Aula 6 – POO 1
Encapsulamento

Profa. Elaine Faria
UFU - 2021

Sobre o Material

- Agradecimentos
  - Aos professores José Gustavo e Fabiano, por gentilmente terem cedido seus materiais.

- Os slides consistem de adaptações e modificações dos slides dos professores José Gustavo e Fabiano

Encapsulamento

- Encapsulamento
  - Permite a ocultação/proteção de dados
  - Garante a transparência de utilização dos componentes do software, facilitando:
    - Entendimento
    - Re-uso
    - Manutenção
  - Dados são acessíveis por meio de interfaces bem definidas
    - Interfaces para acesso aos dados permitem o controle sobre como eles são modificados

Encapsulamento

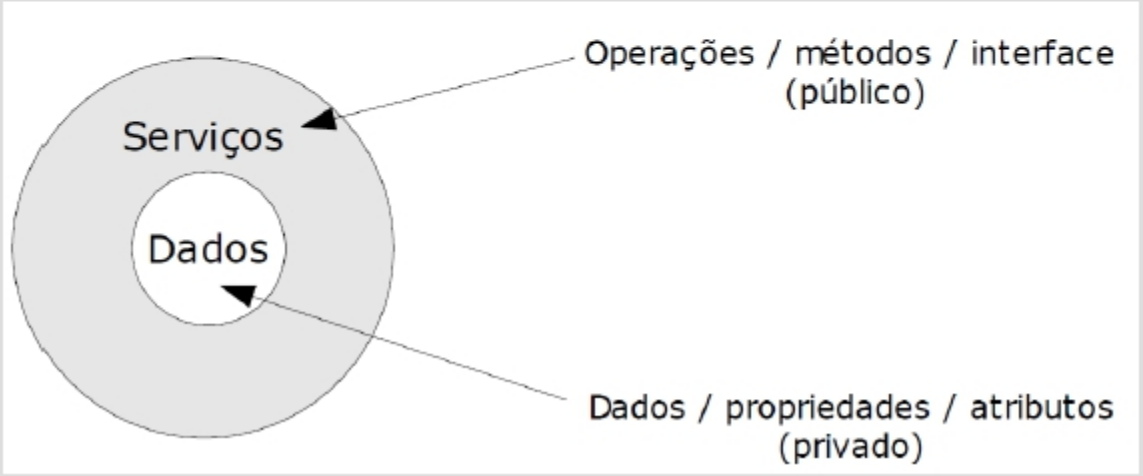
- Encapsulamento
  - Permite que as características e serviços providos por uma classe sejam preservados
  - Possibilita uma visão de caixa preta
    - Não se conhece seu funcionamento internamente, apenas como utilizar
  - Evita que informações de um objeto criado com essa classe sejam corrompidas por objetos externos
  - Conseguido com restrições de acesso

A interface (métodos públicos) de uma classe declara todas as operações acessíveis a outras classes.

Encapsulamento

Programação procedimental:

Programação Orientada a Objetos:



Encapsulamento

- Todo o acesso aos dados é feito por meio de chamada a serviços conhecidos como *getters* e *setters*
- As mudanças na implementação interna do objeto (que preservem a sua interface externa) não afetam o resto do Sistema
- Benefícios
  - Segurança: protege os objetos de terem seus atributos corrompidos por outros objetos.
  - Independência: “escondendo” seus detalhes de implementação, uma classe evita que outras fiquem dependentes de sua estrutura interna.

Encapsulamento

Visão do Cliente

Visão do Programador

Tipos de Visibilidade

Em sistemas orientados a objetos há diferentes tipos de visibilidade de atributos e métodos

- **`public`** → o atributo ou método pode ser invocado (chamado) por objetos de qualquer classe
- **`private`** → o atributo ou método pode ser invocado (chamado) APENAS de dentro do mesmo objeto, e não são visíveis de outros objetos, nem por outros objetos da mesma classe

Tipos de Visibilidade

- **`protected`** → o atributo ou método pode ser invocado (chamado) por objetos do mesmo pacote, e também a partir de objetos DESCENDENTES da classe em questão (mesmo se não estiver no mesmo pacote)
- **`package`** → o atributo ou método pode ser invocado (chamado) por objetos da mesma classe, e também a partir de objetos que estejam no mesmo pacote da classe em questão, sejam eles descendentes ou não. É o valor padrão

Tipos de Visibilidade

- Em java:
  - público: palavra reservada - public
  - privado: palavra reservada - private
  - protegido: palavra reservada - protected
  - package: (default) – sem modificador
- Em UML:
  - público: +
  - privado: -
  - protegido: #
  - pacote: ~

Exemplo - Banco

```
public class Cliente {
    String nome, CPF, telefone;
    int idade;
    Conta conta;
    String usuario,senha;
    void cadastraCliente(String nome1,String CPF1,String
                         telefone1,int idade1,Conta conta1,String
                         usuario1,String senha1) {
        nome = nome1;
        CPF = CPF1;
        telefone = telefone1;
        idade = idade1;
        conta = conta1;
        usuario = usuario1;
        senha = senha1;
    }
}
```

Exemplo - Banco

```
public class Conta {
    int numero;float saldo;String tipo;
    void cadastraConta(int num,float saldo1,String tipo1) {
        numero = num;
        if (saldo1 >= 100) saldo = saldo1;
        else saldo = 100;
        tipo = tipo1;
    }
    boolean sacar(float valor) {
        if (valor <= (saldo+100)) {
            saldo -= valor;
            return true;
        }else
            return false;
    }
    void depositar(float valor) {
        saldo += valor;
    }
}
```

Exemplo - Banco

- No exemplo, todos os atributos das duas classes são package (restrição padrão)
- Dessa forma, é possível a um programador, fazer o seguinte código dentro de um método ***main***

```
Conta c = new Conta();
c.saldo = 50;
c.saldo = c.saldo - 2000;
```

- Isso quebra as regras de negócio do sistema

Exemplo

- Uma boa prática de programação diz que esses atributos devem ser privados
- Dessa forma, classes externas não vão poder acessar os atributos da classe diretamente
  - Sem risco de acesso indevido a esses atributos
- Objetos externos “pedem” para o objeto da classe para ver/modificar suas informações
  - Pedido poderá ou não ser atendido, de acordo com as regras de negócio implementadas por ela

Exemplo

```
public class Conta {
    private int numero;
    private float saldo;
    private String tipo;
    public void cadastraConta(int num,float saldo1,String tipo1) {
        numero = num;
        if (saldo1 >= 100) saldo = saldo1;
        else saldo = 100;
        tipo = tipo1;
    }
    public boolean sacar(float valor) {
        if (valor <= (saldo+100)) {
            saldo -= valor;
            return true;
        }else
            return false;
    }
    public void depositar(float valor) {
        saldo += valor;
    }
}
```

Exemplo – Objetos da Classe

```
public static void main (String args []){
      Conta c1 = new Conta();
      c1.cadastraConta(10, 1000, "Corrente");
      c1.deposita(100);
      c1.sacar(20);
}
```

- Em nenhum momento a classe Principal acessa os atributos *nome*, *número*, *senha* e *tipo*
- Quem faz isso é o método ***cadastraConta***
- Como ele pertence a classe, é o seu próprio objeto quem altera os valores de seus atributos, da maneira que seja melhor/adequada a ele

Exemplo – Objetos da Classe

- Ninguém além desse objeto vai alterar seus atributos
- O objeto decide COMO esses atributos serão acessados e mostrados ao mundo externo, quando chamados
  - Regras definidas na classe
- No exemplo, formatamos como a classe conta exibe o saldo, colocando uma mensagem para o usuário
- Alguns atributos podem nem ser acessíveis por objetos externos

Boa prática de programação

- Listar os **atributos** de uma classe **antes de declarar métodos** da classe
  - Verificação de nomes e tipos das variáveis antes de usá-los nos métodos

Exemplo

```
//Declaracao da classe Circulo.java
public class Circulo
{
    // atributo privado
    private double raio;

    // método alterar raio
    public void setRaio(double r)
    {
        raio = r;
    }

    // método informar raio
    public double getRaio()
    {
        return raio;
    }

    // método exibir dados
    public void exibeDados()
    {
        System.out.println("Raio: " + getRaio());
    }
}// fim da classe
```

Importância do Encapsulamento

- Acesso seguro aos dados dos objetos
- No exemplo: devemos evitar valores de raio negativos
  - Validação pode ser feita no próprio método de acesso

Método de Modificação (*set*)

```
// método alterar raio
public void setRaio(double r)
{
    if (r < 0)
        System.out.println("O raio não pode ser negativo.");
    else
        raio = r;
}
```

Poderíamos melhorar o método e ao invés de imprimir uma mensagem na tela, retornar um valor booleano que indica se foi possível ou não setar o valor do raio

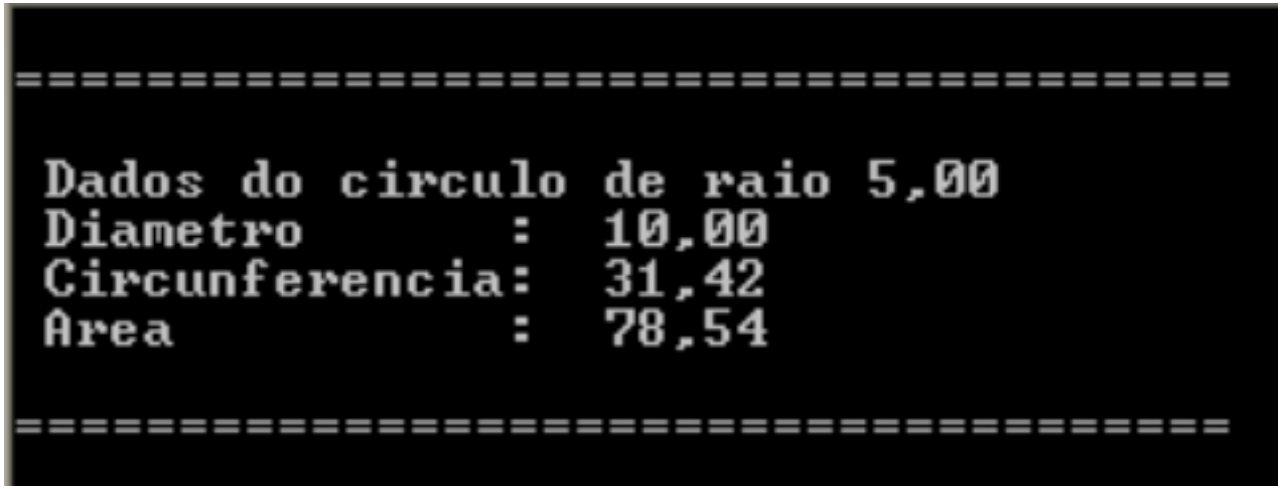
Exercício 1

- Criar uma classe Ponto
  - Atributos: coordenadas (2 dimensões)
    - Valores não podem ser negativos
  - Métodos
    - Inicializar os dados de um ponto
    - Calcular a distancia de outro ponto

Exercício 2

- Modificar a classe Circulo
  - Atributos
    - Nome
    - Centro (Classe Ponto)
  - Métodos
    - Inicializar os dados de um círculo (centro e raio)
    - Calcular diâmetro
    - Calcular área
    - Calcular circunferência
    - Acessar e modificar nome (não pode ser vazio)
    - Exibir os dados

Exibição dos Dados



Exercício 3

- Crie uma classe VEICULO, que tenha como características o modelo, marca, tipo, se ele está ligado (se foi dada a partida) ou não, e carga máxima suportada
- Crie um método que represente a partida sendo dada neste veículo
- Construa uma primeira versão dessa classe SEM UTILIZAR O ENCAPSULAMENTO
- Construa uma segunda versão dessa classe e utilize os conceitos de encapsulamento aprendidos em sala para garantir a segurança no acesso a esses atributos

Encapsulamento

- Implementação de relações com encapsulamento
- Exemplo
  - Considere a seguinte relação bidirecional de 1 para 1.

Encapsulamento

```
class A{

    private B b;
    ...
    public void setB(B aB){
        b=aB;
    }

    public B getB(){
        return b;
    }
    ...

}
```

```
class B{

    private A a;
    ...
    public void setA(A aA){
        a=aA;
    }

    public A getA(){
        return a;
    }
    ...

}
```

Encapsulamento

- Relação unidirecional


```
class A{

    private B b;
    ...
    public void setB(B aB){
        b=aB;
    }

    public B getB(){
        return b;
    }
    ...

}
```

```
class B{
    ...
}
```

Encapsulamento

- Relação de 1 para 2


```
class A{

    private B b1;
    private B b2;
    ...
    public void setB1(B aB){
        b1=aB;
    }
    public B getB1(){
        return b1;
    }

    public void setB2(B aB){
        b2=aB;
    }
```

```
class B{

    private A a;
    ...
    public void setA(A aA){
        a=aA;
    }

    public A getA(){
        return a;
    }
    ...

}
```

Encapsulamento

- Utilização em uma suposta classe cliente

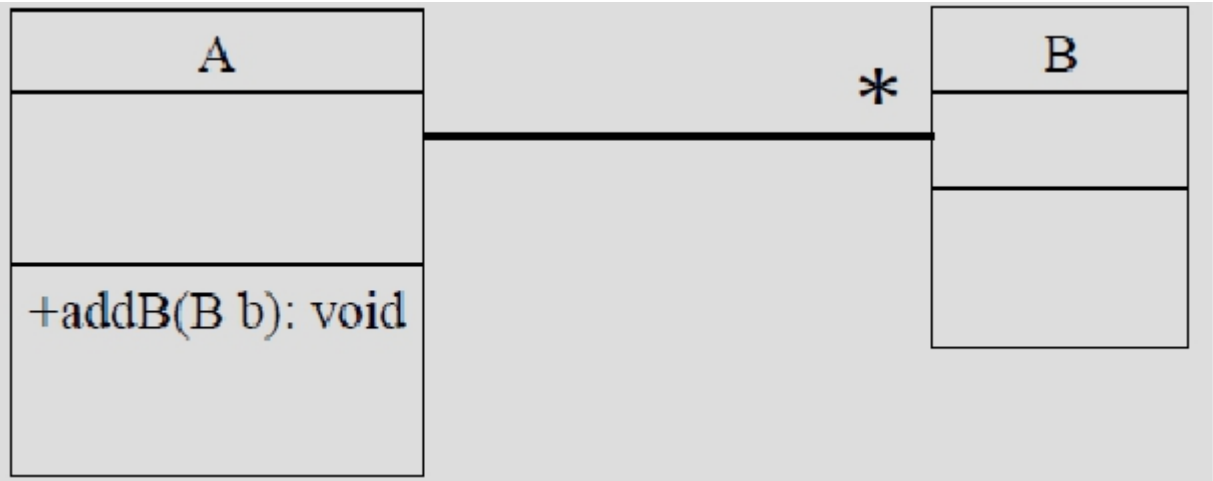
```
class Principal{
        public static void main(String args[]){
                A a = new A();
                B b1 = new B();
                B b2 = new B();
                ...
                a.setB1(b1);
                a.setB2(b2);
                ...
                b1.setA(a);
                b2.setA(a);
                …
        }
}
```

Exercício 4

- Exercício:
  - Modifique a classe A para que as referências à classe B passem a ser armazenadas em um vetor de duas posições.
  - Analise qual o impacto desta modificação na classe cliente, neste caso, a class Principal.

Encapsulamento

- Exemplo: Relações de 1 para n : Diagrama de classes UML (**associação bidirecional**)



Encapsulamento

- Relações de 1 para n : implementação

```
class A{

    private B[] bs;
    private int pos;
    ...
    public A(){
        bs=new B[10];
        pos=0;
    }
    ...

    public void addB(B aB){
        if(pos<bs.length) {
            bs[pos]=aB;
            pos=pos+1;
        }
    }

    public B[] getBs(){
        return bs;
    }
....
}
```

```
class B{

    private A a;
    ...
    public void setA(A aA){
        a=aA;
    }

    public A getA(){
        return a;
    }
    ...

}
```

Encapsulamento

- Utilização em uma classe cliente:

```
class Principal{
      public static void main(String args[]){
            A a = new A();
            B b1 = new B(a);
            B b2 = new B(a);
            …
            B bn = new B(a);
            a.addB(b1);
            ...
            a.addB(bn);
      }
}
```

Encapsulamento

- A classe *ArrayList* implementa a noção de *array* de capacidade variável e "ilimitada".
- Índice começa no zero!
- Importar

```
import java.util.ArrayList;
import java .util.*;
```

Encapsulamento

- **Métodos principais do ArrayList**

void add(int index, Object element) coloca o elemento na posição indicada

void add (Object element) coloca o elemento no fim do *Vetor*

void clear() remove todos os elementos

boolean contains(Object element) retorna *true* se o *Vetor* contém o elemento indicado

Object elementAt(int index) ou

Object get(int index) retorna o elemento que está na posição indicada

Object firstElement() retorna o elemento que está na primeira posição (index=0) do *Vector*

Object remove(int index) remove o elemento que está na posição indicada

boolean remove(Object element) remove a primeira ocorrência do elemento

Object set(int index, Object elelement) substitui o elemento na posição indicada pelo elemento passado pelo argumento

int size() retorna a dimensão atual do *Vetor*

Encapsulamento

- Alteração da implementação usando a classe *ArrayList*

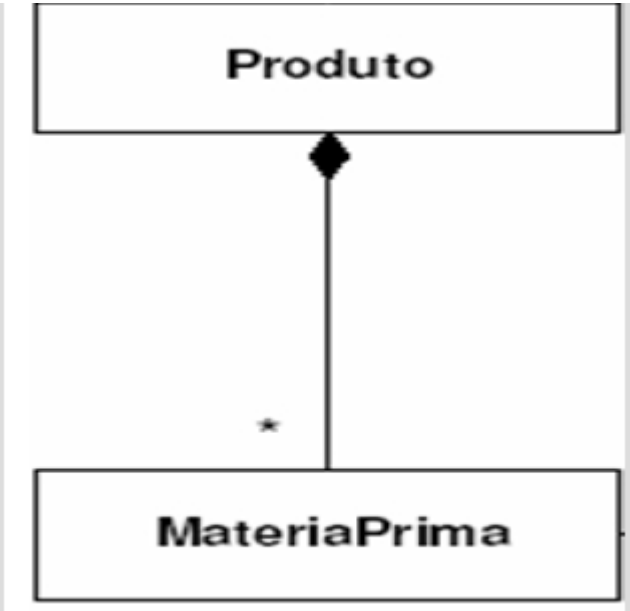
```
class A {
   private ArrayList<B> bs = new
           ArrayList<B>();
   public void addB(B b){
        bs.add(b);
   }
}
```

Encapsulamento

- Qual o impacto desta mudança na classe cliente (neste caso, a **class Principal**) ?

Exercício 5

- Exercício
  - Implemente corretamente, e utilizando encapsulamento, a seguinte relação: **(obs.: unidirecional de produto para matéria prima)**



Referências

- DEITEL, H. M., DEITEL, P. J., **Java: Como Programar,** Bookman, São Paulo, 2002
- BARNES, D. J., KOLLING, M., **Programação Orientada a Objetos com Java**, 2004, ISBN: 8576050129