Linguagem de Programação I

Aula 5

Programação Orientada a Objetos – Composição e Herança

l.bertholdo@ifsp.edu.br

Conteúdo

- Relacionamentos entre Classes
- Composição
- Herança
- Herança x Composição

l.bertholdo@ifsp.edu.br

Relacionamentos entre Classes

- Classes podem se relacionar entre si, de modo que umas possam utilizar os serviços das outras.

- Alguns exemplos de relacionamentos:

  - Classes podem **ser compostas** por outras classes (composição).

    *Um objeto **Cliente** pode ser composto por um objeto **Contato**.*

  - Classes podem **ser do mesmo tipo** que outras classes (generalização/especialização ou herança).

    *Um objeto **Casa** é também um objeto **Imovel**.*

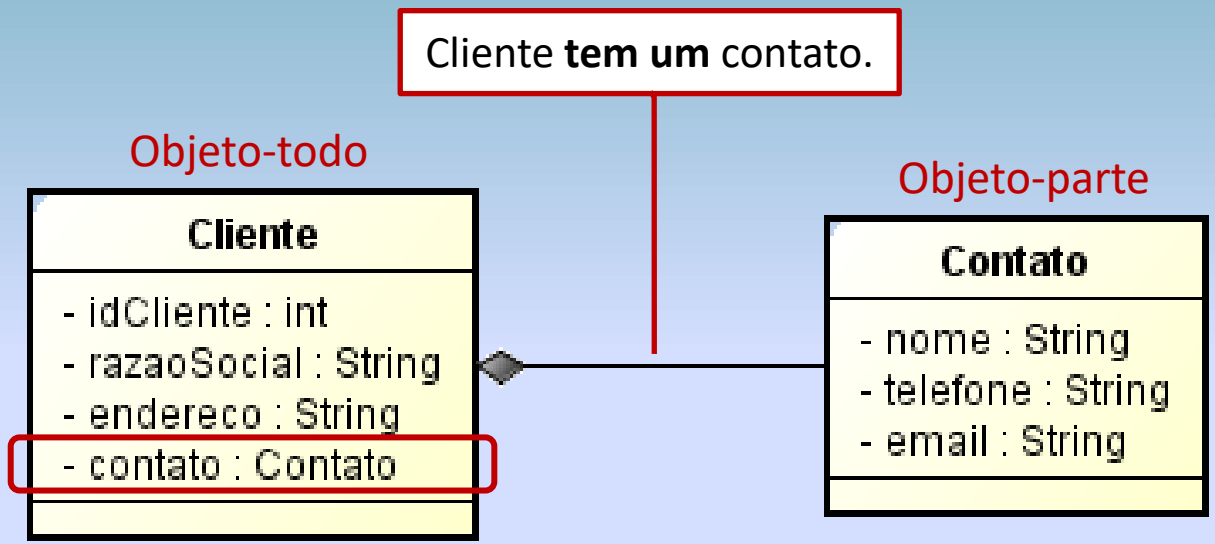
l.bertholdo@ifsp.edu.br

Composição

- Na composição, as entidades são **dependentes**. Seus objetos têm **ciclos de vida atrelados** e um dos objetos não existe sem o outro.
- O vínculo entre as entidades é forte, pois a existência do **objeto-parte** só faz sentido se ele estiver associado a um **objeto-todo**.
- A composição é implementada incluindo um **objeto-parte** como atributo de um **objeto-todo**, sendo que o **objeto-parte** deve ser instanciado (criado com a palavra **new**) pelo **objeto-todo**.
- Assim, o **objeto-todo** é responsável por criar e destruir seus **objetos-parte**.
- Esse relacionamento também é chamado de "**tem-um**".

Composição

- Exemplo



***Contatos** dependem de **Clientes** para existirem.*

*Ao excluir um **cliente**, seu respectivo **contato** também deve deixar de existir.*

Composição

```java
public class Contato {
    private String nome;
    private String telefone;
    private String email;

    public String getContato() {
        return ("\n  Nome: " + this.nome +
                "\n  Telefone: " + this.telefone +
                "\n  E-mail: " + this.email);
    }

    public void setContato(String nome, String telefone, String email) {
        this.nome = nome;
        this.telefone = telefone;
        this.email = email;
    }
}
```

```java
public class Cliente {
    private int idCliente;
    private String razaoSocial;
    private String endereco;
    private Contato contato;

    public Cliente() { this.contato = new Contato(); }

    public int getIdCliente() { return idCliente; }
    public String getRazaoSocial() { return razaoSocial; }
    public String getEndereco() { return endereco; }
    public String getContato() { return contato.getContato(); }

    public void setIdCliente(int idCliente) { this.idCliente = idCliente; }
    public void setRazaoSocial(String razaoSocial) { this.razaoSocial = razaoSocial; }
    public void setEndereco(String endereco) { this.endereco = endereco; }
    public void setContato(String nome, String telefone, String email) {
        this.contato.setContato(nome, telefone, email);
    }
}
```

A instanciação do atributo **contato** no construtor da classe define a composição entre as duas classes.

bertholdo@ifsp.edu.br

Composição

```
public class Inicio {
    public static void main(String[] args) {
        Cliente cliente = new Cliente();
        cliente.setIdCliente(1);
        cliente.setRazaoSocial("Supermercado Araguari Ltda");
        cliente.setEndereco("Rua dos Javalis, 320");
        cliente.setContato("João Ribeiro", "1122227777", "contato@suparaguari.com.br");

        System.out.println("Dados do Cliente:");
        System.out.println("Código: " + cliente.getIdCliente());
        System.out.println("Razão Social: " + cliente.getRazaoSocial());
        System.out.println("Endereço: " + cliente.getEndereco());
        System.out.println("Dados de Contato: " + cliente.getContato());
    }
}
```

O método **setContato** envia os dados do Contato.

```
Dados do Cliente:
Código: 1
Razão Social: Supermercado Araguari Ltda
Endereço: Rua dos Javalis, 320
Dados de Contato:
  Nome: João Ribeiro
  Telefone: 1122227777
  E-mail: contato@suparaguari.com.br
```

O método **getContato** retorna uma *string* contendo os dados do objeto Contato.

Composição

- Testando a **dependência** entre os objetos.

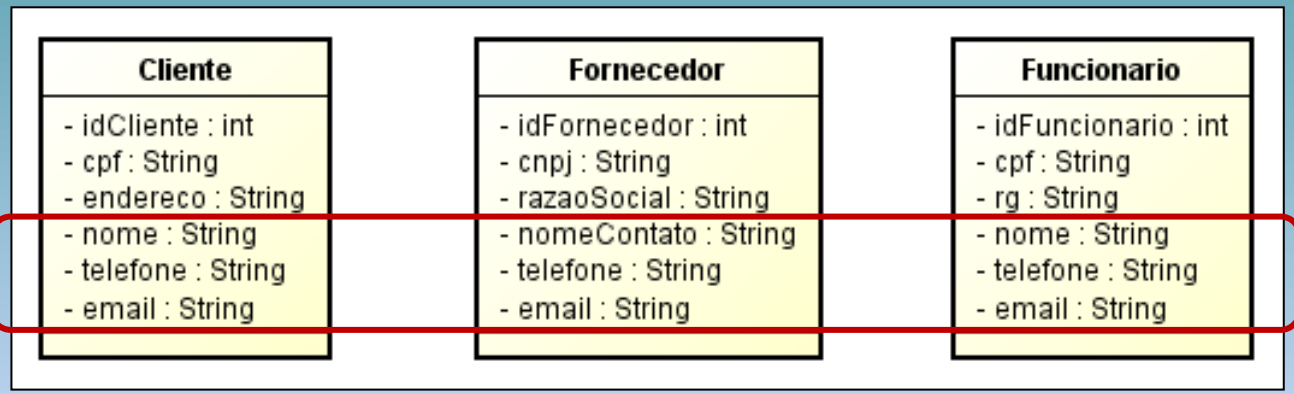
*Ao instanciar **cliente** novamente, os dados do cliente são "zerados", e os dados de contato também.*

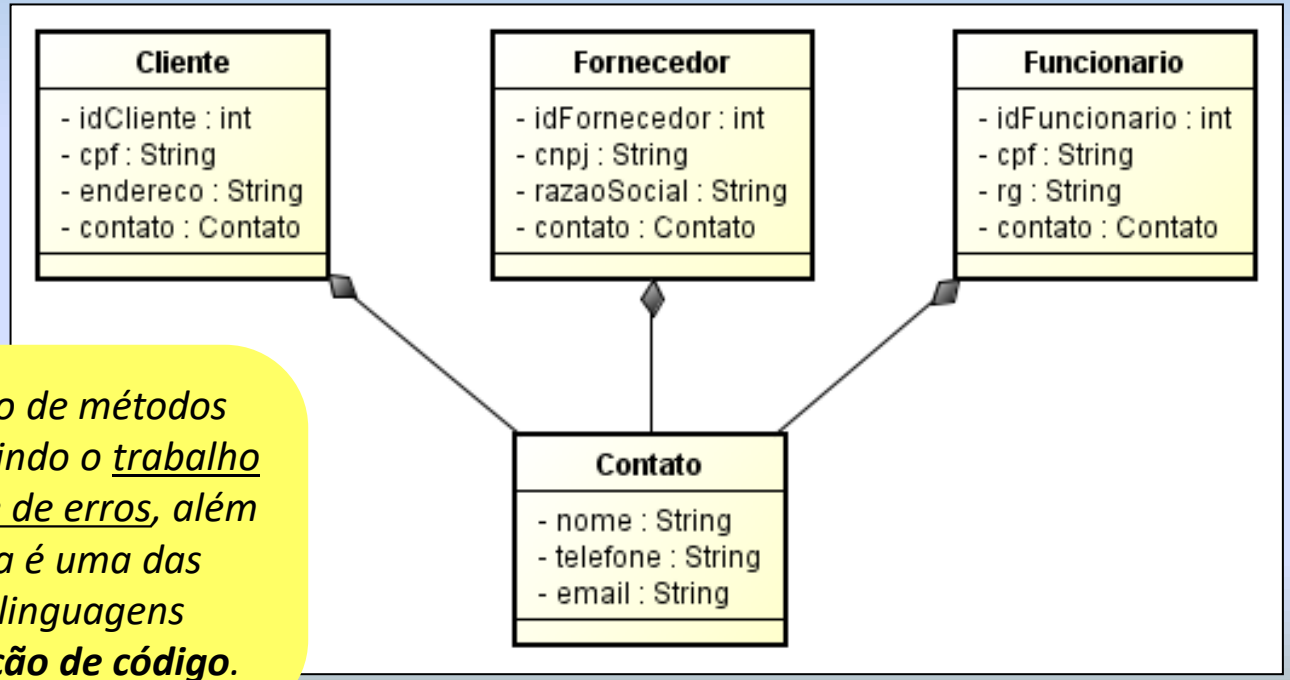
```
public class Inicio {
    public static void main(String[] args) {
        Cliente cliente = new Cliente();
        cliente.setIdCliente(1);
        cliente.setRazaoSocial("Supermercado Araguari Ltda");
        cliente.setEndereco("Rua dos Javalis, 320");
        cliente.setContato("João Ribeiro", "1122227777", "contato@suparaguari.com.br");

        cliente = new Cliente();

        System.out.println("Dados do Cliente:");
        System.out.println("Código: " + cliente.getIdCliente());
        System.out.println("Razão Social: " + cliente.getRazaoSocial());
        System.out.println("Endereço: " + cliente.getEndereco());
        System.out.println("Dados de Contato: " + cliente.getContato());
    }
}
```

```
Dados do Cliente:
Código: 0
Razão Social: null
Endereço: null
Dados de Contato:
  Nome: null
  Telefone: null
  E-mail: null
```

Composição

- O que é melhor?





*A **composição** evita a duplicação de métodos que têm a mesma função, diminuindo o trabalho do programador e a possibilidade de erros, além de facilitar a manutenção. Essa é uma das principais características das linguagens orientadas a objetos: a **reutilização de código**.*

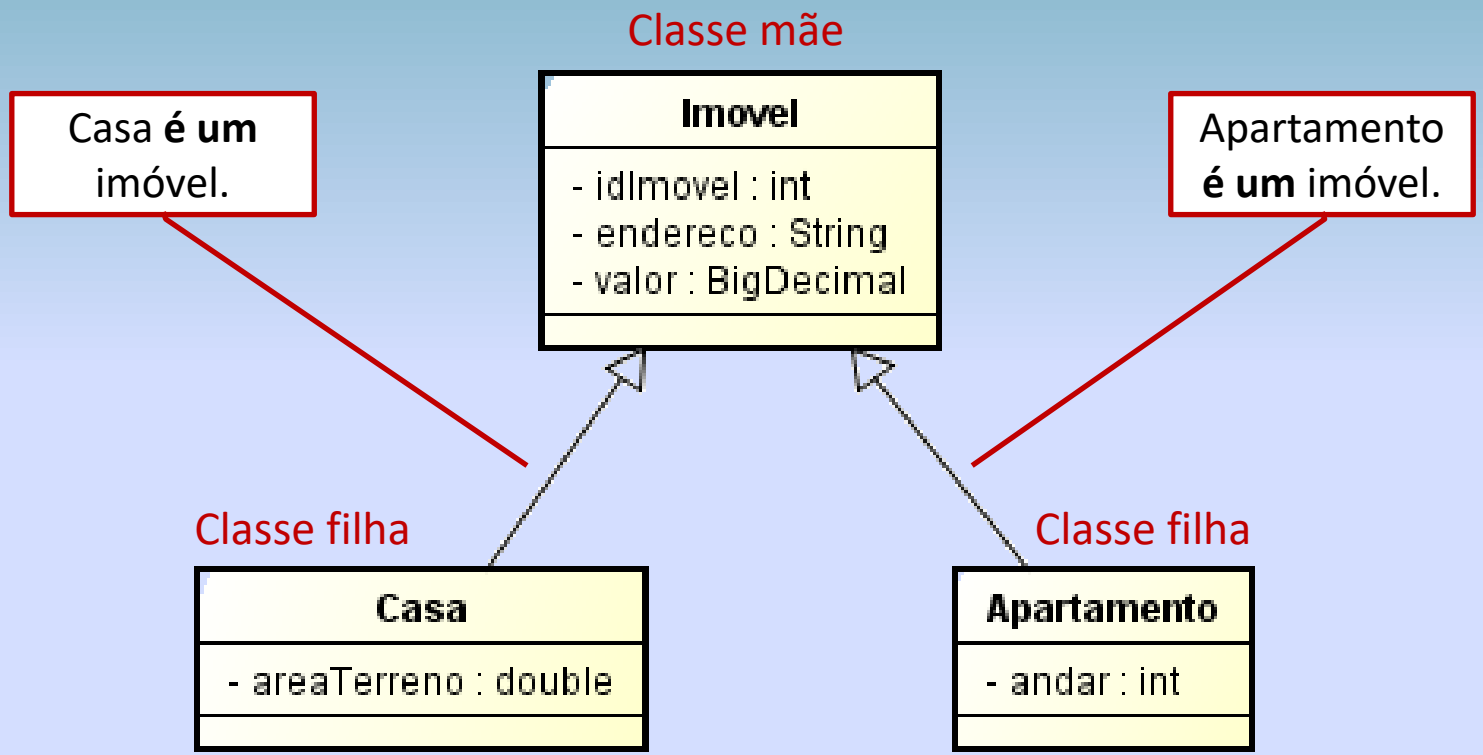
Herança

- A herança permite construir classes baseadas em outras já existentes. A classe original é chamada de **classe mãe** ou **superclasse** e a classe criada a partir da classe original é chamada **classe filha** ou **subclasse**.
- Em uma estrutura de herança, a classe mãe contém atributos e métodos **genéricos**. E as classes filhas contêm atributos e métodos **específicos**, tendo a possibilidade de usar também os atributos e métodos genéricos da classe mãe.
- O objetivo da herança é evitar a duplicidade de código por meio da **reutilização de atributos e métodos** de classes genéricas.
- Esse relacionamento também é chamado de "**é-um**".

l.bertholdo@ifsp.edu.br

Herança

- Exemplo



Herança

- Para que a classe filha possa herdar atributos e métodos de uma classe mãe, o nível de proteção destes elementos na classe mãe deve ser **protected**.
- Em Java, a herança é declarada incluindo-se a palavra **extends** entre o nome da classe filha e o nome a classe mãe.

  Exemplo: class ClasseFilha extends ClasseMae
- Em alguns casos, para a classe filha invocar os atributos e métodos herdados da classe mãe, deve-se usar o operador **super**, seguido de ponto (.) antes do nome do atributo ou método. **Exemplos:** super.atributo, super.metodo().

l.bertholdo@ifsp.edu.br

Herança

- Normalmente, o operador **super** é usado apenas nas seguintes situações:
  - **Atributos:** Quando a classe filha precisa invocar um atributo (público ou protegido) da classe mãe, que também existe com o mesmo nome na classe filha.
  - **Métodos:** Quando a classe filha precisa invocar um método (público ou protegido) da classe mãe, que está sobrescrito na classe filha.
  - **Construtores:** Quando a classe filha precisa invocar o construtor da classe mãe. Nesse caso, não é preciso incluir o nome do construtor na chamada. **Exemplo:** super([<argumentos>]).

Herança

```java
public class Inicio {
    public static void main(String[] args) {
        new Casa(); // Chama o construtor da classe Casa.
        new Apartamento(); // Chama o construtor da classe Apartamento.
    }
}
```

```java
public class Imovel {
    private int idImovel;
    private String endereco;
    private double valor;

    protected void setEndereco(String endereco){
        this.endereco = endereco;
        System.out.println("Endereço cadastrado: " + this.endereco);
    }

    protected void setValor(double valor) {
        this.valor = valor;
        System.out.println("Valor cadastrado: " + this.valor);
    }
}
```

*É comum que os atributos de uma **classe mãe** sejam definidos como **protected**, para facilitar o acesso das classes filhas, de modo que elas não precisem invocar métodos **get** e **set** da classe mãe para acessar e modificar estes atributos.*

Herança

*O uso do operador **super** é facultativo, pois a classe **Casa** não sobrescreve estes métodos. Até mesmo o operador **this** poderia ser usado aqui, já que os métodos foram herdados e, portanto, também pertencem à classe **Casa**.*

```
public class Casa extends Imovel{
    private int areaTerreno;

    public Casa(){
        // Chama o método SetEndereco da classe mãe (Imovel).
        super.setEndereco("Rua Manaus, 220");

        // Chama o método SetValor da classe mãe (Imovel).
        super.setValor(450000);

        // Chama o método SetAreaTerreno da classe Casa.
        this.setAreaTerreno(250);
    }

    public void setAreaTerreno(int areaTerreno) {
        this.areaTerreno = areaTerreno;
        System.out.println("Área cadastrada: " + this.areaTerreno);
    }
}
```

Herança

```
Endereço cadastrado: Rua Manaus, 220
Valor cadastrado: 450000.0
Área cadastrada: 250
Endereço cadastrado: Rua Itu, 301
Valor cadastrado: 300000.0
Andar cadastrado: 7
```

```
public class Apartamento extends Imovel {
    private int andar;

    public Apartamento(){
        // Chama o método SetEndereco da classe mãe (Imovel).
        super.setEndereco("Rua Itu, 301");

        // Chama o método SetValor da classe mãe (Imovel).
        super.setValor(300000);

        // Chama o método SetAndar da classe Apartamento.
        this.setAndar(7);
    }

    public void setAndar(int andar){
        this.andar = andar;
        System.out.println("Andar cadastrado: " + this.andar);
    }
}
```

*O ideal é que classes mãe e filhas fiquem em um **pacote separado**, assim outras classes (de outros pacotes) não poderão acessar o que estiver definido como **protected**.*

Herança x Composição

- Tanto a herança quanto a composição possuem suas vantagens e desvantagens. Porém, na maioria dos casos, a **composição** é mais interessante, porque:
  - Apresenta um **encapsulamento** mais forte, visto que na herança, as superclasses compartilham sua implementação com suas subclasses.
  - Proporciona menor **acoplamento** entre as classes, já que na herança, mudar a superclasse pode afetar todas as suas subclasses.
  - Melhora a **coesão** entre as classes, pois cada classe fica focada apenas em suas próprias reponsabilidades.

Encapsulamento: *Diz respeito à* ***privacidade*** *existente entre as classes, ou seja, o quanto a implementação de uma classe está visível para outras classes.*

Acoplamento: *Diz respeito à* ***dependência*** *existente entre as classes, ou seja, o quanto uma classe pode ser afetada por mudanças em outras classes com as quais ela está associada.*

Coesão: *Diz respeito à* ***responsabilidade*** *de cada classe, ou seja, o quanto cada classe está assumindo apenas as responsabilidades que estão dentro de seu limite de competência.*

Herança x Composição

- Quando usar herança então?
  - Quando a provável subclasse tiver um relacionamento do tipo **“é um”**, e não **“tem um”**, com a superclasse.
  - Quando a provável subclasse for **“um tipo de”** e não **“um papel desempenhado por“**. Exemplos: automóvel é um tipo de veículo, funcionário e cliente são papéis desempenhados por uma pessoa.
  - Quando for desejado, ou necessário, realizar alterações globais nas prováveis subclasses por meio da alteração de uma superclasse.

Referências

- Hélio Engholm Jr.; Análise e Design Orientado a Objetos. São Paulo: Novatec Editora, 2013.
- Martin Fowler; UML Essencial – Um Breve Guia para a Linguagem-padrão de Modelagem de Objetos – 3ª Edição. Porto Alegre: Bookman: 2005.
- Pablo Dall'Oglio; PHP – Programando com Orientação a Objetos – 2ª Edição. São Paulo: Novatec Editora, 2014.
- Rafael Santos; Introdução à Programação Orientada a Objetos usando Java – 2ª edição. Rio de Janeiro: Elsevier, 2013.