Unidade 4: Orientação a Objetos

Polimorfismo e Abstração: Herança

Prof. Daniel Caetano

Objetivo: Revisar e Aplicar os Conceitos de Polimorfismo e Abstração com uso de Herança.

Bibliografia: DEITEL, 2005; HOFF, 1996

INTRODUÇÃO

Nas aulas anteriores, foram revistos os conceitos mais importantes da orientação a objetos, além de uma rápida revisão da linguagem o Java.

Nesta aula vamos interligar esses conceitos com o conceito de classe, e aprofundar a idéia de escopo. Para tanto, o primeiro passo é estudar como podemos declarar uma classe e, posteriormente, como ela pode ser extensão de outra.

1. DECLARANDO UMA CLASSE

Primeiramente, lembremo-nos que a declaração de uma variável ou classe é sempre necessária, para que o compilador Java possa **compreender** o que significa aquilo que estamos escrevendo.

De forma similar aos atributos, a declaração de classes é feita da seguinte forma:

```
[escopo] class <nome da classe> [extends <outra classe>] {
        // definições internas da classe
        // tudo  que estiver aqui *faz parte* da classe
        }
```

Por exemplo:

```
public class Aviao extends Aeronave {
        // Atributos de avião
        ...
        // Métodos de avião
        ...
        }
```

O escopo de uma classe pode ser:

abstract	só existe a declaração da classe, ela não é implementada
final	essa classe não pode ter subclasses
public	um objeto de qualquer classe pode criar objetos desta classe
private	apenas objetos de classes do mesmo arquivo .java podem criar objetos desta classe

Exemplos de classes public e private:

Pessoa.java	**SalaDeAula.java**
public class Pessoa	public class SalaDeAula
private class Cabeca	private class Lousa
private class Corpo	

Aqui temos dois arquivos de código: Pessoa.java e SalaDeAula.java. Como visto anteriormente, cada um deles pode conter várias classes: o **Pessoa.java** contém as classes *Pessoa*, *Cabeca* e *Corpo* e o **SalaDeAula.java** contém as classes *SalaDeAula* e *Lousa*.

Como visto anteriormente, cada arquivo de código pode conter apenas **uma** classe pública, e ela deve ter o mesmo nome do arquivo (**Pessoa.java** tem a classe pública *Pessoa* e **SalaDeAula.java** tem a classe pública *SalaDeAula*). Repare, entretanto, que ambos os arquivos possuem várias outras classes privativas (private). O arquivo **Pessoa.java** contém as classes privativas *Cabeca* e *Corpo*. Já o arquivo **SalaDeAula.java** contém a classe privativa *Lousa*. Mas o que isto significa?

Bem, isto significa que um objeto da classe SalaDeAula pode criar objetos da classe Pessoa. Da mesma forma, objetos da classe Pessoa pode criar objetos da classe SalaDeAula. Isso ocorre porque ambas são **classes públicas**.

Por outro lado, um objeto da classe Pessoa pode criar objetos das classes Cabeca e Corpo, já que elas são classes privativas mas estão no mesmo arquivo que a classe Pessoa (arquivo **Pessoa.java**). Entretanto, um objeto da classe Pessoa **não pode** criar objetos da classe Lousa: a classe Lousa é de acesso privativo de objetos de classes do arquivo SalaDeAula.java.

Analogamente, um objeto da classe SalaDeAula pode criar objetos das classes Lousa, já que esta é uma classe privativa, mas está no mesmo arquivo que a classe SalaDeAula (arquivo **SalaDeAula.java**). Entretanto, um objeto da classe SalaDeAula **não pode** criar objetos das classes Cabeca e Corpo: as classes Cabeca e Corpo são de acesso privativo de objetos de classes do arquivo Pessoa.java.

Para que usamos estes dois tipos de classes? Por que não fazer Cabeca ser uma classe pública em um arquivo Cabeca.java? Bem, a razão para isso é a lógica e, eventualmente, o projeto global.

A razão lógica é que, no modelo sugerido, o objeto cabeça não faz sentido num contexto em que não exista o objeto Pessoa. Se no nosso software, por alguma razão, a cabeça pudesse ser arrancada da pessoa e continuasse a existir como um objeto manipulável por outros objetos do software, faria total sentido modelar a classe Cabeca como um objeto público em um arquivo Cabeca.java.

A questão do Projeto Global leva em consideração que, se uma classe ou objeto **não precisa** ser usado fora de um determinado contexto, ele deve sempre ser declarado como privativo àquele contexto (no caso, o contexto é o arquivo Pessoa.java). Isso serve para evitar problemas de nomenclatura.

Por exemplo: suponhamos existam duas classes: Pessoa e Cachorro. Cada uma delas usa um objeto da classe Cabeca, mas a classe Cabeca do Cachorro é diferente da classe Cabeca da pessoa. Uma forma de resolver o problema é chamar uma das classes de CabecaDePessoa e a outra CabecaDeCachorro; a outra forma é tornar a classe Cabeca da pessoa **private** dentro do arquivo da classe Pessoa e tornar a classe Cabeca do cachorro **private** dentro do arquivo da classe Cachorro.

1.1. A diretiva opcional "extends"

A diretiva *extends* serve para implementar uma classe especializada. Nas aulas anteriores já foi falado extensivamente sobre a classe Pessoa. Esta classe pode ser especializada, por exemplo, como uma classe Homem, por exemplo. As declarações de classes para este caso seriam:

Arquivo Pessoa.java

```
public class Pessoa {
        // definicao da classe pessoa
        }
```

Arquivo Homem.java

```
public class Homem extends Pessoa {
        // definicao da classe Homem
        }
```

Isso significa que a classe Homem é uma extensão (extends) da classe Pessoa. Por essa razão, um objeto da classe Homem se comporta, para todos os efeitos, como um objeto da classe Pessoa (possui os mesmos métodos e atributos públicos e protegidos - o atributo protegido veremos a seguir).

Assim, se um objeto da classe pessoa tem um atributo público chamado *idade*, um objeto da classe Homem **também** vai possuir este mesmo atributo, **sem** declará-lo explicitamente. Observe que quando uma classe extende outra, ela **herda** a interface de sua classe mãe.

Note que o significado semântico das classes Pessoa e Homem **é** intimamente ligado. Num sentido físico, um Homem é um conceito especializado, ou seja, mais completo, que o conceito de uma Pessoa.

2. EXERCÍCIOS

O objetivo deste exercício é exercitar a criação de classes e objetos, exercitando e rememorando alguns dos conceitos de criação de classes.

1. Usando JCreator, NetBeans, Eclipse ou qualquer outro programa de sua preferência, inicie um projeto com a classe Pessoa desenvolvida nas aulas anteriores:

Pessoa.java

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 * @author (seu nome)
 * @version 20xxXXxx_001  <<== Versão é muito importante!
 */

public class Pessoa {
   // Variáveis de Instância
   private String nome;
   private int idade;

   /**
    * Construtor para objetos da classe Pessoa
    * @param      nome   o nome da pessoa
    * @param      idade  a idade da pessoa, deve ser entre 0 e 100
    * @return     não há valor de retorno
    */
   public Pessoa(String nome, int idade) { // <= siga sempre uma ordem lógica!
      // Inicializa variáveis de instância
      setNome(nome);
      setIdade(idade);
      }

   /**
    * Retorna um texto que descreve o objeto
    * @return     String contendo a descrição da pessoa
    */
   public String toString() {
      String tmpString;
      tmpString = "Nome: " + getNome() + "\nIdade: " + getIdade() + "\n";
      return(tmpString);
      }

   /**
    * Retorna o nome da pessoa
    * @return     String contendo o nome
    */
   public String getNome() {
      return (nome);
      }

   /**
    * Retorna a idade da pessoa
    * @return     int contendo a idade
    */
   public int getIdade() {
      return (idade);
      }
```

```
    /**
     * Muda o nome da pessoa
     * @param       nome String contendo o nome
     */
    public void setNome(String novoNome) {
       nome = novoNome;
       }

    /**
     * Muda a idade da pessoa
     * @param       int contendo a idade (0 a 100)
     */
    public void setIdade(int novaIdade) {
       if (novaIdade < 0) novaIdade = 0;
       else if (novaIdade > 100) novaIdade = 100;
       idade = novaIdade;
       }

    /**
     * Rotina de Teste da Classe
     */
    public static void main(String[] args) {
       Pessoa joao = new Pessoa ("João Alves", 18);
       Pessoa maria = new Pessoa ("Maria Alves", 17);
       System.out.println(joao);
       System.out.println(maria);
       }
    }
```

2. Vamos agora criar uma nova classe, em um novo arquivo. Crie-a com o nome **Homem**, no arquivo **Homem.java**.

Homem.java

```
/**
 * A classe Homem
*
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public class Homem {
   // Variáveis de Instância

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      }
   }
```

3. Vamos indicar agora que a classe Homem é derivada da classe Pessoa. Para isso, use o termo **extends** no código:

Homem.java

```
/**
 * A classe Homem
 * @author (seu nome)
 * @version 20xxXXxx_001  <<== Versão é muito importante!
 */
```

```
public class Homem extends Pessoa {
   // Variáveis de Instância

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      }
   }
```

4. O modificador **extends** garante que a classe Homem herde todas as características de pessoa. Assim, os métodos estão disponíveis e podem ser usados normalmente, mas antes é preciso que definamos o construtor da classe Homem.

Homem.java

```
/**
 * A classe Homem
 * @author (seu nome)
 * @version 20xxXXxx_001  <<== Versão é muito importante!
 */

public class Homem extends Pessoa {
   // Variáveis de Instância

   /**
    * Construtor da classe
    */
   public Homem(String nome, int idade) {
      super(nome,idade);
      }

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      Homem joao = new Homem ("João Alves", 18);
      System.out.println(joao);
      }
   }
```

Observe uma novidade: a palavra especial **super**. Da mesma forma que a palavra **this** indica que estamos nos referindo ao objeto atual, a palavra **super** significa que estamos nos referindo à classe superior (superclasse), que neste caso é a classe Pessoa. O que este construtor faz é simplesmente pegar os parâmetros passados pelo usuário (nome e idade) e repassá-los para o construtor da classe Pessoa.

5. Vamos, agora, modificar o método toString da classe Homem. Para isso, **redefiniremos** o método na classe homem, da seguinte forma:

Homem.java

```
/**
 * A classe Homem
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */
```

```
public class Homem extends Pessoa {

    /**
     * Construtor da classe
     */
    public Homem(String nome, int idade) {
       super(nome,idade);
       }

     /**
      * Se objeto for impresso como string, mostra informaþ§es
      * @return     texto descrevendo o homem
      */
     public String toString() {
         return "Nome informado pelo homem: " + getNome() +
             "\nIdade informada pelo homem: " + getIdade() + "\n";
     }

    /**
     * Rotina de Teste da Classe
     */
    public static void main(String[] args) {
       Homem joao = new Homem ("João Alves", 18);
       System.out.println(joao);
       }
    }
```

Como os métodos **get**N**ome** e **getIdade** são definidos como públicos na classe Pessoa, eles podem ser acessadas normalmente por qualquer método da classe Homem como se fossem suas próprios métodos.

6. Execute a classe Homem agora (defina-a como principal, se estiver usando o NetBeans, por exemplo). O seguinte texto deve aparecer:

```
Nome informado pelo homem: João Alves
Idade informada pelo homem: 18
```

7. Reproduza os passos 2 a 6 para criar a classe Mulher, fazendo as correções adequadas. Na repetição do passo 6, lembre-se que o nome da mulher é "Maria Alves" e sua idade é 17.

8. Após compilar a classe Mulher, faça com que ela seja a classe principal, e execute-a. O seguinte texto deve aparecer:

```
Nome informado pela mulher: Maria Alves
Idade informada pela mulher: 17
```

9. Muitas mulheres não gostam, entretanto, de informar sua idade real. Vamos modificar a classe mulher para que ela informe uma idade 10% menor. Uma primeira forma de fazer isso é modificando o código do método toString da classe mulher para o código apresentado abaixo:

```
    /**
     * Se objeto for impresso como string, mostra informaþ§es
     * @return     texto descrevendo o homem
     */
```

```
    public String toString() {
        return "Nome informado pela mulher: " + getNome() +
            "\nIdade informada pela mulher: " + getIdade()*0.9 + "\n";
    }
```

Compile e execute o método *main* da classe Mulher e observe que agora ele informa que a idade é 15.3 anos, quando sabemos que, na verdade, a idade dela é 17 anos. Mas existem dois problemas com esse código:

1) Ninguém diz que tem 15,3 anos. Ou a pessoa diz ter 15 ou diz ter 16 anos.
2) Ainda é possível para algum objeto externo (como o objeto homem) ler a idade real da mulher. Vamos resolver um problema de cada vez.

10. O primeiro problema é o arredondamento da idade. Como idade segue o critério de 'truncamento' como arredondamento (ou seja, simplesmente desprezamos o valor das casas decimais), podemos usar um truque: dizer para que a função toString apenas imprima a parte inteira do número. Isso pode ser feito mudando a função toString da seguinte forma:

```
    /**
     * Se objeto for impresso como string, mostra informaþ§es
     * @return     texto descrevendo o homem
     */
   public String toString() {
      return "Nome informado pela mulher: " + getNome() +
            "\nIdade informada pela mulher: " + (int)getIdade()*0.9 + "\n";
   }
```

Observe o indicador **(int)** que foi acrescentado antes da conta da idade. Ele diz para o Java considerar o resultado daquela conta como um inteiro e, por conseqüência, o Java ignora suas casas decimais. Se compilar e executar o método *main* da classe mulher, verá que agora a idade impressa é 15 anos, sem casas decimais.

11. Agora, o segundo problema. Primeiramente, vejamos como é possível ver a idade original? Bem, é simples: basta alterar o método *main* da classe Mulher da seguinte forma:

Mulher.java

```
/**
 * A classe Mulher
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public class Mulher extends Pessoa {
   // Variáveis de Instância

   /**
    * Construtor da classe
    */
   public Mulher(String nome, int idade) {
      super(nome,idade);
      }
```

```
   /**
     * Se objeto for impresso como string, mostra informaþ§es
     * @return     texto descrevendo o homem
     */
   public String toString() {
      return "Nome informado pela mulher: " + getNome() +
             "\nIdade informada pela mulher: " + (int)getIdade()*0.9 + "\n";
   }

    /**
     * Rotina de Teste da Classe
     */
    public static void main(String[] args) {
       Mulher maria = new Mulher ("Maria Alves", 17);
       System.out.println(maria);
       System.out.println("Idade Real: " + maria.getIdade());
       }
    }
```

Agora o programa imprime a idade que ela informa e a idade real. Se este for o comportamento desejado, ótimo! Entretanto, este não é o comportamento desejado **neste** caso. Não queremos que objetos externos tenham como saber qual é a idade real de um objeto da classe Mulher.

12. Como o atributo **idade** é private e não pode ser acessado diretamente, basta fazer modificações adequadas no método getIdade() da mulher, para que ela passe a informar a idade correta.

> NOTA: CASO o atributo idade fosse **public** na classe Pessoa, uma "tentação" que o programador tem é a de torná-lo **private** na classe Mulher, para que possa fazer as alterações que julgar adequadas. Isso, entretanto, **NÃO** é correto, invalidando o contrato de que classes especializadas devem fazer tudo que uma classe-mãe faz. Você pode tornar **public** algo que era private anteriormente, mas nunca tornar **private** algo que era **public**!

Para fazer as modificações, crie o seguinte método getIdade() na classe mulher (note o uso da palavra "super" para fazer uma referência ao método da classe original, a "superclasse" Pessoa):

```
    /**
     * Recupera a idade de uma mulher, mentindo a idade 10% para baixo
     * @return   a idade "corrigida" da mulher.
     */
    public int getIdade() {
        return (int)(super.getIdade()*0.9);
    }
```

Considerando que a correção de idade agora já é feita no método getIdade da mulher, aproveite e modifique o método toString para:

```
   /**
    * Se objeto for impresso como string, mostra informaþ§es
    * @return     texto descrevendo o homem
    */
```

```
   public String toString() {
      return "Nome informado pela mulher: " + getNome() +
             "\nIdade informada pela mulher: " + getIdade() + "\n";
   }
```

13. Compile a classe Mulher, execute seu método *main* e você poderá observar que, em ambos os casos, a idade impressa é 15, o comportamento que desejávamos. O código completo está a seguir:

```
/**
 * A classe Mulher
 * @author (seu nome)
 * @version 20xxXXxx_001  <<== Versão é muito importante!
 */

public class Mulher extends Pessoa {
   // Variáveis de Instância

   /**
    * Construtor da classe
    */
   public Mulher(String nome, int idade) {
      super(nome,idade);
      }

   /**
    * Se objeto for impresso como string, mostra informaþ§es
    * @return     texto descrevendo o homem
    */
  public String toString() {
     return "Nome informado pela mulher: " + getNome() +
            "\nIdade informada pela mulher: " + getIdade() + "\n";
  }

   /**
    * Recupera a idade de uma mulher, mentindo a idade 10% para baixo
    * @return   a idade "corrigida" da mulher.
    */
   public int getIdade() {
       return (int)(super.getIdade()*0.9);
   }

   /**
    * Rotina de Teste da Classe
    */
   public static void main(String[] args) {
      Mulher maria = new Mulher ("Maria Alves", 17);
      System.out.println(maria);
      System.out.println("Idade Real: " + maria.getIdade());
      }
   }
```

14. Um cuidado importante é EVITAR fazer referência às classes derivadas na classe mãe, isto é, fazer com que a classe Pessoa use explicitamente classes que são dela derivadas, como a classe Mulher ou Homem. Isso cria o que se chama de **referência cíclica**, o que pode trazer alguns problemas ao programador. Vez ou outra, entretanto, a referência cíclica não pode ser evitada.

15. Crie agora a classe Main (no NetBeans ela já existe) com o seguinte código:

Main.java

```
/**
 * A classe Main
 * @author (seu nome)
 * @version 20xxXXxx_001  <<== Versão é muito importante!
 */

public class Main {

   /**
    * Rotina Principal do Programa
    */
   public static void main(String[] args) {
     // Vamos criar referências para três pessoas
     Pessoa p1;
     Pessoa p2;
     Pessoa p3;
     p1 = new Pessoa("Alguém",20);
     p2 = new Homem("Homem",20);
     p3 = new Mulher("Mulher",20);

     imprime(p1);
     imprime(p2);
     imprime(p3);
     }

   public static void imprime(Pessoa umaPessoa) {
      System.out.println(umaPessoa);
      }
   }
```

E veja o resultado. Observe que o Java **aceita** objetos do tipo Homem e Mulher guardados como se fossem objetos do tipo Pessoa. Isso ocorre porque, em essência, tanto **Homem é Pessoa** quanto **Mulher é pessoa**, no sentido que tanto objetos do tipo Homem quanto objetos do tipo Mulher seguramente fazem **tudo** que objetos do tipo Pessoa fazem.

16. Experimente agora modificar a classe do atributo p2 para "Homem:

Main.java

```
   /**
    * Rotina Principal do Programa
    */
   public static void main(String[] args) {
     // Vamos criar referências para três pessoas
     Pessoa p1;
     Homem p2;
     Pessoa p3;
     p1 = new Pessoa("Alguém",20);
     p2 = new Homem("Homem",20);
     p3 = new Mulher("Mulher",20);

     imprime(p1);
     imprime(p2);
     imprime(p3);
     }
```

Verifique se ocorre algum erro. Não deve ter ocorrido! Qualquer local onde for possível usar um objeto do tipo Pessoa, será possível usar um objeto do tipo Homem ou do tipo Mulher, já que ambos são Pessoa!

17. Faça agora um outro experimento: modifique o método imprime assim:

Main.java

```
    public static void imprime(Homem umaPessoa) {
       System.out.println(umaPessoa);
       }
```

E execute. Onde aparecem os erros?

Esses erros surgem pelo fato que apesar de ser garantido que Homem faz tudo que Pessoa faz, não é garantido que Pessoa faça tudo que Homem faz, já que Homem é uma versão **especializada** de Pessoa.

3. ESPECIALIZANDO UM COMPONENTE SWING

Vamos agora especializar um componente Swing pronto?

Uma das coisas mais comuns é precisarmos de um campo de entrada de texto (JTextField) formatado de alguma maneira. Para auxiliar nessa tarefa, a Sun Microsystems acrescentou ao Java duas classes importantes: JFormattedTextField e MaskFormatter, que devem ser usadas em conjunto aqui.

JFormattedTextField é simplesmente uma versão "genérica" do JTextField que permite que definamos uma formatação através de um objeto do tipo MaskFormatter. Vamos ver como usar estes dois componentes para criarmos um JCPFField, ou seja, um campo formatado para CPF.

1. No NetBeans, crie um projeto chamado **ElementosUI** . Isso deve criar automaticamente o pacote **elementosui** e a classe **Main** .

2. No pacote **elementosui**, crie agora uma classe java simples e dê a ela o nome de **JCPFField** . O código abaixo será criado:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;

/**
 *
 * @author djcaetano
 */
public class JCPFField {

}
```

3. Vamos modificar essa classe para que ela seja uma extensão de JFormattedTextField, com um construtor que, incialmente, irá apenas chamar o contrutor da superclasse:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import javax.swing.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
    }
}
```

4. Agora chegou a hora de criar o objeto do tipo MaskFormatter, que é responsável por definir as características da formatação.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        // Cria a máscara formatadora
        MaskFormatter formato;
        formato = new MaskFormatter("###.###.###-##");
    }
}
```

5. Bem, o NetBeans vai indicar um erro na linha de criação do MaskFormatter. Isso ocorre porque pode ocorrer um erro de "formato inválido" na criação da máscara de formatação, e o Java lhe obriga a lidar com esse possível erro (ainda que ele nunca vá ocorrer, se você especificar a máscara corretamente).

Assim, será preciso colocar as operações com o objeto MaskFormatter, incluindo sua criação, dentro de um bloco **try**. Como nunca irá ocorrer o erro em questão (porque estamos

definindo correta e estaticamente a máscara), iremos colocar o **catch** apenas para "agradar" o Java, e não iremos inserir nenhum código dentro.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
        } catch (ParseException ex) { }
    }
}
```

6. O próximo passo é modificar o objeto de máscara para que só aceite caracteres numéricos, de 0 a 9. Isso é feito da seguinte forma:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
           // Define os caracteres válidos
           formato.setValidCharacters("0123456789");
        } catch (ParseException ex) { }
    }
}
```

7. Infelizmente nosso trabalho ainda não está finalizado. Como a máscara do tipo MaskFormatter e o campo JFormattedTextField **precisam trabalhar em conjunto**, isto é, um usa métodos do outro, nós precisamos avisar ao JFormattedTextField (superclasse) qual é o objeto MaskFormatter que ele vai usar, través do método setFormatter:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
           // Cria a máscara formatadora
           formato = new MaskFormatter("###.###.###-##");
           // Define os caracteres válidos
           formato.setValidCharacters("0123456789");
           // Instala formatador no JFormattedTextField
           setFormatter(formato);
        } catch (ParseException ex) { }
    }
}
```

8. Da mesma forma que o JFormattedTextField precisa saber do formatador, o formatador **também** precisa saber do JFormattedTextField. Isso pode ser avisado ao formatador através do médodo **install**, que define para o formatador **em que objeto ele está sendo instalado**.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package elementosui;
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 *
 * @author djcaetano
 */
public class JCPFField extends JFormattedTextField {

    public JCPFField() {
        super(); // Inicializa a superclasse
        MaskFormatter formato;

        try {
```

```
            // Cria a máscara formatadora
            formato = new MaskFormatter("###.###.###-##");
            // Define os caracteres válidos
            formato.setValidCharacters("0123456789");
            // Instala formatador no JFormattedTextField
            setFormatter(formato);
             // Instala o JFormattedTextField no MaskFormatter
            formato.install(this);
        } catch (ParseException ex) { }
    }
}
```

9. Finalizada a nossa classe, no mesmo pacote crie um Formulário JDialog com o nome de JExemplo.

10. Na área do projeto, clique com o botão esquerdo do mouse no nome da classe **JCPFField.java** e a arraste para cima do formulário JDialog que você acabou de criar.

11. Selecione o arquivo **JExemplo.java** na área de projeto e pressione SHIFT+F6, para executar a janela. Observe como o campo se comporta.

4. ATIVIDADE (PARA NOTA!)

- Em DUPLA ou TRIO;
- Vale nota AV1a;
- Entrega pelo e-mail: daniel@caetano.eng.br
- Entrega conforme datas apresentadas pelo professor!
- A data que vale é a do e-mail!

A) Modifique a classe **Pessoa** para que ela tenha os seguintes atributos: nome e cpf, ambos String. A validação mínima do CPF deve ser feita, ou seja, ele deve conter 11 caracteres, caso contrário ele deve ser preenchido com **01234567890** (3,0).

B) Crie uma janela JDialog com os campos Nome e CPF, ambos do tipo JTextField, além de um botão **OK**. Adicionalmente, crie um atributo do tipo Pessoa, chamado **aPessoa**, nessa janela. Não se esqueça de executar a janela no método main da classe Main! (2,0).

C) Modifique o construtor da janela para que, logo depois de InitComponents, ela crie a pessoa como indicado a seguir (2,0):

aPessoa = new Pessoa("Fulano","0000");

D) Modifique o botão OK para que ele faça o seguinte, na ordem (Total: 3,0):

a) Imprima (System.out.println ou JOptionPane) o objeto aPessoa (0,5)
b) Pegue os valores dos campos Nome e CPF e modifiquem o objeto aPessoa, usando os métodos aPessoa.setNome() e aPessoa.setCpf() (2,0).
c) Imprima (System.out.println ou JOptionPane) o objeto aPessoa (0,5).

EXTRAS

E) Substitua o campo CPF do tipo JTextField por um do tipo JCPF Field (1,0).

F) Crie as classes pra campos do tipo JDataField e JCEPField (0,5 cada).

5. BIBLIOGRAFIA

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

SUN MICROSYSTEMS: http://java.sun.com/j2se/5.0/docs/api/index.html