Programação Orientada a Objetos

Classes Abstratas

Sérgio Soares
scbs@cin.ufpe.br

Até aqui

- Quando usar herança?

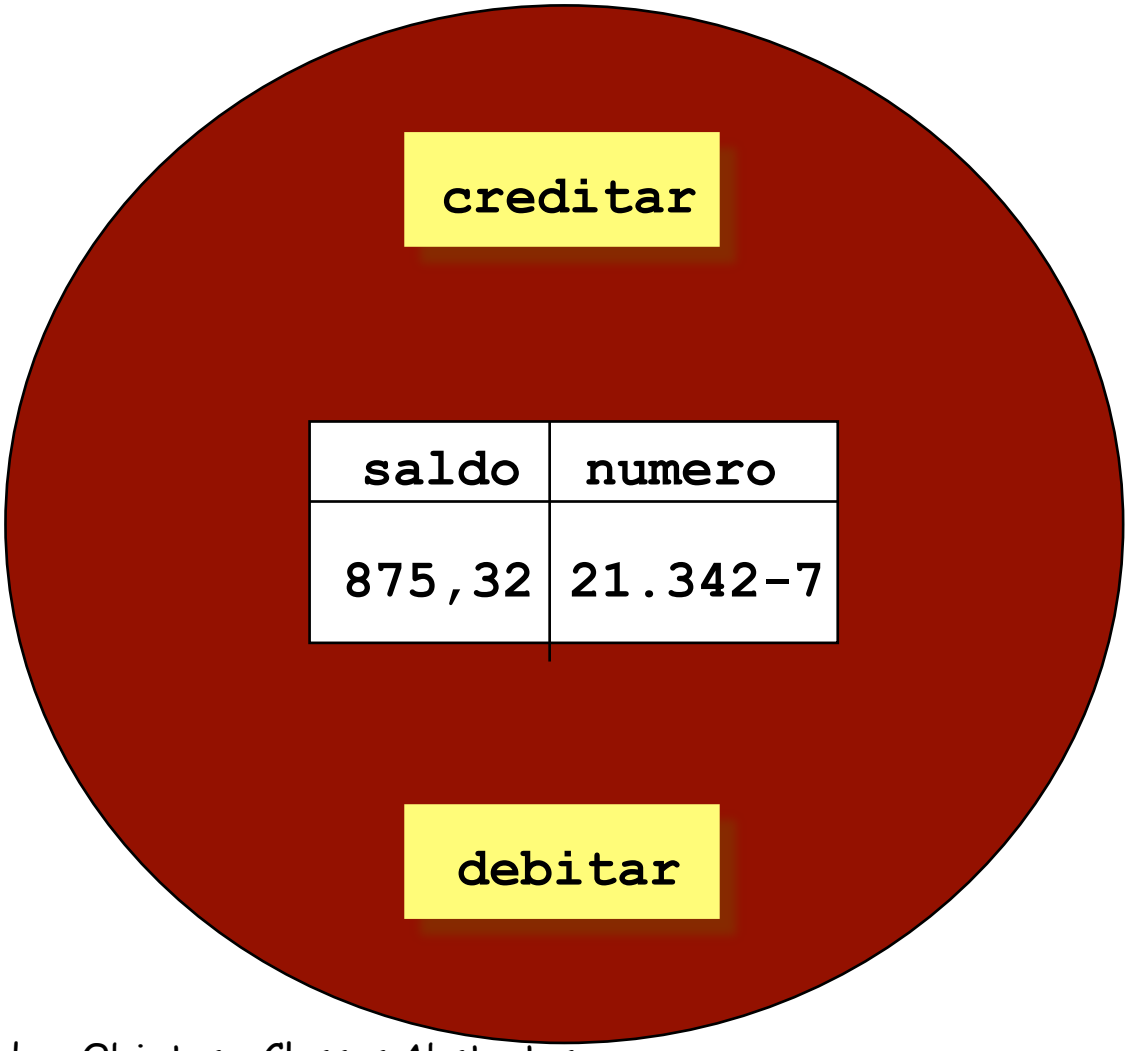
- Ao redefinir um método manter o comportamento herdado!

Adivinhem...

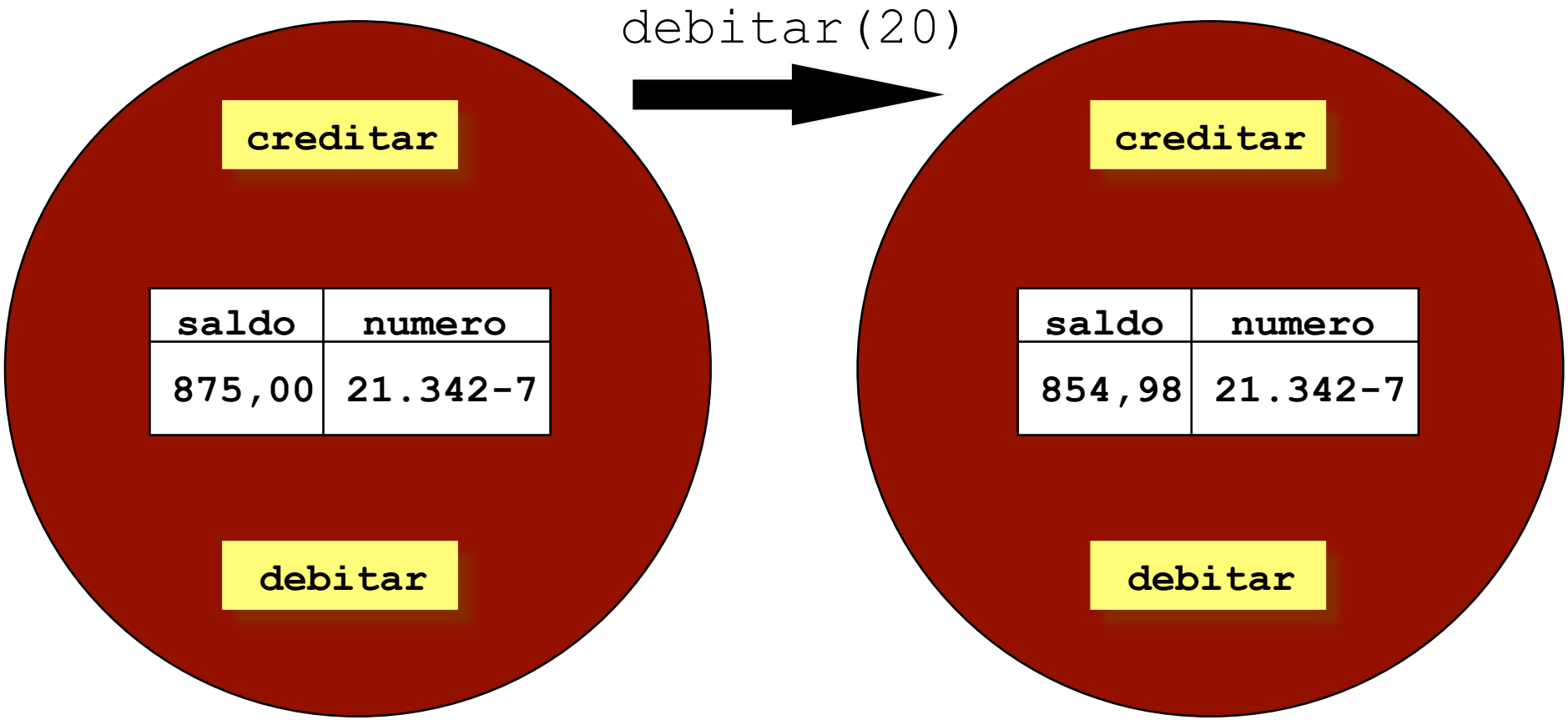
- Surge um novo requisito na aplicação bancária

- Temos de cobrar um imposto em certos tipos de contas
  - Lembram da CPMF?

Objeto Conta Imposto



Estados do Objeto Conta Imposto



Conta Imposto: Assinatura **Sem** herança

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

Conta Imposto: Assinatura
Com herança

```
public class ContaImpostoM extends Conta {

  public ContaImpostoM(String numero) {}
  public void debitar(double valor) {}

}
```

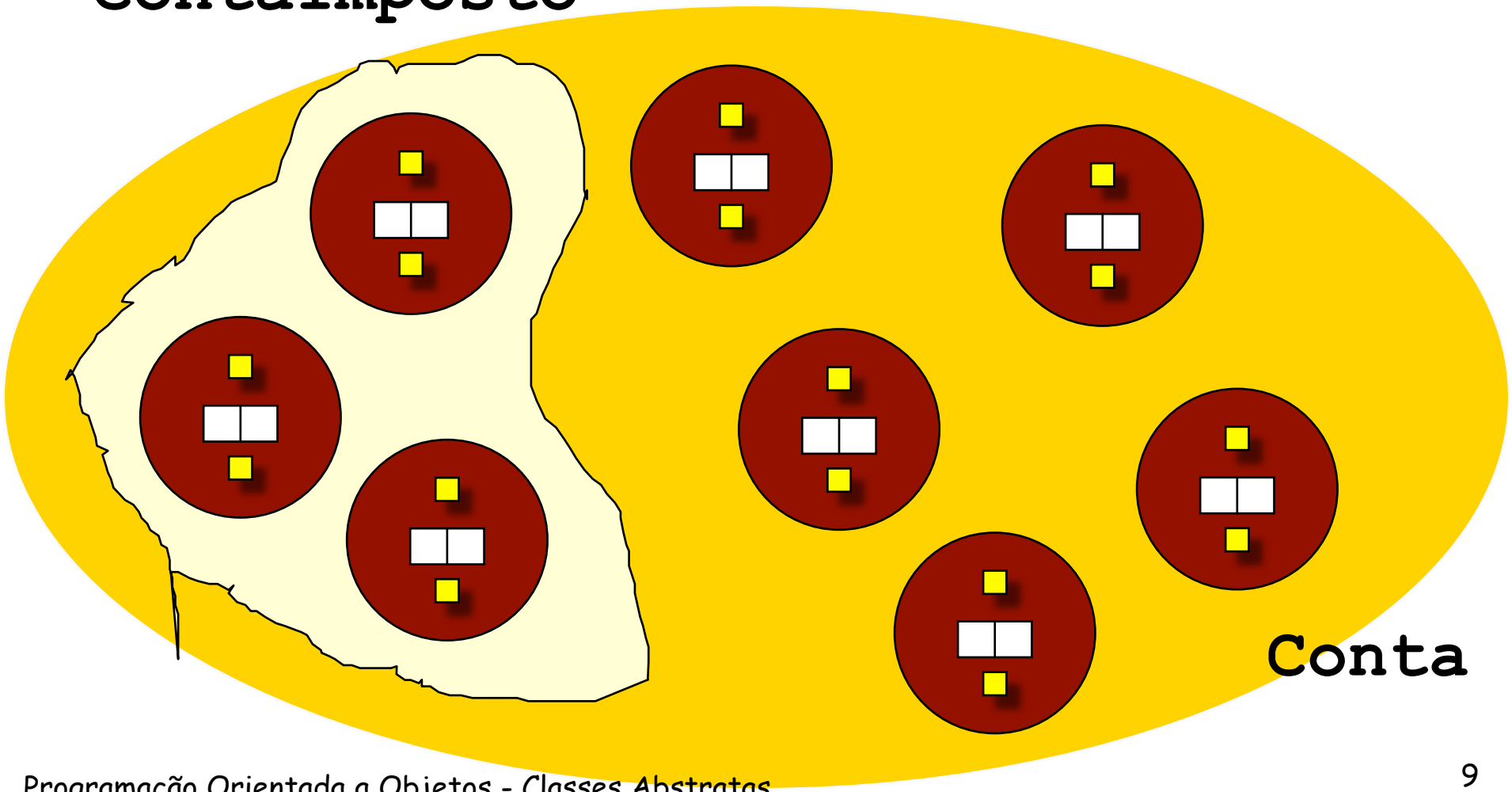
Conta Imposto: Descrição
Com herança

```
public class ContaImpostoM extends Conta {

  private static final double CPMF = 0.0038;

  public ContaImpostoM (String numero) {
    super (numero);
  }
  public void debitar(double valor) {
    double imposto = (valor * CPMF);
    super.debitar(valor + imposto);
  }
}
```

Subtipos e Subclasses



Subclasses e Comportamento (1)

- Objetos da subclasse devem se comportar como os objetos da superclasse
  - Afinal de contas queremos usar objetos da subclasse onde os objetos da superclasse são utilizados

```
public class Banco {
   private Conta[] contas;
   private int indice;
   // ...
}
```

Subclasses e Comportamento (2)

- Redefinições de métodos devem preservar o comportamento (semântica) do método original
  - No que diz respeito ao comportamento (e atributos) herdado
- Grande impacto sobre manutenção/ evolução de software...

Revisão/Otimização de Código

```
...
double m(Conta c) {
   c.creditar(10);
   c.debitar(10);
   return
      c.getSaldo();
}
...
```

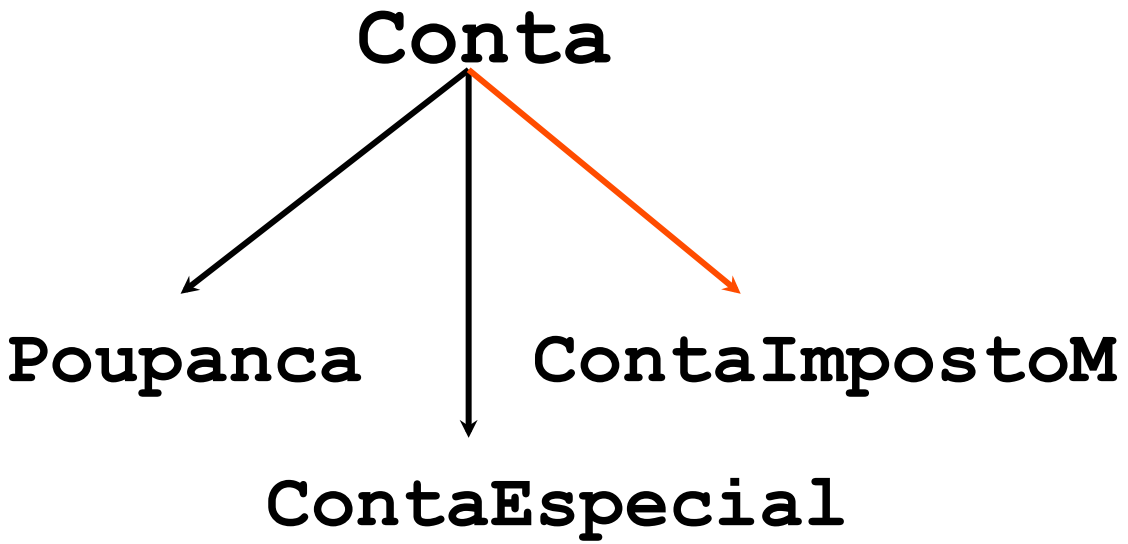
```
...
double m(Conta c) {
    return
       c.getSaldo();
}
...
```

Considerando apenas o retorno do método `m`, as duas opções são sempre equivalentes? Em que contextos?

Subclasses e Evolução de Software

- Deveria ser possível raciocinar sobre o código usando-se apenas a definição dos tipos das variáveis envolvidas (`Conta`)
- O comportamento do código deveria ser independente do tipo do objeto (`Conta`, `ContaEspecial`, `ContaImposto`) associado a uma dada variável em tempo de execução

Reuso sem Subtipos

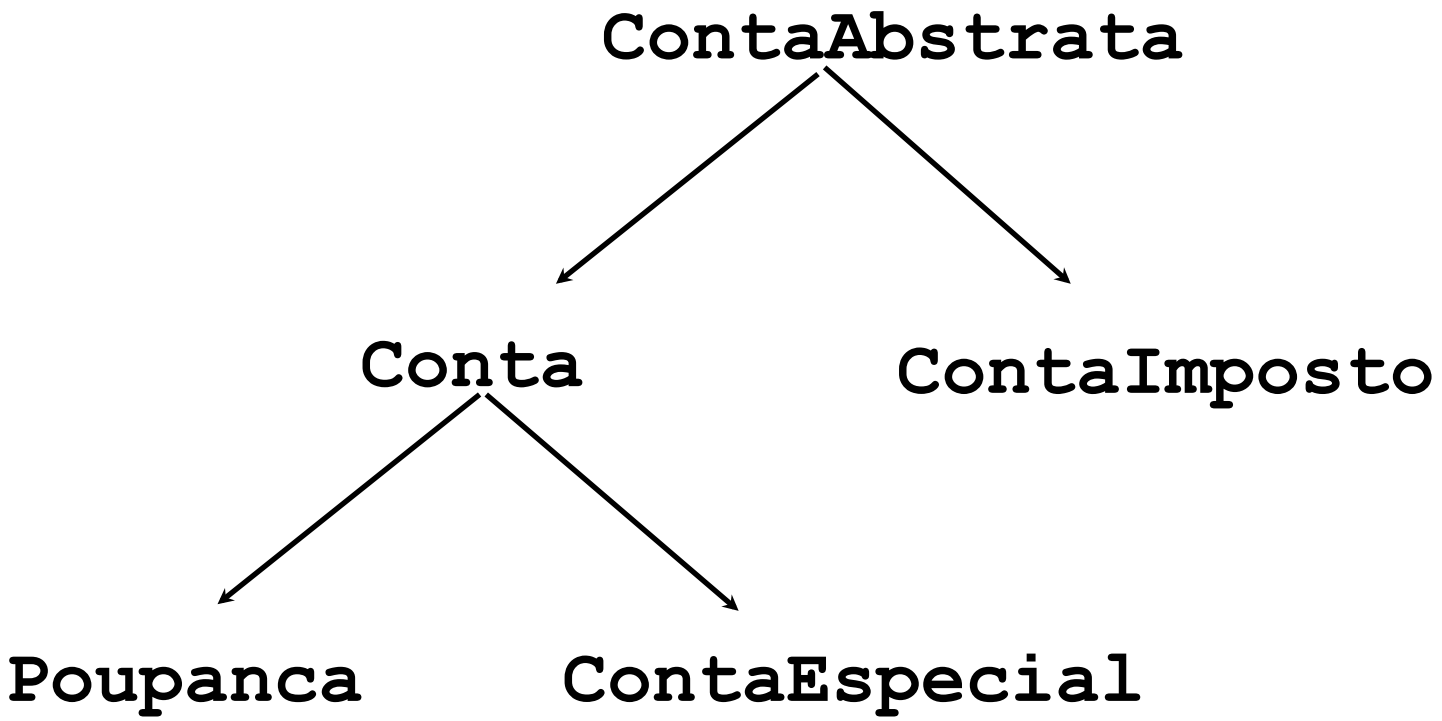


`ContaImpostoM` muda a semântica do método `debitar` e se herdar de `Conta` quebra a noção de subtipos!!!

Qual a alternativa então?

- O que existe de comum entre `Conta` e `ContaImposto`?
  - Vamos criar uma nova classe que contenha essa parte em comum
  - `Conta` e `ContaImposto` devem herdar dessa nova classe
- Atenção: `debitar` é diferente nas duas classes
  - Mas ambas as contas devem permitir debitar um valor ...

Reuso preservando Subtipos



Definindo Classes Abstratas

```
public abstract class ContaAbstrata {
  private String numero;
  private double saldo;
  public ContaAbstrata (String numero) {
    this.numero = numero;
    this.saldo = 0.0;
  }
  public void creditar(double valor) {
    this.saldo = this.saldo + valor;
  }
  public double getSaldo() {
    return this.saldo;
  }
```

Definindo Classes Abstratas

```
  public String getNumero() {
    return this.numero;
  }
  protected void setSaldo(double saldo) {
    this.saldo = saldo;
  }
  public abstract void debitar(double valor);
}
```

O método abstrato não tem implementação, mas

1. Permite programar (não executar) chamando o método da classe abstrata (na classe `Banco`, por exemplo)
2. Obriga que as subclasses concretas implementem o método

Revisão/Otimização de Código

```
...
double m(ContaA c) {
   c.creditar(10);
   c.debitar(10);
   return
      c.getSaldo();
}
...
```

```
...
double m(ContaA c) {
    return
       c.getSaldo();
}
...
```

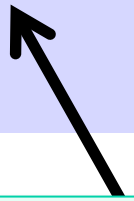
E agora, a modificação é correta?
Em que contextos?

Classes Abstratas

- Possibilita herança de código preservando comportamento (semântica)
  - Não do método `debitar`, claro
- Métodos abstratos:
  - geralmente existe pelo menos um
  - são implementados nas subclasses
- Não cria-se objetos:
  - mas devem ter construtores, para reuso
  - se necessário, métodos `protected` para serem acessados nas subclasses

Contas: Descrição Modificada

```
public class Conta extends ContaAbstrata {
   public Conta(String numero) {
      super (numero);
   }
   public void debitar(double valor) {
      this.setSaldo(this.getSaldo() - valor);
   }
}
```



Implementação do método abstrato observe o uso do método `protected`

Poupanças: Descrição Original

```
public class Poupanca extends Conta {
   public Poupanca(String numero) {
      super (numero);
   }
   public void renderJuros(double taxa) {
      this.creditar(this.getSaldo() * taxa);
   }
}
```

Nada mudou para `Poupanca`

Conta Especial: Descrição Original

```
public class ContaEspecial extends Conta {
   public static final double TAXA = 0.01;
   private double bonus;
   public ContaEspecial (String numero) {
      super(numero);
   }
   public void creditar(double valor) {
      this.bonus = this.bonus + (valor * TAXA);
      super.creditar(valor);
   }
}
```

Nada mudou para `ContaEspecial`

Conta Imposto: Descrição

```
public class ContaImposto extends ContaAbstrata {
   public static final double CPMF = 0.0038;
   public ContaImposto (String numero) {
      super(numero);
   }
   public void debitar(double valor) {
      double imposto = valor * CPMF;
      double total = valor + imposto;
      super.setSaldo(this.getSaldo() - total);
   }
}
```

Implementação do método abstrato observe o uso do método `protected`

Substituição e Ligações Dinâmicas

```
ContaAbstrata ca1, ca2;
ca1 = new ContaEspecial("21.342-7");
ca2 = new ContaImposto("21.987-8");
ca1.debitar(500);
ca2.debitar(500);
System.out.println(ca1.getSaldo());
System.out.println(ca2.getSaldo());
```

Exemplo de um pedaço de um método `main`

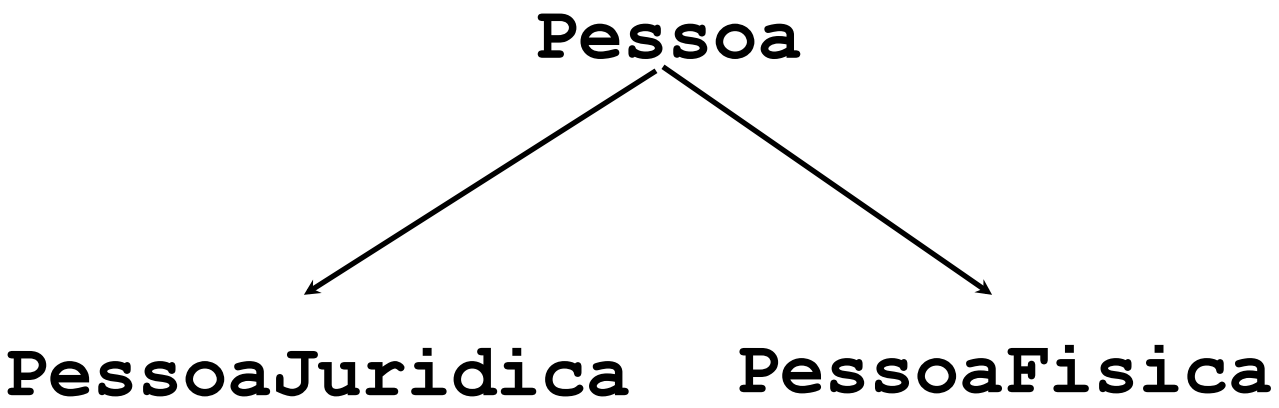
Classes Abstratas: Utilização

- Herdar código sem quebrar noção de subtipos, preservando o comportamento do supertipo
- Generalizar código, através da abstração de detalhes não relevantes
- Projetar sistemas, definindo as suas arquiteturas e servindo de base para a implementação progressiva dos mesmos

Contas: Projeto OO

```
public abstract class ContaProjeto {
   private String numero;
   private double saldo;
   //...
   public abstract void creditar(double valor);
   public abstract void debitar(double valor);
   public String getNumero() {
      return numero;
   protected setSaldo(double saldo) {
      this.saldo = saldo;
   }
   //...
}
```

Pessoa: Reuso e Subtipos



Pessoa: Projeto OO

```
public abstract class Pessoa {
   private String nome;
   ...
   public abstract String getCodigo();
}
```

Pessoa Física: Projeto OO

```
public class PessoaFisica
                        extends Pessoa {
  private String cpf;
  ...
  public String getCodigo() {
    return cpf;
  }
}
```

Pessoa Jurídica: Projeto OO

```
public class PessoaJuridica
                        extends Pessoa {
  private String cnpj;
  ...
  public String getCodigo() {
    return cnpj;
  }
}
```

```
public class RepositorioPessoasArray {
  private Pessoa[] pessoas;
  ...
  public Pessoa procurar(String codigo) {
    Pessoa p = null;
    boolean achou = false;
    for (int i=0; i<indice && !achou; i++) {
      p = pessoas[i];
      if (p.getCodigo().equals(codigo))
        achou = true;
      else
        p = null;
    }
    return p;
  }
}
```