

# Estruturas de Dados

- Como já vimos atrás, podemos criar conceitos mais complexos através do mecanismo de composição/agregação de objectos de classes mais simples:
  - a Turma a partir de Aluno
  - o Stand a partir de Veículo
  - etc.
- Mas para isso precisamos de ter colecções de objectos...

- Até ao momento apenas temos disponível a utilização de arrays

```
Aluno[] alunos = new Aluno[30];
Veiculo[] carros = new Veiculo[10];

for (int i=0; i<alunos.length && !encontrado; i++)
    if (alunos[i].getNota() == 20)
        encontrado = true;

for(Veiculo v: carros)
    System.out.println(v.toString());
```

- Esta é uma solução simples e testada, com o inconveniente de o tamanho da estrutura de dados ser estaticamente definido.
- Será que conseguimos ter uma estrutura de dados, baseada em arrays, que pudesse crescer de forma transparente para o utilizador?
- (primeira solução) Sim, bastando para tal criarmos uma classe com esse comportamento

- Seja essa classe chamada de GrowingArray e, por comodidade, vamos utilizar instâncias de Circulo
- Que operações necessitamos:
  - adicionar um círculo (no fim e numa posição)
  - remover um círculo
  - ver se um círculo existe
  - dar a posição de um círculo na estrutura
  - dar número de elementos existentes

- Documentação com os métodos necessários:

|                              |                                                                                                                                                                    |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constructor Summary          |                                                                                                                                                                    |
| Constructors                 |                                                                                                                                                                    |
| Constructor and Description  |                                                                                                                                                                    |
| GrowingArray()               |                                                                                                                                                                    |
| GrowingArray(int capacidade) |                                                                                                                                                                    |
| Method Summary               |                                                                                                                                                                    |
| All Methods                  | Instance MethodsConcrete Methods                                                                                                                                   |
| Modifier and Type            | Method and Description                                                                                                                                             |
| void                         | add(MyDynamicArray.Circulo c)<br>Adiciona o elemento passado como parâmetro ao fim do array                                                                        |
| void                         | add(int indice, MyDynamicArray.Circulo c)<br>Método que adiciona um elemento numa determinada posição, forçando a que os elementos à direita no array façam shift. |
| boolean                      | contains(MyDynamicArray.Circulo c)<br>Método que determina se um elemento está no array.                                                                           |
| MyDynamicArray.Circulo       | get(int indice)<br>Devolve o elemento que está na posição indicada.                                                                                                |
| int                          | indexOf(MyDynamicArray.Circulo c)<br>Método que determina o índice do array onde está localizada a primeira ocorrência de um objecto.                              |
| boolean                      | isEmpty()<br>Método que determina se o array contém elementos, ou se está vazio.                                                                                   |
| boolean                      | remove(MyDynamicArray.Circulo c)                                                                                                                                   |

# Composição ou agregação?

- Ao criar esta estrutura temos de tomar a decisão se a colecção de elementos assenta em composição ou agregação.
- Quais são as implicações?
- Se for composição os métodos de inserção e get terão de prever a utilização de clone
- é a solução adequada para todas as situações?

# Composição ou agregação?

- Se a estratégia de construção desta estrutura de dados for do tipo Composição, esta colecção terá sempre este comportamento.
- não poderá ser utilizada em estratégias de agregação.
- os métodos de get e set, adição e recuperação (e os construtores) farão sempre clone e gerarão novas referências.

- Declarações iniciais:

```
public class GrowingArray {  
  
    private Circulo[] elementos;  
    private int size;  
  
    /**  
     * variável que determina o tamanho inicial do array,  
     * se for utilizado o construtor vazio.  
     */  
    private static final int capacidade_inicial = 20;  
  
    public GrowingArray(int capacidade) {  
        this.elementos = new Circulo[capacidade];  
        this.size = 0;  
    }  
  
    public GrowingArray() {  
        this(capacidade_inicial);  
    }  
}
```

invocação do  
construtor anterior

- get de um elemento da estrutura de dados

```
/**
 * Devolve o elemento que está na posição indicada.
 *
 * @param indice posição do elemento a devolver
 * @return o objecto que está na posição indicada no parâmetro
 * (deveremos ter atenção às situações em que a posição não existe)
 */
public Circulo get(int indice) {
    if (indice <= this.size)
        return this.elementos[indice];
    else
        return null; // ATENÇÃO!
}
```

- set de uma posição da estrutura

```
/**
 * Método que actualiza o valor de uma determinada posição do array.
 *
 * @param indice a posição que se pretende actualizar
 * @param c o círculo que se pretende colocar na estrutura de dados
 *
 */
public void set(int indice, Circulo c) {
    if (indice <= this.size) //não se permitem "espaços vazios"
        this.elementos[indice] = c;
}
```

- se a estratégia de utilização for de composição, quem chama este método deve previamente fazer clone.

- adicionar um elemento à estrutura de dados

```
/**
 * Adiciona o elemento passado como parâmetro ao fim do array
 *
 * @param c circulo que é adicionado ao array
 *
 */
public void add(Circulo c) {
    aumentaCapacidade(this.size + 1);
    this.elementos[this.size++] = c;
}
```

- método auxiliar que aumenta espaço

```
/**
 * Método auxiliar que verifica se o array alocado tem capacidade
 * para guardar mais elementos.
 * Por cada nova inserção, verifica se estamos a mais de metade
 * do espaço
 * alocado e, caso se verifique, aloca mais 1.5 de capacidade.
 *
 */
private void aumentaCapacidade(int capacidade) {
    if (capacidade > 0.5 * this.elementos.length) {
        int nova_capacidade = (int)(this.elementos.length * 1.5);
        this.elementos = Arrays.copyOf(this.elementos, nova_capacidade);
    }
}
```

```
/**
 * Método que adiciona um elemento numa determinada posição,
 * forçando a
 * que os elementos à direita no array façam shift.
 * Tal como no método de set não são permitidos espaços.
 *
 * @param indice indice onde se insere o elemento
 * @param c circulo que será inserido no array
 *
 */

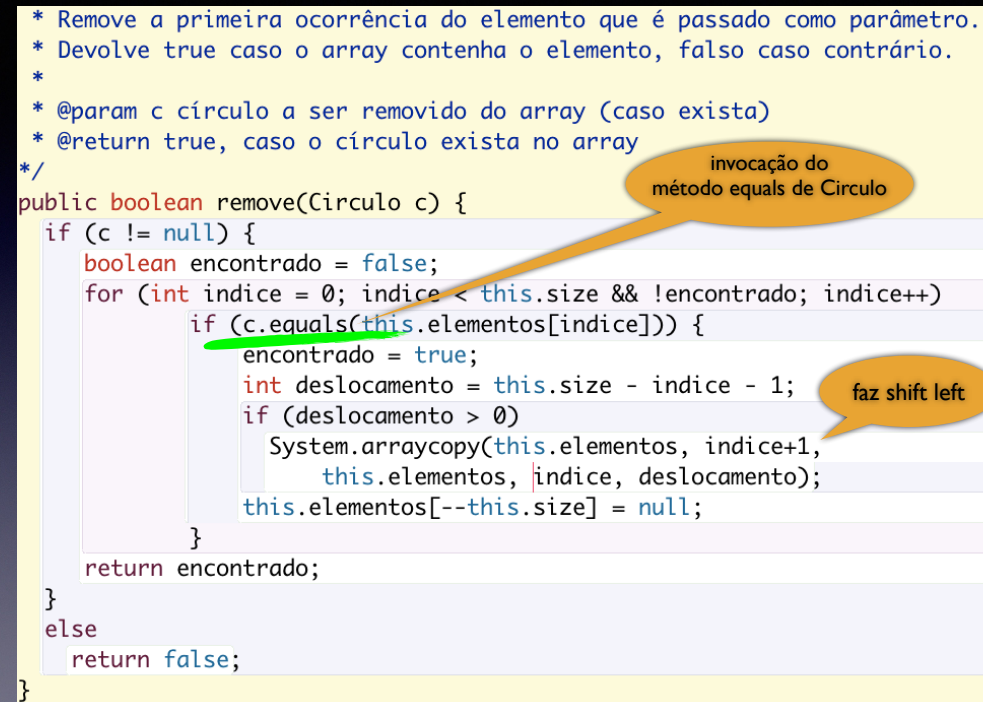
public void add(int indice, Circulo c) {
    if (indice <= this.size) {
        aumentaCapacidade(this.size+1);
        System.arraycopy(this.elementos, indice, this.elementos,
            indice + 1, this.size - indice);
        this.elementos[indice] = c;
        this.size++;
    }
}
```

podemos também  
utilizar o Arrays.copyOfRange

```
/**
 * Remove do array o elemento que está na posição indicada no parâmetro.
 * Todos os elementos à direita do índice sofrem um deslocamento
 * para a esquerda.
 * @param indice índice do elemento a ser removido
 * @return o elemento que é removido do array. No caso do índice não
 * existir devolver-se-á null.
 */
public Circulo remove(int indice) {
    if (indice <= this.size) {
        Circulo c = this.elementos[indice];

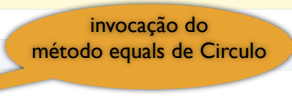
        int deslocamento = this.size - indice - 1;
        if (deslocamento > 0)
            System.arraycopy(this.elementos, indice+1, this.elementos,
                             indice, deslocamento);
        this.elementos[--this.size] = null;
        return c;
    }
    else
        return null;
}
```

```
* Remove a primeira ocorrência do elemento que é passado como parâmetro.
* Devolve true caso o array contenha o elemento, falso caso contrário.
*
* @param c círculo a ser removido do array (caso exista)
* @return true, caso o círculo exista no array
*/
public boolean remove(Circulo c) {
    if (c != null) {
        boolean encontrado = false;
        for (int indice = 0; indice < this.size && !encontrado; indice++)
            if (c.equals(this.elementos[indice])) {
                encontrado = true;
                int deslocamento = this.size - indice - 1;
                if (deslocamento > 0)
                    System.arraycopy(this.elementos, indice+1,
                                    this.elementos, indice, deslocamento);
                this.elementos[--this.size] = null;
            }
        return encontrado;
    }
    else
        return false;
}
```



```
/**
 * Método que determina o índice do array onde está localizada a
 * primeira ocorrência de um objecto.
 *
 * @param c círculo de que se pretende determinar a posição
 * @return a posição onde o círculo se encontra. -1 caso não esteja no
 * array ou o círculo passado como parâmetro seja null.
 */

public int indexOf(Circulo c) {
    int posicao = -1;
    if (c != null) {
        boolean encontrado = false;
        for (int i = 0; i < this.size && !encontrado; i++)
            if (c.equals(this.elementos[i])) {
                encontrado = true;
                posicao = i;
            }
    }
    return posicao;
}
```



invocação do  
método equals de Circulo

```
/**
 * Método que determina se um elemento está no array.
 *
 * @param c círculo a determinar se está no array
 * @return true se o objecto estiver inserido na estrutura de dados,
 * false caso contrário.
 */
public boolean contains(Circulo c) {
    return indexOf(c) >= 0;
}

/**
 * Método que determina se o array contém elementos, ou se está vazio.
 *
 * @return true se o array estiver vazio, false caso contrário.
 */
public boolean isEmpty() {
    return this.size == 0;
}
```

```
public class TesteGA {  
    public static void main(String[] args) {  
  
        Circulo c1 = new Circulo(2,4,4.5);  
        Circulo c2 = new Circulo(1,4,1.5);  
        Circulo c3 = new Circulo(2,7,2.0);  
        Circulo c4 = new Circulo(3,3,2.0);  
        Circulo c5 = new Circulo(2,6,7.5);  
  
        GrowingArray ga = new GrowingArray(10);  
        ga.add(c1.clone());  
        ga.add(c2.clone());  
        ga.add(c3.clone());  
  
        System.out.println("Num elementos = " + ga.size());  
        System.out.println("Posição do c2 = " + ga.indexOf(c2));  
    }  
}
```

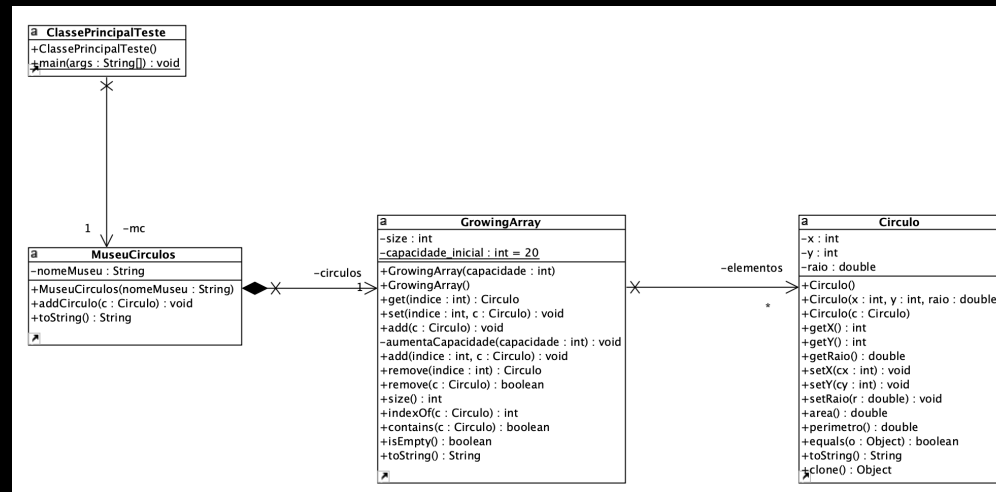


AULA 4 TERMINA AQUI!?

# A utilização de GrowingArray

- Imagine-se que queremos criar uma aplicação em que temos um objecto que tem uma coleção de círculos:
  - A classe GrowingArray é a estrutura de dados
  - Temos de ter uma classe que implemente a lógica de negócio e essa classe é que tem a v.i. do tipo GrowingArray
  - Precisamos de ter uma classe de teste com o método main

# Diagrama de classe



```
public class MuseuCirculos {  
    private String nomeMuseu;  
    private GrowingArray circulos;  
  
    public MuseuCirculos(String nomeMuseu) {  
        this.nomeMuseu = nomeMuseu;  
        this.circulos = new GrowingArray();  
    }  
  
    public void addCirculo(Circulo c) {  
        this.circulos.add(c.clone());  
    }  
  
    public String toString() {  
        StringBuilder sb = new StringBuilder();  
        sb.append("Museu de Circulos --> ");  
        sb.append("Nome : ");  
        sb.append(this.nomeMuseu + "\n");  
        sb.append(this.circulos.toString());  
  
        return sb.toString();  
    }  
}
```

```
public class ClassePrincipalTeste {  
    public static void main(String[] args) {  
        MuseuCirculos mc = new MuseuCirculos("Circulos de P00");  
  
        Circulo c1 = new Circulo(2,4,4.5);  
        Circulo c2 = new Circulo(1,4,1.5);  
        Circulo c3 = new Circulo(2,7,2.0);  
        Circulo c4 = new Circulo(3,3,2.0);  
        Circulo c5 = new Circulo(2,6,7.5);  
  
        mc.addCirculo(c1); // internamente mc vai fazer cópia  
        mc.addCirculo(c2);  
        mc.addCirculo(c3);  
        mc.addCirculo(c4);  
        mc.addCirculo(c5);  
  
        System.out.println("== toString do museu ==");  
        System.out.println(mc.toString());  
    }  
}
```

# Colecções Java

- O Java oferece um conjunto de classes que implementam as estruturas de dados mais utilizadas
- oferecem uma API consistente entre si
- permitem que sejam utilizadas com qualquer tipo de objecto - são parametrizadas por tipo

- Poderemos representar:
  - `ArrayList<Aluno> alunos`
  - `HashSet<Aluno> alunos;`
  - `HashMap<String, Aluno> turmaAlunos;`
  - `TreeMap<String, Docente> docentes;`
  - `Stack<Pedido> pedidosTransferência;`
  - ...

- Ao fazer-se `ArrayList<Aluno>` passa a ser o compilador a testar, e validar, que só são utilizados objectos do tipo `Aluno` no `ArrayList`.
- isto dá uma segurança adicional aos programas, pois em tempo de execução não teremos erros de compatibilidade de tipos
- os tipos de dados são verificados em tempo de compilação

- As colecções em Java beneficiam de:
  - auto-boxing e auto-unboxing, ie, a capacidade de converter automaticamente tipos primitivos para instâncias de classes wrapper.
  - int para Integer, double para Double, etc.
  - o programador não tem de codificar a transformação

- tipos genéricos
  - as colecções passam a ser definidas em função de um tipo de dados que é associado aquando da criação
  - a partir daí o compilador passa a garantir que os conteúdos da colecção são do tipo esperado

# Collections Framework

- O Java Collections Framework agrupa as várias classes genéricas que correspondem às implementações de referência de:
  - Listas:API de List<E>
  - Conjuntos:API de Set<E>
  - Correspondências unívocas:API de Map<K,V>

Abstração de  
implementação!

# A definição Collection

- Da documentação: *“The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.”*
- *“All general-purpose Collection implementation classes (...) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument.”*

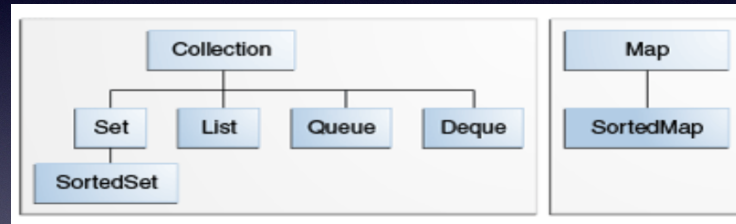
- E ainda:
  - “Many methods in Collections Framework interfaces are defined in terms of the *equals* method. For example, the specification for the *contains(Object o)* method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e)).”"

# A API de Collection

|                        |                                                    |                                                                                                                     |
|------------------------|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| boolean                | <b>add(E e)</b>                                    | Ensures that this collection contains the specified element (optional operation).                                   |
| boolean                | <b>addAll(Collection&lt;? extends E&gt; c)</b>     | Adds all of the elements in the specified collection to this collection (optional operation).                       |
| void                   | <b>clear()</b>                                     | Removes all of the elements from this collection (optional operation).                                              |
| boolean                | <b>contains(Object o)</b>                          | Returns true if this collection contains the specified element.                                                     |
| boolean                | <b>containsAll(Collection&lt;?&gt; c)</b>          | Returns true if this collection contains all of the elements in the specified collection.                           |
| boolean                | <b>equals(Object o)</b>                            | Compares the specified object with this collection for equality.                                                    |
| int                    | <b>hashCode()</b>                                  | Returns the hash code value for this collection.                                                                    |
| boolean                | <b>isEmpty()</b>                                   | Returns true if this collection contains no elements.                                                               |
| Iterator<E>            | <b>iterator()</b>                                  | Returns an iterator over the elements in this collection.                                                           |
| default Stream<E>      | <b>parallelStream()</b>                            | Returns a possibly parallel Stream with this collection as its source.                                              |
| boolean                | <b>remove(Object o)</b>                            | Removes a single instance of the specified element from this collection, if it is present (optional operation).     |
| boolean                | <b>removeAll(Collection&lt;?&gt; c)</b>            | Removes all of this collection's elements that are also contained in the specified collection (optional operation). |
| default boolean        | <b>removeIf(Predicate&lt;? super E&gt; filter)</b> | Removes all of the elements of this collection that satisfy the given predicate.                                    |
| boolean                | <b>retainAll(Collection&lt;?&gt; c)</b>            | Retains only the elements in this collection that are contained in the specified collection (optional operation).   |
| int                    | <b>size()</b>                                      | Returns the number of elements in this collection.                                                                  |
| default Spliterator<E> | <b>spliterator()</b>                               | Creates a <b>Spliterator</b> over the elements in this collection.                                                  |
| default Stream<E>      | <b>stream()</b>                                    | Returns a sequential Stream with this collection as its source.                                                     |
| Object[]               | <b>toArray()</b>                                   | Returns an array containing all of the elements in this collection.                                                 |

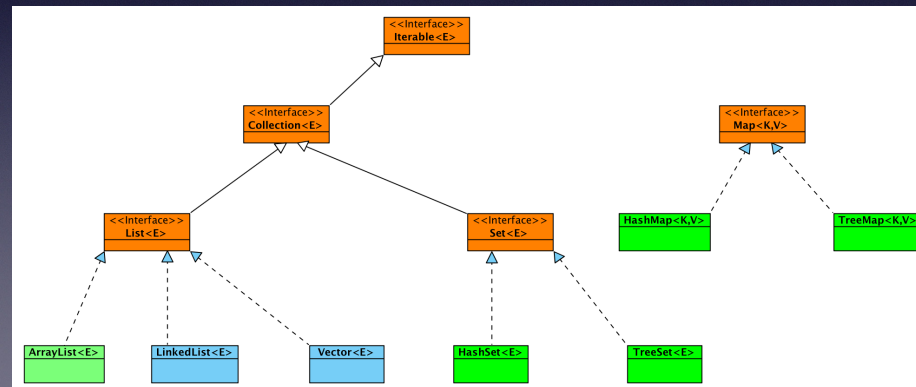
# Estrutura da JCF

- Existe uma arrumação por API (interfaces)

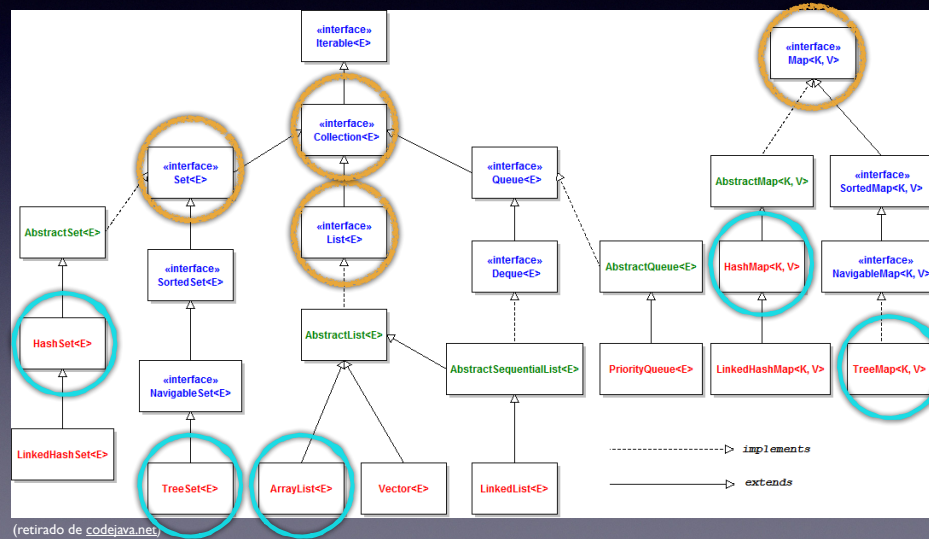


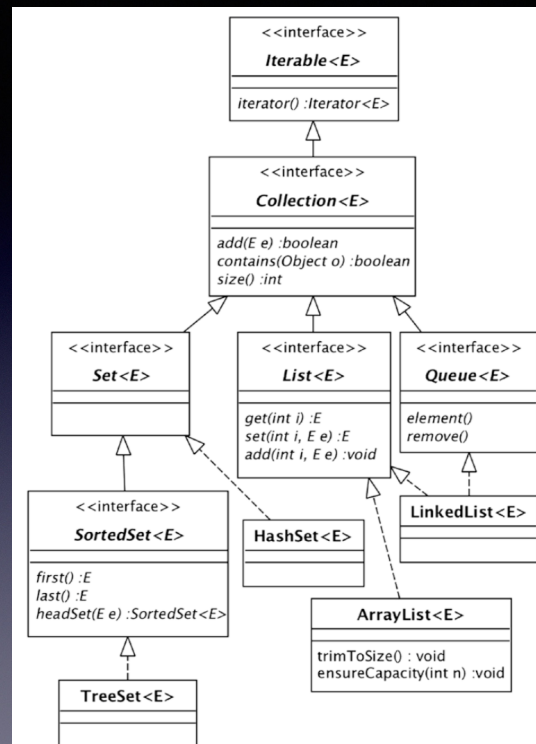
- Todas as colecções, `Collection<E>`, são iteráveis externamente através de `Iterable<E>`

- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)
- *Vamos estudar as que estão a verde*



# Interfaces e Implementações do JFC





- todas as coleções permitem ser iteradas
- implementações de List<E> adicionam métodos com indicação de posição

(retirado de Java Program Design, Edward Sciore)

# ArrayList<E>

- As classes da Java Collections Framework são exemplos muito interessantes de codificação
- Como o código destas classes está escrito em Java é possível ao programador observar como é que foram implementadas

# ArrayList<E>: v.i. e construtores

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer.
     */
    private transient Object[] elementData;

    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param initialCapacity the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
}
```

# ArrayList<E>: existe?

```
public boolean contains(Object o) {  
    return indexOf(o) >= 0;  
}  
  
public int indexOf(Object o) {  
    if (o == null) {  
        for (int i = 0; i < size; i++)  
            if (elementData[i]==null)  
                return i;  
    } else {  
        for (int i = 0; i < size; i++)  
            if (o.equals(elementData[i]))  
                return i;  
    }  
    return -1;  
}
```

Espera-se que  
o equals esteja  
(bem) definido!

# ArrayList<E>: inserir

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
        size - index);
    elementData[index] = element;
    size++;
}
```

# ArrayList<E>: get e set

```
public E get(int index) {  
    rangeCheck(index);  
  
    return elementData(index);  
}  
  
public E set(int index, E element) {  
    rangeCheck(index);  
  
    E oldValue = elementData(index);  
    elementData[index] = element;  
    return oldValue;  
}
```

# Colecções Java

- Tipos de colecções disponíveis:
  - listas (definição em `List<E>`)
  - conjuntos (definição em `Set<E>`)
  - queues (definição em `Queue<E>`)
- noção de família (muito evidente) nas APIs de cada um destes tipos de colecções.

# List<E>

- Utilizar sempre que precise de manter ordem
- O método add não testa se o objecto existe na colecção (admite repetições)
- O método contains verifica sempre o resultado de equals
- Implementação utilizada: **ArrayList<E>**

# ArrayList<E>

|                     |                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Construtores        | ArrayList<E>(); ArrayList<E>(int initialCapacity)                                                                                                                       |
| Adicionar elementos | boolean add(E e); void add(int index, E element);<br>boolean addAll(Collection c); boolean addAll(int index, Collection c)                                              |
| Remover elementos   | boolean remove(Object o); E remove(int index);<br>boolean removeAll(Collection c)<br>boolean retainAll(Collection c)<br>boolean removeIf(Predicate p)                   |
| Consultar           | E get(int index); int indexOf(Object o);<br>int lastIndexOf(Object o); boolean contains(Object o)<br>boolean containsAll(Collection c)<br>boolean isEmpty(); int size() |
| Alterar elementos   | E set(int index, E element); void clear();                                                                                                                              |
| Iteradores externos | Iterator<E> iterator()                                                                                                                                                  |
| Iteradores internos | Stream<E> stream(); void forEach(Consumer c)                                                                                                                            |
| Outros              | boolean equals(Object o); T[] toArray(T[] a)                                                                                                                            |

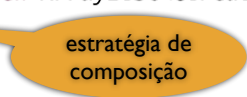
```
import java.util.ArrayList;
public class TesteArrayList {
    public static void main(String[] args) {

        Circulo c1 = new Circulo(2,4,4.5);
        Circulo c2 = new Circulo(1,4,1.5);
        Circulo c3 = new Circulo(2,7,2.0);
        Circulo c4 = new Circulo(3,3,2.0);
        Circulo c5 = new Circulo(2,6,7.5);

        ArrayList<Circulo> circs = new ArrayList<Circulo>();
        circs.add(c1.clone());
        circs.add(c2.clone());
        circs.add(c3.clone());

        System.out.println("Num elementos = " + circs.size());
        System.out.println("Posição do c2 = " + circs.indexOf(c2));

        for(Circulo c: circs)
            System.out.println(c.toString());
    }
}
```



# Percorrer uma coleção

- Podemos utilizar o ciclo for(each) para percorrer uma coleção:

```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = 0.0;

    for(Aluno a: lstAlunos)
        tot += a.getNota();

    return tot/lstAlunos.size();
}
```

```
/**
 * Quantos alunos passam?
 *
 * @return um int com n° alunos que passa
 */
public int quantosPassam() {
    int qt = 0;

    for(Aluno a: lstAlunos)
        if (a.passa()) qt++;

    return qt;
}
```

```
public boolean passa() {
    return this.nota >= Aluno.NOTA_PARA_PASSAR;
}
```

Na classe **Aluno**

- ... mas...
- podemos querer parar antes do fim
- podemos não ter acesso à posição do elemento na colecção (no caso dos conjuntos)

```
/**
 * Alguns alunos passam?
 *
 * @return true se algum aluno passa
 */
public boolean algumPassa() {
    boolean alguem = false;

    for(Aluno a: lstAlunos)
        if (a.passa())
            alguem = true;
    return alguem;
}
```

Esta solução é  
claramente  
ineficiente!

- logo, é necessário um mecanismo mais flexível para percorrer colecções