



# Fundamentos de Programação (T)

Licenciatura em Videojogos

Ano 1 / Semestre 1

## Informações Importantes

- Aula de compensação para revisões
  - Quinta-feira 4/12 das 11 às 13 horas
  - Sala U.2.8
- Teste teórico:
  - Quarta-feira → 10/12/2025 às 15 horas
  - Sala U.0.1

# Sumário

- Programação Orientada a Objectos:
  - Herança
  - Polimorfismo

# Classes e objectos

- Classe = "*blueprint*" que descreve:
  - Atributos (dados)
  - Métodos (comportamentos)
- Objecto = instância concreta de uma classe

```
class Item:
    def __init__(self, name, type):
        self.name = name
        self.type = type

    def Log(self):
        print("Name:", self.name, "| Type:", self.type)
```



# Herança

# O que é Herança?

- Anteriormente vimos isto (aula 7):

```
class InvalidPosition(Exception):  
    pass
```

- Em contrast com isto (aula 8):

```
class Room:  
    pass
```

Podemos usar o `try: / except:`  
da classe `Exception()`

Ambas são classes, tipologias que criámos para se adaptarem às nossas necessidades.

**Qual a diferença?**

`Room()` é uma classe, `InvalidPosition()` é uma subclasse que herda atributos e métodos da classe `Exception()`

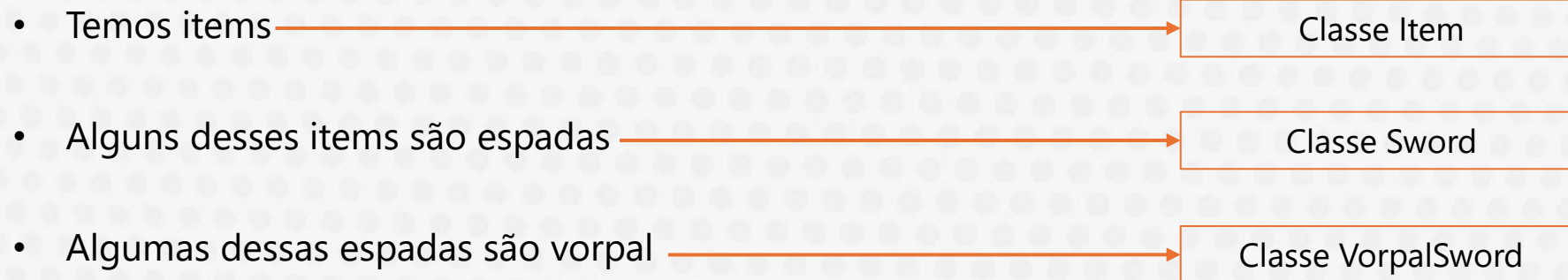
O que significa na prática é que podemos usar a excepção que criámos como se fosse qualquer outra excepção

# O que é Herança?

- Permite criar novas classes a partir de uma classe existente.
- A classe "base" fornece atributos e métodos comuns.
- As subclasses **herdam** esse comportamento e podem:
  - Reutilizar tal como está.
  - Adicionar novos atributos/métodos
  - Alterar (sobrescrever) métodos existentes
- Relação típica:
  - Num jogo temos items
  - Um medkit é um tipo de item
  - Uma arma é outro tipo de item
  - Uma espada é um tipo de arma

# Herança

- Vamos imaginar um RPG:



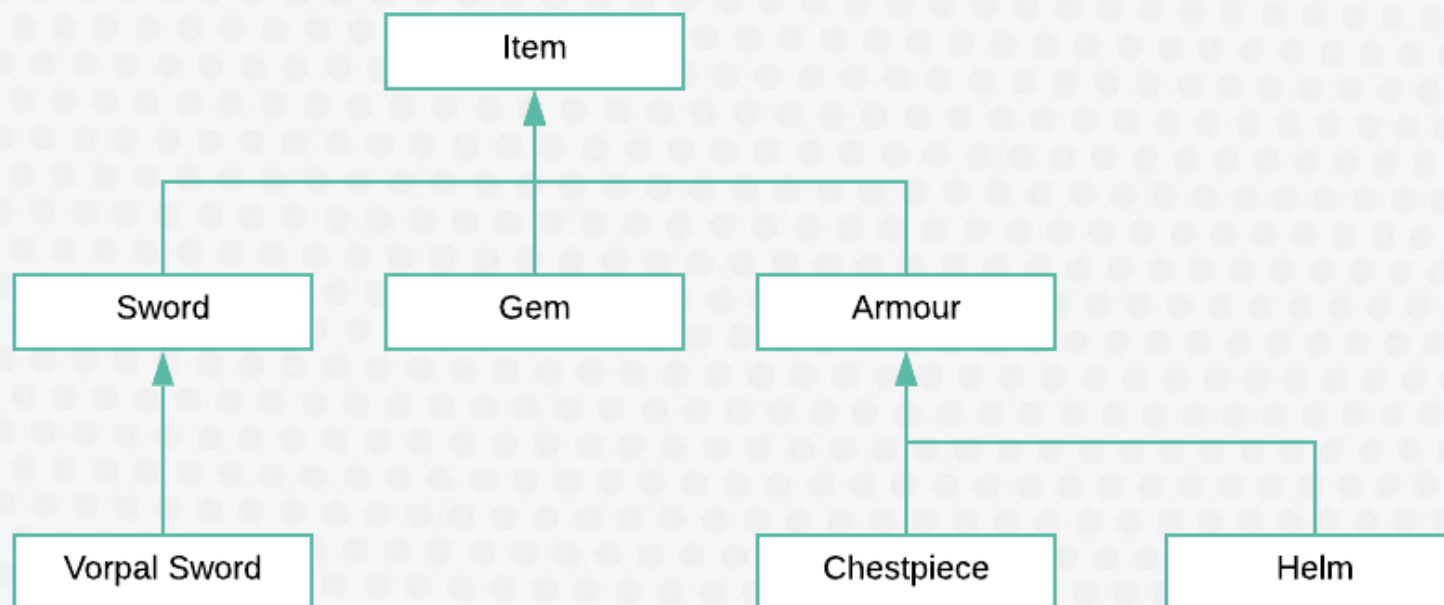
# Herança

- Vamos imaginar um RPG:
  - Temos items (Classe Item)
    - Podem cair dos inimigos
    - Podem ser apanhados
    - Podem ser vendidos por um certo valor
    - Têm uma gráfico/imagem/ícone no inventário
  - Alguns desses items são espadas (Classe Sword)
    - Podem ser equipados na mão do personagem
    - Podem ser usadas em combate
  - Algumas dessas espadas são vorpal (Classe VorpalSword)
    - Têm atributos de magia e quando são usadas num ataque e esse ataque é um crítico, decapita o inimigo
  - Uma vorpal sword pode ser vendida?
    - Claro que sim, mas teríamos de duplicar código por todo o lado, se não tivéssemos herança.



# Herança

- A herança permite-nos definir funcionalidade geral, e especializar quando é necessário.



- As subclasses herdam automaticamente tudo o que pertence à classe base, sem necessidade de repetir código.



# Herança

- A herança permite-nos definir funcionalidade geral, e especializar quando é necessário.
- Vamos ver o código:

```
class Item:
    pass

class Sword(Item):
    pass

class Gem(Item):
    pass

class Armor(Item):
    pass

class VorpalSword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass
```

```
item = Helm()

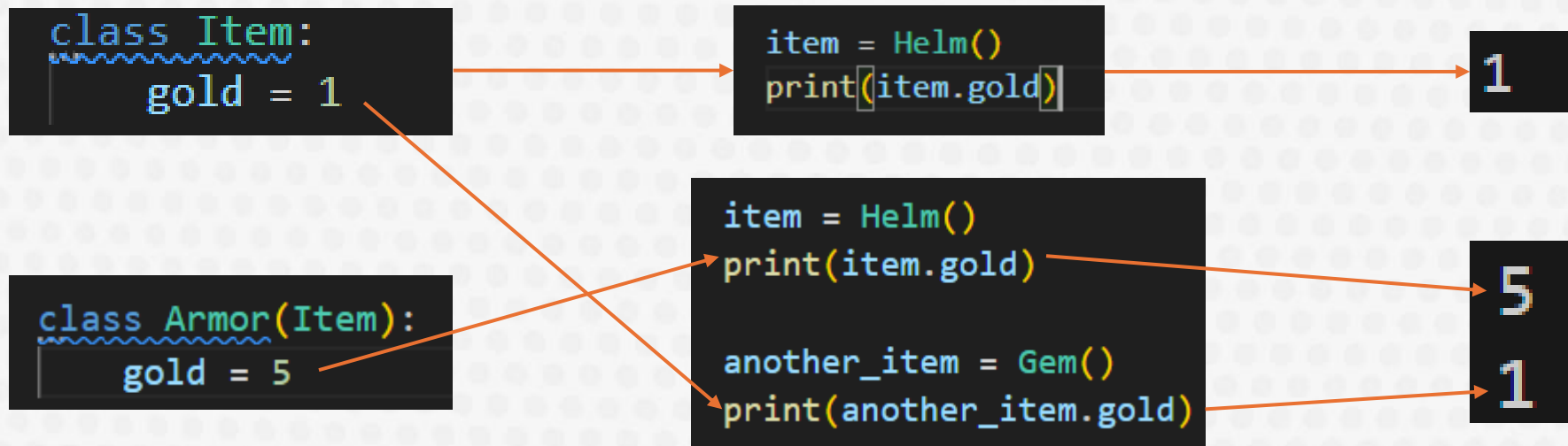
print(isinstance(item, Item))
print(isinstance(item, Armor))
print(isinstance(item, Helm))
print(isinstance(item, Sword))
```

```
True
True
True
False
```



# Herança

- Agora podemos começar a alterar as classes e subclasses:



# Herança

- Podemos também criar funções dentro da herança:

```
class Item:
    gold = 1

    def canSell(self):
        return(self.gold > 0)

class Sword(Item):
    gold = 0

class Gem(Item):
    pass

class Armor(Item):
    gold = 5

class VorpalSword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass

item = Helm()
another_item = Sword()
print(item.canSell(), another_item.canSell())
```

True False



# Herança

- Mas a parte interessante é quando começamos a fazer *override* das funções...

```
class Item:
    gold = 1

    def canSell(self):
        return(self.gold > 0)

class Sword(Item):
    cursed = False

    def canSell(self):
        if (self.cursed):
            return False

        return(self.gold > 0)

class Gem(Item):
    pass

class Armor(Item):
    gold = 5

class Vorpalsword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass
```

```
item = Vorpalsword()
print(item.canSell())
```

True



# Herança

- Mas a parte interessante é quando começamos a fazer *override* das funções...

```
class Item:
    gold = 1

    def canSell(self):
        return(self.gold > 0)

class Sword(Item):
    cursed = False

    def canSell(self):
        if (self.cursed):
            return False

        return(self.gold > 0)

class Gem(Item):
    pass

class Armor(Item):
    gold = 5

class Vorpalsword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass
```

```
item = Vorpalsword()
item.cursed = True
print(item.canSell())
```

False



# Herança

- Mas para não termos que reescrever código:

Código repetido

```
class Item:
    gold = 1

    def canSell(self):
        return(self.gold > 0)

class Sword(Item):
    cursed = False

    def canSell(self):
        if (self.cursed):
            return False
        return(self.gold > 0)

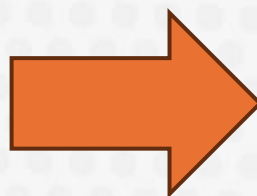
class Gem(Item):
    pass

class Armor(Item):
    gold = 5

class Vorpalsword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass
```



```
class Item:
    gold = 1

    def canSell(self):
        return(self.gold > 0)

class Sword(Item):
    cursed = False

    def canSell(self):
        if (self.cursed):
            return False
        return super().canSell()
```

`super()` permite chamar métodos da classe base, evitando repetir código já existente.

# Herança - Resumidamente

- Herança não serve para criar coisas diferentes. Serve para organizar, reutilizar e especializar funcionalidade.
- Exemplo:

```
class Item:
    def canSell(self):
        return True

class Sword(Item):
    def canSell(self):
        return False
```

- A base (**Item**) define o comportamento geral. A subclasse (**Sword**) decide quando precisa de ser diferente.

# Herança

- Um exemplo mais completo

```
class Item:
    gold = 1

    def __init__(self, name, gold):
        self.name = name
        self.gold = gold

    def display(self):
        print("Name:", self.name)
        print("Sell value:", self.gold)

class Sword(Item):
    damage = 0

    def __init__(self, damage):
        super().__init__("Sword", 1)
        self.damage = damage

    def display(self):
        super().display()
        print("Damage:", self.damage)

class Gem(Item):
    def __init__(self):
        super().__init__("Gem", 10)

class Armor(Item):
    protection = 0

    def __init__(self, protection):
        super().__init__("Armor", 5)
        self.protection = protection

    def display(self):
        super().display()
        print("Protection:", self.protection)

inventory = [Sword(3), Armor(4), Gem(), Item("VendorTrash", 2)]

for item in inventory:
    item.display()
    print("-----")
```



```
• Name: Sword
  Sell value: 1
  Damage: 3
  -----
  Name: Armor
  Sell value: 5
  Protection: 4
  -----
  Name: Gem
  Sell value: 10
  -----
  Name: VendorTrash
  Sell value: 2
  -----
```



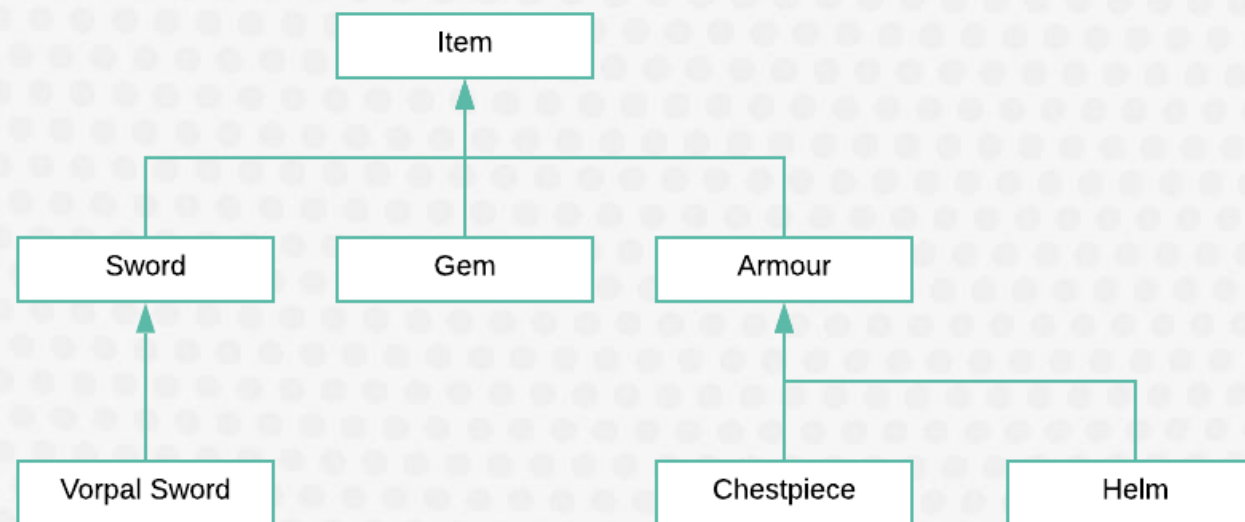


# Polimorfismo



# Polimorfismo

- Polimorfismo é a capacidade de um mesmo objecto ser tratado como vários tipos diferentes.
- Isto só é possível porque existe herança.
- *Helm* é um *Armor*, que é um *Item*.
- Portanto quando necessário, *Helm* pode agir como um *Armor* ou como um *Item*.





# Polimorfismo

- Como vimos anteriormente:

```
class Item:
    pass

class Sword(Item):
    pass

class Gem(Item):
    pass

class Armor(Item):
    pass

class Vorpalsword(Sword):
    pass

class ChestPiece(Armor):
    pass

class Helm(Armor):
    pass
```

```
item = Helm()

print(isinstance(item, Item))
print(isinstance(item, Armor))
print(isinstance(item, Helm))
print(isinstance(item, Sword))
```

True  
True  
True  
False

- ***Helm* é instancia de *Item*, é também instancia de *Armor* e do próprio *Helm*.**

# Polimorfismo

- Polimorfismo em acção:

```
items = [Sword(), Vorpalsword(), Helm()]  
  
for i in items:  
    i.canSell()
```

- Todos os objectos são instâncias da classe base Item.
- O método é o mesmo.
- O comportamento pode ser diferente => Se houver override, se não seguem o canSell() de Items().
- Não precisamos saber que tipo é cada objecto. Basta sabermos que é um Item.



# Polimorfismo

- Com override e sem override:

```
class Item:
    def use(self):
        print("Item used.")

    def canSell(self):
        print("This is a general item and can be sold")

class Sword(Item):
    def use(self):
        print("You attacked with the sword")

class Armor(Item):
    def use(self):
        print("You are now wearing your armor!")

class Helm(Armor):
    def use(self):
        print("You put on your helm!")

class Vorpalsword(Sword):
    def use(self):
        print("Your sword does a magical attack!")

class Gem(Item):
    pass
```

```
items = [Sword(), Armor(), Helm(), Vorpalsword(), Gem()]

for item in items:
    item.canSell()
    item.use()
    print("-----")
```

```
This is a general item and can be sold
You attacked with the sword
-----
This is a general item and can be sold
You are now wearing your armor!
-----
This is a general item and can be sold
You put on your helm!
-----
This is a general item and can be sold
Your sword does a magical attack!
-----
This is a general item and can be sold
Item used.
-----
```

# Polimorfismo - Resumo

- É a capacidade de objectos diferentes partilharem a mesma interface, e de um mesmo objecto ser tratado como vários tipos.
- A mesma função pode ter comportamentos distintos consoante a classe do objecto.
- Isto só é possível devido à herança.
- Permite escrever código mais simples, genérico e reutilizável.
- Recuperando o exemplo:

```
items = [Sword(), Vorpalsword(), Helm()]  
for i in items:  
    i.canSell()
```

- Todos são tratados como `Item`.
- O método é o mesmo mas o resultado pode mudar.
- Chamamos o mesmo método. O objecto decide como responder. Dependendo da existência de *overrides*.

# Polimorfismo - Resumo

- Recuperando outro exemplo:

```
items = [Sword(), Armor(), Helm(), VorpalSword(), Gem()]
for item in items:
    item.canSell()
    item.use()
    print("-----")
```

- Todos os objectos têm um *override* da função `use()`, excepto `Gem()`.
- Logo todos eles irão ter comportamentos diferentes, excepto `Gem()` que na ausência de *override* vai reutilizar a função `use()` de `Item()`.
- Nenhum dos objectos faz *override* de `canSell()` logo reutilizam a função declarada no parent `Item()`
- Sem *override*, o comportamento da classe base é herdado automaticamente.

```
This is a general item and can be sold
You attacked with the sword
-----
This is a general item and can be sold
You are now wearing your armor!
-----
This is a general item and can be sold
You put on your helm!
-----
This is a general item and can be sold
Your sword does a magical attack!
-----
This is a general item and can be sold
Item used.
-----
```



# Exercícios

# Exercícios

Exemplo de execução

1. Criar a base do sistema de itens do jogo:
  - Criar a classe base Item com:
    - atributos: name, value
    - método use() que imprime: "Usaste o item: <nome>"
  - Criar as subclasses:
    - Potion
    - Sword
    - Gem
  - Cada subclasse deve herdar de Item mas:
    - Deve apenas inicializar os seus próprios valores

# Exercícios

Exemplo de execução

2. Agora pega na estrutura do Exercício 1 e modifica-a:

- Fazer override do método use() em:
  - Potion: "You gained 20 health."
  - Sword: "You used the sword to attack"
- Criar uma lista:
  - `inventory = [Potion("Health Potion", 10), Sword("Iron Sword", 40), Gem("Ruby", 100)]`
- Percorrer a lista e chamar use() em todos os items.

# Exercícios

Exemplo de execução

3. Sistema de combate interativo
  - Classes base (recuperadas dos exercícios anteriores)
    - Item com método use(target)
    - Subclasses:
      - Sword → reduz 15 HP do inimigo
      - Potion → recupera 20 HP do jogador
      - Gem → comportamento base (sem efeito)
  - Criar classes Player e Enemy inicializadas com nome e hp. O nome do jogador deverá ser recebido por input
    - Criar uma função de spawn de inimigos que receba a seguinte lista e faça spawn aleatório:
      - `enemy_names = ["Goblin", "Orc", "Skeleton", "Dark Knight", "Slime"]`
      - Inimigos devem ter hp aleatório entre 30 e 50
  - Programa:
    - Criar um while infinito que:
      - Gera um inimigo com spawn\_enemy()
      - Mostra:
        - Nome e HP do inimigo
        - HP do jogador
        - Inventário numerado
        - Pedir input ao jogador, com opção de sair
      - Aplica o item escolhido usando polimorfismo
      - Quando o inimigo chega a 0 HP:
        - Imprimir: "Enemy defeated"
        - Gera novo inimigo
      - O jogo termina apenas se:
        - o jogador escrever exit
        - ou depois de derrotar 5 inimigos