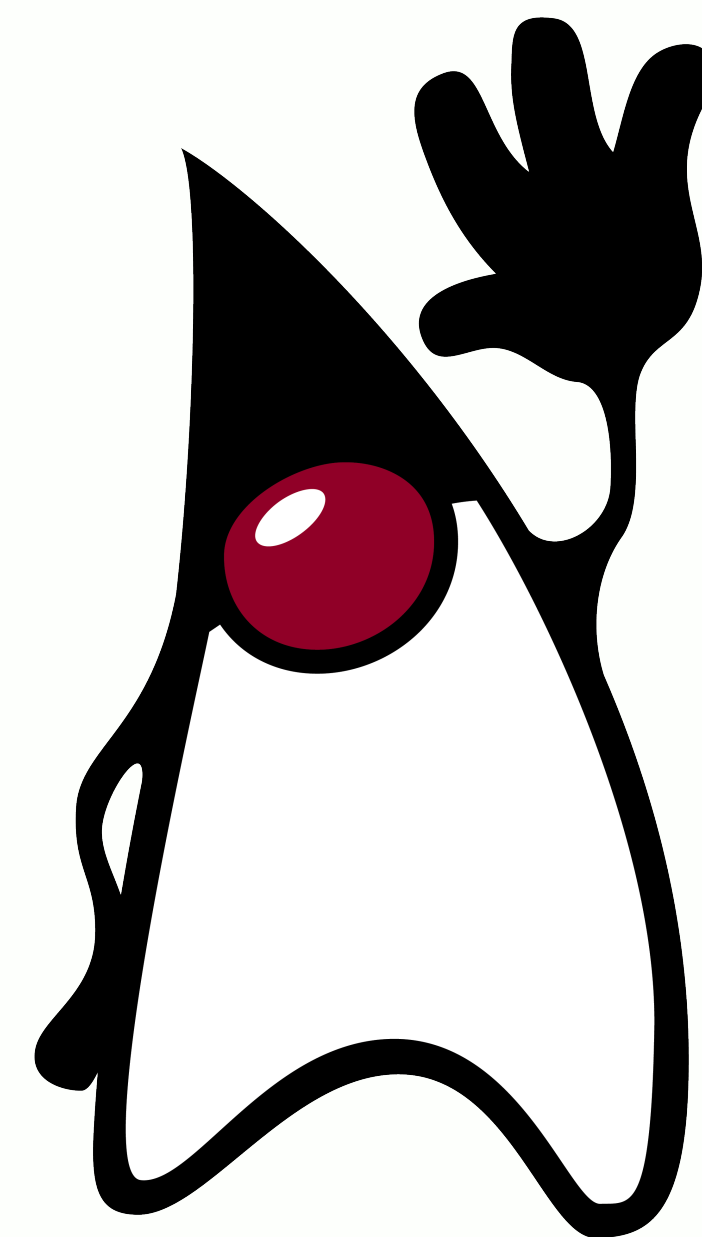




UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

# Herança

QXD0007 - Programação Orientada a Objetos



Prof. Bruno Góis Mateus ([brunomateus@ufc.br](mailto:brunomateus@ufc.br))

# Conteúdo

- Introdução
- Herança
- Sintaxe em Java
- Reescrita
- Verificação dinâmica de tipos

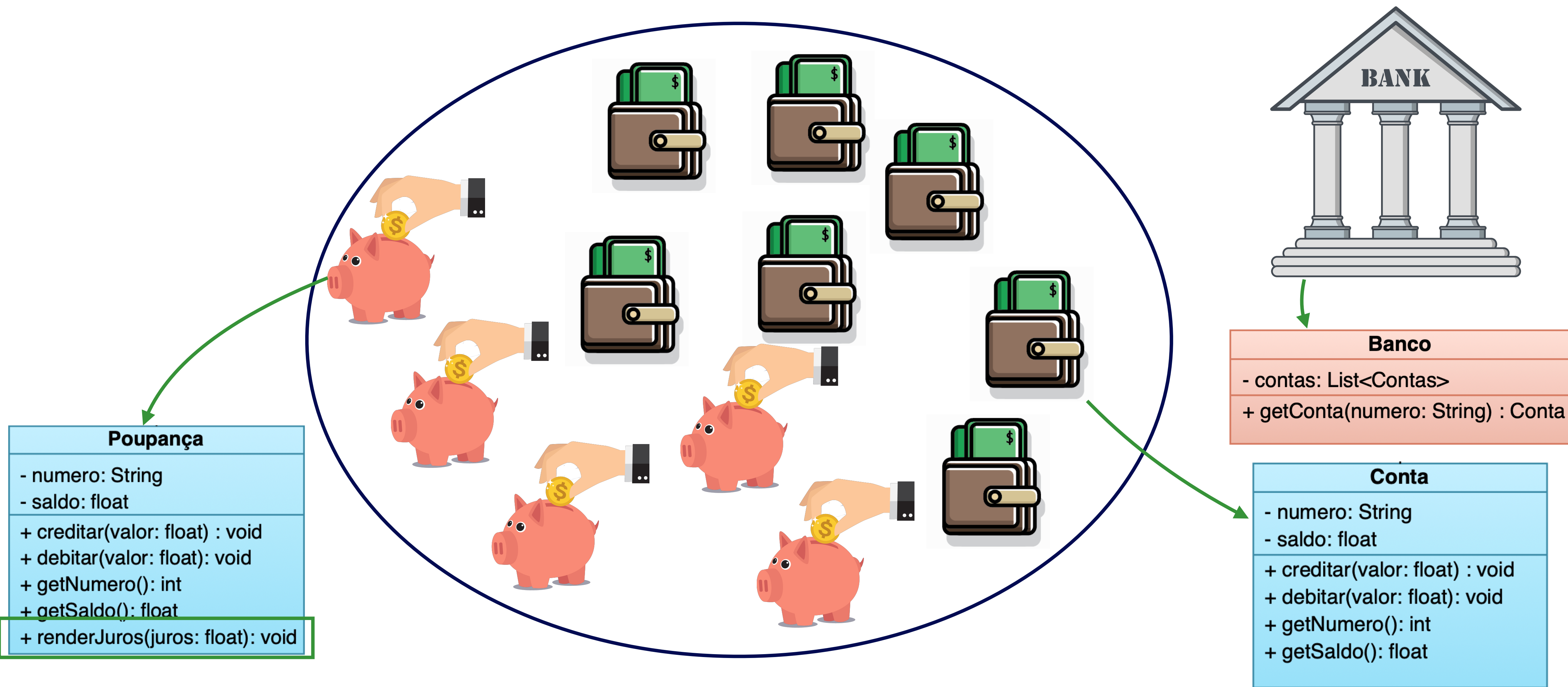
# Introdução



# Introdução

- Talvez a maior vantagem de utilizar programação orientada a objeto seja a promoção do **reuso**
- Podemos promovê-lo de várias formas:
  - Organizando classes a partir de pontos comuns entre elas
  - Criando hierarquia entre classes
- **Herança** é o mecanismo responsável por prover tal funcionalidade

# Introdução



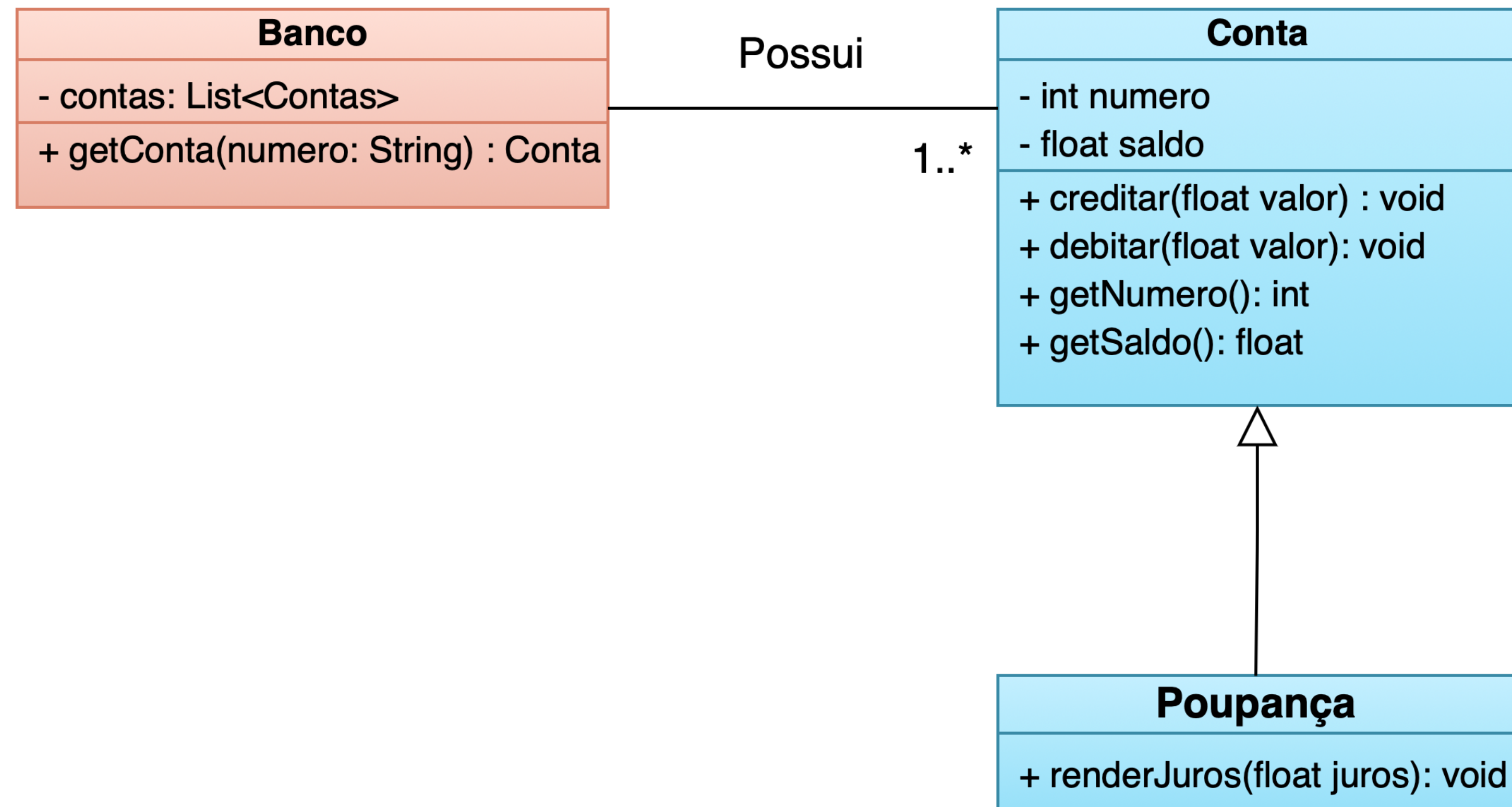
# Herança



# Herança

- Podemos relacionar **duas classes** de forma hierárquica
  - Criando uma relação de **mãe e filha**
    - **Superclasse** e **subclasse**
- A subclasse (filha) **herda características** da superclasse (mãe)

# Herança



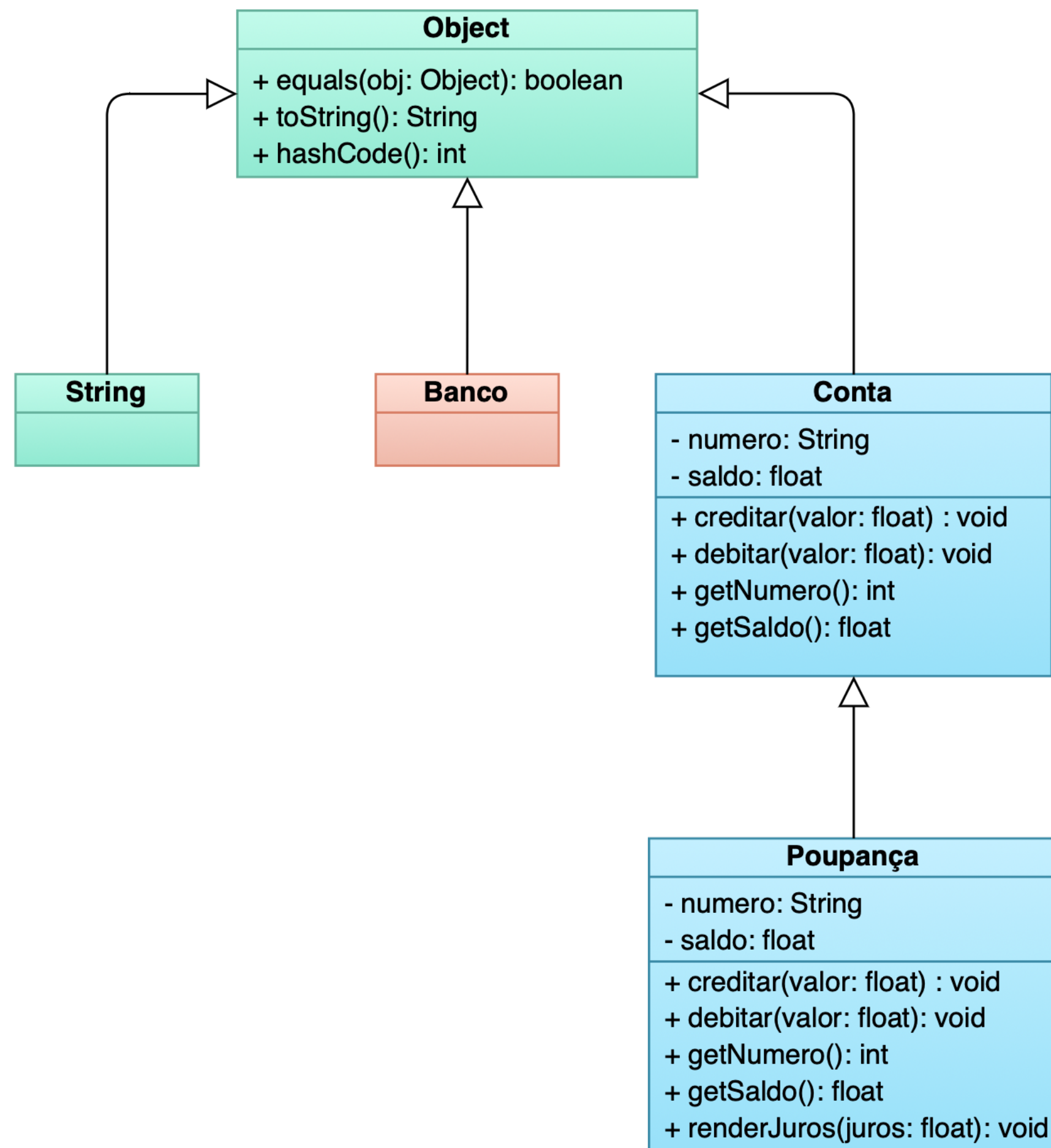
# Herança

## Impactos

- Reuso:
  - A descrição da superclasse pode ser usada para definir a subclasse
- Extensibilidade:
  - Algumas operações da superclasse podem ser redefinidas na subclasse
- Comportamento:
  - Objetos subclasse comportam-se como os objetos da superclasse
- Substituição (Polimorfismo):
  - Objetos da **subclasse** podem ser usados no lugar de objetos da **superclasse**

# Herança

## Em Java



- Todas as classes herdam da classe **Object**
- Implementa o mecanismo de **Herança Simples**
  - Só é possível herdar de **uma classe**
- Atributos e métodos **privados** são herdados, mas **não podem ser acessados diretamente**
- Construtores **não são herdados**

# Sintaxe em Java



# Sintaxe em Java

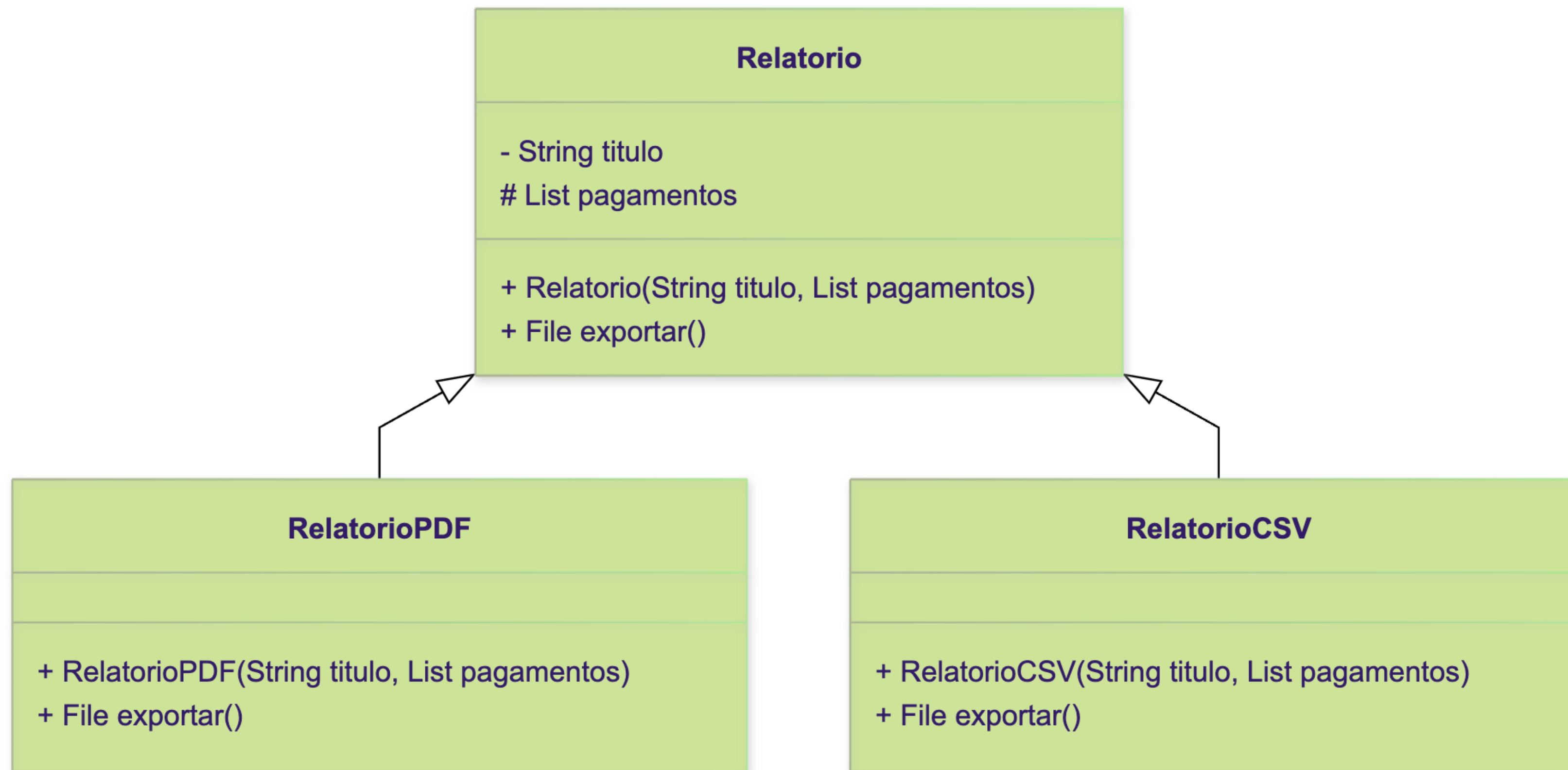
```
public class Conta {  
    private int numero;  
    protected float saldo;  
  
    public Conta(int numero) {  
        this.numero = numero  
    }  
  
    public void creditar(float valor) {}  
    public void debitar(float valor) {}  
    public int getNumero() {}  
    public float getSaldo() {}  
}
```

```
public class Poupanca extends Conta {  
    private float juros;  
  
    public Poupanca(int numero) {  
        super(numero)  
    }  
  
    public void renderJuros() {  
        saldo += juros  
    }  
}
```

# Reescrita



# Reescrita



# Reescrita

```
public class Relatorio {  
    private String titulo;  
    protected List<Pagamentos> pagamentos;  
  
    public Relatorio(String titulo,  
List<Pagamentos> pagamentos) {  
        this.titulo = titulo;  
        this.pagamentos = pagamentos;  
    }  
  
    public File exportar() {  
        // Gerar um arquivo .txt com os dados  
    }  
}
```

```
public class RelatorioCSV extends Relatorio {  
    public RelatorioCSV(String titulo,  
List<Pagamentos> pagamentos) {  
        super(titulo, pagamentos);  
    }  
  
    @Override  
    public File exportar() {  
        //Gera um arquivo .csv  
    }  
}
```

# Reescrita

- A **subclasse** redefine/sobrescreve **métodos** da superclasse com a mesma assinatura
  - Cada método tem **acesso** aos atributos da classe na qual foi definido
- O método da **subclasse** é chamado em tempo de execução
- Só é possível acessar o método da superclasse imediata via **super**

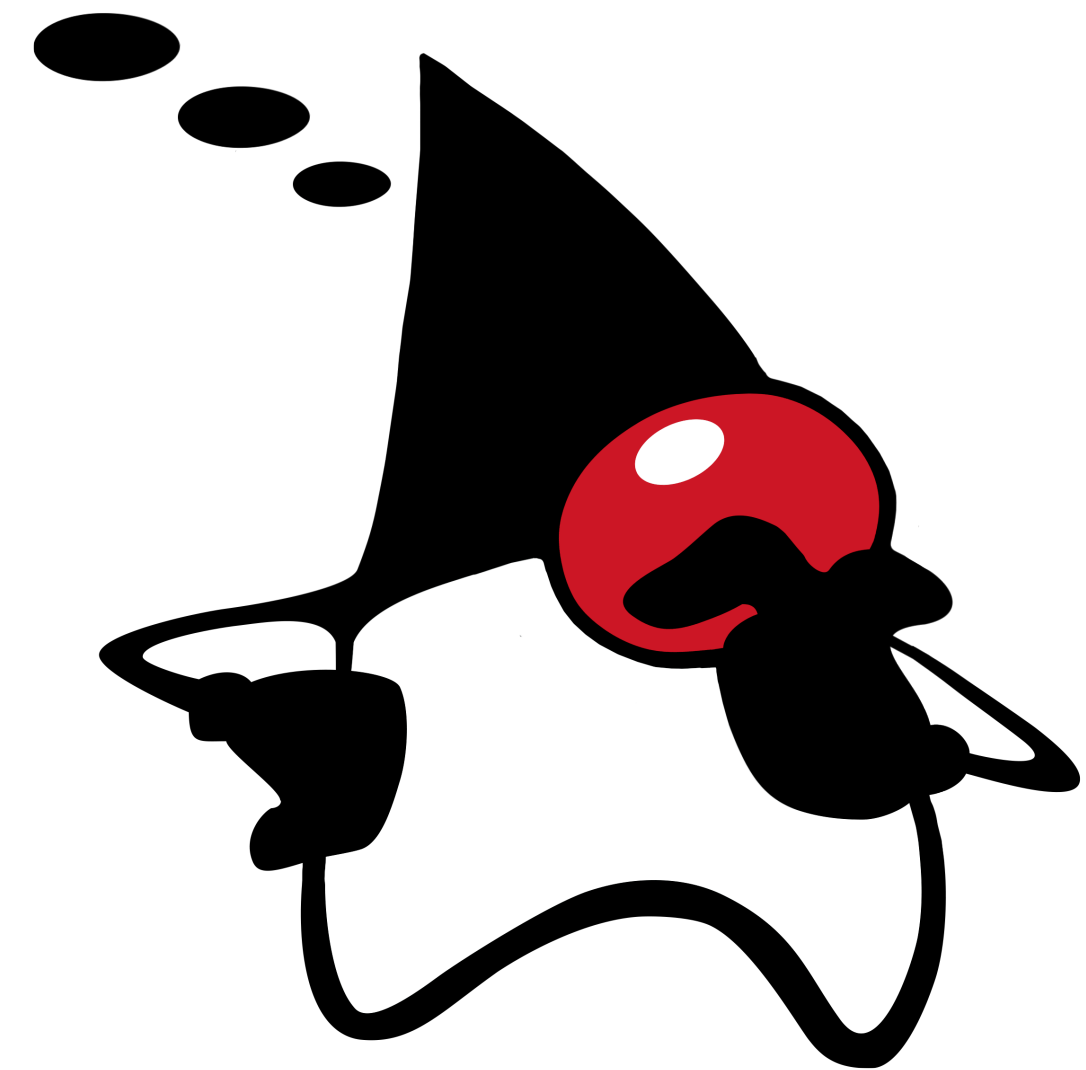
# Reescrita

## Ligação Tardia/Dinâmica de método

```
Conta c1 = new Conta(1);  
Poupanca p1 = new Poupanca(2);  
  
c1.creditar(100);  
p1.creditar(100); ?
```

- O código é escolhido dinamicamente (em **tempo de execução**), não estaticamente (em **tempo de compilação**)
- Escolha é feita com base no **tipo do objeto** associado à variável de acesso ao método

# Polimorfismo



# Polimorfismo

## Subclasses e subtipos

- Classes definem “tipos”
- Subclasses definem “subtipos”

Objetos das subclasses podem ser utilizados onde os objetos dos superclasse são requeridos

# Polimorfismo

## Variáveis Polimórficas

```
Conta c1 = new Conta(1);  
Poupanca p1 = new Poupanca(2);  
Conta c2 = new Poupanca(3);
```

```
c2.creditar(100);  
c1 = p1;  
c1.creditar(100);
```



- Elas podem armazenar objetos do tipo (classe) declarado ou dos subtipos (subclasses) do tipo (classe) declarado!
- Chamamos de polimorfismo de tipo

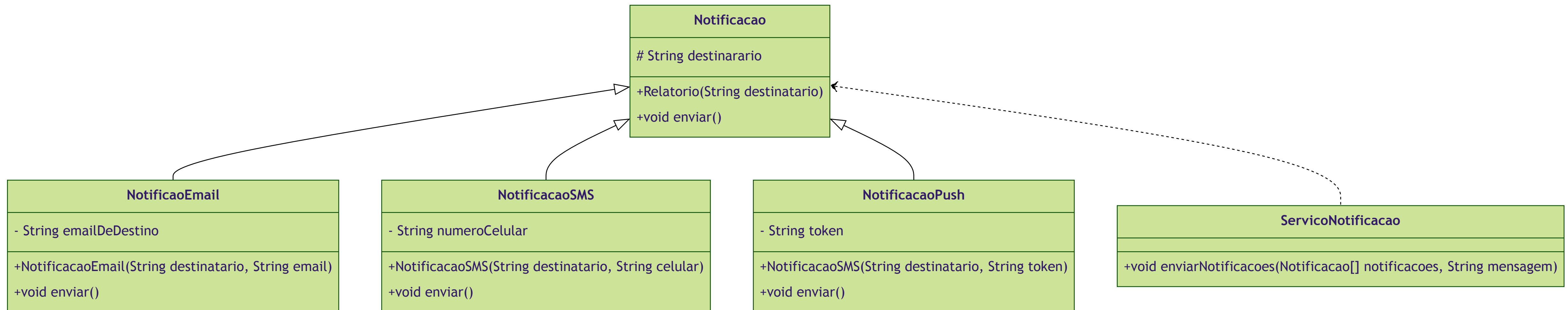
# Polimorfismo

## Verificação dinâmica de tipos

- Em casos onde há **substituição**, podemos forçar o compilador a “**enxergar**” o tipo dinâmico da variável
- A **coerção** facilita a verificação estática de tipos
  - Em tempo de execução, pode levar a erros de tipagem
- Podemos usar o **instanceof** para verificar o tipo dinâmico de uma variável antes de fazermos uma coerção
  - Retorna **false** quando a operação não casa ou **true** caso contrário

# Polimorfismo

## Verificação dinâmica de tipos



# Limitações e Cuidados

## Herança vs Composição

- Evitar **herança excessiva**:
  - Pode tornar o código difícil de entender e manter.
  - Quando usar? Apenas quando a relação "é um" (ex.: Carro é um Veículo).
- **Dependência forte**:
  - Subclasses dependem das mudanças na superclasse.
- Alternativas:
  - **Composição** (relação "tem um").

Por hoje é só

