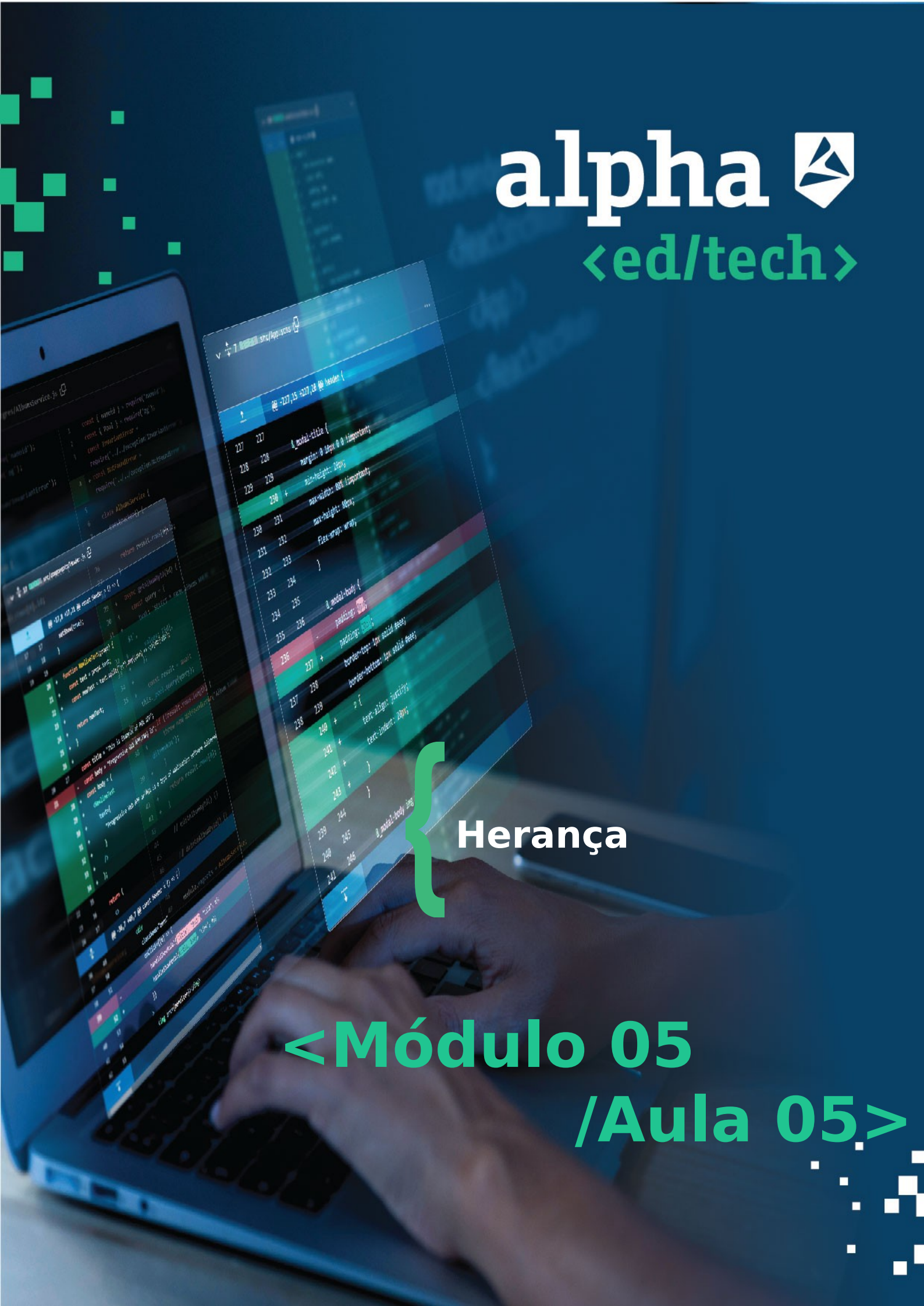

alpha
<ed/tech>
Herança
<Módulo 05
/Aula 05>

Herança

A **herança** é um conceito fundamental na **programação orientada a objetos (OOP)** que permite **criar uma nova classe usando propriedades (atributos) e métodos de uma classe existente**.

A classe existente é chamada de **classe base** ou **classe pai**.
A nova classe é chamada de **classe derivada** ou **classe filha**.

A herança permite a reutilização de código (**reusabilidade**, que é um dos **paradigmas da Orientação a Objetos**) e estabelece uma **relação hierárquica entre classes**.

Herança – usando a declaração class

```python exemplo050501.py
class ClassePai:
    # Atributos e métodos da classe pai

class ClasseFilha(ClassePai):
    # Atributos e métodos adicionais ou modificados
```

Conceitos-chave

Classe Pai (ou Classe Base): É a classe original que fornece características gerais ou comuns.

Classe Filha (ou Classe Derivada): É a nova classe que herda da classe pai. Ela pode adicionar, modificar ou estender os atributos e métodos da classe pai.

Classes `Animal` e `Cachorro`

```python exemplo050502.py
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def fazer_som(self):
        pass

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

    def abanar_rabo(self):
        return "Cachorro está abanando o rabo!"
```

Herança

- `Animal` é a classe pai que tem um método `fazer_som` genérico.

- `Cachorro` é a classe filha que herda de `Animal`. Ele substitui o método `fazer_som` com sua própria implementação e adiciona um novo método `abanar_rabo`.

Usando as Classes definidas anteriormente

```python exemplo050503.py
animal_generico = Animal("Genérico")
print(animal_generico.fazer_som())  # Saída: None

kika = Cachorro("Kika")
print(rex.fazer_som())  # Saída: "Au au!"
print(kika.abanar_rabo())# Saída: "Cachorro está abanando o rabo!"
```

`Cachorro` herda a capacidade de fazer som de `Animal`, mas pode personalizá-la conforme necessário. Isso demonstra como a herança permite a reutilização de código e a criação de hierarquias de classes.

Utilização do método `super()`

O método `super()` é utilizado entre heranças de classes, ele nos proporciona estender/subscrever métodos de uma super classe (classe pai) para uma sub classe (classe filha), através dele se pode definir um novo comportamento para um determinado método construído na classe pai e herdado pela classe filha.

Considere os 2 códigos abaixo como exemplos do uso do método `super()` (observar que qualquer **classe herda da classe geral `object`**)

```python exemplo050504.py
class Pai(object):
    def __init__(self):
        print('Construindo a classe Pai')

class Filha(Pai):
    def __init__(self):
        super(Filha, self).__init__()
```

```python exemplo050505.py
class Pai(object):
  def __init__(self):
    print('Construindo a classe Pai')

class Filha(Pai):
  def __init__(self):
    super().__init__()
```

Herança

No exemplo anterior, temos uma sub classe que herda da super classe e utiliza o `super()` para chamar o método construtor da class `Pai` em seu próprio método construtor. Isso nos proporciona uma melhor **manutenabilidade** do código, tendo em vista que se usássemos de forma direta como no exemplo abaixo:

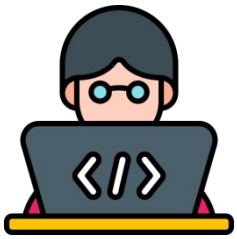
```python exemplo050506.py
class Pai(object):
 def __init__(self):
    print('Construindo a classe Pai')

class Filha(Pai):
 def __init__(self):
    Pai.__init__(self)
```

Fazendo um exemplo mais realista:

```python exemplo050504a.py
class Pai(object):
    def __init__(self, nomePai):
        self.nome = nomePai
        print('Construindo a classe Pai')

    def __str__(self):
        return f"Pai:= {self.nome};"

class Filha(Pai):
    def __init__(self, nomeFilha, nomePai):
        self.nomeFilha = nomeFilha
        super(Filha, self).__init__(nomePai)

    def __str__(self):
        return f"Pai:= {self.nome}; Filha: {self.nomeFilha};"

pmarcotti = Pai("Paulo Marcotti")
print(pmarcotti)

angela = Filha("Angela G Marcotti", pmarcotti.nome)
print(angela)
```

Herança

Teríamos um acesso ao método construtor da classe `Pai` de uma forma amarrada, onde se caso precisássemos modificar a classe herdada, teríamos mais trabalho tendo que modificar em vários outros lugares a mesma, como exemplificado abaixo:



```python exemplo050507.py
class Pai(object):
 def __init__(self):
    print('Construindo a classe Pai')

class Mae(object):
 def __init__(self):
    print('Construindo a classe Mae')

class Filha(Mae):
 def __init__(self):
    Pai.__init__(self)
```

Usando o _super()_ não teríamos tal problema:

```python exemplo050508.py
class Pai(object):
 def __init__(self):
    print('Construindo a classe Pai')

class Mae(object):
 def __init__(self):
    print('Construindo a classe Mae')

class Filha(Mae):
 def __init__(self):
    super().__init__()
```

O código acima executa o método construtor independente da super classe que a sub classe esteja herdando.

Quando precisamos estender métodos da super classe na sub classe, também podemos usar o `super()` a nosso favor. Imagine que precisamos passar novos parâmetros para um método já existente, poderíamos fazer dessa forma:

```python exemplo050509.py
class Pai(object):
  def __init__(self, peso, altura):
     self.peso = peso
     self.altura = altura

class Filha(Pai):
  def __init__(self, peso, altura, cabelo):
     super().__init__(peso, altura)
     self.cabelo = cabelo
```

Herança

Herança Múltipla

A capacidade de herdar características de mais de uma classe. Uma nova classe C que herda da super classe A alguns atributos e comportamentos (métodos) e também herda de outra super classe B alguns outros atributos e outros comportamentos.

Embora a herança múltipla seja um recurso presente em diversas linguagens de programação, o seu uso aumenta a complexidade e dificulta o entendimento.

```python exemplo050510.py
class English(object):
    def greet(self):
        print("Hi!")

class Portuguese(object):
    def greet(self):
        print("Oi!")

class Bilingual(English, Portuguese):
    pass

if __name__ == '__main__':
Bilingual().greet()
```

A classe `English` implementa o método `greet` dizendo “Hi!”

A classe `Portuguese` implementa o método `greet` dizendo “Oi!”

A classe `Bilingual` herda de `English` e `Portuguese` (e por isso tem acesso ao método `greet` de ambas).

Isto é uma herança híbrida. No Python, em casos de herança múltipla, a prioridade é definida **da esquerda para a direita** (como chamada **"left-rigth" fashion**), ou seja:

class Bilingual(English, Portuguese):

Bilingual dará preferência ao método greet de English. O resultado da execução será:

Hi!

Por convenção, programadores Python implementam herança múltipla de forma bem granular, usando Mixins que o código do Django tem muitos bons exemplos disso.

Herança deve ser usada com moderação e em situações apropriadas. Uma hierarquia de classes bem projetada pode melhorar a manutenção do código, mas o uso excessivo de herança pode levar a hierarquias complicadas e difíceis de entender.