Universidade Federal do Cariri

HERANÇA

Profª Paola Accioly
Material baseado nos slides de Thaís Alves Burity Rocha

O que vimos aula passada?

- O que é *information hiding*?
- Mecanismos de implementação de *information hiding* em Java
  - Modificadores de acesso
  - Pacotes
  - Métodos *get* e *set*

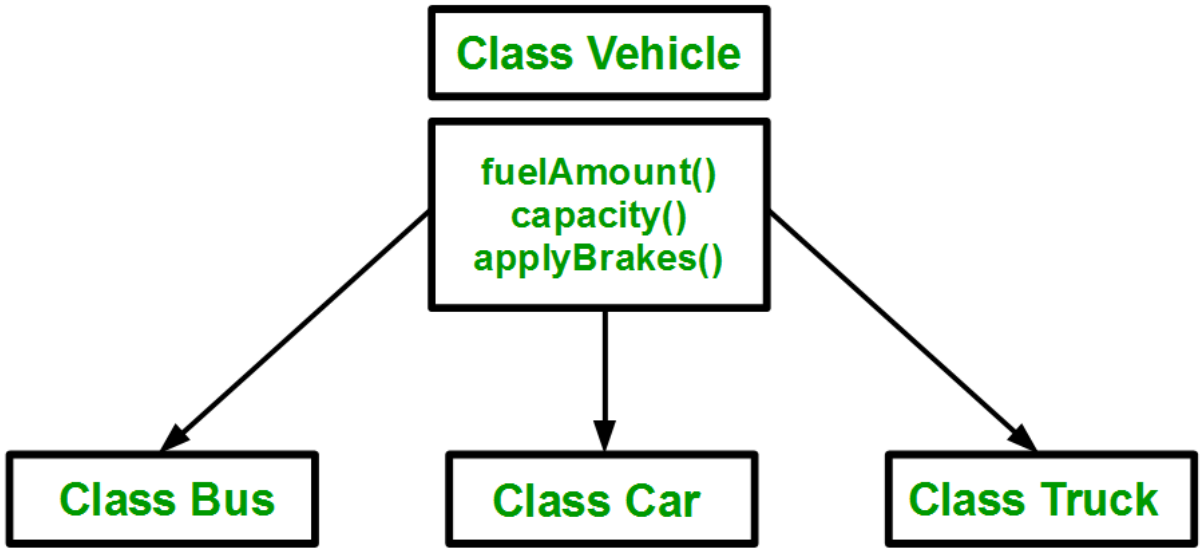
Hoje falaremos sobre herança

O que a família real tem a ver com POO?

- Os filhos podem herdar características físicas e comportamentos dos seus pais

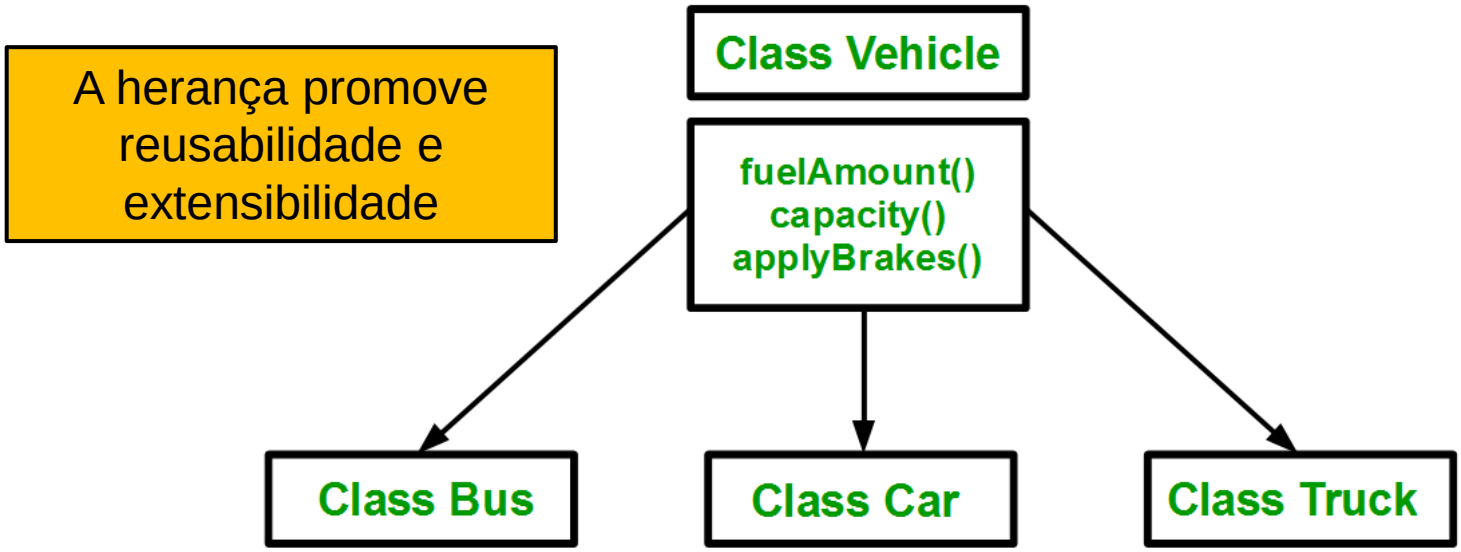
E a POO?

- As classes “filhas” podem herdar da classe “mãe” suas características (atributos) e comportamentos (métodos)

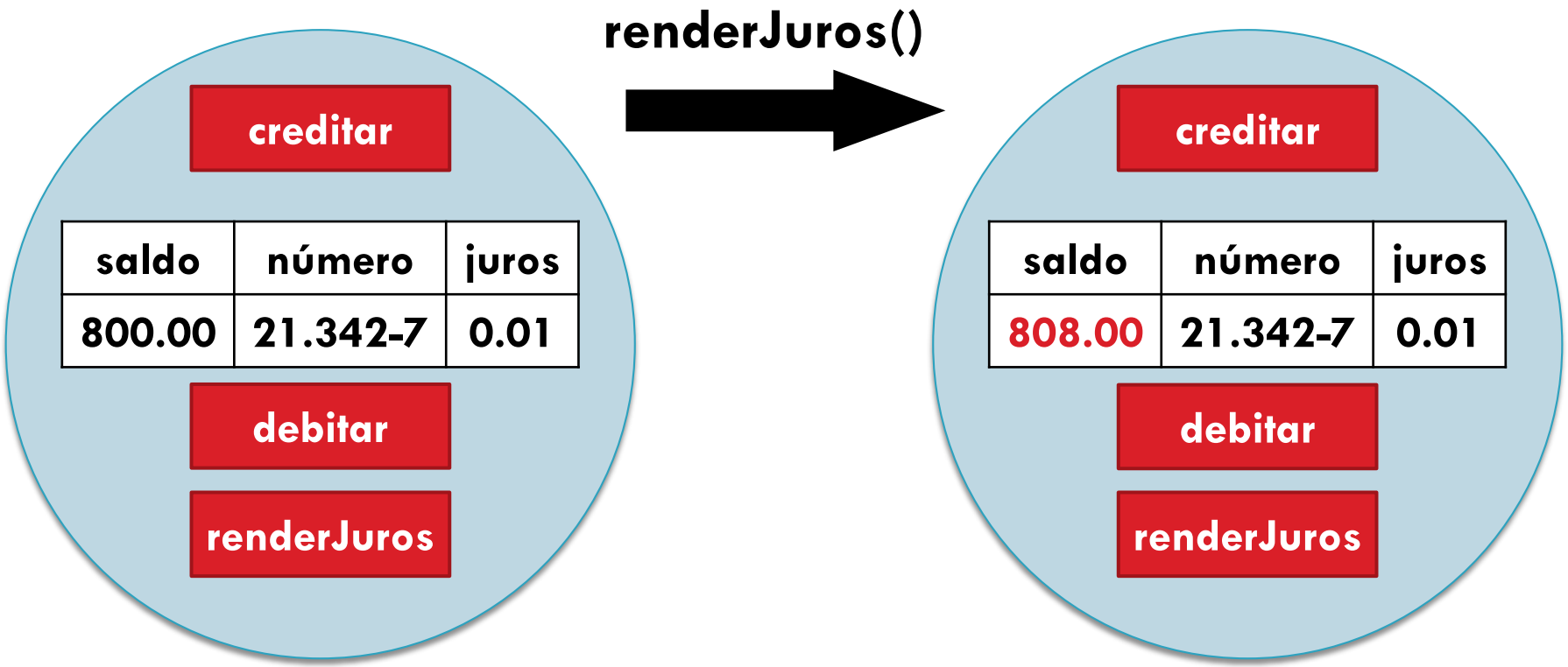


E a POO?

- As classes “filhas” podem herdar da classe “mãe” suas características (atributos) e comportamentos (métodos)



Objeto Poupança



Classe Poupanca

```
public class Poupanca {
    private String numero;
    private double saldo;
    private Cliente cliente;
    private double juros;
    public Poupanca(String n, double s, double j){
        numero = n; saldo = s; juros = j;
    }
    public void creditar(double valor) { saldo += valor; }
    public void debitar(double valor) {...}
    public void transferir(Conta c, double v) {...}
    public double getSaldo() { return saldo; }
    public String getNumero() { return numero; }
    public void setNumero(String n){ numero = n; }
    ...
    public void renderJuros() { creditar(saldo*juros); }
    public double getJuros(){ return juros; }
    public void setJuros(double taxa){ juros = taxa; }
}
```

Classe RepositorioContasVetor

```
public class RepositorioContasVetor {

  public void adicionar (Conta c){...}
  public void adicionar(Poupanca p){...}
  public Conta procurarConta(String num){...}
  public Poupanca procurarPoupanca(String num){...}

  ...

}
```

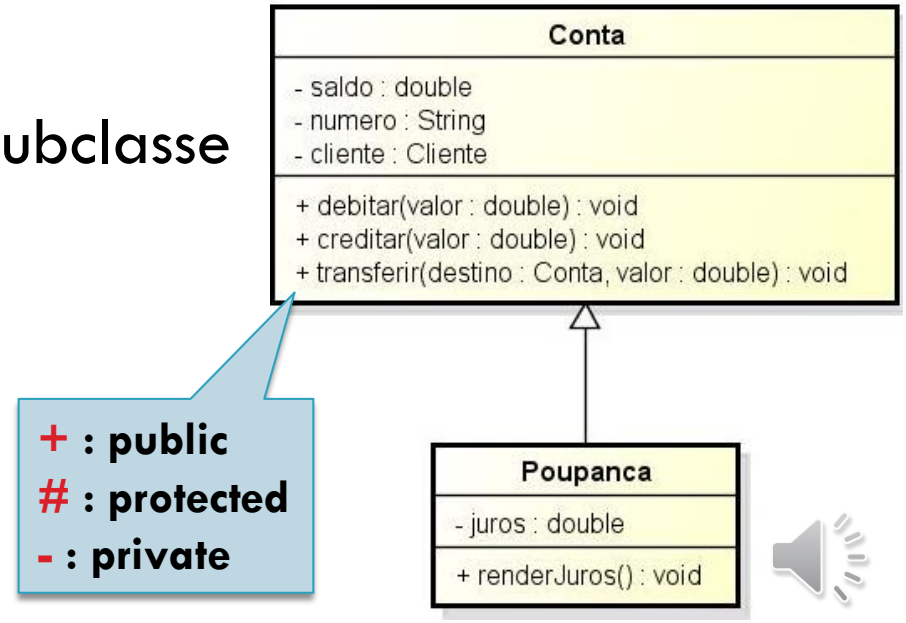
Problema

- Duplicação desnecessária de código...
  - **Poupanca** e **Conta** possuem a maioria dos atributos e métodos iguais
  - Novos métodos em **RepositorioContasVetor** para lidar com **Poupanca**
- Razão: **Uma poupança é um tipo de conta**

Como refletir essa relação no código?

Herança ou Relacionamento É-UM (IS-A)

- Mecanismo para **reutilizar** o código de classes
- **Superclasse:** Classe que será reusada
- **Subclasse:** Classe criada usando herança
- Apenas novos atributos ou métodos precisam ser definidos na subclasse
  - Elementos específicos da subclasse
- "Todo o resto" é herdado



Classe Poupanca

Subclasse

Superclasse

```
public class Poupanca extends Conta {
  private double juros;

  public Poupanca(String n,double s,double j){
     super(n,s);
     this.juros = j;
  }
  public double getJuros(){ return juros; }
  public void setJuros(double t){ juros = t; }
  public void renderJuros() {
     double saldo = this.getSaldo();
     this.creditar(saldo*juros);
  }
}
```

Chama o construtor da superclasse

Herança e construtores

- O construtor da subclasse **sempre** chama o construtor da superclasse
- Se a superclasse contém o construtor sem parâmetros, a subclasse não precisa se preocupar em invocá-lo explícitamente (mas ele será executado)
- **O problema acontece quando a superclasse não possui o construtor sem parâmetros**

O que a subclasse tem acesso?

- **Tudo que for visível, com exceção de construtores**
- Private não é visível por nenhuma outra classe
- Default só é visível por classes no mesmo pacote
- Protected é visível por classes no mesmo pacote e por subclasses
- Public é visível por todas as classes

A subclasse pode acessar atributos private através de getters e setters herdados

protected sem herança

- Mesmo pacote

```
package packA;                    1

public class A {
  protected int valor;
}
```

```
package packA;                    2
public class D {
   public void copiar(){
      A a = new A();
      int x = a.valor;
  }
}
```

- Pacote diferente

```
package packB;                    3
import packA.A;

public class C {
   public void copiar(){
      A a = new A();
      int x = a.valor;
  }
}
```

protected com herança

- Mesmo pacote

```
package packA;                          1

public class A {
  protected int valor;
}
```

```
package packA;                          2

public class B extends A {
   public void copiar() {
      int x = this.valor;
      A a = new A();
      x = a.valor;
   }
}
```

- Pacote diferente

```
package packC;                          3
import packA.A;

public class C extends A {
   public void copiar() {
      int x = this.valor;
      A a = new A();
      x = a.valor;
   }
}
```

Princípio da substituição

- Poupanca é aceito no lugar de Conta
  - Passagem de parâmetro
  - Retorno de método
- Ou seja, as instâncias da subclasse podem ser tratadas de forma unificada

```
RepositorioContas repositorio = new RepositorioContas();
Conta c = new Conta("27.192-3", 580.73);
Poupanca p = new Poupanca("21.342-7", 4700.50, 0.01);
repositorio.adicionar(c);
repositorio.adicionar(p);
```

Explicação

- Variáveis do tipo **Conta** podem referenciar um objeto **Poupanca**

```
Poupanca p = new Poupanca("1234-6", 1900, 0.01);
Conta conta = p;
```

- O **tipo da referência** define os métodos que podem ser chamados, em tempo de compilação

```
p.renderJuros();
conta.renderJuros();
```

Conversão de tipos

- O que fazer para acessar métodos de **Poupanca**?

```
Conta c = new Poupanca("1234-6", 1900, 0.01);
((Poupanca) c).renderJuros();
```

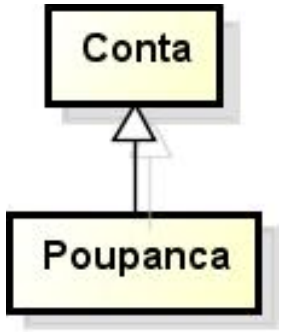
  - **A conversão é da variável de referência apenas**
  - Só vale na mesma hierarquia de classes

- Por segurança, usar operador **instanceof** antes

```
Conta c = new Poupanca("1234-6", 1900, 0.01);
if (c instanceof Poupanca) ((Poupanca) c).renderJuros();
else System.out.print("Poupança inexistente!");
```

Herança múltipla

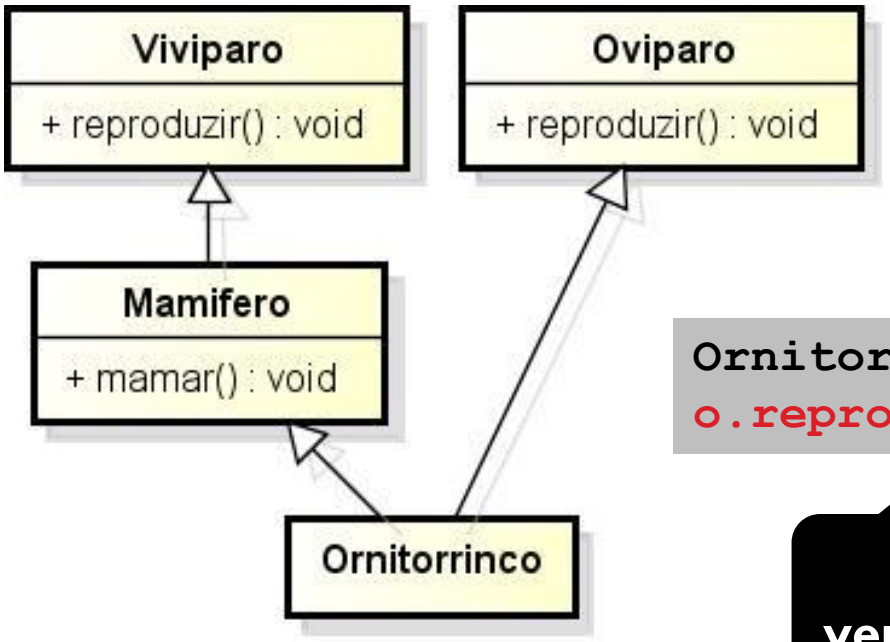
- **Java** permite que uma classe herde de apenas uma superclasse diretamente
  - **Herança simples**



- Outras linguagens OO permitem que uma classe herde diretamente de várias superclasses
  - **Herança múltipla**

Problema de ambiguidade

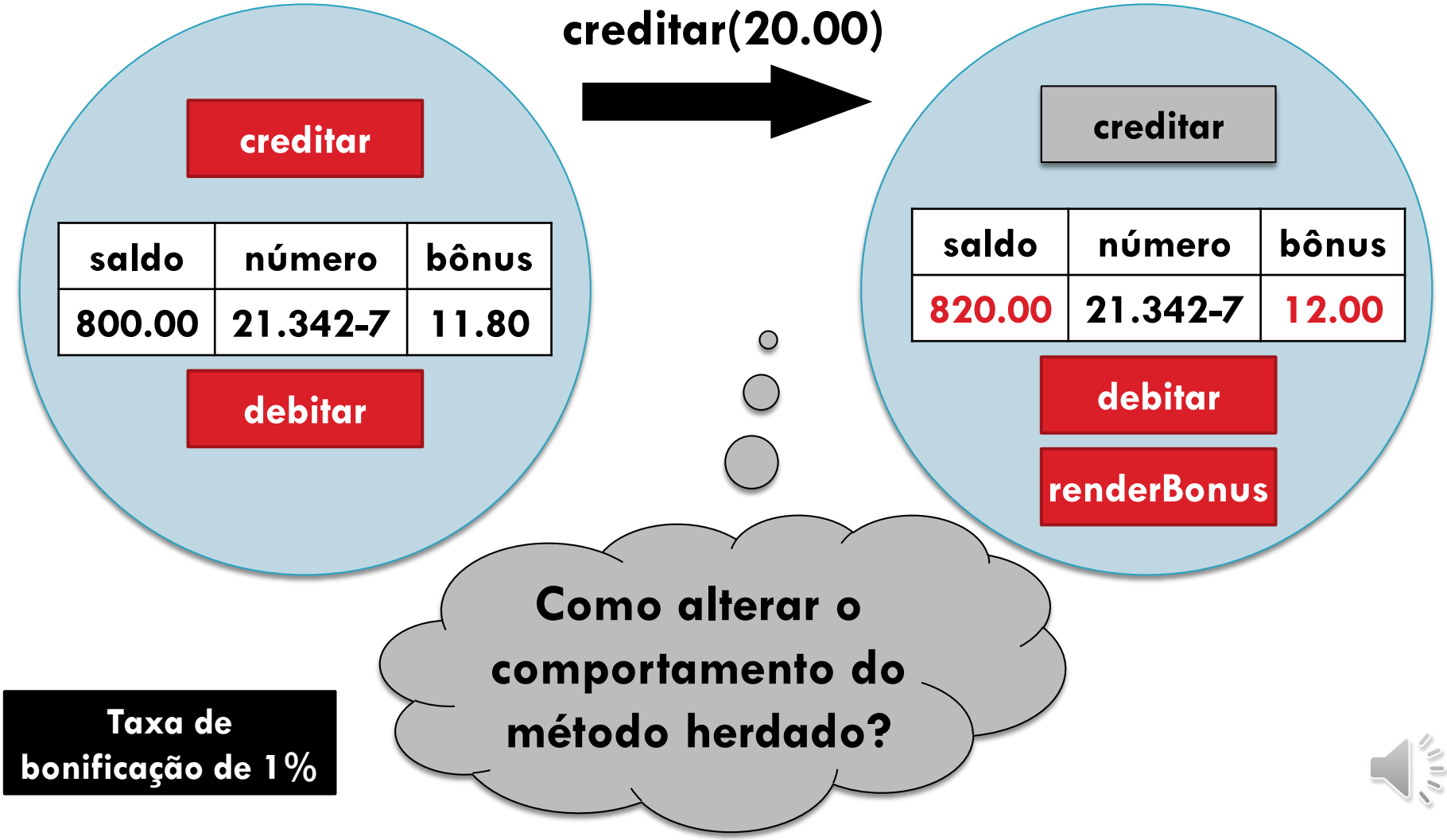
- Herança múltipla aumenta a capacidade de reuso
- Mas pode gerar inconsistência




```
Ornitorrinco o = new Ornitorrinco();
o.reproduzir();
```

Qual será a versão executada?

Conta Bonificada



Sobrescrita de métodos (overriding)

- Possibilidade de uma subclasse possuir um método com mesma assinatura de um método definido na superclasse
- Só existe com herança
- A subclasse personaliza (torna mais específico) o comportamento de um método herdado
- Não infringe o princípio da substituição

Classe ContaBonificada

```
class ContaBonificada extends Conta {
  private double bonus;
  private double taxaBonus = 0.01;
  public ContaBonificada(String n, double s){
    super(n,s);
  }
  public void creditar(double valor) {
    bonus += (valor * taxaBonus); //atualiza bonus
    super.creditar(valor);
  }
  public void renderBonus() {
    super.creditar(bonus);
    bonus = 0;
  }
}
```

Chama creditar() da superclasse

Polimorfismo ...

Código genérico com comportamentos especializados

- Significa que uma chamada de método pode ser executada de várias formas
- Toda conta credita, mas o resultado depende do objeto que executa
  - A JVM decide em tempo de execução
  - Procura na classe do objeto e, caso não encontre, na superclasse

```
Conta c = new ContaBonificada("1234-6", 1900);
c.creditar(100);
c = new Poupanca("1234-0", 1900);
c.creditar(10);
```

Modificador final

- Métodos final não podem ser sobrescritos
- Classes final não podem ter subclasses
  - Exemplo: classe String
- Variável (atributo ou local) final não pode ser alterada após ser inicializada

```
public class Conta{
  private String numero;
  private double saldo;
  private final Cliente titular;
  public static final int AGENCIA_BANCARIA = 236;
  ...
}
```

Constante de classe

Relacionamento IS-A x IS-LIKE-A

- Idealmente a hierarquia suporta os mesmos métodos públicos (IS-A)
  - Substituição perfeita de objetos da superclasse por objetos da subclasse
- Há casos em que a subclasse adiciona métodos, ou seja, estende a superclasse (IS-LIKE-A)
  - Classes Poupança e ContaBonificada
  - A substituição não é perfeita porque os métodos da subclasse não são acessíveis da superclasse
- Ambos os casos são válidos

Boas práticas

- Considere usar herança quando tiver um comportamento compartilhado entre várias classes
  - **creditar()**, **debitar()** e **transferir()** são comportamentos comuns à toda conta
- Não use herança se a subclasse e a superclasse não passarem no teste **É-UM**
  - Chá **É-UMA** Bebida (OK)
  - Bebida **É-UM** Chá (ERRADO)

Boas práticas

- Não use herança **apenas** para poder reusar código
  - **Alarme** emite som
  - **Piano** emite som
  - Mas **Piano** não é um **Alarme**

- Utilizar **anotação** na sobrescrita de métodos

```
@Override
public void creditar(double valor){...}
```

Classe Object

- Toda classe em Java herda de java.lang.Object
- Principais métodos (para a disciplina)
  - equals()
  - toString()
- Esses métodos **devem** ser sobrescritos
- Vamos estudá-los para entender melhor sobre herança (e conceitos relacionados)

Método equals()

- Assinatura: **public boolean equals(Object obj)**
- Sinaliza se 2 objetos são semânticamente iguais
- Por default, é equivalente ao operador ==

```
public boolean equals(Object obj){
    if(obj instanceof Conta){
      Conta conta2 = (Conta)obj;
      if(this.numero.equals(conta2.getNumero())){
        return true;
      }
    }
    return false;
}
```

Método toString()

- Assinatura: **public String toString()**
- Retorna uma representação textual do objeto
- Por default, retorna um "código"
  - <Classe>@<hash code do objeto em hexadecimal>

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

  public String toString(){
    return "Nome: " + nome + "; CPF: " + cpf;
  }
}
```