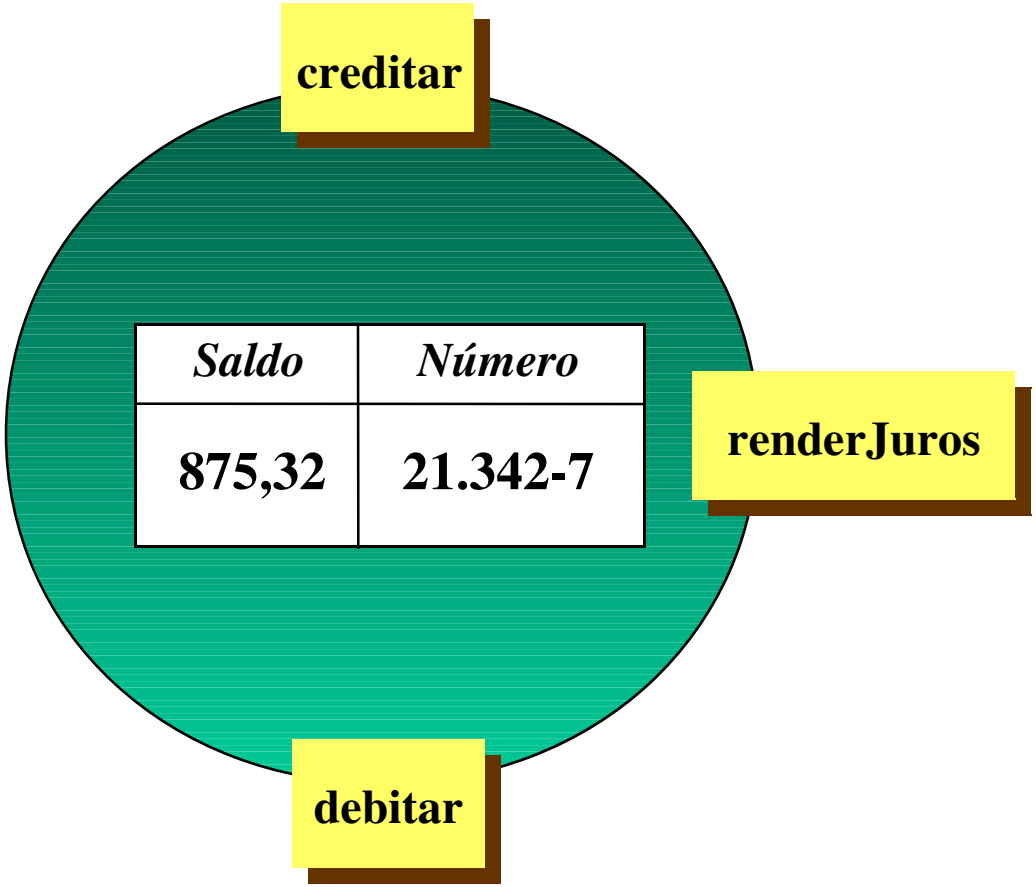
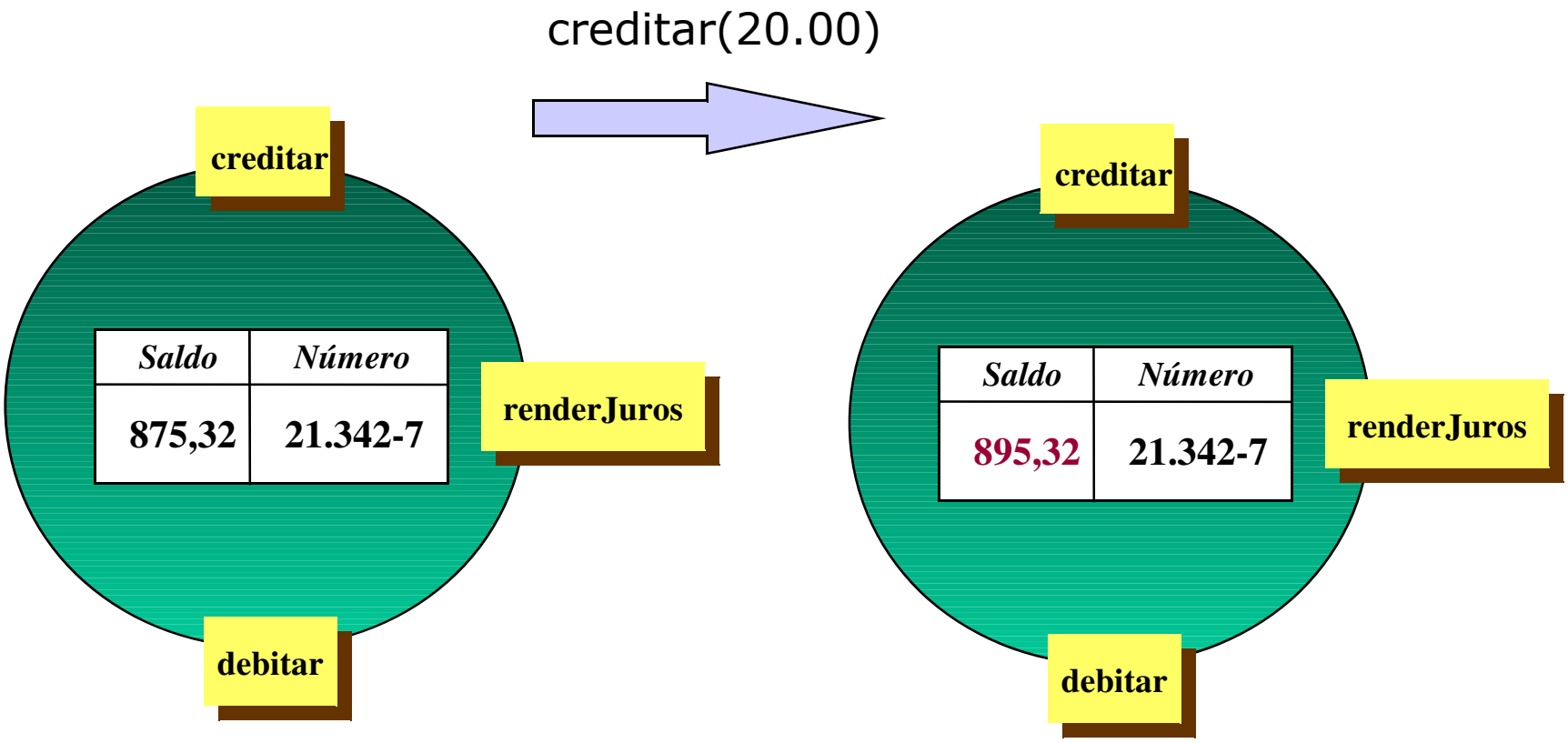
Herança, polimorfismo e ligação dinâmica

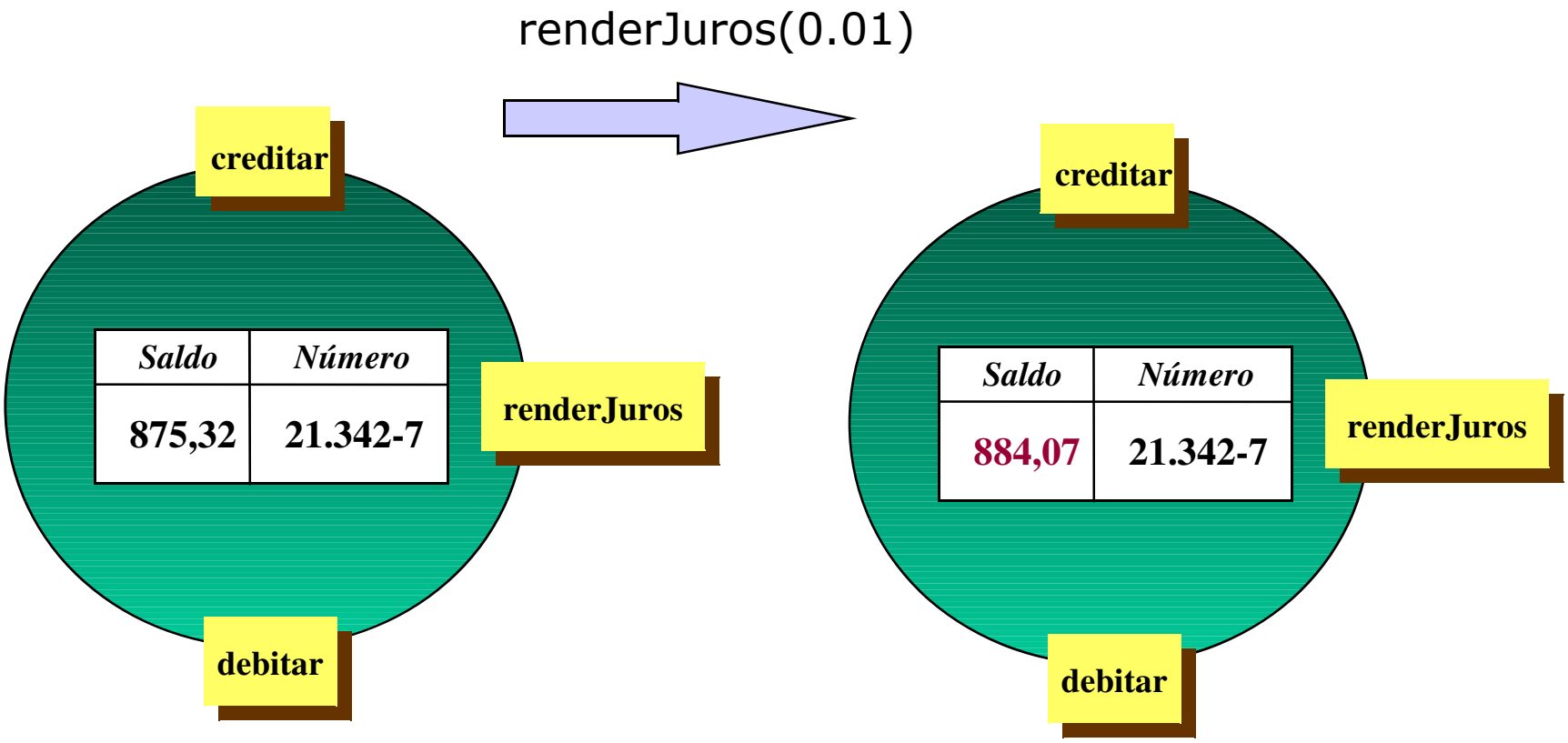
Objeto poupança



Estados do Objeto Poupança



Estados do Objeto Poupança



Classe de Contas: Assinatura

```
class Conta {
   public Conta(String n) {}

   public void creditar(double valor) {}
   public void debitar(double valor) {}
   public String getNumero() {}
   public double getSaldo() {}
}
```

Classe de Poupanças: Assinatura

```
class Poupanca {
   public Poupanca(String n) {}

   public void creditar(double valor) {}
   public void debitar(double valor) {}
   public String getNumero() {}
   public double getSaldo() {}
   public void renderJuros(double t) {}
}
```

Classe de Poupanças: Descrição

```
class Poupanca {
    private String numero;
    private double saldo;

    void creditar(double valor) {
        saldo = saldo + valor;
    }
    String getNumero() {
        return numero;
    }
    ...
    void renderJuros(double t) {
        this.creditar(saldo * t);
    }
}
```

Problemas

- Duplicação desnecessária de código:
  - A definição de `Poupanca` é uma simples extensão da definição de `Conta`
  - Clientes de `Conta` que precisam trabalhar também com `Poupanca` terão que ter código especial para manipular poupanças

- Falta refletir relação entre tipos do "mundo real": uma poupança também é uma conta!

Herança

- O mecanismo de herança permite reutilizar o código de classes existentes
- Apenas novos atributos ou métodos precisam ser definidos
- Herança introduz os conceitos de:
  - Superclasse e Subclasse
  - Redefinição de Métodos
  - Polimorfismo de subtipo

Nova classe Poupança (com herança)

```
class Poupanca extends Conta {

  public Poupanca (String num, Cliente c) {
    super(num, c);
  }
  public void renderJuros(double taxa) {
    double saldoAtual = getSaldo();
    creditar(saldoAtual * taxa);
  }
  //E nada mais!
}
```

subclasse **extends** superclasse

Herança

- Reuso de Código:
  - tudo que a superclasse tem, a subclasse também tem
  - o desenvolvimento pode se basear em o que já está pronto

- Extensibilidade:
  - algumas operações da superclasse podem ser **redefinidas** na subclasse

Herança

- Comportamento:
  - objetos da subclasse comportam-se como os objetos da superclasse

- Princípio da Substituição:
  - objetos da subclasse podem ser usados no lugar de objetos da superclasse
  - Toda Poupanca é uma Conta mas nem toda conta é uma Poupança

Princípio da substituição

```
...
Poupanca  p;
p = new Poupanca("21.342-7");
p.creditar(500.87);
p.debitar(45.00);
System.out.println(p.getSaldo());
...
```

Os métodos creditar e debitar são herdados de Conta

```
...
Conta  c;
c = new Poupanca("21.342-7");
c.creditar(500.87);
c.debitar(45.00);
System.out.println(c.getSaldo());
...
```

Uma poupança pode ser usada no lugar de uma conta

Substituição e Casts

- Nos contextos onde contas são usadas pode-se usar poupanças
  - Onde Conta é aceita, Poupanca também será
- Nos contextos onde poupanças são usadas pode-se usar contas **com o uso explícito de casts**

Casts

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

System.out.println(c.getSaldo());
...
```

cast

renderJuros só está disponível na classe Poupança. Por isso o cast para Poupança é essencial.

instanceof

- O operador **instanceof** verifica a classe de um objeto (retorna true ou false)
- Recomenda-se o uso de instanceof antes de se realizar um cast para evitar erros

```
...
Conta c = procura("123.45-8");

if (c instanceof Poupanca)
  ((Poupanca) c).renderJuros(0.01);
else
  System.out.print(“Poupança inexistente!”)
...
```

Herança e a classe Object

- Toda classe que você define tem uma superclasse
- Se não for usado "extends", a classe estende a classe "Object" de Java.
- A classe Object é a única classe de Java que não estende outra classe

```
class Cliente {}
```

São equivalentes!

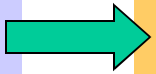
```
class Cliente extends Object {}
```

Construtores e subclasses

```
class ContaBonificada extends Conta {
  private double bonus;
  ...
  public contaBonificada(String num,Cliente c) {
    super(num,c);
  }
}
```

super chama o construtor da superclasse

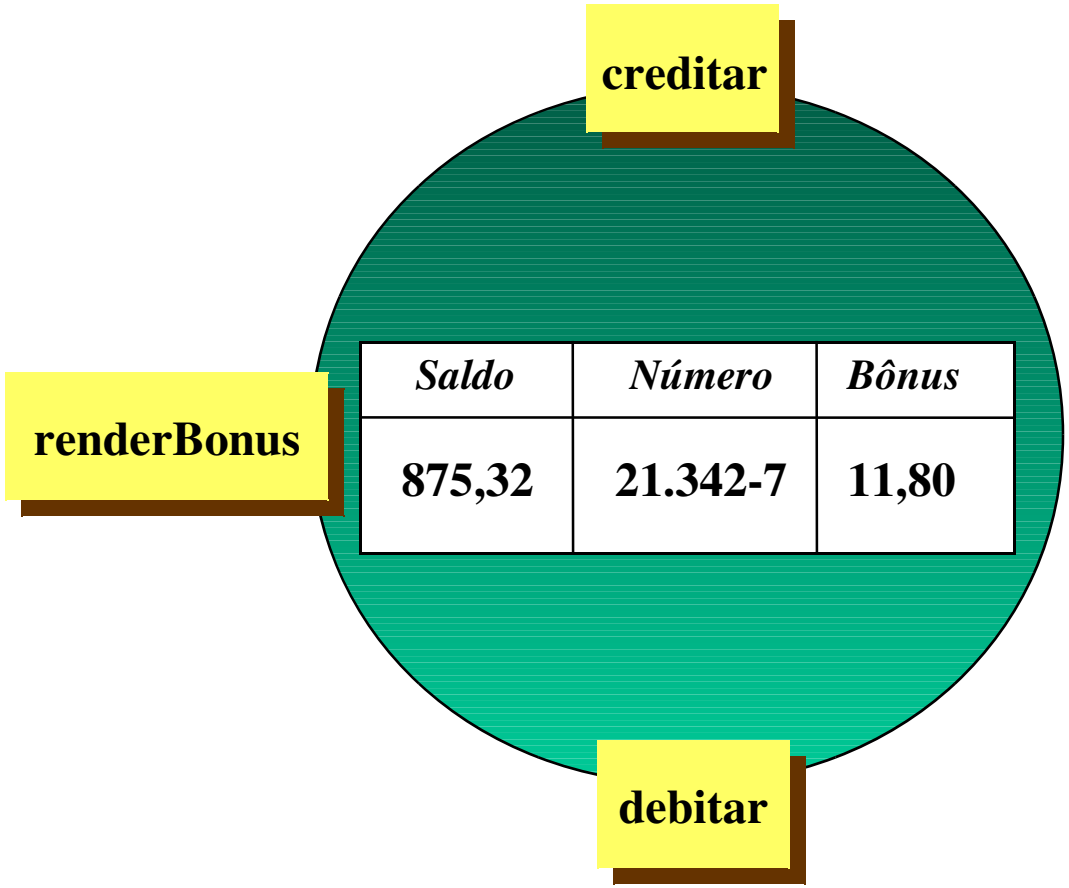
se **super** não for chamado, o compilador acrescenta uma chamada ao construtor default: **super()**



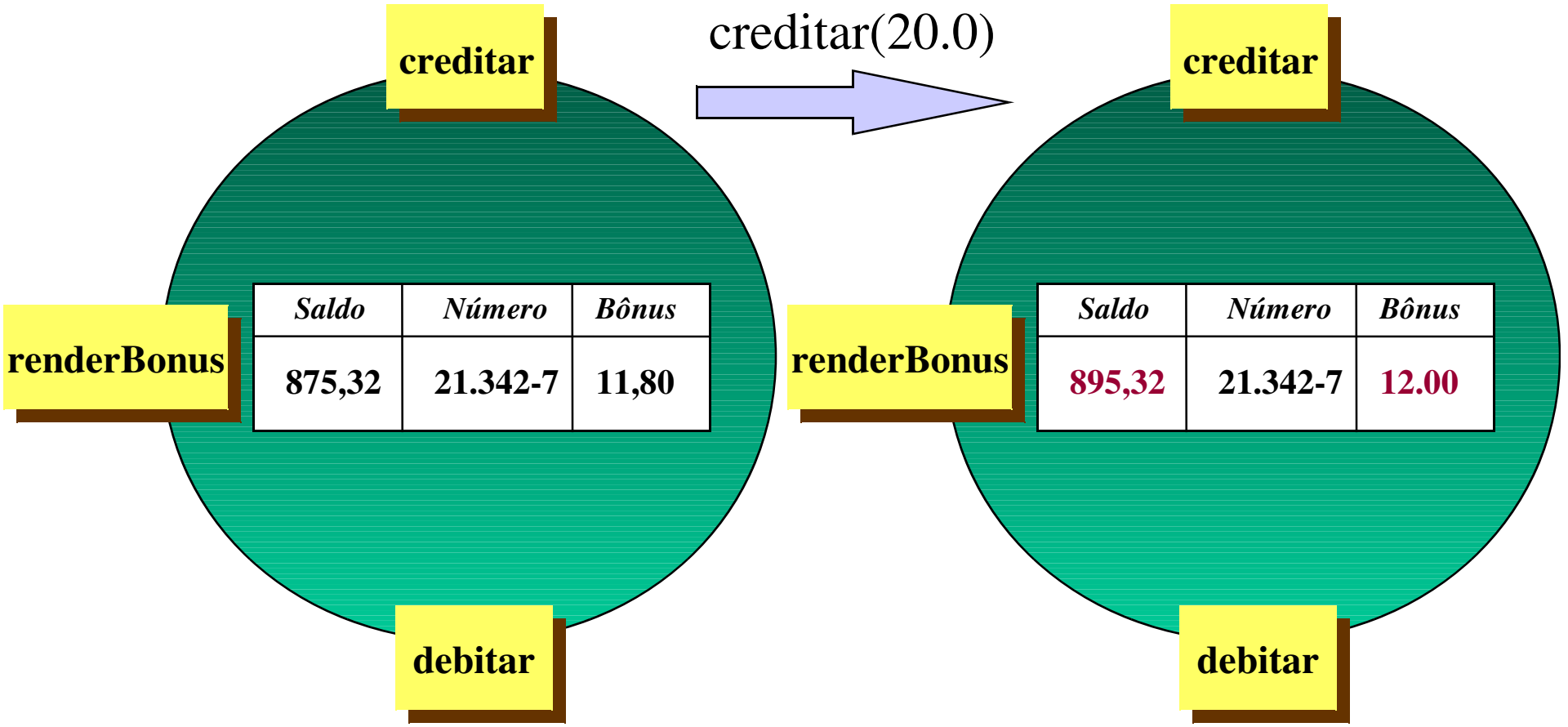
se não existir um construtor default na superclasse, haverá um erro de compilação

Overriding

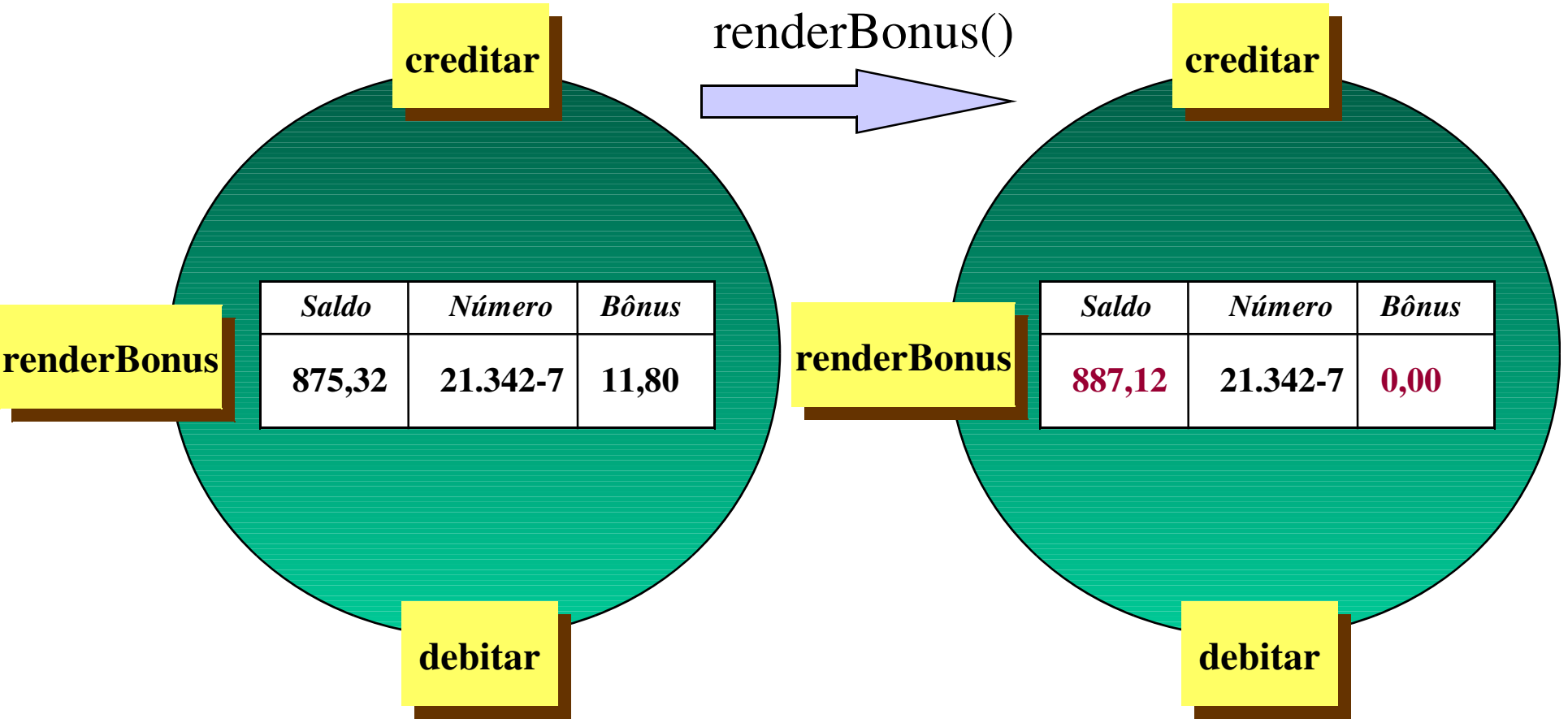
Objeto Conta Bonificada



Estados de uma Conta Bonificada



Estados de uma Conta Bonificada



Contas Bonificadas: Assinatura

```
class ContaBonificada extends  Conta {
  public void renderBonus() {}
  public double getBonus() {}
  public void creditar(double valor) {}

  public ContaBonificada(String num,
              Cliente c) {}
}
```

Contas Bonificadas: Descrição

```
class ContaBonificada extends Conta {
  private double bonus;

  public ContaBonificada(String num, Cliente
    super(num,c);
  }
  public void creditar(double valor) {
    bonus = bonus + (valor * 0.01);
    super.creditar(valor);
  }
  public void renderBonus() {
    super.creditar(bonus);
    bonus = 0;
  }
  public double getBonus() {
    return bonus;
  }

}
```

Redefinição do método creditar

Usando Contas Bonificadas

```
public static void main(String args[]){

  ContaBonificada cb;
  cb = new ContaBonificada("21.342-7");
  cb.creditar(200.00);
  cb.debitar(100.00);
  cb.renderBonus();
  System.out.print(cb.getSaldo());

}
```

Overrinding

- Redefinição de métodos herdados da superclasse
- Para que haja a redefinição de métodos, o novo método deve ter a mesma assinatura (nome e parâmetros) que o método da super classe
- Se o nome for o mesmo, mas os parâmetros forem de tipos diferentes haverá overloading e não redefinição
- Redefinições de métodos devem preservar o comportamento (semântica) do método original
  - a semântica diz respeito ao estado inicial e estado final do objeto quando da execução do método

Polimorfismo e Ligações Dinâmicas

- Dois métodos com o mesmo nome e tipo:
  - qual versão do método usar?
- O método é escolhido dinamicamente (em tempo de execução), não estaticamente (em tempo de compilação)
- A escolha é baseada no tipo do objeto que recebe a chamada do método e não da variável

Ligações Dinâmicas

```
...
Conta c1, c2;
c1 = new ContaBonificada("21.342-7");
c2 = new Conta("12.562-8");

c1.creditar(200.00);
c2.creditar(100.00);
c1.debitar(100.00);
c2.debitar(60.00);

((ContaBonificada) c1).renderBonus();
System.out.println(c1.getSaldo());
System.out.println(c2.getSaldo());
...
```

Qual é o **creditar** chamado?

Overloading

Overloading

- Quando se define um método com mesmo nome, mas com parâmetros de tipos diferentes, não há redefinição e sim

overloading ou sobrecarga

- Overloading permite a definição de vários métodos com o mesmo nome em uma classe. O mesmo vale para construtores
- A escolha do método a ser executado é baseada no tipo dos parâmetros passados

Overloading

```
class Formatacao {
   static String formatar(double d, int
   precisao){...}
   static String formatar(double d) {...}
   static String formatar(int d) {...}
   ...
}
```

Aqui o método **formatar** tem três versões "overloaded"

```
//chama o primeiro método
String s1 = formatar(10.0, 2);
//chama o terceiro método
String s2 = formatar(99);
```

Herança e Modificadores

Uso de protected e private em Herança

- Atributos e métodos com o modificador **protected** podem ser acessados na classe em que são declarados e nas suas **subclasses**

- **Os membros private de uma superclasse são acessíveis apenas em métodos dessa superclasse**

Classes e métodos *final*

- Classes declaradas com o modificador final não podem ter subclasses

```
final class GeradorSenhas {

}
```

  - Usado por segurança
  - String são exemplos de classes final

- Um método que é declarado final não pode ser redefinido em uma subclasse