Herança e Polimorfismo

Prof. Gustavo Wagner

(Alterações)

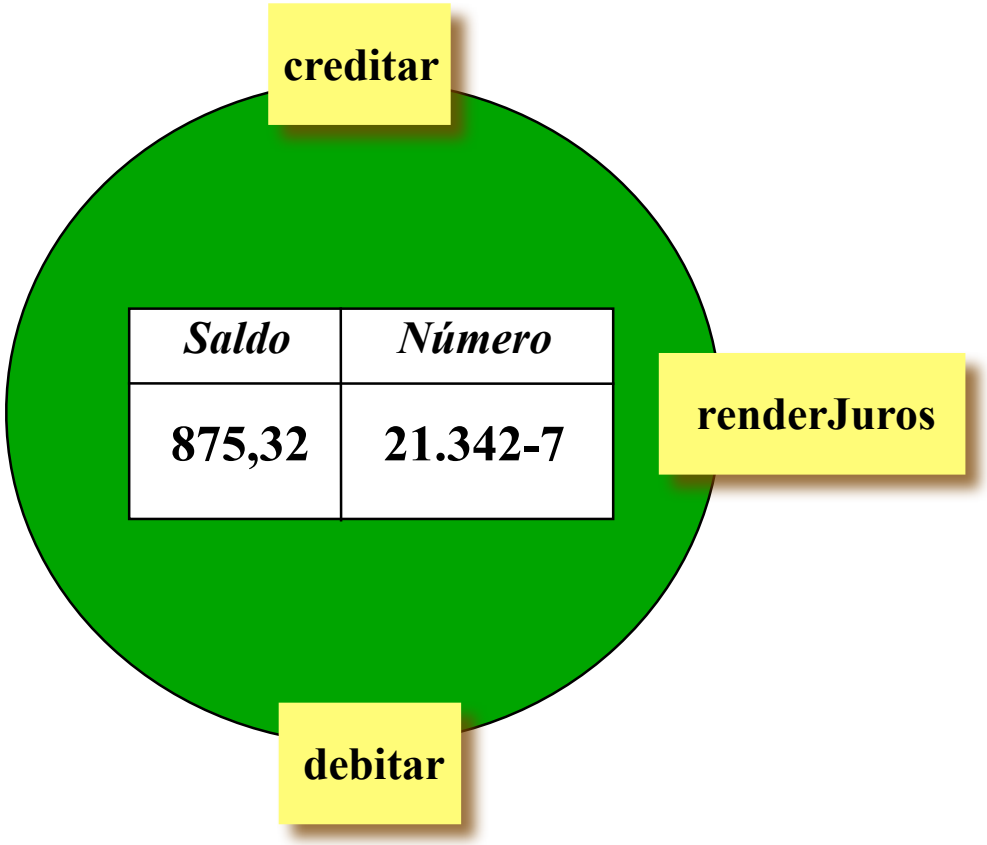
Prof. Tiago Massoni

(Slides Originais)

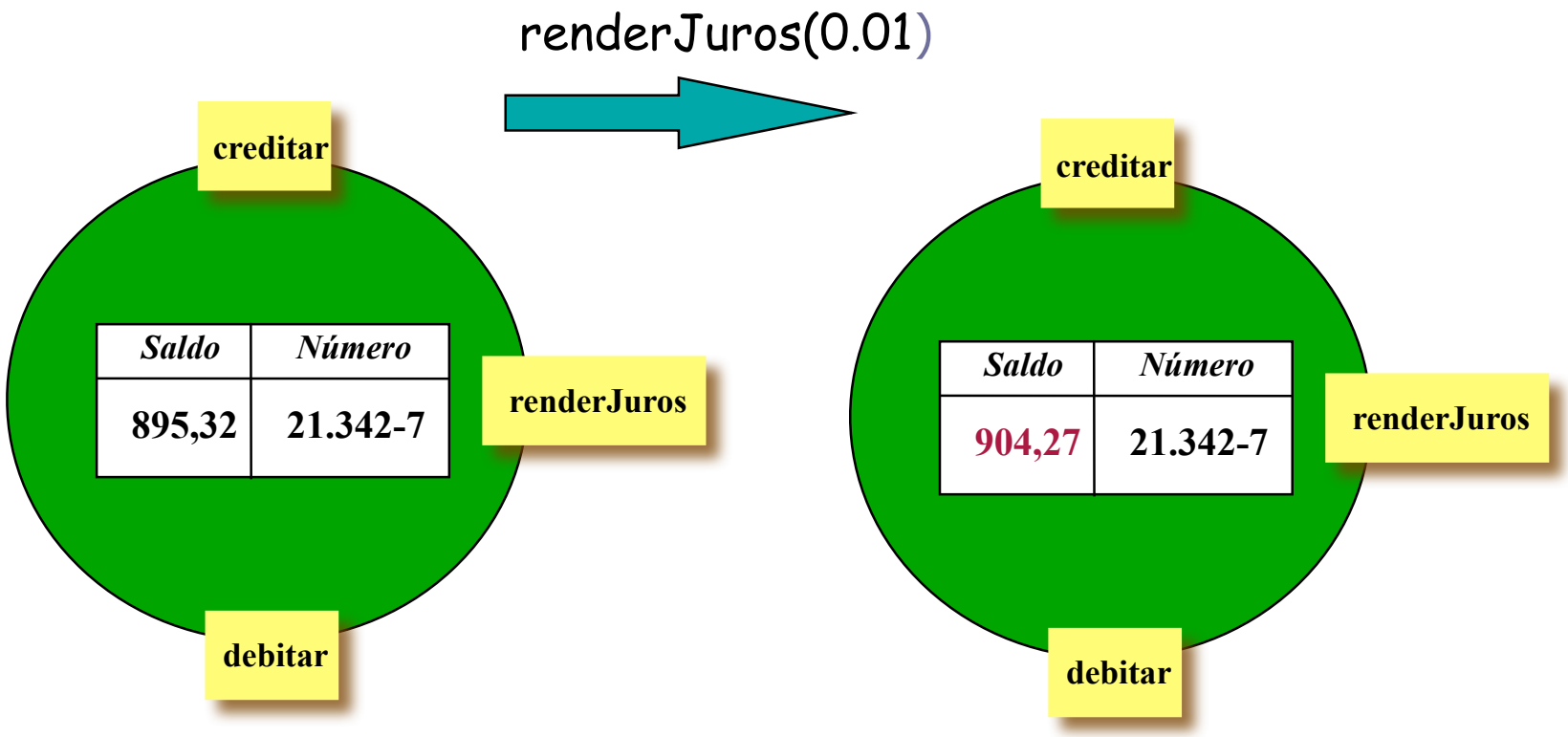
Desenvolvimento de Sistemas

FATEC-PB

Objeto Poupança



Estados do objeto Poupança



Classe de poupanças

```
public class Poupanca {
    private String numero;
    private double saldo;
    ...
    void creditar(double valor) {
        saldo = saldo + valor;
    }
    ...
    void renderJuros(double t) {
        this.creditar(saldo * t);
    }
}
```

Classe RepositorioContasArray

```
public class RepositorioContasArray {

   public void inserirConta(Conta c);
   public void inserirPoupanca(Poupanca p);
   public Conta procurarConta(String num);
   public Poupanca procurarPoupanca(String num);
   ...

}
```

Problemas

- Duplicação desnecessária de código
  - Poupança é muito parecida com Conta
    - Repetição do código de Conta na Poupança
  - Adição de novos métodos no repositório para lidar com Poupança
    - Efeito gigantesco em outros locais
- Como refletir a relação que existe no mundo real: "uma poupança é um tipo de conta"?

Herança

- Reutilizar o código de classes existentes
- Apenas novos atributos ou métodos precisam ser definidos
- Superclasse e Subclasse
- Reuso de Código
  - Tudo que a superclasse tem, a subclasse também tem

Herança em Java

```
public class Poupanca extends Conta {
   //atributos especificos de poupanca
   //metodos especificos de poupanca
}
```

- Restrição de Java
  - Uma classe pode estender apenas uma superclasse diretamente

Herança

- Estrutura herdada
  - Atributos, métodos
- Construtores
  - **não são herdados -> precisam ser implementados na subclasse**
  - construtores da subclasse "**sempre**" utilizam algum construtor da superclasse.

Construtores e subclasses

```
public class Poupanca extends Conta {
  public Poupanca (String num,double saldo,
                          Cliente cliente){
     super(num, saldo, cliente);
  }
}
```

super chama o construtor da superclasse

se **super** não for chamado, o compilador acrescenta uma chamada ao construtor default: **super()**

se não existir um construtor default na superclasse, haverá um erro de compilação

Nova classe Poupanca

```
public class Poupanca extends Conta {

    public Poupanca(String num, double s,
                    Cliente tit) {
       super(num,s,cliente);
    }

    public void renderJuros(double taxa) {
      double saldo = this.getSaldo();
      this.creditar(saldo*taxa);
    }
}
```

Princípio da substituição

```
Conta c1 = new Conta ("12",26.0,cli);
Conta c2 = new Poupanca ("15-1",100.0,cli);

c1 = new Poupanca ("20-1",50.0,cli);
```

- Objetos da subclasse podem ser usados no lugar de objetos da superclasse
  - Toda poupança é uma conta!

- Classe Object

```
   Object qualquer = new Conta("213",
   29,tit);
qualquer = new String ("Nada a ver!");
```

Controle de Acesso

Os níveis de proteção para atributos e métodos que tínhamos eram:

- public - permite acesso a partir de qualquer classe
- private - permite acesso apenas na própria classe

Agora temos:

- protected – permite acesso a partir de qualquer subclasse e classes do mesmo

Impacto causado nas demais camadas

- Todas as camadas lidam com contas
- Princípio da substituição
  - Onde Conta é aceita, Poupança também será
- Logo
  - As demais camadas também aceitam poupanças sem precisar de modificação alguma

Coerções(Casts)

Cast

renderJuros só está na classe Poupança. Compilador não iria deixar passar sem o cast

```
Conta c;
c = new Poupanca("21.342-7");
((Poupanca) c).renderJuros(0.01);
System.out.println(c.getSaldo());
```

instanceof

- Verifica a classe de um objeto
- Recomenda-se o uso de instanceof antes de um cast para evitar erros em tempo de execução
- Dica: instanceof corresponde à pergunta é do tipo?

```
Conta c = procurar("123.45-8");
if (c instanceof Poupanca)
   ((Poupanca) c).renderJuros(0.01);
else
   System.out.print("Não é Poupança!");
```

Cast seguro!

Redefinição de métodos

- Exemplo: nova conta (bonificada)
  - Crédito é um pouco diferente
- Para redefinição (sobreposição = override), método deve ter a mesma assinatura (nome e parâmetros) da superclasse
- Se o nome for o mesmo, mas os parâmetros forem diferentes
  - Aí é sobrecarga (overload)

Redefinição de métodos

```
public class ContaBonificada extends Conta {
  private double bonus;
  public void creditar(double valor) {
    bonus = bonus + (valor * 0.01);
    super.creditar(valor);
  }
  public void renderBonus() {
    super.creditar(bonus);
    bonus = 0;
  }
}
```

Redefinição do método: mesma assinatura!

creditar normal chamado

Polimorfismo

- Habilidade de apelidos de objetos
  - Para a mesma chamada de método
  - Responde de diferentes maneiras
- Para cada invocação no objeto, JVM procura pelo método chamado
  - Se não encontrar, a procura é repassada para a superclasse
  - Ligação dinâmica (dynamic binding)
- Usa redefinição de métodos

Polimorfismo e redefinição

```
public class ContaBonificada extends Conta {
  private double bonus;
  public void creditar(double valor) {
    bonus = bonus + (valor * 0.01);
    super.creditar(valor);
  }
  public void renderBonus() {
    super.creditar(bonus);
    bonus = 0;
  }
}
```

Mesma assinatura, mas comportamento diferente: isso vai definir o polimorfismo

Polimorfismo com ligação dinâmica

```
Conta c1 = repContas.procurar("235");

c1.creditar(200.00);
c1.debitar(100.00);

if (c1 instanceof ContaBonificada)
     ((ContaBonificada)c1).renderBonus();
```

Polimorfismo: mesmo apelido, em tempo de execução, executa diferentemente

Ligação dinâmica: decidiu na hora da execução qual "botão apertar"

Classes Abstratas

Aumenta o poder do polimorfismo

Reuso dos atributos

```
public abstract class ContaGenerica {
  private String numero;
  private double saldo;
  public void creditar(double valor) {
    saldo = saldo + valor;
  }
  public abstract void debitar(double valor);
```

Método sem corpo

```
ContaGenerica c = new Conta(..,30);
ContaGenerica ci = new ContaImposto(..,30);
c.debitar(20.0);
ci.debitar(20.0);
```

O saldo ficou o mesmo?

Exemplo de Polimorfismo

- Vamos definir a classe abstrata SerVivo;
  - Atributos: Idade, nome;
  - Método: falar;
- Vamos definir as classes Homem, Cachorro e Gato, que herdam de SerVivo;
- Vamos colocar objetos em um Vetor de seres vivos;
- Vamos iterar no vetor e chamar o método falar;

Aula Prática

Objetivos da prática

- Exercitar herança e polimorfismo
- Estender o sistema bancário com herança e analisar seus efeitos

Exercício

- Implemente a classe Poupança
  - Estendendo a definição da classe Conta
  - Possui seu próprio construtor
  - Possui um método específico chamado renderJuros(double taxa)
- Implemente a classe ContaBonificada
  - Estendendo a definição de conta, com atributo bonus e método renderBonus

Exercício

- Estenda a interface texto criada no sistema em camadas, adicionando duas novas opções
  - Cadastrar conta bonificada
  - Cadastrar Poupanca
- Que camadas vão mudar?