigo(), nf1);
notasFiscaisMap.put(nf2.getCodigo(), nf2);
notasFiscaisMap.put(nf3.getCodigo(), nf3);

//Obtem nf2
NotaFiscal nf = (NotaFiscal)notasFiscaisMap.get(2);
```

LinkedHashMap

Muito similar ao LinkedHashSet, porém esta é a versão que implementa a interface Map, logo ao armazenar os objetos é necessária uma chave. E, ao contrário do HashMap, o LinkedHashMap é organizado, o que significa dizer que durante a iteração dos elementos ele respeita a ordem que estes foram inseridos na coleção.

Se a ordem de inserção for importante e precisarmos identificar os elementos pelas suas chaves devemos utilizar esta implementação da interface Map.

Abaixo temos o mesmo código exibido anteriormente porém a implementação utilizada agora é a LinkedHashMap:

```
//Criacao do LinkedHashMap
Map notasFiscaisMap = new LinkedHashMap();

//Criacao das Notas Fiscais
NotaFiscal nf1 = new NotaFiscal(1, Calendar.getInstance().getTime(), 123.56);
NotaFiscal nf2 = new NotaFiscal(2, Calendar.getInstance().getTime(), 127.00);
NotaFiscal nf3 = new NotaFiscal(3, Calendar.getInstance().getTime(), 100.25);

//Armazenamento das notas fiscais no HashMap
notasFiscaisMap.put(nf1.getCodigo(), nf1);
notasFiscaisMap.put(nf2.getCodigo(), nf2);
notasFiscaisMap.put(nf3.getCodigo(), nf3);

//Obtem nf2
NotaFiscal nf = (NotaFiscal)notasFiscaisMap.get(2);
```

TreeMap

Assim como o TreeSet é um Set que ordena os elementos de acordo com alguma regra, o TreeMap também ordena seus elementos através da chave por alguma regra. Quando esta ordem não é definida pela interface Comparable ou por um objeto Comparator1 o TreeMap busca a ordem natural dos elementos.

A seguir é apresentado um trecho de código mostrando o uso do TreeMap:

```
//Criacao do TreeMap utilizando Generics
Map<Integer, String> treeMap = new TreeMap<Integer, String>();

//Inserindo os valores
treeMap.put(5, "Numero 5");
treeMap.put(1, "Numero 1");
```

```
treeMap.put(100, "Numero 100");
treeMap.put(22, "Numero 22");

//For-Each, percorrendo todos os elementos do Map
for(String str : treeMap.values())
{
	//Imprimindo os valores encontrados
	System.out.println(str);
}
```

Resultado da execução deste código:

```
Numero 1
Numero 5
Numero 22
Numero 100
```

Observe que a declaração **Map<Integer, String>** utiliza generics. Ao declarar um Map desta forma estou garantindo que todas as chaves serão do tipo Integer e todos os valores serão do tipo String.

QUEUE

A última das interfaces a ser analisada, a interface Queue é projetada para armazenar listas de elementos a serem executados/processados. Os elementos inseridos em uma coleção que implemente esta interface, normalmente, possuirá comportamento similar a uma Fila, isto é, o primeiro elemento a ser inserido será o primeiro elemento a ser processado.

Os métodos comumente utilizados com esta coleção são os seguintes:

Método	Descrição
add(objeto)	Adiciona o elemento, joga uma exceção se não for possível incluir
offer(objeto)	Idêntico ao add, porém não joga exceção em caso de falha
remove()	Remove o elemento, joga uma exceção se não houver elementos
poll()	Idêntico ao remove(), porém retorna nulo caso não hajam elementos
element()	Retorna elemento elemento inicial, joga exceçao caso não haja
peek()	idêntico ao element(), porém retorna nulo caso não haja elemento

PriorityQueue

Diferentemente de uma fila convencional, onde o primeiro elemento a ser inserido é o primeiro a ser removido, na PriorityQueue nem sempre a ordem de inserção representa a ordem de saída dos elementos. Esta ordem é administrada pela coleção que priorizará a ordem de saída dos elementos.

Esta ordem é definida pelo próprio objeto quando este implementa a interface Comparable, por um objeto Comparator ou pela ordem natural dos elementos.

A seguir mostraremos trecho de código que exemplifica o uso da PriorityQueue:

```
//Criando a minha Queue utilizando Generics
Queue<Integer> filaPrioridade = new PriorityQueue<Integer>();
//Adicionando elementos
filaPrioridade.add(5);
filaPrioridade.add(5);
filaPrioridade.add(2);
filaPrioridade.add(3);
filaPrioridade.add(0);
filaPrioridade.add(-2);

//Removendo elementos e imprimindo
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
System.out.println("poll: "+filaPrioridade.poll());
```

Resultado obtido:

```
poll: -2
poll: 0
poll: 2
poll: 3
poll: 5
poll: 5
poll: null
```

No exemplo apresentado, a ordem de prioridade foi definida naturalmente uma vez que tratavam-se de números inteiros. A ordem natural é, neste caso, do menor para o maior. Ao fim da execução o resultado "null" significa que não existem mais elementos na fila pois todos já foram removidos.

A classe Collections

Da mesma forma que para arrays (vetores), Java fornece uma classe com um conjunto de funcionalidades muito utilizadas (ordenação, pesquisa, preenchimento e etc.) na manipulação das estruturas de dados do Framework Collections também existe uma classe similar (Collections)

A classe Collections (mesmo nome do framework) é para as coleções do framework a mesma coisa que a classe Arrays é para os vetores (arrays) convencionais, portanto, quando você precisar destes tipos de funcionalidades, irá apenas adaptar às suas necessidades, se for necessário.

A seguir iremos explorar brevemente as principais funcionalidades.

Ordenação

O algoritmo de ordenação implementado na classe Collections ordena os elementos em ordem ascendente. Temos um exemplo de código a seguir:

```
//Criação do ArrayList
List arrayList = new ArrayList();
//Adicionando elementos
arrayList.add("Miguel");
arrayList.add("Albert");
arrayList.add("Fernando");
arrayList.add("Mario");

//Ordenando
Collections.sort(arrayList);
//Imprimindo o resultado
System.out.println(arrayList);
```

Resultado obtido:

```
[Albert, Fernando, Mario, Miguel]
```

Mistura/Desordenação

Este algoritmo é o oposto ao anterior, ao invés de ordenar ele desordena (mistura) os elementos dentro de um List. A primeira vista pode parecer pouco útil, mas existem situações onde você irá querer desordenar sua estrutura de dados para obter um elemento aleatoriamente.

Por exemplo, abaixo temos uma possível implementação para a modalidade de sorteio cujos nomes dos elementos a serem sorteados, normalmente pessoas, são escritos em pequenos pedaços de papéis e o primeiro a ser escolhido é o vencedor.

```
//Criação do ArrayList
List arrayList = new ArrayList();
//Adicionando elementos
arrayList.add("Miguel");
arrayList.add("Albert");
arrayList.add("Fernando");
arrayList.add("Mario");

//Misturando elementos
Collections.shuffle(arrayList);
//Sorteando o primeiro nome e imprimindo o resultado
System.out.println(arrayList.get(0));
```

Resultado obtido em uma execução

```
Fernando
```

Perceba que o elemento na posição 0 (zero), pela ordem de inserção seria "Miguel" e, no entanto, obtivemos o elemento "Fernando", o que confirma, a mudança de posição destes elementos dentro do List.

Pesquisa:

Quando falamos em estruturas de dados obrigatoriamente devemos pensar em algoritmos de ordenação e pesquisa, pois, de que adianta uma estrutura de dados cujos elementos não conseguimos localizar? Ao manipularmos estruturas de dados sempre devemos ter em mente que iremos ordená-las ou pesquisá-las.

Por isto a importância deste algoritmo. A pesquisa efetuada pela Classe Collections retorna um número inteiro positivo ou zero se o elemento for encontrado e negativo se o elemento não existe na coleção. Quando o elemento existe na coleção o número representa o seu índice (posição), por outro lado, quando o elemento não existe, o número, em módulo, representa a posição que ele deveria estar (ponto de inserção), neste caso, esta posição somente é válida se a coleção estiver ordenada antes da pesquisa.

A seguir temos um exemplo:

```
//Criação do ArrayList
List arrayList = new ArrayList();
//Adicionando elementos
arrayList.add("Miguel");
arrayList.add("Albert");
arrayList.add("Fernando");
arrayList.add("Mario");

//Pesquisando
int resultado = Collections.binarySearch(arrayList, "Albert");
//Imprimindo resultado
```

```
System.out.println("Resultado da pesquisa: "+resultado);
```

Resultado da execução:

```
Resultado da pesquisa: 1
```

O resultado obtido, conforme esperado, indica que o nome "Albert" encontra-se na segunda posição da coleção, lembrando que todas as coleções iniciam-se pelo índice 0 (zero), logo, o valor 1 indica a segunda posição.

Outras funcionalidades úteis

- Reverse – Inverte a ordem dos elementos;
- Fill – Preenche a coleção com o elemento especificado;
- Swap – troca os elementos de posição;
- addAll – Adiciona os elementos a coleção;
- frequency – Conta a quantidade de ocorrências do elemento especificado;
- disjoint – Verifica quando duas coleções não possuem nenhum elemento em comum.

EXERCíCIOS

Aprenda com quem também está aprendendo, veja e compartilhe as suas respostas no nosso Fórum:

Exercícios – Módulo 09 – Coleções – Framework Collections

1. Implemente código que possua um ArrayList e insira 200 Strings nesta lista.

2. Faça um ArrayList e insira as Strings "String 1", "String 2" e "String 3" cada uma duas vezes, percorra todos os elementos e imprima (System.out.println()). Observe o resultado. Quantos elementos têm a coleção?

3. Faça um HashSet e insira as Strings "String 1", "String 2" e "String 3" cada uma duas vezes, percorra todos os elementos e imprima (System.out.println()).

Observe o resultado. Quantos elementos têm a coleção?

4. Faça um HashMap e insira as Strings "String 1", "String 2" e "String 3" cada uma duas vezes, utilize o numeral como chave, percorra todos os elementos e imprima (System.out.println()). Observe o resultado. Quantos elementos têm a coleção?

5. Adicione os números 100, 20, 200, 30, 80, 40, 100, 200 a um List, percorra todos os elementos utilizando *for-enhanced* (for-each) e calcule a média.

6. Adicione os números 100, 20, 200, 30, 80, 40, 100, 200 a um List, percorra todos os elementos utilizando um Iterator e calcule a média.

7. Adicione os números 100, 20, 200, 30, 80, 40, 100, 200 a um Set, percorra todos os elementos utilizando *for-enhanced* (for-each) e calcule a média.

8. Porque as médias encontradas nos exercícios 6 e 7 ficaram diferentes?

9. Repita o exercício anterior porém, sem repetir os códigos das chaves. Observe o resultado. Quantos elementos têm a coleção?

10. Utilizando a classe Collections desordene a coleção criada no exercício 2. Imprima (System.out.println()) o resultado.

11. Utilizando a classe Collections pesquise por "String 2" na coleção criada no exercício 2. Imprima (System.out.println()) o resultado.

12. Utilizando a classe Collections ordente a coleção criada no exercício 2. Imprima (System.out.println()) o resultado.

13. Compare o desempenho das coleções HashMap e HashSet. Insira nas duas coleções um total de 20.000 alunos e pesquise por um deles, compare o tempo de pesquisa em cada uma das coleções.

14. Implemente uma classe ContaCorrente com os atributos Agencia, Numero,

Nome, CPF e Saldo. Sobreescreva os métodos equals() de forma que duas contas correntes sejam consideradas iguais apenas se possuírem a mesma Agencia e Numero, e sobreescreva o método hashCode().

15. Crie um HashSet e adicione 5 (cinco) ContaCorrente diferentes. Imprima o resultado (System.out.println()).

16. Crie uma classe Aluno com os atributos Nome , RG e Idade. Sobreescreva o método equals() de forma que dois alunos sejam considerados iguais apenas se possuírem o mesmo RG.

17. Qual interface do Framework Collections você utilizaria para representar uma Turma de faculdade, considerando que cada aluno está matriculado apenas uma vez?

18. Implemente código com a estrutura de dados definida no exercício anterior, adicione alguns alunos com o mesmo RG, utilize a classe Aluno do exercício 16. Percorra todos os elementos, observe o resultado. Existem alunos nesta turma com o mesmo código?

19. Sobreescreva o método hashCode da classe Aluno, desenvolvida no exercício 16, observando a implementação do método equals().

20. Faça novamente o exercício 8 porém, utilizando a nova classe Aluno, agora com o método hashCode() implementado.