Colecções

- Uma colecção é um objecto capaz de agrupar múltiplos elementos numa única unidade de representação e processamento, possuindo propriedades de estrutura e funcionamento próprias
- A *Java Collections Framework* é uma arquitectura unificada, constituída por interfaces, classes abstractas, classes concretas e métodos que visa representar e manipular colecções
  - A JCF contém, entre outras coisas, a *Collections Interface*, as APIs fundamentais que estruturam todas as classes, dividindo-as em 4 famílias: conjuntos, listas, correspondências e filas

Colecções

- A *Collections Interface* define 4 interfaces fundamentais, definindo as características das classes que as implementam:
  - Set: conjunto de objectos, não existindo a noção de posição (primeiro, último, …), nem sendo permitidos duplicados
  - List: sequência de objectos, existindo a noção de ordem e permitindo duplicados
  - Map: correspondências unívocas entre objectos (chave – valor), em que as chaves são um conjunto, logo sem elementos repetidos
  - Queue: estruturas lineares do tipo FIFO

Colecções

- A versão 1.5 do Java (JAVA 5), trouxe modificações sintácticas e de tipos de dados, passando a verificação de tipos a ser feita em tempo de compilação
- Há 3 modificações relacionadas com colecções:
  - Auto-Boxing e Auto-Unboxing: conversão automática de valores dos tipos primitivos (int, …) em objectos das classes correspondentes (Integer, …) quando são inseridos em colecções e a conversão contrária quando são lidos de colecções
  - Iterador foreach: ciclo for aplicado a colecções que podem ser iteradas, substituindo os iteradores explícitos
  - Tipos genéricos: adicionam às colecções generalidade e segurança em tempo de compilação

Tipos genéricos

- Um tipo genérico é um tipo referenciado (classe ou interface) que usa na sua definição um ou mais tipos de dados (não primitivos) como parâmetro, tipos esses que serão mais tarde substituídos por tipos concretos quando o tipo genérico for instanciado
  - Por exemplo, Stack <E> representa uma pilha de "alguma coisa", sendo que esta pode ser um qualquer tipo não primitivo

Tipos genéricos

- Um tipo parametrizado é o resultado de uma instanciação do tipo genérico para um valor concreto da variável de tipo E
  - Por exemplo Stack <String> ou Stack <Estudante>
- O tipo parametrizado resultante da instanciação pode ser usado em declaração de variáveis, parâmetros e resultados de métodos
- As colecções beneficiaram da introdução de tipos genéricos, uma vez que é possível fazer a sua programação em função de um tipo genérico de dados a armazenar, o qual é substituído por um tipo concreto no momento da instanciação

Tipos genéricos

- Os tipos genéricos foram também utilizados na definição das interfaces da JCF
- Uma das interfaces mais úteis é a List <E> que herda de Iterable <E> e de Collection <E>
- Todas as classes que implementam List <E> têm assim que implementar todos os métodos definidos nestas interfaces. Exemplos:
  - add (E o);
  - add (int index E o);
  - boolean equals (Object o);
  - boolean contains (Object o);
  - List<E> sublist (int de, int ate);

Tipos genéricos

- As classes que implementam List <E> devem satisfazer:
  - Existe uma ordem nos seus elementos (é uma lista de objectos de tipo E)
  - Cada elemento ocupa uma posição referenciada por um índice inteiro a partir de 0
  - Os elementos podem ser inseridos em qualquer ponto da lista
  - A lista pode conter elementos em duplicado e elementos null

Tipos genéricos

- Existem 4 classes que implementam List <E>
  - ArrayList <E>
  - Vector <E>
  - Stack <E> (é subclasse de Vector)
  - LinkedList <E>

Classe ArrayList

- A classe ArrayList <E> implementa listas utilizando um array dinâmico, ou seja redimensionável em execução
- Para criar um arraylist é necessário definir qual o tipo dos seus elementos (classe, classe abstracta ou interface, já que nenhuma colecção armazena valores primitivos)
  - Por exemplo, se quisermos guardar uma lista de nomes teremos E = String, pelo que teremos a criação de uma ArrayList <String>:

    ```
    ArrayList <String> amigos = new ArrayList <String>(); ou
    ArrayList <String> amigos = new ArrayList <String>(50);
    ```


Classe ArrayList

- Sobre esta nova estrutura é possível fazer as operações definidas na interface List <E>, por exemplo:

  ```
  int tam1 = amigos.size ();
  amigos.add ("João");
  String nomeAmigo = amigos.get(0);
  ```

- A instrução `amigos.add(new Integer (127));` geraria um erro de compilação, pois o compilador sabe que amigos é um `ArrayList<String>`, pelo que não pode aceitar elementos de outros tipos (ainda que objectos)

Iteradores

- A iteração sobre colecções pode ser feita da maneira clássica usando índices:

```
for (int i=0; i<amigos.size(); i++)
    System.out.println ("Amigo: " + amigos.get(i));
```

- No entanto, existe a interface Iterator<E> (superinterface de Collection<E>) que obriga à existência de um método iterator(); que cria um objecto Iterator <E>, um iterador automático sobre uma colecção de elementos de tipo E
- Este trabalha sobre a colecção usando 3 métodos:
  - hasNext(); next(); remove();

Iteradores

- Uma das formas de usar um iterador:

```
Iterator <String> it = amigos.iterator();
while (it.hasNext())
    System.out.println ("Amigo: " +it.next());
```

  - É claro que it.next() devolve uma String, pois é esse o tipo dos elementos do arrayList e do Iterator
- O Java 5 introduziu uma forma ainda mais compacta de fazer iteração sobre colecções, usando uma nova forma de ciclo for, denominada foreach
  - A sua forma genérica é:

```
for (Tipo elem : col_iterável <Tipo>) instruções
```

Iteradores

- O que no nosso exemplo dará:

```
for (String nome : amigos)
    System.out.println ("Amigo: " +nome);
```

- Esta forma de for pode ser usada com qualquer colecção iterável (por exemplo arrays)
- A sua utilização só faz sentido quando se quer iterar sobre toda a colecção
- Por exemplo, num algoritmo de pesquisa tal não faz sentido, pois só interessa procurar até encontrar o pretendido
  - Nesse caso usa-se o while associado a um iterador, incluindo a expressão de pesquisa na sua condição

```
while (it.hasNext() && !encontrado) ....
```


Iteradores

- As listas (e só estas) têm também um método `listIterator()` que devolve um iterador especial `ListIterator<E>`, que acrescenta métodos que possibilitam:
  - Navegar na lista em sentido contrário, com `hasPrevious()` e `previous()`
  - Saber o índice do próximo elemento a ser iterado em qualquer dos sentidos, `nextIndex()` e `previousIndex()`
  - Fazer `remove()`, `set(E e)` e `add(E e)` em qualquer momento da iteração
    - `remove()` e `set(E e)`, removem e alteram o último elemento iterado, respectivamente

Iteradores

- Um exemplo de utilização de um ListIterator<E> num algoritmo de inversão de um arraylist:

```
Public ArrayList<String> inverte (ArrayList<String> lst){
    ArrayList<String> aux = new ArrayList<String>(lst);
    ListIterator<String> normal = aux.listIterator();
    ListIterator<String> inv = aux.listIterator(aux.size());
    int conta = 0; int meio = lst.size()/2;
    while(conta<meio) {
        String temp = normal.next(); conta++;
        normal.set(inv.previous());
        inv.set(temp);
    }
    return aux;
}
```


Iteradores

- Alguns dos métodos de ArrayList usam iteradores para implementar métodos que trabalham com grande quantidade de dados. Por exemplo:
  - addAll(colecção): junta ao fim da lista receptora os elementos da colecção parâmetro, pela ordem dada pelo iterador desta (as colecções devem ser compatíveis)
  - removeAll(colecção): remove do receptor os elementos da lista parâmetro
  - retainAll(colecção): remove do receptor os elementos que não pertencem à lista parâmetro
  - containsAll(colecção): verifica se todos os elementos da lista parâmetro pertencem à lista receptora