

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++;

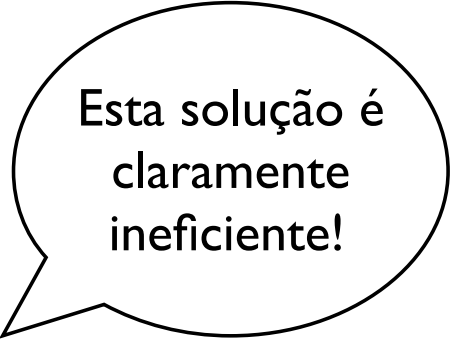
    return qt;
}
```

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

Na classe **Aluno**

- ... mas...
- podemos querer parar antes do fim
- podemos não ter acesso à posição do elemento na colecção (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 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 iterator 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 = colecção.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 elementos 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();
}
```