Programação Orientada a Objetos

Aula 5 - Objetos e suas inter-relações
Parte 2

Prof. Ronald Chiesse de Souza
ronaldsouza@dcc.ufrj.br

Conceitos

- Inner Classes
- Agregação versus composição
- Métodos e classes abstratas

Inner classes

Inner Classes em Java

- Considere a classe Carro, com atributos e métodos como marca, modelo, acelera(), freia(), etc.

- E se quisermos armazenar as informações do motor do carro? Faz sentido colocar na classe Carro?
  - Ideal (mais organizado) é criar uma outra classe, a classe Motor.

Inner Class

- Faz sentido a classe Motor existir fora da classe Carro? O sistema usa a classe Motor em outro lugar?
  - Se não é o caso, então o ideal é declarar a classe Motor ***dentro*** da classe Carro, como uma ***inner class***
- **Inner class** organiza melhor o código, pois permite criar um tipo complexo mas que não deve ser independente.
- Acesso externo à classe Motor depende de um objeto do tipo Carro!

```
class Carro{
  class Motor{
    String modelo;
    public void liga(){}
  }
  Motor motor;
  String cor, modelo;
  public void liga(){ motor.liga(); }
}

class CarroTeste {
  public static void main(String[]
args) {
    Carro carro = new Carro();
    carro.motor = carro.new Motor();
    carro.liga();
  }
}
```

Inner Class

- Mas o código ao lado pode melhorar!

→ O que não está fazendo sentido nele?

```
class Carro{
  class Motor{
    String modelo;
    public void liga(){}
  }
  Motor motor;
  String cor, modelo;
  public void liga(){ motor.liga(); }
}

class CarroTeste {
  public static void main(String[]
args) {
    Carro carro = new Carro();
    carro.motor = carro.new Motor();
    carro.liga();
  }
}
```

Inner Class

- Mas o código ao lado pode melhorar!

→ O que não está fazendo sentido nele?

R: Manipulação direta de um objeto 'motor' a partir da main().

```
class Carro{
  class Motor{
    String modelo;
    public void liga(){}
  }
  Motor motor;
  String cor, modelo;
  public void liga(){ motor.liga(); }
}

class CarroTeste {
  public static void main(String[]
args) {
    Carro carro = new Carro();
    carro.motor = carro.new Motor();
    carro.liga();
  }
}
```

Inner Class

- 1ª tentativa:

Criar o construtor para a classe Carro, e tornar o objeto ‘motor’ privado.

→ **É o bastante?**

```
public class Carro{
  class Motor{
    String modelo;
    public void liga(){}
  }
  private Motor motor;
  private String cor, modelo;
  public Carro(){motor = new Motor();}
  public void liga(){ motor.liga(); }
}
```

Inner Class

- 1ª tentativa:

Criar o construtor para a classe Carro, e tornar o objeto ‘motor’ privado.

→ **É o bastante?**

R: Não. A seguinte situação (na driver class) ainda pode acontecer – o que não faz sentido.

```
public class Carro{
  class Motor{
    String modelo;
    public void liga(){}
  }
  private Motor motor;
  private String cor, modelo;
  public Carro(){motor = new Motor();}
  public void liga(){ motor.liga(); }
}

class CarroTeste {
  public static void main(String[]
args) {
    Carro carro = new Carro();
    Carro.Motor m = carro.new Motor();
    m.liga();
  }
}
```

Inner Class

- 2ª tentativa:

Criar o construtor para a classe Carro, e tornar **a classe** Motor **e o objeto** 'motor' privados.

→ Agora sim! ☺

```
public class Carro{
  private class Motor{
    String modelo;
    public void liga(){}
  }
  private Motor motor;
  private String cor, modelo;
  public Carro(){motor = new Motor();}
  public void liga(){ motor.liga(); }
}

class CarroTeste {
  public static void main(String[]
args) {
    Carro carro = new Carro();
    carro.liga();
  }
}
```

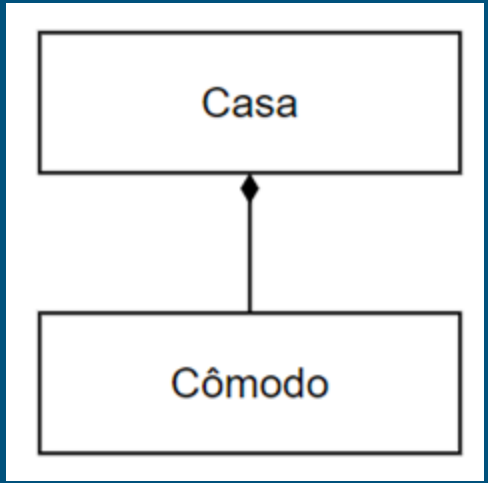
Agregação e Composição

Agregação e Composição

- Em alguns casos, a herança não consegue representar bem a relação entre duas classes. Para isso existem outros dois paradigmas usados, agregação e composição.
- Diferente da herança, nessa modelagem usamos classes diferentes (com métodos e atributos distintos), que se relacionam através de atributos que apontam para a outra classe.

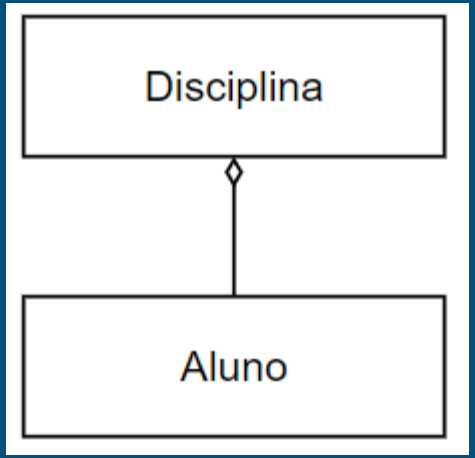
Composição

- A composição é um paradigma onde uma classe “contém” outra dentro de si, através do uso de atributos.
- Nessa modelagem, os objetos contidos só existem dentro da classe, e que quando o objeto que contém é deletado, os outros objetos dentro dele também são.
- No nosso exemplo, uma casa contém vários cômodos, que só existem dentro do contexto da casa. É impossível existir um cômodo que não esteja atrelado a uma casa. E um cômodo só pode estar conectado a uma casa.



Agregação

- Já no caso da agregação, esse relacionamento é um pouco mais “solto”. Uma classe agrega a outra, mas não é algo exclusivo. Um mesmo objeto pode ser referenciado em vários outros objetos diferentes.
- Da mesma forma, quando deletamos um objeto container, os objetos contidos não são deletados, pois existem de forma independente.
- No nosso exemplo, uma disciplina contém alunos, porém, o mesmo aluno pode estar em várias disciplinas, e encerrar uma disciplina não some com os alunos.



Métodos e Classes Abstratas

Classes Abstratas

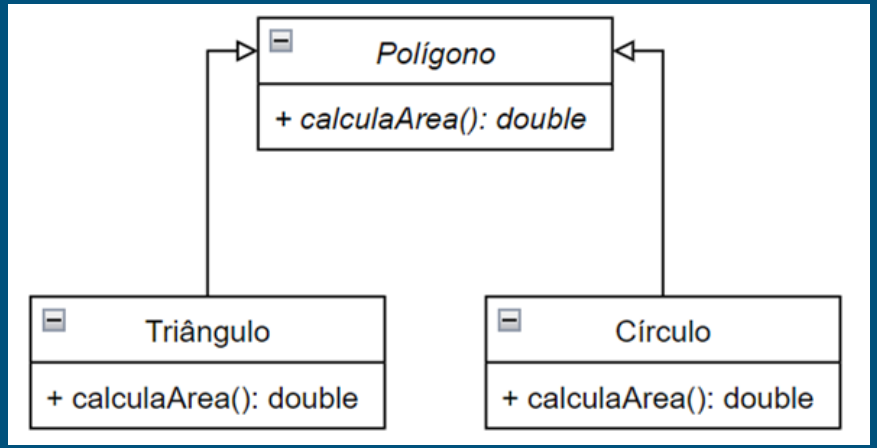
- Normalmente quando definimos uma classe para ser usada em um esquema de herança, ela pode ser instanciada normalmente. Porém, nem sempre é desejado que isso aconteça.
- Para isso, existem as classes abstratas, que nada mais são que classes que não podem ser instanciadas.
- Elas são úteis quando queremos que uma classe seja usada apenas para herança, por não representar algo concreto o suficiente para ser um objeto na modelagem.

Métodos Abstratos

- De forma semelhante, métodos abstratos são métodos que não possuem implementação, e que precisam necessariamente serem implementados pelas classes filhas.
- Eles são úteis quando é necessário que todas as classes em uma modelagem tenham uma função, mas sua execução seja completamente diferente em cada caso.
- Dessa forma, toda classe que herdar de uma classe com um método abstrato tem que implementar esse método, ou o código não compila.

Métodos e Classes Abstratos - Exemplo

- Expandindo nosso exemplo anterior de polígonos. Esperamos que todos que herdam tenham uma função que calcule a área. Porém, cada um com sua própria implementação.
- Dessa forma, tanto triângulo quanto Círculo precisam implementar o método.



Declaração de Polígono e Triângulo

```
public abstract class
Poligono {
  // uma classe abstrata pode
ter atributos

  // não faz sentido ter
construtor declarado

  public abstract double
calculaArea();

  // métodos aqui
}
```

```
public class Triangulo extends Poligono {
  // atributos específicos
  double altura, base;

  // construtor(es)

  public double calculaArea(){
    return this.altura*this.base/2;
  }

  // métodos aqui
}
```