

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;
 - ...

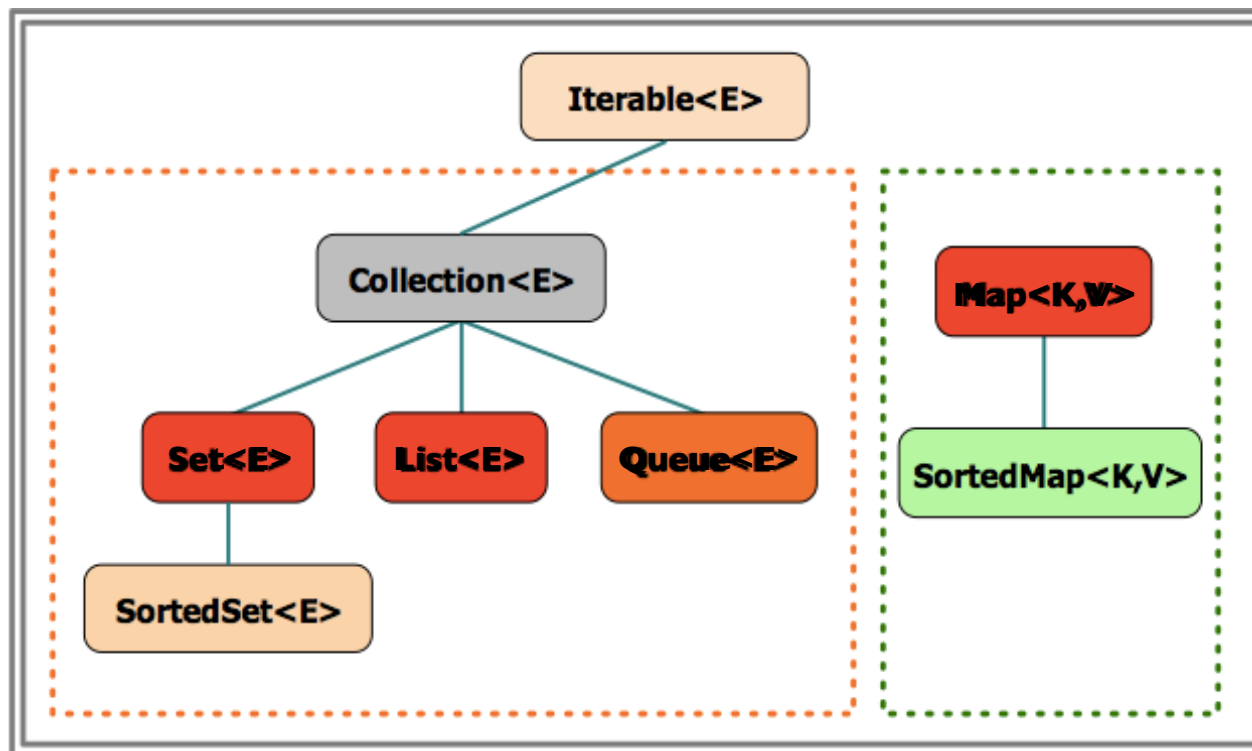
- Ao fazer-se `ArrayList<Aluno>` passa a ser o compilador a testar, e validar, que só são utilizados objectos do tipo `Aluno` no `arraylist`.
- isto dá uma segurança adicional aos programas, pois em tempo de execução não teremos erros de compatibilidade de tipos
- os tipos de dados são verificados em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

JCF

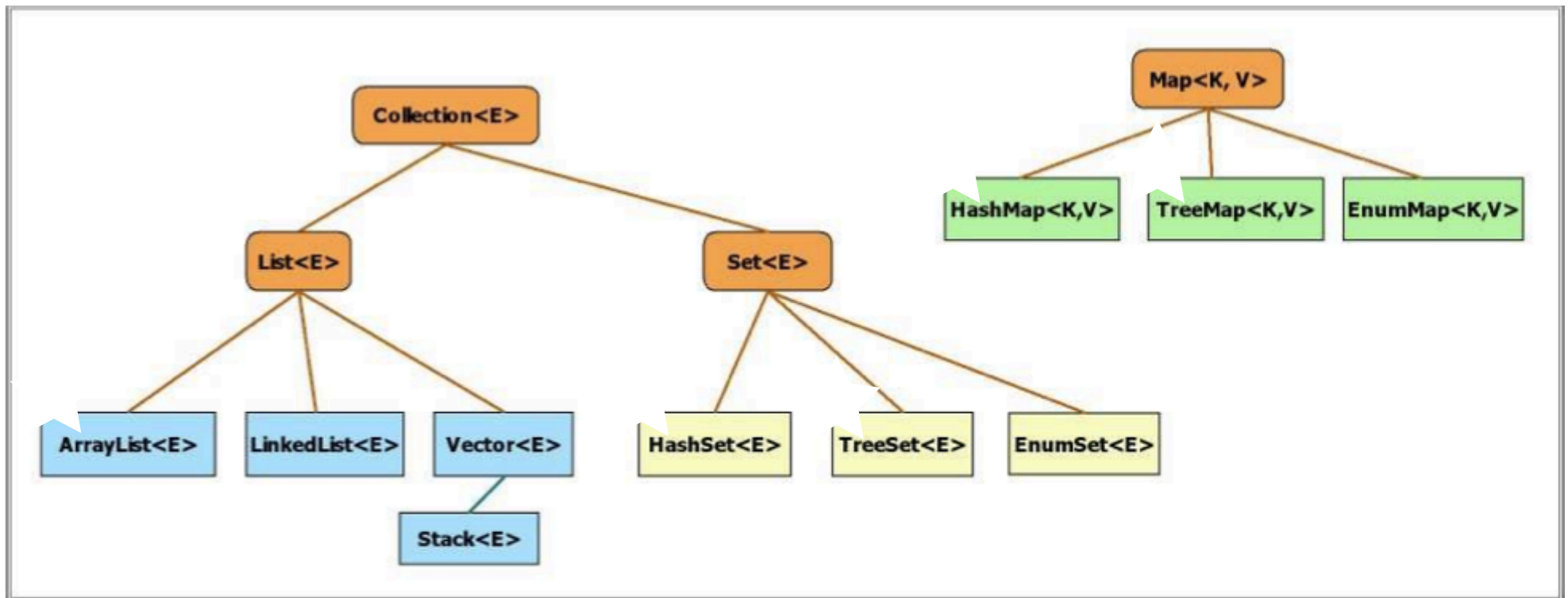
- O JCF (Java Collections Framework) agrupa as várias classes genéricas que correspondem às implementações de referência de:
 - Listas:API de **List<E>**
 - Conjuntos:API de **Set<E>**
 - Correspondências unívocas:API de **Map<K,V>**

Estrutura da JCF

- Existe uma arrumação por API (interfaces)



- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)



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;  
}
```

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
- O método contains verifica sempre o resultado de equals
- Implementação utilizada: **ArrayList<E>**

ArrayList<E>

Categoria de Métodos	API de ArrayList<E>
Construtores	<code>new ArrayList<E>()</code> <code>new ArrayList<E>(int dim)</code> <code>new ArrayList<E>(Collection)</code>
Inserção de elementos	<code>add(E o); add(int index, E o);</code> <code>addAll(Collection); addAll(int i, Collection);</code>
Remoção de elementos	<code>remove(Object o); remove(int index);</code> <code>removeAll(Collection); retainAll(Collection)</code>
Consulta e comparação de conteúdos	<code>E get(int index); int indexOf(Object o);</code> <code>int lastIndexOf(Object o);</code> <code>boolean contains(Object o); boolean isEmpty();</code> <code>boolean containsAll(Collection); int size();</code>
Criação de Iteradores	<code>Iterator<E> iterator();</code> <code>ListIterator<E> listIterator();</code> <code>ListIterator<E> listIterator(int index);</code>
Modificação	<code>set(int index, E elem); clear();</code>
Subgrupo	<code>List<E> sublist(int de, int ate);</code>
Conversão	<code>Object[] toArray();</code>
Outros	<code>boolean equals(Object o); boolean isEmpty();</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());
    }
}

```

Percorrer uma coleção

- Podemos utilizar o foreach 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;
}
```

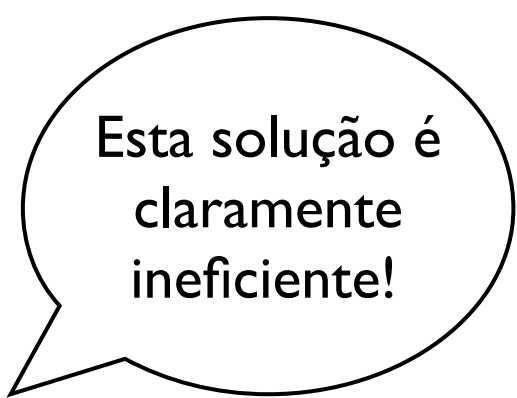
```
public boolean passa() {
    return this.nota >= Aluno.NOTA_PARA_PASSAR;
}
```

Na classe **Aluno**

- ... mas...
- podemos querer parar antes do fim
- podemos não ter acesso and posição do elemento na colecção (no caso dos conjuntos)
- estamos sempre a repetir o código do ciclo

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

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



Esta solução é
claramente
ineficiente!

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

Iteradores externos (Java 5)

- O **Iterator** é um padrão de concepção identificado 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

- Um iterador de uma lista poderia ser:



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

- Iterator API

Method Summary

Methods

Modifier and Type	Method and Description
boolean	<code>hasNext()</code> Returns <code>true</code> if the iteration has more elements.
E	<code>next()</code> Returns the next element in the iteration.
void	<code>remove()</code> 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 (Java 8)

- 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);  
    }  
}
```

- Java 5

```
/**
 * 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);
}
```

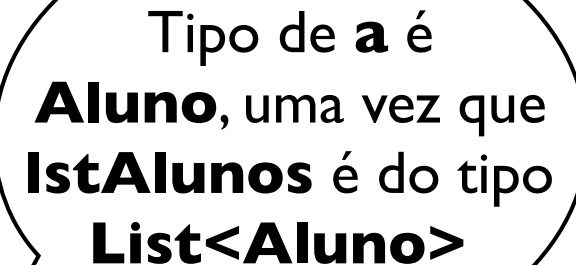
- Java 8 - forEach()

```
/**
 * 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);});
}
```

Expressões Lambda

- **(Tipo p, ...) -> {corpo do método}**
- Um método *anônimo*
- Expressão pode ser simplificada:

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



Tipo de **a** é **Aluno**, uma vez que **lstAlunos** é do tipo **List<Aluno>**

Streams

- Todas as coleções implementam o método **stream()**
- Streams: sequências de valores que podem ser passados numa *pipeline* de operações.
- As operações alteram os valores (produzindo novas Streams ou *reduzindo* o valor a um só)

```
public int quantosPassam() {  
    int qt = 0;  
  
    for(Aluno a: lstAlunos)  
        if (a.passa()) qt++;  
  
    return qt;  
}
```

```
public long quantosPassam() {  
    return lstAlunos.stream().filter(a -> a.passa()).count();  
}
```

- Coleccções implementam método **stream()**
 - Produz uma Stream
- Alguns dos principais métodos da API de **Stream**
 - `allMatch()` - determina se todos os elementos fazem match com o predicado fornecido
 - `anyMatch()` - determina se algum elemento faz match
 - `noneMatch()` - determina se nenhum elemento faz match
 - `count()` - conta os elementos da Stream
 - `filter()` - filtra os elementos da Stream usando um predicado
 - `map()` - transforma os elementos da Stream usando uma função
 - `collect()` - junta os elementos da Stream numa lista ou String
 - `reduce()` - realiza uma redução (fold)
 - `sorted()` - ordena os elementos da Stream
 - `toArray()` - retorna um array com os elemtnos da Stream

- **alguemPassa()** - utilizando Streams...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    return lstAlunos.stream().anyMatch(a -> a.passa());
}
```

```
/**
 * 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();
        if (a.passa())
            alguem = true;
    }
    return alguem;
}
```

Referências a métodos

- Classe::método
 - Permitem referir um método pelo seu nome
 - Úteis nas expressões lambda
 - Objecto que recebe a mensagem está implícito no contexto

```
public boolean alguemPassa() {  
    return lstAlunos.stream().anyMatch(Aluno::passa);  
}
```

- **getLstAlunos()**

```
public List<Aluno> getLstAlunos() {  
    return lstAlunos.stream().map(Aluno::clone).collect(Collectors.toList());  
}
```

```
public List<Aluno> getLstAlunos() {  
    List<Aluno> res = new ArrayList<>();  
  
    for(Aluno a: lstAlunos)  
        res.add(a.clone());  
    |  
    return res;  
}
```


- remover alunos utilizando Streams

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    lstAlunos = lstAlunos.stream()
        .filter(a -> a.getNota() >= nota)
        .collect(Collectors.toList());
}
```

mas...

```
public void removerPorNota(int nota) {
    lstAlunos.removeIf(a -> a.getNota() < nota);
}
```

```
/**
 * 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();
    }
}
```

- Existem Steams Especificas para os tipos primitivos
 - IntStream - **mapToInt(...)**
 - DoubleStream - **mapToDouble(...)**
 - ...
- Alguns dos principais métodos específicos
 - average() - determina a média
 - max() - determina o máximo
 - min() - determina o mínimo
 - sum() - determina a soma

- **media()** - utilizando Streams...

```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = lstAlunos.stream()
                           .mapToDouble(Aluno::getNota)
                           .sum();
    return tot/lstAlunos.size();
}
```

```
public double media() {
    double tot = 0.0;

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

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