# Módulo 3 - Interfaces

**Objetivos**

- Entender o benefício das interfaces para a flexibilização e reutilização de software.
- Entender o conceito de referências polimórficas, e seus benefícios.
- Entender o padrão de desenho Estratégia (*Strategy*).
- Saber programar para uma interface.

## Motivação: objetos com comportamento adaptável

O código seguinte apresenta dois procedimentos para ordenar um vetor de inteiros, por ordem crescente (esquerda) e decrescente (direita). O algoritmo de ordenação é talvez o mais naïve que se conhece, o Bubble Sort (mas essa questão não é importante para o que se quer ilustrar). Repare como as implementações são quase iguais, apenas diferem na guarda do if (aliás, apenas diferem num único caracter!). Imaginemos agora que se pretendia outra ordenação de inteiros segundo outro *critério*, por exemplo, pares antecedem os ímpares. (Qual seria a solução? Copiar o código, dar outro nome ao procedimento, e alterar o if?)

```
static void sortAscending(int[] v) {
  for(int i = 0; i < v.length; i++)
    for(int j = 1; j < v.length-i; j++)
      if(v[j] < v[j-1]) {
        int t = v[j-1];
        v[j-1] = v[j];
        v[j] = t;
      }
}
```

```
static void sortDescending(int[] v) {
  for(int i=0; i < v.length; i++)
    for(int j=1; j < v.length-i; j++)
      if(v[j] > v[j-1]) {
        int t = v[j-1];
        v[j-1] = v[j];
        v[j] = t;
      }
}
```

Para números, só há dois critérios de ordenação óbvios (crescente e decrescente). Porém, numa situação onde é necessário ordenar objetos mais complexos (p.e. Alunos com número, nome, idade, etc), podem haver múltiplos critérios de ordenação úteis numa aplicação. Ao recorrer a uma solução de *copy-paste*, estaremos a duplicar muito código, o que não é desejável por si só, bem como a aumentar a chance de introduzir erros neste processo (em engenharia de software há muitos *bugs* derivados de *copy-paste*). Por fim, ainda necessitamos de entender, pelo menos superficialmente, o código do algoritmo para fazer a substituição corretamente.

## Interfaces

As interfaces consistem num mecanismo de programação que ajuda a contornar estes problemas, “externalizando” parte do comportamento de um procedimento/função. No código acima, o que seria desejável era ter um única implementação do algoritmo, onde o cálculo lógico da condição do if fosse um parâmetro. Acontece que tal parâmetro não forneceria simplesmente um valor, mas sim uma função booleana que decide se `v[j]` deve estar antes de `v[j-1]`. Como em Java, ao contrário de outras linguagens (p.e. funcionais), não podemos passar uma função como parâmetro, a solução consistirá em ter um objeto que disponibiliza a função pretendida. Porém, queremos permitir diferentes objetos (funções de decisão distintas), e para isso podemos definir um tipo de objeto abstrato, que chamaremos de *interface*.

As interfaces têm um nome (o tipo abstrato que representam), e definem de operações (implicitamente públicas) sem fornecer a sua implementação. A ideia é a interface ser uma especificação que declara quais as operações que os objetos compatíveis terão que ter disponíveis. Para o exemplo dado, faria sentido a seguinte interface para representar um critério de ordenação. A operação booleana, dados os inteiros a e b, devolverá verdadeiro caso a seja considerado anterior a b. Porém, apenas vemos uma declaração, sem implementação, a qual ficará a cargo de classes que se declarem compatíveis com a interface.

```
interface SortingPolicy {
  boolean isBefore(int a, int b);
}
```

Para definir uma classe compatível com determinada interface, terão que ser disponibilizadas implementações para todas as operações declaradas pela mesma, bem como declarar que a classe é compatível com a interface através da diretiva *implements*. Uma classe pode implementar várias interfaces (`implements InterfaceA, InterfaceB, ...`), embora isso não seja o mais comum. A classe seguinte é compatível com a interface apresentada, implementando o critério de ordenação crescente (a vem antes de b se a < b).

```
class Ascending implements SortingPolicy {
  public boolean isBefore(int a, int b) {
    return a < b;
  }
}
```

## Polimorfismo

O polimorfismo é um mecanismo que permite ter uma forma uniforme para manipular objetos de tipos diferentes. Em termos práticos, isto permite ter uma variável de um tipo à qual podem ser atribuídos objetos de tipos diferentes. Por exemplo:

```
SortingPolicy policy = new Ascending();
```

A variável policy do tipo SortingPolicy é polimórfica, porque lhe podem ser atribuídos objetos de vários tipos (formas) diferentes, desde que compatíveis com a interface. O código que inclua uma invocação de uma operação numa variável polimórfica, por exemplo `policy.isBefore(4, 5)` não tem que depender do tipo do objeto concreto que fornece a implementação da operação. Isto consiste numa grande vantagem para o nosso exemplo, pois podemos ter uma versão do método que implementa a ordenação que é independente do critério de ordenação (pois este é fornecido num parâmetro).

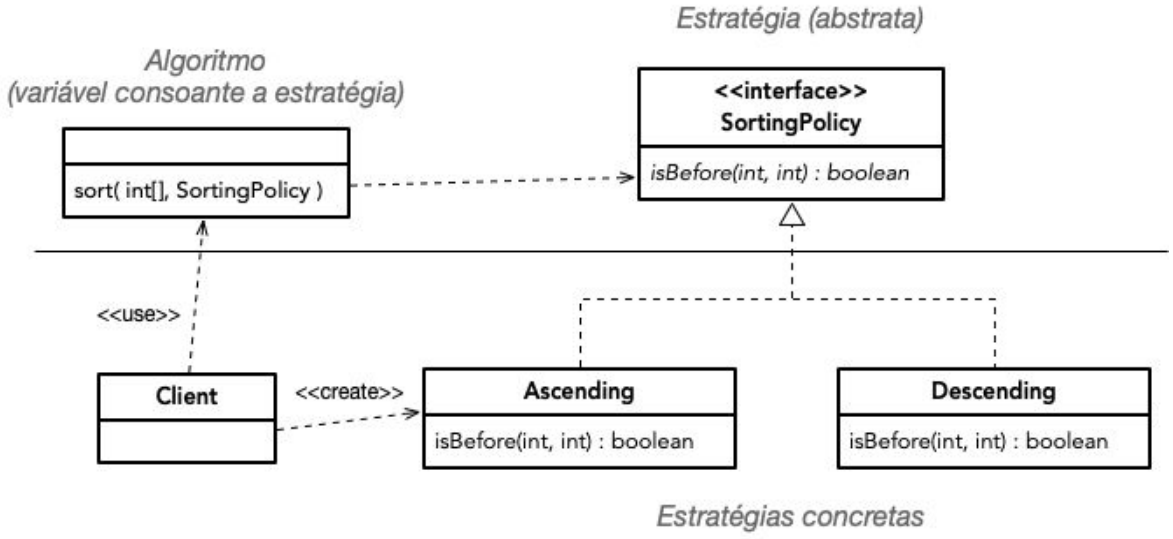
```
static void sort(int[] v, SortingPolicy policy) {
  for(int i = 0; i < v.length; i++)
    for(int j = 1; j < v.length-i; j++)
      if(policy.isBefore(v[j], v[j-1])) {
        int t = v[j-1];
        v[j-1] = v[j];
        v[j] = t;
      }
}
```

Nesta implementação, o parâmetro *policy* aceita qualquer objeto compatível com a interface, ficando a decisão de que objeto utilizar a cargo de quem invoca. O seguinte código invoca o método utilizando o critério de ordenação crescente (definido na classe apresentada anteriormente).

```
int[] array = {6, 3, 2, 7, 4};
sort(array, new Ascending());       // [2, 3, 4, 6, 7]
```

## Padrão Estratégia (*Strategy* ou *Policy*)

O exemplo descrito acima é um caso de aplicação do Padrão Estratégia, muito frequente em desenvolvimento com objetos. A ideia chave consiste no facto de haver um algoritmo, que varia consoante determinada estratégia (no exemplo, critério de ordenação), representada abstratamente por uma interface. Abstratamente, e não concretamente, porque a interface apenas declara *qual* a operação que deve estar disponível, mas não *como* é que essa operação se comporta. As estratégias concretas implementadas em classes, podem ser múltiplas, e não são do conhecimento do algoritmo. Isto significa que o código cliente que utiliza o algoritmo, pode fazê-lo fornecendo qualquer objeto compatível com a interface.



## Exercício 3.1 - ArrayList ordenado

Desenvolva uma classe que representa um vetor de inteiros que está sempre ordenado de acordo com um critério variável (representado pela interface SortingPolicy acima). A inserção de cada elemento não será necessariamente na cauda do vetor, mas sim numa posição que mantenha o vetor ordenado. Ao executar o código seguinte, que insere inteiros na ordem {1, 4, 2, 8}, obteríamos os elementos na ordem {8, 4, 2, 1} (descendente).

```
SortedIntArray sortedArray = new SortedIntArray(new Descending());
sortedArray.add(1);
sortedArray.add(4);
sortedArray.add(2);
sortedArray.add(8);
for(int i = 0; i < sortedArray.size(); i++)
  System.out.println(sortedArray.get(i));

> 8
> 4
> 2
> 1
```

Ao criar um objeto SortedIntArray é passado um objeto compatível com a interface SortingPolicy. Esse objeto é guardado, por forma a que no método de inserção (add) seja possível determinar em que posição será guardado o novo elemento. Desta forma, os elementos estarão sempre por ordem após cada inserção. A implementação a desenvolver pode ser concretizada à custa de um ArrayList para guardar os elementos. Será útil utilizar a operação add(index, element) para adicionar numa posição que não seja a cauda.

```
public class SortedIntArray {
  private ArrayList<Integer> array;
  private SortingPolicy policy;

  public SortedIntArray(SortingPolicy policy) {
    this.array = new ArrayList<>();
    this.policy = policy;
  }

  public int size() { ... }
  public void add(int element) { ... }
  public int get(int index) { ... }
}
```

## Interface java.util.Comparator

O Java utiliza na sua biblioteca a interface java.util.Comparator com um propósito similar à interface do exemplo anterior (SortingPolicy), porém um pouco mais elaborada. Por um lado, a interface é genérica (adaptável para vários tipos, não apenas para inteiros), e por outro, a comparação de elementos também pode indicar que os mesmos são equivalentes.

```
public interface Comparator<T> {
  int compare(T first, T second);
}
```

A interface define a operação *compare*, que dados dois elementos, retorna a sua relação de ordem através de um inteiro com valor: *negativo*, quando primeiro elemento é anterior ao segundo; *zero*, quando os dois elementos são equivalentes; *positivo*, quando o segundo elemento é anterior ao primeiro.

| *Retorno de compare(first, second)* | *negativo* | *zero* | *positivo* |
|---|---|---|---|
| *Relação entre first e second* | first < second | first = second | first > second |

O <T> representa um parâmetro de tipo (genérico), e ao ser definido numa implementação da interface por um tipo concreto C vai implicar que os elementos do código do tipo T serão considerados do tipo C. A classe seguinte implementa um comparador para Strings, onde o critério é o comprimento das mesmas, sendo que as Strings maiores antecedem as menores.

```
public class StringLengthComparator implements Comparator<String> {
  public int compare(String first, String second) {
    return second.length() - first.length();
  }
}

StringLengthComparator comp = new StringLengthComparator();
int tie = comp.compare("POO", "PCD");   // 0 (3 - 3), empate
int c = comp.compare("POO", "IP");    // -1 (2 - 3), "POO" é mais prioritária
```

A interface Comparator é largamente utilizada nas bibliotecas do Java, sempre que é necessário especificar critérios de ordem/prioridade. Por exemplo, em algoritmos de ordenação de vetores e listas, e em estruturas de dados para filas prioritárias e conjuntos ordenados (estas abordadas mais à frente). Desta forma, torna-se essencial entender o propósito e lógica desta interface para que se consiga utilizar eficaz e corretamente as bibliotecas do Java.

## Exercício 3.2 - Comparador de alunos

Considerando uma classe Aluno com atributos de acordo com o seguinte esboço:

```
class Aluno {
  int numero;
  String nome;
  int anoMatricula;
  int anoNascimento;
  ...
}
```

Escreva a classe completa e desenvolva comparadores relativos a uma ordem:

1. Crescente por número;

2. Crescente por nome, alfabeticamente;

   **Dica:** a classe String tem uma constante String.CASE_INSENSITIVE_ORDER que é referente a um comparador de String por ordem alfabética crescente.

3. Decrescente por ano de nascimento (mais novos primeiro), desempatando com o ano de matrícula (mais recentes primeiro), e por fim segundo o critério (2).

Teste os comparadores ordenando vetores e ArrayList de Aluno.

A classe java.util.Arrays contém métodos auxiliares para manipular vetores, por exemplo, métodos de cópia, pesquisa binária, impressão, e ordenação.

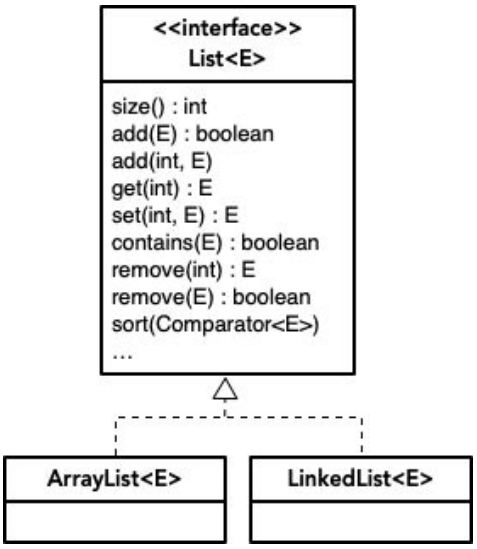
```
Comparator<Aluno> comp = ...
Aluno[] array = ...
Arrays.sort(array, comp);
```

A classe ArrayList tem um método disponível para ordenar os seus elementos de acordo com um comparador.

```
ArrayList<Aluno> list = ...
list.sort(comp);
```

## Interface java.util.List

As bibliotecas do Java definem várias interfaces como abstrações de estruturas de dados, como por exemplo listas, conjuntos, filas, etc. Por agora, foquemo-nos na interface para representar listas, java.util.List<E>. Esta abstração representa uma sequência de elementos, os quais podem ser manipulados através de um índice de base zero. As principais operações da interface apresenta-se em baixo.




**size()** devolve o número de elementos da lista.

**add(e)** adiciona o elemento **e** à lista (devolve sempre verdadeiro).

**add(i, e)** adiciona o elemento **e** à lista no índice **i**.

**get(i)** retorna o elemento contido no índice **i**.

**set(i, e)** substitui pelo elemento **e**, o elemento no índice **i** (devolvendo o elemento que lá estava).

**contains(e)** verifica se existe o elemento **e** na lista.

**remove(i)** remove (e devolve) o elemento no índice **i**.

**remove(e)** remove a primeira ocorrência do elemento **e** da lista, devolvendo verdadeiro caso exista (e seja removido).

**sort(c)** ordena a lista de acordo com o comparador **c**.

A classe ArrayList que tem sido abordada é uma das implementações desta interface. Outra classe que a implementa, que mencionamos agora apenas a título de exemplo (sem detalhes), é a java.util.LinkedList (lista ligada). Ambas as classes representam listas, porém concretizam-nas de formas distintas. ArrayList é baseada num vetor, como explicado anteriormente, ao passo que a Linked List é baseada em nós ligados. Embora sejam classes distintas, relacionam-se através da interface List (polimorfismo), implicando que têm as mesmas operações disponíveis.

Desta forma, ao ter uma referência do tipo List, a mesma pode apontar tanto para um objeto ArrayList, LinkedList, ou outro de outra classe compatível (i.e. que implementa List). Este é o benefício do polimorfismo, especialmente importante nos tipos dos parâmetros dos métodos (flexibilizando a sua utilização), bem como do retorno dos mesmos (facilitando a evolução da sua implementação).

## Métodos com parâmetros e retorno polimórficos

Considere o seguinte exemplo de função, *propositadamente escrita de forma naïve*, que recebe uma lista de palavras e um prefixo, e devolve uma nova lista contendo todas as palavras que têm o prefixo dado, respeitando a ordem original. Embora a função esteja correta em termos do resultado, a função é desnecessariamente pouco flexível, dada a escolha do tipo do parâmetro *list* e do tipo de retorno.

```
LinkedList<String> filterWordsPrefix(ArrayList<String> list, String prefix) {
  LinkedList<String> filtered = new LinkedList<>();
  for(String word : list)
    if(word.startsWith(prefix))
      filtered.add(word);
  return filtered;
}

LinkedList<String> words = ...      // lista de grande dimensao
ArrayList<String> wordsArray = ...  // copia de words
LinkedList<String> wordsFiltered = filterWordsPrefix(wordsArray, "qqcoisa");
```

O facto do parâmetro ser do tipo ArrayList força o código cliente a converter (copiando) a lista de palavras que tinha numa LinkedList (que poderia ser o adequado para o problema) numa ArrayList. Perante um volume de dados de grande dimensão, dada o custo da cópia, mais valeria implementar outra função com o parâmetro do tipo LinkedList. Porém, iríamos ter uma função praticamente igual (duplicação de código), o que não é desejável.

Acontece que as operações utilizadas pela função são operações da interface List, e não específicas de ArrayList. Por outro lado, a utilização de LinkedList como lista de retorno também não é a melhor (ocupa mais espaço, sem oferecer neste caso nenhum benefício). Porém, alterar esta lista para ArrayList iria quebrar o código cliente (erro de incompatibilidade na variável *wordsFiltered*). A versão da função em baixo, com tipos mais abstratos, evitava todos estes problemas, pois a LinkedList podia ser passada diretamente sem cópia, e sendo List o tipo de retorno inicialmente permitia a alteração interna sem quebrar o código cliente.

```
List<String> filterWordsPrefix(List<String> list, String prefix) {
  ArrayList<String> filtered = new ArrayList<>();
  for(String word : list)
    if(word.startsWith(prefix))
      filtered.add(word);
  return filtered;
}
```

## Exercício 3.3 - Lista (polimórfica) de Strings para texto

Quando pretendemos converter uma lista de Strings para texto, utilizando um separador entre palavras (p.e. vírgula), somos forçados a fazer um código repetitivo, ter em conta o tratamento dos separadores, etc. Desenvolva uma função que dadas uma lista (polimórfica) de Strings e uma String para indicar o separador, retorne uma String contendo os elementos separados com o separador dado. Teste a função com os dois tipos de lista, ArrayList e LinkedList.

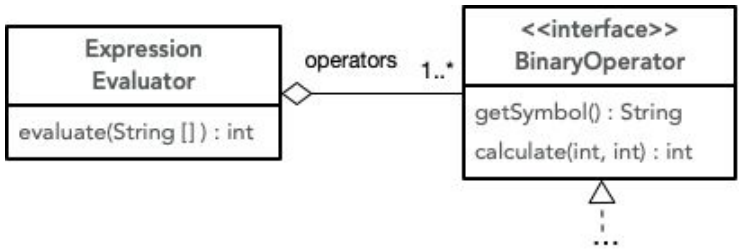
```
List<String> list = ...                    // ["a", "b", "c"]
String text = separatedBy(list, ", ");     // "a, b, c"
```

## Exercício 3.4 - Avaliador de expressões adaptável

Neste exercício a ideia é abordar novamente o problema do Exercício 1.5, desta vez à luz das interfaces. Esse exercício deverá ser resolvido previamente, antes de abordar este. A solução para o exercício original, na parte relativa aos cálculos com os diferentes operadores, teria provavelmente um pedaço de código análogo ao seguinte:

```
if(operator.equals("+"))
  result = a + b;
else if(operator.equals("-"))
  result = a - b;
...
```

Este código não tem nada de errado, mas está moldado para os operadores (+, -, x, /). Para acrescentar operadores, ou usar um conjunto diferente, teríamos que entender parte da implementação do algoritmo e modificar esta parte código. Neste exercício pretende-se fazer uma versão do avaliador de expressões que seja adaptável a quaisquer conjuntos de operadores binários (que atuam sobre dois inteiros). Por exemplo, poderíamos ter o operador '^' para a potência, ou os operadores lógicos *and*, *or*, e *xor* (atuando sobre inteiros, zero: falso, positivos: verdadeiro). A solução passará por abstrair a noção de operador binário para uma interface, e permitir configurar a calculadora com um conjunto de objetos compatíveis. Ao encontrar um símbolo, a operação de cálculo será delegada no objeto correspondente.




```
List<BinaryOperator> operators = new ArrayList<>();
operators.add(new Addition());
operators.add(new Power());
ExpressionEvaluator calc = new ExpressionEvaluator(operators);
int r = calc.evaluate("(", "(", "2", "^", "8", ")", "+", "2", ")");  // 258
```