Aula anterior

Revisão de AED I

Programação Orientada a Objeto

Prof. Diego Silva Caldeira Rocha

Sumário (Preâmbulo)

- Programação Orientada a Objeto
  - Definições
  - Classe / Objeto
  - Construtores
  - Encapsulamento
  - Herança

Definições

- **Programação Orientada a Objetos (POO):** Método de implementação onde os programas são organizados como coleções de objetos cooperativos, cada objetos representando uma instância de uma classe.

Definições

Classe

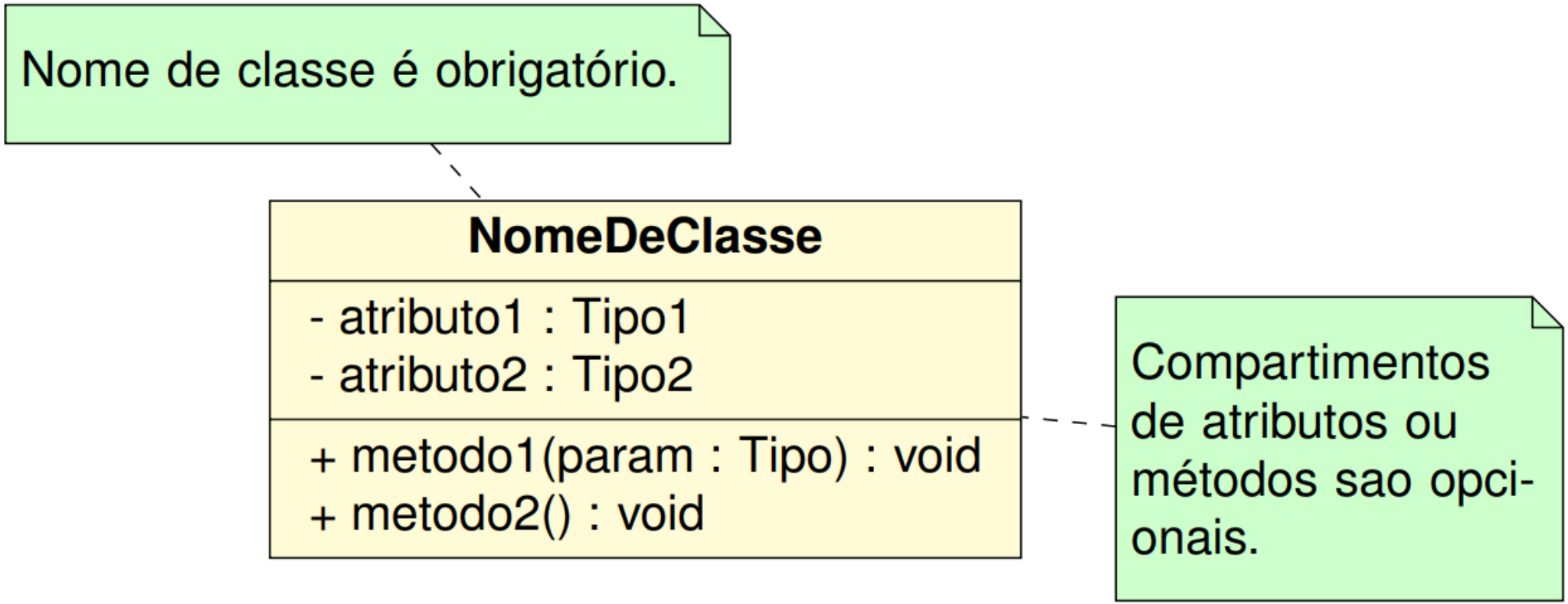
Uma classe é um tipo definido pelo usuário que contém o molde, a especificação para os objetos, assim como o tipo inteiro contém o molde para as variáveis declaradas como inteiros.

Objeto

Um objeto é uma instância de uma classe, ou seja, é a materialização de uma construção feita com base na forma/classe definida. Uma variável declarada de um tipo A (classe) seria um objeto da classe A.

- **Classe** é um tipo, um conjunto de regras.
- **Objeto** é uma variável do tipo classe.

UML



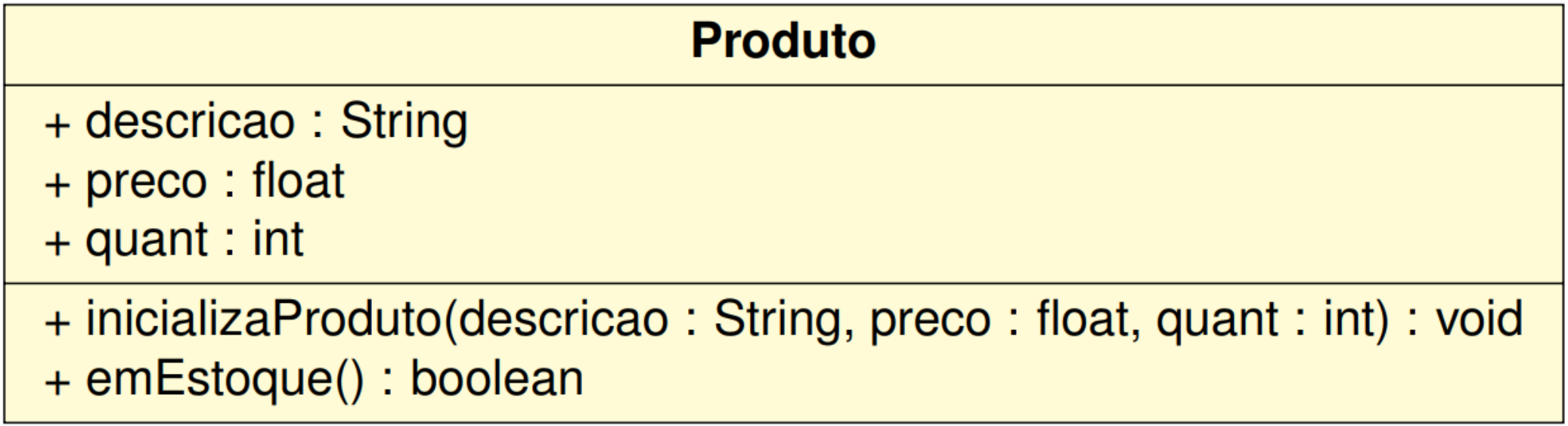
UML (*Unified Modeling Language*) permite representar classes e objetos para fins de modelagem de dados.

Exemplo: Estoque de produto

Criar uma classe Produto para um sistema de gerenciamento de estoque.

- Atributos:
  - descricao : String
  - preco : **float**
  - quant : **int**
- Métodos:
  - emEstoque() : bool
  - incializaProduto(String, **float**, **int**)

Definindo a classe de Produto: UML



Definindo a Classe Produto

```
class Produto {
    String descricao;
    float preco;
    int quant;

    boolean emEstoque()
    {
        return (quant > 0);
    }

    void inicializaProduto(String d, float p, int q)
    {
        descricao = d;
        preco = p;
        quant = q;
    }
}
```

Definindo a Classe Produto

```
class Produto {
    String descricao;
    float preco;
    int quant;

    boolean emEstoque()
    {
        return (quant > 0);
    }

    void inicializaProduto(String d, float p, int q)
    {
        descricao = d;
        preco = p;
        quant = q;
    }
}
```

Usando a Classe Produto

```
class Aplicacao {
    public static void main(String args[])
    {
        Produto p = new Produto();

        p.descricao = "Shulambs";
        p.preco = 1.99F;
        p.quant = 200;

        System.out.println("Produto: " + p.descricao);
        System.out.println("Preço: " + p.preco);
        System.out.println("Estoque: " + p.quant);

        if (p.emEstoque())
            System.out.println("Produto em estoque.");
    }
}
```

Criando Objetos

Produto p;

- Cria-se uma referência para um objeto do tipo Produto, mas não se aloca a memória para armazenar o objeto.
- Variável p aponta para NADA (**null**)

p = **new** Produto();

- Cria-se efetivamente o objeto Produto.
- Faz com que a referência p aponte para Produto.

Construindo um objeto

- Objetos são instâncias de uma classe:
  - Lê-se instância como sendo um elemento com o tipo da classe e um estado corrente individual.
- Exemplo:
  - Classe $\longrightarrow$ Produto (tipo com descricao, preco e quantidade)
  - Objeto de Produto $\longrightarrow$ $p = (Shulambs; R\$1,99; 200)$
- Ao criar um objeto sua memória é inicializada.
- Se não for definido um modo de inicialização o compilador usa valores padrão. Ex:
  - p = **new** Produto(); cria $(null, 0.0, 0)$

Construtores

- Objetos são instâncias de uma classe:
  - Lê-se instância como sendo um elemento com o tipo da classe e um estado corrente individual.
- Exemplo:
  - Classe $\longrightarrow$ Produto (tipo com descricao, preco e quantidade)
  - Objeto de Produto $\longrightarrow$ $p = (Shulambs; R\$1,99; 200)$
- Ao criar um objeto sua memória é inicializada.
- Se não for definido um modo de inicialização o compilador usa valores padrão. Ex:
  - p = **new** Produto(); cria $(null, 0.0, 0)$

Construtores

- Em Java se não for definido um modo de inicialização o compilador usa valores padrões. Ex:
  - p = **new** Produto(); cria $(null, 0.0, 0)$
- Construtores são usados para inicializar objetos com valores diferentes do padrão.
- Construtores:
  - Possuem o mesmo nome da classe.
  - Não possuem valores de retorno.
- Uma classe pode ter de 0 a muitos construtores.

Classe Produto usando Construtor

```
class Produto {
    ...

    Produto(String d, float p, int q)
    {
        if (d.length() >= 3)
            descricao = d;
        if (p > 0)
            preco = p;
        if (q >= 0)
            quant = q;
    }

    Produto() {
        descricao = "Novo Produto";
        preco = 0.01F;
        quant = 0;
    }
}
```

Classe Produto usando Construtor

```
class Produto {
    ...

    Produto(String d, float p, int q)
    {
        if (d.length() >= 3)
            descricao = d;
        if (p > 0)
            preco = p;
        if (q >= 0)
            quant = q;
    }

    Produto() {
        descricao = "Novo Produto";
        preco = 0.01F;
        quant = 0;
    }
}
```

Classe Produto usando Construtor

```
class Aplicacao {
    public static void main(String args[])
    {
        Produto p1 = new Produto();

        Produto p2 = new Produto("Shulambs", 1.99F, 200);

        System.out.println("Produto: " + p1.descricao);
        System.out.println("Preço: " + p1.preco);
        System.out.println("Estoque: " + p1.quant);

        System.out.println("Produto: " + p2.descricao);
        System.out.println("Preço: " + p2.preco);
        System.out.println("Estoque: " + p2.quant);

    }
}
```

Encapsulamento: ocultando informações

- Objetiva separar aspectos visíveis de um objeto ou classe de seus detalhes de implementação
- Interface:
  - tudo aquilo que o usuário do objeto vê/acessa.
- Permite alterar a implementação de um objeto sem impactos em outros módulos do sistema.
- Permite que seus dados sejam protegidos de acesso ilegal.
- Em geral, desejamos ocultar determinados dados e/ou métodos do cliente/usuário da aplicação.

Ocultando informações

Exemplo:

- Acessar o campo quant e definir um estoque negativo pode invalidar o Produto.

Solução:

- Encapsulamento.

Modificadores de acesso

- Modificadores de acesso controlam a visibilidade dos componentes na aplicação.
- Ao nível da classe: **public** ou *package-private* (sem modificador explícito).
  - Classe declarada como **public** é visível a todas as classes do programa.
  - Classe sem modificador de acesso é visível apenas em seu pacote.
- Ao nível dos membros (atributos e métodos): **public**, **private**, **protected**, ou *package-private* (sem modificador explícito).

Modificadores de acesso

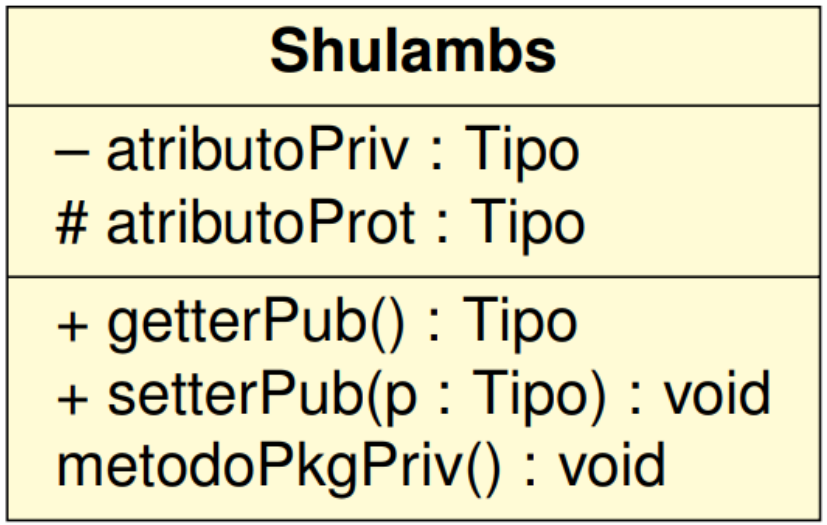
O Java possui 4 modificadores de acesso ao nível dos membros:

- **private**: membros declarados com acesso privado são acessíveis apenas na própria classe.
- *package-private*: membros declarados sem modificador de acesso são acessíveis apenas às classes dentro do mesmo pacote.
- **protected**: membros declarados com acesso protegido são acessíveis às classes do pacote e adicionalmente por suas subclasses.
- **public**: membros declarados com acesso público são acessíveis de qualquer lugar do programa.

Princípios da ocultação de informação

- Use o nível de acesso mais restrito e que faça sentido para um membro particular.
- Use **private** a menos que haja uma boa razão para não fazê-lo.
- Evite campos **public** exceto para constantes. Campos públicos aumentam o acoplamento em relação a uma implementação específica e reduz a flexibilidade do sistema a mudanças.

Encapsulamento na UML




```
class Shulambs {
    private Tipo atributoPriv;
    protected Tipo atributoProt;

    public Tipo getterPub() {
      ...
    }
    public void setterPub(Tipo p) {
      ...
    }

    void metodoPkgPriv() {
      ...
    }
}
```

Classe Produto: encapsulamento

```
public class Produto {
    private String descricao;
    private float preco;
    private int quant;

    public bool emEstoque() {
        return (quant > 0);
    }

    public Produto(String d, float p, int q)  {
        ...
    }

    public Produto() {
        ...
    }
}
```

Métodos de Acesso (*getters e setters*)

- Métodos *get*: acessam o valor de um atributo privado.
  - Valores podem ser tratados antes de serem exibidos.
  - Ex: atributo booleano sendo exibido como V ou F atributo numérico e seu correspondente string.

  Métodos *set*: atribuem um valor a um atributo privado.
  - Valores devem ser validados/tratados antes de serem atribuídos.
  - Ex: número do `dia` numa classe Data depende do atributo `mes`.

Classe Produto: (*getters e setters*)

```
public String getDescricao() { return descricao; }
public float getPreco() { return preco; }
public int getQuant() { return quant; }

public void setDescricao(String d) {
    if (d.length() >= 3)    descricao = d;
}
public void setPreco(float p) {
    if (preco > 0)   preco = p;
}
public void setQuant(int q) {
    if (quant >= 0)   quant = q;
}
public Produto(String d, float p, int q)
{
    setDescricao(d);
    setPreco(p);
    setQuant(q);
}
```

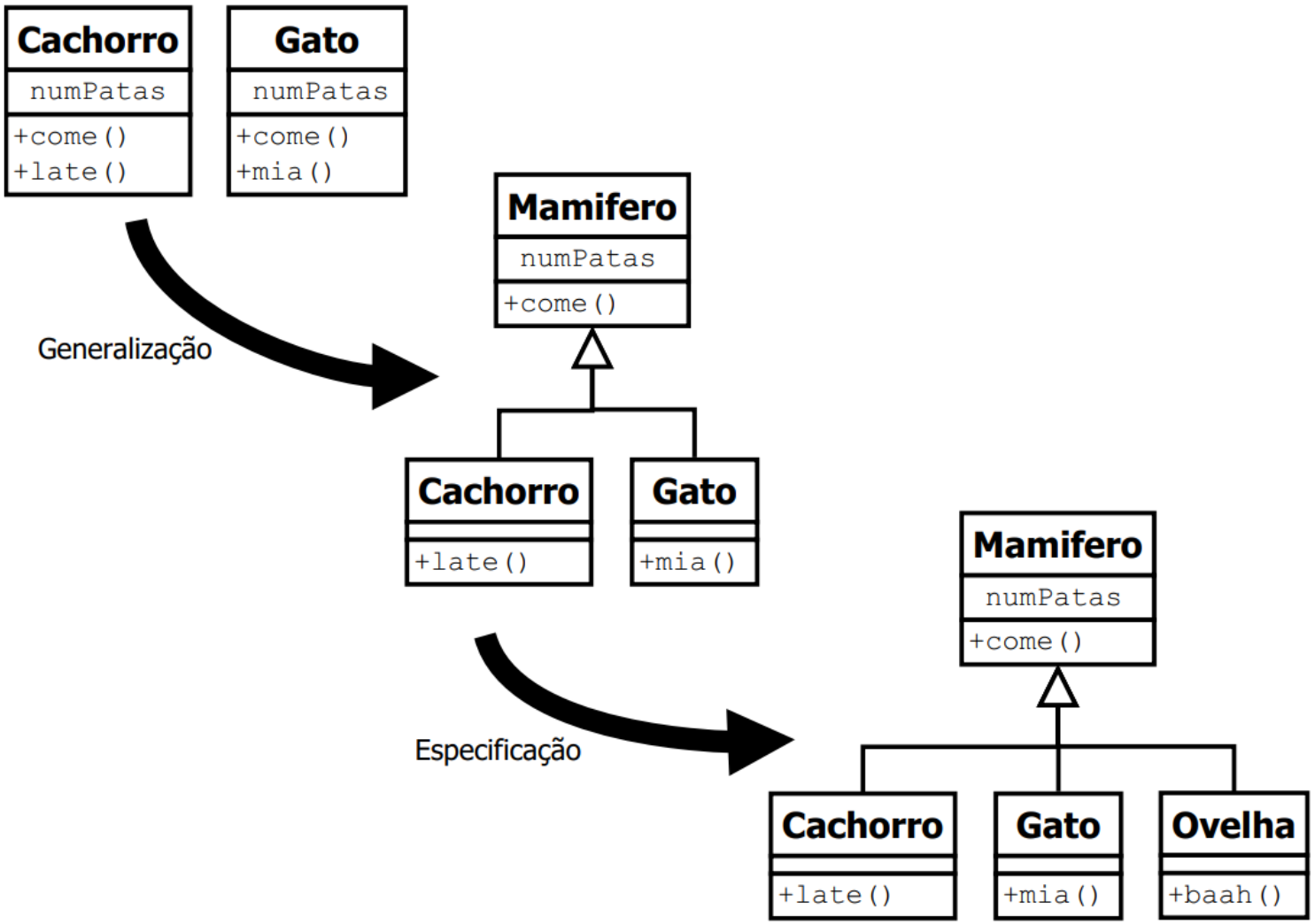
Classe Produto: (*getters e setters*)

```
class Aplicacao {
    public static void main(String args[])
    {
        Produto p1 = new Produto();
        Produto p2 = new Produto("Shulambs,1.99F,200);

        p1.setDescricao("Cool Shulambs");
        p1.setPreco(2.49F);
        p1.setQuant(10);

        System.out.println("Produto: " + p1.getDescricao());
        System.out.println("Preço: " + p1.getPreco());
        System.out.println(" Estoque: " + p1.getQuant());

        System.out.println("Produto: " + p2.getDescricao());
        System.out.println("Preço: " + p2.getPreco());
        System.out.println(" Estoque: " + p2.getQuant());

    }
}
```

Herança

- Mecanismo para definição de uma classe em termos de outra classe existente.
- Relação: *é um tipo de* / *é um*.
- Herança permite o reuso do comportamento de uma classe na definição de outra.
- A classe derivada herda todas as características de sua classe base, podendo adicionar novas características.
- Baseada em dois princípios fundamentais do projeto de software:
  - especificação *(top-down)*
  - generalização / abstração *(bottom-up)*

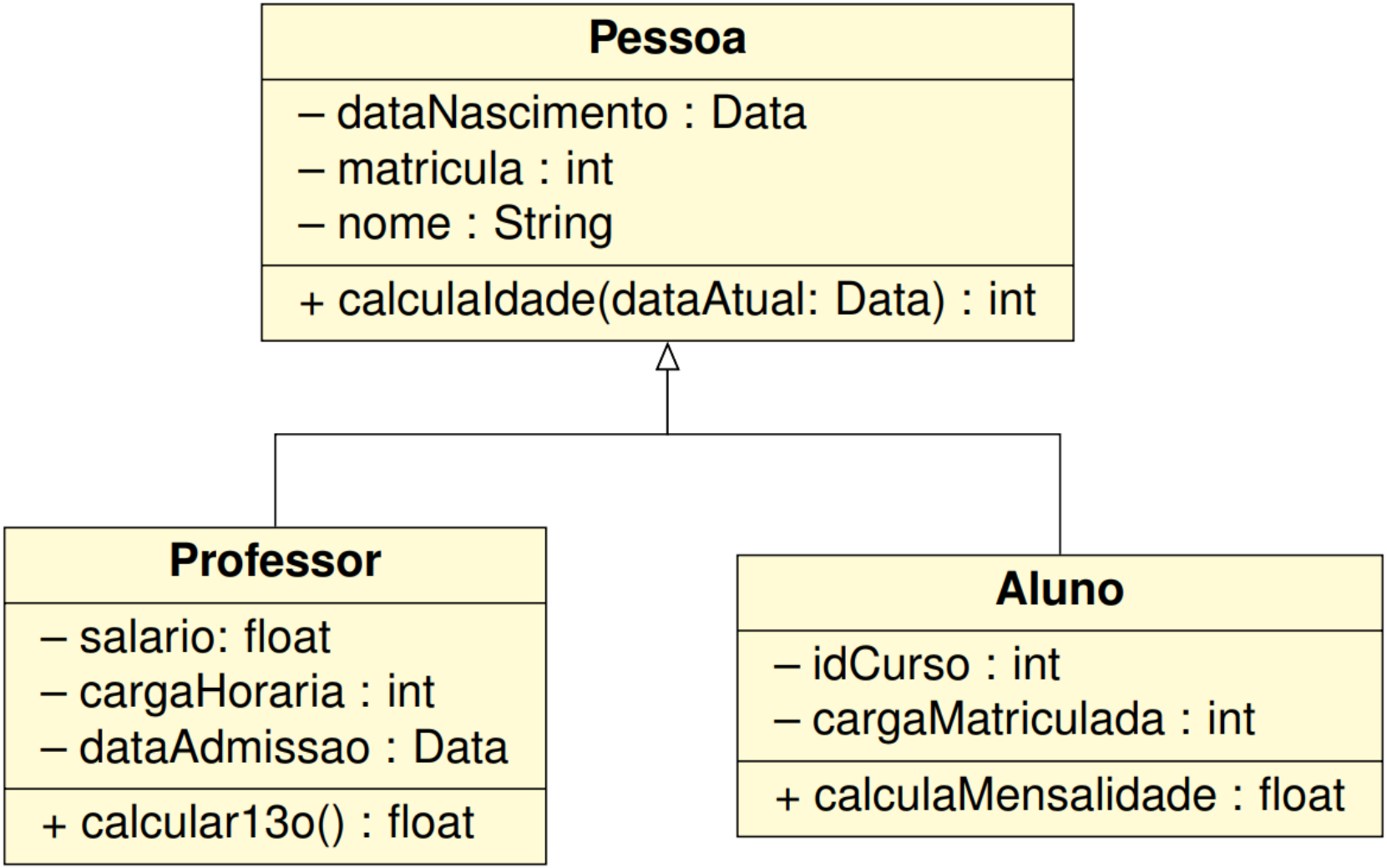
Generalização / Especialização



Herança Simples

- Novas classes, chamadas de classes derivadas (ou subclasses), são definidas a partir de apenas uma classe base (ou superclasse).
- Exemplos:
  - Um professor é uma pessoa.
  - Um ônibus é um veículo.
  - Um automóvel é um veículo.
- Membros da classe base podem ser redefinidos na classe derivada.
- Em Java, qualquer classe herda da classe Object.
- Em Java, usa-se a palavra chave **extends**, para indicar herança.

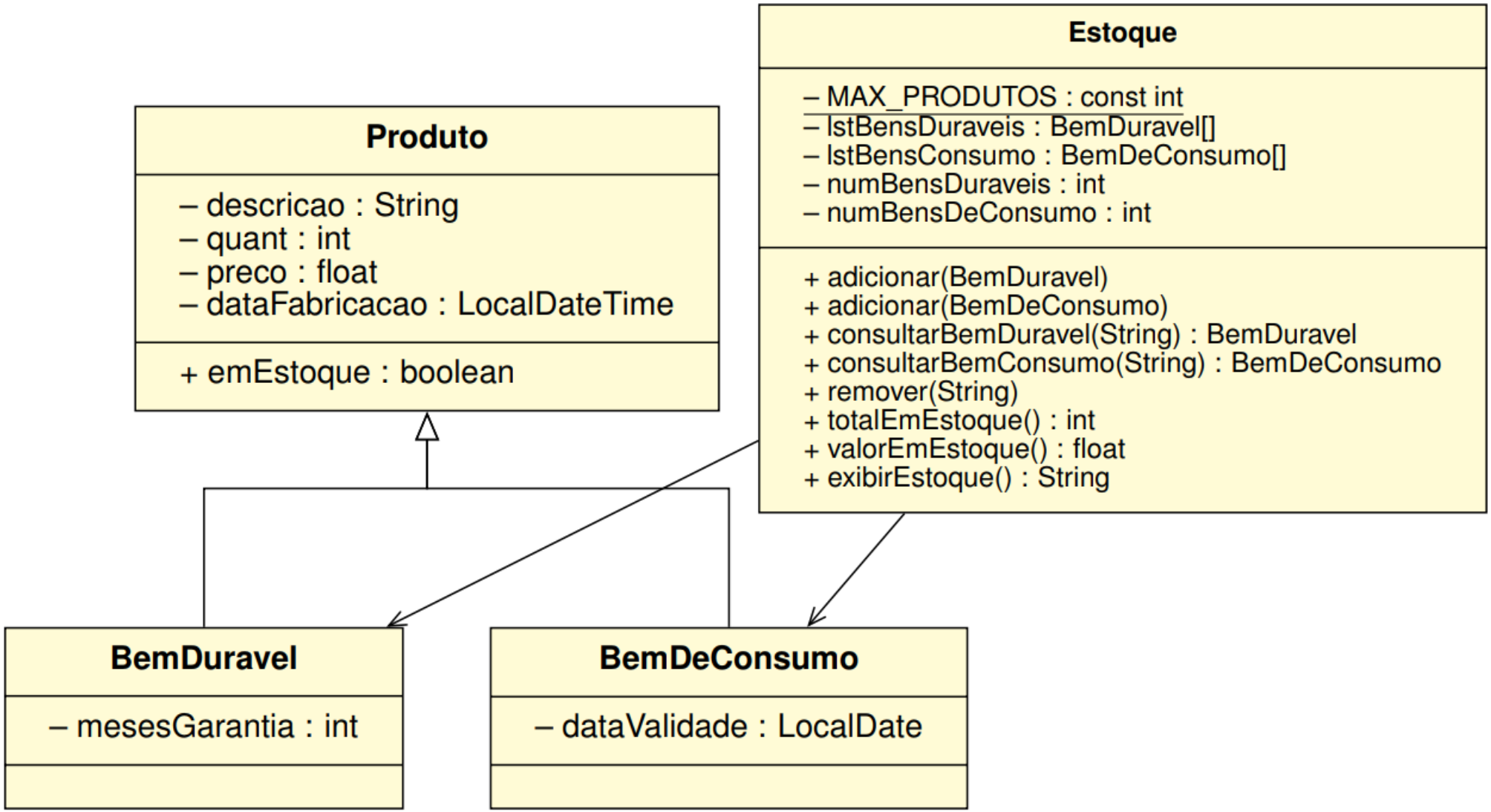
Exemplo de herança simples



Construtores em classe estendidas

- Nova classe deve escolher qual construtor da superclasse a chamar.
- Em um construtor da subclasse pode-se chamar diretamente o construtor da superclasse: **super**().
- A assinatura do método **super**() deve ser a mesma assinatura do construtor que se deseja chamar.
- Se não especificar construtor executa-se chamada ao construtor padrão da classe base (se houver).

Exemplo: herança de Produto



Exemplo: Bem de Consumo

```
public class BemDeConsumo extends Produto {
    private LocalDate dataValidade;

    public LocalDate getDataValidade() {  return dataValidade;  }
    public void setDataValidade(LocalDate dataValidade) {
        // a data de fabricação deve ser anterior à data de validade.
        if (getDataFabricacao().isBefore(dataValidade.atStartOfDay()))
            this.dataValidade = dataValidade;
    }
    public BemDeConsumo() {
        super();
        // o default é uma validade de 6 meses.
        dataValidade = LocalDate.now().plusMonths(6);
    }
    public BemDeConsumo(String d, float p, int q,
                        LocalDateTime f, LocalDate v) {
        super(d, p, q, f);
        setDataValidade(v);
    }
}
```

Exemplo: Bem Durável

```
public class BemDuravel extends Produto {
    private int mesesGarantia;

    public int getMesesGarantia() { return mesesGarantia; }
    public void setMesesGarantia(int mesesGarantia) {
        if (mesesGarantia > 0)
            this.mesesGarantia = mesesGarantia;
    }
    public BemDuravel() {
        super();
        // o valor default é garantia de 6 meses.
        mesesGarantia = 6;
    }
    public BemDuravel(String d, float p, int q,
                      LocalDateTime f, int g) {
        super(d, p, q, f);
        setMesesGarantia(g);
    }
}
```

Resumo

- Na aula de Programação Orientada a Objetos (POO), abordamos os conceitos fundamentais dessa abordagem de desenvolvimento de software. Aqui está um resumo dos principais tópicos discutidos:

1. Definições:
   POO é um paradigma de programação que se baseia na ideia de modelar o mundo real por meio de objetos, que podem conter dados na forma de atributos e comportamentos na forma de métodos.
2. Classe / Objeto:
   Uma classe é um modelo ou um molde para criar objetos. Ela define os atributos e métodos que os objetos terão. Um objeto é uma instância de uma classe, ou seja, é uma representação concreta do conceito definido pela classe.
3. Construtores:
   Construtores são métodos especiais usados para inicializar objetos quando são criados. Eles permitem a definição de como um objeto deve ser configurado inicialmente.
4. Encapsulamento:
   Encapsulamento é um conceito que preconiza esconder os detalhes de implementação de um objeto e fornecer uma interface clara e consistente para interagir com ele. Isso é alcançado por meio do uso de modificadores de acesso, como public, private e protected, que controlam a visibilidade dos membros de uma classe.
5. Herança:
   Herança é um mecanismo que permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse). Isso promove a reutilização de código e facilita a criação de hierarquias de classes.

- Esses conceitos são fundamentais para compreender e aplicar a POO de forma eficaz na construção de sistemas de software robustos e modularizados.

Próxima aula

Programação defensiva