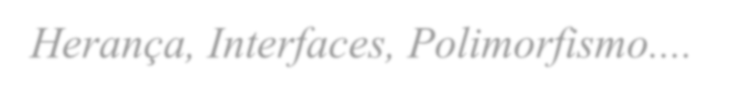
Pontifícia Universidade Católica do Paraná Escola Politécnica

*Curso de Engenharia de Computação Programação Orientada a Objetos*

Não tenha pressa, mas não perca tempo.

José Saramago



*TDE 2 – Herança, Interfaces, Polimorfismo....*

Considerando os códigos-fonte do projeto TDE2, disponíveis no arquivo TDE2.zip e as referências bibliográficas sugeridas como material de apoio às aulas e indicadas na definição deste TDE no AVA Canvas, responda às questões abaixo de maneira completa, autocontida, e que sirva de sistematização conceitual e de resumo dos principais temas iniciais da disciplina.

Na classe TestaAnimal, temos:

**import** Animais.\*;

**import** Interfaces.AnimalSelvagem;

1. Explique a diferença entre estes dois tipos de **import**.

**R: O primeiro uso do import “puxa”/importa todas as classes que estão dentro do package/pacote Animais, como Animal, Canideo, Cao, etc. A utilização do asterisco (\*) significa que todas as classes e interfaces no pacote Animais serão importadas.**

**O segundo uso do import “puxa”/importa uma classe específica do pacote Interfaces, no caso, a AnimalSelvagem.**

1. Para que serve um *package*?

**R: Um pacote Java organiza classes Java em namespaces, fornecendo um namespace exclusivo para cada tipo que ele contém. Classes no mesmo pacote podem acessar os membros privados e protegidos do pacote umas das outras. Os packages são uma ferramenta fundamental em Java para organizar, modularizar, e proteger o código. Eles permitem uma melhor gestão de projetos, evitam conflitos de nomes e promovem boas práticas de desenvolvimento, como a reutilização de código e a manutenção facilitada.**

Em relação a Polimorfismo:

1. Explique o conceito de polimorfismo e porque ele é importante.

**R: Definimos Polimorfismo como um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas.**

**O Polimorfismo é um mecanismo por meio do qual selecionamos as funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução.**

**Com o Polimorfismo, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes.**

**Por exemplo: podemos assumir que uma bola de futebol e uma camisa da seleção brasileira são artigos esportivos, mas o cálculo deles em uma venda é calculado de formas diferentes.**

1. Encontre no projeto um exemplo de polimorfismo e explique.

**R: O polimorfismo no código fornecido é exemplificado pelo uso do array Animal[] zoo e pela chamada dinâmica de métodos como fazerBarulho() (entre outros), que permitem que diferentes tipos de animais respondam de maneira específica às chamadas de método, demonstrando a flexibilidade e o poder da programação orientada a objetos.**

**Polimorfismo via Array de Animais: O array Animal[] zoo é declarado como um array de objetos da classe Animal. No entanto, ele contém objetos de diferentes subclasses (Cao, Gato, Leao, etc), todas elas derivadas de Animal.**

**Ligação Dinâmica: Quando zoo[i].fazerBarulho() é chamado dentro do loop for, a implementação específica do método fazerBarulho() que é executada depende do tipo real do objeto referenciado por zoo[i]. Isso demonstra a ligação dinâmica (ou despacho dinâmico), onde o método apropriado é determinado em tempo de execução.**

**Instanceof e Casting: Dentro do loop, há uma verificação if (zoo[i] instanceof Canideo). Isso verifica se o objeto atual no array é uma instância da interface Canideo. Se for, o objeto é convertido (cast) para Canideo e o método esconderOsso() é chamado. Esta é uma aplicação prática do polimorfismo, onde métodos específicos de subclasses ou interfaces são chamados com base no tipo real do objeto.**

1. O que é *ligação (ou vinculação) tardia*; encontre um exemplo no projeto onde isto ocorre é explique.

**R: A ligação tardia é um aspecto crucial da programação orientada a objetos que permite que o comportamento dos métodos seja decidido em tempo de execução, com base no tipo real do objeto. Na classe TestaAnimal, isso é claramente exemplificado pela chamada ao método fazerBarulho() no array Animal[] zoo, onde diferentes implementações do método são invocadas dependendo da subclasse específica (Cao, Gato, Leao) do objeto.**

1. No caso da sobrecarga, ocorre *ligação (ou vinculação) inicial (ou antecipada)*; encontre um exemplo no projeto onde isto ocorre é explique o conceito.

**R:** **A sobrecarga de métodos e construtores oferece flexibilidade e clareza no código, permitindo múltiplas maneiras de inicializar objetos ou executar métodos com diferentes conjuntos de parâmetros. A ligação antecipada garante que essas decisões são feitas em tempo de compilação, resultando em um código mais eficiente em tempo de execução e ajudando a evitar erros relacionados a métodos não encontrados.**

**Explicação da Ligação Antecipada no Uso de Construtores:**

**Quando o código chama new Cao("Scooby-Doo", 53.9F), o compilador já sabe, em tempo de compilação, que deve usar o construtor Cao(String nome, float peso). Da mesma forma, new Gato() chama o construtor sem parâmetros Gato().**

**A decisão de qual construtor chamar é feita pelo compilador com base nos argumentos passados. Isso é diferente da ligação tardia, onde a decisão de qual método específico invocar é feita em tempo de execução com base no tipo real do objeto.**

Na classe TestaAnimal, podemos escrever:

gato1.acordar();

Porém, não podemos fazer:

gato1.acordado = **true**;

Devido ao encapsulamento usado na classe Animal.

1. Qual a importância do encapsulamento? De que forma o encapsulamento aumenta a segurança de uma classe?

**R: O encapsulamento refere ao conceito de esconder os detalhes internos de uma classe e só permitir que ela seja manipulada por meio de métodos públicos definidos. Isso significa que os dados internos da classe (os atributos) são protegidos do acesso direto de fora da classe, e qualquer interação com esses dados deve passar por métodos (getters, setters e outros métodos públicos).**

**Principais Importâncias do Encapsulamento:**

1. **Proteção de Dados;**
2. **Flexibilidade e Manutenção;**
3. **Facilidade de Uso;**
4. **Segurança.**

**Explicação da Segurança Fornecida pelo Encapsulamento: 1. Controle de Acesso Direto, por exemplo, “acordado” é um atributo privado. 2. Métodos públicos para controle de acesso, por exemplo, os métodos getters e setters. 3. Isolamento de mudanças internas, por exemplo, Se a forma como o estado acordado é gerido mudar, isso não afetará o código externo, desde que os métodos acordar(), dormir() e isAcordado() permaneçam os mesmos. Isso facilita a manutenção e evolução do código.**

1. São quatro os modificadores de acesso: private, *default* ou *package-private*, protected e

public; explique cada um destes modificadores.

**R: O modificador private limita a visibilidade dos membros ao escopo da própria classe. Isso significa que apenas métodos definidos dentro da mesma classe podem acessar ou modificar esses membros. Esse nível de acesso é crucial para proteger os dados internos da classe contra alterações indesejadas ou acidentais por outras partes do programa, garantindo que a integridade dos dados seja mantida. Por exemplo, se um atributo é marcado como private, como private int value;, ele só pode ser acessado e modificado por métodos dentro da mesma classe, como private void setValue(int value).**

**O modificador default (ou package-private) é aplicado quando nenhum modificador de acesso é especificado. Ele permite que os membros da classe sejam acessíveis apenas por outras classes no mesmo pacote. Isso promove uma organização modular do código, onde classes relacionadas e pertencentes ao mesmo pacote podem interagir livremente, mas o acesso é restrito fora do pacote. Por exemplo, se declararmos int value; sem nenhum modificador, este atributo será visível apenas para outras classes no mesmo pacote.**

**O modificador protected amplia a visibilidade dos membros para incluir todas as classes no mesmo pacote e também qualquer subclasse, mesmo que estejam em pacotes diferentes. Esse modificador é particularmente útil em cenários de herança, onde é necessário permitir que subclasses acessem e modifiquem membros definidos na classe base. Por exemplo, protected int age; pode ser acessado e modificado por subclasses de qualquer pacote, facilitando a reutilização e extensão do código.**

**O modificador public oferece o nível mais alto de visibilidade, permitindo que os membros sejam acessíveis de qualquer lugar no programa. Isso é ideal para métodos e atributos que fazem parte da API pública da classe, proporcionando uma interface clara e estável para outros componentes interagirem. Por exemplo, public String type; é acessível a partir de qualquer outra classe, independentemente do pacote em que esteja.**

A classe Cao *extends* a classe Canideo.

1. O que significa fazer com que uma classe *extends* outra classe?

**R: Quando uma classe em Java usa a palavra-chave extends para estender outra classe, isso significa que ela está herdeando os atributos e métodos da classe pai (ou superclasse). Este conceito é conhecido como herança, um dos pilares fundamentais da programação orientada a objetos (POO). A herança permite que uma classe (a subclasse) reutilize código de outra classe (a superclasse), promovendo a reutilização, a organização e a manutenção eficiente do código.**

**Quando uma classe Cao estende a classe Canideo, Cao herda todos os atributos e métodos definidos em Canideo, bem como quaisquer atributos e métodos definidos nas classes ascendentes de Canideo (por exemplo, se Canideo estende outra classe, Animal). Isso significa que Cao pode acessar e utilizar esses atributos e métodos como se fossem definidos dentro dela mesma. Além disso, Cao pode adicionar seus próprios atributos e métodos ou sobrescrever (override) métodos da superclasse para fornecer implementações específicas.**

1. O que significa herança? O que de fato é herdado?

**R: É um mecanismo que permite criar uma nova classe baseada em uma classe existente. A nova classe, chamada de subclasse ou classe derivada, herda os atributos e métodos da classe existente, conhecida como superclasse ou classe base. A herança promove a reutilização de código e a criação de hierarquias de classes que refletem relações do mundo real, como "um cão é um tipo de animal".**

**O que é Herdado:**

1. **Atributos (Campos);**
2. **Métodos;**
3. **Construtores;**
4. **Implementações de Interface.**

A classe Felino também *extends* a classe Animal, porém ela é uma classe abstrata (*abstract*).

1. O que é uma classe abstrata?

**R: Uma classe abstrata em Java é uma classe que não pode ser instanciada diretamente; ou seja, você não pode criar objetos a partir dela. Em vez disso, ela serve como uma classe base para outras classes que podem ser instanciadas. Uma classe abstrata pode conter métodos abstratos e métodos concretos.**

**Métodos abstratos são declarados sem implementação, ou seja, apenas com a assinatura do método, e as subclasses são obrigadas a fornecer a implementação desses métodos. Métodos concretos têm implementação na própria classe abstrata e podem ser herdados pelas subclasses.**

1. Qual o objetivo/finalidade de uma classe abstrata?

**R: O objetivo de uma classe abstrata é fornecer uma base comum para outras classes. Ela permite a definição de um conjunto de métodos e atributos que todas as suas subclasses devem possuir, garantindo uma estrutura consistente e promovendo a reutilização de código. Além disso, classes abstratas permitem definir comportamentos padrão que podem ser compartilhados por várias subclasses, enquanto ainda permitem que essas subclasses forneçam implementações específicas para métodos abstratos.**

1. Que outras classes neste projeto foram definidas como classes abstratas?

**R: Animal, Canideo e o Felino, como já citado.**

**Exemplo Classe Abstrata: No TDE2, podemos imaginar que foi criado uma aplicação para gerenciar um zoológico. Sabemos que todos os animais no zoológico compartilham certas características e comportamentos, como nome, peso, a capacidade de dormir e acordar, mas diferentes tipos de animais fazem sons diferentes. Em vez de definir essas características e comportamentos em cada tipo de animal (o que levaria à duplicação de código), pode-se criar uma classe abstrata Animal que define essas características e comportamentos comuns.**

Na classe Animal o método fazerBarulho()é um método abstrato.

1. Para que serve a definição de um método abstrato em uma classe abstrata?

**R: A definição de um método abstrato em uma classe abstrata serve para forçar as subclasses a implementar esse método. Ele estabelece um contrato que as subclasses devem seguir, garantindo que todas as subclasses tenham uma implementação específica desse método. No caso do método fazerBarulho() na classe Animal, isso significa que qualquer classe que estenda Animal precisa fornecer sua própria versão de como o animal faz barulho. Isso é útil quando você deseja que diferentes subclasses tenham comportamentos específicos, mas quer garantir que todas elas implementem certos métodos.**

1. Uma classe abstrata pode conter métodos concretos? Qual a finalidade?

**R: Sim, uma classe abstrata pode conter métodos concretos. A finalidade de incluir métodos concretos em uma classe abstrata é fornecer uma implementação padrão que pode ser usada ou herdada pelas subclasses. Isso permite que você compartilhe código comum entre várias subclasses, evitando a duplicação e facilitando a manutenção do código. Por exemplo, na classe Animal, os métodos dormir(), acordar(), getNome(), e setNome() são métodos concretos que fornecem comportamento padrão para todas as subclasses de Animal.**

1. Uma classe abstrata pode conter método construtor? Qual a finalidade?

**R: Sim, uma classe abstrata pode conter métodos construtores. A finalidade dos construtores em uma classe abstrata é inicializar os atributos comuns da classe base. Embora você não possa instanciar diretamente uma classe abstrata, suas subclasses podem chamar os construtores da classe abstrata para garantir que os atributos definidos na classe base sejam inicializados corretamente. No exemplo da classe Animal, os construtores são usados para inicializar os atributos nome, peso e acordado, assegurando que todos os animais, independentemente do tipo específico, tenham esses atributos corretamente configurados ao serem criados.**

A classe Cão *extends* a classe Canídeo, que por sua vez *extends* a classe Animal, que por sua vez

*extends* a classe Object (de forma transparente). Na classe Zoo:

1. Ao executar a.getNome(), de onde vem o método getNome()?

**R: O método getNome() vem da classe Animal. Na hierarquia de classes, Cao herda de Canideo, que herda de Animal. A classe Animal define o método getNome() e, como todas as subclasses herdam métodos e atributos das suas superclasses, Cao também possui esse método. Portanto, quando executamos a.getNome(), estamos utilizando o método definido na classe Animal.**

1. Ao executar ae.salve\_me(), de onde vem o método salve\_me()?

**R: O método salve\_me() vem da interface AnimalExtincao. No contexto do projeto, ae é uma instância de uma classe que implementa a interface AnimalExtincao. As interfaces são contratos que garantem que qualquer classe que as implemente fornecerá uma implementação concreta dos métodos definidos na interface. Assim, salve\_me() é um método que deve ser implementado por qualquer classe que declare implementar AnimalExtincao.**

1. Ao executar zoo.get(i).toString(), de onde vem o método toString()?

**R: O método toString() vem da classe Object. Na hierarquia de classes Java, todas as classes herdam implicitamente da classe Object, que é a superclasse base de todas as classes. A classe Object fornece uma implementação padrão de toString(), que pode ser sobrescrita por qualquer classe para fornecer uma representação de string mais específica do objeto. No caso da classe Animal, o método toString() é sobrescrito para fornecer uma representação mais específica do objeto Animal. Portanto, quando zoo.get(i).toString() é executado, ele chama a implementação sobrescrita de toString() da classe Animal, se houver, caso contrário, ele usa a implementação padrão da classe Object.**

1. Explique a coerção efetuada na linha abaixo, por que ela é necessária?

AnimalSelvagem as = (AnimalSelvagem)a;

**R: A coerção (ou casting) na linha acima é necessária para converter o objeto a, que é do tipo Animal, para o tipo AnimalSelvagem. No contexto do projeto, a é uma instância da classe Animal ou de uma de suas subclasses, mas o compilador não pode garantir que a implementa a interface AnimalSelvagem apenas pelo tipo declarado. Para poder chamar métodos definidos na interface AnimalSelvagem, é necessário realizar o casting para garantir ao compilador que a é de fato uma instância de uma classe que implementa essa interface. Isso permite usar os métodos específicos da interface AnimalSelvagem no objeto a.**

1. Por que foi necessário efetuar o teste abaixo antes da coerção?

**if** (a **instanceof** AnimalExtincao) {

**R: O teste instanceof é necessário antes do casting para garantir que o objeto a realmente implementa a interface AnimalExtincao. Em Java, tentar fazer um casting de um objeto para um tipo que ele não implementa ou não herda resulta em uma ClassCastException. Para evitar essa exceção e garantir que o programa funcione corretamente, o teste instanceof verifica se a é uma instância de uma classe que implementa a interface AnimalExtincao antes de realizar o casting. Este teste é uma medida de segurança para assegurar que a coerção será válida e segura, permitindo que os métodos da interface sejam chamados sem risco de exceções.**

1. AnimalSelvagem e AnimalExtincao são interfaces. Explique o que acontece quando tratamos a variável a (do tipo Animal) através de uma interface. O que poderemos acessar através das variáveis ae e as?

**R: Quando tratamos a variável a (do tipo Animal) através de uma interface como AnimalSelvagem ou AnimalExtincao, usamos polimorfismo, que permite que um objeto seja tratado como uma instância de uma interface que ele implementa. Isso restringe o acesso aos métodos definidos na interface, mesmo que a classe concreta tenha outros métodos.**

**- ae (do tipo AnimalExtincao): Podemos acessar métodos definidos na interface AnimalExtincao, como salve\_me().**

**- as (do tipo AnimalSelvagem): Podemos acessar métodos definidos na interface AnimalSelvagem, como atacar(Animal animal).**

1. AnimalExtincao **extends** AnimalSelvagem, logo, se criarmos uma classe que **implements**

AnimalExtincao, quais métodos ela seria obrigada a implementar devido à interface?

**R: Se uma classe implementa a interface AnimalExtincao e esta interface estende AnimalSelvagem, então a classe será obrigada a implementar todos os métodos definidos tanto na interface AnimalExtincao quanto na interface AnimalSelvagem. No caso:  
-> atacar(Animal animal) (definido na interface AnimalSelvagem).**

**-> salve\_me() (definido na interface AnimalExtincao).**

1. O Gato-Maracajá é um felino da América do Sul que está em extinção. Crie uma classe chamada GatoMaracaja que seja subclasse de Felino e implemente a interface AnimalExtincao. Implemente os métodos que foram exigidos e cite quais são e quem exigiu.

**R: Para criar a classe GatoMaracaja, faremos com que ela seja uma subclasse de Felino e implemente a interface AnimalExtincao. Isso exigirá que a classe GatoMaracaja implemente os métodos atacar(Animal animal) da interface AnimalSelvagem (herdada por AnimalExtincao) e salve\_me() da interface AnimalExtincao.**

**Código:**

**package Animais;**

**import Interfaces.AnimalExtincao;**

**public class GatoMaracaja extends Felino implements AnimalExtincao {**

**public GatoMaracaja(String nome, float peso) {**

**super(nome, peso);**

**this.tipo = FELINO.SELVAGEM;**

**}**

**public void atacar(Animal animal) {**

**System.out.println(getNome() + " está atacando " + animal.getNome());**

**}**

**public void salve\_me() {**

**System.out.println(getNome() + " está em extinção. Salve-me!");**

**}**

**public void fazerBarulho() {**

**System.out.println(getNome() + " está fazendo barulho: Miau!");**

**}**

**}**

**-> atacar(Animal animal): Este método é exigido pela interface AnimalSelvagem, que é uma superinterface de AnimalExtincao. Como AnimalExtincao estende AnimalSelvagem, qualquer classe que implemente AnimalExtincao deve fornecer uma implementação para atacar(Animal animal).**

**-> salve\_me(): Este método é exigido diretamente pela interface AnimalExtincao.**

Modifique a classe Roedor incluindo *abstract* em seu cabeçalho.

1. Esta mudança criou um erro em outra classe. Em qual classe isto ocorreu e qual o motivo do erro?

**R: Ao modificar a classe Roedor para incluir abstract em seu cabeçalho, o erro ocorreu na classe Zoo. Isso aconteceu porque a classe Roedor passou a ser abstrata e, portanto, não pode ser instanciada diretamente. Como resultado, qualquer tentativa de criar um objeto diretamente da classe Roedor no Zoo geraria um erro de compilação, pois classes abstratas não podem ser instanciadas.**

Crie a classe Esquilo, com o seguinte código:

**package** Animais;

**public class** Esquilo **extends** Roedor {

}

Atualize a linha com erro (gerada na questão anterior) pela seguinte codificação:

Animal esquilo = **new** Esquilo();

Arrume então os erros que apareceram em função desta mudança.

**R: Ao modificar a classe Roedor para ser abstrata (abstract) e criar a classe Esquilo que estende Roedor, a linha com erro foi corrigida para Animal esquilo = new Esquilo();. Isso resolve o problema porque agora Esquilo é uma subclasse concreta de Roedor, permitindo sua instanciação. É importante ajustar todas as referências anteriores a Roedor para usar corretamente Esquilo, onde aplicável, garantindo assim a correção dos erros causados pela mudança para abstrato.**

**Código:**

**package Animais;  
  
public class Esquilo extends Roedor {  
  
 public Esquilo(String nome) {  
 super(nome);  
 }  
  
 @Override  
 public void fazerBarulho() {  
 System.*out*.println(getNome() + " fazendo barulho: squick squick...");  
 }  
}**

**Esta instância de Esquilo [ Animal esquilo = new Esquilo(); ]é tratada como um Animal, permitindo polimorfismo e o uso genérico de métodos e atributos definidos em Animal.**

Ao rodar a classe, tudo funciona normalmente.

1. Explique como se pode criar uma instância de Esquilo e associá-la a uma variável declarada como sendo do tipo Animal?

**R: Para criar uma instância de Esquilo e associá-la a uma variável declarada como Animal, utilizamos o conceito de herança e polimorfismo. A classe Esquilo estende (extends) Roedor, que por sua vez estende Animal. Isso significa que Esquilo é um tipo mais específico de Animal e pode ser tratado como Animal devido à relação de herança.**

**Animal esquilo = new Esquilo();**

**Nesta linha de código, estamos criando uma nova instância de Esquilo e atribuindo-a à variável esquilo que é do tipo Animal. Isso é possível porque Esquilo é uma subclasse de Animal, portanto pode ser referenciado polimorficamente como Animal.**

1. A classe Esquilo está vazia, sem construtor explícito, entretanto criou-se uma instância desta classe. Como isto funciona?

**R: A classe Esquilo possui um construtor padrão implícito. Como não foi definido nenhum construtor explícito na classe Esquilo, o compilador Java automaticamente insere um construtor padrão sem argumentos. Este construtor padrão é chamado quando criamos uma instância de Esquilo utilizando new Esquilo();. O super() no construtor da classe Esquilo irá chamar o construtor padrão de Roedor, no qual, é sua superclasse.**

1. Por que a classe Esquilo não foi obrigada a implementar o método **public void**

fazerBarulho()?

**R: A classe Esquilo não foi obrigada a implementar o método fazerBarulho() porque ela é uma subclasse de Roedor, e Roedor já é uma classe concreta que implementa o método abstrato fazerBarulho() da classe Animal.**

1. Na classe Zoo foram feitos três tipos diferente de loop:
   1. Loop 1, for com size e get

**R: Circunstâncias adequadas: Este tipo de loop é útil quando precisamos acessar elementos do ArrayList por índice. É eficiente quando precisamos iterar sobre todos os elementos sequencialmente e precisamos do índice para outras operações além de apenas acessar o elemento.**

**Exemplo: Se precisamos realizar operações específicas baseadas no índice do elemento, como modificar ou remover elementos do ArrayList.**

* 1. Loop 2, while com iterator

**R: Circunstâncias adequadas: Utilizar um Iterator é preferível quando precisamos remover elementos durante a iteração ou quando não precisamos do índice do elemento. O Iterator fornece métodos seguros para remover elementos durante a iteração sem causar exceções de modificação concorrente.**

**Exemplo: Quando precisamos remover elementos do ArrayList baseado em alguma condição durante a iteração sem causar problemas de concorrência.**

* 1. Loop 3, for each

**R: Circunstâncias adequadas: O for each é ideal quando precisamos apenas iterar sobre todos os elementos sem necessidade de acesso por índice ou remoção de elementos. É a forma mais simples e limpa de percorrer uma coleção de elementos.**

**Exemplo: Quando precisamos realizar operações simples em todos os elementos da coleção, como exibir informações ou realizar operações que não alteram a estrutura da coleção.**

Comente em que circuntâncias cada uma destas estratégias de navegar pelo ArrayList seria mais indicada.

Uma das grandes diferenças entre o código-fonte do projeto TDE2 e o código-fonte do projeto Animal que vem sendo construído durante as aulas remotas é que nas classes deste TDE2 em discussão foram inseridos comandos de saída de dados (System.***out***.println) nos métodos das próprias classes de dados.

1. Apesar de ser prático para demonstrar algum conceito e testar rapidamente uma ideia, não é indicado codificar comando de entrada/saída nas classes que modelam o domínio da aplicação.

Explique o motivo.

**R: Não é recomendado codificar comandos de entrada/saída diretamente nas classes que modelam o domínio da aplicação devido aos princípios de responsabilidade única e separação de preocupações. Os principais motivos são:**

1. **Separação de Responsabilidades: Classes de domínio devem focar apenas na representação e manipulação dos dados relacionados ao seu conceito específico, sem misturar responsabilidades com lógica de saída de dados.**
2. **Reutilização: Lógica de saída específica reduz a flexibilidade e portabilidade das classes de domínio.**
3. **Manutenção: Misturar lógica de apresentação com lógica de domínio dificulta a manutenção e pode impactar negativamente a estabilidade e legibilidade do código.**
4. **Testabilidade: Classes com lógica de saída embutida são mais difíceis de testar, complicando a captura e verificação das saídas de console.**

**Para manter um código limpo, modular, reutilizável e fácil de evoluir, é recomendado separar claramente as camadas de responsabilidade.**

**Exemplo prático: Imagine que você está construindo um modelo de casa usando peças de Lego. Cada conjunto de peças representa diferentes partes da casa, como a sala, o quarto, a cozinha, etc. Agora, imagine que você decide pintar cada peça com uma cor diferente para identificar melhor cada parte da casa.**

**Nesse exemplo, as peças de Lego são como as classes que você cria em um programa de computador, e cada parte da casa representa um conceito específico, como um animal (Cao, Gato, etc.) em nosso caso. Pintar as peças de Lego com cores diferentes é como adicionar lógica de saída de dados diretamente nas classes. Embora seja útil para identificar cada parte da casa rapidamente, essa abordagem pode tornar as peças menos versáteis se você quiser usá-las para construir outros tipos de edifícios no futuro.**

**Em vez disso, é mais eficaz manter as peças de Lego neutras e sem pintura, deixando a identificação das partes da casa para um mapa ou rótulo separado. Isso mantém suas peças de Lego mais flexíveis e reutilizáveis para diferentes projetos de construção. Da mesma forma, separar a lógica de saída das classes de domínio em um programa de computador permite que você as utilize de maneira mais flexível e eficiente em diferentes partes do seu sistema.**