

Programação Orientada a Objeto

André Luiz Forchesatto
andreforchesatto@gmail.com

Apresentação

- Especialista em Ciência da Computação pela UFSC;
- Graduado em Tecnologia em Informática pela Unoesc;
- Atuação
 - Sócio Camtwo Sistemas;
 - Desenvolvedor Java desde 2002;
 - Professor;
- Contato
 - **andreforchesatto@gmail.com**
 - @forchesatto

Agenda

- Revisão Básica: Abstrações, Classes, Métodos e atributos
- Encapsulamento
- Associações
- Herança
- Polimorfismo
- Princípios SOLID

Classes

- Um número de pessoas ou coisas agrupadas devido a certas semelhanças ou traços comuns.
- Uma descrição de um ou mais Objetos com um conjunto uniforme de atributos e serviços

Conta
Número Agência Cliente Data abertura Limite
sacar(valor) depositar(valor)



Nova Instância

Novos objetos

Conta1	Conta2	Conta3
Número: 123456 Agência: Chapecó Cliente: André Data abertura: 10/04/2008 Limite: 1000	Número: 45678 Agência: Xanxerê Cliente: Pedro Data abertura: 11/04/2009 Limite: 1300	Número: 9876 Agência: Xaxim Cliente: Cristiano Data abertura: 10/04/2010 Limite: 10500

Classe & Objeto no Java

- Deve-se utilizar a palavra reservada **class** para representar uma classe;
- Para criar um objeto deve-se utilizar o operador **new**; (**instanciação**)

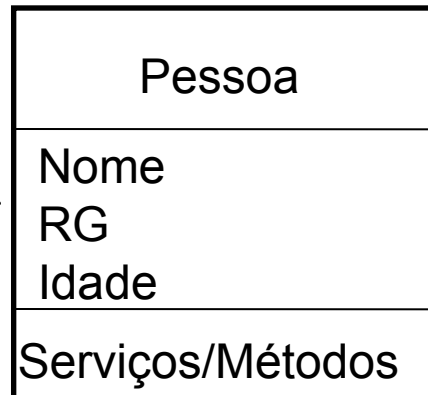
```
class Conta {  
  
}
```

```
...  
Conta conta = new Conta();  
...
```

Atributos

- Qualquer propriedade, qualidade ou característica que pode ser atribuída a uma pessoa ou coisa;
- Um atributo é um dado para o qual cada Objeto em uma Classe tem seu próprio valor;

Representação Atributo →



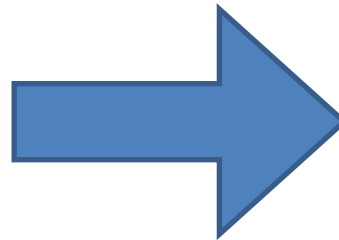
Atributos

Para um sistema bancário quais atributos para a classe conta?

Nome: EMPRESA.XYZ Agência/Conta: XXXXX / XXXXX-X
Data: Horário:

Extrato de Conta Corrente e Conta Investimento

Data	Lançamento	Valor (R\$)	Saldo (R\$)
01/01/2010	SALDO ANTERIOR		50.000,00
02/01/2010	PAGAMENTO FORNECEDORES	- 644,14	
02/01/2010	PAGAMENTO FORNECEDORES	- 720,00	
03/01/2010	PAGAMENTO FORNECEDORES	- 2.955,00	
03/01/2010	PAGAMENTO FORNECEDORES	- 11.090,18	
04/01/2010	PAGAMENTO FORNECEDORES	- 13.370,00	
04/01/2010	TARIFA BANCÁRIA	- 50,00	
05/01/2010	TARIFA BANCÁRIA	- 10,00	
05/01/2010	TARIFA BANCÁRIA	- 10,00	
06/01/2010	SALDO CONTA CORRENTE		21.150,68
06/01/2010	RECEBIMENTOS CLIENTES	2.525,00	
07/01/2010	RECEBIMENTOS CLIENTES	13.500,00	
07/01/2010	RECEBIMENTOS CLIENTES	1.520,00	
08/01/2010	TARIFA BANCÁRIA	- 10,00	
08/01/2010	SALDO CONTA CORRENTE		38.685,68
09/01/2010	PAGAMENTO FORNECEDORES	- 135,00	
09/01/2010	PAGAMENTO IMPOSTOS	- 1.348,00	
10/01/2010	PAGAMENTO FORNECEDORES	- 5.486,00	
10/01/2010	SALDO CONTA CORRENTE		31.716,68



Conta

Número
Agência
Cliente
Data abertura
Limite
Saldo

Exemplo

```
public class Main {  
    public static void main(String[] args) {  
        Conta conta = new Conta();  
        //Aponta para uma posição da memória  
        //Valores dos atributos  
        conta.numero = 1234;  
        conta.cliente = "André";  
        conta.limite = 1000.0;  
        conta.agencia = "0053-x";  
        conta.dataAbertura = new Date();  
    }  
}
```

```
class Conta {  
    //Atributos da classe  
    Integer numero;  
    String agencia;  
    String cliente;  
    Date dataAbertura;  
    Double limite;  
}
```

Métodos

- As coisas que um objeto pode fazer se chama **Métodos**;
- Toda ação que o objeto sabe executar;
- Pode modificar o estado de um objeto;
- Um método pode receber **N** parâmetros e retornar um único valor;

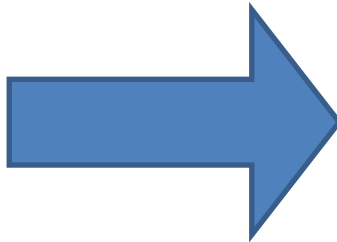
Métodos

Para um sistema bancário quais métodos para a classe conta?

Nome: EMPRESA XYZ Agência/Conta: XXXXX / XXXXX-X
Data: Horário:

Extrato de Conta Corrente e Conta Investimento

Data	Lançamento	Valor (R\$)	Saldo (R\$)
01/01/2010	SALDO ANTERIOR		50.000,00
02/01/2010	PAGAMENTO FORNECEDORES	- 644,14	
02/01/2010	PAGAMENTO FORNECEDORES	- 720,00	
03/01/2010	PAGAMENTO FORNECEDORES	- 2.955,00	
03/01/2010	PAGAMENTO FORNECEDORES	- 11.090,18	
04/01/2010	PAGAMENTO FORNECEDORES	- 13.370,00	
04/01/2010	TARIFA BANCÁRIA	- 50,00	
05/01/2010	TARIFA BANCÁRIA	- 10,00	
05/01/2010	TARIFA BANCÁRIA	- 10,00	
06/01/2010	SALDO CONTA CORRENTE		21.150,68
06/01/2010	RECEBIMENTOS CLIENTES	2.525,00	
07/01/2010	RECEBIMENTOS CLIENTES	13.500,00	
07/01/2010	RECEBIMENTOS CLIENTES	1.520,00	
08/01/2010	TARIFA BANCÁRIA	- 10,00	
08/01/2010	SALDO CONTA CORRENTE		38.685,68
09/01/2010	PAGAMENTO FORNECEDORES	- 135,00	
09/01/2010	PAGAMENTO IMPOSTOS	- 1.348,00	
10/01/2010	PAGAMENTO FORNECEDORES	- 5.486,00	
10/01/2010	SALDO CONTA CORRENTE		31.716,68



Conta

...

saca()
deposita()
verificaSaldo()
transfereParaOutraConta()
alteraLimite()

Métodos - Exemplo

```
class Conta {  
    ...  
    void deposita(Double valor) {  
        saldo += valor;  
    }  
  
    Double verificaSaldo() {  
        return saldo;  
    }  
    ...  
}
```

Encapsulamento



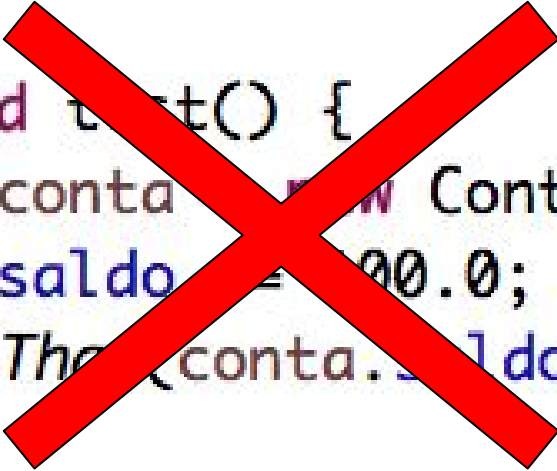
Encapsulamento

- Vem do conceito de encapsular, esconder, proteger;
- É uma maneira de expor somente o que interessa ao consumidor da classe;
- Proteger o acesso a atributos;
- Criar uma padronização de acesso a lógicas de negócio, escondendo a implementação.



Encapsulamento

```
@Test  
public void test() {  
    Conta conta = new Conta();  
    conta.saldo = 100.0;  
    assertEquals(conta.saldo, is(100.00));  
}
```



Encapsulamento

```
@Test
public void test() {
    Conta conta = new Conta();
    conta.depositar(100.0);
    assertThat(conta.getSaldo(), is(100.00));
}
```



```
3 public class Conta {
4
5     private Double saldo = 0.0;
6
7     public void depositar(Double valor) {
8         saldo += valor;
9     }
10
11     public Double getSaldo() {
12         return saldo;
13     }
14 }
```

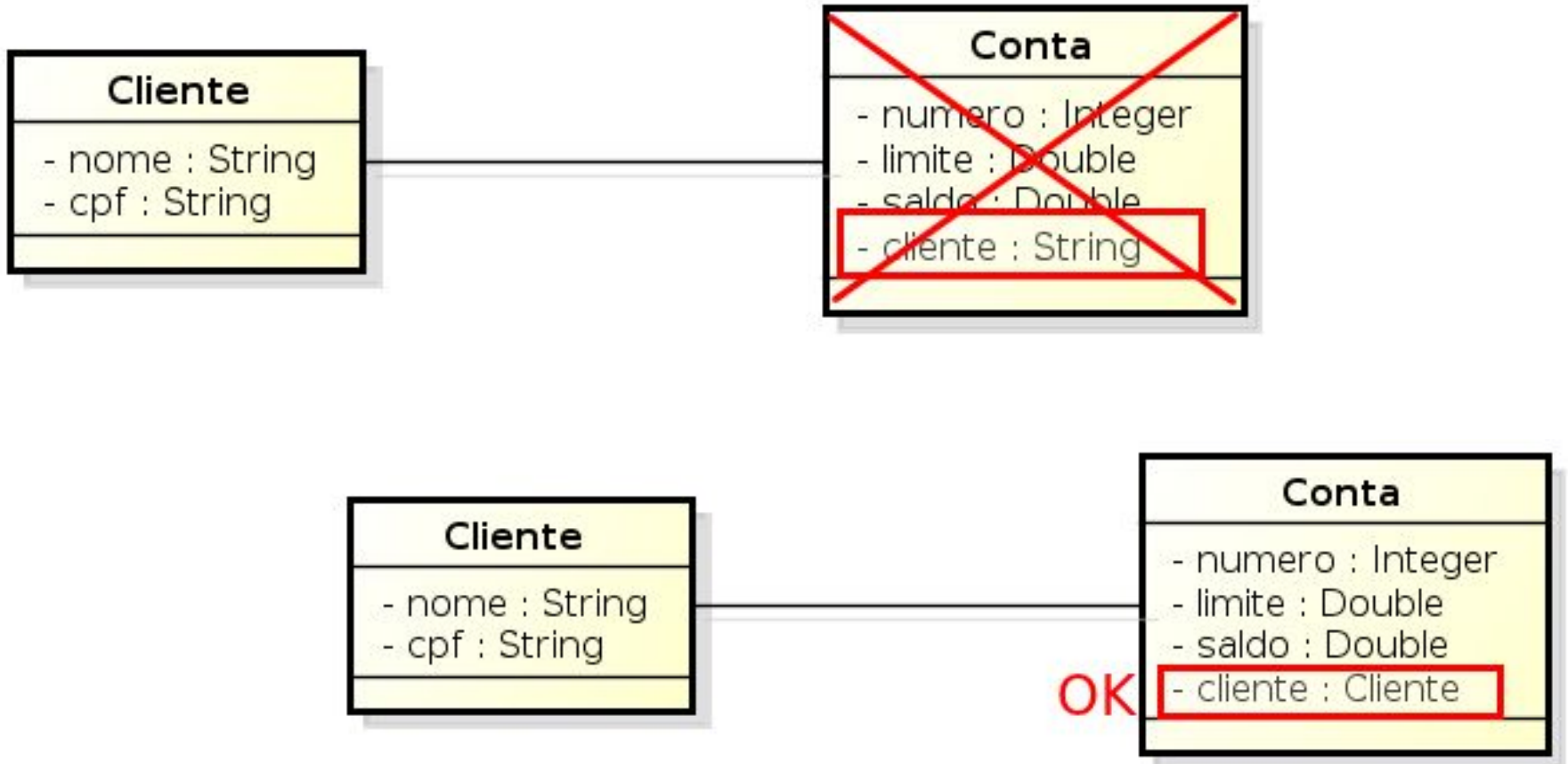

Encapsulamento

- Existem quatro maneiras de proteger o acesso a métodos e atributos em Java:
- **private** acesso somente dentro da mesma classe;
- **public** acesso por todas as classes do sistema;
- **protected** acesso somente na classe e suas filhas; (herança)
- **Sem qualificador** acesso somente dentro do mesmo pacote.

Associações

- Associação é o relacionamento entre duas classes através de atributos
- O atributo que representa a associação sempre deverá ser do tipo da classe associada
- Existem vários tipos de associações:
 - Unidirecional
 - Bidirecional
 - Composição
 - Agregação

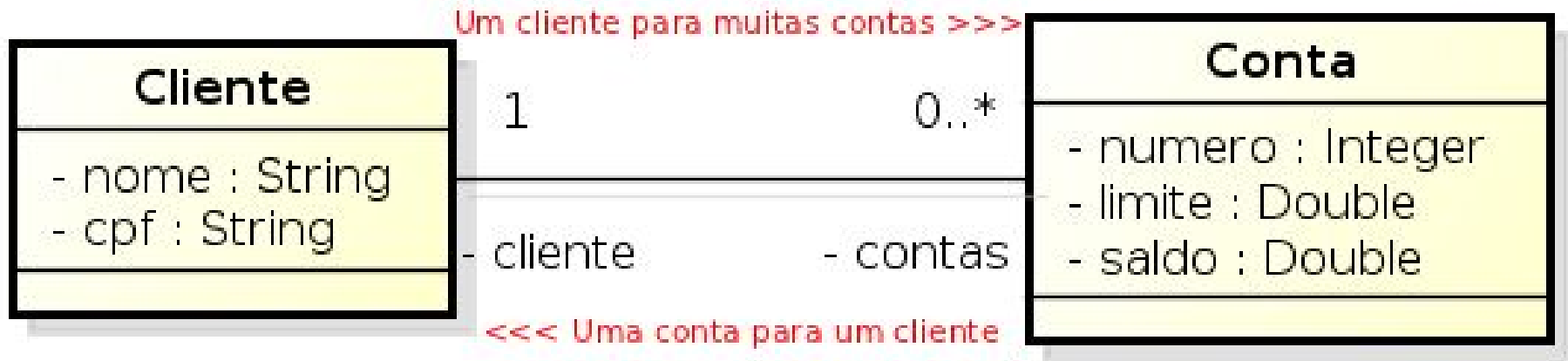
Associações



Cardinalidade associação

- Símbolos para representar associação:
 - Mais que um ou muitos: *
 - Exatamente um: 1
 - Zero ou mais: 0..*
 - Um ou mais: 1..*
 - Zero ou um: 0..1
 - Faixa especifica: 2..4

Cardinalidade associação



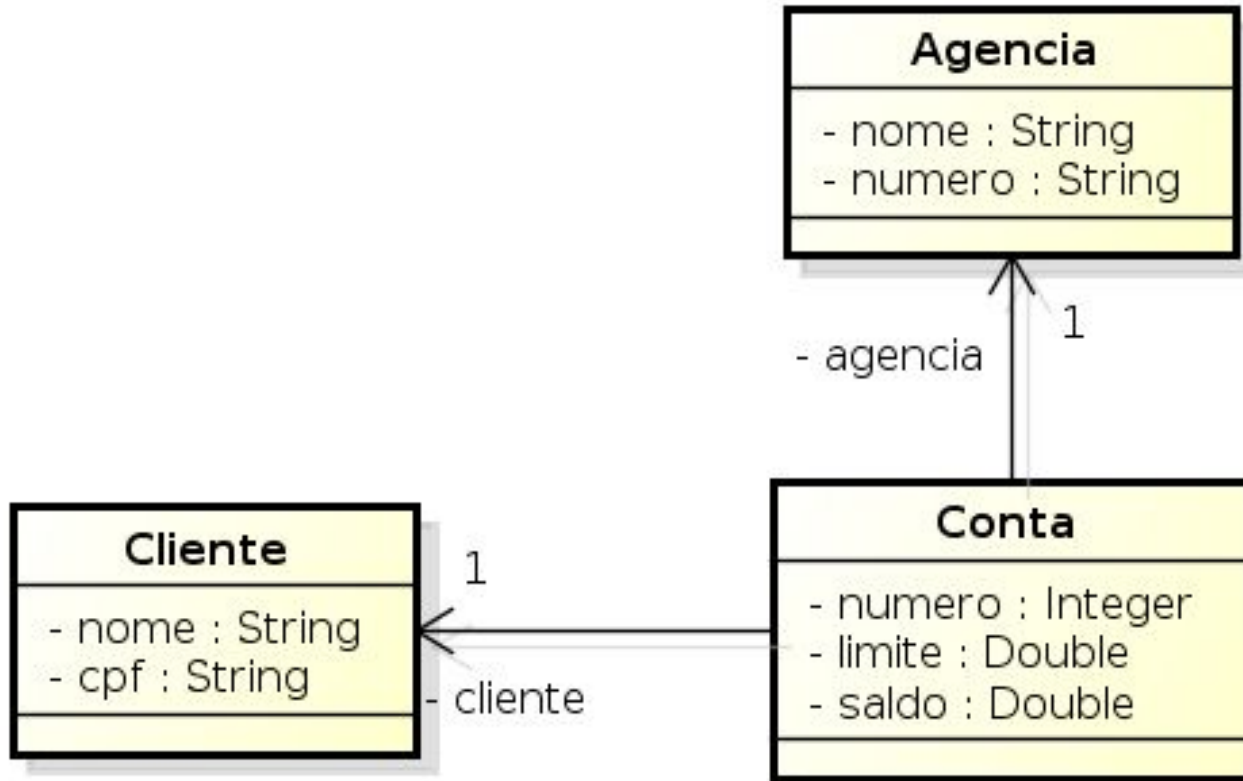
Na modelagem podemos omitir os atributos de relacionamento, pois já está subentendido que ele vai existir.

Associação unidirecional

- Quando indicamos a direção de uma associação
- Limitamos a possibilidade de navegação entre os objetos
- Indica que só teremos atributos de relacionamento em uma das classes



Exemplo UML



Exemplo Java

```
public class Agencia {  
  
    private String nome;  
    private String numero;
```

```
    public class Conta {  
  
        private Integer numero;
```

```
        private Agencia agencia; // Relacionamento com Agência
```

```
        private Cliente cliente; // Relacionamento com Cliente
```

```
public class Cliente {  
  
    private String nome;  
    private String cpf;
```

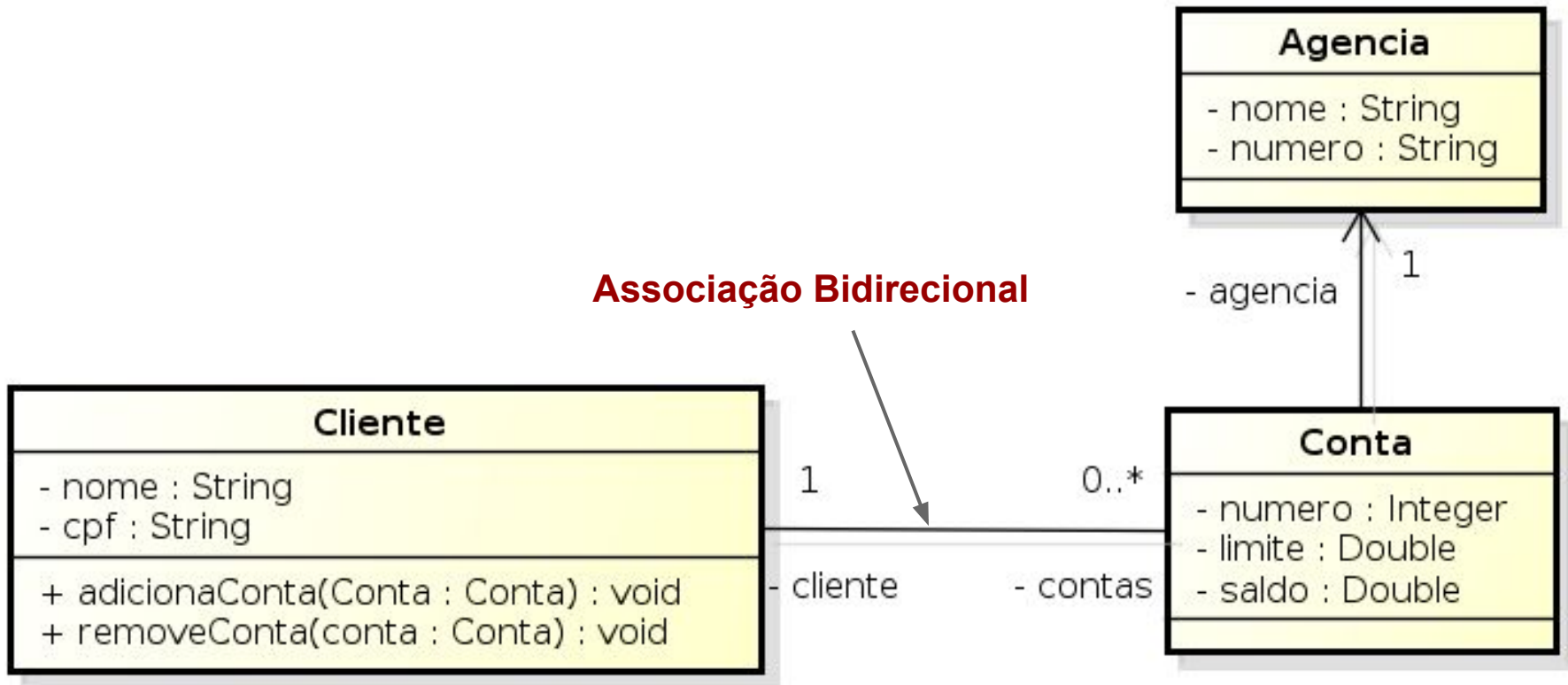

Exemplo Java - instanciando objetos

```
public static void main(String args[]) {  
  
    Cliente cliente = new Cliente("André", "98765");  
    Agencia agencia = new Agencia("Xanxerê", "8765-X");  
  
    Conta conta = new Conta(12345, agencia, cliente, new Date(), 1000.0);  
  
    System.out.println("Numero da conta:" + conta.getNumero());  
    System.out.println("Nome do cliente:" + conta.getCliente().getNome());  
    System.out.println("Nome da agência:" + conta.getAgencia().getNome());  
  
}
```

Associação bidirecional

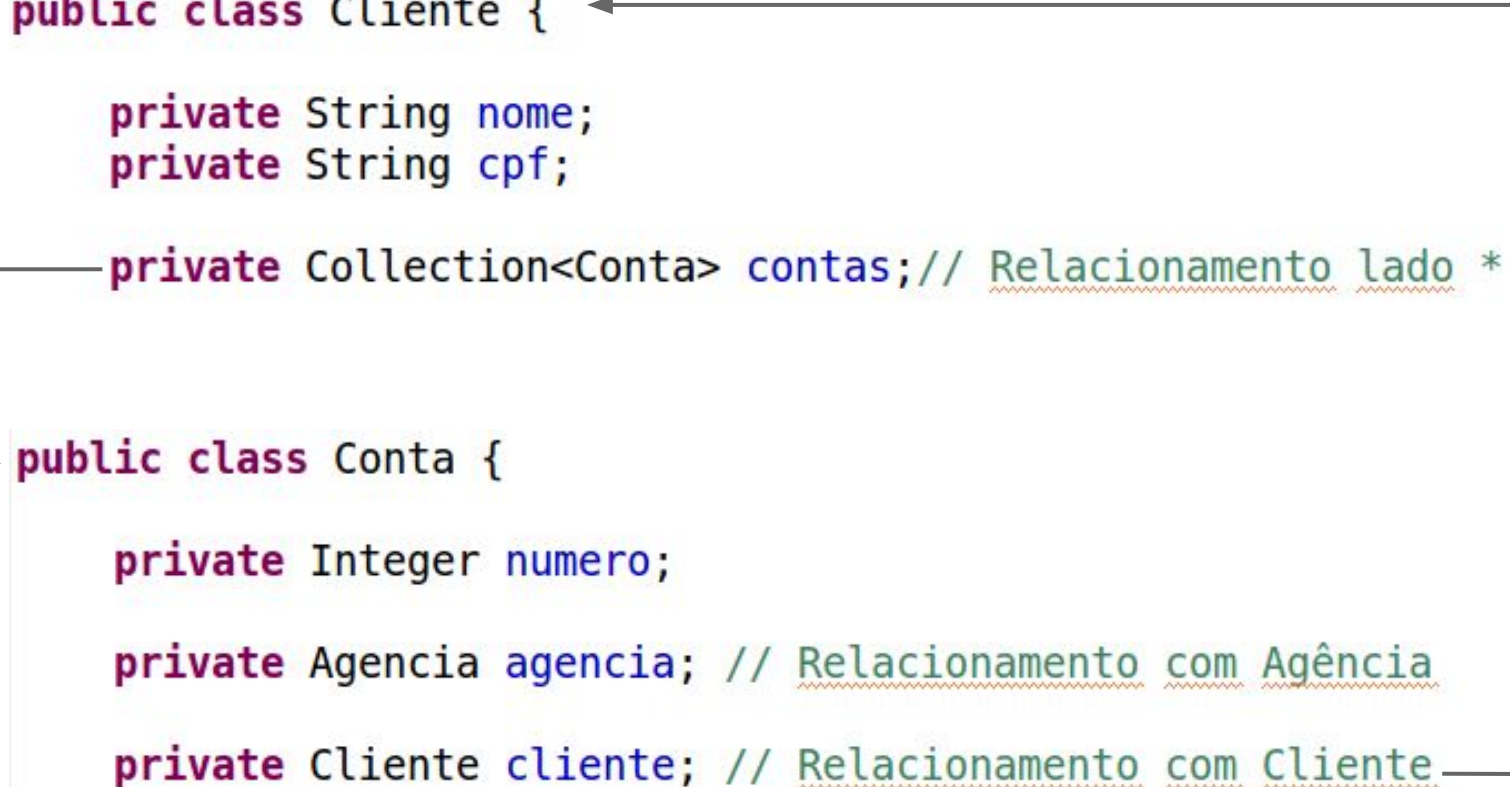
- Indica que é possível construir navegação nas duas direções da associação
- O lado *(muitos) da associação deve ser representado por uma **Collection**
- Deverá ser encapsulada a lógica de acesso a Collection, métodos como **adiciona** e **remove** devem ser criados

Exemplo UML



Exemplo Java

```
public class Cliente {  
    private String nome;  
    private String cpf;  
    private Collection<Conta> contas; // Relacionamento lado *  
}  
  
public class Conta {  
    private Integer numero;  
    private Agencia agencia; // Relacionamento com Agência  
    private Cliente cliente; // Relacionamento com Cliente
```



Exemplo Java

Métodos para encapsular o acesso a lista de contas da classe cliente.

```
public void adicionaConta(Conta conta) {  
    if (contas == null) {  
        contas = new ArrayList<Conta>();  
    }  
    contas.add(conta);  
}  
  
public void removeConta(Conta conta) {  
    if (contas != null && !contas.isEmpty()) {  
        contas.remove(conta);  
    }  
}
```

É necessário ter
setContas?

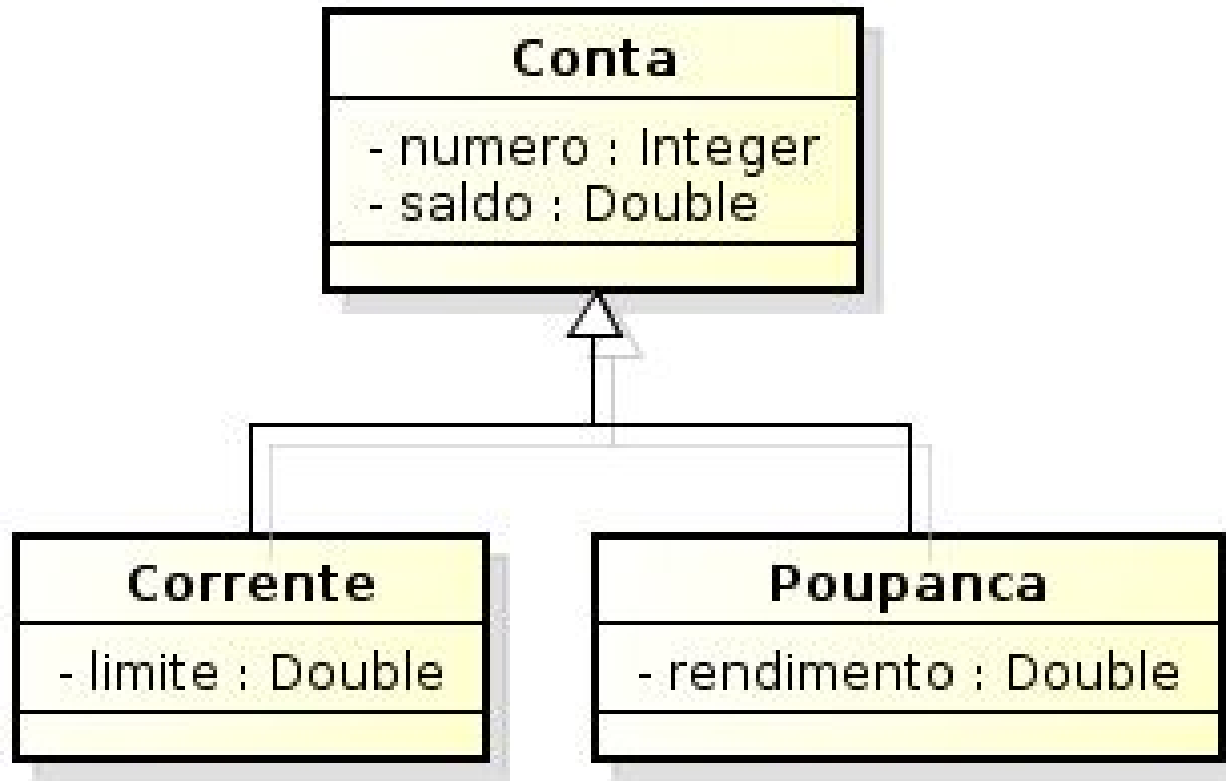
Prática

- Criar a classe Cliente;
- Criar a classe Agencia;
- Ajustar a classe Conta conforme modelo anterior;
- Ajustar testes existentes;
- Criar testes unitários para validar a lógica de adicionar contas ao cliente.

Herança

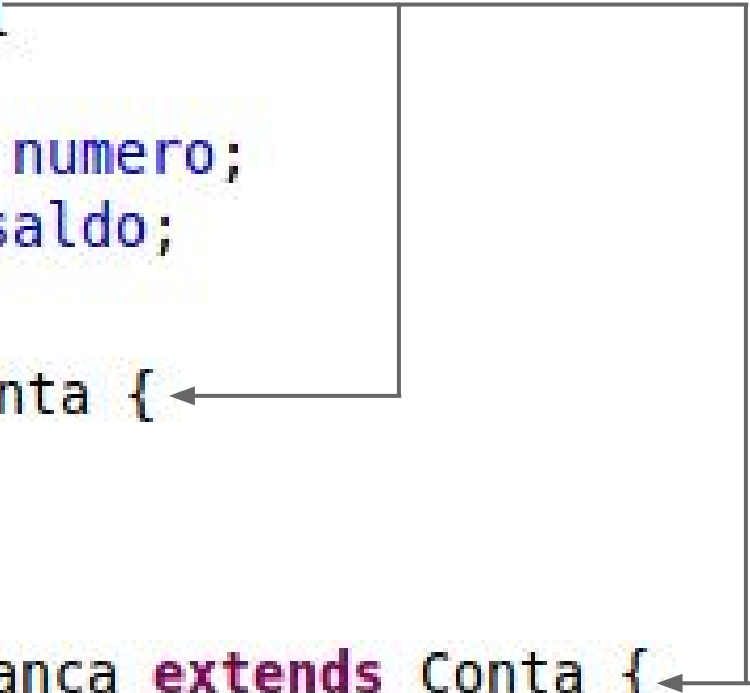
- É um mecanismo de hierarquia entre classes, onde uma classe mais especializada (filha) herda as propriedades da classe mais geral (pai)
- A classe mais geral é denominada **superclasse** e a classe mais especializada é chamada **subclasse**.
- Atributos e métodos comuns são definidos no nível mais alto da hierarquia (pai)
- A classe filha deve ter, no mínimo, uma propriedade de distinção em relação à classe pai

Exemplo Herança em UML



Exemplo Herança em Java

```
public class Conta {  
    private Integer numero;  
    private Double saldo;  
  
public class Corrente extends Conta {  
    private Double limite;  
  
    public class Poupanca extends Conta {  
        private Double rendimento;
```



```
}
```

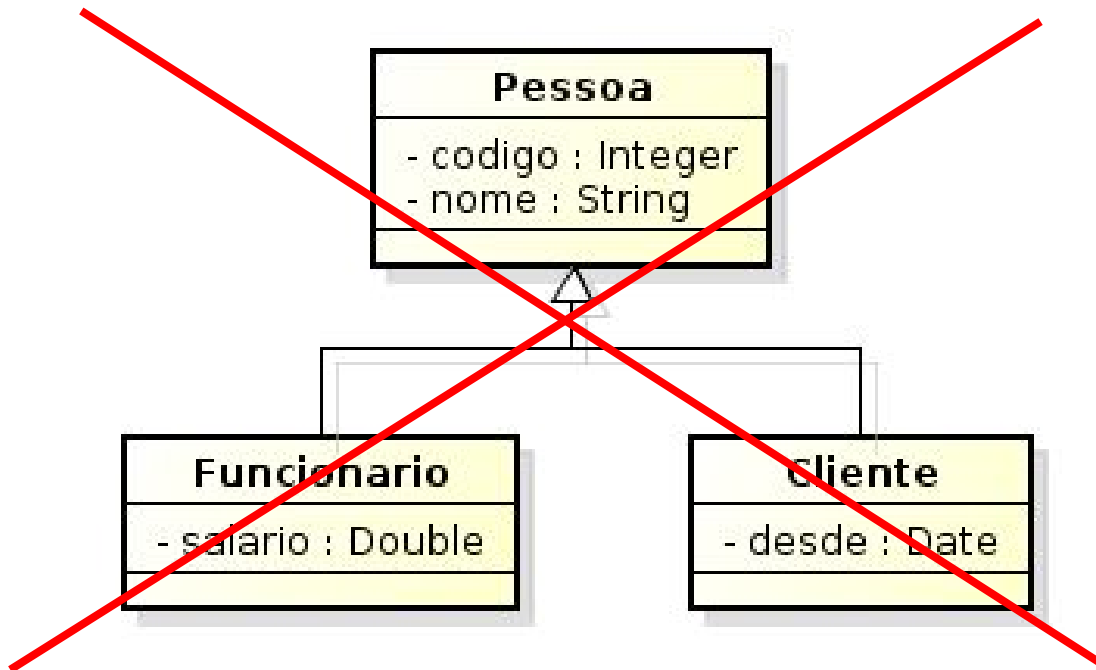
The diagram illustrates the inheritance structure. A horizontal line connects the opening curly brace of the `Conta` class to the opening curly brace of the `Corrente` class, with a downward arrow pointing to `Corrente`. Another horizontal line connects the opening curly brace of the `Conta` class to the opening curly brace of the `Poupanca` class, with a downward arrow pointing to `Poupanca`. This indicates that both `Corrente` and `Poupanca` inherit from `Conta`.

Exemplo Herança

```
Poupanca c1 = new Poupanca();  
c1.setNumero(1); // atributo da Classe Conta (Pai)  
c1.setRendimento(150.00);
```

```
Corrente c2 = new Corrente();  
c2.setNumero(2); // atributo da Classe Conta (Pai)  
c2.setLimite(500.00);
```

Exemplo onde **não** usar Herança



Sempre que possível favoreça a associação ou composição no lugar da herança.

Classes Abstratas

- Classes que não possuem instâncias
- Exemplo:

```
public abstract class OlaMundo {  
    . . .  
}
```

new OlaMundo(); ERRO!!!!

Métodos Abstratos

- Só possuem a assinatura
- Declarados em classes abstratas (pai)
- Sua implementação é feito pelas classes filhas
- Pode ser definido com o modificador **protected**
- Exemplo:

```
public abstract int calcular(int x, int t);
```

Polimorfismo

- Do grego "muitas formas", ou seja, um único nome representando um código diferente, selecionado por algum mecanismo automático

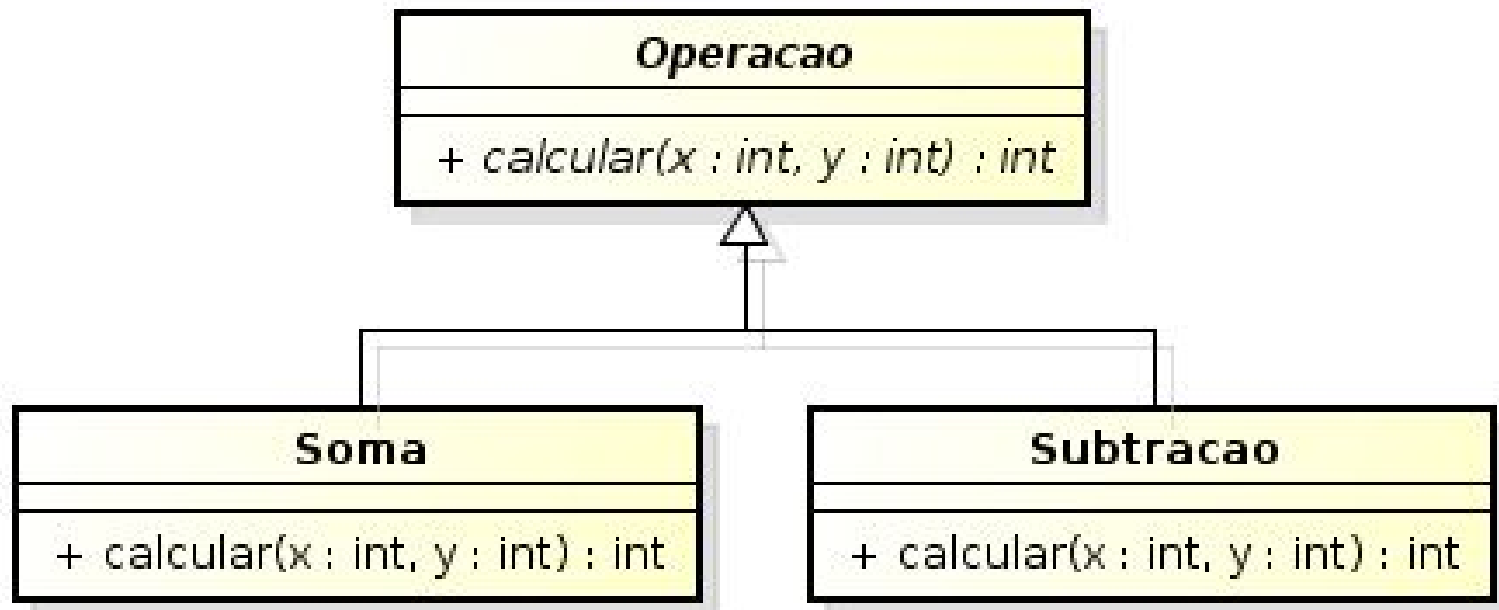
“Um nome, vários comportamentos”

- Permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam

Exemplos Polimorfismo

- Ontem sai para **dançar** com uns amigos, mas acabamos **dançando** porque não conseguimos encontrar um lugar que nos agradasse
- José **cantou** a noite inteira no Karaoke e João **cantou** a noite inteira a namorada de José

Exemplo Polimorfismo em UML



Exemplo Polimorfismo em Java

```
public abstract class Operacao {  
  
    public abstract int calcular(int x, int y);  
  
}
```

```
public class Soma extends Operacao {  
  
    @Override  
    public int calcular(int x, int y) {  
        return x + y;  
    }  
  
}
```

```
public class Subtracao extends Operacao {  
  
    @Override  
    public int calcular(int x, int y) {  
        return x - y;  
    }  
  
}
```

Exemplo Polimorfismo em Java

```
public class UsoPolimorfismo {  
  
    public static void mostrarCalculo(Operacao operacao, int x, int y) {  
        System.out.println("O resultado é: " + operacao.calcular(x, y));  
    }  
  
    public static void main(String args[]) {  
        // Calculo da soma  
        UsoPolimorfismo.mostrarCalculo(new Soma(), 10, 10);  
  
        // Calculo da subtração  
        UsoPolimorfismo.mostrarCalculo(new Subtracao(), 10, 5);  
    }  
}
```

Com o uso do polimorfismo podemos deixar nosso código mais fácil de ser evoluído. Neste caso não vamos precisar de *if's* para imprimir o resultado da conta.

Sobrescrita

- Consiste em termos a mesma assinatura de método tanto na classe Pai quanto na classe Filha
- Utilizada para definir o comportamento de um método de mesmo nome já definido na superclasse
- A visibilidade do método que sobrescreve não pode ser mais restritiva do que o método sobrescrito
- Podemos dizer que o método **calcular()** do exemplo anterior sofre **sobrescrita**

Exemplo Sobrescrita

```
public class Conta {
```

```
// ...
```

```
private Double saldo;
```

```
public Double verificaSaldo() {  
    return saldo;  
}
```

```
// ...
```

```
public class Corrente extends Conta {
```

```
// ..
```

```
private Double limite;
```

```
@Override
```

```
public Double verificaSaldo() {  
    return super.verificaSaldo() + limite;  
}
```

```
// ..
```

```
}
```

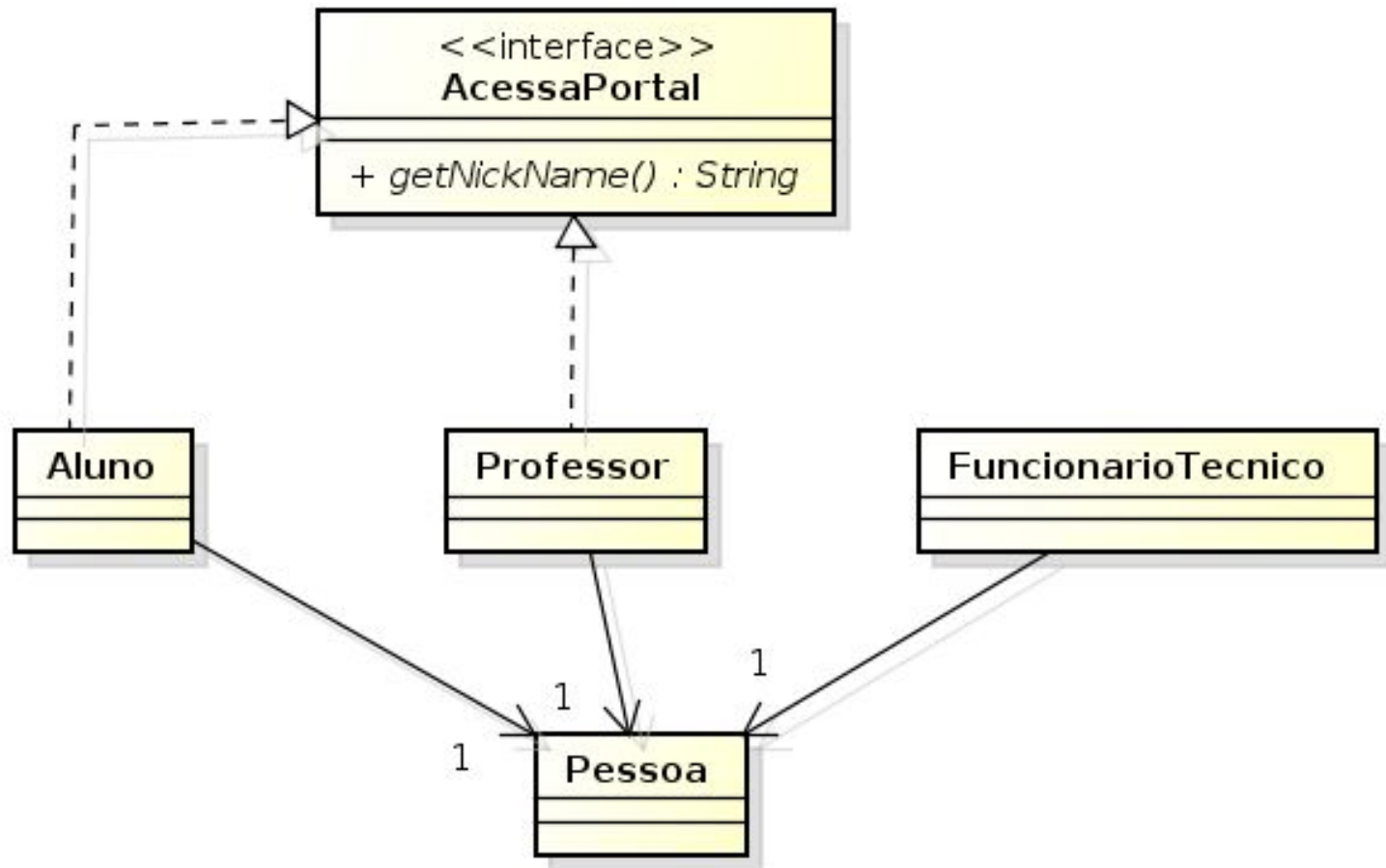
Interface

- Contrato firmado de implementação;
- Conjunto de métodos relacionados que não possuem implementação;
- Cada método pode ser considerado uma cláusula do contrato;
- Todos os métodos são abstratos e públicos por padrão;
- Todo atributo é considerado uma constante *static* e *final*;
- Diminui o acoplamento entre as classes;

Interface

- Informa **o que o objeto deve fazer** e não como ele faz ou o que ele tem;
- As classes concretas que irão "assinar" o contrato serão obrigadas a implementar todos os métodos da interface;
- Uma interface **pode herdar de mais de uma interface**;
- Duas regras de ouro: (Livro Design Patterns)
 - "Evite herança, prefira composição"
 - "Programe voltado a interface e não a implementação"

Interface - Exemplo



Interface - Java

```
public interface AcessoPortal {  
  
    String getNickName();  
  
}
```

```
public class Professor  
    implements AcessoPortal {  
  
    private String nickName;  
  
    public Professor(String nickName) {  
        this.nickName = nickName;  
    }  
  
    @Override  
    public String getNickName() {  
        return nickName;  
    }  
  
}
```

```
public class Aluno  
    implements AcessoPortal {  
  
    private String nickName;  
  
    public Aluno(String nickName) {  
        this.nickName = nickName;  
    }  
  
    @Override  
    public String getNickName() {  
        return nickName;  
    }  
  
}
```

Interface - Java

```
public class Portal {  
  
    public static void main(String[] args) {  
        Portal portal = new Portal();  
        Professor professor = new Professor("prof. André");  
        portal.acessa(professor);  
        Aluno aluno = new Aluno("Colorado");  
        portal.acessa(aluno);  
  
    }  
  
    public void acessa(AcessaPortal acessaPortal) {  
        System.out.println("Bem vindo: " + acessaPortal.getNickName());  
    }  
  
}
```

Princípios SOLID

- **S**ingle Responsibility Principle (SRP), ou, Princípio da Responsabilidade Única.
- **O**pen Closed Principle (OCP), ou Princípio do Aberto Fechado.
- **L**iskov Substitution Principle (LSP), ou Princípio da Substituição de Liskov.
- **I**nterface Segregation Principle (ISP), ou Princípio da Segregação de Interfaces.
- **D**ependency Inversion Principle (DIP), ou Princípio da Inversão de Dependências.

Princípio da Responsabilidade Única

Esse princípio diz que as classes devem ser coesas, ou seja, terem uma única responsabilidade. Classes assim tendem a ser mais reutilizáveis, mais simples, e propagam menos mudanças para o resto do sistema.

Princípio do Aberto Fechado

Diz que as classes devem poder ter seu comportamento facilmente estendidas quando necessário, por meio de herança, interface e composição. Ao mesmo tempo, não deve ser necessário abrir a própria classe para realizar pequenas mudanças. No fim, o princípio diz que devemos ter boas abstrações espalhadas pelo sistema.

Princípio da Substituição de Liskov

Esse princípio diz que precisamos ter cuidado para usar herança. Herança é um mecanismo poderoso, mas deve ser usado com parcimônia, evitando os casos de Gato-estende-Cachorro, apenas por possuírem algo em comum.

Princípio da Segregação de Interfaces

Esse princípio diz que nossos módulos devem ser enxutos, ou seja, devem ter poucos comportamentos. Interfaces que tem muitos comportamentos geralmente acabam se espalhando por todo o sistema, dificultando manutenção.

Princípio da Inversão de Dependências

Esse princípio diz que devemos sempre depender de abstrações, afinal abstrações mudam menos e facilitam a mudança de comportamento e as futuras evoluções do código.

Trabalho Final