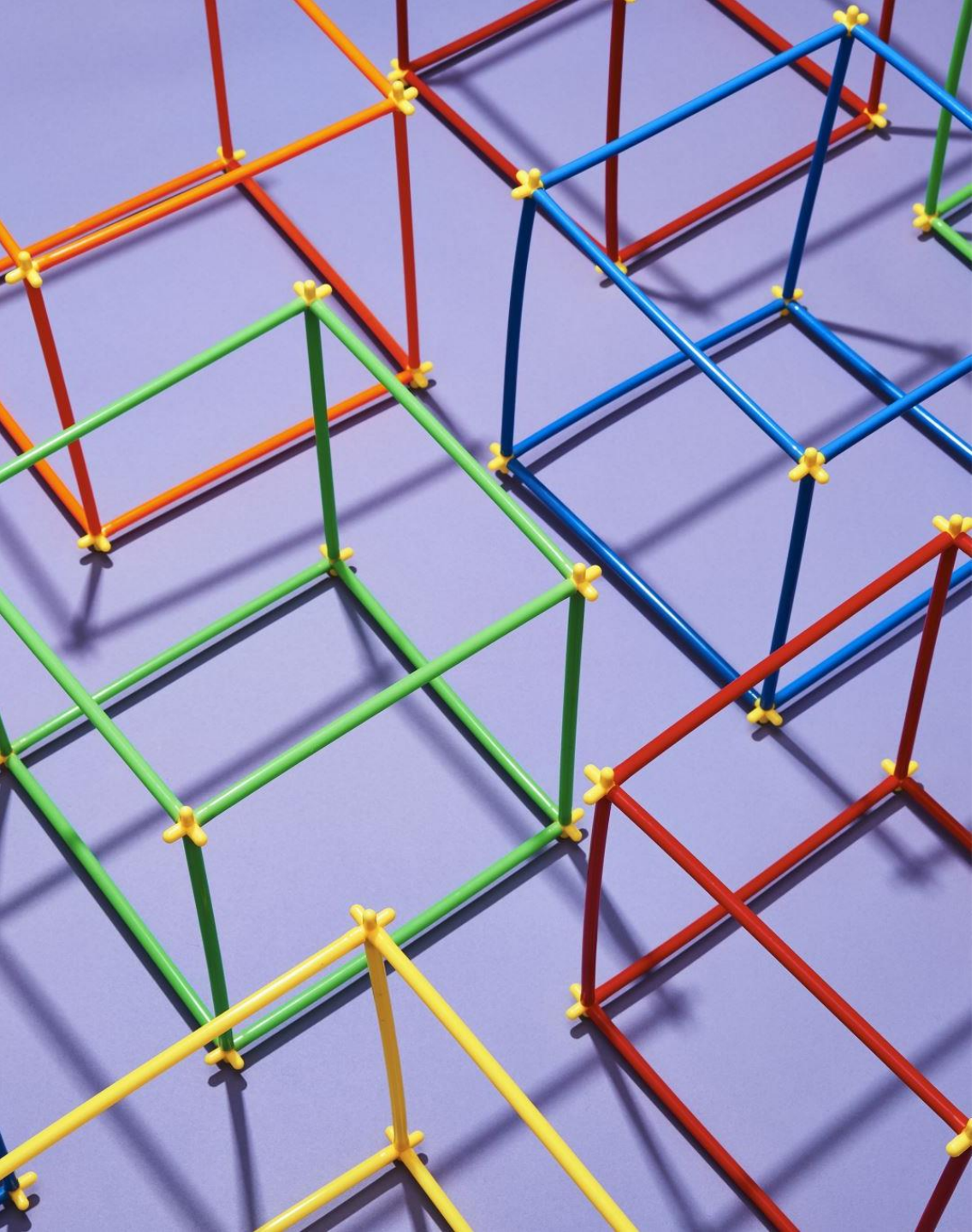
Semana 4 – Paradigma Orientação à Objetos

Prof. Cassiano Moralles

História e Conceitos

O CONCEITO DE PROGRAMAÇÃO ORIENTADA A OBJETOS TEM SUAS RAÍZES NO SIMULA 67, MAS NÃO FOI COMPLETAMENTE DESENVOLVIDO ATÉ A EV)OLUÇÃO DO SMALLTALK QUE RESULTOU NO SMALLTALK 80 (EM 1980)

UMA **LINGUAGEM ORIENTADA A OBJETOS** DEVE FORNECER SUPOR TE PARA TRÊS RECURSOS CHAVE DE LINGUAGEM: **TIPOS DE DADOS ABSTRATOS**, **HERANÇA** E VINCULAÇÃO DINÂMICA DE **CHAMADAS A MÉTODOS**.



Tipos de Dados Abstratos e Construções de Encapsulamento

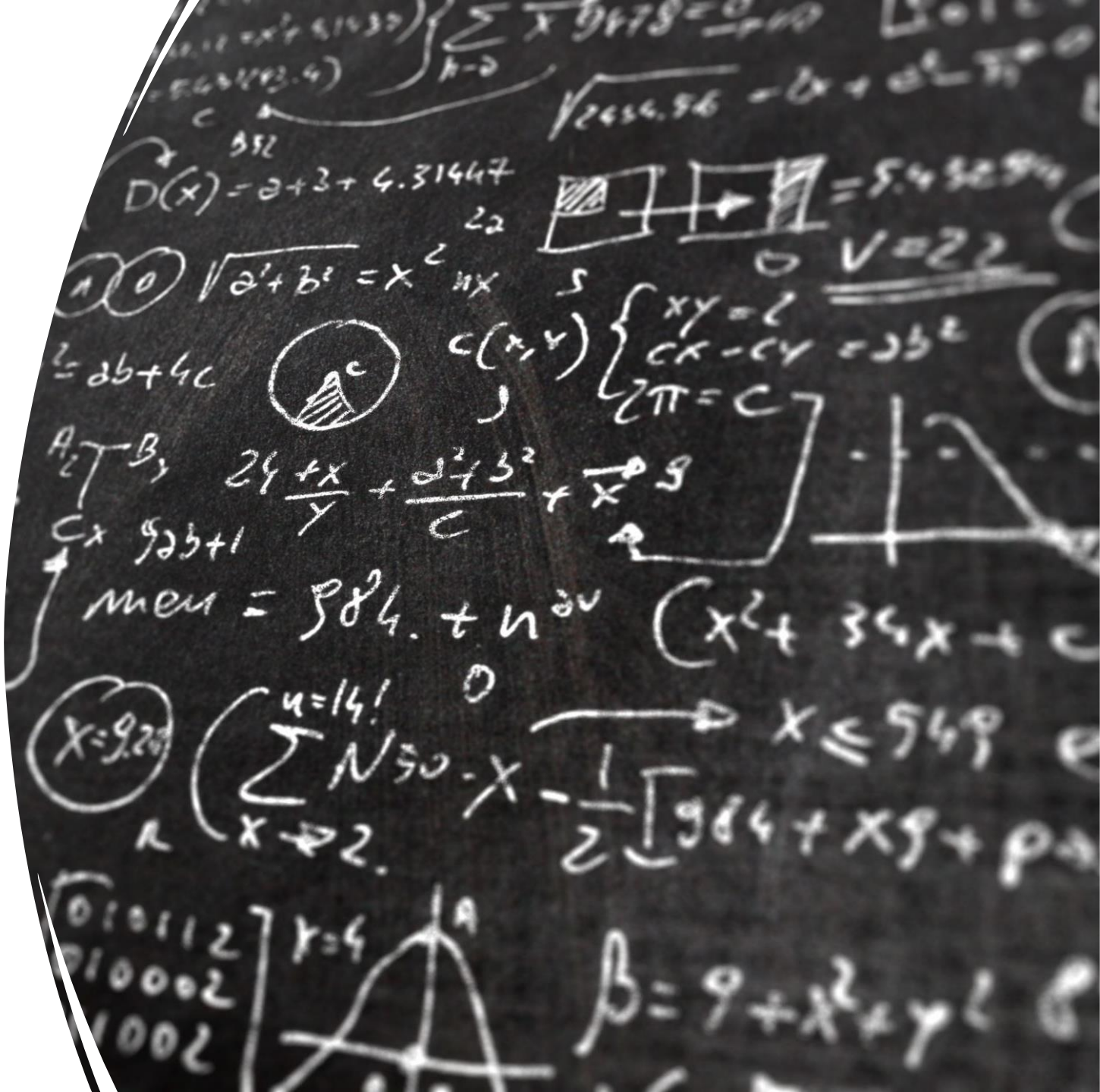
- Dentre as novas ideias dos últimos 50 anos nas metodologias e no projeto de linguagens de programação, a abstração de dados é uma das mais profundas.
- Construções que suportam tipos de dados abstratos são encapsulamentos dos dados de operações em objetos do tipo.
- Encapsulamentos que contêm múltiplos tipos são necessários para a construção de programas maiores.

Tipos de Dados Abstratos e Construções de Encapsulamento

- Uma abstração é uma visão ou representação de uma entidade que inclui apenas os atributos mais significativos. A abstração permite agrupar exemplares de entidades em grupos onde seus atributos comuns não precisam ser considerados individualmente.
- Por exemplo, ao definir aves como criaturas com as seguintes características: duas asas, duas pernas, um rabo e penas, uma descrição de um corvo, ao ser identificado como uma ave, não precisa incluir esses atributos. O mesmo se aplica a piscos-de-peito-ruivo, pardais e pica-paus de barriga amarela.

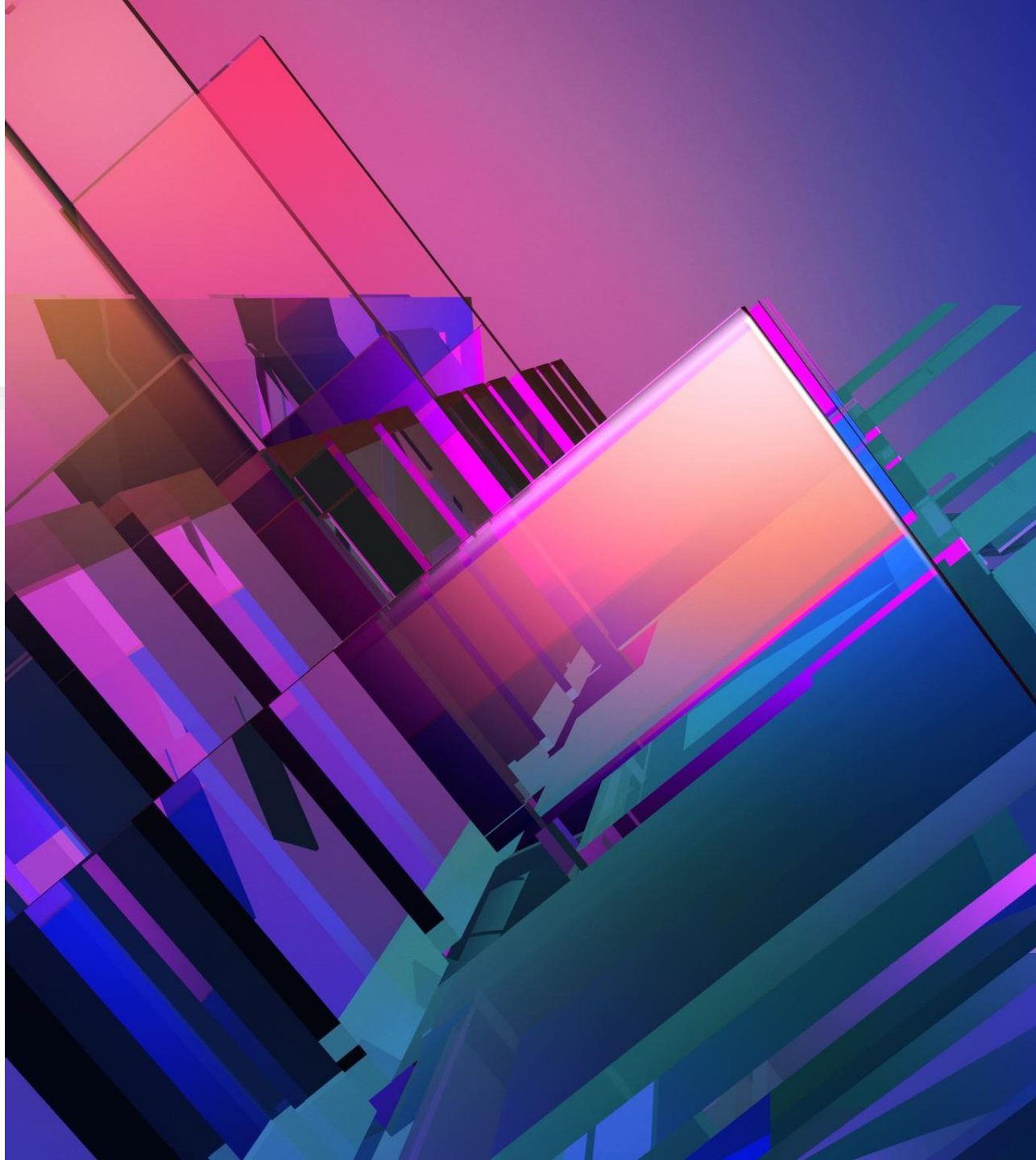
Tipos de Dados Abstratos e Construções de Encapsulamento

- No mundo das linguagens de programação, a abstração é uma ferramenta poderosa contra a complexidade da programação.
- Seu objetivo é simplificar o processo de desenvolvimento, permitindo que os programadores se concentrem nos atributos essenciais enquanto ignoram os detalhes subordinados.



Tipos de Dados Abstratos e Construções de Encapsulamento

- Os dois tipos fundamentais de abstração nas linguagens de programação contemporâneas são a abstração de processos e a abstração de dados.
- O conceito de abstração de processo - Todos os subprogramas são abstrações de processo, porque fornecem uma maneira pela qual um programa especifica um processo, sem fornecer os detalhes de como ele é realizado (ao menos no programa chamador).
- Um exemplar de um tipo de dados abstrato é chamado de um **objeto**.



Tipos de Dados Abstratos e Construções de Encapsulamento

- A programação orientada a objetos é uma melhoria do uso de abstração de dados em desenvolvimento de software, e a abstração de dados é um de seus componentes mais importantes.

Tipos de Dados Abstratos e Construções de Encapsulamento

- Ponto flutuante como um tipo de dados abstrato
  - O formato real do valor de dado em uma célula de memória de ponto flutuante é oculto do usuário, e as únicas operações disponíveis são as fornecidas pela linguagem.
- Tipos de dados abstratos definidos pelo usuário
  - (1) uma definição de tipo que permita às unidades de programa declararem variáveis do tipo, mas que oculte a representação de seus objetos; e (2) um conjunto de operações para manipular os objetos.

create(stack)	Cria e possivelmente inicializa um objeto de pilha
destroy(stack)	Libera o armazenamento para a pilha
empty(stack)	Um predicado ou função booleana que retorna verdadeiro (*true*) se a pilha especificada é vazia e falso (*false*) caso contrário.
push(stack, element)	Insere o elemento especificado na pilha especificada
pop(stack)	Remove o elemento do topo da pilha especificada
top(stack)	Retorna uma cópia do elemento do topo da pilha especificada

Questões De Projeto Para Tipos De Dados Abstratos

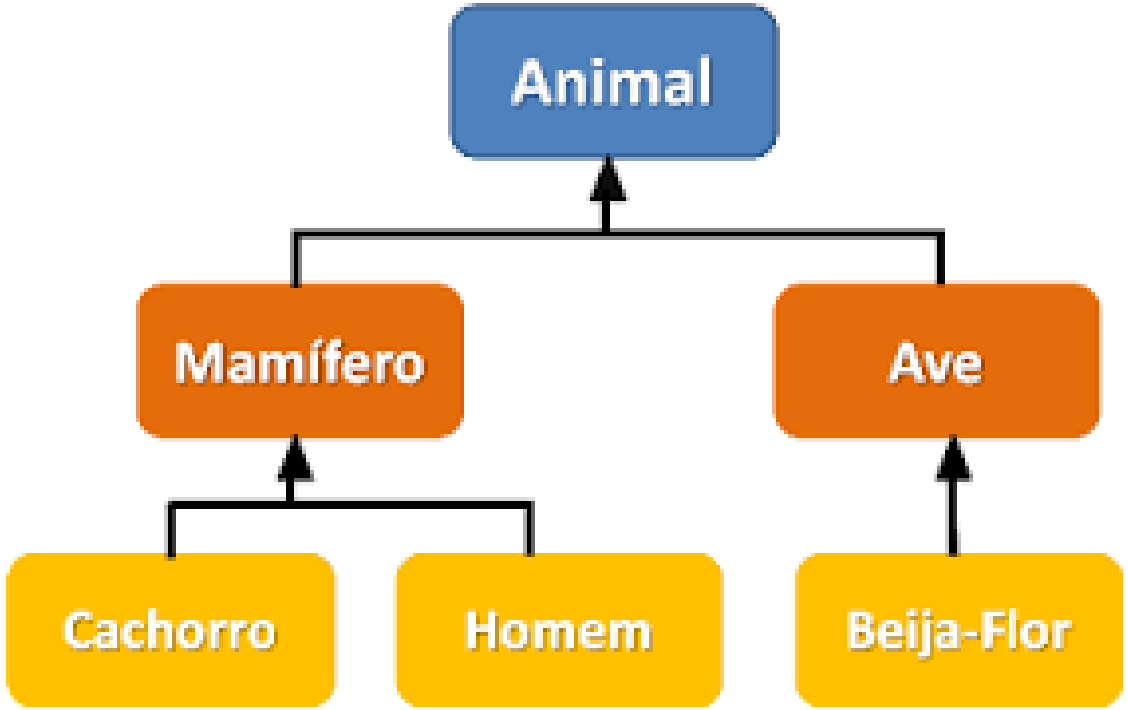
- Deve fornecer uma unidade sintática que envolva a declaração do tipo e os protótipos dos subprogramas que implementam as operações em objetos do tipo.
- Torná-los visíveis aos clientes da abstração, permitindo que declarem variáveis do tipo abstrato e manipulem seus valores.
- A representação do tipo e as definições dos subprogramas que implementam as operações podem aparecer dentro ou fora dessa unidade sintática.

Herança

- Aumento de produtividade -> reuso de software;
- A herança oferece uma solução eficaz tanto para a modificação quanto para a organização de programas ao permitir que novos tipos abstratos de dados herdem dados e funcionalidades de tipos existentes. Isso facilita o reuso sem exigir mudanças nos tipos reutilizados.
- Programadores podem partir de um tipo existente e projetar um descendente modificado para atender a novos requisitos, enquanto a herança também fornece um framework para definir hierarquias de classes relacionadas, refletindo os relacionamentos de descendência no espaço do problema.

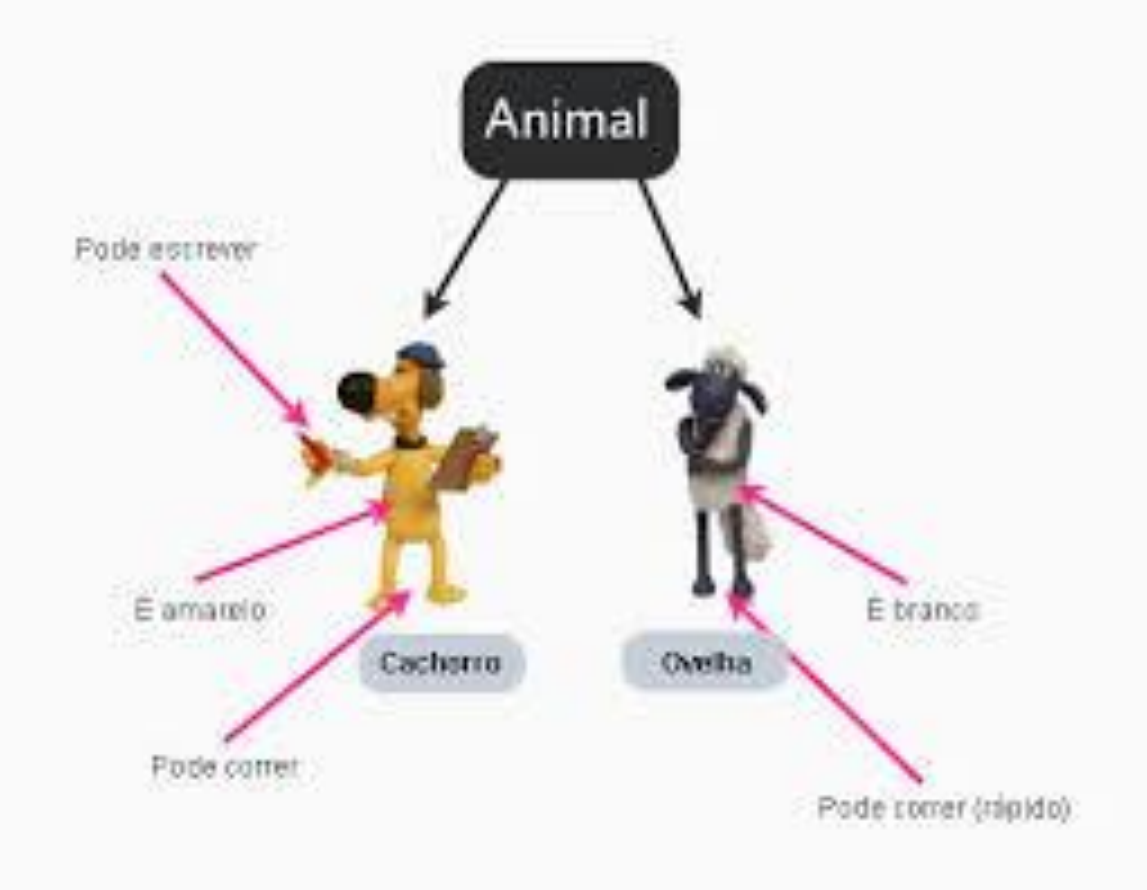
Herança

- Os tipos abstratos de dados em linguagens orientadas a objetos, seguindo a nomenclatura de SIMULA 67, são normalmente chamados de classes. Assim como as instâncias de tipos abstratos de dados, as instâncias de classes são chamadas de objetos. Uma classe é definida por meio de herança de outra classe é chamada de classe derivada ou subclasse. Uma classe da qual a nova é derivada é sua classe pai ou superclasse.



Herança

- Os subprogramas que definem as operações em objetos de uma classe são chamados de métodos. As chamadas a métodos são algumas vezes chamadas de mensagens. A coleção completa de métodos de um objeto é chamada de protocolo de mensagens ou interface de mensagens. Computações em um programa orientado a objetos são especificadas por mensagens enviadas de objetos para outros ou, em alguns casos, para classes.



Herança

- Diferenças mais comuns entre uma classe pai e suas subclasses:
  - A classe pai pode definir alguns de seus membros como tendo acesso privado, fazendo com que esses métodos não sejam visíveis na subclasse.
    - Utilizando o modificador de acesso **private**.

java

```java
// Superclasse
class Animal {
    private String nome; // Atributo privado

    public void setNome(String nome) {
        this.nome = nome; // Método público para acessar o atributo privado
    }

    public String getNome() {
        return nome; // Método público para acessar o atributo privado
    }

    public void fazerSom() {
        System.out.println("Animal faz um som");
    }
}
```

```java
// Subclasse Cachorro
class Cachorro extends Animal {
    public void fazerSom() {
        System.out.println("Cachorro late");
    }

    public void mostrarNome() {
        // Acessando o nome através dos métodos públicos da classe pai
        System.out.println("O nome do cachorro é: " + getNome());
    }
}

// Classe principal para testar
public class Main {
    public static void main(String[] args) {
        Cachorro meuCachorro = new Cachorro();
        meuCachorro.setNome("Rex");
        meuCachorro.fazerSom();   // Saída: Cachorro late
        meuCachorro.mostrarNome(); // Saída: O nome do cachorro é: Rex
    }
}
```

Herança

- A subclasse pode adicionar membros àqueles herdados da classe pai.
  - uma das principais vantagens da herança é a capacidade de uma subclasse não apenas herdar atributos e métodos da classe pai, mas também adicionar novos membros (atributos e métodos) que são específicos à subclasse. Essa capacidade permite a especialização e extensão da funcionalidade da classe pai, sem modificar diretamente a classe original.

```java
// Subclasse Cachorro
class Cachorro extends Animal {
    private String raca; // Novo atributo específico da subclasse

    public void setRaca(String raca) {
        this.raca = raca;
    }

    public String getRaca() {
        return raca;
    }

    // Novo método específico da subclasse
    public void cavar() {
        System.out.println("Cachorro está cavando");
    }

    // Sobrescrevendo o método da superclasse
    @Override
    public void fazerSom() {
        System.out.println("Cachorro late");
    }
}
```

Herança

- A subclasse pode modificar o comportamento de um ou mais métodos herdados. Um método modificado tem o mesmo nome, e geralmente o mesmo protocolo, daquele que está sendo modificado. É dito que o novo método sobrescreve o método herdado, chamado então de método sobrescrito. O propósito mais comum de um método sobrescrever outro é para fornecer uma operação específica para objetos da classe derivada, mas não é apropriado para objetos da classe pai

```java
// Subclasse Cachorro
class Cachorro extends Animal {
    private String raca; // Novo atributo específico da subclasse

    public void setRaca(String raca) {
        this.raca = raca;
    }

    public String getRaca() {
        return raca;
    }

    // Novo método específico da subclasse
    public void cavar() {
        System.out.println("Cachorro está cavando");
    }

    // Sobrescrevendo o método da superclasse
    @Override
    public void fazerSom() {
        System.out.println("Cachorro late");
    }
}
```

Herança

Métodos de instância e variáveis de instância operam apenas nos objetos da classe.

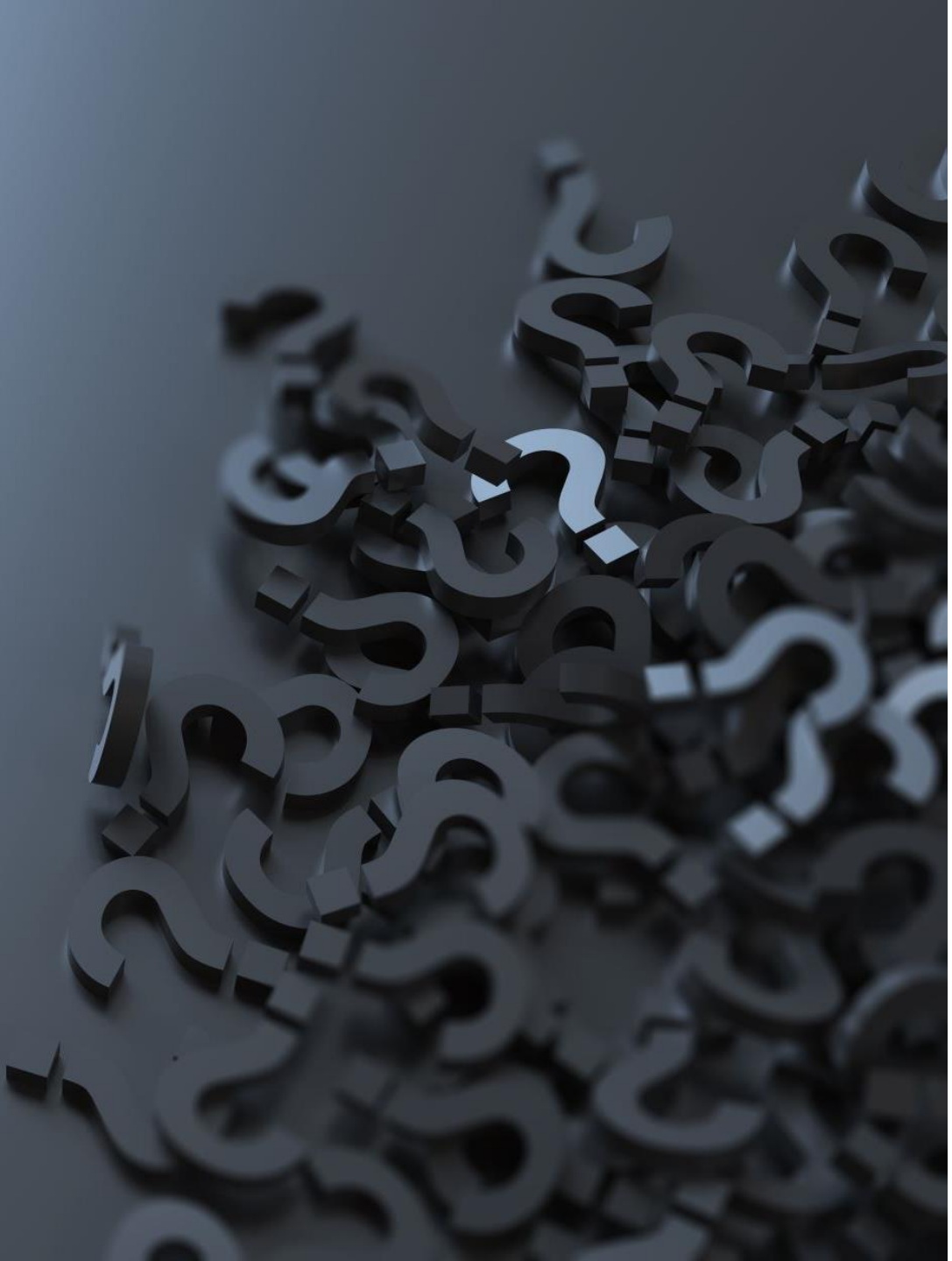
Variáveis de classe pertencem à classe, em vez de ao seu objeto, então existe apenas uma cópia para a classe.

Métodos de classe podem realizar operações na classe e também em objetos da classe.

Se uma nova classe é uma subclasse de uma única classe pai, herança simples.

Se uma classe tem mais de uma classe pai, herança múltipla.

Desvantagem da herança como forma de aumentar a possibilidade de reuso é que ela cria dependências entre classes em uma hierarquia. Isso vai contra uma das vantagens dos tipos abstratos de dados.



Vinculação dinâmica - Polimorfismo

- Permite que objetos de diferentes classes sejam tratados de maneira uniforme através de uma interface comum. Existem dois tipos principais de polimorfismo em POO: polimorfismo em tempo de compilação (também conhecido como sobrecarga de métodos) e polimorfismo em tempo de execução (também conhecido como substituição de métodos ou polimorfismo de subtipos).

```java
// Subclasse Cachorro
class Cachorro extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("Cachorro late");
    }
}

// Subclasse Gato
class Gato extends Animal {
    @Override
    public void fazerSom() {
        System.out.println("Gato mia");
    }
}

// Classe principal para testar
public class Main {
    public static void main(String[] args) {
        Animal meuAnimal1 = new Cachorro();
        Animal meuAnimal2 = new Gato();

        meuAnimal1.fazerSom();  // Saída: Cachorro late
        meuAnimal2.fazerSom();  // Saída: Gato mia
    }
```

Vinculação dinâmica - Polimorfismo

- **Polimorfismo em Tempo de Execução**
  - Uma classe derivada pode sobrescrever métodos da classe base, e o método a ser executado é determinado em tempo de execução, com base no tipo real do objeto. Isso permite que um método tenha comportamentos diferentes dependendo do objeto que o invoca.

Vinculação dinâmica - Polimorfismo

Uma classe abstrata é uma classe que não pode ser instanciada diretamente e serve como um modelo ou base para outras classes.

Ela pode conter métodos abstratos, que são métodos declarados sem implementação.

Classes abstratas e métodos abstratos são utilizados para definir uma interface comum para um grupo de subclasses, garantindo que essas subclasses implementem os métodos essenciais definidos pela classe abstrata.

Características de Classes Abstratas

- **Não Instanciável:** Uma classe abstrata não pode ser usada para criar objetos diretamente. Ela só pode ser utilizada como superclasse.
- **Métodos Abstratos:** Pode conter métodos abstratos que são declarados, mas não implementados. Subclasses concretas são obrigadas a fornecer implementações para esses métodos.
- **Métodos Concretos:** Pode também conter métodos concretos (com implementação), que podem ser herdados pelas subclasses.
- **Definição de Interface Comum:** Fornece uma interface comum que todas as subclasses concretas devem seguir, promovendo a consistência no design.

Características de Métodos Abstratos

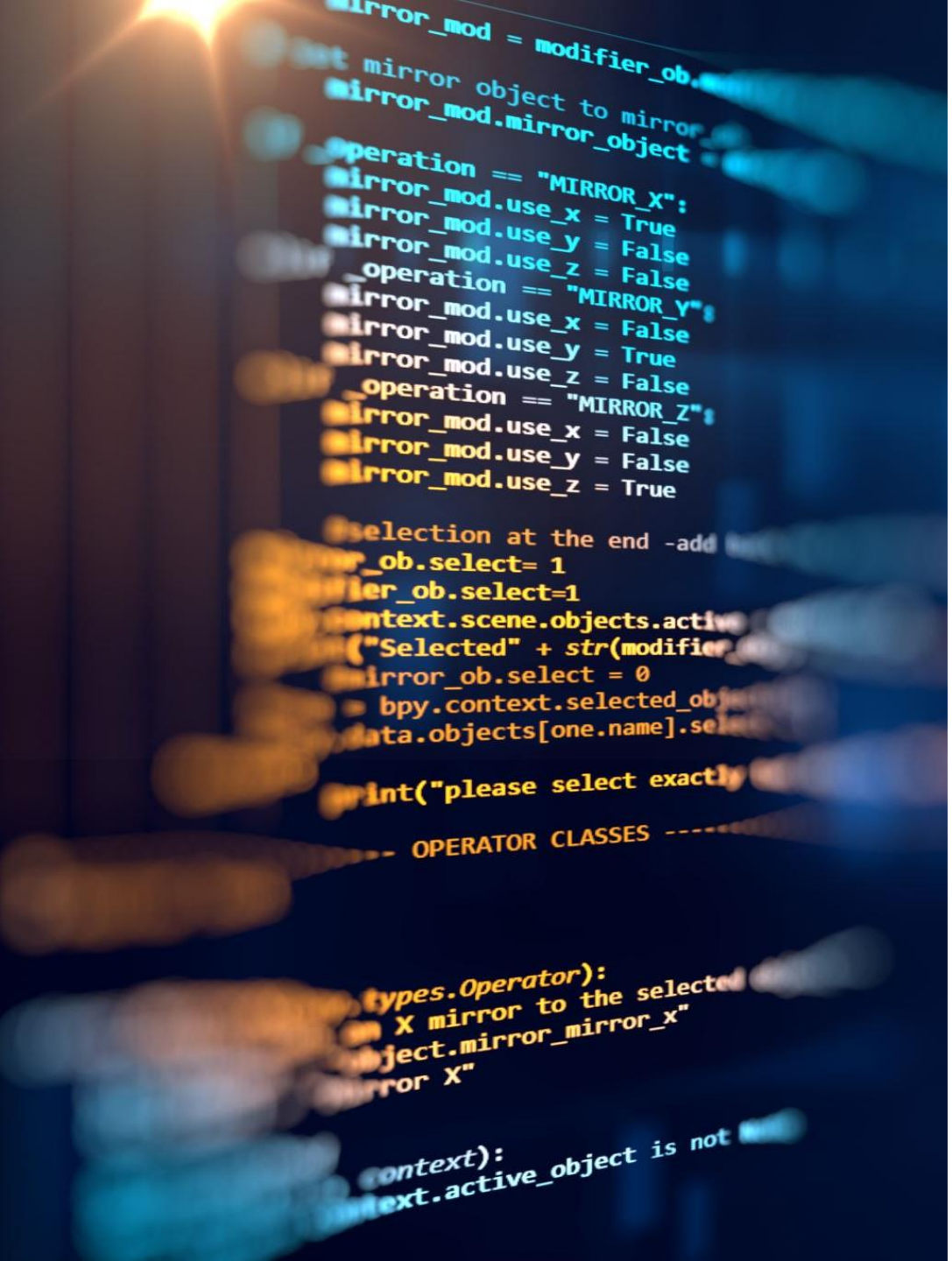
- **Sem Implementação:** Um método abstrato é declarado, mas não tem corpo (implementação).
- **Implementação Obrigatória:** Subclasses que herdam uma classe abstrata devem implementar todos os métodos abstratos da superclasse.
- **Definição de Contrato:** Define um contrato que todas as subclasses devem cumprir, garantindo que certas funcionalidades serão providas.

Exemplo

```java
// Classe abstrata
abstract class Animal {
    private String nome;

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

    public String getNome() {
        return nome;
    }

    // Método abstrato
    public abstract void fazerSom();
}

// Subclasse concreta Cachorro
class Cachorro extends Animal {
    // Implementação do método abstrato
    @Override
    public void fazerSom() {
        System.out.println("Cachorro late");
    }
}

// Subclasse concreta Gato
class Gato extends Animal {
    // Implementação do método abstrato
    @Override
    public void fazerSom() {
        System.out.println("Gato mia");
    }
}

// Classe principal para testar
public class Main {
    public static void main(String[] args) {
        Animal meuCachorro = new Cachorro();
        meuCachorro.setNome("Rex");
        System.out.println(meuCachorro.getNome() + " faz:");
        meuCachorro.fazerSom();  // Saída: Cachorro late

        Animal meuGato = new Gato();
        meuGato.setNome("Felix");
        System.out.println(meuGato.getNome() + " faz:");
        meuGato.fazerSom();  // Saída: Gato mia
    }
}
```

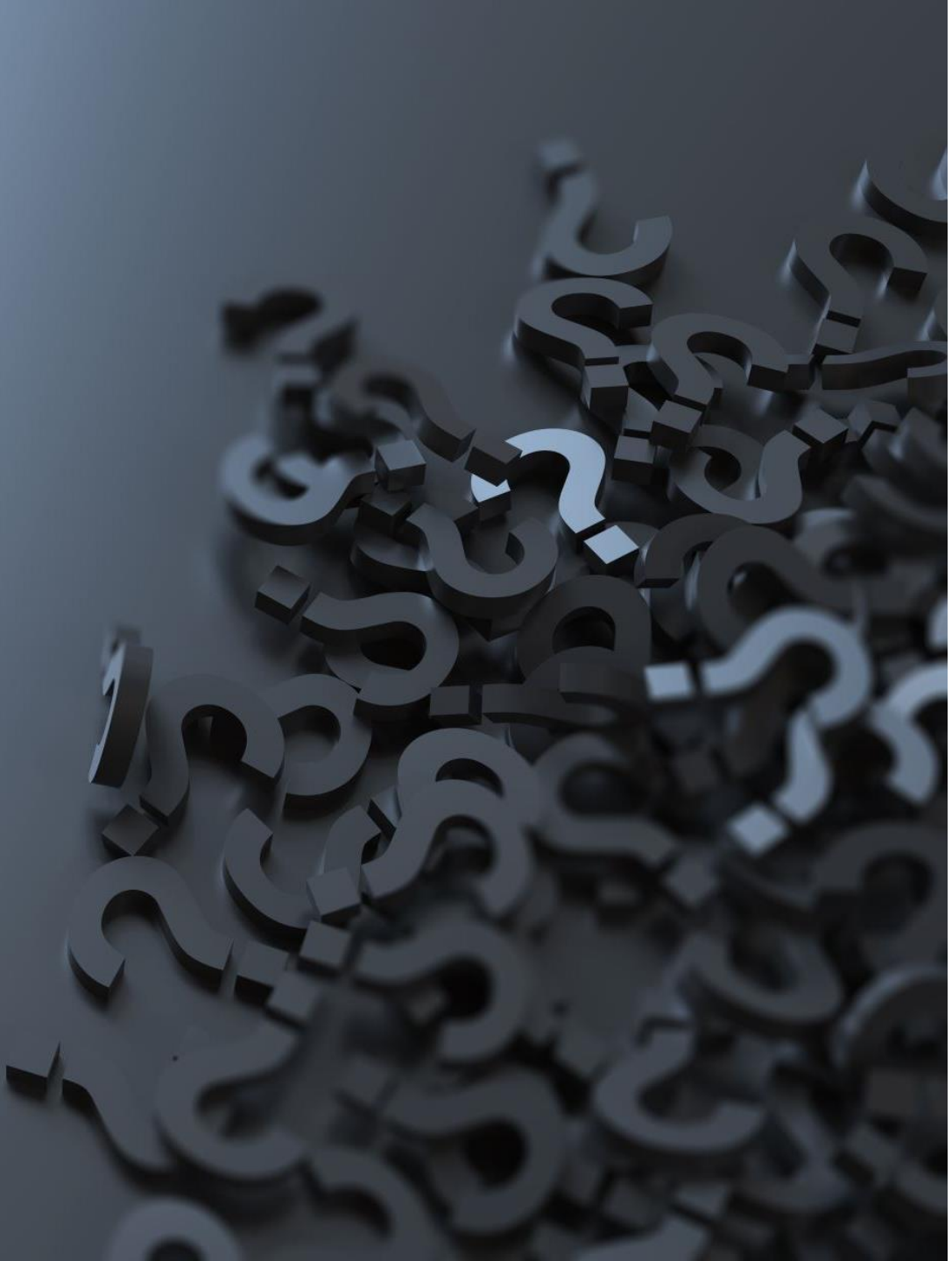
Benefícios de Classes Abstratas e Métodos Abstratos

- **Encapsulamento de Comportamentos Comuns:** Permitem encapsular comportamentos comuns que podem ser compartilhados por várias subclasses.
- **Contratos Claros:** Fornecem uma maneira clara de definir contratos que todas as subclasses devem seguir, garantindo que certas funcionalidades serão implementadas.
- **Facilitam a Extensão:** Facilita a extensão de funcionalidades em um sistema, promovendo reutilização de código e melhor organização.



Benefícios do Polimorfismo

- **Flexibilidade e Extensibilidade**: Permite que o código seja mais flexível e extensível, facilitando a adição de novos tipos sem modificar o código existente.
- **Reutilização de Código**: Promove a reutilização de código, já que o mesmo método pode ser usado para diferentes tipos de objetos.
- **Interoperabilidade**: Facilita a interoperabilidade entre diferentes classes, permitindo que classes de diferentes hierarquias sejam tratadas de forma consistente através de interfaces comuns.

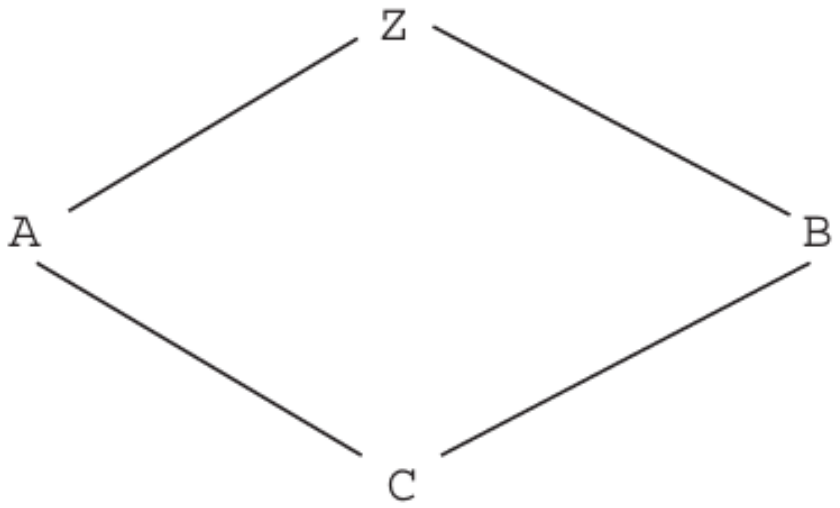


Questões De Projeto Para Programação Orientada A Objetos

- **A exclusividade dos objetos:** ideal seria projetar um sistema de objetos que absorve todos os outros conceitos de tipo;
- **As subclasses são subtipos?** Só se seguir os passos do pai!
- **Verificação de tipos e polimorfismo:** A alternativa óbvia à verificação de tipos estática é prorrogar a verificação de tipos até que a variável polimórfica seja usada para chamar um método.

Questões De Projeto Para Programação Orientada A Objetos

- **Herança simples e múltipla:** complexidade e eficiência. De quem herdo A ou B que vieram de z?
- **Alocação e liberação de objetos**
- **Vinculação estática e dinâmica:** permitir escolha do usuário? Estática mais rápida.
- **Classes aninhadas:** visibilidade dos recursos.
- **Inicialização de objetos:** os objetos devem ser inicializados manual mente ou por meio de algum mecanismo implícito?



Um exemplo de herança diamante.