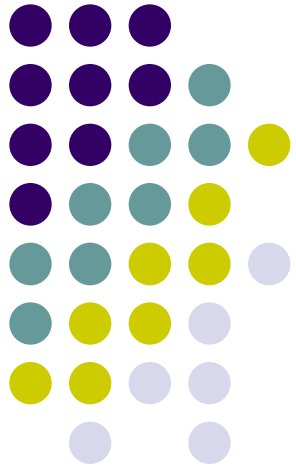
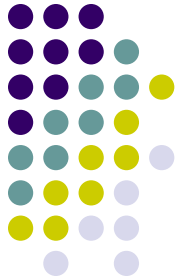
Coleções

Alberto Costa Neto
DComp - UFS

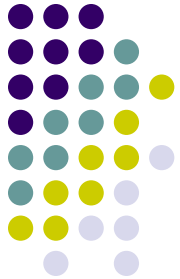


Coleções
[Conteúdo]



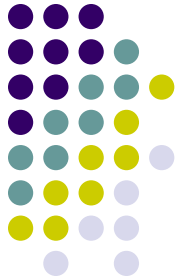
- Introdução
- Interfaces e Classes para Coleções
- Coleções Especiais
- Classes de Coleções Históricas
- Suporte Algorítmico
- Questões de Utilização
- Novas coleções da versão 1.4
- Exercícios
- Bibliografia

Coleções
[Introdução]



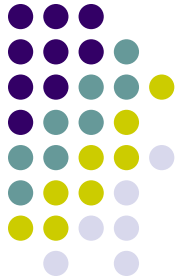
- Até a versão 1.1, o conjunto de classes era pequeno e possuía algumas limitações (desempenho)
- Na versão 1.2, foi introduzido um framework de coleções contendo um conjunto de interfaces e classes bem projetado para armazenar e manipular grupos de dados

Coleções
[Introdução]



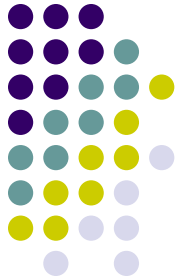
- Oferece muitos dos tipos abstratos de dados (ADT's) estudados em disciplinas de estrutura de dados, tais como:
  - Conjuntos (sets)
  - Listas (lists)
  - Árvores (trees)
  - Matrizes (arrays)
  - Tabelas hash (hashtables) e Mapas (maps)

Coleções
[Introdução]



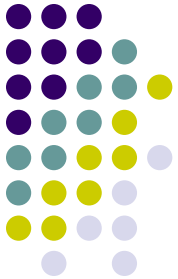
- As classes que implementam as coleções são orientadas a objetos, especificando:
  - O armazenamento dos dados
  - As operações que manipulam os dados

Coleções
[Interfaces e Classes para Coleções]

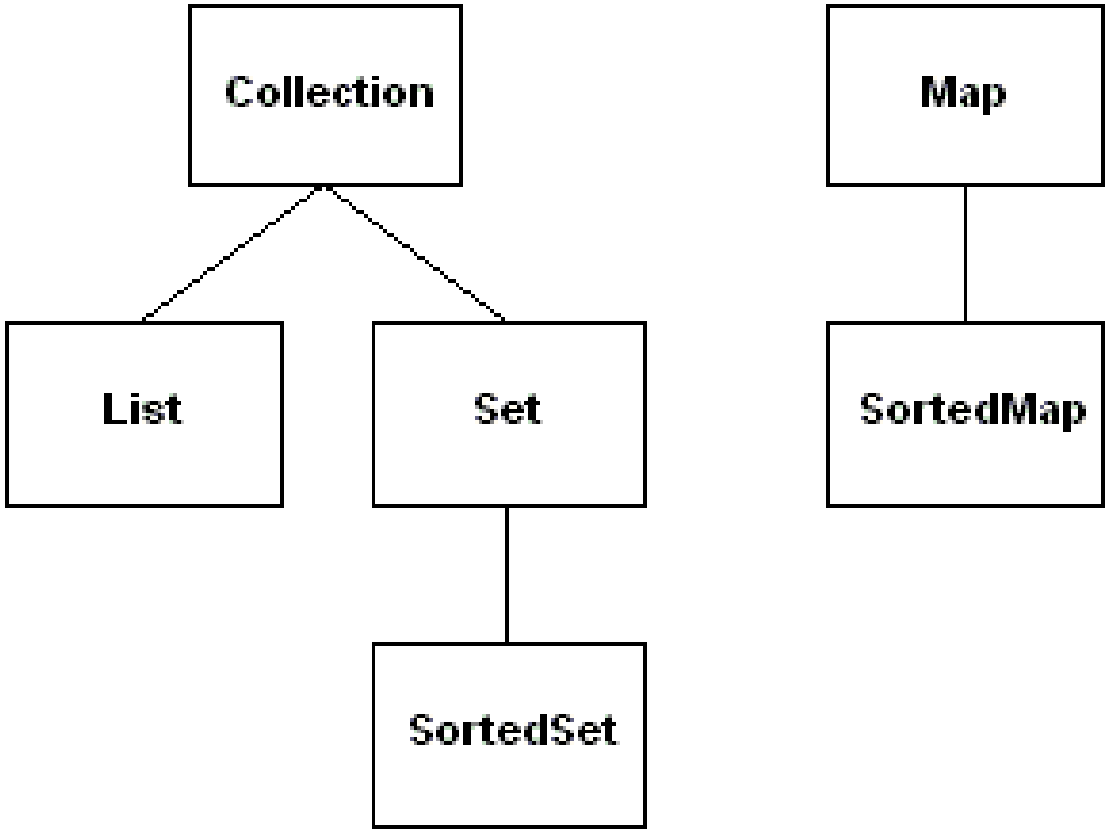


- O mais importante no framework de coleções de Java é entender as interfaces:
  - Há várias implementações (classes) para cada uma delas
  - Deve-se sempre trabalhar com as interfaces para permitir mudar de implementação sem grandes repercussões nos programas

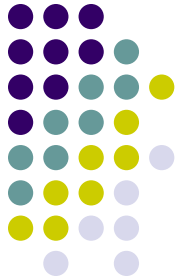
Coleções
[Interfaces e Classes para Coleções]



- Interfaces disponíveis

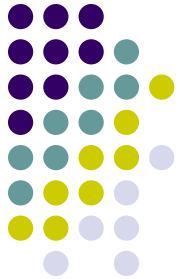


Coleções
[Interfaces e Classes para Coleções]



- Funções básica das interfaces:
  - **java.util.Collection:** representa um grupo de objetos em que são permitidas duplicatas
  - **java.util.List:** estende Collection, permite duplicatas e introduz a noção de posição através de índices
  - **java.util.Set:** estende Collection, mas proíbe a existência de duplicatas
  - **java.util.SortedSet:** estende Set, garantindo ordenação
  - **java.util.Map:** representa um grupo de objetos (valores) acessíveis através de chaves. Não aceita chaves duplicadas
  - **java.util.SortedMap:** estende Map, garantindo ordenação

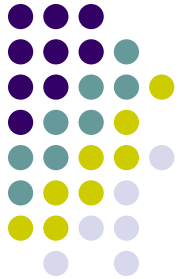
Coleções
[Interfaces e Classes para Coleções]



- Implementações das interfaces

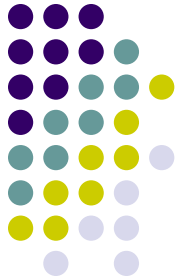
Interface	Implementações				Histórica
Tipo de Estrutura	**Tabela Hash**	**Array**	**Árvore Balanceada**	**Lista Duplamente Encadeada**	
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		Hashtable Properties

Coleções
[Interfaces e Classes para Coleções]



- Interface Collection
  - Representar qualquer grupo de objetos
  - Usada para trabalhar com um grupo de elementos da maneira mais geral possível

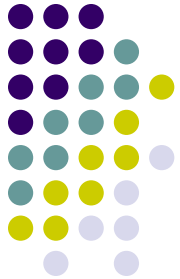
Coleções
[Interfaces e Classes para Coleções]



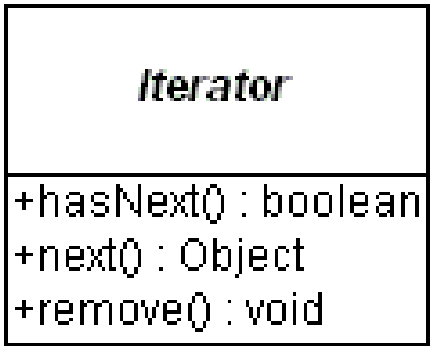
- Principais métodos
  - Adição e Remoção
    - add, remove e clear
  - Consulta
    - equals, size, isEmpty, contains e iterator
  - Manipulação de Grupos
    - containsAll
    - addAll
    - removeAll
    - retainAll
  - Conversão para Array
    - toArray

Collection
+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]

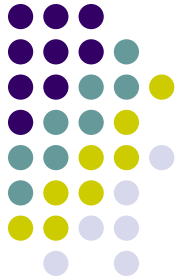
Coleções
[Interfaces e Classes para Coleções]



- Interface Iterator
  - Através dessa interface é possível percorrer os elementos de uma coleção do início até o fim e removê-los da coleção
  - O método iterator() da interface Collection retorna um Iterator, ou seja, um objeto que implementa a interface java.util.Iterator
  - É bastante semelhante à interface java.util.Enumeration



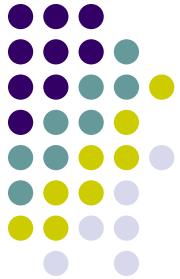
Coleções
[Interfaces e Classes para Coleções]



- Interface Set
  - Estende a interface Collection
  - Proíbe duplicatas
  - Não traz novos métodos, além dos definidos em Collection
  - Há 2 implementações para a interface Set (HashSet e TreeSet)

Set
+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]

Coleções
[Interfaces e Classes para Coleções]



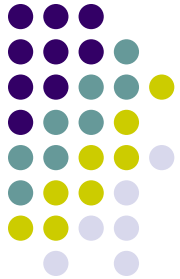
- java.util.HashSet é mais eficiente quando o acesso é baseado na chave
- java.util.TreeSet deve ser utilizado quando deseja-se acessar o Set de forma ordenada
- Os objetos inseridos em um Set devem implementar a interface java.lang.Comparable
  - int comparteTo()
- Exemplo
  - TesteSet

Coleções
[Interfaces e Classes para Coleções]

- Interface List
  - Estende a interface java.util.Collection
  - Permite duplicatas
  - Adiciona operações orientadas a posição
    - add, addAll, get, indexOf, lastIndexOf, remove e set
  - Trabalha com partes (subconjuntos) da lista
    - listIterator e subList

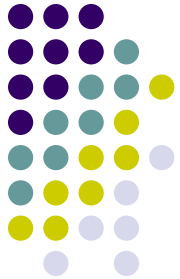
List
+add(element : Object) : boolean +add(index : int, element : Object) : void +addAll(collection : Collection) : boolean +addAll(index : int, collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +get(index : int) : Object +hashCode() : int +indexOf(element : Object) : int +iterator() : Iterator +lastIndexOf(element : Object) : int +listIterator() : ListIterator +listIterator(startIndex : int) : ListIterator +remove(element : Object) : boolean +remove(index : int) : Object +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +set(index : int, element : Object) : Object +size() : int +subList(fromIndex : int, toIndex : int) : List +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]

Coleções
[Interfaces e Classes para Coleções]



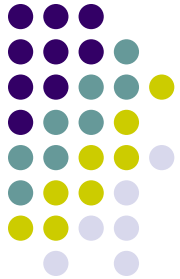
- java.util.ArrayList e java.util.LinkedList são as implementações de List
- ArrayList mantém um array com os elementos.
  - Quando precisa-se de mais espaço, um novo array é criado e os elementos do array antigo são copiados para o novo
- LinkedList utiliza uma lista duplamente encadeada

Coleções
[Interfaces e Classes para Coleções]



- ArrayList é mais apropriada quando os elementos são acessados de forma aleatória e são inseridos / removidos normalmente no final
- Caso contrário, é melhor utilizar LinkedList
- Filas, Pilhas e Deques podem ser facilmente implementados usando LinkedList
- Exemplo
  - TesteList

Coleções
[Interfaces e Classes para Coleções]



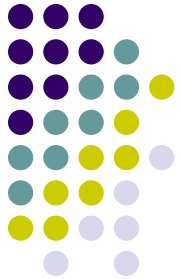
- Métodos específicos da classe LinkedList

LinkedList
+addFirst(element : Object) : void +addLast(element : Object) : void +getFirst() : Object +getLast() : Object +removeFirst() : Object +removeLast() : Object

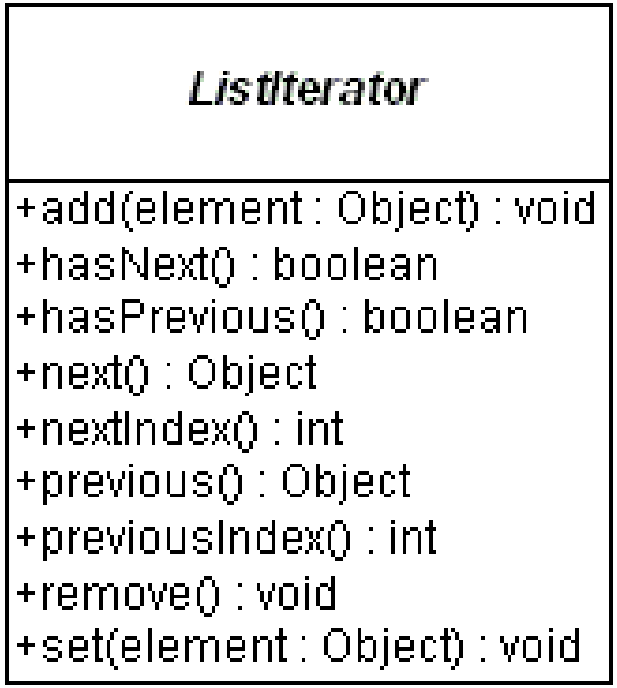
```
LinkedList fila = ...;
fila.addFirst(elemento);
Object o = fila.removeLast();
```

```
LinkedList pilha = ...;
pilha.addFirst(elemento);
Object objeto = pilha.removeFirst();
```

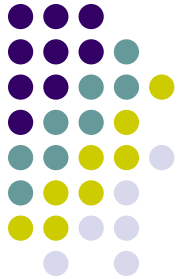
Coleções
[Interfaces e Classes para Coleções]



- Interface ListIterator
  - Estende a interface Iterator
  - Acesso bidirecional
  - Prevê adição, remoção e substituição de elementos da coleção
  - Exemplo
    - TesteListIterator



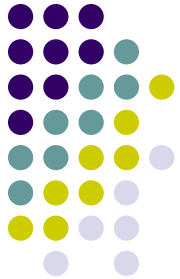
Coleções
[Interfaces e Classes para Coleções]



- Interface Map
  - Não é uma extensão da interface Collection
  - Trabalha com associações entre chave e valor
  - Não permite chaves duplicadas

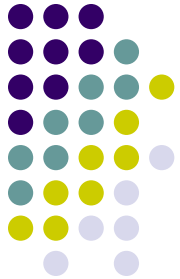
Map
+clear() : void +containsKey(key : Object) : boolean +containsValue(value : Object) : boolean +entrySet() : Set +get(key : Object) : Object +isEmpty() : boolean +keySet() : Set +put(key : Object, value : Object) : Object +putAll(mapping : Map) : void +remove(key : Object) : Object +size() : int +values() : Collection

Coleções
[Interfaces e Classes para Coleções]



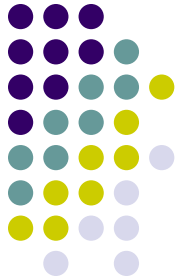
- Os métodos podem ser separados em três conjuntos de operações:
  - Alteração
    - put, remove, putAll e clear
  - Consulta
    - get, containsKey, containsValue, size e isEmpty
  - Visões alternativas
    - keySet, values e entrySet

Coleções
[Interfaces e Classes para Coleções]

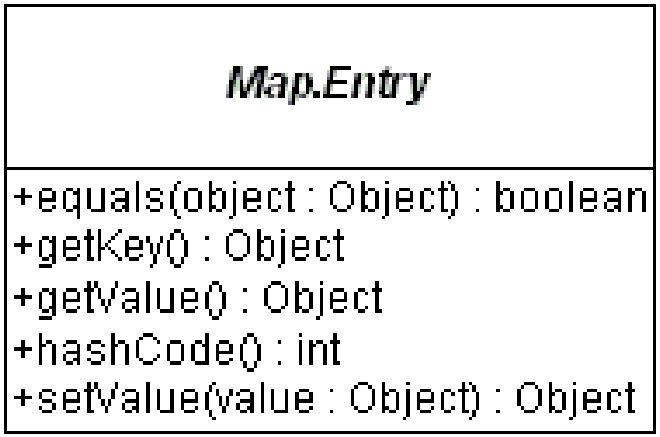


- Classes HashMap e TreeMap
  - Implementam a interface Map
  - Para inserir e remover elementos de um Map, HashMap oferece a melhor alternativa
  - Se for necessário ordenar pela chave, a melhor alternativa é o TreeMap
  - Dependendo do tamanho da coleção pode ser mais rápido adicionar elementos a HashMap e depois converter para um TreeMap para acessar os elementos de forma ordenada
  - Exemplo
    - TesteMap

Coleções
[Interfaces e Classes para Coleções]



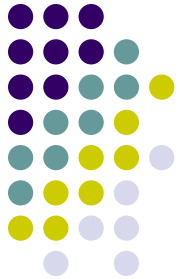
- Interface Map.Entry
  - Um Map.Entry representa um par chave/valor armazenado em um Map
  - O método entrySet definido em Map retorna um Set com objetos que implementam a Map.Entry
  - Exemplo
    - TesteMapEntry



Coleções
[Interfaces e Classes para Coleções]

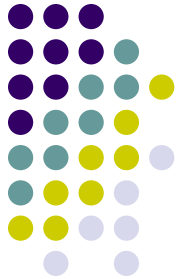
- Interface Comparable
  - Indica que uma classe tem ordenação natural
  - É possível ordenar uma coleção de objetos que a implementam
  - Permite criar algoritmos de classificação mais genéricos

Coleções
[Interfaces e Classes para Coleções]

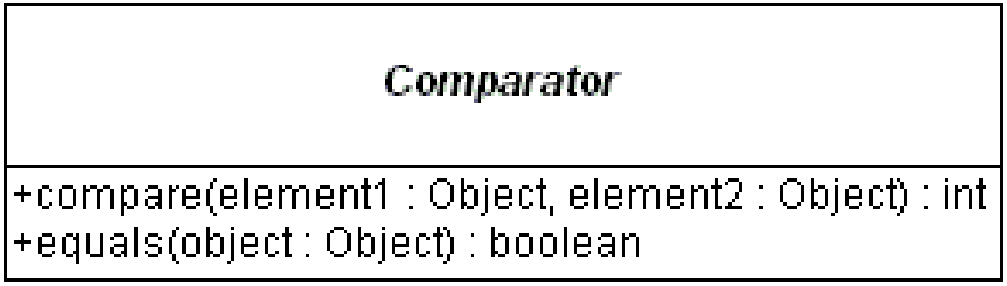


- O método compareTo() compara a instância corrente com o objeto passado
  - Se a instância corrente vem antes do argumento na ordem, o valor retornado é negativo
  - Se a instância corrente vem depois do argumento na ordem, o valor retornado é positivo
  - Caso contrário, o valor retornado é zero. Um valor zero significa que os objetos assumem o mesmo lugar em na ordem natural
- Exemplo
  - Pessoa

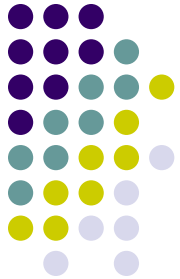
Coleções
[Interfaces e Classes para Coleções]



- Interface Comparator
  - Quando uma classe não foi projetada para implementar java.lang.Comparable, a solução é criar uma classe que implemente a interface java.util.Comparator
  - Além disso, é interessante utilizar um Comparator quando se precisa de um tipo de ordenação diferente do definido pela implementação de Comparable

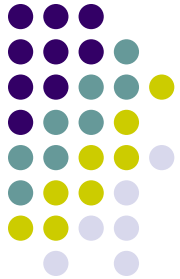


Coleções
[Interfaces e Classes para Coleções]

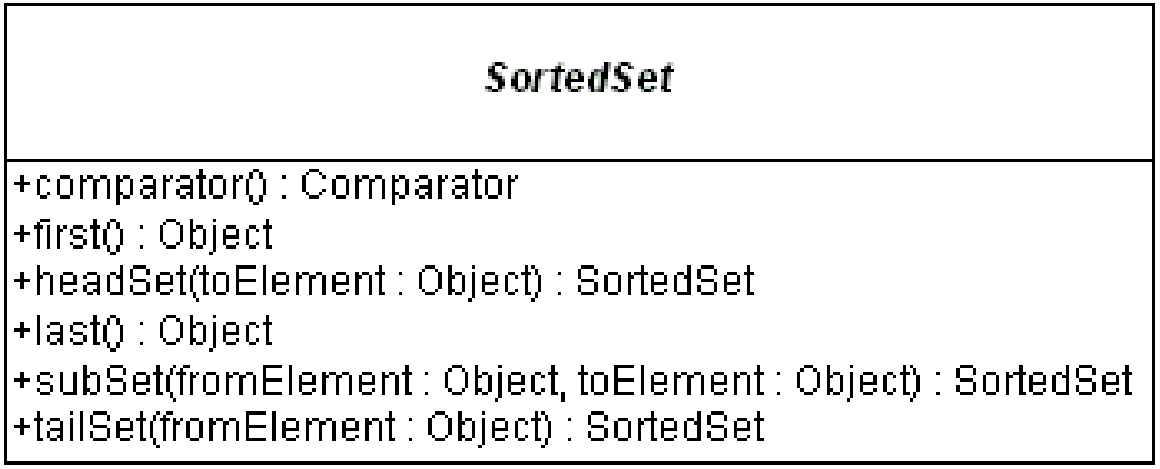


- Os valores de retorno do método compare() são similares ao do método compareTo() declarado na interface Comparable.
  - Se o primeiro elemento vem antes do segundo elemento na ordem, é retornado um valor negativo
  - Se o primeiro elemento vem depois, então um valor positivo é retornado
  - Caso contrário, o valor zero é retornado. Entretanto, o valor zero não significa que os elementos são iguais mas que os dois assumem a mesma ordem
  - Para verificar se são iguais deve-se usar o método equals() que retorna true quando são iguais e false caso contrário
- Exemplo
  - PessoaComparatorNome

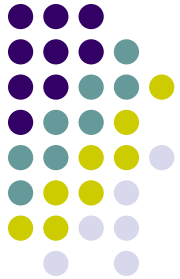
Coleções
[Interfaces e Classes para Coleções]



- Interface SortedSet
  - Extensão de Set que representa um conjunto ordenado
  - A classe java.util.TreeSet implementa SortedSet
  - Oferece acesso aos extremos do Set assim como a subconjuntos do Set

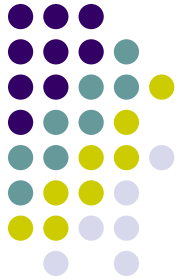


Coleções
[Interfaces e Classes para Coleções]

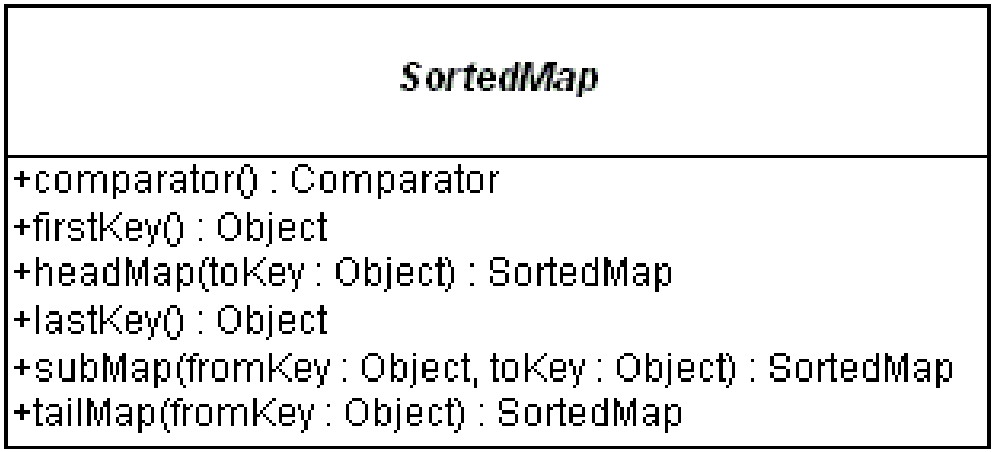


- Quando trabalhando com subconjuntos da lista, mudanças no subconjunto são refletidas no Set de origem
- Além disso, modificações no Set de origem refletem no subconjunto
- Isto funciona porque subconjuntos são identificados pelos elementos nos extremos, e não por índices
- Os elementos adicionados a um SortedSet devem implementar Comparable ou deve ser provido um Comparator no construtor da classe que implementa a interface SortedSet, no caso java.util.TreeSet

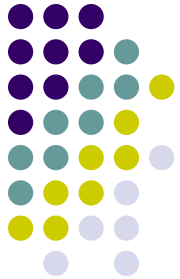
Coleções
[Interfaces e Classes para Coleções]



- Interface SortedMap
  - Extensão de Map que representa um conjunto ordenado de pares chave-valor
  - A classe java.util.TreeMap implementa SortedMap
  - Oferece acesso aos extremos do Set assim como a subconjuntos do Set

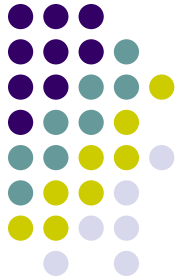


Coleções
[Interfaces e Classes para Coleções]



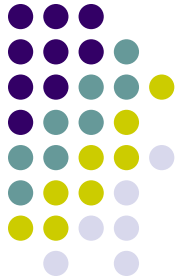
- A interface traz métodos de acesso aos finais do Map e para criar subconjuntos de Map
- SortedMap é parecido com SortedSet, exceto que a ordenação é feita através das chaves
- Como Map não admite mais de uma chave com o mesmo valor, ao tentar adicionar um novo par chave-valor ao Map, o valor antigo será trocado pelo novo

Coleções
[Coleções Especiais]



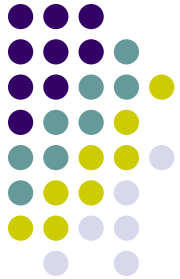
- Para manter o framework de coleções simples, funcionalidades adicionais são oferecidas através de wrappers, seguindo o Design Pattern Decorator
  - Os wrappers (decoradores) delegam as chamadas dos seus métodos às coleções, mas adicionam funcionalidades antes ou depois
  - java.util.Collections implementa a criação de coleções especiais (decoradores)

Coleções
[Coleções Especiais]



- Coleções somente para leitura
  - Às vezes é necessário tratar uma coleção como somente para leitura após terminar de adicionar os elementos para prevenir modificações acidentais.
  - Para prover essa capacidade, a classe Collections traz seis factory methods – seguindo o design pattern factory method – um para cada tipo de coleção (Collection, List, Map, Set, SortedMap, SortedSet)

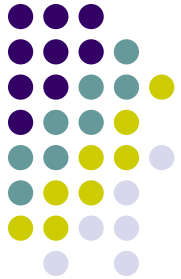
Coleções
[Coleções Especiais]



- Ao chamar uma operação que modifica a coleção, uma UnsupportedOperationException é lançada indicando que não é suportada
- Métodos

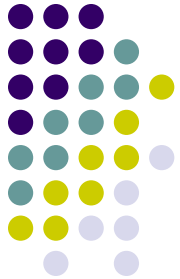
```
Collection unmodifiableCollection(Collection collection)
List unmodifiableList(List list)
Map unmodifiableMap(Map map)
Set unmodifiableSet(Set set)
SortedMap unmodifiableSortedMap(SortedMap map)
SortedSet unmodifiableSortedSet(SortedSet set)
```

Coleções
[Coleções Especiais]



- Coleções sincronizadas
  - Uma diferença chave entre as coleções históricas e as novas implementações é que elas não são sincronizadas, ou seja, não seguras quando se trabalha com *multithreading*
    - Isso foi feito permitir que sincronização não seja usada quando não for necessária (não penaliza o desempenho)
    - java.util.Collections permite envolver (decorar) as coleções existentes em outras com sincronização

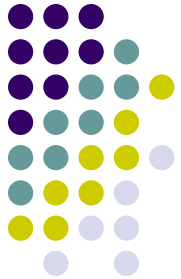
Coleções
[Coleções Especiais]



- Tornar um coleção somente para leitura também torna uma coleção segura em ambientes *multithreading* já que evita que ela seja modificada (imutabilidade), evitando assim o *overhead* da sincronização
- Métodos

```
Collection synchronizedCollection(Collection collection)
List synchronizedList(List list)
Map synchronizedMap(Map map)
Set synchronizedSet(Set set)
SortedMap synchronizedSortedMap(SortedMap map)
SortedSet synchronizedSortedSet(SortedSet set)
```

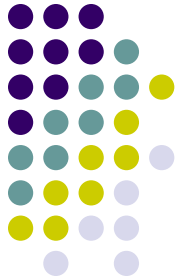
Coleções
[Coleções Especiais]



- Coleções singleton
  - java.util.Collections provê uma maneira fácil de criar um Set que possui um único elemento, através do método singleton(Object o).
  - Isto evita o trabalho de criar o Set e adicionar o objeto em um passo posterior. O Set retornado é somente para leitura

```
Set set = Collections.singleton("Singleton");
```

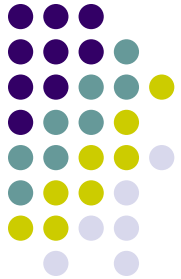
Coleções
[Coleções Especiais]



- Coleções com múltiplas cópias
  - A classe Collections permite criar uma lista imutável com múltiplas cópias do mesmo elemento
  - Método

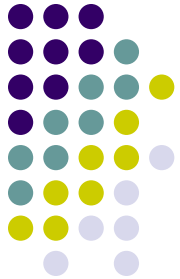
```
List nCopies(int n, Object element)
```

Coleções
[Coleções Especiais]



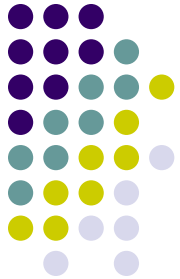
- Coleções vazias
  - Collections disponibiliza coleções vazias e imutáveis através das constantes
    - EMPTY_LIST (um List vazio)
    - EMPTY_SET (um Set vazio)

Coleções
[Classes de Coleções Históricas]



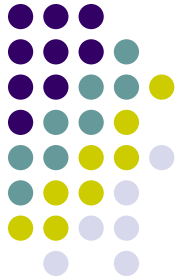
- Arrays
  - Arrays são definidos como coleções de tamanho fixo de elementos de um mesmo tipo
  - São as únicas estruturas de dados que suportam o armazenamento de tipos primitivos de dados
  - Ao criar um array, é necessário especificar o tamanho e o tipo do dado que se deseja armazenar
  - Para fazer uma cópia de um array pode-se utilizar o método arraycopy da classe System

Coleções
[Classes de Coleções Históricas]



- Classes Vector e Stack
  - Um java.util.Vector é uma classe de coleção histórica que funciona como um array redimensionável e com capacidade de armazenar elementos heterogêneos
  - Vector foi adaptada para funcionar no novo framework como uma classe que implementa List
  - Ainda assim, é melhor utilizar ArrayList

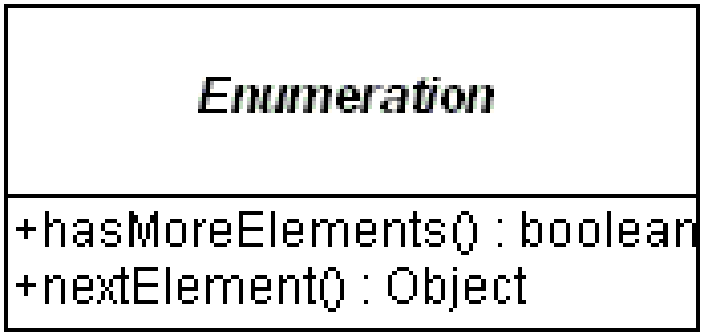
Coleções
[Classes de Coleções Históricas]



- Ao fazer a transição de Vector para List basta trocar o método setElementAt pelo método set, definido na interface List
- A classe java.util.Stack estende Vector, adicionando os métodos push(Object o) e Object pop()
  - É importante tomar cuidado pois é possível acessar e modificar os elementos intermediários da pilha através dos métodos herdados de Vector

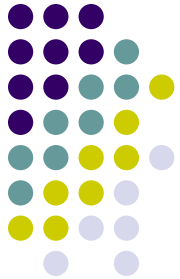
Coleções
[Classes de Coleções Históricas]

- Interface Enumeration
  - Iteração na coleção
  - Foi substituída pela interface Iterator
  - Como nem todas as bibliotecas suportam a interface nova, pode ser necessário usá-la



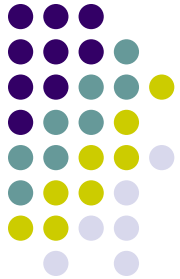

```
Enumeration e = ...;
while (e.hasMoreElements()) {
    Object o = e.nextElement();
    ....
}
```

Coleções
[Classes de Coleções Históricas]



- Classes Dictionary
  - Forma uma base para as classes de coleções históricas que armazenam pares chave-valor
  - Só possui métodos abstratos. Em outras palavras, deveria ser uma interface.
  - Substituída por java.util.Map no novo *framework*
  - java.util.Hashtable e java.util.Properties são as implementações disponíveis de Dictionary.
  - Métodos elements e keys

Coleções
[Classes de Coleções Históricas]

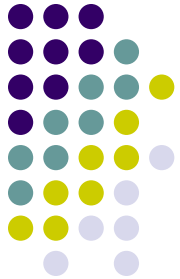


- Classes Hashtable
  - É um dicionário genérico que permite armazenar objetos
  - Foi adaptada para que implementasse a interface Map

Coleções
[Classes de Coleções Históricas]

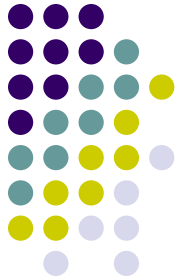
- Classes Properties
  - É uma especialização de Hashtable para trabalhar com chaves e valores do tipo String, sem *casting (*métodos setProperty, getProperty e getProperties)
  - Suporta também a leitura e gravação de pares chave-valor através de streams (load e store)
  - (A utilização mais comum dessa classe é na obtenção das propriedades do sistema, através do método getProperties() da classe java.lang.System

Coleções
[Classes de Coleções Históricas]



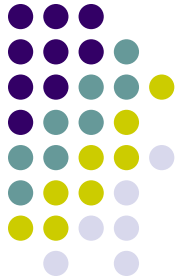
- Classe BitSet
  - É uma representação alternativa de um conjunto
    - Dado um número finito de n objetos, é possível associar um único inteiro com cada objeto
    - Então, cada possível subconjunto dos objetos correspondentes a um vetor de n-bits, com cada bit “on” ou “off” a depender da presença ou não do objeto no subconjunto
    - Para pequenos valores de n, um vetor de bits pode ser uma representação extremamente compacta
    - Entretanto, para grandes valores de n, um vetor de bits pode ser ineficiente quando a maioria dos bits está “off”

Coleções
[Classes de Coleções Históricas]



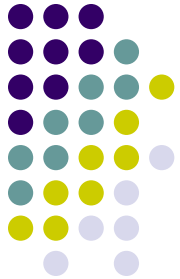
- Os métodos set, get e clear permitem alterar, obter e limpar o valor de um bit
- Operações lógicas (AND, OR, XOR e AND NOT)

Coleções
[Suporte Algorítmico]



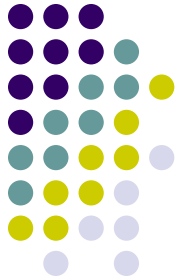
- As classes java.util.Collections e java.util.Arrays oferecem suporte a vários algoritmos para coleções
- Há operações para ordenação, busca, verificação de igualdade e manipulação

Coleções
[Suporte Algorítmico]



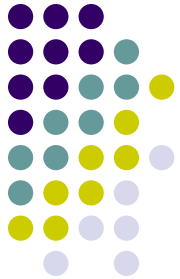
- Ordenação
  - Ao contrário de Set e Map, não há uma implementação ordenada de List
  - Além disso, antes do framework de coleções, não havia suporte para a ordenação de arrays
  - O novo framework trouxe a capacidade de ordenar arrays e listas
  - Em qualquer tipo de ordenação, todos os itens devem implementar java.lang.Comparable. Caso contrário, a exceção java.lang.ClassCastException é lançada

Coleções
[Suporte Algorítmico]



- A ordenação de uma List é feita através de um dos dois métodos sort() da classe Collections
  - Se o tipo de elemento implementa Comparable pode-se usar a versão sort(List list)
  - Caso contrário será necessário fornecer um objeto java.lang.Comparator e usar o método sort(List list, Comparator comparator)
  - As duas versões modificam o objeto List passado.

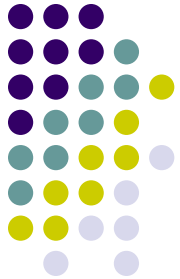
Coleções
[Suporte Algorítmico]



- A ordenação de arrays é feita através de métodos definidos na classe java.util.Arrays
- Há dois métodos para ordenar cada um dos sete tipos primitivos (excetuando-se boolean). Um para ordenar o array completo e outro para ordenar parte do array

```
void sort(<tipo> array[])
void sort(<tipo> array[], int fromIndex, int toIndex)
```

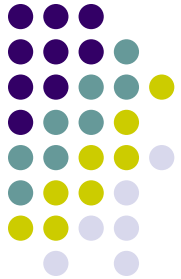
Coleções
[Suporte Algorítmico]



- Há outros quatro para ordenar arrays de objetos, diferenciando-se por ordenar parte ou todo o array e por utilizar ou não um Comparator

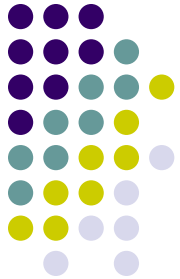
```
void sort(Object array[])
void sort(Object array[], int fromIndex, int toIndex)
void sort(Object array[], Comparator comparator)
void sort(Object array[], int fromIndex, int toIndex,
Comparator comparator)
```

Coleções
[Suporte Algorítmico]



- Busca
  - As classes Collections e Arrays provêem mecanismos para fazer buscas em List e arrays, assim como encontrar os valores mínimos e máximos em objetos Collection
  - Quando o método contains() de List é chamado para encontrar um elemento, ele assume que a lista está desordenada

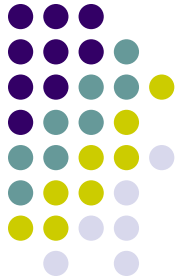
Coleções
[Suporte Algorítmico]



- Busca em List
  - Porém, se a lista já está ordenada, é possível fazer uma busca mais rápida através dos métodos binarySearch
  - Se os objetos da lista implementam Comparable, não é necessário passar um Comparator
  - Não é interessante fazer busca binária em LinkedList, já que nesse tipo de estrutura, o acesso aleatório é bastante lento

```
int binarySearch(List list, Object key)
int binarySearch(List list, Object key, Comparator comp)
```

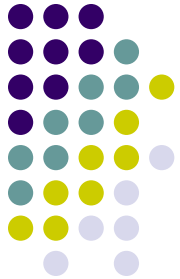
Coleções
[Suporte Algorítmico]



- Busca em Array
  - Há sete métodos binarySearch() na classe Arrays para os sete tipos primitivos (exceto boolean) e dois para busca em arrays de objetos (um deles requer um Comparator)

```
int binarySearch(byte[] a, byte key)
int binarySearch(short[] a, short key)
int binarySearch(int[] a, int key)
int binarySearch(long[] a, long key)
int binarySearch(float[] a, float key)
int binarySearch(double[] a, double key)
int binarySearch(char[] a, char key)
int binarySearch(Object[] a, Object key)
int binarySearch(Object[] a, Object key, Comparator c)
```

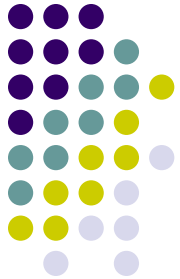
Coleções
[Suporte Algorítmico]



- É possível obter os elementos extremos de uma coleção: o mínimo e o máximo
- Se a coleção já está ordenada basta pegar o primeiro e o último
- Entretanto, para coleções não ordenadas, pode-se utilizar os métodos min() e max() da classe Collections
- Se os objetos na coleção não implementam Comparable, então um Comparator deve ser fornecido

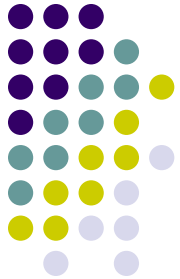
```
Object max(Collection collection)
Object max(Collection collection, Comparator comparator)
Object min(Collection collection)
Object min(Collection collection, Comparator comparator)
```

Coleções
[Suporte Algorítmico]



- Manipulação
  - As classes Collections e Arrays oferecem várias maneiras de manipular os elementos de List ou array (não há para Set e Map)
  - Com uma List, a classe Collections pode substituir todos os elementos por um único elemento, copiar uma lista inteira para outra, inverter todos os elementos ou misturá-los
    - Ao copiar de uma lista para outra, se a lista de destino é maior, nada é feito com os elementos restantes

Coleções
[Suporte Algorítmico]

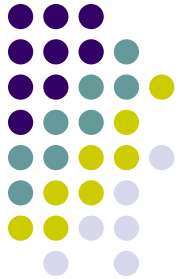


- Métodos para manipulação de List

```
void fill(List list, Object element)
void copy(List source, List destination)
void reverse(List list)
void shuffle(List list)
void shuffle(List list, Random random)
```

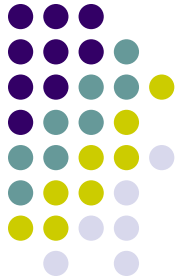
- A classe Arrays também permite trocar um array inteiro ou parte dele por um único elemento
- Todos os métodos são da forma fill(array, element) ou fill(array, fromIndex, toIndex, element), onde array pode ser um array de tipos primitivos ou de objetos

Coleções
[Questões de Utilização]



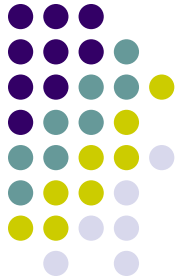
- O framework de coleções foi projetado tal que as classes novas do framework e as históricas possam ser convertidas
- Seria bom que todo o código novo utilizasse o framework mas algumas vezes não é possível

Coleções
[Questões de Utilização]



- Convertendo de coleções históricas para coleções novas
  - Há métodos de conveniência para converter de muitas das classes e interfaces de coleções originais para classes e interfaces novas
  - Eles servem como pontes quando é necessário uma nova coleção mas se tem uma coleção histórica
  - É possível passar de um Vector para List, de Hashtable para Map ou de Enumeration para qualquer Collection.

Coleções
[Questões de Utilização]

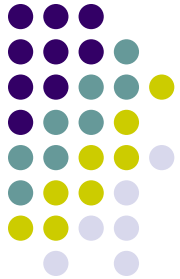


- Para passar de qualquer array para uma List
  - Método asList(Object array[]) da classe Arrays

```
String nomes[] = {"José", "Maria", "Pedro", "Clara"};
List lista = Arrays.asList(nomes);
```

- Como as classes Vector e Hashtable se adaptaram ao novo *framework,* não há trabalho algum de conversão

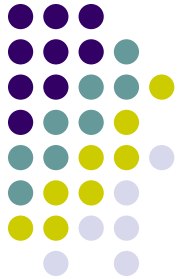
Coleções
[Questões de Utilização]



- Converter de Enumeration para algo no novo framework requer um pouco mais de trabalho
  - Criar uma coleção do novo framework
  - Adicionar cada elemento nela

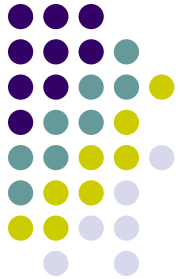
```
Enumeration e = ...;
Collection colecao = new LinkedList();
while (e.hasMoreElements()) {
 colecao.add(e.nextElement());
}
```

Coleções
[Questões de Utilização]



- Convertendo das coleções novas para coleções históricas
  - Além de suportar o uso das classe de coleções antigas dentro do novo framework de coleções, há também suporte para usar as bibliotecas novas e antigas que só suportam as coleções originais
  - É fácil converter de Collection para um array, Vector ou Enumeration, assim como de Map para Hashtable

Coleções
[Questões de Utilização]



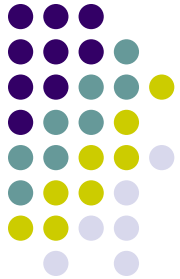
- Collection para array
  - O método toArray de Collection coloca os objetos contidos na coleção em um novo array

```
Collection colecao = ...;
Object array[] = colecao.toArray();
```

- Collection para Vector
  - O construtor de Vector aceita um Collection

```
Dimension dims[] = { new Dimension (0,0),
                     new Dimension (0,0) };
Collection c = Arrays.asList(dims);
Vector v = new Vector(c);
```

Coleções
[Questões de Utilização]



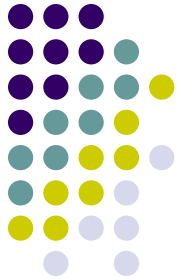
- Collection para Enumeration
  - A classe java.util.Collections inclui um método estático para fazer a conversão

```
Collection collection = ...;
Enumeration enum = Collections.enumeration(collection);
```

- Map para Hashtable
  - O construtor de Hashtable aceita um Map

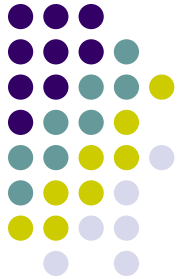
```
Map map = ...;
Hashtable hashtable = new Hashtable(map);
```

Coleções
[Novas coleções da versão 1.4]



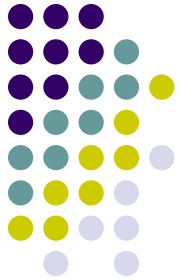
- LinkedHashSet
  - Diferencia-se de HashSet por manter uma lista duplamente encadeada na ordem de inserção
  - Percorre (com Iterator) na seqüência de inserção no Set
- WeakHashMap
  - Diferencia-se de HashMap por usar WeakReference para referenciar os objetos
  - Permite que o coletor de lixo libere o valor associado à chave não mais referenciada no programa
  - Funciona apenas para objetos que testam a identidade (através de ==) no método equals, o que não é o comum

Coleções
[Novas coleções da versão 1.4]



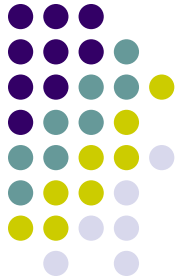
- LinkedHashMap
  - Diferencia-se de HashMap por manter uma lista duplamente encadeada na ordem de inserção
  - Percorre na seqüência de inserção no Map (chaves, valores e entries)
- IdentityHashMap
  - Utiliza como função hash o método System.identityHashCode (baseia-se no endereço de memória)
  - Utiliza == no lugar de equals para comparação

Coleções
[Exercícios]



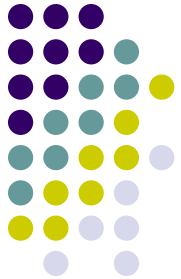
1) Escreva um programa que compare o desempenho das classes HashSet e TreeSet. A comparação deve ser feita medindo os tempos de inserção, remoção, obtenção e navegação.

Coleções
[Exercícios]



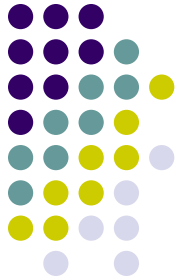
2) Implementar uma classe chamada OrderedList que armazena os objetos em uma lista, que deve ser passada no construtor. Implemente métodos semelhantes aos definidos na interface List.

Coleções
[Exercícios]



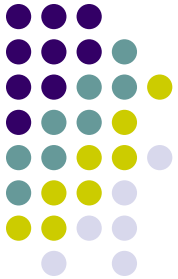
3) Implemente um programa que receba um conjunto de números na linha de comando, coloque-os em uma List. Em seguida, utilize a classe Collections para ordenar a lista de forma decrescente através de um Comparator.

Coleções
[Exercícios]



4) Implemente um programa semelhante ao da exercício 3, mas utilizando um array no lugar de List e a classe Arrays no lugar de Collections.

Coleções
[Bibliografia]



- Deitel (capítulos 19 a 21)
- Core Java 2 – Fundamentals (capítulo 2)