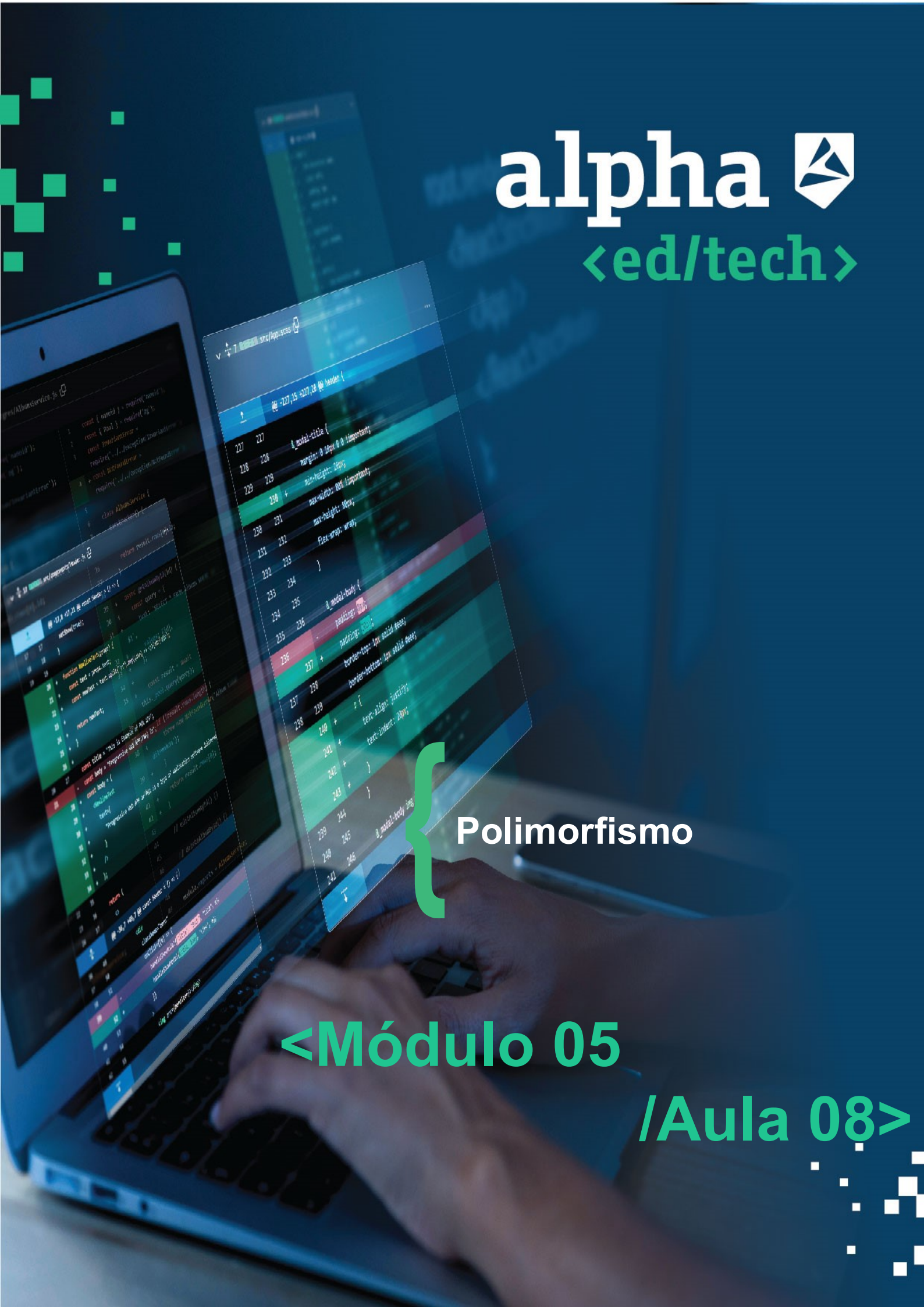
alpha
<ed/tech>
Polimorfismo
<Módulo 05
/Aula 08>

# Polimorfismo

## Motivação para usar Polimorfismo

•Ter uma interface única

•Utilizar um objeto sem saber seu tipo concreto

## Definição de Polimorfismo

**Polimorfismo** significa "**várias formas**".
**Funções polimórficas** são funções que **funcionam para múltiplos tipos** concretos.
O polimorfismo é um **conceito fundamental** na programação **orientada a objetos**, referindo-se à **capacidade de objetos de classes diferentes responderem a uma mesma mensagem de maneira específica a cada classe**.

```python
class Funcionario:
    def cargo(self) -> str:
        return "apenas funcionario“

class Chefe(Funcionario):
    def cargo(self) -> str:
        return "boss de alguem“

class Dono(Funcionario):
    def cargo(self) -> str:
        return "dono da bagaça“

def mostrar_cargo(funcionario: Funcionario) -> None:
    cargo = funcionario.cargo()
    mensagem = f"Nivel Hierarquico: {cargo}“
    print(mensagem)

pessoa1 = Funcionario()
pessoa2 = Chefe()
pessoa3 = Dono()
mostrar_cargo(pessoa1)
mostrar_cargo(pessoa2)
mostrar_cargo(pessoa3)
```

# Polimorfismo

## `Dataclasses` no Polimorfismo

Funções polimórficas podem ser montadas com `dataclasses`, observe o código a seguir:

```python
from dataclasses import dataclass

@dataclass
class Usuario:
    id: int
    nome: str

@dataclass
class Admin(Usuario):
    nivel: int

@dataclass
class UsuarioExterno(Usuario):
    fonte: str

def mostra_usuario(u:Usuario) -> None:
    print("+------------------+")
    print("| Dados do usuario: ")
    print("| id: {u.id}: ")
    print("| nome: {u.nome}: ")
    print("+------------------+")

usuario1 = Admin(1, "Gabriel", 3)
usuario2 = UsuarioExterno(2, "Maria", "Google")

mostra_usuario(usuario1)
mostra_usuario(usuario2)
```

## Tipos de Polimorfismo:

•Polimorfismo de Sobrecarga (Compile-time): Múltiplas funções ou métodos com o mesmo nome, mas diferentes parâmetros.
•Polimorfismo de Substituição (Run-time): Permite que um objeto de uma classe derivada seja tratado como um objeto da classe base.

# Polimorfismo

## Exemplo de Polimorfismo

```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 emitir_som(animal):
       return animal.fazer_som()

cachorro = Cachorro()
gato = Gato()
print(emitir_som(cachorro))  # Saída: Au au!
print(emitir_som(gato))      # Saída: Miau!
```

## Vantagens do Polimorfismo:

•**Flexibilidade**: Facilita a criação de código flexível e expansível.
•**Reutilização de Código**: Permite o uso genérico de classes base para tratar objetos derivados.

## Interfaces e Abstração:

•**Interfaces**: Definem contratos que classes devem cumprir, contribuindo para o polimorfismo.
•**Abstração**: Habilidade de lidar com objetos de forma mais conceitual, ignorando detalhes específicos de implementação.

## Importância no Desenvolvimento:

•**Legibilidade**: Código mais claro e compreensível.
•**Manutenibilidade**: Facilita a manutenção e extensão do sistema.

O polimorfismo contribui para a expressividade e eficácia na modelagem de sistemas, permitindo uma abordagem mais genérica e adaptável às necessidades de programação orientada a objetos.

# Polimorfismo

## Classe Abstrata

Classe Abstrata é uma forma de definir uma interface (conjunto de métodos), sem implementar.

Ela te avisa se faltar algum método ou atributo.

```python
from abc import ABCMeta, abstractmethod

class Funcionario(metaclass=ABCMeta):

    @abstractmethod
    def cargo(self) -> str:
        pass

class Chefe(Funcionario):
    def cargo(self) -> str:
        return "chefe da galera"

class Dono(Funcionario):
    def cargo(self) -> str:
        return "dono"

def mostrar_cargo(funcionario: Funcionario) -> None:
    cargo = funcionario.cargo()
    mensagem = f"Nivel Hierarquico: {cargo}"
    print(mensagem)

pessoa1 = Chefe()
pessoa2 = Dono()

mostrar_cargo(pessoa1)
mostrar_cargo(pessoa2)
```

Note que se removermos a definição do método `cargo` do `Dono` por exemplo, iremos causar um erro, pois ele deve implementar todos os métodos da classe abstrata.

# Polimorfismo

## Protocols

**`Protocols`** são a forma de representar qualquer tipo que tenha determinados atributos e métodos.

Adiciona o conhecido **`duck-typing`**: um pássaro que caminha como um pato, nada como um pato e grasna como um pato, pode ser considerado um pato.

```
from typing import Protocol

class Funcionario(Protocol):
    def cargo(self) -> str:
        pass

class Chefe(Funcionario):
    def cargo(self) -> str:
        return "chefe da galera"

class Dono(Funcionario):
    def cargo(self) -> str:
        return "dono"

def mostrar_cargo(funcionario: Funcionario) -> None:
    cargo = funcionario.cargo()
    mensagem = f"Nivel Hierarquico: {cargo}"
    print(mensagem)

pessoa1 = Chefe()
pessoa2 = Dono()

mostrar_cargo(pessoa1)
mostrar_cargo(pessoa2)
```

A diferença entre classe abstrata e protocolos é que o tipo concreto não precisa depender do protocolo, mas precisa da classe abstrata.