

# 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.
<b>Iterator&lt;E&gt;</b>	<b>iterator()</b>	Returns an iterator over the elements in this collection.
default <b>Stream&lt;E&gt;</b>	<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 <b>Splitter&lt;E&gt;</b>	<b>splitter()</b>	Creates a <b>Splitter</b> over the elements in this collection.
default <b>Stream&lt;E&gt;</b>	<b>stream()</b>	Returns a sequential Stream with this collection as its source.
<b>Object[]</b>	<b>toArray()</b>	Returns an array containing all of the elements in this collection.

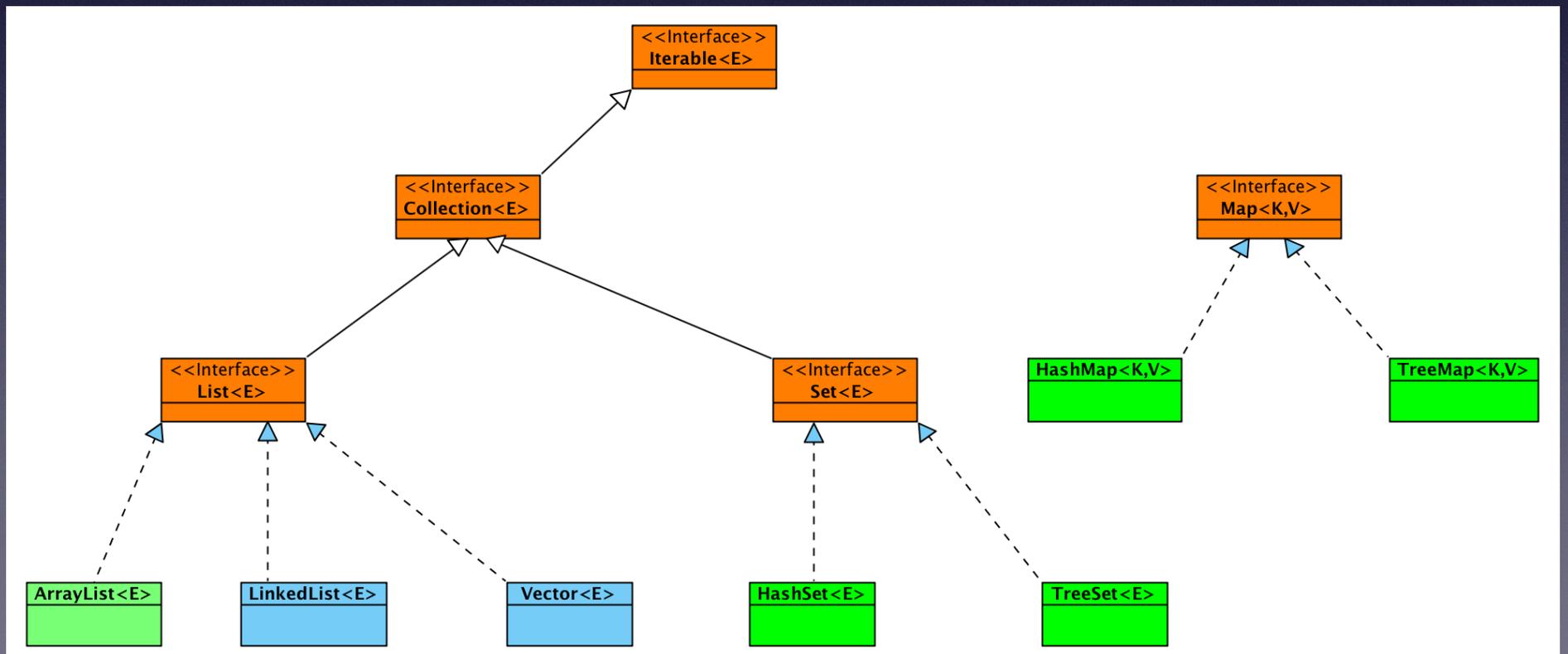
# Estrutura da JCF

- Existe uma arrumação por API (interfaces)

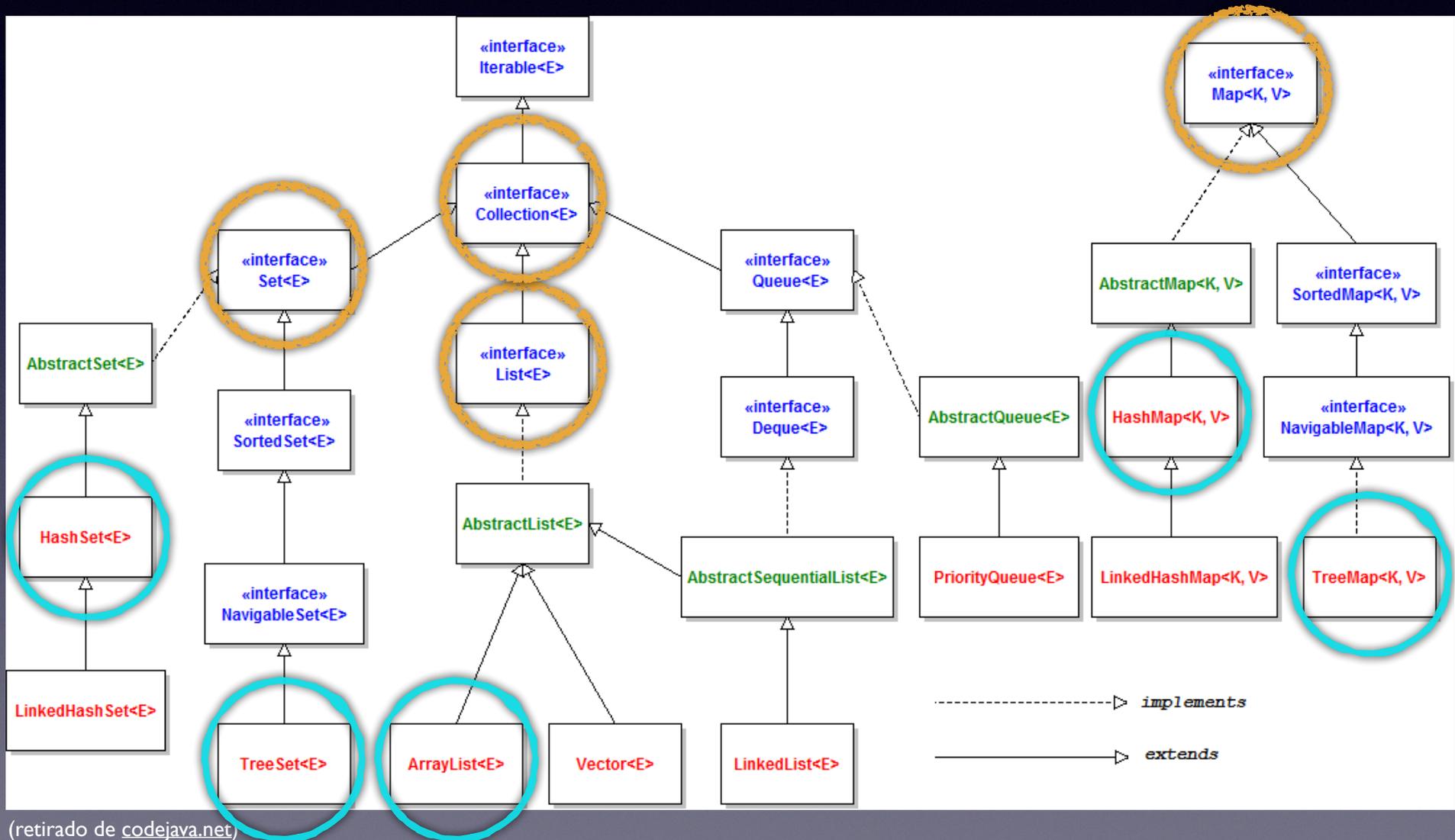


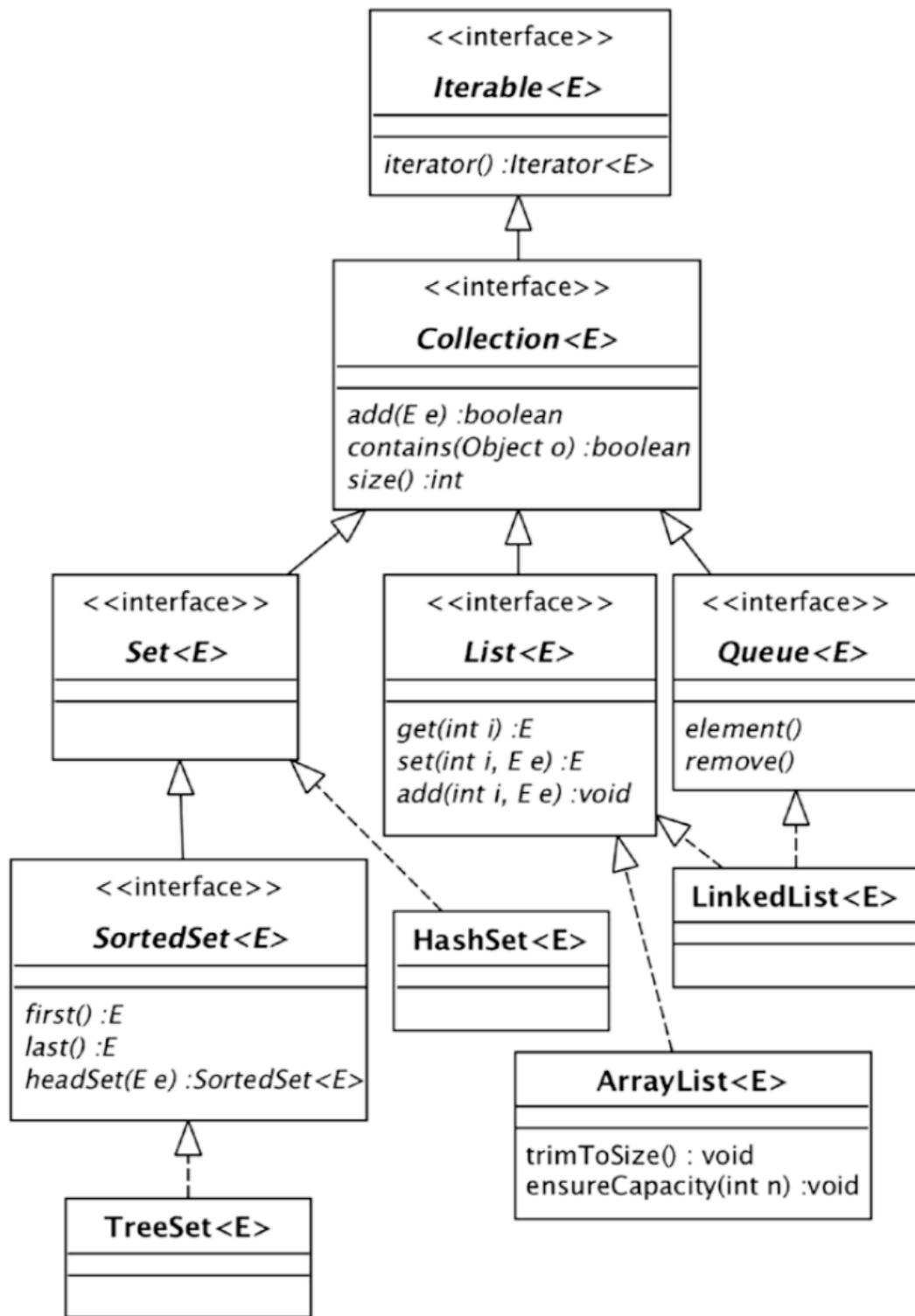
- Todas as coleções, `Collection<E>`, são iteráveis externamente através de `Iterable<E>`

- 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

# 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;

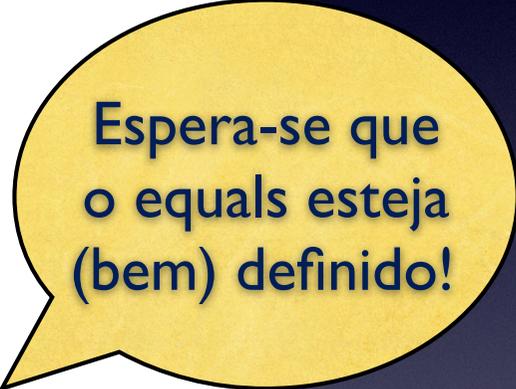
    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param initialCapacity the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        |...
        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	<code>ArrayList&lt;E&gt;(); ArrayList&lt;E&gt;(int initialCapacity)</code>
Adicionar elementos	<code>boolean add(E e); void add(int index, E element); boolean addAll(Collection c); boolean addAll(int index, Collection c)</code>
Remover elementos	<code>boolean remove(Object o); E remove(int index); boolean removeAll(Collection c) boolean retainAll(Collection c) boolean removeIf(Predicate p)</code>
Consultar	<code>E get(int index); int indexOf(Object o); int lastIndexOf(Object o); boolean contains(Object o) boolean containsAll(Collection c) boolean isEmpty(); int size()</code>
Alterar elementos	<code>E set(int index, E element); void clear();</code>
Iteradores externos	<code>Iterator&lt;E&gt; iterator()</code>
Iteradores internos	<code>Stream&lt;E&gt; stream(); void forEach(Consumer c)</code>
Outros	<code>boolean equals(Object o); T[] toArray(T[] a)</code>

```
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());
    }
}
```

estratégia de  
composição

# 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 nº alunos que passa
 */
public int quantosPassam() {
    int qt = 0;

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

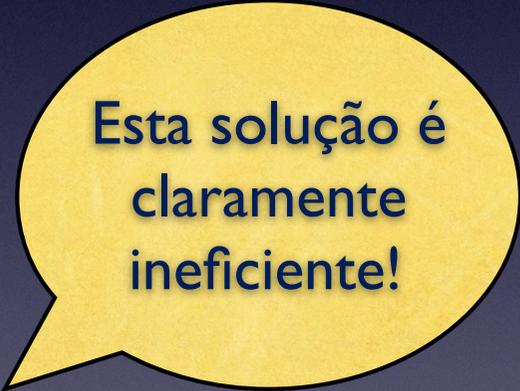
    return qt;
}
```

```
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 coleção (no caso dos conjuntos)
- estamos sempre a repetir o código do ciclo

```
/**  
 * Algum aluno passa?  
 *  
 * @return true se algum aluno passa  
 */  
public boolean alguemPassa() {  
    boolean alguem = false;  
  
    for(Aluno a: lstAlunos)  
        if (a.passa())  
            alguem = true;  
    return alguem;  
}
```



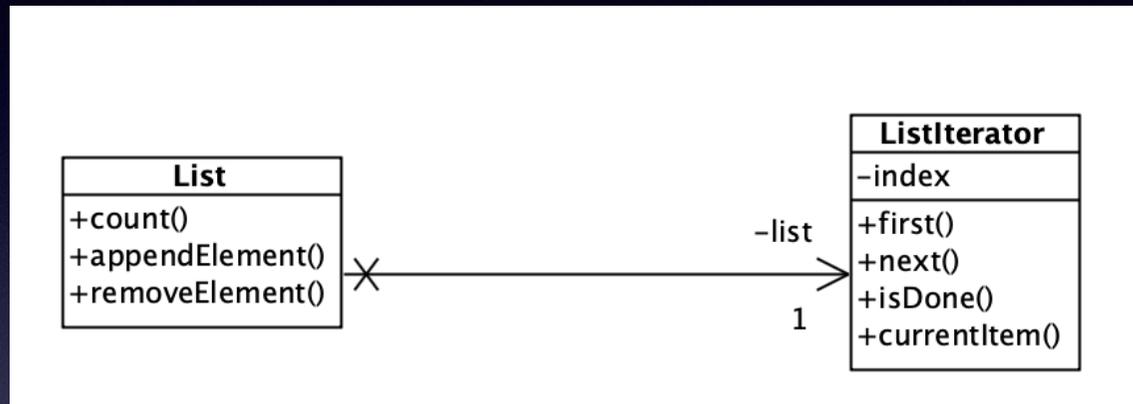
Esta solução é claramente ineficiente!

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

# Iteradores externos

- O **Iterator** é um padrão de concepção bem conhecido e que permite providenciar uma forma de aceder aos elementos de uma colecção de objectos, sem que seja necessário saber qual a sua representação interna
  - basta para tal, que todas as colecções saibam criar um iterator!
  - não precisamos saber como tal é feito!

- Um iterador de uma lista poderia ser:



- o iterador precisa de ter mecanismos para:
  - acessar ao objecto apontado
  - avançar
  - determinar se chegou ao fim

- Iterator API

## Method Summary

### Methods

Modifier and Type	Method and Description
boolean	<b>hasNext()</b> Returns <code>true</code> if the iteration has more elements.
<b>E</b>	<b>next()</b> Returns the next element in the iteration.
void	<b>remove()</b> Removes from the underlying collection the last element returned by this iterator (optional operation).

- Utilizando Iterators...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    boolean alguem = false;
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext() && !alguem) {
        a = it.next();
        alguem = a.passa();
    }
    return alguem;
}
```

- remover alunos...

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext()) {
        a = it.next();
        if (a.getNota() < nota)
            it.remove();
    }
}
```

# Iterator<E>

- Em resumo...
- Todas as colecções implementam o método: **Iterator<E> iterator()** que cria um iterador activo sobre a colecção
- Padrão de utilização:

```
Iterator<E> it = coleção.iterator();  
E elem;  
  
while(it.hasNext()) {  
    elem = it.next();  
    // fazer algo com elem  
}
```

- Procurar:

```
boolean encontrado = false;
Iterator<E> it = coleção.iterator();
E elem;

while(it.hasNext() && !encontrado) {
    elem = it.next();
    if (criterio de procura sobre elem)
        encontrado = true;
}
// fazer alguma coisa com elem ou com encontrado
```

- Remover:

```
Iterator<E> it = coleção.iterator();
E elem;

while(it.hasNext()) {
    elem = it.next();
    if (criterio sobre elem)
        it.remove();
}
```

# Iteradores internos

- Todas as colecções implementam o método: **forEach()**
- Aceita uma função para *trabalhar* em todos os elementos da coleção
- É implementado com um foreach...

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

- Iterador externo

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    for(Aluno a: lstAlunos)
        a.sobeNota(bonus);
}
```

- Iterador interno - forEach()

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach((Aluno a) -> {a.sobeNota(bonus);});
}
```