Estrutura de Dados

Collections Java

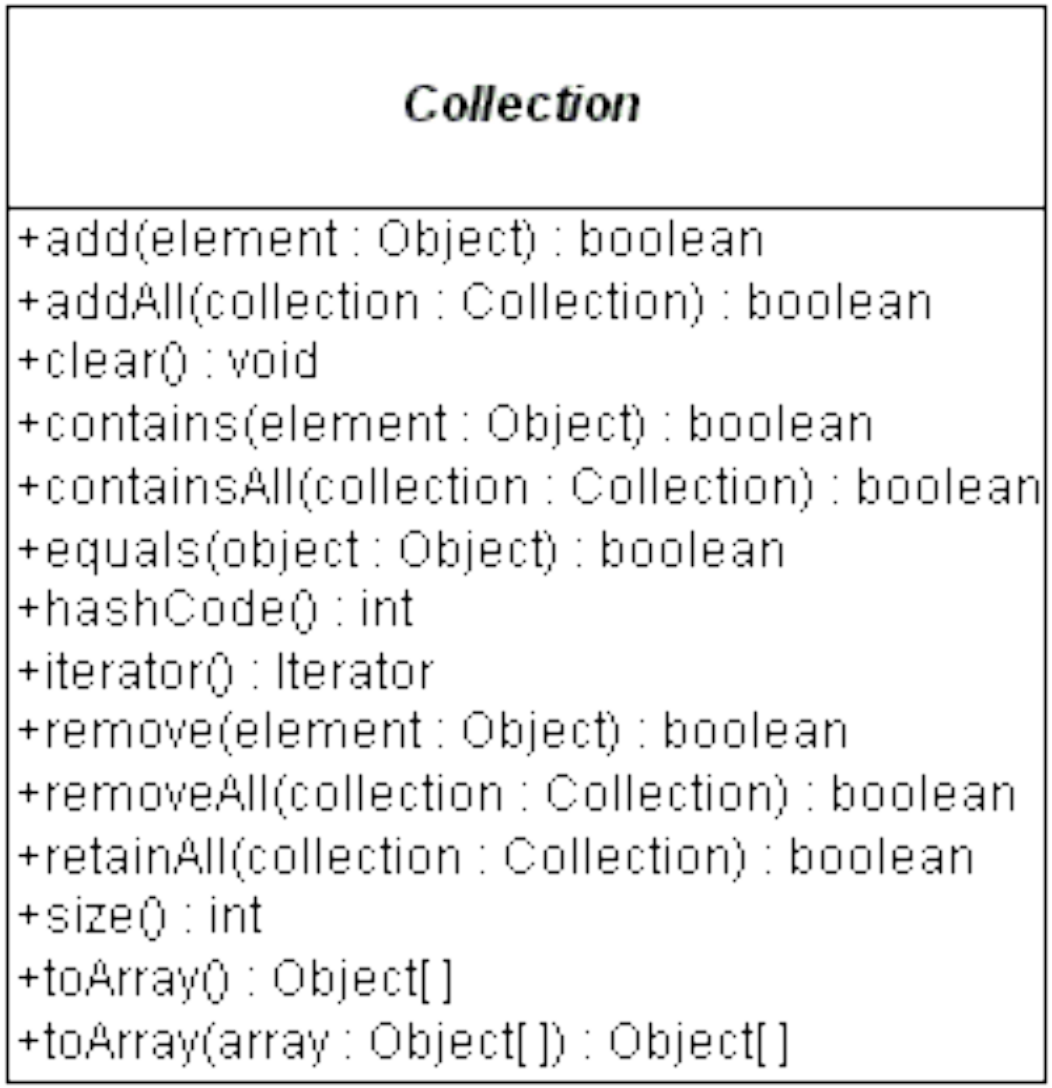
http://dl.dropbox.com/u/3025380/ED/aula17.pdf

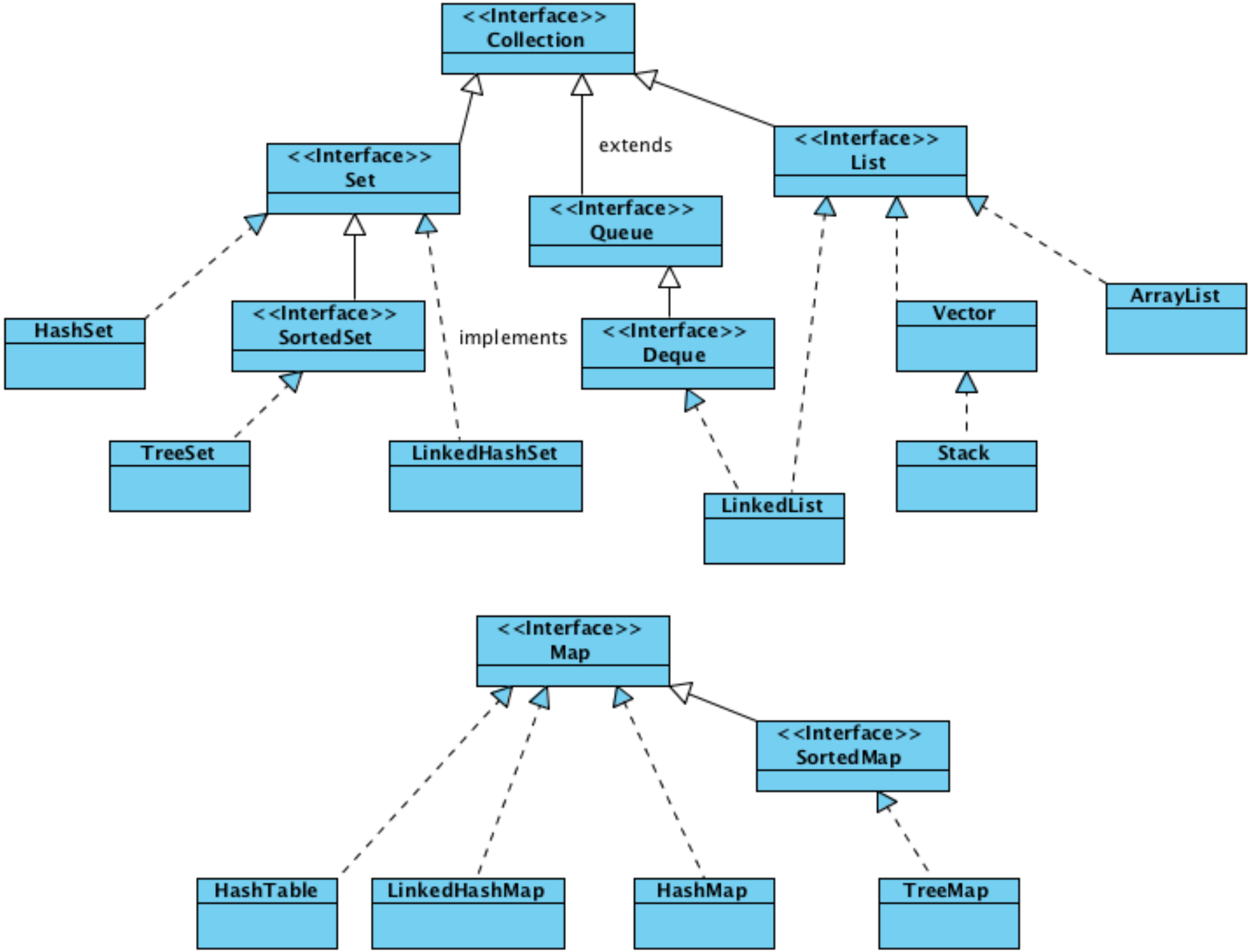
flavio.ceci@unisul.br

Introdução

- O Framework Java Colelctions é um conjunto de interfaces e classes dentro dos pacotes:
  - java.util;
  - java.util.concurrent.
- Tem como função prover uma série de recursos para tratar uma coleção de dados.
- Interface base para todos os tipos de coleção.

Collection


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[ ]


<<Interface>>
Collection
<<Interface>>
Set
extends
<<Interface>>
List
<<Interface>>
Queue
HashSet
<<Interface>>
SortedSet
implements
<<Interface>>
Deque
Vector
ArrayList
TreeSet
LinkedHashSet
LinkedList
Stack
<<Interface>>
Map
<<Interface>>
SortedMap
HashTable
LinkedHashMap
HashMap
TreeMap

Interface ***List***

List

- Interface que extende Collection, e que define coleções ordenadas (sequências), onde se tem o controle total sobre a posição de cada elemento, identificado por um índice numérico.
- Na maioria dos casos, pode ser encarado como um "array de tamanho variável" pois, como os arrays primitivos, é acessível por índices, mas além disso possui métodos de inserção e remoção.

ArrayList *implements List*

- Implementação de List que utiliza internamente um array de objetos.
- Em uma inserção onde o tamanho do array interno não é suficiente, um novo array é alocado (de tamanho igual a 1.5 vezes o array original), e todo o conteúdo é copiado para o novo array.
- Em uma inserção no meio da lista (índice < tamanho), o conteúdo posterior ao índice é deslocado em uma posição.

ArrayList *implements List*

- Esta implementação é a recomendada quando o tamanho da lista é previsível (evitando realocações) e as operações de inserção e remoção são feitas, em sua maioria, no fim da lista (evitando deslocamentos), ou quando a lista é mais lida do que modificada (otimizado para leitura aleatória).

LinkedList *implements List*

- Implementação de List que utiliza internamente uma lista encadeada.
- A localização de um elemento na n-ésima posição é feita percorrendo-se a lista da ponta mais próxima até o índice desejado.
- A inserção é feita pela adição de novos nós, entre os nós adjacentes, sendo que antes é necessária a localização desta posição.

LinkedList *implements List*

- Esta implementação é recomendada quando as modificações são feitas em sua maioria tanto no início quanto no final da lista, e o percorrimento é feito de forma sequencial (via Iterator) ou nas extremidades, e não aleatória (por índices).
- Um exemplo de uso é como um fila (FIFO - First-In-First-Out), onde os elementos são retirados da lista na mesma sequência em que são adicionados.

Vector *implements List*

- Implementação de List com o mesmo comportamento da ArrayList, porém, totalmente sincronizada.
- Por ter seus métodos sincronizados, tem performance inferior ao de uma ArrayList, mas pode ser utilizado em um ambiente multitarefa (acessado por várias threads) sem perigo de perda da consistência de sua estrutura interna.

Vector *implements List*

- Em sistemas reais, essa sincronização acaba adicionando um overhead desnecessário, pois mesmo quando há acesso multitarefa à lista (o que não acontece na maioria das vezes), quase sempre é preciso fazer uma nova sincronização nos métodos de negócio, para \'atomizarem\' operações seguidas sobre o Vector.

Vector *implements List*

- Por exemplo, uma chamada ao método contains(), seguida de uma chamada do método add() caso o elemento não exista, podem causar condições de corrida, se há acessos de várioas threads ao mesmo tempo.
- Assim, acaba sendo necessária uma sincronização adicional, englobando estas duas operações e tornando desnecessária a sincronização inerente da classe.

Vector *implements List*

- Portanto, a não ser que hajam acessos simultâneos de várias threads à lista, e a sincronização simples, provida pela classe Vector, seja o bastante, prefira outras implementações como ArrayList e LinkedList, que oferecem performance superior.

Stack *implements List*

- Implementação de List que oferece métodos de acesso para uso da lista como uma pilha (LIFO - Last-In-First-Out), como push(), pop() e peek().
- Estende Vector, portanto herda as vantagens e desvantagens da sincronização deste.
- Pode ser usado para se aproveitar as implementações das operações específicas de pilha.

Interface ***Set***

Set

- Interface que define uma coleção, ou conjunto, que não contém duplicatas de objetos.
- Isto é, são ignoradas as adições caso o objeto ou um objeto equivalente já exista na coleção.
- Por objetos equivalentes, entenda-se objetos que tenham o mesmo código hash (retornado pelo método hashCode()) e que retornem verdadeiro na comparação feita pelo método equals().

Set

- Não é garantida a ordenação dos objetos, isto é, a ordem de iteração dos objetos não necessariamente tem qualquer relação com a ordem de inserção dos objetos.
- Por isso, não é possível indexar os elementos por índices numéricos, como em uma List.

HashSet *implements Set*

- Implementação de Set que utiliza uma tabela hash (a implementação da Sun utiliza a classe HashMap internamente) para guardar seus elementos.
- Não garante a ordem de iteração, nem que a ordem permanecerá constante com o tempo (uma modificação da coleção pode alterar a ordenação geral dos elementos).

HashSet *implements Set*

- Por utilizar o algoritmo de tabela hash, o acesso é rápido, tanto para leitura quanto para modificação.

LinkedHashSet *Implements Set*

- Implementação de Set que estende HashSet, mas adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, uma iteração sobre seus elementos (utilizando o Iterator) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior).
- Internamente, é mantida uma lista duplamente encadeada que mantém esta ordem.

LinkedHashSet *Implements Set*

- Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leve queda na performance em relação à HashSet, mas ainda é mais rápida que uma TreeSet, que utiliza comparações para determinar a ordem dos elementos.

Interface ***Map***

Map

- Interface que define um array associativo, isto é, ao invés de números, objetos são usados como chaves para se recuperar os elementos.
- As chaves não podem se repetir (seguindo o mesmo princípio da interface Set), mas os valores podem ser repetidos para chaves diferentes.
- Um Map também não possui necessariamente uma ordem definida para o percorrimento.

HashMap *implements Map*

- Implementação de Map que utiliza uma tabela hash para armazenar seus elementos.
- O tempo de acesso aos elementos (leitura e modificação) é constante (muito bom) se a função de hash for bem distribuída, isto é, a chance de dois objetos diferentes retornarem o mesmo valor pelo método hashCode() é pequena.

LinkedHashMap *implements Map*

- Implementação de Map que estende HashMap, mas adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, uma iteração sobre seus elementos (utilizando o Iterator) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior).
- Internamente, é mantida uma lista duplamente encadeada que mantém esta ordem.

LinkedHashMap *implements Map*

- Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leve queda na performance em relação à HashMap, mas ainda é mais rápida que uma TreeMap, que utiliza comparações para determinar a ordem dos elementos.

Hashtable *implements Map*

- Assim como o Vector, a Hashtable é um legado das primeiras versões do JDK, igualmente sincronizado em cada uma de suas operações.
- Pelos mesmos motivos da classe Vector, dê preferência a outras implementações, como a HashMap, LinkedHashMap e TreeMap, pelo ganho na performance.

Próxima aula prova

- Conteúdo:
  - Arvores binárias;
  - Hashing;
  - Rever os exercícios.