Programação Orientada a Objetos: Uma Aventura com com Cachorros! 🐕

Vamos desvendar os segredos da POO de um jeito divertido! Prepare-se para uma jornada com nossos amigos caninos.

FIESP SESI SENAI IRS
SENAI
ESCOLA SENAI "A. JACOB LAFER"

Os 4 Pilares da POO: Nossos Heróis!

Encapsulamento

Protege os dados como um cofre.

Herança

Compartilha características como família.

Polimorfismo

Adapta-se a diferentes situações.

Abstração

Simplifica o complexo.

Encapsulamento: O Escudo Protetor

O encapsulamento protege os dados confidenciais. Impede o acesso direto e manipulação indevida. Ele garante a integridade e segurança dos dados.

Benefício	Descrição
Segurança	Protege contra acessos não autorizados
Integridade	Mantém os dados consistentes
Flexibilidade	Permite modificar a implementação sem afetar o resto do código

```python
class Cachorro:
    def __init__(self, nome, comida, sono):
        # Atributos privados (encapsulados)
        self.__nome = nome
        self.__comida = comida
        self.__sono = sono

    # Getters e Setters controlam o acesso
    @property
    def nome(self):
        return self.__nome

    @property
    def comida(self):
        return self.__comida

    @comida.setter
    def comida(self, valor):
        if valor >= 0:  # Validação
            self.__comida = valor

    def comer(self):
        if self.__comida > 0:
            self.__comida -= 1
            print(f"{self.__nome} comeu!")
        else:
            print(f"{self.__nome} está sem comida!")
```

1. Encapsulamento: Segredo Segredo Revelado!

Imagine uma cápsula que protege os dados do nosso cachorro. Só métodos específicos podem acessá-los.

Assim, evitamos "fuçadas" indesejadas e mantemos tudo seguro!

Proteção

Dados seguros e controlados.

Controle

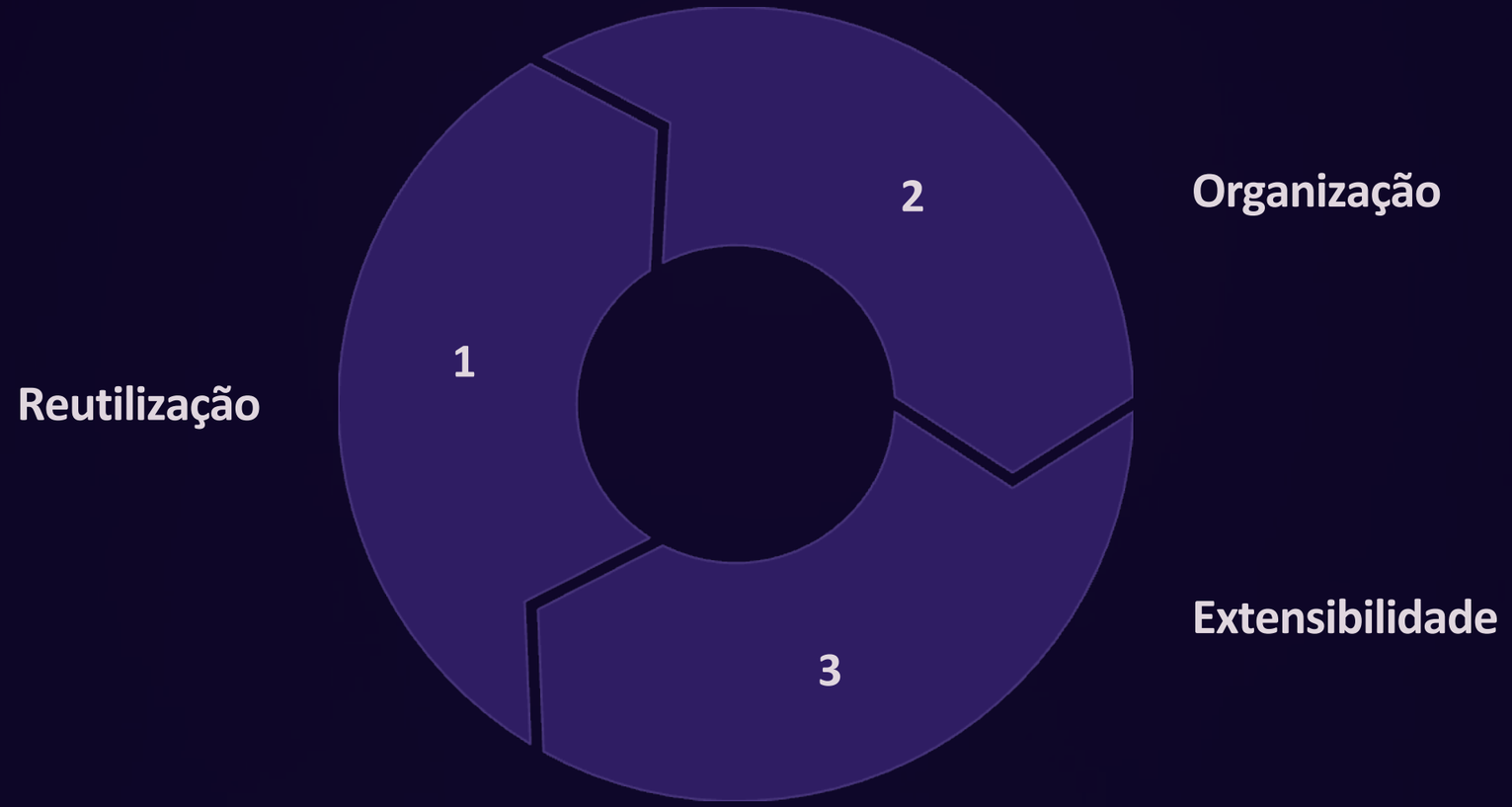
Acesso restrito às informações.

Manutenção

Facilidade para atualizar o sistema.

Herança: Construindo sobre o Legado

A herança permite criar novas classes baseadas em classes existentes. Isso promove a reutilização de código e reduz a duplicação. As subclasses herdam atributos e métodos da superclasse.



2. Herança: De Pai para Filho!

Criamos raças de cachorros a partir de uma classe base.

Cada raça herda características e adiciona suas particularidades.

Reutilização de código é a chave!

Reutilização

Código aproveitado ao máximo!

Hierarquia

Organização clara das classes.

```python
class Animal:
    def __init__(self, nome):
        self.__nome = nome
        self.__energia = 100

    def dormir(self):
        self.__energia = 100

class Cachorro(Animal):
    def __init__(self, nome, comida, sono):
        super().__init__(nome)  # Chama o construtor da classe pai
        self.__comida = comida
        self.__sono = sono

    def latir(self):
        print("Au au!")

class CachorroGuarda(Cachorro):
    def __init__(self, nome, comida, sono, area_patrulha):
        super().__init__(nome, comida, sono)
        self.__area_patrulha = area_patrulha

    def patrulhar(self):
        print(f"Patrulhando a área {self.__area_patrulha}")
```

Polimorfismo: A Arte da Adaptação

O polimorfismo permite que objetos de diferentes classes sejam tratados de forma uniforme. Um método pode ter diferentes implementações em diferentes classes.

1

Flexibilidade

Código adaptável

2

Manutenção

Facilidade para atualizar

3

Extensibilidade

Novas classes facilmente adicionadas

```python
class Animal:
    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "Au au!"

class Gato(Animal):
    def fazer_som(self):
        return "Miau!"

def comunicar_animal(animal):
    print(animal.fazer_som())

# Usando polimorfismo
rex = Cachorro("Rex", 3, False)
felix = Gato("Felix")

comunicar_animal(rex)    # Imprime: Au au!
comunicar_animal(felix)  # Imprime: Miau!
```

3. Polimorfismo: Mil Faces!

Um mesmo método, diferentes ações! Imagine o método "latir".

Cada raça late de um jeito único. Flexibilidade total!

1 Flexibilidade

Código adaptável a diferentes situações.

2 Manutenção

Facilidade para atualizar o sistema.

3 Uniformidade

Objetos tratados de forma igual.

Abstração: Simplificando o Caos

A abstração permite ocultar detalhes complexos e mostrar apenas o essencial. Simplifica o modelo e facilita o entendimento.

4. Abstração: Menos é Mais!

Escondemos detalhes complexos e mostramos apenas o essencial.

Criamos um modelo simplificado do nosso cachorro.

Foco no que importa para o usuário!

1

Simplicidade

Complexidade sob controle.

2

Foco

Essencial em destaque.

3

Facilidade

Desenvolvimento mais rápido.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def fazer_som(self):
        pass

    @abstractmethod
    def mover(self):
        pass

class Cachorro(Animal):
    def __init__(self, nome, comida, sono):
        self.__nome = nome
        self.__comida = comida
        self.__sono = sono

    def fazer_som(self):
        return "Au au!"

    def mover(self):
        return "Andando com 4 patas"
```