Objetivos

- *Apresentar as principais estruturas de dados disponíveis na API de* **Collections** *em Java*
  - *Conceitos, formas de armazenar objetos, tipos de coleções em Java, métodos utilitários da API Java, ordenação, busca e recuperação de objetos, boas práticas de programação*
- *Usar estruturas de dados em Java*
  - *vetores, listas, árvores, pilhas, filas, tabelas hash, conjuntos, mapas e seus métodos.*
- *Exercícios e vários exemplos de códigos que você deve testar para adquirir novos conhecimentos*
  - *Você* **deve** *codificá-los na IDE de sua preferência*

O que são coleções?

- Representam itens de dados que possuem algum tipo de relação entre si
  - Pasta de correio (coleção de cartas)
  - Diretório do sistema (coleção de arquivos)
  - Lista telefônica (coleção de mapeamentos entre os nomes dos assinantes e seus números de telefones)
- Em Java, uma coleção é, na verdade, um **objeto** que agrupa referências para estes itens de dados
  - A partir daí, podemos armazenar, recuperar e manipular estes itens de dados em nossos programas

Java Collections API

- A plataforma Java possui uma variedade de coleções (presentes no pacote java.util)
  - Um rico conjunto de classes e interfaces onde as mais diversas e importantes **estruturas de dados** já se encontram implementadas e prontas para serem utilizadas no programa, diretamente.
  - Ex.: vetores, conjuntos, pilhas, filas, listas, árvores, tabelas hash, etc.
- Oferecer, ao programador, diferentes formas de colecionar dados com base em fatores como:
  - Eficiência no acesso, busca ou inserção da informação
  - Forma de organização dos dados

Tipos de coleções em Java

- **Vetores**
  - *Mecanismo nativo da linguagem para colecionar tanto valores primitivos como referências para objetos*
  - *Estrutura pré-dimensionada em tempo de compilação*
  - *Forma eficiente e simples de manipular dados*
- **Coleções**
  - *Não oferecem suporte a tipos primitivos (somente se empacotados dentro de objetos – classes wrapper)*
  - *Classes e interfaces do pacote java.util*
  - *Oferece implementações de métodos estáticos para manipulação de coleções e também vetores*

Vetores

- *Ao escolher um vetor, lembre-se de alguns pontos:*
  - *tamanho fixo. É preciso criar um novo vetor e copiar o conteúdo do antigo para o novo. Vetores não podem ser redimensionados depois de criados.*
  - *quantidade máxima de elementos é obtida através da propriedade length (comprimento do vetor)*
  - *verificados em tempo de execução. Tentativa de acessar índice inexistente provoca, na execução, um erro do tipo ArrayIndexOutOfBoundsException*
  - *tipo definido. Pode-se restringir o tipo dos elementos que podem ser armazenados*

Vetores

- *Lembre-se que vetores também são objetos e devem residir no heap de memória*
  - *Pode-se declarar e inicializar um vetor diretamente*

```
public class UsaVetor {
 public static void main(String[] args){

   Livro[] a; // referência do vetor Livro[] é null
   Livro[] b = new Livro[5]; // referências Livro null


   Livro[] c = new Livro[4];

   for (int i = 0; i < c.length; i++) {

     c[i] = new Livro(); // refs. Livro inicializadas

   }

   Livro[] d = {new Livro(), new Livro(), new Livro()};

   a = new Livro[] {new Livro, new Livro};
}
```

Como retornar vetores

- *Como qualquer vetor (mesmo de primitivos) é um objeto, só é possível manipulá-lo via referências*
  - *Atribuir um vetor a uma variável, na verdade, copia a referência do vetor para a variável*

```
int[] vet = intArray; // se intArray for int[]
```

  - *Retornar um vetor através de um método, na verdade, retorna a referência para o vetor*

```
int[] aposta = Sena.getDezenas();
```

```
public static int[] getDezenas() {
   int[] dezenas = new int[6];

   for (int i = 0; i < dezenas.length; i++) {
     dezenas[i] = Math.ceil((Math.random()*50));
   }
   return dezenas; // retorna a referência!
 }
```

Como copiar vetores

- *Método utilitário de java.lang.System*

```
static void arraycopy(origem_da_copia, offset,
                      destino_da_copia, offset,
                      num_elementos_a_copiar)
```

  - *Exemplo:*

```
int[] um = {12, 22, 3};
int[] dois= {9, 8, 7, 6, 5};
System.arraycopy( um, 0, dois, 1, 2 );
```

  - *Resultado:* `dois {9, 12, 22, 6, 5};`

- *Vetores de objetos*
  - *Apenas as referências são copiadas*

java.util.Arrays

- O pacote contém uma classe utilitária com vários métodos estáticos para manipulação de vetores
- Os métodos suportam vetores de quaisquer tipo
- Principais métodos (sobrecarregados p/ vários tipos):
  - void Arrays.sort()
    - Usa QuickSort para tipos primitivos
    - Para objetos, a classe do objeto deve implementar a interface Comparable
  - boolean Arrays.equals(vetor1, vetor2)
  - int Arrays.binarySearch(vetor, chave)
  - void Arrays.fill(vetor, dado)

Ordenação

- *Para ordenar objetos, é preciso compará-los*
  - *Como estabelecer os critérios de comparação, já que* `equals()` *(Object) informa apenas se um objeto é igual a outro, mas não informa se "é maior" ou "menor" ?*
- *Solução: interface java.lang.Comparable*
  - *Método a implementar:*

```
public int compareTo(Object obj);
```

- *Para implementar, retorne:*
  - *um inteiro menor que zero, se objeto atual for "menor" que o recebido como parâmetro*
  - *um inteiro maior que zero, se objeto atual for "maior" que o recebido como parâmetro*
  - *zero, se objetos forem iguais*

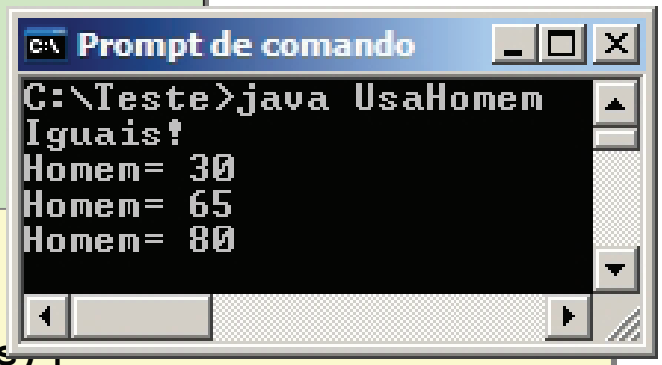
Exemplo

```
public class Homem implements Comparable {
 private int idade;

 public Homem(int idade){
    this.idade = idade;
 }
 public int compareTo(Object obj){
      Homem h = (Homem)obj;
      if ( this.idade > h.idade ) return 1;
      else if ( this.idade < h.idade )return -1;
      else return 0;
 }
 public String toString(){
   return "Homem= " + idade;
 }
}
```

Estude o código do método compareTo

Vetor ordenado!




```
import java.util.Arrays;
public class UsaHomem {
 public static void main(String[] args){
   Homem h1 = new Homem(30); Homem h2 = new Homem(30);
   if(h1.compareTo(h2)== 0) System.out.print("Iguais");
   Homem[] vh = { new Homem(80), new Homem(65), h1 };
   Arrays.sort(vh); // usa compareTo para ordenar vetor
   for ( Homem homem : vh ) System.out.println(homem);
 }
}
```

Novo "for" (Java 5)

Comparator

- *Viu-se que Comparable exige que a classe do objeto a ser comparado implemente a interface*
  - *O que fazer se você não tem acesso para modificar ou mesmo estender a classe?*
  - *E se você deseja que uma classe possa ser comparada por outros diferentes critérios (altura, peso, etc.)*
- *Solução: interface java.util.Comparator*
  - *Crie uma classe que implemente a interface acima e passe-a como segundo parâmetro de Arrays.sort()*
- *Método a implementar:*

```
public int compare(Object o1, Object o2);
```

Não confunda Comparable e Comparator!

- Ao projetar classes novas, considere implementar *java.lang.Comparable*
  - Objetos poderão ser ordenados mais facilmente
  - Critério de ordenação faz parte do objeto
  - *compareTo()* compara objeto atual com um outro
- *java.util.Comparator* não faz parte do objeto comparado
  - Implementação de Comparator é uma classe utilitária
  - Use quando objetos não forem Comparable ou quando quiser adicionar outros critérios de ordenação
  - *compare()* compara dois objetos recebidos

Outras funções úteis de Arrays

`boolean equals(vetor1, vetor2)`

- *Retorna true apenas se vetores tiverem o mesmo conteúdo (mesmas referências) na mesma ordem*
- *Só vale para comparar vetores do mesmo tipo primitivo ou vetores de objetos*

`void fill(vetor, valor)`

- *Preenche o vetor (ou parte do vetor) com o valor passado como argumento (tipo deve ser compatível)*

`int binarySearch(vetor, valor)`

- *Retorna inteiro com posição do valor no vetor ou valor negativo com a posição onde deveria estar*
- *Não funciona se o vetor não estiver ordenado*
- *Se houver valores duplicados não garante qual irá ser localizado*

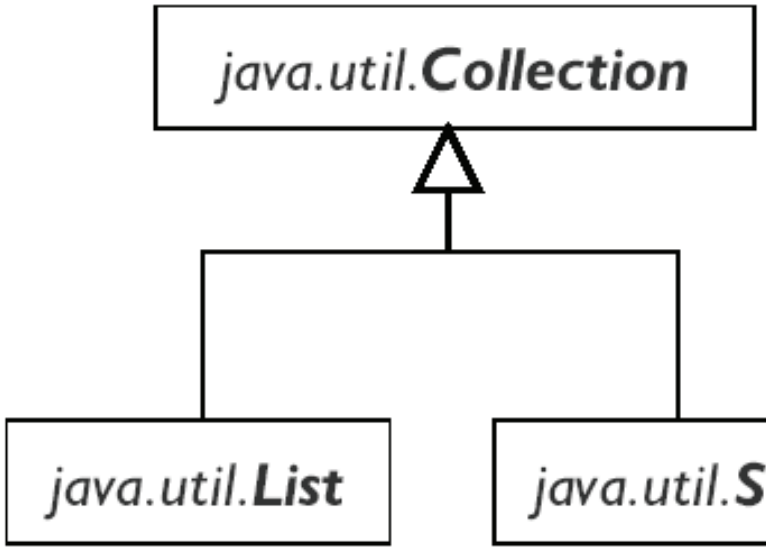
Coleções

- Classes e interfaces do pacote *java.util* que representam listas, conjuntos e mapas
  - Solução *flexível* para armazenar *objetos*
  - Quantidade de objetos armazenada nas coleções não é fixa, como acontece com os vetores
- Poucas interfaces (duas servem de base)
  - *add(), remove()* - métodos de interface *Collection*
  - *put(), get()* - principais métodos de interface *Map**
- Coleções são manipuladas independentemente dos detalhes de implementação

* Não herda diretamente de Collection, mas é considerada como parte do framework

Interfaces

Coleções de elementos individuais



- seqüência definida
- elementos indexados

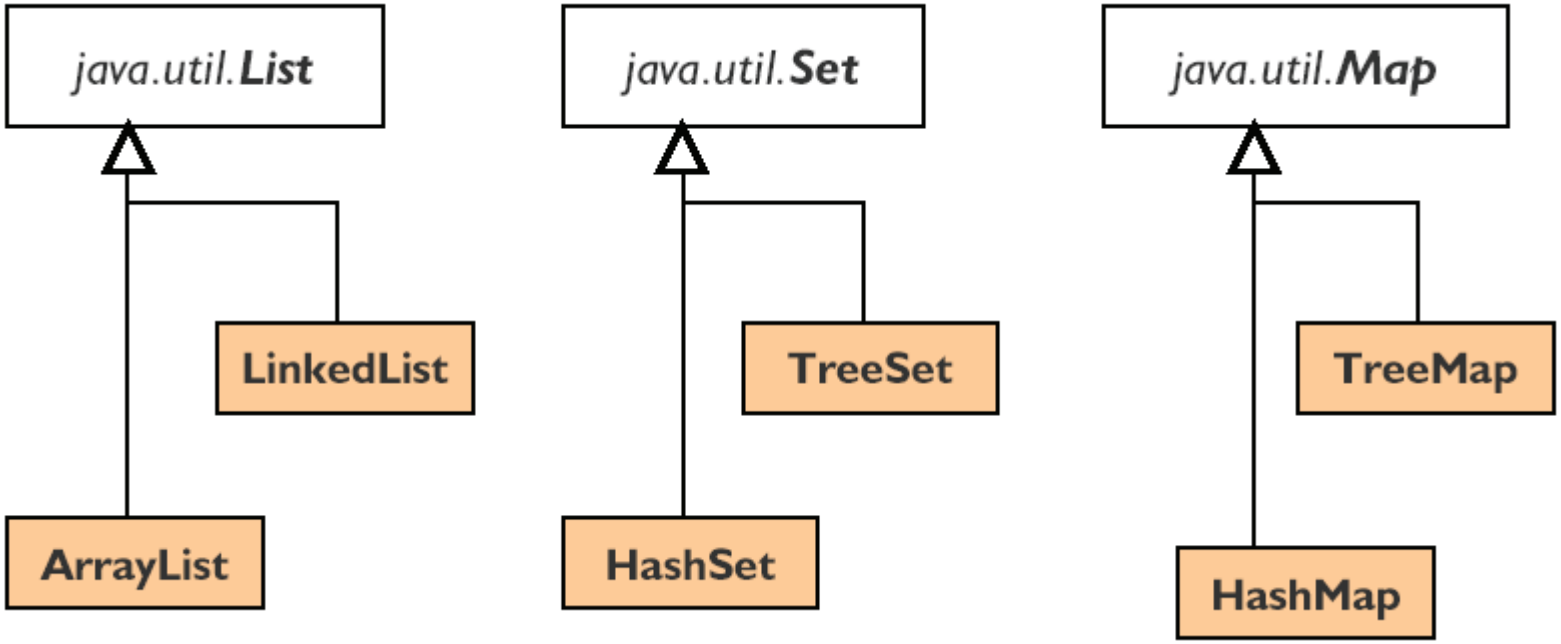
- seqüência arbitrária
- elementos não repetem

Coleções de pares de elementos

java.util.**Map**

- Pares chave/valor (vetor associativo)
- **Collection** de valores (podem repetir)
- **Set** de chaves (unívocas)

Principais implementações concretas



- *Foram omitidos alguns detalhes, como:*
  - *Classes abstratas intermediárias*
  - *Interfaces intermediárias*
  - *Implementações concretas menos utilizadas*

Interface Collection

- *A interface Collection é um supertipo de todas as outras interfaces (exceto as do tipo Map)*
- *Através de Collections, torna-se possível aplicar alguns dos principais conceitos de orientação a objetos:* **o polimorfismo**
- *Nesta interface, são disponibilizados métodos de acesso aos dados de forma genérica*
  - *Isso permite utilizar a estrutura de dados sem tratá-la de maneira particular (lista, vetores ou conjuntos, por exemplo)*

Interface Collection

- Principais sub-interfaces
  - List
  - Set
- Principais métodos (herdados por todas as subclasses)
  - boolean add(E o): adiciona objeto na coleção
  - boolean contains(Object o)
  - boolean isEmpty()
  - Iterator iterator(): retorna iterator
  - boolean remove(Object o)
  - int size(): retorna o número de elementos
  - Object[] toArray(T[]a): converte coleção em Array

Interface List

- *A interface List é um supertipo para as estruturas de dados que garantem a ordem dos dados inseridos*
- *As estruturas de listas permitem que sejam adicionados elementos duplicados (ou cópias)*
- *As estruturas de dados mais importantes que implementam esta interface: Vector, ArrayList e LinkedList*
- *Além das operações definidas em Collection, a interface List ainda define outras (principais):*
  - `E get(int index)`
  - `E set(int index, Object o)`

Interface List

- Principais subclasses:
  - ArrayList
  - Vector
  - LinkedList
- Principais métodos:
  - void add(int index, Object o): adiciona objeto na posição indicada (empurra elementos existentes para a frente)
  - Object get(int index): recupera objeto pelo índice
  - int indexOf(Object o): procura objeto e retorna índice da primeira ocorrência
  - Object set(int index, Object o): grava objeto na posição indicada (apaga qualquer outro que ocupava a posição).
  - Object remove(int index)
  - ListIterator listIterator(): retorna um iterator

LinkedList, Vector ou ArrayList ?

- **ArrayList**
  - Escolha natural quando for necessário usar um vetor redimensionável: mais eficiente para leitura
  - Implementado internamente com vetores
  - Ideal para acesso aleatório
- **LinkedList**
  - Muito mais eficiente que ArrayList para remoção e inserção no meio da lista
  - Ideal para implementar pilhas, deques, filas unidirecionais e bidirecionais. Possui métodos para manipular essas estruturas
  - Ideal para acesso seqüencial
- Todas as implementacões da interface List não são "sincronizadas" (exceto **Vector**)

Exemplo de código

- *Criação:*

```
List lista = new ArrayList();
```

- *Adicionar elementos:*

```
lista.add( new Carro("Fusca") );
lista.add( "uma string" );
```

- *Obter elementos:*

```
Carro cl = (Carro) lista.get(0);
String s = (String)lista.get(1);
```

- *Listar elementos:*

```
System.out.println(lista);
String saida = lista.toString();
for( Object j : lista ) System.out.println(j);
```

Interface Set

- A interface Set representa uma coleção desordenada de dados que não permite elementos duplicados
- Estende a interface Collection, mas não possui métodos adicionais
  - Os mesmos métodos de Collection, porém, foram redesenhados para impedir o uso de cópias
- As estruturas de dados mais importantes que implementam esta interface: HashSet e TreeSet
- A escolha de qual classe usar vai depender de fatores como desempenho e facilidade de uso

Interface Set

- Principais subclasses:
  - TreeSet (implements SortedSet)
  - HashSet (implements Set)
- Principais métodos alterados
  - boolean add(Object): só adiciona o objeto se ele já não estiver presente (usa equals() para saber se o objeto é o mesmo)
  - contains(), retainAll(), removeAll(), ...: redefinidos para lidar com restrições de não-duplicação de objetos (esses métodos funcionam como operações sobre conjuntos)

HashSet ou TreeSet ?

- **HashSet**
  - Usa ,internamente, uma tabela hash para armazenar objetos
  - Precisa do método equals() para verificar se há cópias
  - Funciona apenas se os objetos inseridos implementam equals() e hashCode()
- **TreeSet**
  - Implementa um conjunto usando uma árvore binária
  - Além de evitar cópias, TreeSet também garante a ordem dos elementos presentes no conjunto
  - Como precisa colocar na ordem, os objetos precisam implementar a interface Comparable ou Comparator

Exemplo de código

- *Qual é a saída do programa abaixo?*

```
import java.util.*;
public class ExemploSets {
 public static void main(String[] args){
   Set set = new HashSet();
   set.add( "Bernardo" );
   set.add( "Elisabeth" );
   set.add( "Maria" );
   set.add( "Elisabeth" );
   set.add( "Clara" );

   System.out.println( set );

   Set sortedSet = new TreeSet( set );
   System.out.println( sortedSet );
 }
}
```

Interface Map

- Caracteriza uma família de coleções bem diferente:
  - São conhecidas como estruturas de dicionários
  - Um Map é um objeto que mapeia chaves para valores
- Objetos Map não podem conter chaves duplicadas
  - E cada chave só pode mapear um valor apenas
- Principais subclasses: HashMap, TreeMap e HashTable
  - void put(Object key, Object value): acrescenta um
  - Object get (Object key): recupera um objeto
  - Set keySet(): retorna um Set de chaves
  - Collection values(): retorna um Collection de valores

HashMap, HashTable ou TreeMap ?

- **HashMap**
  - Implementação baseada em HashTable
  - Mais performance que HashTable, já que não é synchronized
  - Usa hashCode() para otimizar a busca por uma chave
- **HashTable**
  - Implementação synchronized da interface Map
  - Mais segura, porém mais lenta que HashMap
  - É necessário que objetos implementem hashCode()
- **TreeMap**
  - Implementa um mapa ordenado (com base na chave)
  - Contém métodos para manipular elementos ordenados
  - Objetos devem implementar Comparable ou Comparator

Exemplo de código

- *Criação:*

```
Map map = new HashMap();
```

- *Adicionar elementos:*

```
map.put( "832.307.562", "José Silva" );
map.put( "275.455.321", "Maria José" );
```

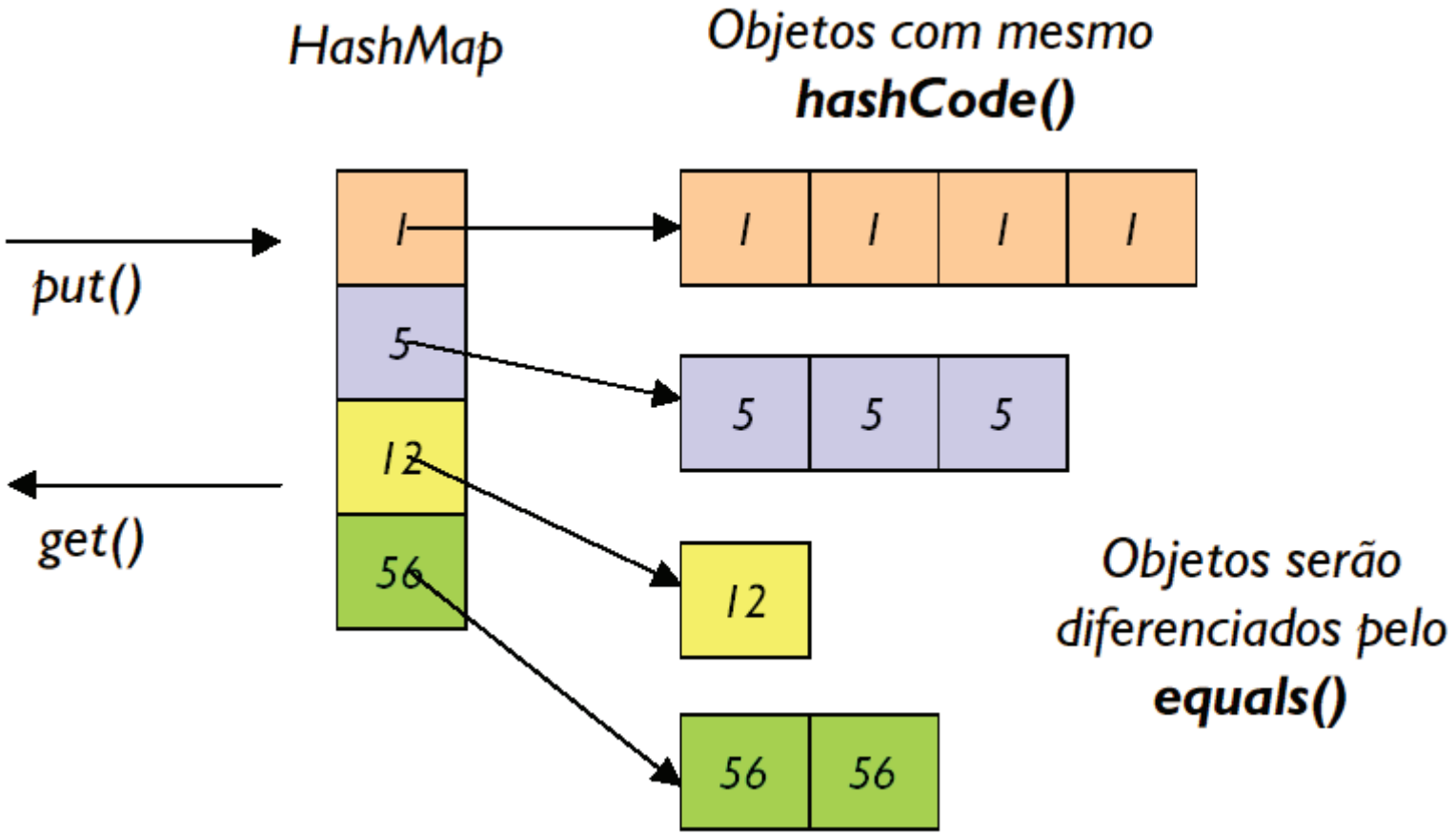
- *Obter elementos:*

```
String m = (String) mapa.get("275.455.321");
```

- *Listar elementos:*

```
System.out.println(mapa);
String saida = mapa.toString();
```

HashMap: como funciona



Coleções legadas de Java 1.0/1.1

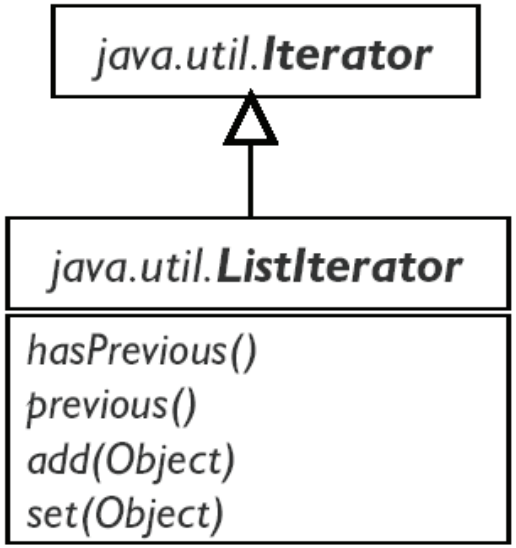
- São thread-safe e, portanto, menos eficientes
- Vector, implementa List
  - Use ArrayList, e não Vector, em novos programas
- Stack, subclasse de Vector
  - Implementa métodos de pilha: void push(Object), Object pop() e Object peek()
- Hashtable, implementa Map
  - Use HashMap, e não Hashtable, em novos programas
- Enumeration: tipo de Iterator
  - Retornada pelo método elements() em Vector, Stack, Hashtable e por vários outros métodos de classes mais antigas da API Java
  - boolean hasMoreElements(): equivalente a Iterator.hasNext()
  - Object nextElement(): equivalente a Iterator.next()
  - Use Iterator, e não Enumeration, em novos programas

Iterators

- *Para navegar dentro de uma Collection e selecionar cada objeto em determinada sequencia*
  - *Uma coleção pode ter vários Iterators*
  - *Isola o tipo da Coleção do resto da aplicação*
  - *Método iterator() (de Collection) retorna Iterator*

```
package java.util;
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

- *ListIterator possui mais métodos*
  - *Método listIterator() de List retorna ListIterator*



Exemplo de código

- *Como obter um iterator para navegar sobre uma coleção qualquer (suporte padrão)?*

```
List lista = new LinkedList();
//obtem o iterator
Iterator i = lista.iterator();
while( i.hasNext() ){
    Conta c = (Conta)i.next();
    System.out.println(c.getNumero());
    if( c.getSaldo()< 0 )
      i.remove();
}
```

- *No exemplo acima, poderíamos fazer diferente?*

Desvantagens das coleções (v.1.4)

- Não havia uma sintaxe que pudesse restringir o tipo específico dos objetos guardados (tudo era Object)
  - Aceitavam qualquer objeto: um objeto Gato podia ser adicionado numa lista de objetos Rato
  - Era tarefa do programador tomar estes cuidados
  - Era necessário um cast p/ recuperar um objeto lista

```
List toca = new ArrayList();
toca.add( new Rato("Rato 1") );
toca.add( new Rato("Rato 2") );
toca.add( new Rato("Rato 3") );
toca.add( new Gato("Félix") );
for( int i=0; i < toca.size(); i++ )
   Rato r = (Rato) toca.get(i);
   r.roer();
}
```

Ocorrerá ClassCastException quando o objeto retornado for um Gato em vez de um Rato

Não havia verificação em tempo de compilação

Note a presença de um cast a cada recuperação do objeto

Tipos Genéricos (v. 1.5 Java Tiger)

- Toda a API de coleções foi adaptada para permitir o uso de tipos genéricos
  - Maior segurança nas checagens de tipo
  - Não há necessidades de tantos cast's
  - Sem genéricos, era tarefa do programador lembrar que tipos de elementos estavam sendo manipulados na coleção
  - Permite a verificação dos tipos em tempo de compilação
  - Restringe tipos permitidos ao passarmos como parâmetro para a coleção

Solução com Genéricos

- *Genéricos permitem associar um tipo à referência*
  - *Exemplo: se você deseja definir uma lista que aceite apenas objetos Rato, faça:*

```
List<Rato> toca = new ArrayList<Rato>();
```

  - *Isto não é um ArrayList de Object, mas um ArrayList de objetos da classe Rato*
  - *Sintaxe básica: TipoGenérico<TipoEspecifico>*
- *Agora, não precisamos mais do cast!*

```
Rato r = toca.get(2);
```

Para um bom aproveitamento:

- *Codifique os exemplos mostrados nestes slides e verifique pontos de dúvidas*
- *Resolva todas as questões da* ***lista de coleções***
- *Procure o professor ou monitor da disciplina e questione conceitos, listas, etc.*
- *Não deixe para codificar tudo e acumular assunto para a primeira avaliação.*
  - *Este é apenas um dos assuntos abordados na prova!*