Classe Abstrata e Interface

Nivelamento de Lógica de Programação e Programação Orientada a Objeto

Classe Abstrata e Interface

Tipos Abstratos e Interfaces

Anteriormente, definimos que os pilares da orientação a objetos são quatro, e já falamos sobre três deles: **abstração**, **encapsulamento** e **herança**.

Nessa aula, vamos abordar o **polimorfismo**.

A abstração, às vezes, pode ser confundida com os tipos abstratos.

Podemos definir uma classe como abstrata, no sentido que não é possível representar instâncias diretas daquela classe.

Em Java, isso implica que até pode haver um construtor, mas apenas para ser herdado por seus subtipos e não para ser invocado diretamente usando `new`.

Classe Abstrata

Considerando o exemplo anterior da aula anterior, da escola, podemos definir que os instrutores serão sempre prestadores de serviço ou do quadro de funcionários.

Vamos chamar de `InstrutorTemporario` e `InstrutorFixo`, respectivamente:

Classe Abstrata

```
abstract class Instrutor extends Pessoa {
    private final String area;
    protected Instrutor(String nome, String area) {
        super(nome);
        this.area = area;
    }
}
class InstrutorTemporario extends Instrutor {
    private final LocalDate dataInicio;
    private final LocalDate dataFim;
    public InstrutorTemporario(String nome, String area, LocalDate dataInicio, LocalDate
dataFim) {
        super(nome, area);
        this.dataInicio = dataInicio;
        this.dataFim = dataFim;
    }
}
class InstrutorFixo extends Instrutor {
    private final LocalDate dataInicio;
    private final String senioridade;
    public InstrutorFixo(String nome, String area, LocalDate dataInicio, String senioridade) {
        super(nome, area);
        this.dataInicio = dataInicio;
        this.senioridade = senioridade;
    }
}
```

Classe Abstrata

Usar classes abstratas permite definir comportamentos comuns no tipo, no entanto, restringindo que os objetos devem ser sempre específicos.

Ou seja, é uma abordagem fortemente centrada em evitar repetição de código.

Classe Abstrata

Usar classes abstratas permite definir comportamentos comuns no tipo, no entanto, restringindo que os objetos devem ser sempre específicos.

Ou seja, é uma abordagem fortemente centrada em evitar repetição de código.

Métodos abstratos

Métodos abstratos

É possível definir apenas a assinatura de um método sem especificar sua implementação ou os detalhes de seu comportamento.

Chamamos esse método de "abstrato" e sua sintaxe é:

```
abstract class Instrutor extends Pessoa {
    abstract String resumoInstrutor();
    //resto do código da classe
}
```

Se há pelo menos um método abstrato, então a classe **DEVE** ser abstrata.

Métodos abstratos

Métodos abstratos são partes fundamentais na definição de contratos de API. Logo, uma classe abstrata define que realiza um comportamento, porém esse comportamento (sua implementação) pode variar de acordo com cada subtipo.

Veja que os subtipos `InstrutorTemporario` e `TemporarioFixo` definem atributos próprios, então o `resumoInstrutor` do primeiro poderia focar no período do contrato enquanto o outro poderia ser a sua senioridade e tempo de vínculo.

Atenção! Quando um tipo define um método abstrato todos os seus subtipos são **OBRIGADOS** a implementá-los.

Interfaces

Interfaces

Uma **interface** define um contrato de API, onde existe um comportamento esperado, mas sem conhecer os detalhes do seu funcionamento.

Essa é a definição estabelecida na POO e que acabou causando certo conflito e estranheza nas últimas versões do Java.

Interfaces

Antigamente, as interfaces em Java definiam apenas assinaturas de métodos.

Em outras palavras, uma interface era uma classe abstrata onde todos os seus métodos são abstratos.

No entanto, no Java 8 surgiu a funcionalidade *"métodos default"*, que são métodos não-abstratos e possuem implementação.

O objetivo dessa funcionalidade era permitir um comportamento padrão para uma funcionalidade do contrato para não quebrar a compatibilidade com os subtipos já existentes.

Interfaces

```
interface Pagavel {
    double calcularSalario();
    default String gerarRecibo() {
        return "Recibo gerado para o valor de: "  + calcularSalario();
    }
}

abstract class Instrutor extends Pessoa implements Pagavel {
    private final BigDecimal salario;
    public BigDecimal getSalario() { return salario; }
    //TODO resto da classe
}

class InstrutorTemporario extends Instrutor {
    //TODO resto da classe
    @Override
    public double calcularSalario() {
        return this.getSalario().multiply(BigDecimal.valueOf(1.3)).doubleValue();
    }
}
```

Quando usar Interface e Classe Abstrata?

- **Use uma interface** quando você precisa definir um contrato para um comportamento que pode ser implementado por várias classes não relacionadas. Pense em um "contrato" de comportamento.

  Exemplo: um Animal pode ter um voar(), e tanto um Pássaro quanto um Morcego (que não estão na mesma hierarquia de herança) podem implementar essa interface.

- **Use uma classe abstrata** quando você precisa compartilhar código e estado entre classes que têm uma relação "é um". Pense em uma classe base para uma hierarquia. Exemplo: Animal como uma classe abstrata que tem um método comer() (com implementação) e um método fazerSom() (abstrato), que classes como Cachorro e Gato devem implementar de forma diferente.

Em resumo, a interface foca no comportamento (o que a classe faz), enquanto a classe abstrata foca na estrutura e no comportamento de uma hierarquia de classes.

Polimorfismo

Polimorfismo

O último fundamento da POO é o **polimorfismo**, que define que uma classe pode ser representada de várias formas, explorando os tipos dos quais ela for subtipo.

Exceto `Object`, todos os tipos em Java possuem duas ou mais representações.

Poderíamos definir uma coleção ou array com todos os subtipos de instrutores representados pelo seu tipo comum `Instrutor`, ou ainda todos os instrutores e alunos representados como `Pessoa`.

Quando aplicamos o polimorfismo ficam visíveis apenas os comportamentos e atributos definidos no tipo da variável.

Polimorfismo

```
void main() {
    Pessoa aluno = new Aluno();
    aluno.getMatricula(); // não está acessível e causa erro de compilação
    Instrutor instrutor = new InstrutorTemporario();
    instrutor.getSalario(); //funciona porque está definido em Instrutor
    instrutor.getDataInicio(); // não está acessível e causa erro de compilação
}
```

Polimorfismo

Baixo acoplamento

O polimorfismo é sempre recomendável quando lidando com interfaces, classes abstratas e herança para promover o baixo acoplamento no projeto - também chamado de "desacoplamento" e "dependência fraca".

Considere o conceito de interface, no sentido amplo.

Pensando numa interface de conexão de vídeo no padrão HDMI, temos muitos dispositivos de saída de imagem e muitos outros de entrada de vídeo.

Todos os que são compatíveis com o padrão HDMI são compatíveis entre si, formando um par entrada e saída.

Dessa forma, não é necessário se restringir a fabricante ou modelo específico, pois qualquer "implementação" do padrão deve funcionar bem.

Polimorfismo

Baixo acoplamento

Da mesma forma, devemos sempre buscar nos apoiar nas interfaces definidas, permitindo liberdade e generalização na construção do projeto de software.

Polimorfismo

Checagem de tipo

Em cenários de polimorfismo pode ser necessário checar o tipo específico de uma variável.

Neste caso, em Java, usamos a palavra-chave instance of.

```
class Financeiro {
    public void realizarPagamento(Pagavel pagavel) {
        if (pagavel instanceof Instrutor) {
            Instrutor instrutor = (Instrutor) pagavel;
            System.out.println(instrutor.resumoContrato());
        }
        System.out.println(pagavel.gerarRecibo());
    }
}
```

Enumeração

Enumeração

Existem situações onde queremos restringir as possíveis instâncias ou valores para uma determinada classe.

Considere o atributo senioridade que definimos em `InstrutorFixo`.

Deixar como campo aberto de texto pode permitir ocorrência de inconsistências ou valores que não condizem com a realidade da escola.

Vamos considerar três níveis definidos como L1, L2 e L3.

Podemos resolver esse problema usando enumerações ("enumerations" ou "enums").

Enumeração

A sintaxe mais simples é :

```
enum Senioridade {
    L1, L2, L3
}
```

Isso pode parecer muito estranho! Mas se prestarmos mais atenção, vamos perceber que esses valores são todas as instâncias possíveis de uma classe chamada Senioridade.

Isso fica mais evidente se definirmos atributos para a enum.

```
enum Senioridade {
   L1("Junior"), L2("Pleno"), L3("Senior");
   private final String descricao;

   Senioridade(String descricao) { this.descricao = descricao; }
   public String getDescricao() { return descricao; }
}
```

Exercícios

Exercício 1: Sistema de Gerenciamento de Biblioteca Parte 2

Evolução do projeto: Implementação de uma interface `Emprestavel` para definir comportamentos comuns em diferentes itens da biblioteca, como livros e revistas.

Criação de classes concretas (Livro, Revista) que implementam essa interface, demonstrando o uso de abstrações para reduzir o acoplamento.

Desafio:

Adicionar uma interface `Catalogavel` para itens que podem ser catalogados na biblioteca, e implementar métodos que permitam a busca e filtragem de itens usando polimorfismo e baixo acoplamento.

Exercício 2: Cadastro e Listagem de Clientes

Cenário: Uma loja virtual precisa cadastrar clientes e listar seus dados.

Dados de Entrada:

- Nome
- Email
- Tipo do cliente (Pessoa Física ou Jurídica)

O que Manipular?

- Criar um modelo que suporta múltiplos tipos de clientes sem alterar a estrutura principal.
- Cada tipo de cliente deve ter comportamentos distintos, como exibição formatada do CPF/CNPJ.
- Listar todos os clientes cadastrados utilizando um método funcional.

Exercício 3: Escolha de Métodos de Pagamento

Cenário: O cliente precisa escolher como deseja pagar a compra.

Dados de Entrada:

- Valor total da compra.
- Método de pagamento escolhido (Cartão de Crédito, Boleto, Pix).

O que Manipular?

- Criar um sistema que aceite diferentes métodos de pagamento sem precisar alterar a lógica de compra.
- Processar o pagamento conforme o método escolhido.
- Exibir mensagens distintas dependendo da forma de pagamento.

Exercício 4.1: Sistema de Logística de Pacotes

Cenário: Projetar um sistema para gerenciar e transportar diferentes tipos de pacotes usando veículos variados.

Requisitos:

- **Pacotes:** Existem diferentes tipos de pacotes, como os frágeis e os pesados. Todos os pacotes devem ter um peso e um destino. O sistema deve garantir que o peso de um pacote possa ser obtido.

- **Veículos:** O sistema deve modelar diferentes veículos de transporte, como caminhões e vans. Cada veículo tem uma capacidade de carga máxima. Eles devem ter a capacidade de "entregar" um pacote, embora a forma como cada um faz a entrega seja diferente. O sistema precisa garantir que um veículo pode ser "abastecido".

- **Status de Entrega:** O sistema deve ser capaz de rastrear o estado de um pacote, que pode ser `EM_TRANSITO`, `ENTREGUE` ou `PENDENTE`.

Continua

Exercício 4.2: Sistema de Logística de Pacotes (Continuação)

Cenário: Projetar um sistema para gerenciar e transportar diferentes tipos de pacotes usando veículos variados.

Seu desafio é:

Usando enum, interface e classe abstrata, projete as classes necessárias para que, no seu programa principal (main), você possa:

1. Criar diferentes tipos de pacotes.
2. Criar diferentes tipos de veículos com suas capacidades.
3. Iterar sobre uma lista de veículos e pacotes, e para cada veículo, verificar se ele pode carregar o pacote e, se sim, realizar a entrega.

Obrigada