Programação Orientada a Objetos

Pacotes, entrada de dados, classes wrappers e encapsulamento

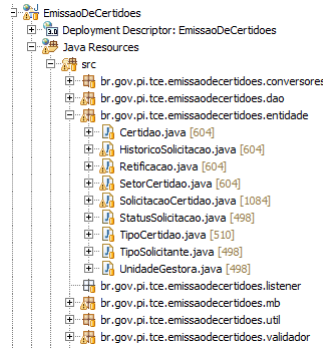
Ely – ely.miranda@ifpi.edu.br

Pacotes

- As classes em Java podem ser agrupadas em pacotes:
  - Semelhante a pastas;
  - Em um pacote podem existir várias classes;
  - Utilizados Java para agrupar classes correlatas;
  - Fornecem uma para o gerenciamento e organização semelhantes a pastas e diretórios;
  - Evitam conflito de nomes;
- Utilizam a notação de pontos em vez de "\" ou "/":
  - Ex: o pacote java.util é na verdade um diretório \java\util\

Pacotes

- Utilizam por convenção notação de “DNS reverso”:



Definindo pacotes

- Pacotes são definidos usando-se a palavra reservada **`package`** no início do código fonte:

  `package` <nomePacote>; // ou

  `package` <nomePacote>.<nomeSubpacote>;

- Ex:

```
package br.ifpi.poo.banco.entidades;


class Conta {
   String numero;
   String saldo;
}
```

Importando pacotes

- Classes de diferentes pacotes não são visíveis entre si e precisam ser importadas;
- Usa-se a palavra reservada **`import:`**

Ex:

```
//importa a classe Vector e JOptionPane
import java.util.Vector;
import javax.swing.JOptionPane;
//Importa todas as classes do pacote
  java.util
import java.util.*
```


Importando pacotes

- Por padrão, todos os programas Java importam o pacote **java.lang:**
  - Classes desse pacote não precisam ser importadas explicitamente;
  - Exemplos de classes que estão nele: String, Integer, System

package x imports

- Em uma classe, deve-se:
  - Primeiro declarar a que pacote a classe pertence;
  - Posteriormente declarar as importações:
  - Ex:

```
package br.ifpi.poo.banco.cadastros;

import br.ifpi.poo.banco.entidades.Conta;
import java.util.Date;

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


Leitura básica

- Necessário criar objeto da classe Scanner:
  - Definir uma variável do tipo Scanner;
  - Importar a classe java.util.Scanner;
- Criar um objeto para ler do teclado:

```
import java.util.Scanner;
...
Scanner sc = new Scanner(System.in);
```

- Ler dados:

```
int x = sc.nextInt();// Lê um int
```

Leitura básica

- Métodos da classe Scanner:
  - nextByte();
  - nextShort();
  - nextInt();
  - nextLong();
  - nextFloat();
  - nextDouble();
  - nextBoolean();
  - next(); //lê uma cadeia de caracteres
  - nextLine(); //lê uma linha

ely.miranda@tce.pi.gov.br

Entrada e saída de dados

```
import java.util.Scanner;

public class SomaNumeros {

   public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);

     System.out.println("Digite um valor para A:");
     int a = sc.nextInt();

     System.out.println("Digite um valor para B:");
     int b = sc.nextInt();
     int soma = a + b;
     System.out.println("A Soma de A + B = " + soma);
   }
}
```

Torna visível o tipo Scanner, semelhante a um #include

Cria um "objeto" para leitura de dados

Lê dois inteiros

Imprime o resultado

ely.miranda@tce.pi.gov.br

Entrada e saída de dados

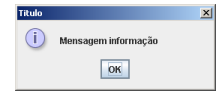
```
import java.util.Scanner;

public class ConcatenaStrings {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.println("Digite o seu nome:");
    String nome =  sc.next();

    System.out.println("Digite o seu sobrenome:");
    String sobrenome =  sc.next();

    System.out.println("Seu nome completo é:" + nome + " " +
                                                  sobrenome));
  }
}
```


Diálogos

- Diálogos são mensagens que tomam a frente da apliação até que se clique em um botão;
- Há mensagens de informação, erro e aviso;
- Há diálogos em que só se exibe uma mensagem, outro que exigem tomadas de decisão ou entrada de dados;
- A classe javax.swing.JOptionPane possui vários métodos estáticos para criar diálogos;

Diálogos - Mensagens

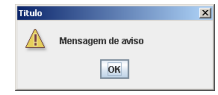
- JOptionPane.showMessageDialog(pai, mensagem, titulo, tipoMensagem)
  - Exibem mensagens para o usuário



Ex:

```
JOptionPane.showMessageDialog(null,
    "Mensagem informação",
    "Titulo",
    JOptionPane.INFORMATION_MESSAGE);
```

- Outros tipos de mensagem:
  - JOptionPane.ERROR_MESSAGE
  - JOptionPane.WARNING_MESSAGE
  - JOptionPane.QUESTION_MESSAGE



Diálogos - Confirmação

- JOptionPane.showConfirmDialog(pai, mensagem, titulo, conjuntoBotoes, tipoMensagem)
  - Exibe um diálogo em que o usuário deve clicar em um botão de acordo com o seu interesse
  - Ex:

```
JOptionPane.showConfirmDialog(null,
  "Pergunta?", "Confirme",
  JOptionPane.OK_CANCEL_OPTION,

  JOptionPane.QUESTION_MESSAGE);
```

Diálogos - Confirmação

- O showConfirmDialog() retorna um inteiro que corresponde ao botão clicado.
- Há constantes pré-declaradas dentro da própria classe que podem ser testadas como resposta:
  - JOptionPane.CANCEL_OPTION
  - JOptionPane.OK_OPTION
  - JOptionPane.CLOSED_OPTION (fechar a janela sem clicar em nada)

Diálogos - Confirmação

```
...
int resposta = JOptionPane.showConfirmDialog(null,
                "Pergunta?", "Confirme",
                 JOptionPane.OK_CANCEL_OPTION,
                 JOptionPane.QUESTION_MESSAGE);

if (resposta == JOptionPane.OK_OPTION) {
   System.out.println("clicou no ok");
} else {
   System.out.println("cancelou ou fechou a janela");
}
...
```

Diálogos – Entrada de dados

- JOptionPane.showInputDialog(pai, mensagem, titulo, tipoMensagem):
  - Exibe um diálogo em que o usuário deve preencher uma caixa de texto e clicar em um botão;
  - É retornada a String preenchida pelo usuário ou num, caso ele não clique no OK;
  - Ex:

```
JOptionPane.showInputDialog(null,
     "Seu nome:", "Pergunta",
      JOptionPane.QUESTION_MESSAGE));
```


Diálogos – Entrada de dados

- O showInputDialog() retorna apeanas String;
- Caso necessário ler um tipo numérico, deve-se fazer uma conversão através de classes "wrappers"
- Ex 1: Lendo Strings:

```
Conta c = new Conta();
c.numero = JOptionPane.showInputDialog(null, "Numero:", "Pergunta");
```

- Ex 2: Lendo números:

```
c.saldo = Double.parseDouble(JOptionPane.showInputDialog(null,
                             "Saldo:", "Pergunta"));
```

Classes wrappers

- São classes que encapsulam tipos primitivos;
- Cada tipo primitivo possui uma “wrapper” class correspondente;
- Usamos principalmente seus métodos de conversão:

```
byte b = Byte.parseByte("10");
int i = Integer.parseInt("100");
double d = Double.parseDouble("12.32");
```

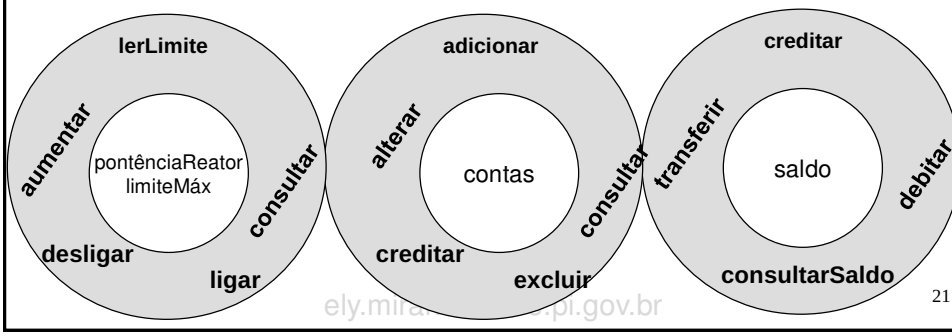

Encapsulamento

- É uma técnica utilizada para esconder detalhes internos de um objeto para o usuário;
- Um objeto deve “encapsular” todo o **estado** e expõe apenas parte do seu **comportamento**:
  - Nem todo elemento de uma classe deve ser visível (e alterável indiscriminadamente);
  - Dados importantes devem ser protegidos do acesso indevido (de programadores);
- Com encapsulamento, obtém-se uma visão simplificada e protegida da classe.

Encapsulamento

- Exemplos:
  - Reator com atributos potência e limite máximo;
  - Banco com várias contas;
  - Conta com o atributo saldo.

  E se o acesso a esses atributos fosse restritos?



Encapsulamento

- Nos três exemplos anteriores, é importante ocultar o estado pois:
  - Nenhum dos atributos mencionados deve ser manipulado diretamente sem validações;
  - Apenas métodos que mantenham integridade devem ter acesso aos dados;
  - A leitura dos dados pode ser liberada, mas a escrita deve ser criteriosa;

Exemplo

```
class Conta {
   String numero;
   double saldo;
   double limite;

   void sacar(double valor) {
      if (valor <= this.saldo)
         this.saldo = this.saldo - valor;
   }
}
```

Usar o método saca, previne que o saldo fique em um estado inconsistente

No entanto os atributos ainda são acessíveis!

Exemplo de violação

```
class Testa {
   public static void main(String args[]) {
      Conta c = new Conta();
      c.limite = 100;
      c.saldo = -2000;
   }
}
```

O acesso direto ao atributo não permite validação

Solução

- Obrigar ao utilizador a chamar o método sacar e não permitir o acesso direto ao atributo;
- Para isso, basta declarar os atributos usando a palavra **private** e os métodos de acesso como **public**:

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

   public void sacar(double valor) {
      if (valor <= this.saldo)
         this.saldo = this.saldo - valor;
   }
}
```


Modificadores de acesso e visibilidade

- Modificadores de acesso são palavras reservadas utilizados para garantir o encapsulamento;
- Podem ser aplicados a classes, atributos, métodos e construtores) individualmente;
- Modificadores existentes:
  - **public:** é visível em qualquer lugar;
  - **protected:** só é visível na mesma classe e em suas subclasses;
  - **private:** só é visível dentro da mesma classe.

Modificadores de acesso e visibilidade

- E quando não se especifica um modificador?
  - Tem-se a visibilidade **package*;***
  - É o padrão;
  - Um atributo ou método com a visibilidade package só é visível em classes do mesmo pacote;

Convenção

- É muito comum que atributos sejam **private**, e quase todos os métodos sejam **public:**
  - Assim, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos;
  - Entretanto, métodos usados apenas como auxiliares em uma classe devem ser privados;

Padrão JavaBean

- Define convenções para a visibilidade dos elementos das classes;
- Características:
  - Construtor público sem argumentos
  - Atributos de **private**.
  - Métodos de acesso (acessors / getter) e/ou de alteração (mutators / setter)
  - Para cada atributo usa-se a convenção **getAtributo()** ou **isAtributo()** (boolean) e **setAtributo()**.

Exemplo de JavaBean

```
public class Loja {
   private String nome;

   public Loja() {
      // Construtor vazio
   }

   public Loja(String nome) {
      this.nome = nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }

   public String getNome() {
      return this.nome;
   }
}
```

Exemplo de leitura (get) e escrita(set)

```
public class Conta {
    private String numero;
    private double saldo;

    public Conta(String numero, double saldoInicial) {
        this.numero= numero;
        //Nota: deve-se validar o saldoInicial antes
        this.saldo = saldoInicial;
    }
    public String getNumero() {
        return numero;
    }
    public void setNumero(String numero) {
        this.numero = numero;
    }
    public double getSaldo() {
        return saldo;
    }
    //demais métodos: creditar, debitar...
}
```

o setSaldo não existe por segurança

Vantagens e consequências do encapsulamento

- Vantagens:
  - Diminuindo a visibilidade de uma classe, ocultamos detalhes de implementação;
  - Facilitam-se alterações na aplicação, pois uma regra só precisa ser modificada em um único lugar;
  - Facilita o aprendizado, pois o mínimo de funcionalidades é exposta;
- Consequência direta:
  - Diminui-se o acoplamento e tornamos nosso código mais utilizável e fácil de manter;

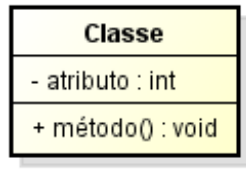
Representação UML

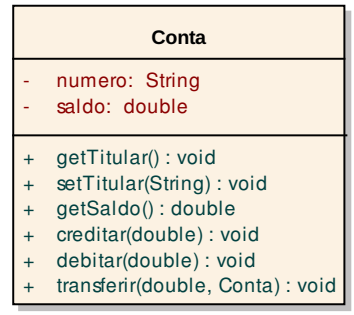
- Em UML, a visibilidade é definida pelos caracteres abaixo:

  + Publica

  # protegida

  - privada





Programação Orientada a Objetos

Pacotes, entrada de dados, classes wrappers e encapsulamento

Ely – ely.miranda@ifpi.edu.br