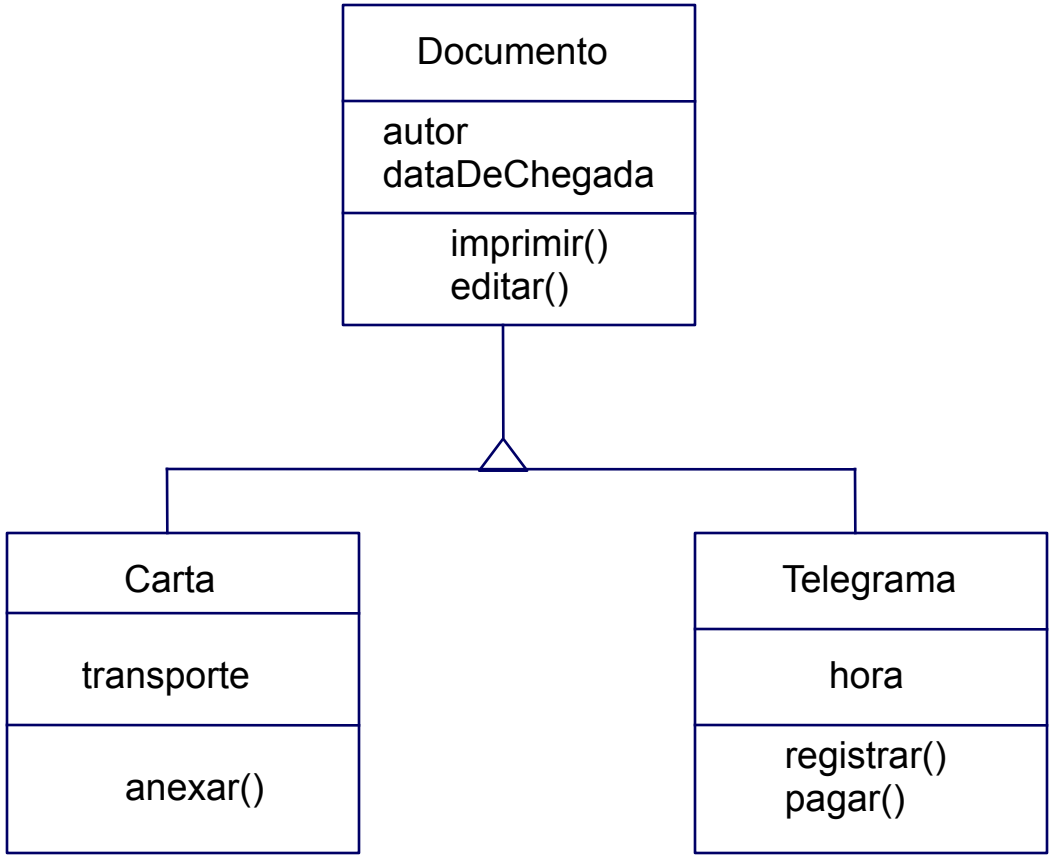
# Herança

- Exemplo de herança
- Herança em Python
- Formas de herança
- Usos de herança
- Herança múltipla
- Agregação X Generalização
- Classes Abstratas X Classes Concretas

# Herança (1)

- Herança é um mecanismo para derivar novas classes a partir de classes existentes.
- A classe derivada herda a representação de dados e operações de sua classe base.
- Pode-se adicionar novas operações, estender a representação dos dados ou redefinir a implementação de operações existentes.

# Herança (2)


Documento
autor
dataDeChegada
imprimir()
editar()
Carta
transporte
anexar()
Telegrama
hora
registrar()
pagar()

# Herança (3)

- **Classe Derivada** ou **Subclasse** ou **Classe Filha**: é uma classe que herda parte dos seus atributos e métodos de outra classe.

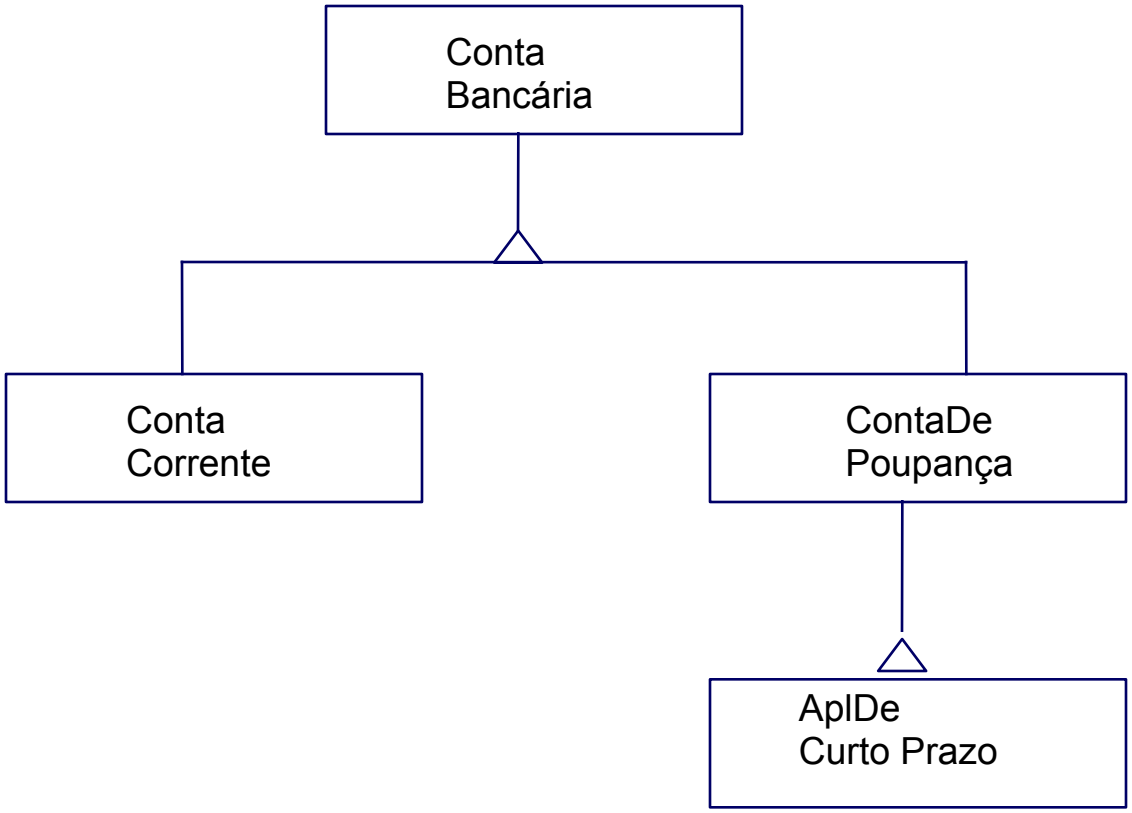
- **Classe Base** ou **Superclasse** ou **Classe Pai**: é uma classe a partir da qual classes novas podem ser derivadas.

# Herança (4)

- **Classes Ancestrais** são aquelas das quais uma superclasse herda.

- **Classes Descendentes** são aquelas que herdam de uma subclasse.

# Exemplo de herança



# Herança em Python (1)

```
class ContaBancaria:

  def __init__ (self, sal=0):
    self.__saldo = sal


  def  deposita(self, valor):
    self.__saldo = self.__saldo + valor


  def retornaSaldo(self):
     return self.__saldo
```

# Herança em Python (2)

```
class ContaDePoupanca(ContaBancaria):
  def __init__(self):

       self.__indice = 0

  def calcula():

  def retira():
```

# Herança em Python (3)

```
class ContaCorrente (ContaBancaria):
  def __init__(self):

     self.__limite = 0

     self.__taxa = 0

  def descontaCheque(self, valor):
```

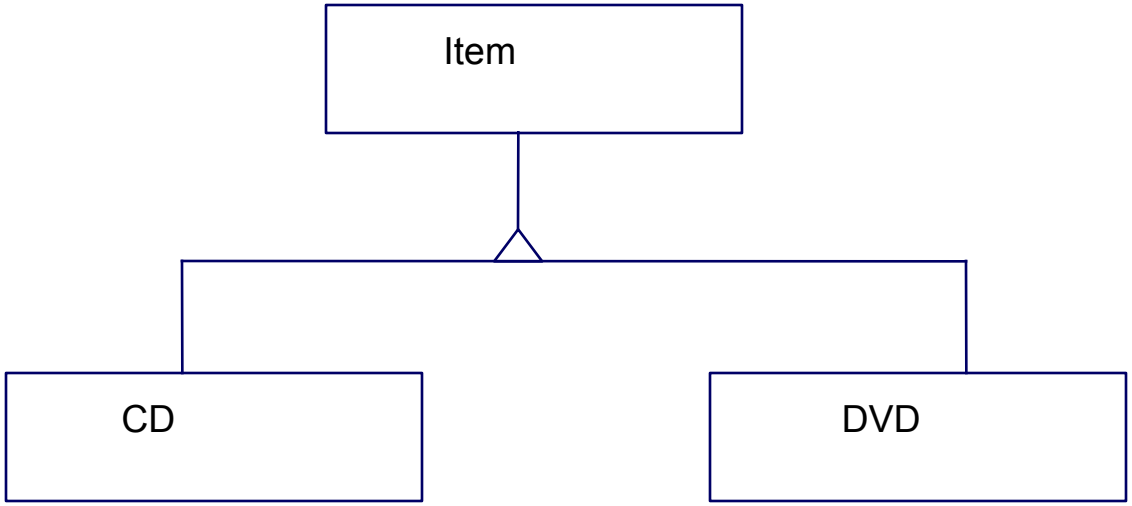
# Herança em Python (4)

```
class AplCurtoPrazo(ContaDePoupanca):
  def __init__(self):

        self.__fundoDisponivel = 0

  def retornaFundoDisponivel(self):
     return self.__fundoDisponivel
```

# Outro exemplo de herança



# construtores (1)

```
class Item:

    def __init__(self, theTitle, time):
        self.__title = theTitle
        self.__playingTime = time
        self.__gotIt = False
        self.__comment = ""

    # métodos omitidos
```

# construtores (2)

```
class CD (Item):

def __init__(self, theTitle, theArtist, tracks, time):

        super().__init__(theTitle, time)
        self.__artist = theArtist
        self.__numberOfTracks = tracks


    # métodos omitidos
```

# Formas de herança

- Especialização
- Generalização
- Limitação
- Especificação

# Especialização

- Uma classe Janela fornece as operações gerais de janela (mover, redimensionar, transformar em ícone e assim por diante).
- Uma subclasse especializada JanelaTexto herda as operações de Janela e adiciona facilidades que permitem a exibição e edição de texto.
- JanelaTexto satisfaz todas as propriedades esperadas de uma janela em geral.

# Generalização

- Oposto de especialização.
- Aqui, uma subclasse estende o comportamento da classe pai para criar um tipo mais geral de objeto.
- Uma base de classes existentes que não se deseja modificar ou não se pode modificar.
- Uma classe Janela tenha sido definida para exibir desenhos de fundo preto-e-branco.
- Pode-se criar uma subclasse JanelaColorida que permitisse uma cor de fundo diferente de preto-e-branco.

# Limitação

- O comportamento da subclasse é mais restritivo do que o comportamento da classe pai.
- Uma fila de duas extremidades (ou uma estrutura de dados deque). Elementos podem ser adicionados ou removidos de qualquer uma das extremidades
- O programador deseja escrever uma classe Pilha
- O programador pode definir uma classe Pilha como uma subclasse da classe Deque e modificar ou sobrescrever métodos existentes e indesejáveis de modo que eles produzam uma mensagem de erro se forem usados.

# Especificação

- Garante que classes mantenham uma certa interface comum, isto é, implementem os mesmos métodos.

- A classe filha meramente implementa o comportamento descrito (mas não implementado) pela classe pai.

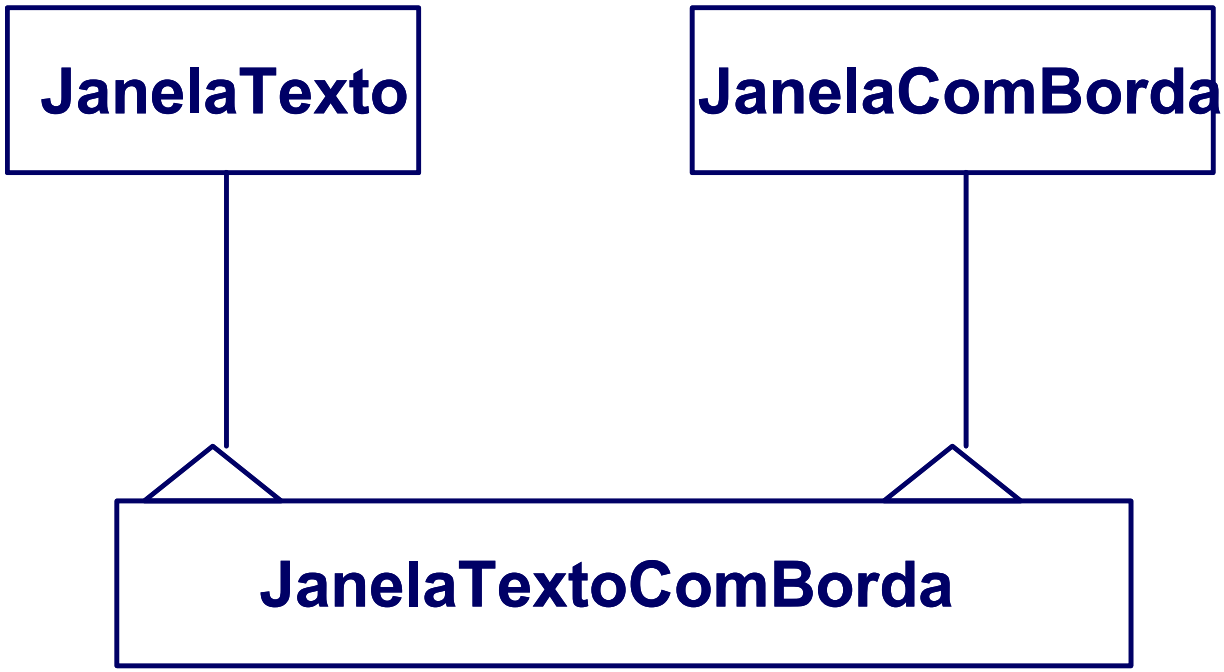
# Especificação

- Especificação é de fato um caso especial de especialização,
- exceto que as subclasses não são refinamentos de um tipo existente, mas sim realizações de uma especificação abstrata e incompleta.
- Em tais casos, a classe pai é algumas vezes conhecida como classe de especificação abstrata.

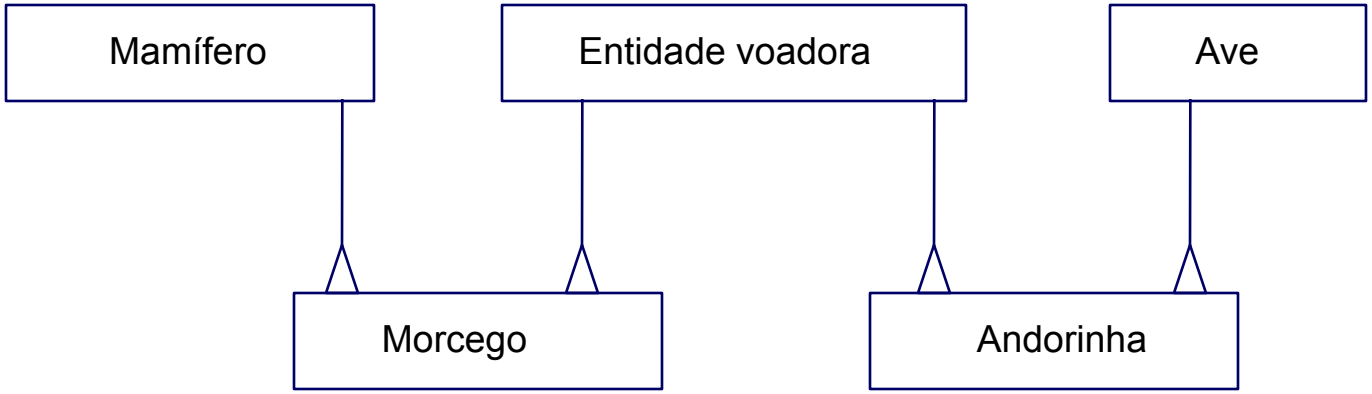
# Herança múltipla

- Em muitas situações, entretanto, é desejável que uma classe herde de mais de uma classe.
- Esse mecanismo é chamado de herança múltipla.
- Com ela, pode-se combinar diversas classes existentes para produzir uma nova classe.

# Exemplo de herança múltipla



# Outro exemplo



# Resolução de conflitos

- A subclasse agora pode herdar atributos e métodos de várias classes no mesmo nível da hierarquia.
- O que ocorrerá se esses atributos e métodos possuírem os mesmos nomes ?
- Teremos conflitos que precisam ser resolvidos.
- Observem que conflitos não ocorrem em uma hierarquia de classes com herança simples, pois a subclasse utiliza o atributo ou método imediato (sobreposição).

# Estratégias para resolução de conflitos

- **Linearização –** Essa estratégia especifica uma ordem linear e total das classes, e determina que a busca do atributo ou método seja efetuada da ordem estabelecida. Essa abordagem é implementada por Python.
- **Renomeação –** Essa estratégia requer a alteração dos nomes de atributos e operações que sejam conflitantes. Essa abordagem é implementada por Eiffel.
- **Qualificação –** Sempre que ocorrer ambigüidade deve-se usar o operador de escopo ::, como por exemplo, em C++. Esse operador permite a qualificação (isto é, identificação única) do atributo ou método conflitante. Essa abordagem é implementada por C++.

# Considerações sobre as estratégias para resolução de conflitos

- Linearização esconde o problema da resolução de conflitos do programador, mas introduz uma ordem obrigatória na herança de classes.

- A renomeação e a qualificação são estratégias que promovem uma maior flexibilidade para o programador decidir a aplicabilidade dos métodos e/ou atributos herdados.

# Vantagem e desvantagem da herança múltipla

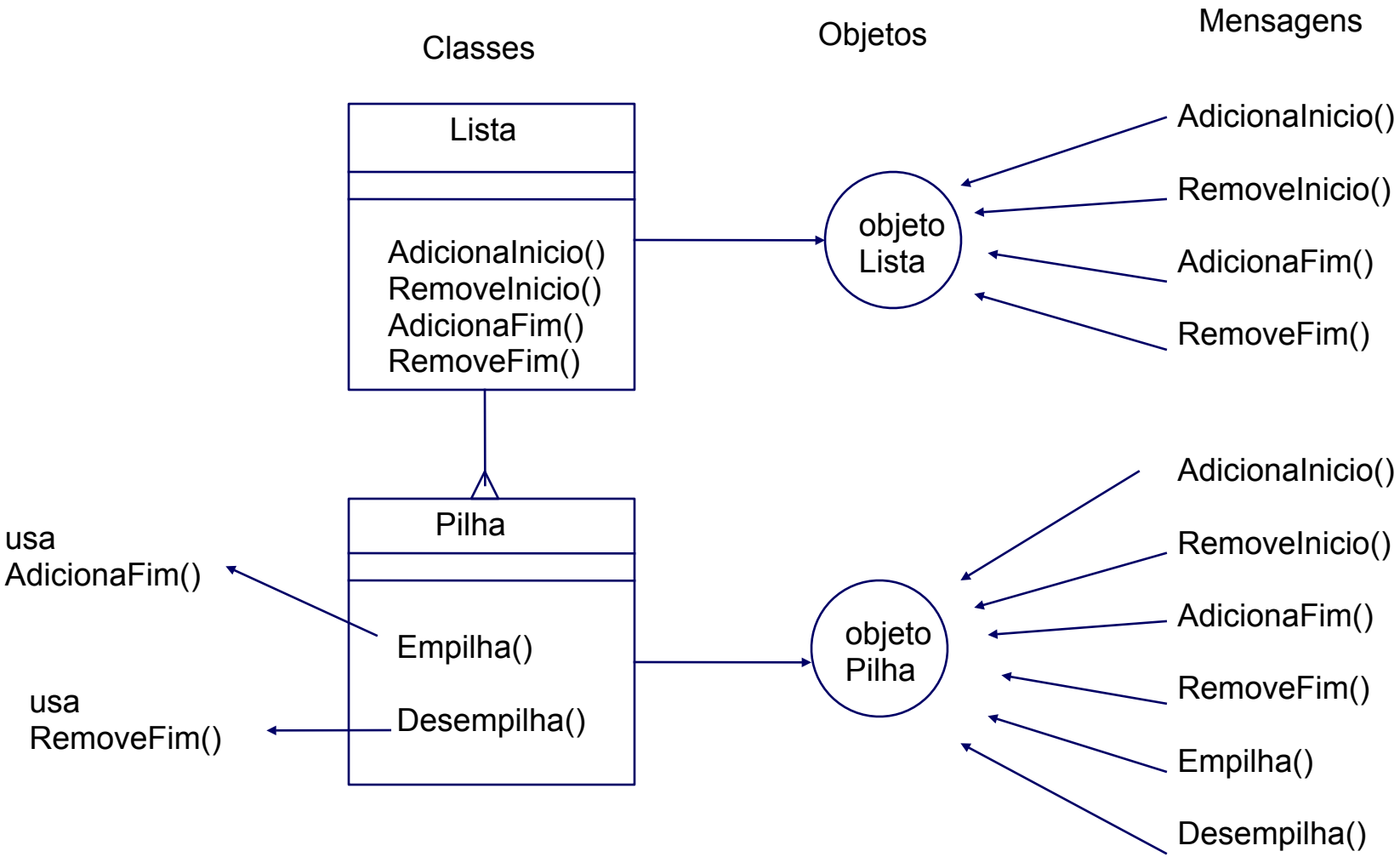
- Aumento da oportunidade de reutilização de comportamento.

- A desvantagem é uma perda da simplicidade conceitual e de implementação.

# Herança múltipla em Python

- Python suporta herança múltipla de classes.
- class ObjetoGrafico (Persistente, Gráfico):
- class Fox (Animal, Drawable):
- As classes podem herdar de duas ou mais classes

# Herança de implementação



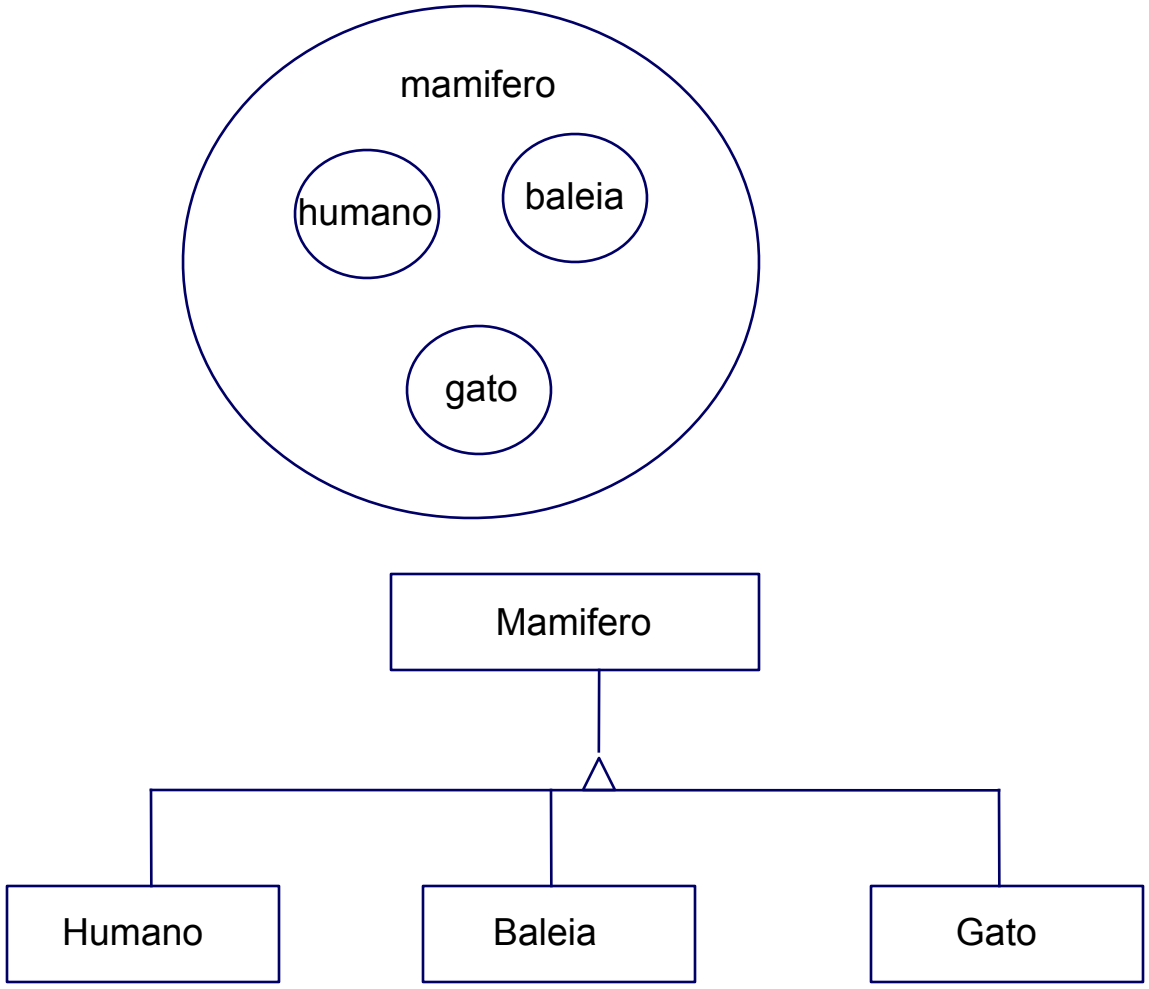
# Herança de comportamento (1)

- O mecanismo de herança é empregado para construção de hierarquias de tipos.
- Uma hierarquia de tipos é composta de subtipos e supertipos.
- Definição de Subtipo:
  - **Um tipo S é um subtipo de T se e somente se S proporciona pelo menos o comportamento de T.**
- Um objeto do tipo T pode ser substituído por um objeto do tipo S.

# Herança de comportamento (2)

- A noção de tipo/subtipo assemelha-se à noção de conjunto/subconjunto.
- Todos os objetos Baleia é um subconjunto dos objetos Mamífero.

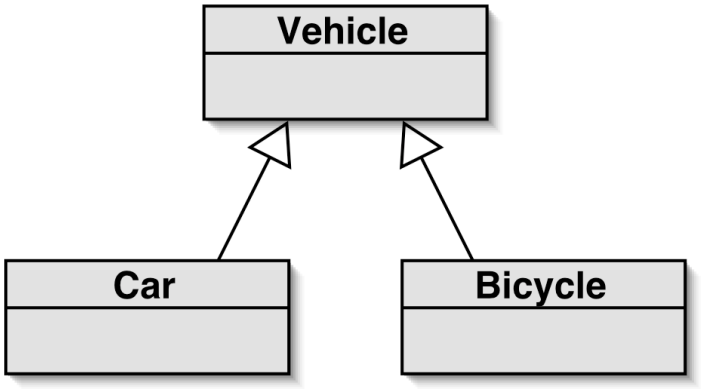
# Herança de comportamento (3)



# Subtipos (1)

- Classes definem tipos.
- Subclasses definem subtipos.
- Objetos das subclasses podem ser utilizados onde os objetos dos supertipos são requeridos.
  (Isso é chamado **substituição**.)

# Subtipos (2)




*Objetos da subclasse podem ser atribuídos a variáveis da superclasse*

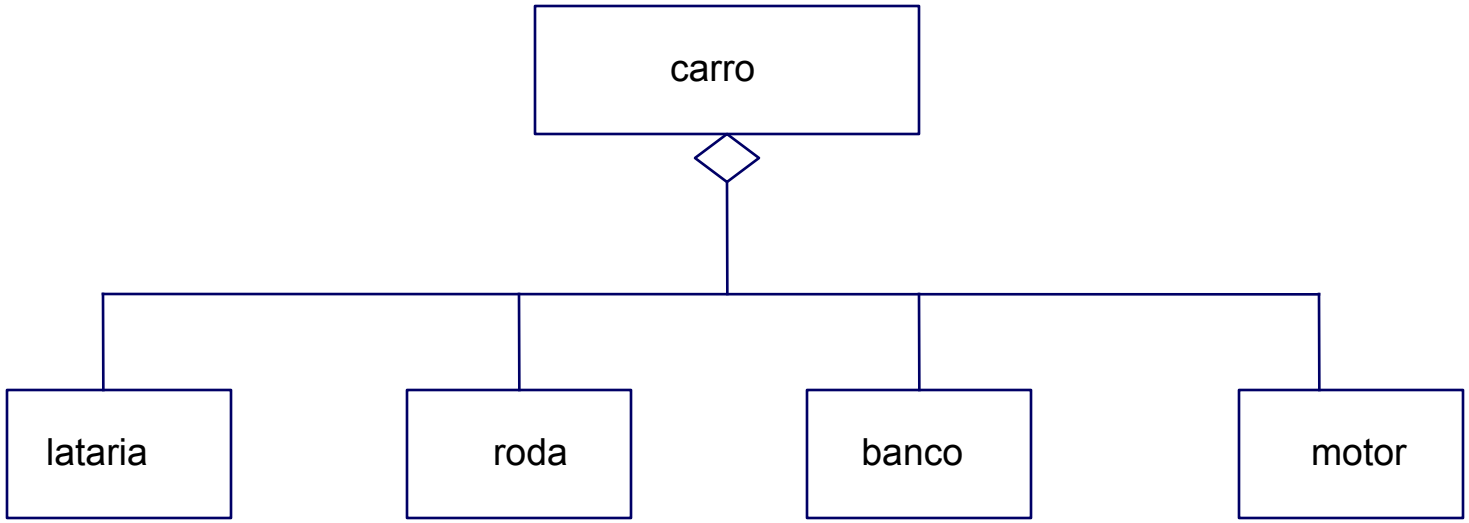
# Agregação X Generalização (1)

- O relacionamento de generalização é muito conveniente para a construção e manutenção de classes relacionadas com a aplicação.
- Quando a herança múltipla está envolvida a generalização pode ser confundida com a agregação.

# Agregação X Generalização (2)

- Herança múltipla permite que novos objetos sejam definidos pela **fusão** da estrutura e do comportamento de outros objetos diferentes.
- Agregação permite que novos objetos sejam definidos pela **composição** da estrutura e do comportamento de outros objetos diferentes.

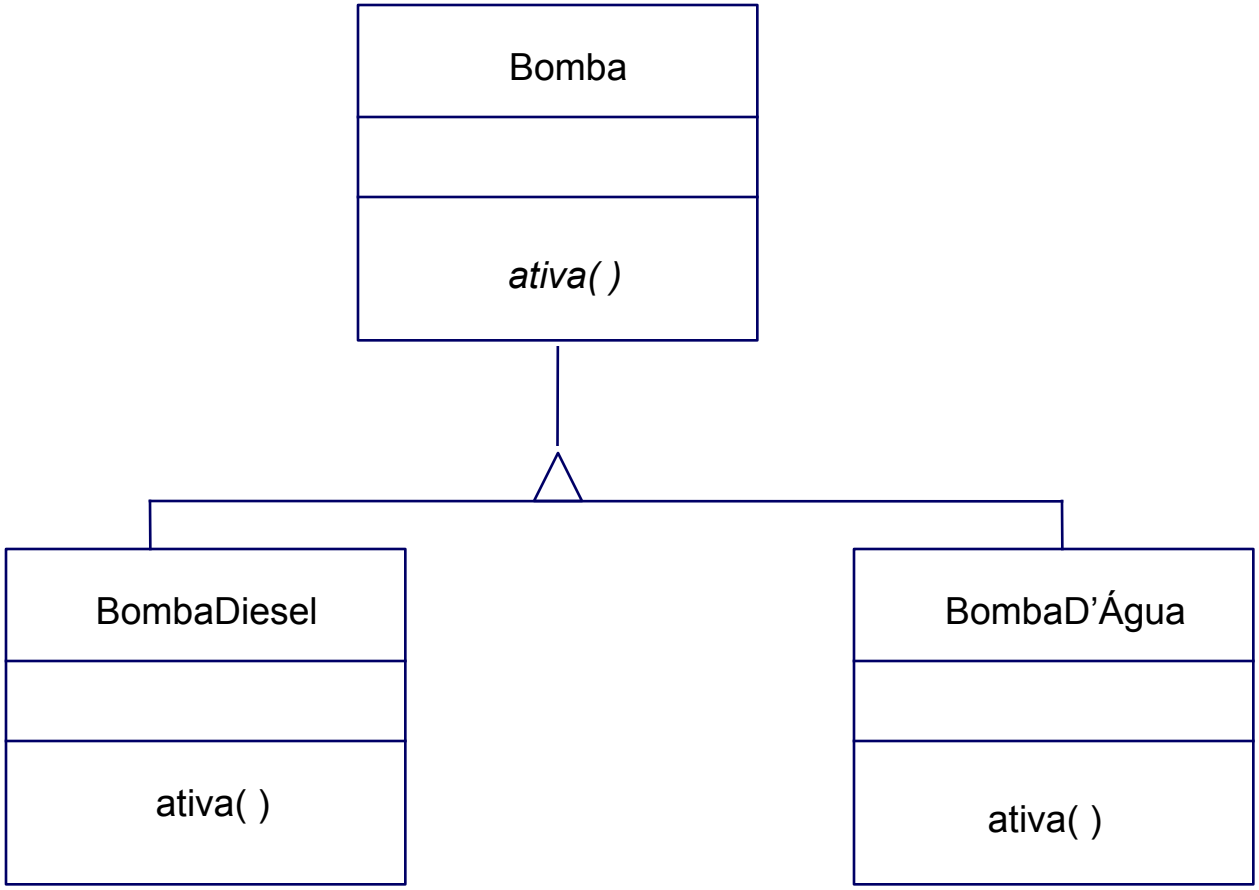
# Agregação X Generalização (3)



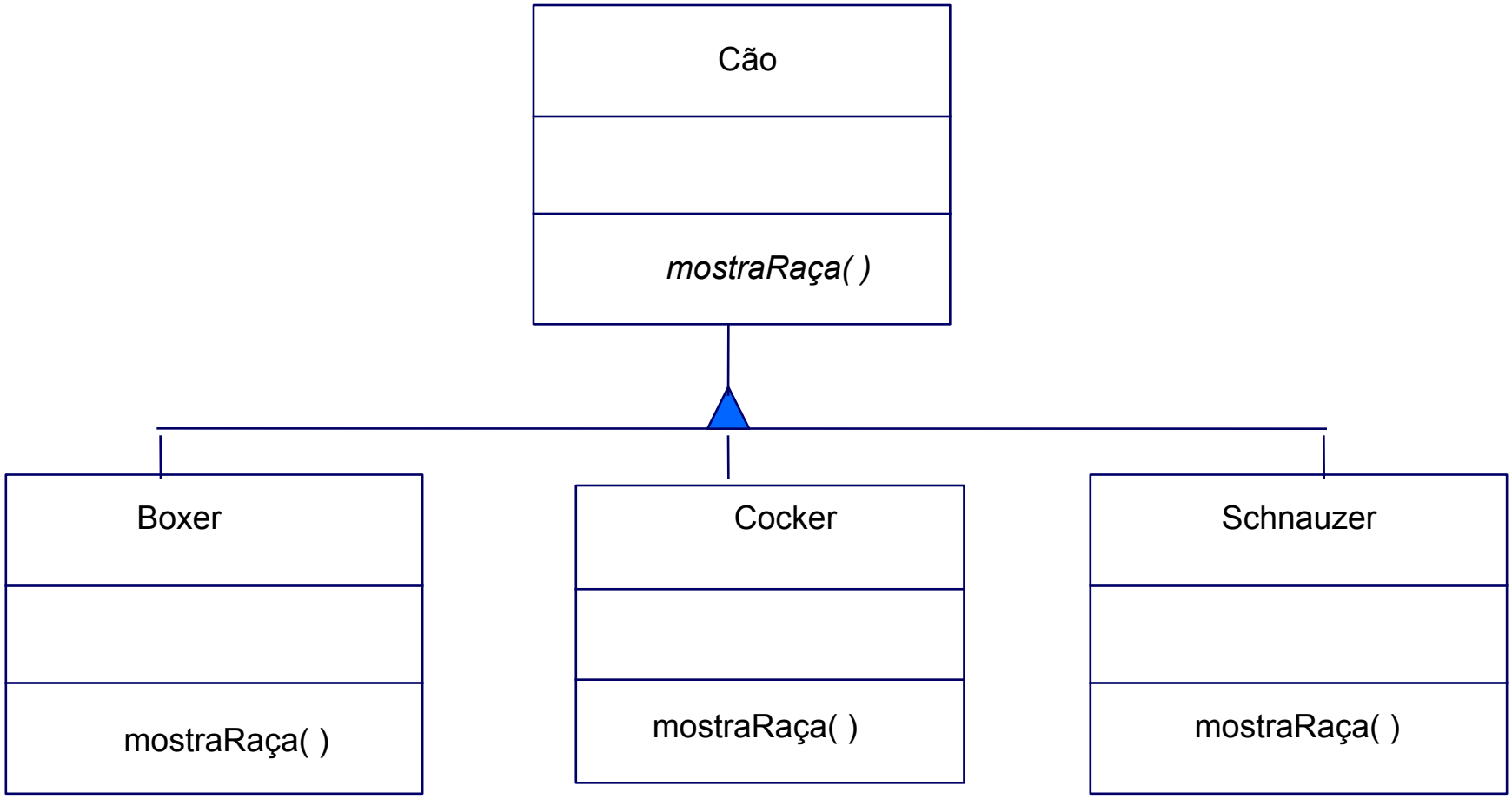
# Classes Abstratas X Classes Concretas (1)

- Uma classe abstrata é uma classe que não tem instâncias diretas.
- Uma classe concreta é uma classe que pode ser instanciada.
- Uma operação abstrata define a forma de uma operação para qual cada subclasse deve providenciar sua própria implementação.

# Classes Abstratas X Classes Concretas (2)



# Classes Abstratas X Classes Concretas (3)



# Classes Abstratas X Classes Concretas (4)

- Duas idéias principais envolvidas com a noção de classe abstrata:
  - **Presença de operações abstratas e**
  - **Habilidade de criar instâncias**
- A presença de uma operação abstrata implica na inabilidade de instanciar objetos.

# Classes Abstratas X Classes Concretas (5)

- Por padrão, o Python não fornece classes abstratas.
- Python vem com um módulo que fornece a base para a definição de classes Abstract Base (ABC)
- O nome do módulo é ABC.

# Classes Abstratas X Classes Concretas (6)

- abc funciona decorando métodos da classe base como abstratos e então registrando classes concretas como implementações da base abstrata.
- Um método se torna abstrato quando decorado com a palavra-chave @abstractmethod.

# Classes Abstratas X Classes Concretas (7)

```
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noofsides(self):
        pass
```