Curso de Java 8 para Web

Professor
Antonio Benedito Coimbra Sampaio Jr

abc | Treinamentos

www.abctreinamentos.com.br

Segunda

Disciplina

JAVA 8 - Pacotes, Tratamento de Exceções, Applets, Genéricos, Collections, Lambdas, Streams e Interfaces Gráficas

- **UNIDADE 1:** Pacotes, Erros e Exceções
- **UNIDADE 2:** Applets, Anotações e Entrada/Saída
- **UNIDADE 3:** Genéricos
- **UNIDADE 4: Framework Collections**
- **UNIDADE 5:** Novidades Java 8
- **UNIDADE 6:** Aplicações Gráficas em Java

UNIDADE 4

FRAMEWORK COLLECTIONS

Introdução ao Framework Collections

Framework Collections

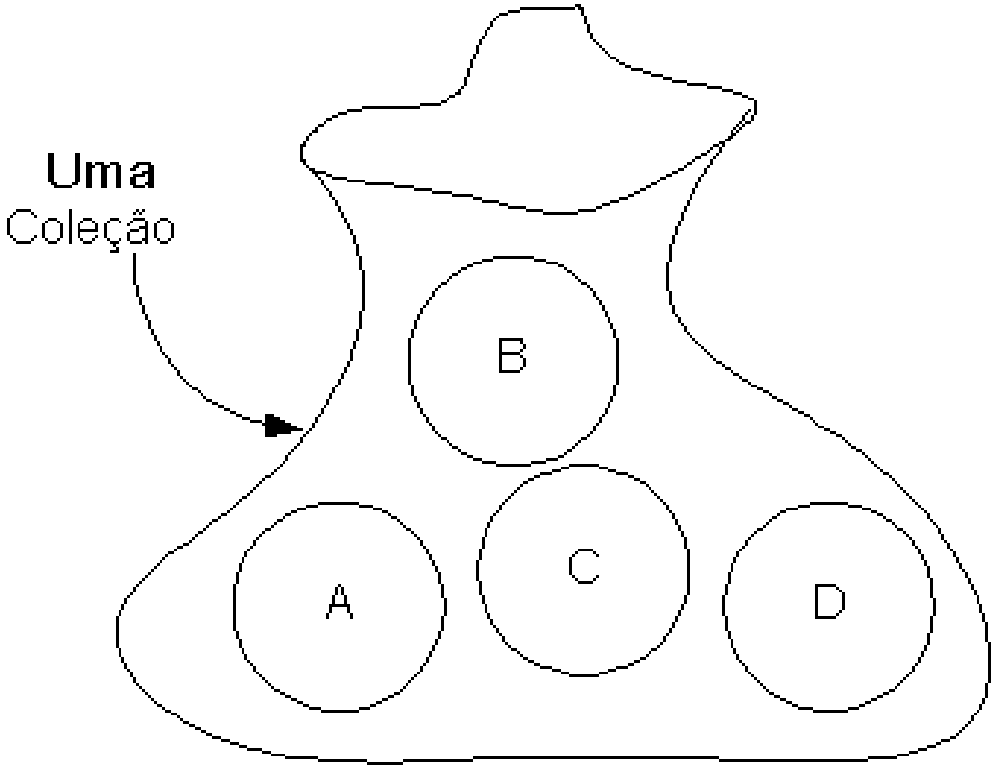
Definição:

- O ***Framework Collections*** é uma arquitetura unificada para a representação e a manipulação de coleções. É composta de **Interfaces** (são os tipos de dados que representam as coleções), **Implementações** (dessas interfaces) e os **Algoritmos** (prontos para realizarem pesquisa e ordenação, entre outras funcionalidades).

- Os principais benefícios desse framework são:
  - 1. Redução do esforço de programação;
  - 2. Melhoria da qualidade dos programas escritos;
  - 3. Redução da curva de aprendizado de novas APIs;
  - 4. Estímulo para o reuso de software;
  - 5. Vários algoritmos prontos para usar.

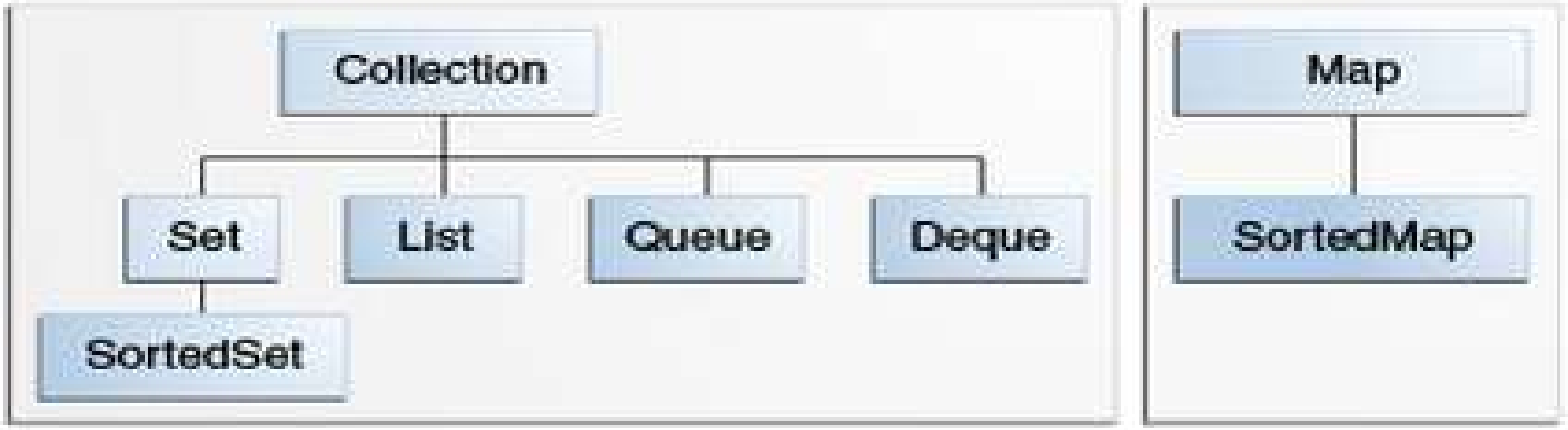
Framework Collections

- Uma **Coleção é uma estrutura de dados** que possibilita agrupar outros objetos, tendo por objetivo: adicionar, remover e pesquisar um determinado objeto dentro da coleção.



Arquitetura

- O ***Framework Collections*** é uma arquitetura baseada na hierarquia de interfaces abaixo:

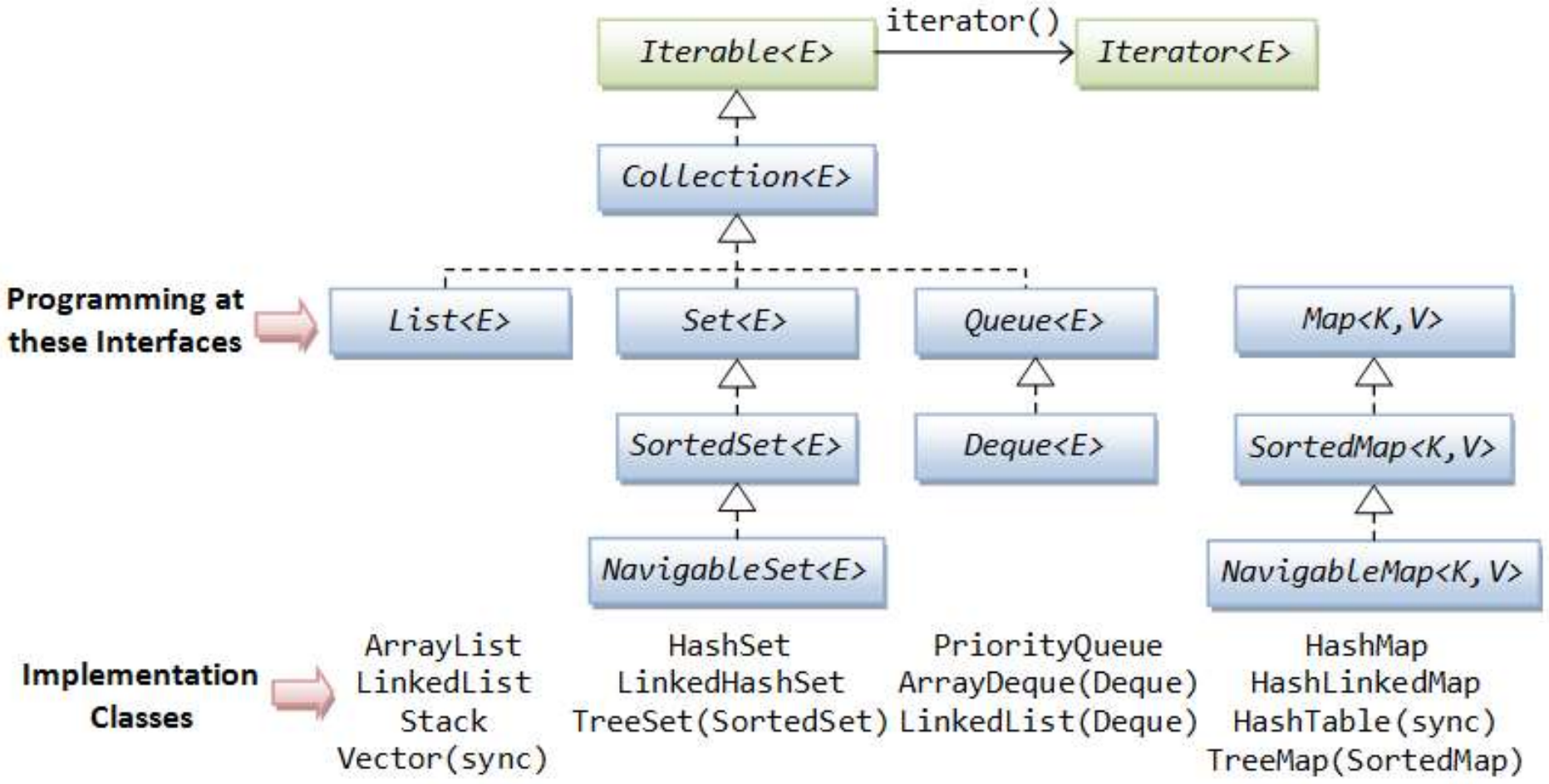


- As **Interfaces** acima listadas diferem entre si na forma de organizar os dados, nas implementações das operações de adição, remoção e pesquisa, bem como na eficiência da realização dessas operações.

- O ***Framework Collections*** é definido no pacote **java.util**.

Arquitetura

- Diagrama de Classes do ***Framework Collections.***



© Chua Hock-Chuan

Interfaces

`public interface Collection<E>`

É o elemento raiz de toda hierarquia de coleções. Define um grupo de objetos conhecidos como elementos.

`public interface Set<E>`

Define uma coleção que não pode conter elementos duplicados.

`public interface SortedSet<K,V>`

Define uma coleção do tipo Set com os seus elementos ordenados.

`public interface List<E>`

Define uma coleção de elementos ordenados e que podem ser duplicados. Representa a estrutura de dados **Lista**.

`public interface Queue<E>`

Define uma coleção de elementos que podem ser ordenados no padrão FIFO (*first-in-first-out*). Representa a estrutura de dados **Fila**.

Interfaces

`public interface Deque<E>`

Define uma coleção de elementos ordenados no padrão FIFO (*first-in-first-out*). Representa a estrutura de dados **Fila duplamente encadeada**.

`public interface Map<K,V>`

Cada objeto da coleção tem uma chave (*key*) associada. Não permite elementos duplicados.

`public interface SortedMap<E>`

Define uma coleção do tipo Map cujos elementos são ordenados pela chave.

Resumo:

- As interfaces *Collection, Map, Set, List, Queue e Deque* são utilizadas para armazenar um conjunto de objetos de forma não ordenada. Já as interfaces *SortedSet e SortedMap* são utilizadas para armazenar um conjunto de objetos de forma ordenada.

Interface java.util.Collection

- É a Interface base para todos os tipos de coleções. Define os seguintes métodos para a manipulação de objetos das coleções:
  - ***int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element)*** e ***Iterator<E> iterator().***

- Além desses métodos, define os seguintes métodos para a manipulação de coleções inteiras:
  - ***boolean containsAll(Collection<?> c),***
    boolean addAll(Collection<? extends E> c),
    boolean removeAll(Collection<?> c),
    boolean retainAll(Collection<?> c) e ***void clear().***

- Com o advento do Java 8, foram criados mais dois métodos nesta interface: ***Stream<E> stream()*** e ***Stream<E> parallelStream().***

Novos Métodos definidos no Java 8

- **java.lang.Iterable**
  - default void forEach(Consumer<? super T> action)
  - default Spliterator<T> spliterator()
- **java.util.Collection**
  - default boolean removeIf(Predicate<? super E> filter)
  - default Spliterator<E> spliterator()
  - default Stream<E> stream()
  - default Stream<E> parallelStream()
- **java.util.Map**
  - default V getOrDefault(Object key, V defaultValue)
  - putIfAbsent(K key, V value)
  - .... etc

Exercício

- 1) [FCC - 2009 – TJ/SE] Uma lista Java é uma coleção ordenada de elementos do mesmo tipo, conhecida por sequência. Os elementos de uma lista podem ser acessados pela sua posição, isto é, seu índice e são derivados da interface:

  a) java.util.LinkedList, que estende a interface Collection.

  b) java.util.Collection, que estende a interface Set.

  c) java.util.Set, que estende a interface Collection.

  d) java.util.Collection, que estende a interface List.

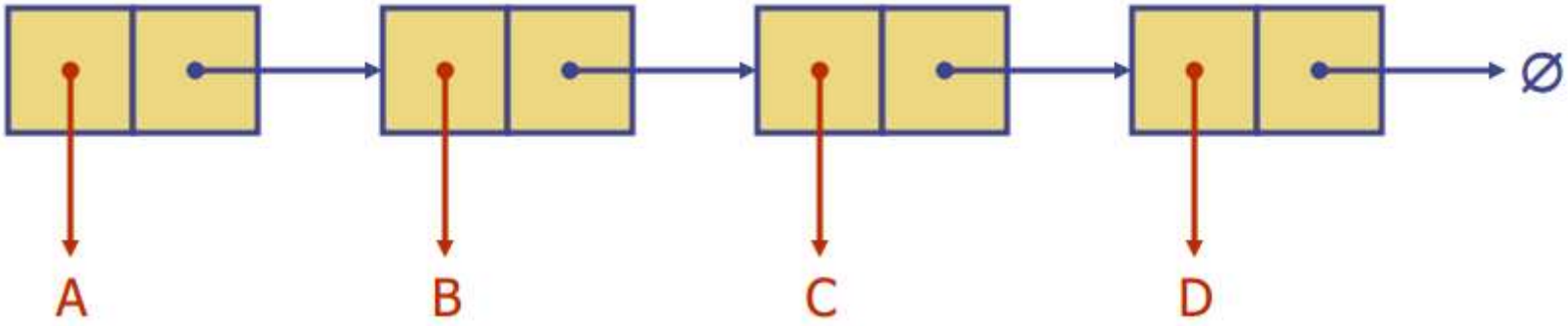
  e) java.util.List, que estende a interface Collection.

Lista, Pilha e Fila

Revisão de Estrutura de Dados

LISTA

- É uma estrutura de dados que consiste em uma coleção de nós dispostos linearmente, onde cada elemento tem um antecessor (exceto o primeiro) e um sucessor (exceto o último).

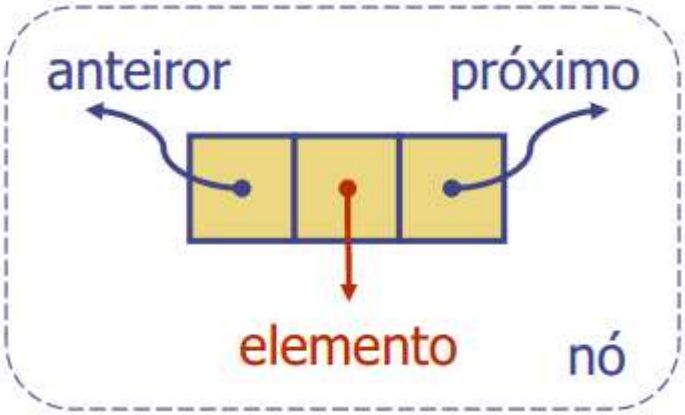


- Pode ser implementada como Vetor (array) ou como **Lista Simplesmente Encadeada** (cada nó conhece a posição do nó seguinte) ou **Lista Duplamente Encadeada** (cada nó conhece a posição do nó anterior (prev) e do nó seguinte (next)).

Revisão de Estrutura de Dados

LISTA

- Exemplo de um Elemento de uma Lista Duplamente Encadeada.



- A Lista pode ser mantida ordenada ou não.

- As operações mais importantes de uma coleção do tipo Lista são:
  - Adicionar elementos
  - Remover elementos
  - Pesquisar elementos

Revisão de Estrutura de Dados

PRINCIPAIS IMPLEMENTAÇÕES DE LISTA EM JAVA

- **PILHA**
  - Adota a política LIFO (*last-in-first-out*) para manipular os elementos da Lista.
  - Realiza as operações de Adição e Remoção no início da Lista.
  - Principal implementação:

```
public class Stack<E> extends Vector<E>
```

- **FILA SIMPLESMENTE ENCADEADA**
  - Adota a política FIFO (*first-in-first-out*) para manipular os elementos da Lista.
  - Realiza as operações de Adição no fim da Lista e de Remoção no início da Lista.
  - Principal implementação:

```
public interface Queue<T>
```

Revisão de Estrutura de Dados

PRINCIPAIS IMPLEMENTAÇÕES DE LISTA EM JAVA

- **FILA DUPLAMENTE ENCADEADA**
  - Adota a política FIFO (*first-in-first-out*) para manipular os elementos da Lista.
  - Realiza as operações de Adição no fim da Lista e de Remoção no início da Lista.
  - Principal implementação:

public interface **Deque<T>**

Exercícios

- 1) [ESAF - 2012 - Receita Federal] Assinale a opção correta.

a) Uma fila é um tipo de lista linear em que todas as categorias são inseridas em um extremo, ficando as classes restritas ao outro extremo.

b) Uma pilha é um tipo de lista linear em que todas as operações de inserção e remoção são realizadas numa mesma extremidade.

c) Uma fila é um tipo de lista colinear em que inserções parametrizadas são realizadas no mesmo extremo que as remoções.

d) Uma pilha é um tipo de lista encadeada em que todas as operações de inserção e retrieve são realizadas na extremidade mais próxima.

e) Uma pilha é um fila linear em que todas as operações de carry e stand são realizadas numa mesma extremidade.

Exercícios

- 2) [FCC - 2012 – TST] As pilhas e as filas são estruturas de dados essenciais para os sistemas computacionais. É correto afirmar que:

a) A fila é conhecida como lista LIFO - Last In First Out.

b) A política de atendimento aos processos por um único processador, implementada por fila circular, seria adequada para controlar a fila de arquivos a serem impressos em uma impressora.

c) A pilha é conhecida como lista FIFO - First In First Out.

d) Uma política de acesso dos processos ao processador por tempo compartilhado é implementada por uma pilha.

e) A pilha pode ser utilizada para armazenar informações sobre as sub-rotinas (funções) ativas em um programa de computador em execução.

Interface Set

Interface java.util.Set

- É a Interface que define uma coleção que não contém objetos duplicados, isto é, não é permitida a adição de um objeto idêntico a outro já existente na coleção. Também é conhecida como Conjunto!

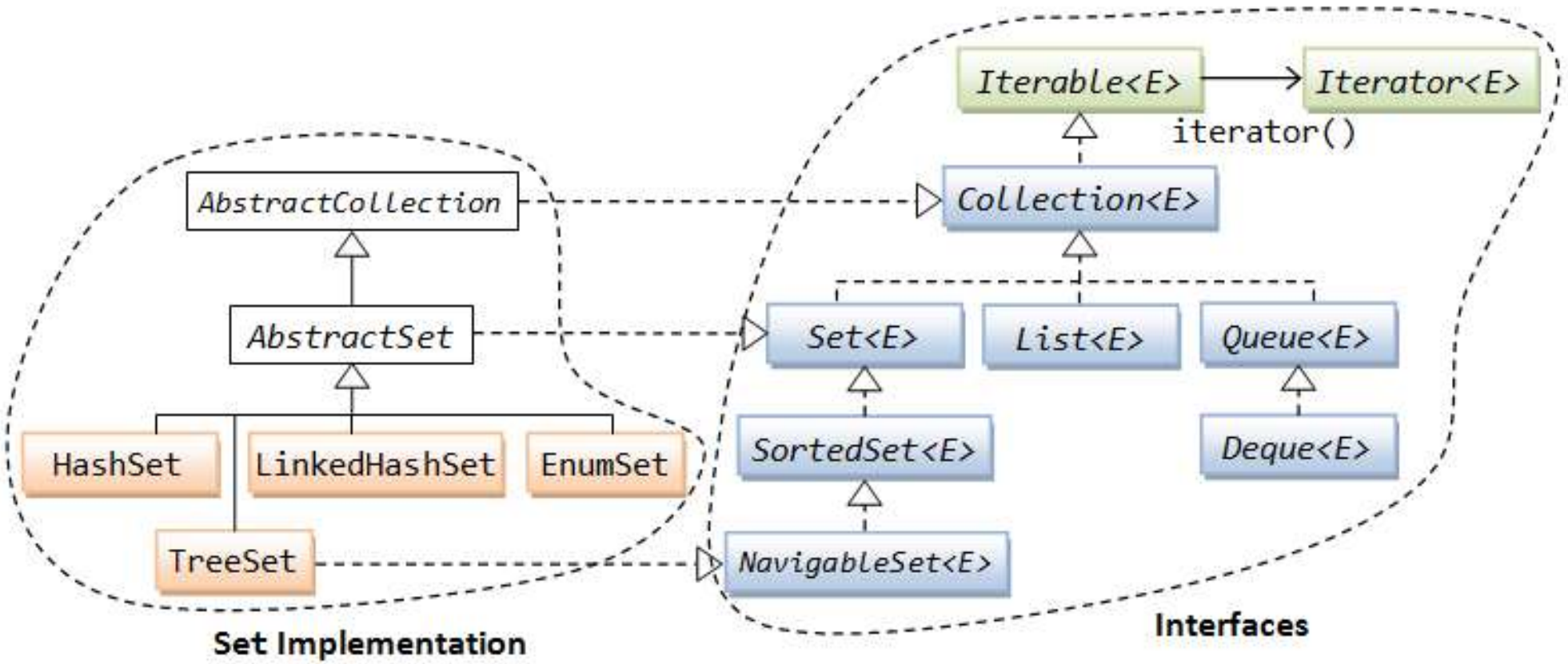
- Não é garantida a ordenação dos objetos, portanto não é possível indexar os elementos por índices numéricos.

- Os métodos definidos em *Collection* são herdados por esta interface.

- A seguir, são apresentados exemplos de utilização de quatro métodos herdados em dois conjuntos Set (s1 e s2):
  - ***s1.containsAll(s2):*** retorna '*true*' se s2 for um subconjunto de s1.
  - ***s1.addAll(s2):*** transforma s1 em uma união de s1 e s2.
  - ***s1.retainAll(s2):*** transforma s1 em uma interseção de s1 e s2.
  - ***s1.removeAll(s2):*** transforma s1 na diferença entre s1 e s2.

Interface java.util.Set

- Diagrama de Classes da ***Interface Set.***



© Chua Hock-Chuan

Interface java.util.Set

- O Java fornece três implementações da Interface Set:
  - **HashSet**
  - **TreeSet**
  - **LinkedHashSet**

- A **classe HashSet** utiliza uma tabela *hash* para guardar os seus elementos, sem garantir a ordem de iteração, nem que a mesma permanecerá constante com o tempo. Por utilizar o algoritmo de tabela *hash*, o acesso é rápido, tanto para leitura quanto para modificação.

- A **classe LinkedHashSet** é uma subclasse de Hashset que adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, garante a ordem com que os elementos presentes no conjunto são recuperados.

Interface java.util.Set

- A **classe TreeSet** oferece um conjunto ordenado de elementos por intermédio de árvore balanceada *red-black*.

Sintaxe Padrão

```
Collection semDuplicacao = new HashSet(comDuplicacao);
```

```
Collection semDuplicacao = new LinkedHashSet(comDuplicacao);
```

```
Collection semDuplicacao = new TreeSet(comDuplicacao);
```

Classe HashSet

```
import java.util.*;
public class HashSetApp {
  public static void main(String args[])
  {
   HashSet<String> lista = new HashSet<>();
   lista.add("vermelho");
   lista.add("verde");
   lista.add("verde");
   lista.add("amarelo");
   System.out.println(lista);
  }
}
```

```
Saída: [amarelo, vermelho, verde]
```

Classe LinkedHashSet

```
import java.util.*;
public class LinkedHashSetApp {
  public static void main(String args[])
  {
   LinkedHashSet<String> lista =
                              new LinkedHashSet<>();
   lista.add("vermelho");
   lista.add("verde");
   lista.add("verde");
   lista.add("amarelo");
   System.out.println(lista);
  }
}
```

```
Saída: [vermelho, verde, amarelo]
```

Classe TreeSet

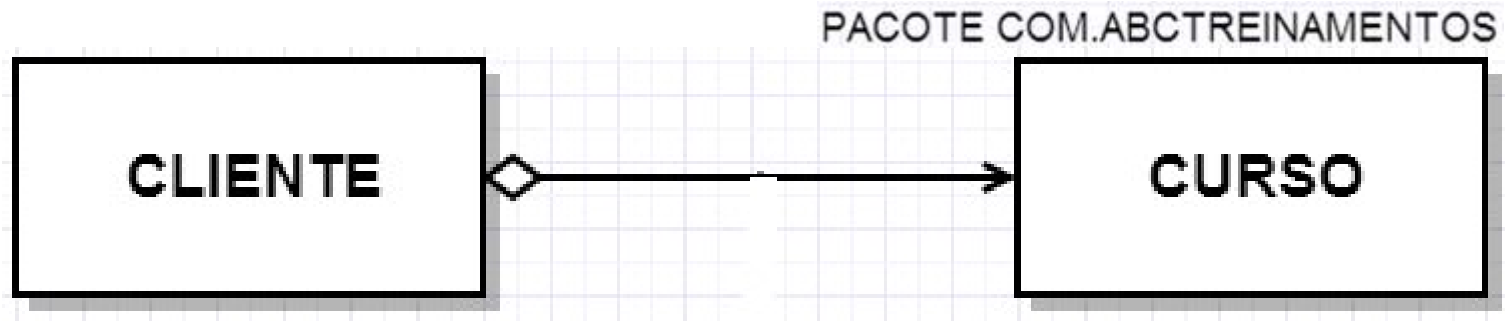
```
import java.util.*;
public class TreeSetApp {
  public static void main(String args[])
  {
   TreeSet<String> lista =
                              new TreeSet<>();
   lista.add("vermelho");
   lista.add("verde");
   lista.add("verde");
   lista.add("amarelo");
   System.out.println(lista);
  }
}
```

```
Saída: [amarelo, verde, vermelho]
```

Exercícios

- 1) Implementar as classes **HashSetApp**, **LinkedHashSetApp** e **TreeSetApp**.

- 2) Escreva uma aplicação em Java que represente o diagrama de classes abaixo:



- Obs: um cliente possui um conjunto de cursos não repetidos.

- 3) No exercício anterior, criar pelos menos dois objetos clientes (**A** e **B**). Informar quais foram os mesmos cursos feitos por **A** e **B**; quais os cursos feitos por **A** que **B** não fez; e, quais os cursos feitos por **B** que **A** não fez.

Interface List

Interface java.util.List

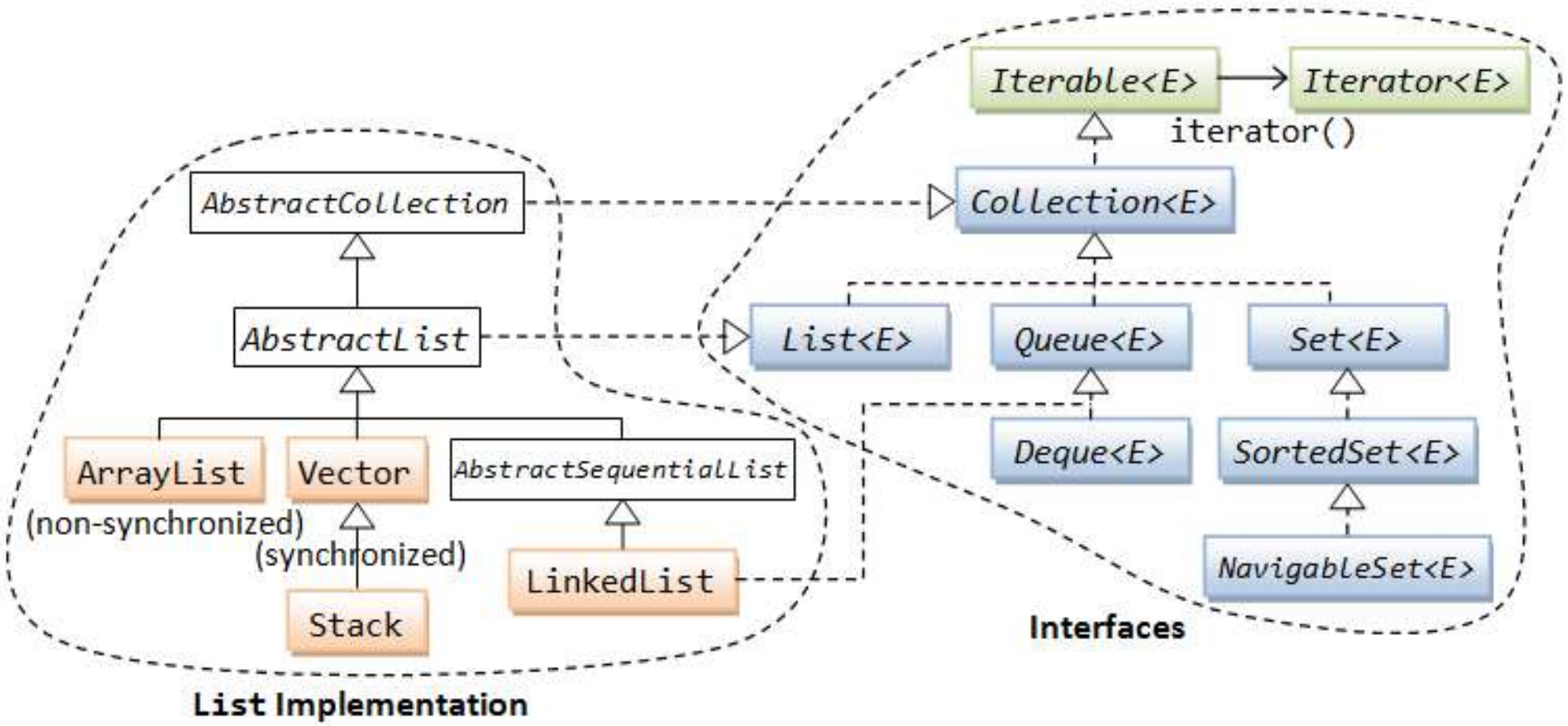
- É a Interface que define uma coleção de elementos ordenados (que podem estar duplicados) e cujo acesso é realizado por meio de um índice numérico que representa a posição de cada elemento.

- Os métodos definidos em *Collection* são herdados por esta interface.

- O Java fornece duas implementações da Interface List:
  - **ArrayList**
  - **LinkedList**

Interface java.util.List

- Diagrama de Classes da ***Interface List.***



© Chua Hock-Chuan

Classe ArrayList

- A **classe ArrayList** utiliza internamente um vetor (array) de objetos, cujo tamanho inicial é de 10 posições. Caso não seja suficiente, um novo vetor é alocado com tamanho igual a 1.5 vezes maior e todo o conteúdo é copiado para este novo vetor.

- Esta implementação é preferível 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).

Classe ArrayList

```
import java.util.*;
public class ArrayListApp {
    public static void main(String args[])
    {
      ArrayList<String> lista = new ArrayList<>();
      lista.add("vermelho");
      lista.add("verde");
      lista.add("verde");
      lista.add("amarelo");
      System.out.println(lista);
    }
}
```

```
Saída: [vermelho, verde, verde, amarelo]
```

Classe ArrayList - Ordenado

```
import java.util.*;
public class ArrayListApp {
    public static void main(String args[])
    {
      ArrayList<String> lista = new ArrayList<>();
         lista.add("vermelho");
         lista.add("verde");
         lista.add("verde");
         lista.add("amarelo");
         Collections.sort(lista);
         System.out.println(lista);
       }
}
```

```
Saída: [amarelo, verde, verde, vermelho]
```

Classe LinkedList

- A **classe LinkedList** utiliza internamente uma lista duplamente encadeada e a busca pelos seus elementos é feita de forma sequencial (via padrão Iterator) ou nas extremidades, e não de forma aleatória (por índices).

- Um exemplo de uso é como um fila (FIFO), onde os elementos são retirados da lista na mesma sequência em que são adicionados.

Classe LinkedList

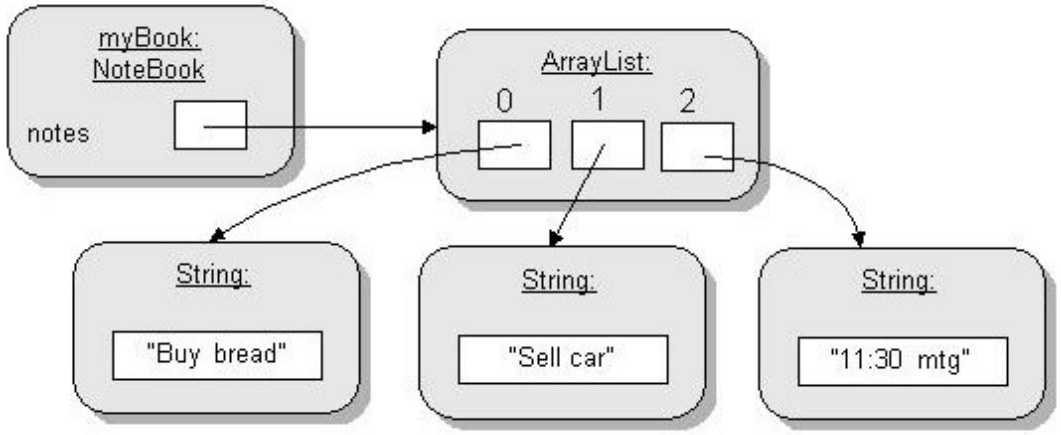
```
import java.util.*;
public class LinkedListApp {
  public static void main(String args[])
  {
    LinkedList<String> lista = new LinkedList<>();
    lista.add("vermelho");
    lista.add("verde");
    lista.add("verde");
    lista.add("amarelo");
    System.out.println(lista);
    lista.removeFirst();
    lista.removeLast();
    System.out.println(lista);
  }
}
```

```
Última Saída: [verde, verde]
```

Exercícios

- 1) Implementar as classes **ArrayListApp** e **LinkedListApp**.

- 2) Criar uma Aplicação para representar a estrutura de dados abaixo.



- 3) Criar uma Aplicação para inserir 10 mil elementos em um ArrayList e imprimir os seus valores.

- 4) Calcular o tempo gasto para realizar a operação acima.
(utilizar **System.currentTimeMillis()** para cronometrar o tempo gasto)

Interfaces Queue e Dequeue

Interface java.util.Queue

- É a Interface que define uma coleção de elementos cujo acesso se dá por meio de uma **Lista Simplesmente Encadeada**.

- A adição de elementos é feita no fim da Fila e a remoção no início da Fila.

- Os métodos definidos em *Collection* são herdados por esta interface.

- A seguir, são apresentados os seus principais métodos:

  - ***add(e), remove(), element(), offer(e), poll(), peek()***

Interface java.util.Dequeue

- É a Interface que define uma coleção de elementos cujo acesso se dá por meio de uma **Lista Duplamente Encadeada**.

- A adição de elementos pode ser feita no início ou no fim da Fila e a remoção no início ou fim da Fila.

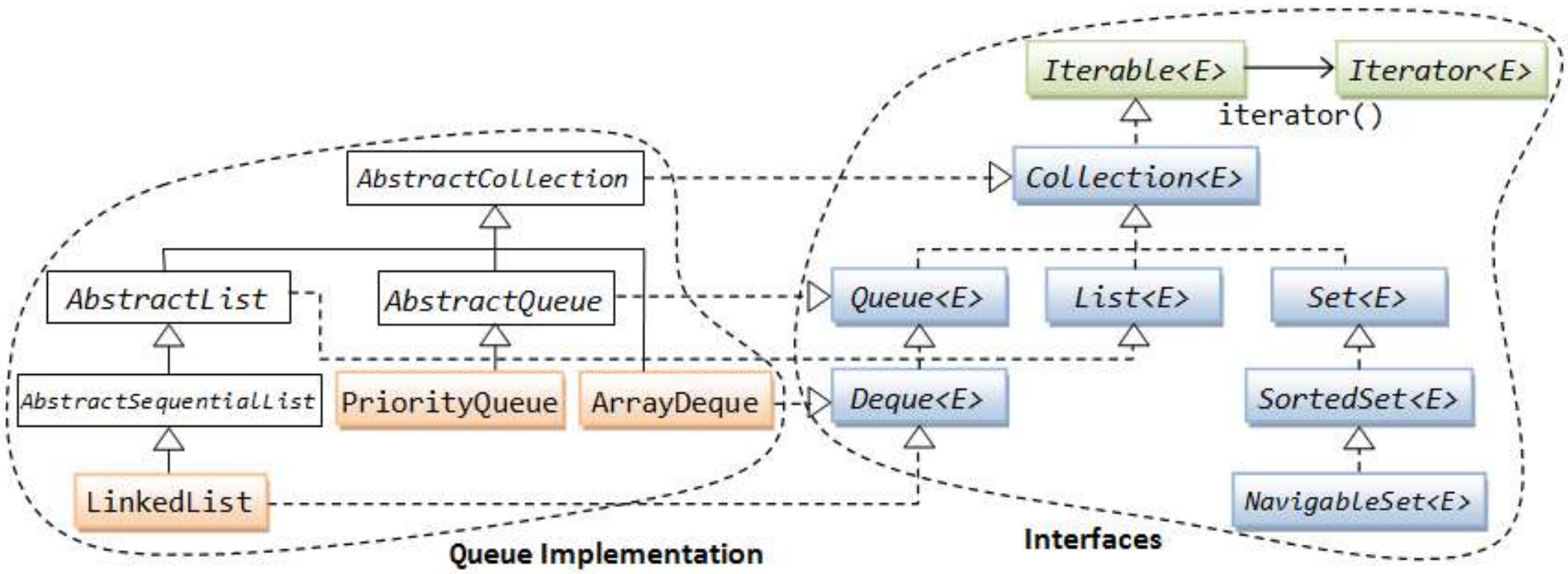
- Os métodos definidos em *Collection* são herdados por esta interface.

- A seguir, são apresentados os seus principais métodos:
  - ***addFirst(e), offerFirst(e), addLast(e) offerLast(e)***
  - ***removeFirst(), pollFirst(), getFirst(), peekFirst()***
  - ***removeLast(), pollLast(), getLast(), peekLast()***

- O Java fornece a implementação desta interface com a classe **ArrayDeque**.

Interface java.util.Dequeue

- Diagrama de Classes das ***Interface Queue e Dequeue.***



© Chua Hock-Chuan

Classe ArrayDeque

```
import java.util.*;
public class DequeApp {
  public static void main(String args[])
  {
    Deque<String> lista = new ArrayDeque<>();
    lista.add("vermelho");
    lista.add("verde");
    lista.add("verde");
    lista.add("amarelo");
    System.out.println(lista);
    lista.removeFirst();
    lista.removeLast();
    System.out.println(lista);
  }
}
```

```
Última Saída: [verde, verde]
```

Exercícios

- 1) Implementar a classe **DequeApp**.

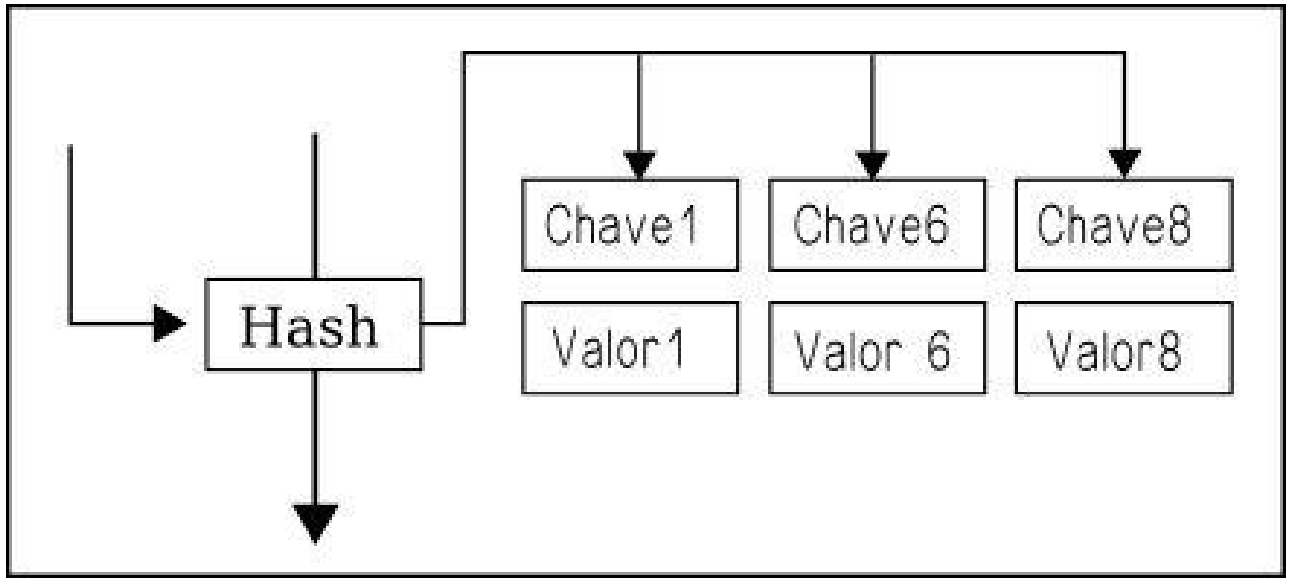
- 2) Aplicar os métodos addFisrt(e), addLast(e), peekFirst(), peekLast().

Interface Map

Interface java.util.Map

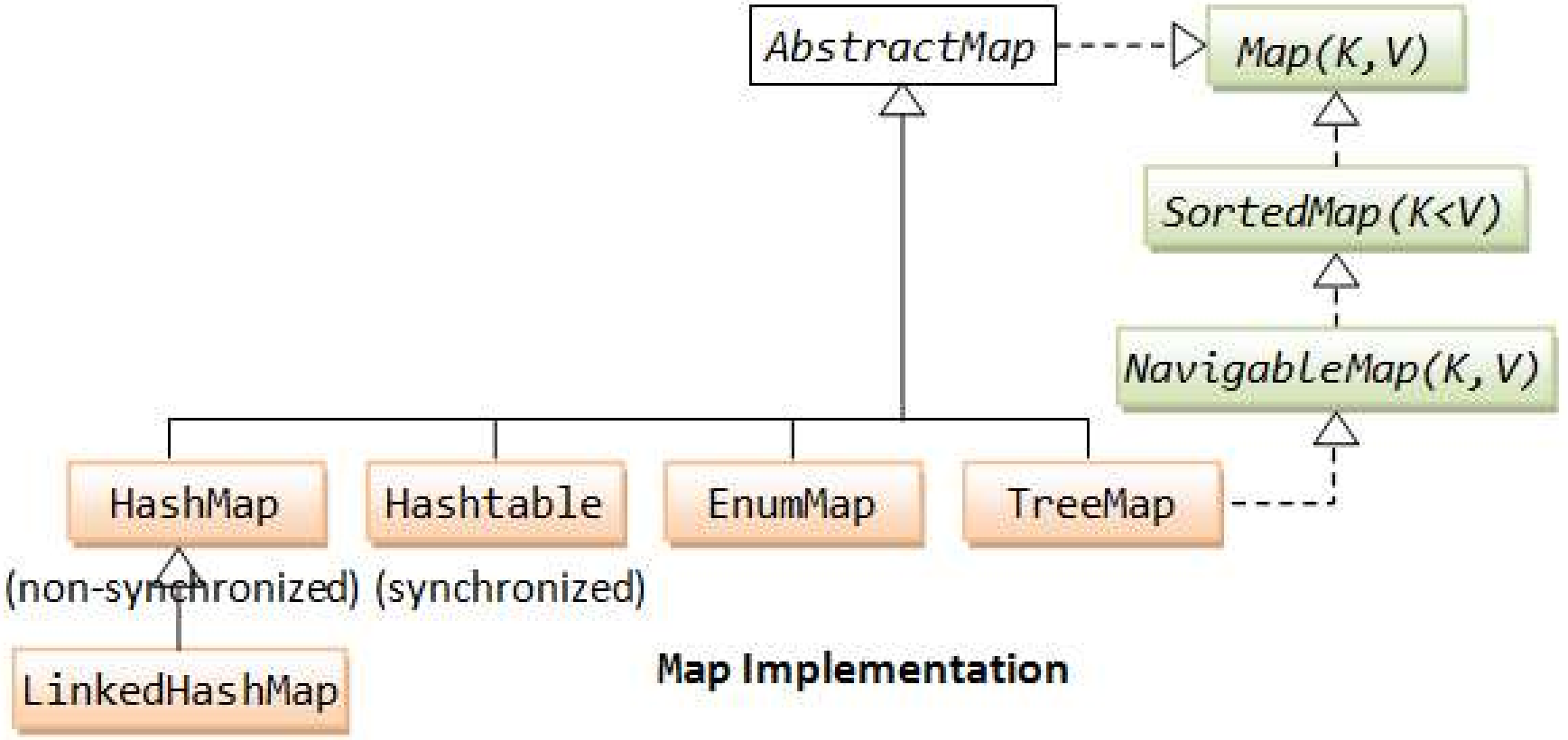
- É a Interface base para todos os tipos de coleções baseados em Mapa.

- Um Mapa (também chamados de vetores associativos) é utilizado para organizar coleções de objetos, cujos índices de acesso não precisam necessariamente ser valores inteiros positivos seqüenciais.



- Acima uma implementação de **Map** (**HashMap**) baseada no uso de chaves (*keys*) geradas a partir de uma **Tabela Hash**.

Interface java.util.Map



Interface java.util.Map

- É a Interface base para todos os tipos de coleções baseados em Mapa.

- Um Mapa (também chamados de vetores associativos) é utilizado para organizar coleções de objetos, cujos índices de acesso não precisam necessariamente ser valores inteiros positivos seqüenciais.

- Um Mapa armazena pares (chave, valor) chamados itens. Chaves e valores podem ser de qualquer tipo.

- A chave é utilizada para achar um elemento rapidamente. Estruturas especiais (tais como **Tabelas Hashs**) são utilizadas para que a pesquisa seja rápida.

- O Mapa pode ser mantido ordenado ou não (com respeito às chaves).

Interface java.util.Map

- **TABELA HASH**
  - É uma estrutura de dados especial, que associa chaves de pesquisa a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.

http://pt.slideshare.net/CriatividadeZeroDocs/aula-sobre-tabela-hash

Interface java.util.Map

- Os seus principais métodos são:
  - ***put(), get(), remove(), containsKey()***
  - ***containsValue(), size(), empty()***
  - ***putAll(), clear()***
  - ***keySet(), entrySet(), values()***

- O Java fornece três implementações da Interface Map:
  - **HashMap**
  - **LinkedHashMap**
  - **TreeMap**

Classe HashMap

- A **classe HashMap** utiliza internamente tabela hash para armazenar as chaves de um Mapa.
- O tempo de acesso aos seus elementos (leitura e escrita) é muito bom.
- Não garante a ordem dos elementos que foram adicionados à coleção.
- Permite chaves e valores nulos.
- Não é sincronizado, isto é, em um ambiente multithread a sua atualização é um problema.

Classe HashMap

```
import java.util.*;
public class HashMapApp {
   public static void main(String[] args) {
       HashMap<Integer, String> mapa =
              new HashMap<>();
    mapa.put(1, "Bruno");
    mapa.put(2, "Umberto");
    mapa.put(3, "Fernando");
    mapa.put(3, "Caetano");
    mapa.put(4, "Umberto");
     System.out.println(mapa);
   }
}
```

```
{1=Bruno, 2=Umberto, 3=Caetano, 4=Umberto}
```

Classe LinkedHashMap

- A classe **LinkedHashMap** é subclasse de **HashMap** e adiciona previsibilidade à ordem de iteração sobre os elementos, isto porque mantém ordenados os elementos por ordem de inserção.

- Permite chaves e valores nulos.

- Não é sincronizado.

Classe LinkedHashMap

```
import java.util.*;
public class LinkedHashMapApp {
   public static void main(String[] args) {
    Map<Integer, String> mapa =
              new LinkedHashMap<>();
     mapa.put(4, "Bruno");
     mapa.put(2, "Umberto");
     mapa.put(3, "Fernando");
     mapa.put(3, "Caetano");
     mapa.put(1, "Umberto");
      System.out.println(mapa);
   }
}
```

```
{4=Bruno, 2=Umberto, 3=Caetano, 1=Umberto}
```

Interface java.util.SortedMap

- É a Interface que estende Map, adicionando a semântica de ordenação natural dos seus elementos, de forma análoga à interface SortedSet.

- Esta interface herda os métodos de Map e oferece os novos métodos abaixo:

  - comparator(), subMap(K, K);

  - headMap(K), tailMap(K);

  - firstKey(), lastKey();

- A classe **TreeMap** implementa esta interface.

Classe TreeMap

- A classe **TreeMap** utiliza internamente uma ***árvore red-black*** (árvores binárias semibalanceadas) que garante que as chaves contidas serão mantidas em ordem ascendente.

- Deve ser utilizada quando for necessário um Mapa ordenado, mesmo após a realização de operações de inserção e remoção.

- Não permite chaves nulas e não é sincronizado.

Classe TreeMap

```
import java.util.*;
public class TreeMapApp {
    public static void main(String[] args)
    {
       Map<Integer, String> mapa = new TreeMap<>();
       mapa.put(4, "Bruno");
       mapa.put(2, "Umberto");
       mapa.put(3, "Fernando");
       mapa.put(3, "Caetano");
       mapa.put(1, "Umberto");
        System.out.println(mapa);
    }
}
```

```
Saída: {1=Umberto, 2=Umberto, 3=Caetano, 4=Bruno}
```

Exercícios

- 1) Implemente as classes **HashMapApp**, **LinkedHashMapApp** e **TreeMapApp**.

- 2) Escreva uma aplicação em Java que represente o diagrama de classes abaixo:

Obs: a classe LojaVirtual terá um atributo estático **pagamento** que irá armazenar um Mapa de valores <Cliente, List<Curso>>.

Interfaces Auxiliares e Classes Utilitárias

Interfaces Auxiliares e Classes Utilitárias

- **INTERFACES AUXILIARES**
  - O ***framework Collections*** também define uma série de interfaces auxiliares, que definem operações sobre os objetos retornados de uma coleção: **`Iterator, ListIterator, Comparable, Comparator, Enumeration e RandomAccess.`**

- **CLASSES UTILITÁRIAS**
  - São aquelas que possuem apenas métodos estáticos e são utilizadas pelas classes definidas no framework Collections. Os principais exemplos são: **`Collections, Arrays e Properties.`**

Classes Properties

- É uma classe que implementa a interface Map e muito útil para trabalhar com documentos XML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE properties SYSTEM
         "http://java.sun.com/dtd/properties.dtd">
<properties>
   <entry key='1'>http://www.abctreinamentos.br</entry>
   <entry key='2'>http://www.uol.com.br</entry>
   <entry key='3'>http://www.oglobo.com.br</entry>
</properties>
```

sites.xml

```
public static void main(String[] args) throws Exception
  {
    Properties sites = new Properties();
    sites.loadFromXML(new FileInputStream("sites.xml"));
    System.out.println(sites.getProperty("1"));
  }
```

Coleções x Vetores

Vantagens de Coleções

- Solução flexível para armazenar objetos. **A quantidade armazenada de objetos não é fixa**, como ocorre com vetores;

- Poucas interfaces permitem maior reuso.

Desvantagens de Coleções

- Não aceitam tipos primitivos (só empacotados). **Vetores aceitam**;

- O **acesso a um elemento de um vetor é mais rápido** de que o acesso a um elemento de Coleções.

Exercício

- 1) Criar o arquivo ‘sites.xml’ para ser acessado pela aplicação **PropertiesApp**.

Considerações Finais

Considerações Finais

- Deve-se selecionar a implementação da classe apenas no momento de instanciação das coleções, usando variáveis do tipo de suas interfaces:

```
ArrayList lista = new ArrayList();
// não recomendado - A notação a seguir é mais flexível
List lista = new ArrayList();
```

- Fazendo o uso do tipo da interface, ao invés do tipo concretamente empregado, impede de utilizar métodos específicos de implementação, permitindo que as mudanças nesta ou na eventual substituição do tipo concreto tenham impacto mínimo na aplicação desenvolvida.

Dica: Programe voltado à interface, não à implementação!

Como escolher entre Coleções?

- Sugestões retiradas do site http://www.dsc.ufcg.edu.br/~jacques/cursos/p2/html/ed/colecoes.htm
- O desenvolvedor possui as seguintes opções quando fizer uso de coleções:
  - **Lista** (java.util.List, java.util.Queue, java.util.Dequeue),
  - **Conjunto** (java.util.Set)
  - **Mapa** (java.util.Map).

- Se for necessário manter elementos duplicados, usar Lista! **Contudo, se for necessário fazer muita pesquisa, evite o uso de Lista!**

- Se não for necessário manter elementos duplicados e nem usar chaves, usar Conjunto! **Contudo, se for necessário o uso de chaves, usar Mapa!**

Como escolher entre Coleções?

- **Qual a implementação de Lista que deverá ser escolhida?**
  - Usar ArrayList se for necessário acessar os elementos por índice (ex. pesquisa binária);
  - Usar LinkedList se for necessário inserir ou remover elementos do meio da Lista com frequência.

- **Qual a implementação de Conjunto que deverá ser escolhida?**
  - Usar HashSet se não for necessário obter um conjunto ordenado;
  - Usar TreeSet se for necessário obter um conjunto ordenado.

- **Qual a implementação de Mapa que deverá ser escolhida?**
  - Usar HashMap se não for necessário obter um mapa ordenado;
  - Usar TreeMap se for necessário obter um mapa ordenado.

RESUMO

TÓPICOS APRESENTADOS

- Nesta aula nós estudamos:

  - **Introdução ao Framework Collections**
  - **Lista, Pilha e Fila**
  - **Interface Set**
  - **Interface List**
  - **Interfaces Queue e Dequeue**
  - **Interface Map**
  - **Interfaces Auxiliares e Classes Utilitárias**
  - **Considerações Finais**

ATIVIDADE PARA SE APROFUNDAR

- **1) Criar uma classe Fifa2017 que possui uma lista de Jogadores. Esta última classe é representada pelos atributos (nome, pontuação, clube, país).**

- **A classe Jogador deve implementar a interface Comparable<T>.**
- **Criar o método compareTo(Jogador jogador) para ser possível comparar o jogador pelo seu atributo pontuação.**
- **Apresentar os cinco melhores jogadores do Fifa2017, em ordem decrescente.**