**Programação Orientada a Objetos**

# Classe personagem

class Personagem:

    def \_\_init\_\_ (*self*,*nome*,*vida*,*ataque*,*defesa*,*estamina*):

*self*.\_nome = *nome* # Atributo privado

*self*.\_vida = *vida*

*self*.\_ataque = *ataque*

*self*.\_defesa = *defesa*

*self*.\_estamina = *estamina*

    # Método getter para o nome

    def get\_nome(*self*):

        return *self*.\_nome

    # Método setter para o nome

    def set\_nome(*self*, *novo\_nome*):

*self*.\_nome = *novo\_nome*

    def get\_vida(*self*):

        return *self*.\_vida

    def set\_vida(*self*, *nova\_vida*):

*self*.\_vida = *nova\_vida*

    def get\_ataque(*self*):

        return *self*.\_ataque

    def set\_ataque(*self*, *novo\_ataque*):

*self*.\_ataque = *novo\_ataque*

    def get\_defesa(*self*):

        return *self*.\_defesa

    def set\_defesa(*self*, *nova\_defesa*):

*self*.\_defesa = *nova\_defesa*

    def get\_estamina(*self*):

        return *self*.\_estamina

    def set\_estamina(*self*, *nova\_estamina*):

*self*.\_estamina = *nova\_estamina*

# Método abstrato para atacar outro personagem

    def atacar(*self*, *alvo*):

        pass

    # Método abstrato para defender de um ataque

    def defender(*self*, *alvo*):

        pass

    # Método abstrato para recuperar estamina

    def recuperar(*self*):

        pass

    # Método especial para representar o personagem como uma string

    def \_\_str\_\_(*self*):

        return f"{*self*.\_nome}, {*self*.\_vida}, {*self*.\_ataque}, {*self*.\_defesa}, {*self*.\_estamina}"

# Classe de um herói que herda de Personagem

class Heroi(Personagem):

    def \_\_init\_\_(*self*, *nome*, *vida*, *ataque*, *defesa*, *estamina*, *classe*):

        super().\_\_init\_\_(*nome*, *vida*, *ataque*, *defesa*, *estamina*) # Chama o construtor da classe mãe

*self*.classe = *classe* # Atributo público que define a classe do herói

    # Método polimórfico para atacar outro personagem

    def atacar(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            dano = *self*.\_ataque - *alvo*.get\_defesa() # Calcula o dano causado

            if dano > 0: # Verifica se o dano é positivo

*alvo*.set\_vida(*alvo*.get\_vida() - dano) # Reduz a vida do alvo

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} e causou {dano} de dano.")

            else: # Se o dano for negativo ou zero, não há efeito

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} mas não causou dano.")

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode atacar

            print(f"{*self*.\_nome} não tem estamina para atacar.")

    # Método polimórfico para defender de um ataque

    def defender(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            bonus = *self*.\_defesa \* 0.1 # Calcula o bônus de defesa

*self*.set\_defesa(*self*.get\_defesa() + bonus) # Aumenta a defesa do personagem

            print(f"{*self*.\_nome} se defendeu e aumentou sua defesa em {bonus}.")

*alvo*.atacar(*self*) # Chama o método atacar do alvo

            # Restaura a defesa original do personagem

*self*.set\_defesa(*self*.get\_defesa() - bonus)

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode defender

            print(f"{*self*.\_nome} não tem estamina para se defender.")

    # Método polimórfico para recuperar estamina

    def recuperar(*self*):

        if *self*.\_estamina < 10: # Verifica se a estamina está abaixo do máximo

            recupera = 2 # Define a quantidade de estamina recuperada

*self*.set\_estamina(*self*.get\_estamina() + recupera) # Aumenta a estamina do personagem

            print(f"{*self*.\_nome} recuperou {recupera} de estamina.")

        else: # Se a estamina está no máximo, não há efeito

            print(f"{*self*.\_nome} já está com a estamina máxima.")

    # Método especial para representar o herói como uma string

    def \_\_str\_\_(*self*):

        return f"{super().\_\_str\_\_()}, {*self*.classe}" # Chama o método da classe mãe e adiciona a classe

# Classe de um vilão que herda de Personagem

class Vilao(Personagem):

    def \_\_init\_\_(*self*, *nome*, *vida*, *ataque*, *defesa*, *estamina*, *maldade*):

        super().\_\_init\_\_(*nome*, *vida*, *ataque*, *defesa*, *estamina*) # Chama o construtor da classe mãe

*self*.maldade = *maldade* # Atributo público que define o nível de maldade do vilão

    # Método polimórfico para atacar outro personagem

    def atacar(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            dano = *self*.\_ataque + *self*.maldade - *alvo*.get\_defesa() # Calcula o dano causado com o bônus de maldade

            if dano > 0: # Verifica se o dano é positivo

*alvo*.set\_vida(*alvo*.get\_vida() - dano) # Reduz a vida do alvo

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} e causou {dano} de dano.")

            else: # Se o dano for negativo ou zero, não há efeito

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} mas não causou dano.")

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode atacar

            print(f"{*self*.\_nome} não tem estamina para atacar.")

# Método polimórfico para defender de um ataque

    def defender(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            bonus = *self*.\_defesa \* 0.1 # Calcula o bônus de defesa

*self*.set\_defesa(*self*.get\_defesa() + bonus) # Aumenta a defesa do personagem

            print(f"{*self*.\_nome} se defendeu e aumentou sua defesa em {bonus}.")

*alvo*.atacar(*self*) # Chama o método atacar do alvo

*self*.set\_defesa(*self*.get\_defesa() - bonus) # Restaura a defesa original do personagem

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode defender

            print(f"{*self*.\_nome} não tem estamina para se defender.")

    # Método polimórfico para recuperar estamina

    def recuperar(*self*):

        if *self*.\_estamina < 10: # Verifica se a estamina está abaixo do máximo

            recupera = 2 + *self*.maldade // 10 # Define a quantidade de estamina recuperada com o bônus de maldade

*self*.set\_estamina(*self*.get\_estamina() + recupera) # Aumenta a estamina do personagem

            print(f"{*self*.\_nome} recuperou {recupera} de estamina.")

        else: # Se a estamina está no máximo, não há efeito

            print(f"{*self*.\_nome} já está com a estamina máxima.")

    # Método especial para representar o vilão como uma string

    def \_\_str\_\_(*self*):

        return f"{super().\_\_str\_\_()}, {*self*.maldade}" # Chama o método da classe mãe e adiciona a maldade

# Classe de um mago que herda de herói

class Mago(Heroi):

    def \_\_init\_\_(*self*, *nome*, *vida*, *ataque*, *defesa*, *estamina*, *classe*, *magia*):

        super().\_\_init\_\_(*nome*, *vida*, *ataque*, *defesa*, *estamina*, *classe*) # Chama o construtor da classe mãe

*self*.magia = *magia* # Atributo público que define o nível de magia do mago

    # Método polimórfico para atacar outro personagem

    def atacar(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            dano = *self*.\_ataque + *self*.magia - *alvo*.get\_defesa() # Calcula o dano causado com o bônus de magia

            if dano > 0: # Verifica se o dano é positivo

*alvo*.set\_vida(*alvo*.get\_vida() - dano) # Reduz a vida do alvo

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} e causou {dano} de dano.")

            else: # Se o dano for negativo ou zero, não há efeito

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} mas não causou dano.")

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode atacar

            print(f"{*self*.\_nome} não tem estamina para atacar.")

    # Método polimórfico para defender de um ataque

    def defender(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            bonus = *self*.\_defesa \* 0.1 + *self*.magia \* 0.05 # Calcula o bônus de defesa com o bônus de magia

*self*.set\_defesa(*self*.get\_defesa() + bonus) # Aumenta a defesa do personagem

            print(f"{*self*.\_nome} se defendeu e aumentou sua defesa em {bonus}.")

*alvo*.atacar(*self*) # Chama o método atacar do alvo

*self*.set\_defesa(*self*.get\_defesa() - bonus) # Restaura a defesa original do personagem

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode defender

            print(f"{*self*.\_nome} não tem estamina para se defender.")

    # Método polimórfico para recuperar estamina

    def recuperar(*self*):

        if *self*.\_estamina < 10: # Verifica se a estamina está abaixo do máximo

            recupera = 2 + *self*.magia // 10 # Define a quantidade de estamina recuperada com o bônus de magia

*self*.set\_estamina(*self*.get\_estamina() + recupera) # Aumenta a estamina do personagem

            print(f"{*self*.\_nome} recuperou {recupera} de estamina.")

        else: # Se a estamina está no máximo, não há efeito

            print(f"{*self*.\_nome} já está com a estamina máxima.")

    # Método especial para representar o mago como uma string

    def \_\_str\_\_(*self*):

        return f"{super().\_\_str\_\_()}, {*self*.magia}" # Chama o método da classe mãe e adiciona a magia

# Classe de um guerreiro que herda de herói

class Guerreiro(Heroi):

    def \_\_init\_\_(*self*, *nome*, *vida*, *ataque*, *defesa*, *estamina*, *classe*, *forca*):

        super().\_\_init\_\_(*nome*, *vida*, *ataque*, *defesa*, *estamina*, *classe*) # Chama o construtor da classe mãe

*self*.forca = *forca* # Atributo público que define o nível de força do guerreiro

    # Método polimórfico para atacar

    def atacar(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            dano = *self*.\_ataque + *self*.forca - *alvo*.get\_defesa() # Calcula o dano causado com o bônus de força

            if dano > 0: # Verifica se o dano é positivo

*alvo*.set\_vida(*alvo*.get\_vida() - dano) # Reduz a vida do alvo

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} e causou {dano} de dano.")

            else: # Se o dano for negativo ou zero, não há efeito

                print(f"{*self*.\_nome} atacou {*alvo*.get\_nome()} mas não causou dano.")

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode atacar

            print(f"{*self*.\_nome} não tem estamina para atacar.")

    # Método polimórfico para defender de um ataque

    def defender(*self*, *alvo*):

        if *self*.\_estamina > 0: # Verifica se tem estamina suficiente

            bonus = *self*.\_defesa \* 0.1 + *self*.forca \* 0.05 # Calcula o bônus de defesa com o bônus de força

*self*.set\_defesa(*self*.get\_defesa() + bonus) # Aumenta a defesa do personagem

            print(f"{*self*.\_nome} se defendeu e aumentou sua defesa em {bonus}.")

*alvo*.atacar(*self*) # Chama o método atacar do alvo

*self*.set\_defesa(*self*.get\_defesa() - bonus) # Restaura a defesa original do personagem

*self*.\_estamina -= 1 # Reduz a estamina do personagem

        else: # Se não tem estamina suficiente, não pode defender

            print(f"{*self*.\_nome} não tem estamina para se defender.")

    # Método polimórfico para recuperar estamina

    def recuperar(*self*):

        if *self*.\_estamina < 10: # Verifica se a estamina está abaixo do máximo

            recupera = 2 + *self*.forca // 10 # Define a quantidade de estamina recuperada com o bônus de força

*self*.set\_estamina(*self*.get\_estamina() + recupera) # Aumenta a estamina do personagem

            print(f"{*self*.\_nome} recuperou {recupera} de estamina.")

        else: # Se a estamina está no máximo, não há efeito

            print(f"{*self*.\_nome} já está com a estamina máxima.")

# Método especial para representar o guerreiro como uma string

    def \_\_str\_\_(*self*):

        return f”{super().\_\_str\_\_()}, {*self*.forca}” # Chama o método da classe mãe e adiciona a força

# Cria um herói do tipo mago chamado Astolfo com vida 100, ataque 10, defesa 5, estamina 10 e magia 20

Astolfo = Mago(“Astolfo”, 100, 10, 5, 10, “Mago”, 20)

print(Astolfo)

# Cria um herói do tipo guerreiro chamado Arthur com vida 120, ataque 15, defesa 10, estamina 10 e força 15

rthur = Guerreiro(“Arthur”, 120, 15, 10, 10, “Guerreiro”, 15)

print(rthur)

# Cria um vilão chamado FelipeNeto com vida 80, ataque 12, defesa 8, estamina 10 e maldade 25

FelipeNeto = Vilao(“FelipeNeto”, 80, 12, 8, 10, 25)

print(FelipeNeto)

# Simula uma batalha entre os personagens

Astolfo.atacar(FelipeNeto)

FelipeNeto.defender(Astolfo)

Arthur.atacar(FelipeNeto)

FelipeNeto.atacar(Arthur)

Astolfo.defender(FelipeNeto)

FelipeNeto.recuperar()

Arthur.recuperar()

Astolfo.recuperar()