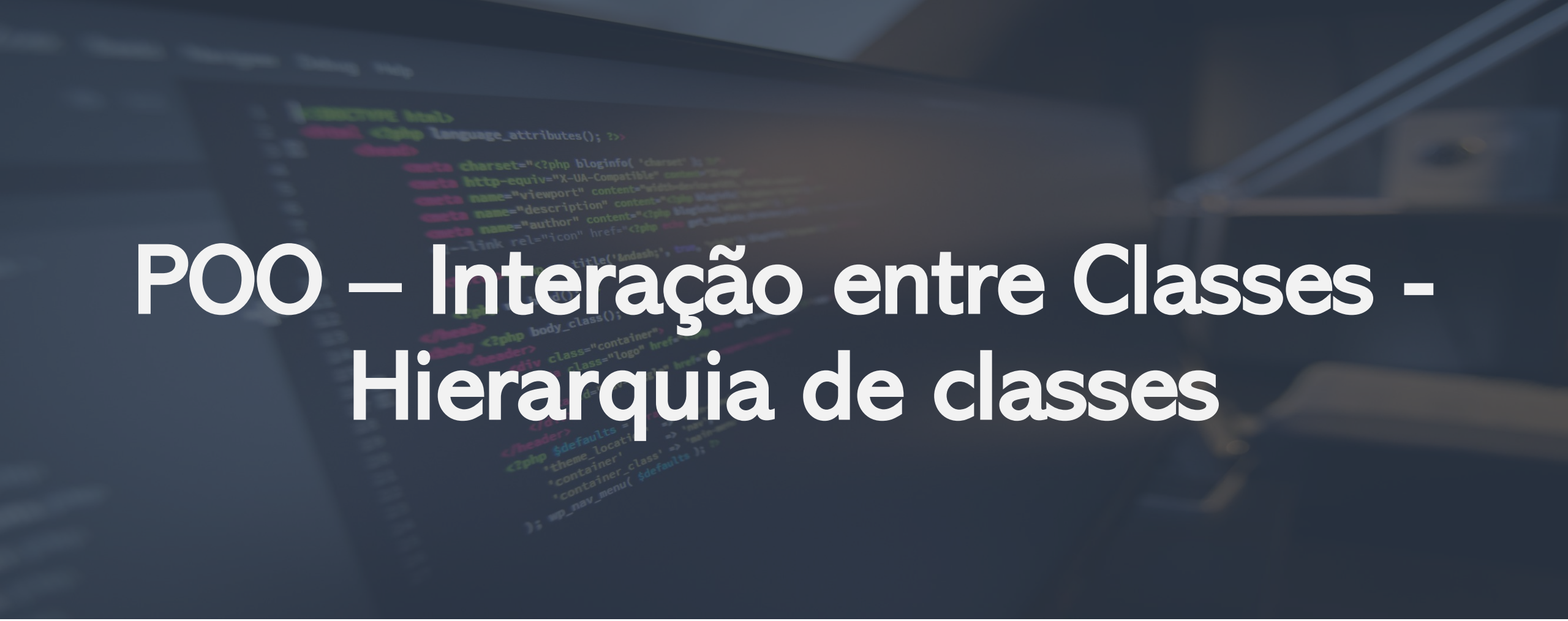


POO – Interação entre Classes - Hierarquia de classes

Prof. Antonio David Viniski

PUCPR

Conceitos

1. Associação
2. Agregação
3. Composição
4. Superclasse
5. Subclasse
6. Generalização
7. Especialização
8. Herança
9. Encadeamento de construtores
10. Sobrecarga de método

Associação entre Classes

Tipos de associação

- Agregação
  - O objeto da classe associada existe de forma independente
- Composição
  - A existência de uma classe depende de outra classe

Tipos de associação - Agregação

- Supondo que precisamos de mais informações do professor do Curso, como nome, formação, e-mail.
  - Adicionar essas informações na classe Curso não é viável, pois toda alteração dos dados de professor exigiria uma alteração em todos os cursos que esse professor ministra
  - Com isso, torna-se necessária a criação da classe Professor
- Vamos implementar a agregação de Professor em curso.

Tipos de associação – Agregação - solução

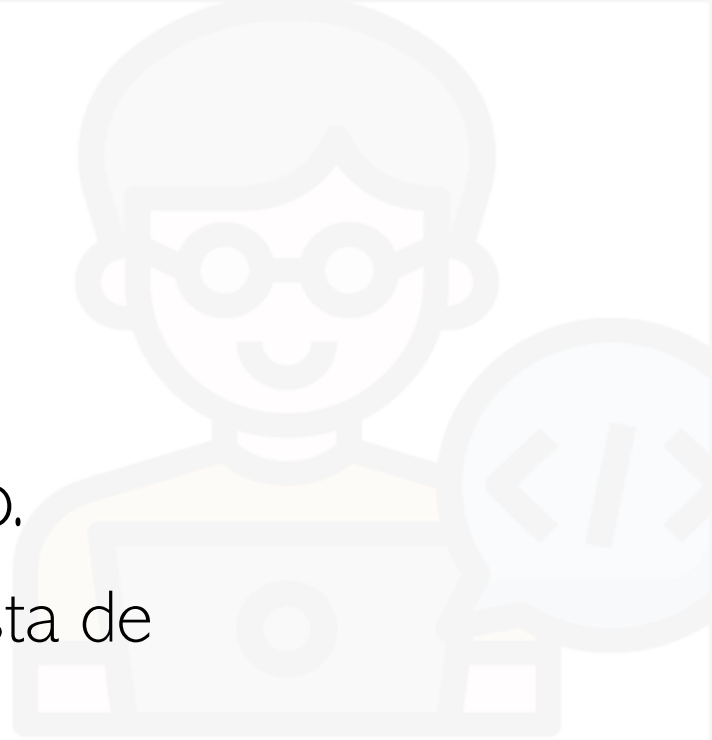
```
class Professor:
    def __init__(self, nome: str, formacao: str, email: str):
        self.__nome: str = nome
        self.__formacao: str = formacao
        self.__email: str = email
```

```
class Curso:
    def __init__(self, nome: str, data: str, professor: Professor, horas: int, valor: float):
        self.__nome: str = nome
        self.__data: str = data
        self.__professor: Professor = professor
        self.__horas: int = horas
        self.__valor: float = valor

antonio = Professor("Antônio David Viniski", "Doutor em Informática", "antonio.david@pucpr.br")

curso = Curso("Programação Orientada a Objetos", "07/10/2023", antonio, 20, 1000.00)
curso2 = Curso("Banco de Dados", "31/01/2024", antonio, 24, 1200.00)
```

Exercício 1

- Supondo que precisamos de mais informações do cliente titular da Conta no Banco, como nome, cpf, e-mail.
  - Criar a classe Cliente com seus atributos privados.
  - Realizar a agregação entre Cliente e Conta
- **Funcionalidade adicional:** Criar a estrutura para adicionar um **Gerente** para a **Conta** do **Cliente**.

Tipos de associação - Composição



- Agora vamos pensar na ementa de um curso.
  - Todo curso possui uma ementa, ou seja, uma lista de conteúdos que serão abordados
  - Toda ementa é criada exclusivamente para um curso, ou seja, ela vai existir somente associada a um curso
  - A ementa tem uma data de criação e também a data de modificação

Tipos de associação – Composição - Ementa

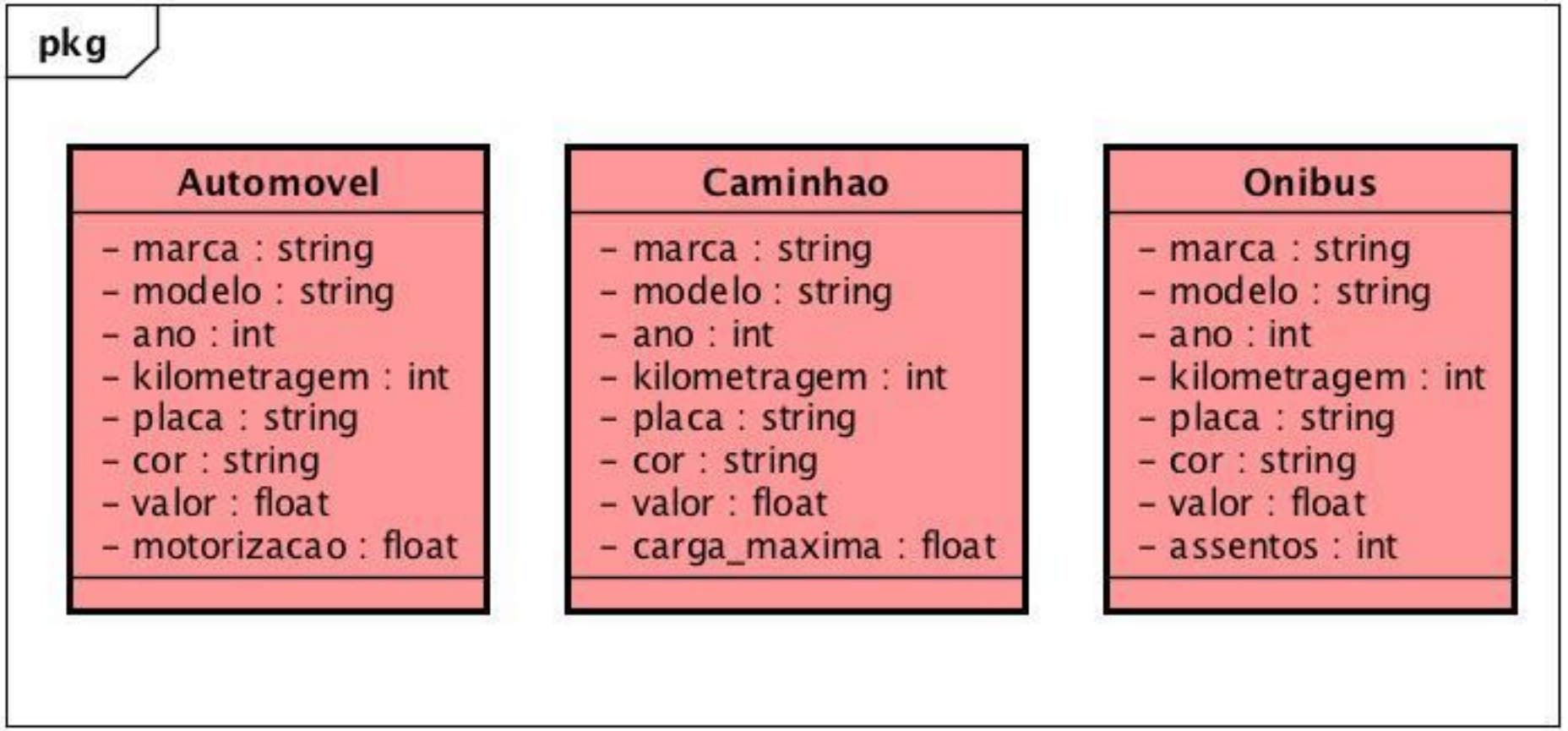
- Vamos criar a ementa do curso de Programação Orientada a Objetos
  - Precisamos de um método para adicionar itens a ementa
  - Precisamos também remover itens da ementa
  - Precisamos também atualizar a data de modificação toda vez que uma alteração é feita na ementa.

Exercício 2

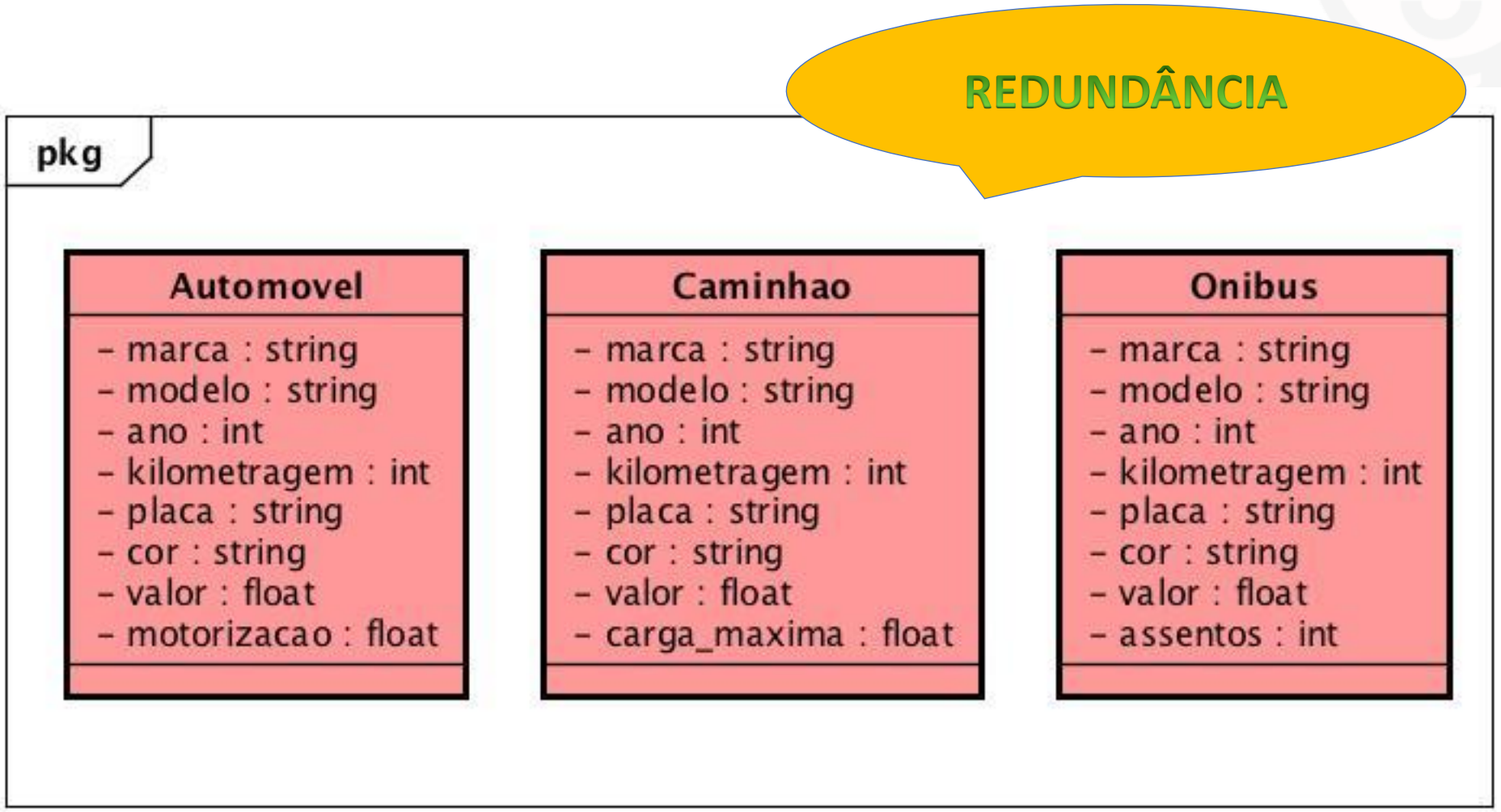
- Considere que precisa ser armazenado o histórico de cada uma das transações efetuadas na Conta do Cliente.
- Esse histórico pertence exclusivamente a conta, ou seja, não existe sem que a conta exista.
  - Criar a classe Histórico, que possui uma lista de transações, contendo o horário, o tipo de transação e valor
  - Realizar a Composição entre Conta e Histórico


Herança

Modelo de Classes - UML



Modelo de Classes - UML



Implementação - Python

```
class Automovel:
    def __init__(self, marca, modelo, ano, kilometragem, placa, cor, valor, motorizacao):
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__kilometragem = kilometragem
        self.__placa = placa
        self.__cor = cor
        self.__valor = valor
        self.__motorizacao = motorizacao

# métodos
```

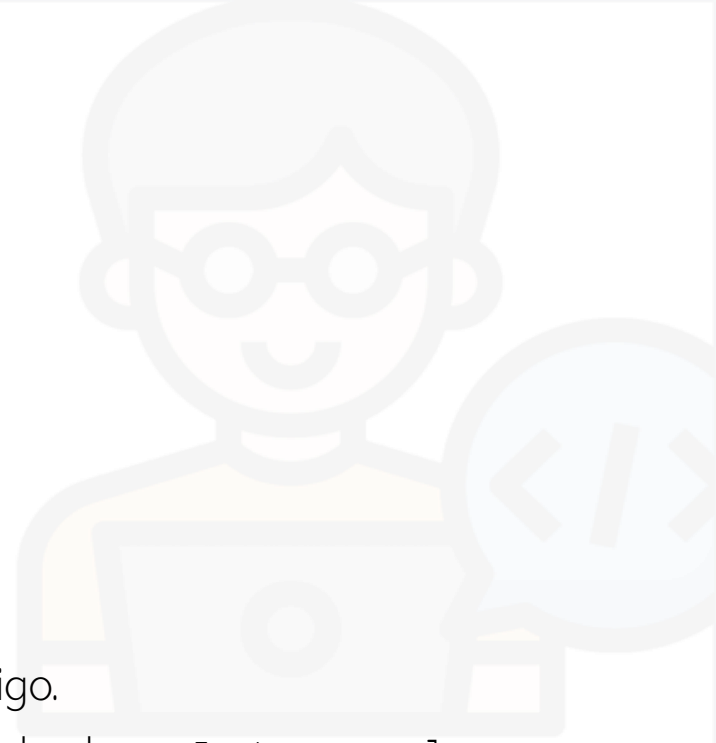
Implementação - Python

```
class Caminhao:
    def __init__(self, marca: str, modelo: str, ano: int, kilometragem: int,
                 placa: str, cor: str, valor: float, carga_maxima: float):
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__kilometragem = kilometragem
        self.__placa = placa
        self.__cor = cor
        self.__valor = valor
        self.__carga_maxima = carga_maxima

# métodos
```

Implementação - Python

```
class Onibus:
    def __init__(self, marca: str, modelo: str, ano: int, kilometragem: int,
                 placa: str, cor: str, valor: float, assentos: int):
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__kilometragem = kilometragem
        self.__placa = placa
        self.__cor = cor
        self.__valor = valor
        self.__assentos = assentos

    # métodos
```

Efeitos nocivos da redundância



1. Rápido crescimento do volume de código

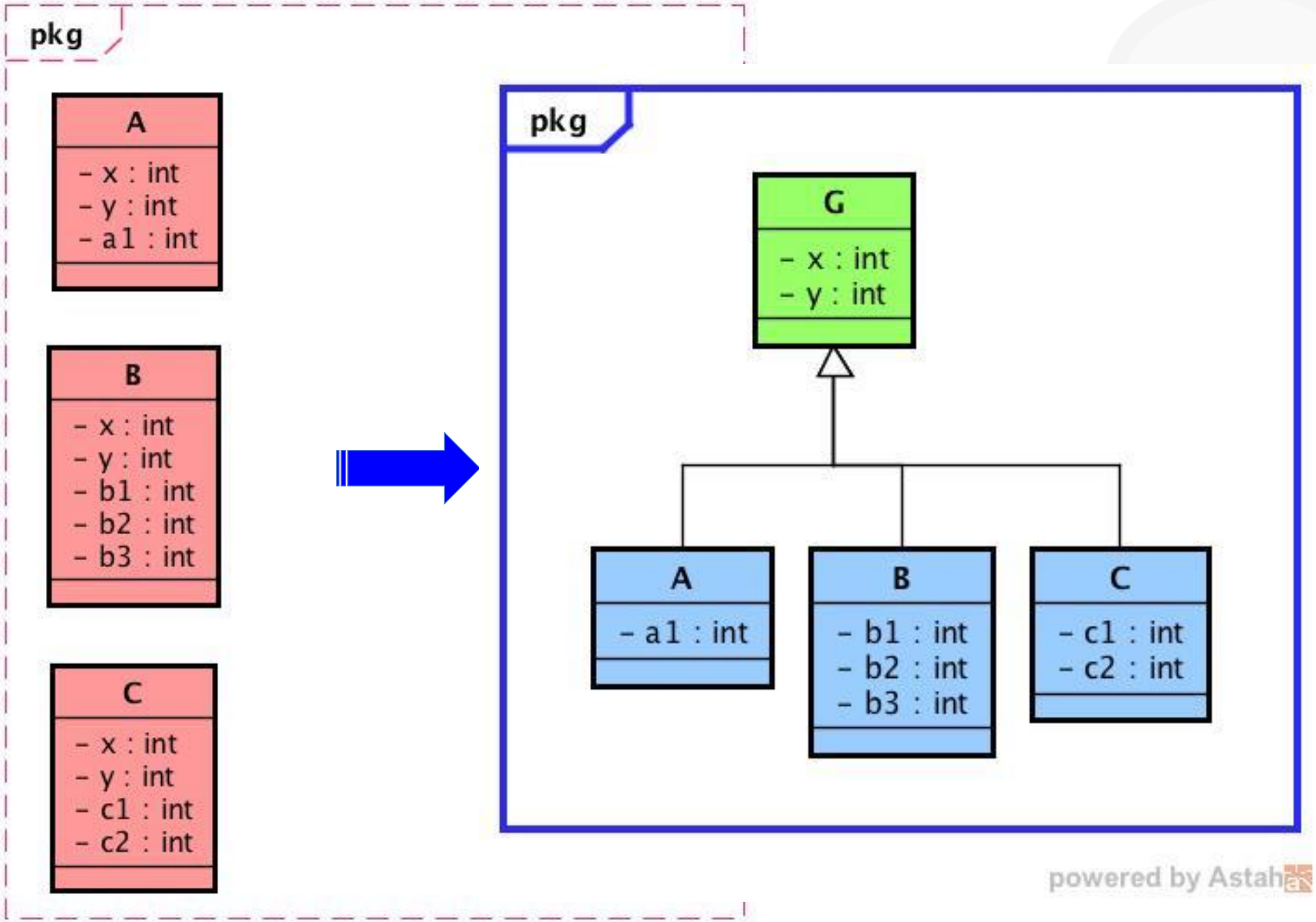
- A criação de uma nova classe implica em replicar parte significativa do código.
- Por exemplo, a classe `Motocicleta` replicaria quase todos os atributos da classe `Automovel`.

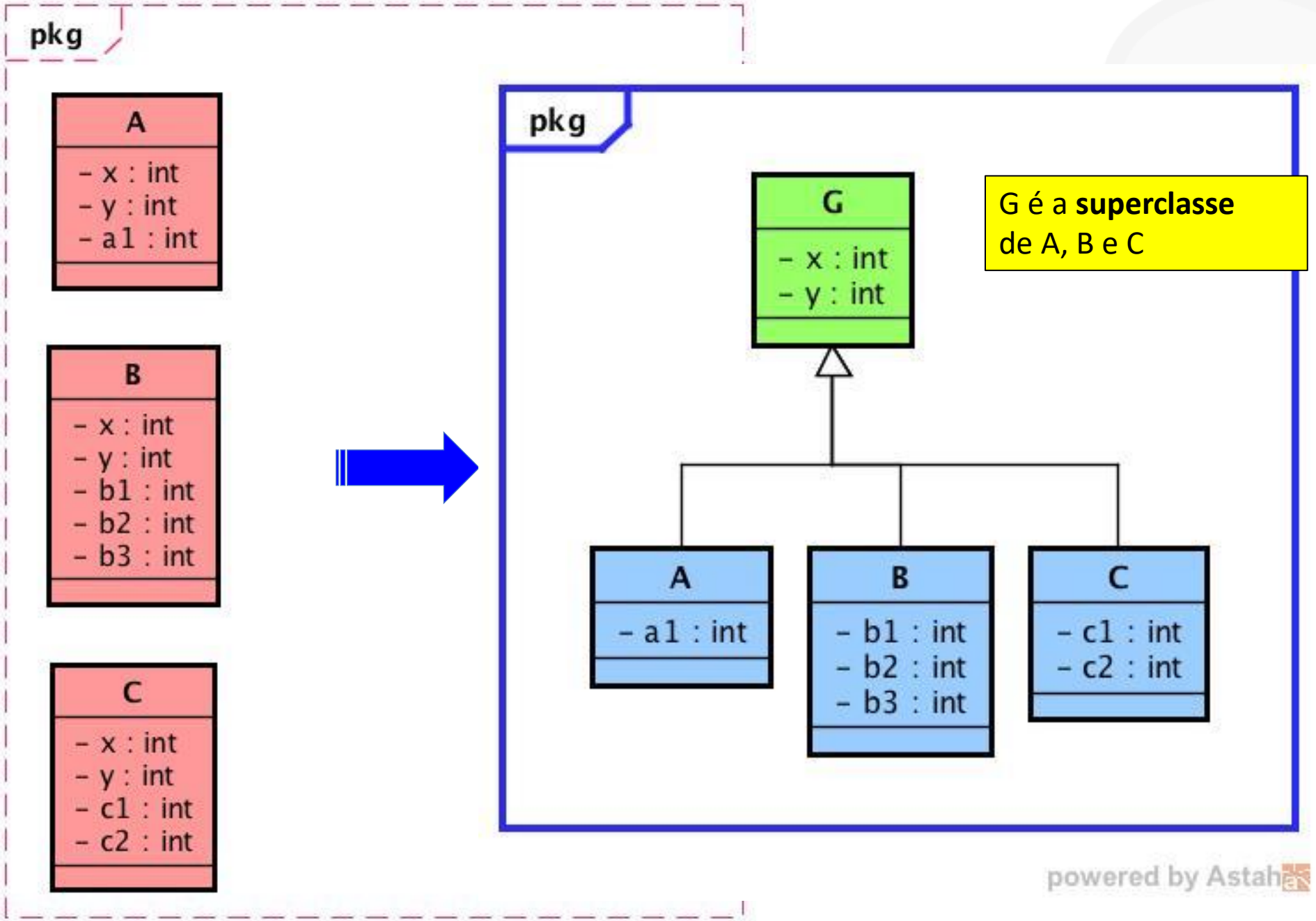
2. Manutenção complexa

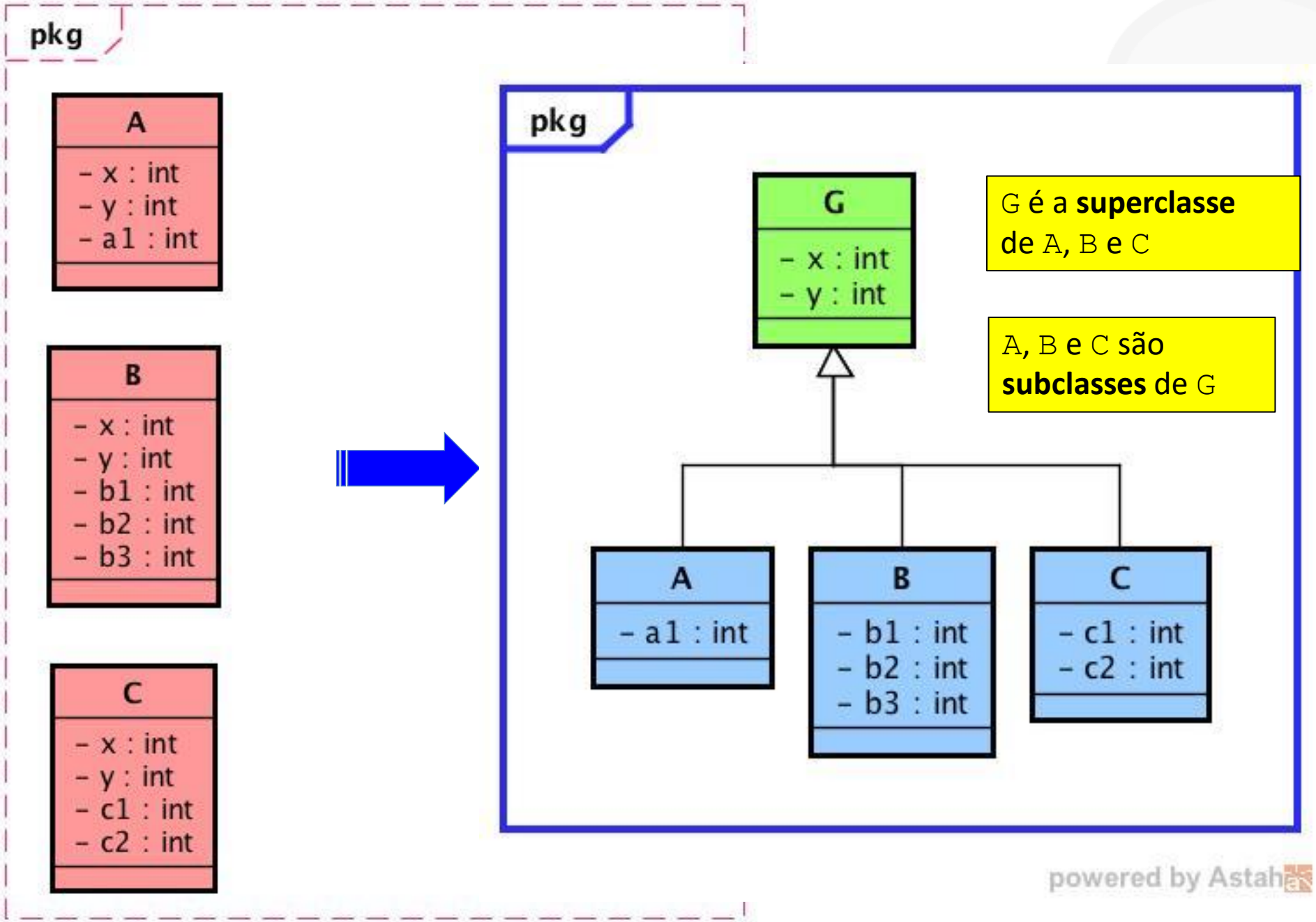
- Qualquer modificação de um membro comum entre as classes implica em atualizar o código de cada classe.
- Por exemplo, a alteração do nome do atributo `valor` para `preco` exigiria a atualização do código das três classes: `Automovel`, `Caminhao` e `Onibus`.

Eliminação da redundância

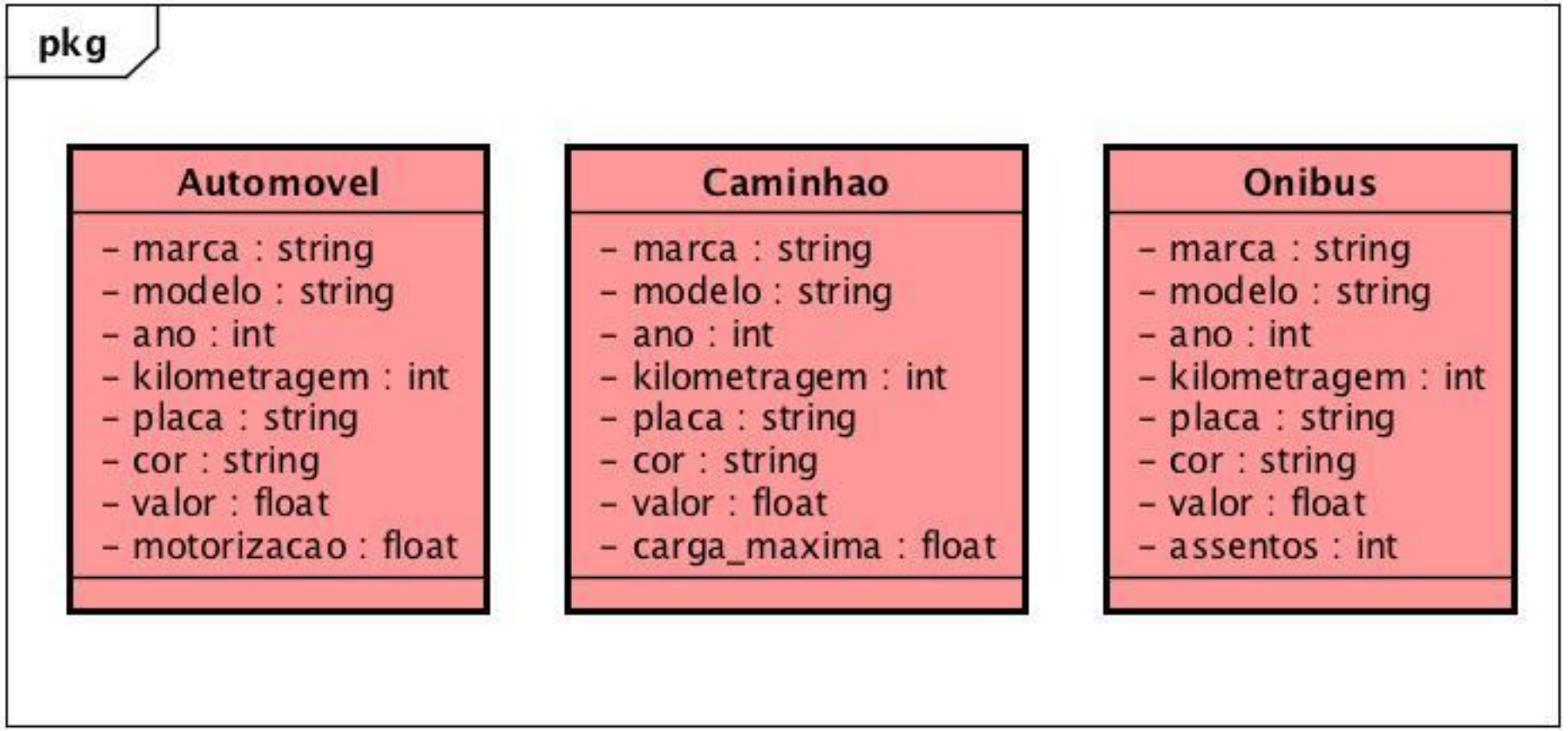
1. Criar uma nova classe contendo os membros comuns das classes existentes.
2. Remover os membros comuns das classes existentes.
3. Estabelecer um relacionamento de hierarquia entre cada classe existente e a nova classe criada:
   - Cada classe existente torna-se uma **subclasse** da nova classe
   - A nova classe torna-se a **superclasse** de cada classe existente

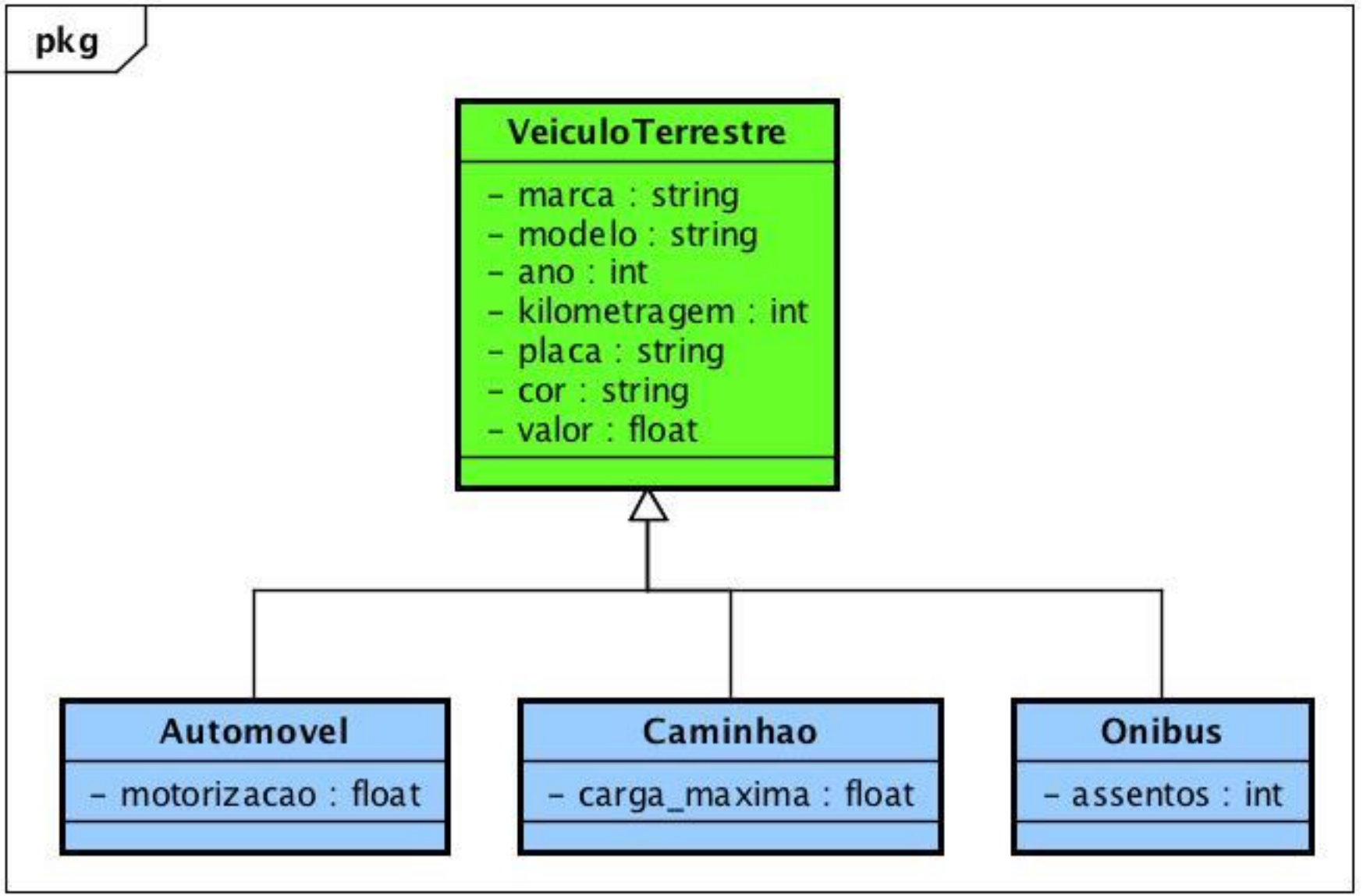

pkg
A
- x : int
- y : int
- a1 : int
B
- x : int
- y : int
- b1 : int
- b2 : int
- b3 : int
C
- x : int
- y : int
- c1 : int
- c2 : int
pkg
G
- x : int
- y : int
A
- a1 : int
B
- b1 : int
- b2 : int
- b3 : int
C
- c1 : int
- c2 : int
powered by Astah


pkg
A
- x : int
- y : int
- a1 : int
B
- x : int
- y : int
- b1 : int
- b2 : int
- b3 : int
C
- x : int
- y : int
- c1 : int
- c2 : int
pkg
G
- x : int
- y : int
G é a **superclasse** de A, B e C
A
- a1 : int
B
- b1 : int
- b2 : int
- b3 : int
C
- c1 : int
- c2 : int
powered by Astah


pkg
A
- x : int
- y : int
- a1 : int
B
- x : int
- y : int
- b1 : int
- b2 : int
- b3 : int
C
- x : int
- y : int
- c1 : int
- c2 : int
pkg
G
- x : int
- y : int
G é a superclasse de A, B e C
A, B e C são subclasses de G
A
- a1 : int
B
- b1 : int
- b2 : int
- b3 : int
C
- c1 : int
- c2 : int
powered by Astah

Eliminando redundância ...




pkg
VeiculoTerrestre
- marca : string
- modelo : string
- ano : int
- kilometragem : int
- placa : string
- cor : string
- valor : float
Automovel
- motorizacao : float
Caminhao
- carga_maxima : float
Onibus
- assentos : int
powered by Astah

Implementação - Python

```
class VeiculoTerrestre:
    def __init__(self, marca, modelo, ano, kilometragem, placa, cor, valor):
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__kilometragem = kilometragem
        self.__placa = placa
        self.__cor = cor
        self.__valor = valor

    # métodos
```

Implementação - Python

```
class Automovel(VeiculoTerrestre):
    def __init__(self, marca, modelo, ano, kilometragem, placa, cor, valor, motorizacao):
        super().__init__(marca, modelo, ano, kilometragem, placa, cor, valor)
        self.__motorizacao = motorizacao

    # métodos
```

A classe VeiculoTerrestre entre parênteses após o nome da classe Automovel é usada para estabelecer o relacionamento de hierarquia entre duas classes.

class *Subclasse(Superclasse):*

O termo **super** é responsável por acessar diretamente os métodos da classe genérica (superclasse), como por exemplo, o método super().__init__ inicializa os atributos da superclasse

Implementação - Python

```
class Caminhao(VeiculoTerrestre):
    def __init__(self, marca, modelo, ano, kilometragem, placa, cor, valor, carga_maxima):
        super().__init__(marca, modelo, ano, kilometragem, placa, cor, valor)
        self.__carga_maxima = carga_maxima

    # métodos
```

Implementação - Python

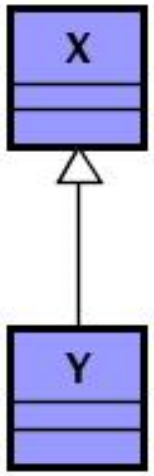
```
class Onibus(VeiculoTerrestre):
    def __init__(self, marca, modelo, ano, kilometragem, placa, cor, valor, assentos):
        super().__init__(marca, modelo, ano, kilometragem, placa, cor, valor)
        self.__assentos = assentos

    # métodos
```

Terminologia e efeitos

Terminologia e efeitos

class Y(X)

- X é **superclasse** de Y
- Y é **subclasse** de X

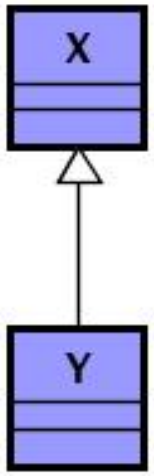


Terminologia e efeitos

class Y (X)

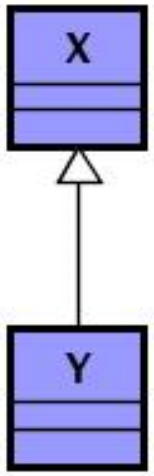
- X é **superclasse** de Y
- Y é **subclasse** de X

- X é a **classe base** de Y
- Y é uma **classe derivada** de X



Terminologia e efeitos

class Y (X)



- X é **superclasse** de Y
- Y é **subclasse** de X

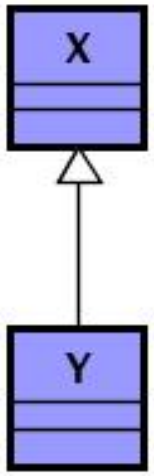
- X é a **classe base** de Y
- Y é uma **classe derivada** de X

- Y é uma **extensão** de X
- Y **estende** X

powered by Astah

Terminologia e efeitos

class Y (X)



- X é **superclasse** de Y
- Y é **subclasse** de X

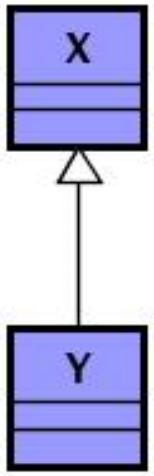
- X é a **classe base** de Y
- Y é uma **classe derivada** de X

- Y é uma **extensão** de X
- Y **estende** X

- X é uma **generalização** de Y
- Y é uma **especialização** de X

Terminologia e efeitos

class Y (X)



- X é **superclasse** de Y
- Y é **subclasse** de X

- X é a **classe base** de Y
- Y é uma **classe derivada** de X

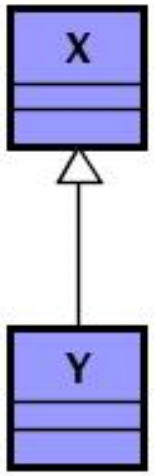
- Y é uma **extensão** de X
- Y **estende** X

- X é uma **generalização** de Y
- Y é uma **especialização** de X

- X e Y possuem um relacionamento de **herança**
- Y **herda** os membros (atributos e métodos) de X

Terminologia e efeitos

class Y (X)



- X é **superclasse** de Y
- Y é **subclasse** de X

- X é a **classe base** de Y
- Y é uma **classe derivada** de X

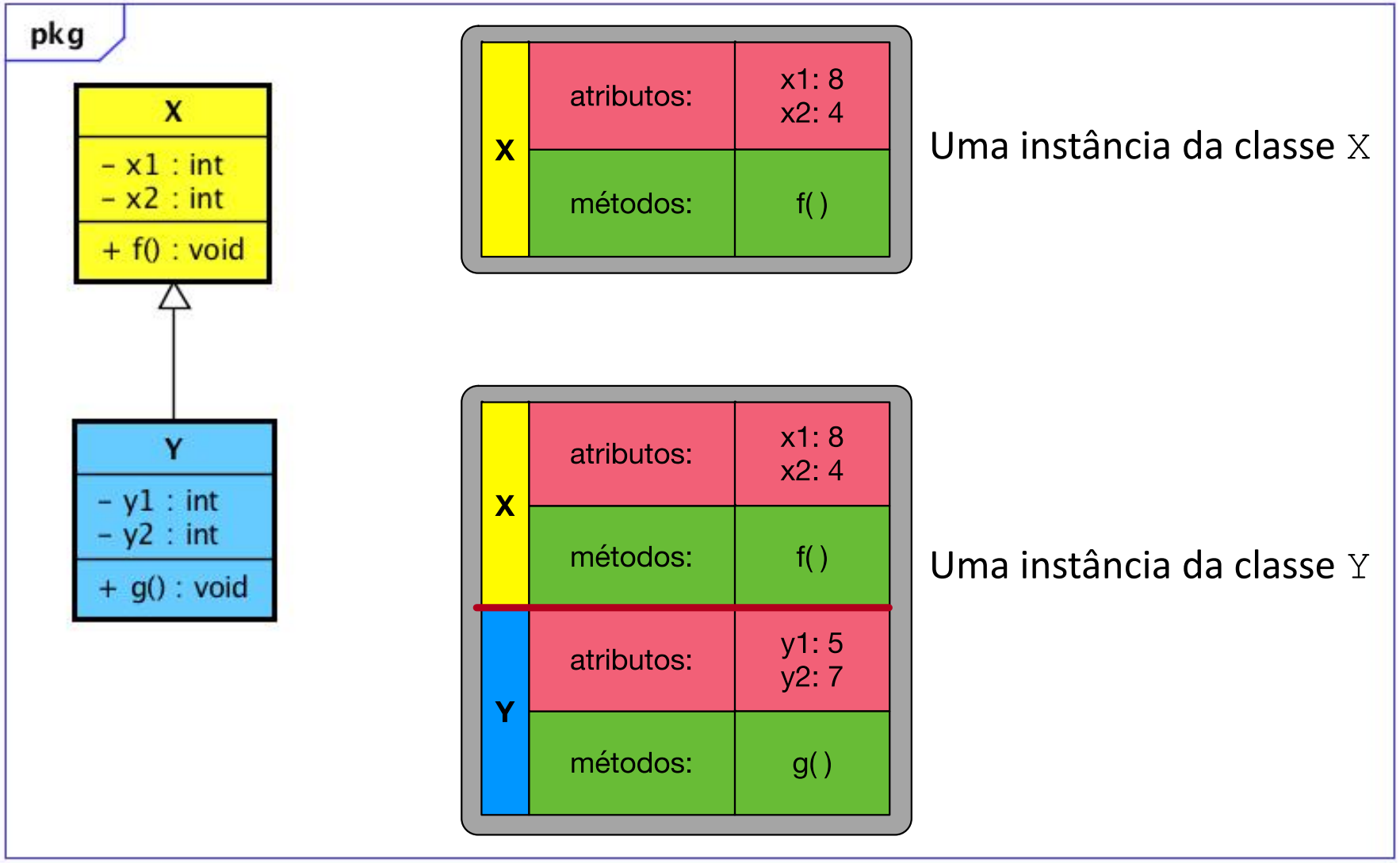
- Y é uma **extensão** de X
- Y **estende** X

- X é uma **generalização** de Y
- Y é uma **especialização** de X

- X e Y possuem um relacionamento de **herança**
- Y **herda** os membros (atributos e métodos) de X

- Uma instância de Y é uma instância de X
- Um objeto da classe Y possui todos os membros definidos por Y e X

Instanciação



Encadeamento de construtores

```
class X:
    def __init__(self, x1: int, x2: int):
        self.__x1: int = x1
        self.__x2: int = x2
```

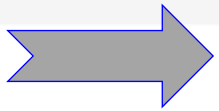
```
class Y(X):
    def __init__(self, x1: int, x2: int, y1: int, y2: int):
        super().__init__(x1, x2) # chama a inicialização da superclasse
        self.__y1: int = y1
        self.__y2: int = y2
```

Herança de método

```
class A:
    def r(self):
        print("@")

    def s(self):
        print("#")
```

```
class B(A):
    def f(self):
        self.r()
        self.s()
```

```
a = A()
b = B()
a.r()
a.s()
b.r()
b.s()
b.f()
```



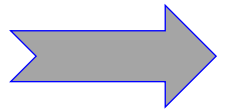
```
@
#
@
#
@
#
```

Sobrescrita de método

```
class A:
    def r(self):
        print("@")

    def s(self):
        print("#")
```

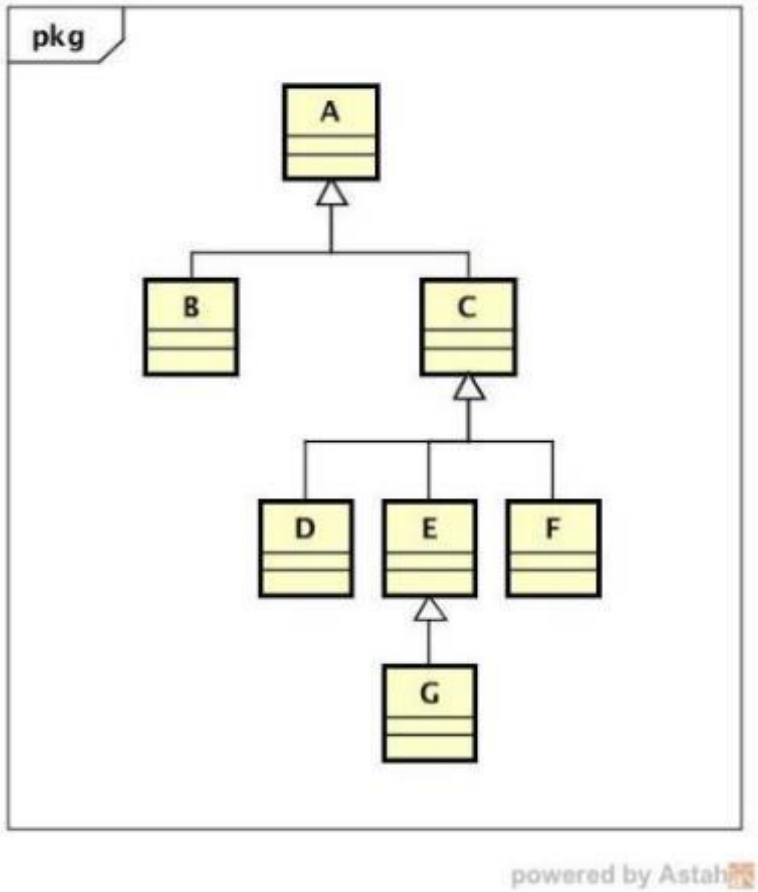
```
class C(A):
    def r(self):
        print("%")

    def s(self):
        super().s()
        print("$")
```

```
a = A()
c = C()
a.r()
a.s()
c.r()
c.s()
```



```
@
#
%
#
$
```

Hierarquia de classes



Pode haver qualquer quantidade de níveis.

G é subclasse de E, que é subclasse de C, que é subclasse de A.

A é superclasse de C, que é superclasse de E, que é superclasse de G.

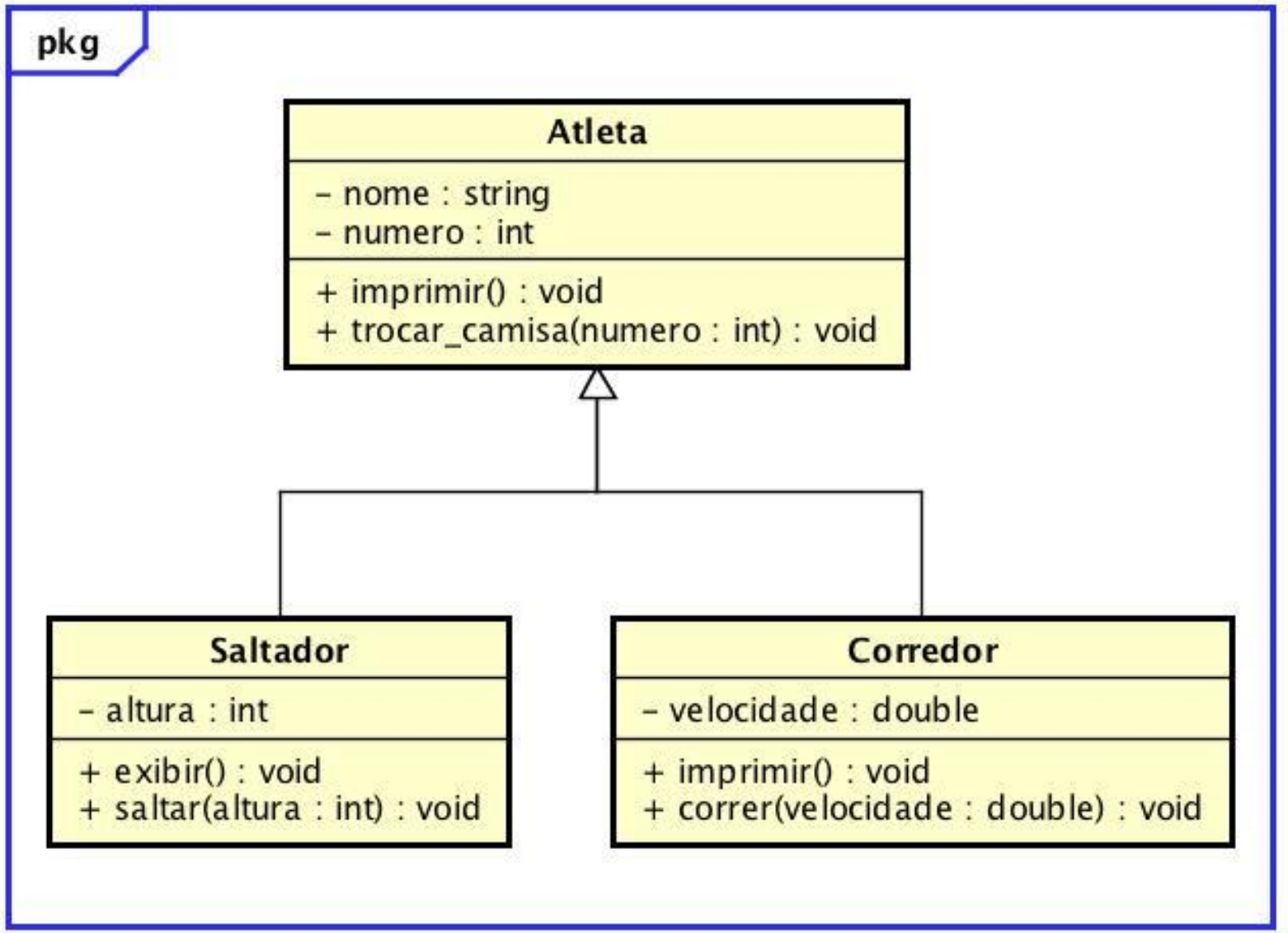
A é superclasse direta de B e C e é superclasse indireta de D, E, F e G.

G é subclasse direta de E e é subclasse indireta de C e A.

A é a **raiz** da hierarquia.

Em Python, a raiz de toda hierarquia é a classe `Object`.

Exemplo Herança


pkg
Atleta
- nome : string
- numero : int
+ imprimir() : void
+ trocar_camisa(numero : int) : void
Saltador
- altura : int
+ exibir() : void
+ saltar(altura : int) : void
Corredor
- velocidade : double
+ imprimir() : void
+ correr(velocidade : double) : void
powered by Astah

Classe Atleta

```
class Atleta:
    def __init__(self, nome, numero):
        self.__nome = nome
        self.__numero = numero

    def imprimir(self):
        print(self.__nome)
        print(self.__numero)

    def trocar_camisa(self, numero):
        self.__numero = numero
```

Classe Saltador – tipo de atleta

```
from atleta import Atleta


class Saltador(Atleta):
    def __init__(self, nome, numero, altura):
        super().__init__(nome, numero)
        self.__altura = altura

    def exibir(self):
        self.imprimir()
        print(self.__altura)

    def saltar(self, altura):
        self.__altura = altura
```

Classe Corredor – tipo de Atleta

```
from atleta import Atleta


class Corredor(Atleta):
    def __init__(self, nome, numero, velocidade):
        super().__init__(nome, numero)
        self.__velocidade = velocidade

    def imprimir(self):
        super().imprimir()
        print(self.__velocidade)

    def correr(self, velocidade):
        self.__velocidade = velocidade
```

```
from atleta import Atleta
from corredor import Corredor
from saltador import Saltador

if __name__ == "__main__":

    falcao = Atleta("Paulo Roberto Falcao", 5)
    sotomayor = Saltador("Javier Sotomayor", 76, 245)
    bolt = Corredor("Usain Bolt", 709, 37.58)

    falcao.imprimir()
    sotomayor.imprimir()
    sotomayor.exibir()
    bolt.imprimir()

    falcao.trocar_camisa(10)
    sotomayor.trocar_camisa(31)
    bolt.trocar_camisa(2163)
    sotomayor.saltar(233)
    bolt.correr(36.92)

    falcao.imprimir()
    sotomayor.exibir()
    bolt.imprimir()
```

```
Paulo Roberto Falcao
5
Javier Sotomayor
76
Javier Sotomayor
76
245
Usain Bolt
709
37.58
Paulo Roberto Falcao
10
Javier Sotomayor
31
233
Usain Bolt
2163
36.92
```

Exercício 3

- Herança
  - O banco pode ter dois tipos de conta
    - Conta Corrente
    - Conta Poupança
  - Além do mais, uma mesma conta corrente pode estar associada a uma conta poupança também
    - Criar uma estrutura de herança entre Conta (Generalização) e ContaCorrente e ContaPoupança (especializações).
    - ContaPoupança tem rendimento, tem seu saldo próprio e suas transações próprias.
    - ContaCorrente tem taxa de serviços, saldo, limite.