## 1. Introdução teórica

### Interfaces

Interface, no contexto de Programação, é um recurso de **Projeto Orientado a Objetos** que descreve, antes de se codificar uma determinada classe, como essa classe deverá apresentar-se a outras classes, expondo quais operações ela permitirá que essas outras classes tenham acesso.

Portanto, uma interface é como um protótipo de uma ou mais classes, e especifica quais **métodos públicos** deverão ser codificados nas classes que implementem (sigam) esse protótipo. A interface, portanto, é como um contrato que estabelece as obrigações (de codificação) das classes que aderirem a essa interface. Portanto, todos os métodos declarados em uma interface deverão ser implementados nas classes que adiram à interface.

Definir interfaces é uma fase do processo de projetar um software, antes mesmo de se saber quais classes serão efetivamente modeladas. Assim, nas fases iniciais do projeto de software, pensa-se sobre o que deverá ser realizado pelas classes (quais métodos as futuras classes deverão possuir para serem usados por outras classes e por aplicações) e não sobre como cada método deverá ser codificado internamente (já que há várias maneiras de se codificar cada método e, no início, saber quais técnicas serão usadas no código é algo ainda nebuloso).

Python não possui o conceito de interfaces. Ao contrário, Python possui a possibilidade de definir algo semelhante através do uso de Classes Abstratas (Como foi explicado no Capítulo 14 da apostila de Python).

Já em Java e em C#, uma interface descreve um conjunto de métodos que deverão ser codificados nas classes que implementarem a interface.

Uma interface é declarada através da palavra reservada **interface** e contém somente constantes e métodos abstratos. Todos os membros declarados na interface devem ser públicos e nenhum detalhe de codificação dos métodos pode ser implementado. A implementação dos métodos será feita nas classes que **aderirem à interface**, ou seja, o código interno de cada método previsto na interface será feito pelas classes que **implementarem** tal interface. Portanto, cada classe que aderir a uma interface poderá implementar os métodos previstos de maneira diferente de outras classes, usando recursos e técnicas mais adaptados à realidade que cada classe representa.

### Uma aplicação de Interface

Pensemos em classes de manutenção de coleções de registros (dados relacionados). Existem diferentes implementações dessas classes. Por exemplo, usando vetores de objetos lidos de arquivos ou de tabelas de bancos de dados (como já estudamos), ou usando outras estruturas, como listas ligadas e árvores (que ainda estudaremos).

No entanto, todas essas implementações seguem a mesma diretriz: elas devem possuir métodos para ler e armazenar os dados, pesquisar dados, incluir novos dados, excluir dados existentes, alterar dados, listar dados, ordenar os dados, dentre outras operações. Esses métodos, em geral, sempre terão de ser codificados em situações de manutenção de registros. No entanto, a forma de cada estrutura de dados implementar (codificar) tais métodos já previstos varia muito, pois depende dos recursos e limitações de cada estrutura de dados e da realidade de cada classe.

No entanto, podemos usar uma interface para definir quais métodos uma classe de manutenção de coleções de registros deverá implementar, antes mesmo de decidirmos sobre qual estrutura de dados concreta usaremos nas fases posteriores do processo de projeto orientado a objetos.

Pensemos nas classes de manutenção de vetores e na classe abstrata VetorDados que estudamos no 1º semestre, em Python. Todas aquelas classes possuíam métodos semelhantes e que tiveram de ser implementados em cada uma delas. Portanto, essas classes **seguem a mesma interface**, que poderíamos chamar de **ManterDados**. Vamos lembrar dos métodos que codificamos para essas classes, e que toda classe de manutenção de vetores de objetos deveria ter:

```
// os métodos definidos abaixo são necessários em
// qualquer classe de manutenção de vetor de dados
void lerDados(String nomeArquivo) throws FileNotFoundException, Exception;
void gravarDados(String nomeArquivo) throws IOException;
```

```
void incluirNoFinal(Tipo novoDado);
void incluirEm(Tipo novoDado, int posicaoDeInclusao) throws Exception;
void excluir(int posicaoDeExclusao);   // remove dado dessa posição
Tipo valorDe(int posicaoDeAcesso);     // retorna dado do índice posicaoDeAcesso
void alterar(Tipo novosDados, int posicaoDeAlteracao);
boolean existe(Tipo procurado);    // informa se procurado está ou não no vetor
void ordenar();                  // ordena o vetor com os dados
boolean estaVazio();           // informa se vetor com os dados está ou não vazio
Situacao getSituacao();          // retorna situação atual do programa
void setSituacao(Situacao novaSituacao);    // muda situação do programa
int getTamanho();            // retorna tamanho físico do vetor com os dados

// métodos necessários para navegação dentro do vetor com os dados
boolean estaNoInicio();
boolean estaNoFim();
void irAoInicio();
void irAoFim();
void irAoAnterior();
void irAoProximo();
```

**Tipo**, citado acima, é a classe específica que representa a entidade cujos dados serão armazenados no vetor de dados da classe que implementar os métodos acima.

**Situacao**, citada acima, é uma enumeração (ou tipo enumerado). É uma coleção de palavras que podemos usar em uma classe para tornar mais inteligível o significado de alguma variável. No caso de uma classe de manutenção de vetor de dados, as situações pelas quais ele passa podem ser descritas pelo enumerador declarado abaixo:

```
enum Situacao {navegando, incluindo, excluindo, alterando, editando, buscando}
```

Uma interface de manutenção de dados pode ser declarada usando os métodos acima. Observe que essa interface informa os métodos que qualquer tipo de estrutura de armazenamento (vetor, lista ligada, árvore binária) necessitará implementar para que possa ser usada em um programa de manutenção de dados, independentemente da estrutura específica que se escolher, futuramente, para armazenar os dados. Portanto, interfaces servem como um recurso de **projeto de software** que permite unificar o vocabulário a ser usado em classes, permite unificar as ideias e estabelece uma diretriz para nortear a codificação das futuras classes. Outra vantagem de usar interfaces na organização do projeto é que, como todas as classes de manutenção de dados deverão ter os mesmos métodos públicos, pois seguem a mesma interface, é possível trocar a classe que implementa essa interface por outra classe que também a implementa, sem ter que alterar o código da aplicação, já que os métodos da nova classe têm as **mesmas assinaturas** que os métodos da classe substituída.

Para a manutenção de registros de Estudantes, poderíamos declarar uma interface como a que se segue:

```
public interface ManterDados {
  enum Situacao {navegando, incluindo, excluindo, alterando, editando,
                 buscando}
  // os métodos definidos abaixo são necessários em
  // qualquer classe de manutenção de vetor de dados
  void lerDados(String nomeArquivo) throws FileNotFoundException, Exception;
  void gravarDados(String nomeArquivo) throws IOException;
  void incluirNoFinal(Estudante novoDado);
  void incluirEm(Estudante novoDado, int posicaoDeInclusao) throws Exception;
  void excluir(int posicaoDeExclusao);        // remove estudante dessa posição
  Estudante valorDe(int posicaoDeAcesso);     // retorna estudante do índice
  posicaoDeAcesso
  void alterar(Estudante novosDados, int posicaoDeAlteracao);
  boolean existe(Estudante procurado);    // informa se procurado está ou não
  no vetor
  void ordenar();                 // ordena o vetor com os dados
```

```
    boolean estaVazio();            // informa se vetor com os dados está ou não
    vazio
    Situacao getSituacao();         // retorna situação atual do programa
    void setSituacao(Situacao novaSituacao);    // muda situação do programa
    int getTamanho();           // retorna tamanho físico do vetor com os dados

    // métodos necessários para navegação dentro do vetor com os dados
    boolean estaNoInicio();
    boolean estaNoFim();
    void irAoInicio();
    void irAoFim();
    void irAoAnterior();
    void irAoProximo();

    int getOnde();       // retorna o índice em que a pesquisa binária parou
}
```

O código acima seria gravado em um arquivo chamado ManterDados.java, e incorporado a um projeto de aplicação Java.

A partir dele, podemos desenvolver uma classe ManterEstudantes, que implementa os métodos descritos na interface e, também, outros métodos específicos dessa classe. Por exemplo:

```
import java.io.*;
import static java.lang.System.out;

public class ManterEstudantes implements ManterDados {
    int qtosDados,              // tamanho lógico do vetor interno dados
        onde,                  // índice onde a pesquisa binária parou
            posicaoAtual;           // índice do registro atualmente visitado
        Estudante[] dados;          // vetor interno que armazena os registros
        Situacao situacao;          // o que a aplicação está realizando agora

    public void lerDados(String nomeArquivo) throws FileNotFoundException,
    Exception
    {
    ...
    }

    @Override
    public void gravarDados(String nomeArquivo) throws IOException
    {
    ...
    }

    public Boolean existe(Estudante dadoProcurado) { ... }
    public void incluirNoFinal(Estudante novoDado) { ... }
    public void incluirEm(Estudante novoDado, int posicaoDeInclusao) { ... }
    public void excluir(int posicaoDeExclusao) { ... }
    public Estudante valorDe(int indiceDeAcesso) { ... }
    public void alterar(int posicaoDeAlteracao, Estudante novoDado) { ... }
    public void trocar(int origem, int destino) { ... }
    public void ordenar() { ... }
    public Boolean estaVazio() { ... }
    public Boolean estaNoInicio() { ... }
    public Boolean estaNoFim() { ... }
    public void irAoInicio() { ... }
    public void irAoFim() { ... }
    public void irAoAnterior() { ... }
    public void irAoProximo() { ... }
    public int getPosicaoAtual() { ... }
    public void setPosicaoAtual(int novaPosicao) { ... }
    public Situacao getSituacao() { ... }
    public void setSituacao(Situacao novaSituacao) { ... }
```

```
}
```

O código acima foi praticamente copiado do projeto CadastroEstudantes, que estudamos em aula, e utiliza também métodos codificados na classe Estudante, como leuLinhaDoArquivo() e formatoDeArquivo().

## 2. Descrição do projeto

Esse projeto deve ser desenvolvido em dupla, em linguagem Java, usando o **IDE IntelliJ Idea**, para manter a uniformidade na correção.

Ele deve aprimorar o projeto CadastroEstudantes, desenvolvido em aula, com as opções de seletor já existentes, e adicionando algumas novas operações, que serão descritas abaixo.

No início da execução do programa, deve-se ler um arquivo texto contendo siglas (6 caracteres) de até 15 disciplinas e armazená-las em um vetor de strings. Cada posição desse vetor conterá a sigla da disciplina correspondente à mesma posição do vetor notas da classe Estudante. Por exemplo:

| | 0 | 1 | 2 | 3 | ... | 14 |
|---|---|---|---|---|---|---|
| Siglas | Biolog | Matema | Portug | TecPro | ... | |

| | | | | | notas | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | curso | ra | nome | qn | 0 | 1 | 2 | 3 | ... | 14 |
| 0 | 19 | 24105 | Maria | 4 | 8.5 | 10.0 | 7.2 | 7.5 | | |
| 1 | 39 | 24110 | João | 4 | 3.5 | 8.0 | 5.6 | 8.0 | | |
| 2 | 19 | 24203 | Renata | 4 | 4.8 | 7.2 | 6.5 | 5.8 | | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | | |

As operações 3 e 4, já codificadas no projeto base, devem passar a exibir cabeçalhos contendo as siglas das disciplinas acima de cada coluna de notas.

Deve-se adicionar as seguintes operações ao seletor do programa:

**5 – Alterar estudantes**

Deve ser repetitiva, até que o usuário digite um RA igual a 0, de maneira semelhante ao que está feito nas opções anteriores. Deve-se mostrar os dados do RA digitado e solicitar ao usuário que digite novamente os dados de cada atributo. Para não alterar algum dos atributos, o usuário deverá digitar [Enter] ao invés de uma string com o novo valor e o programa deverá tratar essa digitação e seu significado. Deve-se permitir a alteração de todos os dados armazenados nesse Estudante, menos seu RA, que é chave primária.

**6 – Ir ao início**

**7 – Ir ao próximo**

**8 – Ir ao anterior**

**9 – Ir ao último**

As operações acima servem para exibir na tela os registros armazenados no vetor interno dados de ManterEstudantes, de forma sequencial, ou seja, permitem a navegação entre os registros e sua exibição na tela. Devem ser implementadas pela chamada do respectivo método de ManterEstudantes e a exibição do registro armazenado no índice definido pelo atributo posicaoAtual de ManterEstudantes.

## 10 – Estatísticas:

- Disciplina com maior número de estudantes com aprovação
- Disciplina com maior número de estudantes com retenção
- Qual estudante teve a maior média de notas?
- Desse estudante, quais as disciplinas com maior e menor notas?
- Média aritmética dos alunos em cada disciplina
- Na disciplina com a menor média, qual foi o aluno com a maior nota?
- Na disciplina com a maior média, qual foi o aluno com a menor nota?

Essas funcionalidades da operação 10 **não devem** ser implementadas na classe ManterEstudantes, e sim na classe do programa principal, pois essas estatísticas são necessidades dessa aplicação específica, e não da classe de armazenamento de dados em si. Para acessar os dados do objeto ManterEstudantes (estud) e realizar os cálculos, devem-se usar os métodos disponíveis em ManterEstudantes.