

## Ficha de trabalho #6

---

**Agregação**  
**Composição**  
**ArrayLists**  
**Método clone()**

---

As relações entre classes podem ser de agregação e de composição (entre outros tipos já referidos – ficha #5).

Uma **agregação** ocorre quando uma classe (agregadora) contém elementos de outra classe (objetos agregados). Numa agregação os objetos agregados têm existência independente do objeto agregador, ou seja: o objeto agregado não pertence ao objeto agregador e pode existir após eliminação do objeto agregador. A seguir apresenta-se um exemplo de uma agregação:



Podemos representar uma figura (classe agregadora) como uma classe que contém (agrega) objetos do tipo Ponto (os vértices da figura). Os pontos não pertencem à figura e podem continuar a existir após a sua eliminação.

Uma forma de implementar a agregação neste exemplo seria a existência de um array de pontos dentro da classe Figura. No entanto, a utilização de arrays implica sabermos o número de elementos. Para contornar esta limitação, podemos utilizar um tipo de dados, que também permite armazenar conjuntos de objetos do mesmo tipo: a classe **ArrayList**, que tem algumas funções pré-definidas:

```
ArrayList<Ponto> pontos = new ArrayList(); //Cria um ArrayList de objetos do tipo Ponto chamado pontos
pontos.size(); //Retorna o tamanho do ArrayList pontos
pontos.add(p1); //Adiciona o Ponto p1 ao ArrayList pontos
pontos.clear(); //Limpa o ArrayList pontos
pontos.contains(p1); //Verifica se o ArrayList pontos contém o ponto p1
pontos.indexOf(p1); //Retorna o índice do Ponto p1 no ArrayList pontos
pontos.isEmpty(); //Verifica se o ArrayList pontos está vazio
pontos.remove(0); //Remove o elemento no índice 0 do ArrayList pontos
pontos.remove(p1); //Remove o Ponto p1 do ArrayList pontos
pontos.get(0); //Retorna o elemento no índice 0 do ArrayList pontos
//Percorre o ArrayList pontos e mostra cada um dos seus elementos:
for(Ponto p: pontos) {
    System.out.println(p);
}
```

Pode tornar-se útil termos um método clone na nossa classe, com a seguinte estrutura:

```
public Ponto clone() {
    return new Ponto(this);
}
```

Clona o ponto, chamando o construtor de cópia da classe ponto, com a palavra reservada this (cria um clone deste ponto)

A classe `Figura` podia ter a estrutura seguinte:

```
public class Figura {
    private String nome;
    private ArrayList<Ponto> vertices;

    private static final String NOME_OMISSAO = "Não definido";

    public Figura() {
        this.nome = NOME_OMISSAO;
        this.vertices = new ArrayList<Ponto>();
    }

    public Figura(String nome) {
        this.nome = nome;
        this.vertices = new ArrayList<Ponto>();
    }

    public Figura(Figura f) {
        this.nome = f.nome;
        this.vertices = new ArrayList<Ponto>();
        for(Ponto v: f.vertices) {
            this.vertices.add(new Ponto(v));
        }
    }

    public Figura clone() {
        return new Figura(this);
    }

    public String getNome() {
        return this.nome;
    }

    public ArrayList<Ponto> getVertices() {
        ArrayList<Ponto> novos = new ArrayList<>();
        for(Ponto v: this.vertices) {
            novos.add(new Ponto(v));
        }
        return novos;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setVertices(ArrayList<Ponto> vertices) {
        this.vertices = new ArrayList<Ponto>();
        for(Ponto v: vertices) {
            this.vertices.add(new Ponto(v));
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Figura other = (Figura) obj;
        if (!this.nome.equals(other.nome)) {
            return false;
        }
        for(Ponto p: this.vertices) {
            if (!other.vertices.contains(p)) {
                return false;
            }
        }
        return true;
    }

    @Override
    public String toString() {
        String descricao = "Figura " + this.nome + "com vertices:";
        for(Ponto p: this.vertices) {
            descricao += "\n- " + p;
        }
        return descricao;
    }

    public void adicionarVertice(Ponto v) {
        this.vertices.add(new Ponto(v));
    }
}
```

A classe `Figura` tem como atributo uma lista de pontos chamada `vértices`.

Tem os **construtores**:

1. Vazio, que preenche com o valor por omissão e cria a lista de `vértices`
2. Completo, que preenche o nome e cria a lista de `vértices`
3. Cópia, que cria a figura por cópia dos atributos de uma figura recebida por parâmetro, criando um novo ponto para cada um dos pontos da figura recebida

**Nota:** o construtor completo neste caso não vai preencher todos os atributos, deixando a lista de `vértices` criada, mas por preencher

Tem ainda os métodos:

- **clone**, que clona uma figura, chamando o construtor de cópia
- **Getters**, que no caso da lista de pontos retorna uma nova lista de pontos com pontos criados por cópia dos pontos existentes na lista de `vértices`
- **Setters**, que no caso da lista de pontos redefine a lista de `vértices`, um a um, por cópia dos recebidos por parâmetro
- **equals**, que compara o nome e todos os `vértices` de duas figuras
- **toString**, que mostra a descrição textual da figura, mostrando todos os seus `vértices`
- **adicionarVertice**, que adiciona um novo `vértice` à lista de pontos, copiando o ponto que recebe por parâmetro

É comum que existam métodos para **adicionar**, **remover** e **editar** componentes da classe agregadora.

Depois de implementarmos a classe figura, temos diversas formas de criar figuras no main, como por exemplo:

```
public class TesteFiguras {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(0,0);
        Ponto p2 = new Ponto(0,1);
        Ponto p3 = new Ponto(1,0);
        Ponto p4 = new Ponto(1,1);

        Figura quadrado = new Figura();
        quadrado.adicionarVertice(p1);
        quadrado.adicionarVertice(p2);
        quadrado.adicionarVertice(p3);
        quadrado.adicionarVertice(p4);
        System.out.println(quadrado);
    }
}
```

- Criar 4 pontos: p1, p2, p3 e p4
- Criar uma figura chamada quadrado a partir do construtor vazio
- Adicionar os pontos como vértices da figura
- Mostrar a descrição textual da figura

```
public class TesteFiguras {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(0,0);
        Ponto p2 = new Ponto(0,1);
        Ponto p3 = new Ponto(1,0);

        Figura t = new Figura("triangulo");
        t.adicionarVertice(p1);
        t.adicionarVertice(p2);
        t.adicionarVertice(p3);
        System.out.println(t);
    }
}
```

- Criar 3 pontos: p1, p2 e p3
- Criar uma figura chamada triângulo através do construtor completo
- Adicionar cada um dos pontos como vértice da figura
- Mostrar a descrição textual da figura

Uma **composição** é uma agregação mais forte, em que a existência do objeto agregado depende da existência do objeto agregador. Por exemplo:



Podemos representar um carro como uma classe a que pertence um objeto do tipo Matricula. A matrícula pertence ao carro e deixa de existir se o carro for eliminado. Ao atribuirmos uma matrícula a um carro não precisamos de criar uma cópia da matrícula recebida, como acontecia para os vértices da figura no exemplo da agregação.

Para além do referido acima, a implementação de uma composição é semelhante à implementação de uma agregação.

1. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar turmas. Uma **turma** é caracterizada pelo ano de escolaridade (int), identificação (String) e contém uma lista de alunos. Cada **aluno** é caracterizado pelo seu nome (String), número de aluno (int, gerado automática e sequencialmente), data de nascimento (Data), número de telefone (String) e freguesia (String). Devem ser implementados nas classes os construtores vazio, completo e de cópia, getters e setters e os métodos clone(), equals(Object obj) e toString(). Devem ainda ser implementados os seguintes métodos:

#### 1.1. Aluno:

- alterarNrTelefone(String novoTel), que altera o número de telefone do aluno
- hasNr(int n), que retorna true caso o nº de aluno seja n e false caso contrário
- hasFreguesia(String freguesia), que retorna true caso a freguesia do aluno seja igual à passada por parâmetro e false caso contrário

#### 1.2. Turma:

- adicionarAluno(Aluno a), que adiciona o Aluno a à turma
- adicionarAluno(String nome, Data dtNascimento, String tel, String freguesia), que adiciona o aluno à turma, com base nas suas características passadas por parâmetro
- adicionarAluno(String nome, int anoN, int mesN, int diaN, String tel, String freguesia), que adiciona o aluno à turma, com base nas suas características passadas por parâmetro
- removerAluno(Aluno a), que remove o aluno a da turma
- removerAluno(int n), que remove da turma o aluno com número de aluno n
- alterarNrTelefone(Aluno a, String novoTel), que altera o número de telefone do Aluno a para novoTel
- alterarNrTelefone(int n, String novoTel), que altera o número de telefone do Aluno com número de aluno n para novoTel
- obterAlunos(String freguesia), que retorna uma lista de alunos cuja freguesia é igual à passada por parâmetro

#### 1.3. Deve ser implementada uma classe de teste com as seguintes funcionalidades:

- Criar alunos
- Criar uma turma
- Adicionar os alunos à turma
- Alterar o número de telefone de um dos alunos (diretamente)
- Alterar o número de telefone de um dos alunos (pelo número de aluno)
- Eliminar um dos alunos da turma (diretamente)
- Eliminar um dos alunos da turma (pelo número de aluno) e mostrar a turma
- Mostrar os alunos que vivem numa determinada freguesia
- Verificar que as alterações diretas não funcionam corretamente caso o método equals de Aluno não esteja implementado

2. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a Javadoc, para representar livrarias. Uma **livraria** é caracterizada por um nome (String), uma morada (String) e a lista de livros existentes. Um **livro** é representado pelo seu título (String), autor (String), ISBN (int) e preço (double).

Devem ser implementados nas classes os construtores vazio, completo e de cópia, getters e setters e os métodos clone(), equals(Object obj) e toString(). Devem ainda ser implementados os seguintes métodos:

#### 2.1. Livro

- aplicarDesconto(double percentagem), que altera o preço do livro, aplicando um desconto (considerar que a percentagem é representada entre 0 e 1  $\rightarrow 10\% = 0.1$ )

#### 2.2. Livraria:

- adicionarLivro(Livro l), que adiciona o livro l à livraria
- adicionarLivro(String titulo, String autor, int isbn, double preco), que adiciona um livro à livraria com base nas suas características
- removerLivro(Livro l), que remove o livro l da livraria
- removerLivro(int isbn), que remove da livraria o livro com o isbn passado por parâmetro
- atribuirDescontoLivro(Livro l, double percentagem), que atribui ao livro l um desconto de uma determinada percentagem
- atribuirDescontoLivro(int isbn, double percentagem), que atribui um desconto de uma determinada percentagem ao livro cujo isbn é igual ao passado por parâmetro
- atribuirDescontoAutor(String nomeAutor, double percentagem), que atribui um desconto de uma determinada percentagem a todos os livros cujo autor tem o nome passado por parâmetro

#### 2.3. Deve ser implementada uma classe de teste para testar as funcionalidades implementadas:

- Criar livros
- Criar uma livraria
- Adicionar os livros à livraria
- Remover diretamente um dos livros
- Remover um dos livros pelo ISBN
- Atribuir um desconto diretamente a um livro
- Atribuir um desconto a um livro pelo ISBN
- Atribuir descontos a todos os livros de um autor

3. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a Javadoc, para representar mercearias. Uma **mercearia** é caracterizada por um nome (String), uma morada (String) e uma lista de produtos existentes. Um **produto** é representado pelo nome (String), tipo (String – exemplo: “legume”, “carne”, ...), código de barras (int), preço (double) e quantidade em stock (int – só vendem produtos embalados). Devem ser implementados nas classes os construtores vazio, completo e de cópia, getters e setters e os métodos clone(), equals(Object obj) e toString(). Devem ainda ser implementados os seguintes métodos:

### 3.1. Produto

- calcularPreco(int qtd), que retorna o preço da quantidade pretendida
- retirar(int qtd), que diminui a quantidade de produto vendida
- adicionar(int qtd), que aumenta a quantidade de produto vendida
- isTipo(String t), que retorna true caso o tipo do produto seja t e false caso contrário

### 3.2. Mercearia:

- adicionarProduto(Produto p), que adiciona o produto p ao stock da mercearia
- adicionarProduto(String nome, String tipo, int codBarras, double preco, int quantidade), que adiciona um produto ao stock da mercearia com base nas suas características
- removerProduto(Produto p), que remove o produto p da lista de produtos
- removerProduto(int cod), que remove um produto da lista de produtos pelo seu código de barras
- venderProduto(Produto p, int qtd), que efetua uma venda (diminuição da quantidade em stock) de uma quantidade do produto p
- venderProduto(int cod, int qtd), que efetua uma venda (diminuição da quantidade em stock) de uma quantidade do produto com um determinado código de barras
- comprarProduto(Produto p, int qtd), que efetua uma compra (aumento da quantidade em stock) de uma quantidade do produto p
- comprarProduto(int cod, int qtd), que efetua uma compra (aumento da quantidade em stock) de uma quantidade do produto com um determinado código de barras
- obterProdutos(String t), que retorna uma lista de produtos do tipo t

### 3.3. Deve ser implementada uma classe de teste para testar as funcionalidades implementadas:

- Criar produtos
- Adicionar os produtos à mercearia
- Remover um dos produtos, diretamente e pelo código de barras
- Efetuar uma venda, diretamente e pelo código de barras
- Efetuar uma compra, diretamente e pelo código de barras
- Mostrar os legumes