Programação Orientada a Objetos

Prof. Márcio Miguel Gomes

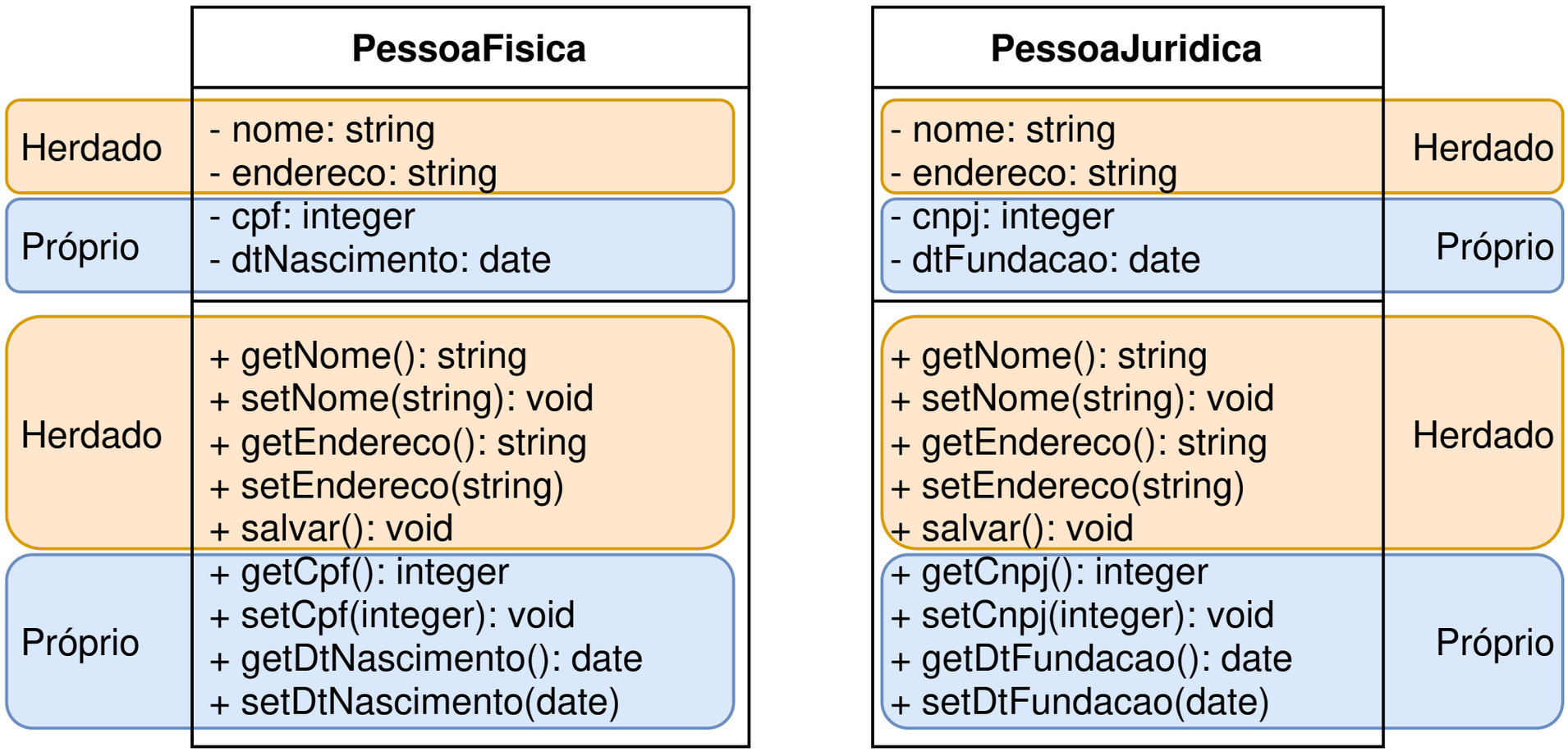
Herança - Problema

- As classes PessoaFisica e PessoaJuridica possuem **atributos** e **métodos** em **comum**, gerando **duplicidades** e dificultando **manutenção** de código

PessoaFisica		**PessoaJuridica**
- nome: string - endereco: string	Atributos em comum	- nome: string - endereco: string
- cpf: integer - dtNascimento: date	Atributos específicos	- cnpj: integer - dtFundacao: date
+ getNome(): string + setNome(string): void + getEndereco(): string + setEndereco(string) + salvar(): void	Métodos em comum	+ getNome(): string + setNome(string): void + getEndereco(): string + setEndereco(string) + salvar(): void
+ getCpf(): integer + setCpf(integer): void + getDtNascimento(): date + setDtNascimento(date)	Métodos específicos	+ getCnpj(): integer + setCnpj(integer): void + getDtFundacao(): date + setDtFundacao(date)

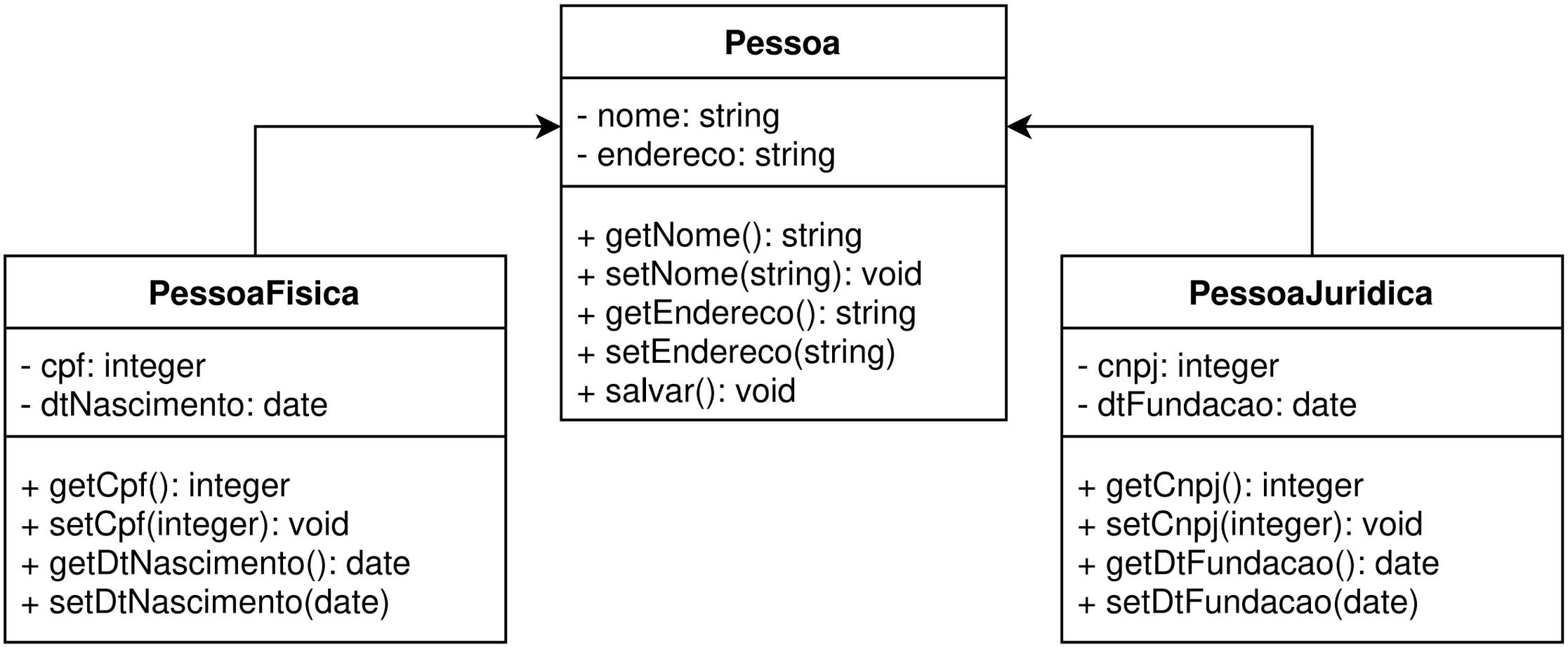
Herança - Solução

- O princípio da herança nos permite criar uma **superclasse** que define **atributos** e **métodos** em **comum** e usá-la como um "**modelo padrão**", a partir do qual fazemos **adaptações**



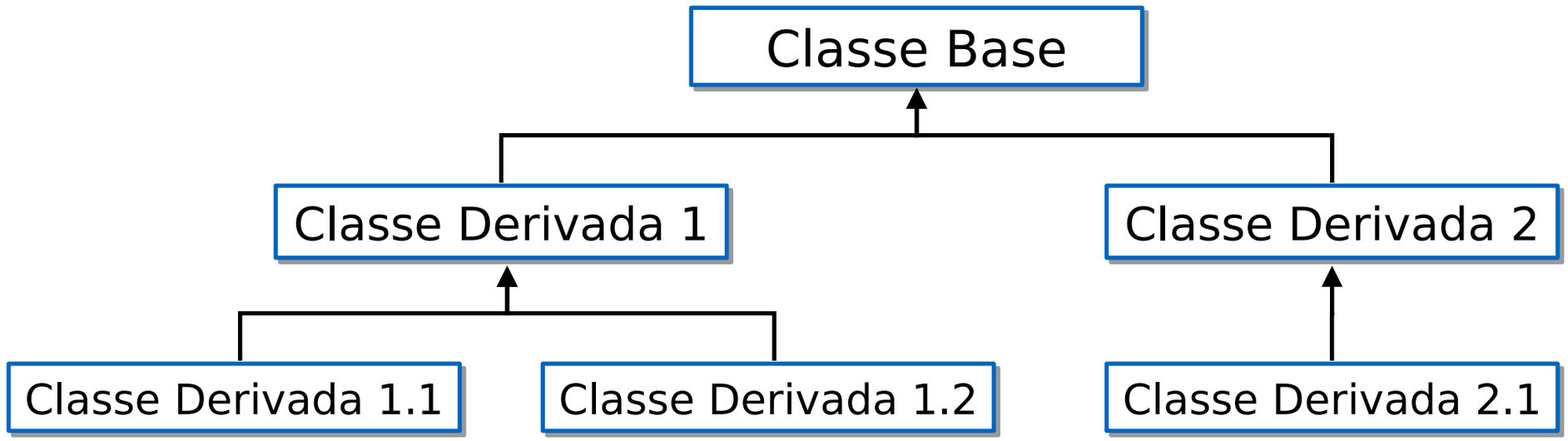
Herança - Solução

- Aqui, a **superclasse** Pessoa concentra os **atributos** e **métodos compartilhados** pelas outras **subclasses**, evitando **duplicidades** e facilitando a **manutenção**



Herança - Conceito

- Criação de uma **superclasse** básica “**pai**” com métodos e atributos **gerais**
- Criação de **subclasses** “**filhas**” derivadas a partir de uma superclasse “pai”, **herdando** suas características



Herança - Conceito

- São **herdados** da classe “pai” para as classes “filhas” os **atributos** e os **métodos públicos** e **protegidos**
- **Não** são herdados os **construtores**, **destrutores**, atributos e métodos **privados**
- Os elementos **herdados** são **acessados normalmente**, como se tivessem sido declarados e implementados na classe “filha”

Herança - Exemplo

```python
class Caixa:
  def set_altura(self, val):
    self._altura = val

  def set_largura(self, val):
    self._largura = val

  def get_altura(self):
    return self._altura

  def get_largura(self):
    return self._largura

class CaixaCor(Caixa):
  def set_cor(self, val):
    self._cor = val

  def get_cor(self):
    return self._cor
```

```python
if __name__ == '__main__':
  caixaCor = CaixaCor()
  caixaCor.set_cor('Verde')
  caixaCor.set_altura(5)
  caixaCor.set_largura(7)

  print('Cor:', caixaCor.get_cor())
  print('Altura:', caixaCor.get_altura())
  print('Largura:', caixaCor.get_largura())
```

Herança - Construtor

- Quando uma classe é **instanciada**, o seu **construtor** é chamado **automaticamente**
- Se a classe for **derivada** de alguma outra, o **construtor** da classe base **deve ser chamado na primeira linha de código**
- Se a classe base **também** for **derivada** de outra, o processo deve ser **repetido recursivamente** até que uma classe não derivada seja alcançada
- Isso é fundamental para se manter **consistência** do objeto recém criado

Herança - Destrutor

- Quando um objeto é **liberado**, o seu **destrutor** é chamado **automaticamente**
- Se a classe do objeto for **derivada** de alguma outra, o **destrutor** da classe base **deve ser chamado na última linha de código**
- Se a classe base **também** for **derivada** de outra, o processo deve ser **repetido recursivamente** até que uma classe não derivada seja alcançada
- Isso é fundamental para **liberar** todos os **recursos** alocados para o objeto

Construtor e Destrutor

```python
class Primeira:
  def __init__(self, id):
    self._id = id
    print('Construtor Primeira - id:', self._id)

  def __del__(self):
    print('Destrutor Primeira - id:', self._id)

  def get_id(self):
    return self._id

class Segunda(Primeira):
  def __init__(self, id, nr):
    super().__init__(id)
    self._nr = nr
    print('Construtor Segunda – id:',
      self.get_id(), '- nr:', self._nr)

  def __del__(self):
    print('Destrutor Segunda – id:',
      self.get_id(), '- nr:', self._nr)
    super().__del__()
```

```python
if __name__ == '__main__':
  p = Primeira(1)
  s = Segunda(2, 3)
  del p
  del s
```

Polimorfismo - Conceito

- Polimorfismo = *poli* + *morphos*
- Polimorfismo descreve a **capacidade** de um código de programação **comportar-se de diversas formas**, dependendo do **contexto**
- É um dos recursos mais poderosos de linguagens orientadas a objetos
- Permite trabalhar em um **nível alto** de **abstração**
- Facilita a incorporação de novos recursos em um sistema existente

Polimorfismo
Redefinição de Métodos

- E se definíssemos dois métodos com mesmo nome nas classes “pai” e “filha” ?
- **Não existe sobrecarga em uma hierarquia** (depende da linguagem)
- A definição de **métodos** com **mesmo nome** em classes base e derivada **não** os deixa **disponíveis**, mesmo com assinaturas distintas
- A última definição **esconde** a anterior (sobrescrita)
- Continuam acessíveis, mas não de forma direta

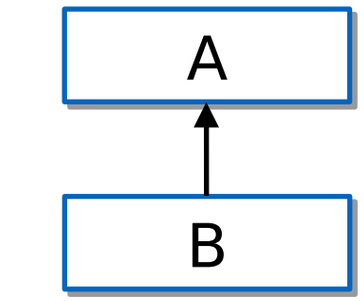
Polimorfismo - Exemplo

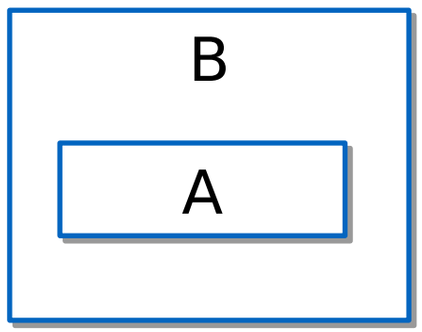
```python
class A:
  def f1(self):
    print('A.f1()')

  def f2(self):
    print('A.f2()')


class B(A):
  # Sobrescrita do método f1 de A
  def f1(self, val=None):
    if val:
      print('B.f(', val, ')')
    else:
      # Executa f1 da superclasse
      super().f1()
```

```python
if __name__ == '__main__':
  b = B()
  b.f1(10)
  b.f1()
  b.f2()
```

Polimorfismo - Classes

- Sendo B derivado de A, todos os membros disponíveis em A também estarão em B
- B é um superconjunto de A: todas as operações que podem ser feitas com objetos de A também o podem através de objetos de B
- Um objeto da classe B também é um objeto da classe A: isso significa a possibilidade de se converter um objeto de B para A





Polimorfismo - Classes

```
class Pessoa:
  def __init__(self, nome, endereco):
    self._nome = nome
    self._endereco = endereco

  def exibe_dados(self):
    print()
    print('Nome:', self._nome)
    print('Endereco:', self._endereco)

class PessoaFisica(Pessoa):
  def __init__(self, nome, endereco, cpf, dt_nascimento):
    super().__init__(nome, endereco)
    self._cpf = cpf
    self._dt_nascimento = dt_nascimento

  def exibe_dados(self):
    super().exibe_dados()
    print('CPF:', self._cpf)
    print('Nascimento:', self._dt_nascimento)
```

Polimorfismo - Classes

```
class PessoaJuridica(Pessoa):
  def __init__(self, nome, endereco, cnpj, dt_fundacao):
    super().__init__(nome, endereco)
    self._cnpj = cnpj
    self._dt_fundacao = dt_fundacao

  def exibe_dados(self):
    super().exibe_dados()
    print('CNPJ:', self._cnpj)
    print('Fundação:', self._dt_fundacao)

if __name__ == '__main__':
  pessoas = []
  pessoas.append(
    Pessoa('Nome Geral', 'Endereço Geral'))
  pessoas.append(
    PessoaFisica('Fulano de Tal', 'Rua A, 123', '458.245.123-47', '03/08/2000'));
  pessoas.append(
    PessoaJuridica('Acme Corp.', 'Rua B, 456', '93.479.720/0001-73', '15/10/2012'));

  for pessoa in pessoas:
    if isinstance(pessoa, Pessoa):
      pessoa.exibe_dados()
```

Representação

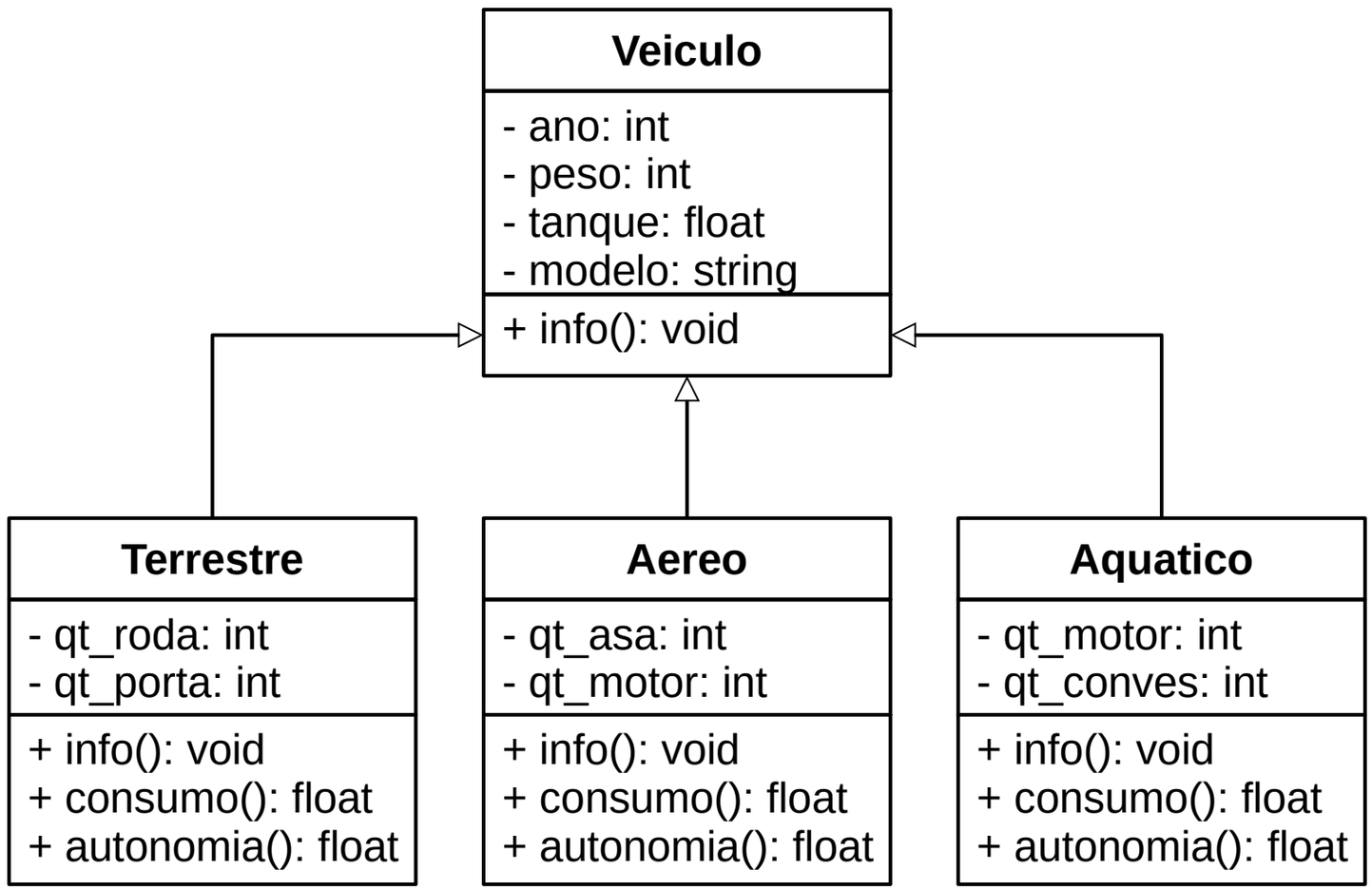
- O Python possui 2 métodos para **representar** objetos, o que **facilita** e **padroniza** o desenvolvimento do código
- **__str__()**
  - Método que converte o objeto em um texto para apresentação ao **usuário**
- **__repr__()**
  - Método que converte o objeto em um texto para uso por **desenvolvedores**

Representação

```python
class Pessoa:
  def __init__(self, nome, endereco):
    self._nome = nome
    self._endereco = endereco

  def __repr__(self):
    return f'{self._nome};{self._endereco}'

  def __str__(self):
    return f'Nome: {self._nome}\nEndereco: {self._endereco}'


if __name__ == '__main__':
  pessoa = Pessoa('Nome Geral', 'Endereço Geral')
  print(pessoa)
  print(repr(pessoa))
```

Atividade

- Criar a superclasse “Veiculo” e as subclasses “Terrestre”, “Aereo” e “Aquatico” conforme modelo



Atividade

- O construtor das subclasses deve chamar o construtor da superclasse passando os parâmetros necessários
- O consumo de combustível da subclasse “Terrestre” é dado pela equação: 1/((peso * 0,00005) + (qt_roda * 0,005)), em km/l
- O consumo de combustível da subclasse “Aereo” é dado pela equação: 1/((peso * 0,00003) + (qt_motor * 0,01)), em km/l
- O consumo de combustível da subclasse “Aquatico” é dado pela equação: 1/((peso * 0,00002) + (qt_motor * 0,02)), em km/l
- A autonomia será proporcional ao volume do tanque e ao consumo calculado
- O método “info()” das subclasses deve mostrar na tela as características da superclasse, e em seguida, as características da subclasse
- Criar objetos das subclasses e chamar o método “info()” para mostrar os dados na tela