Universidade Federal de Uberlândia

Programação Orientada a Objetos

Encapsulamento

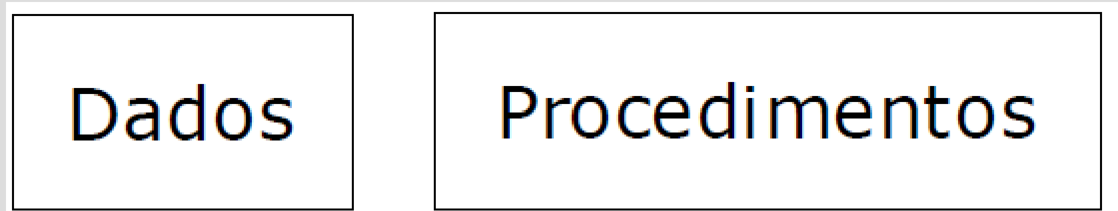
Prof. Fabiano Azevedo Dorça

Encapsulamento

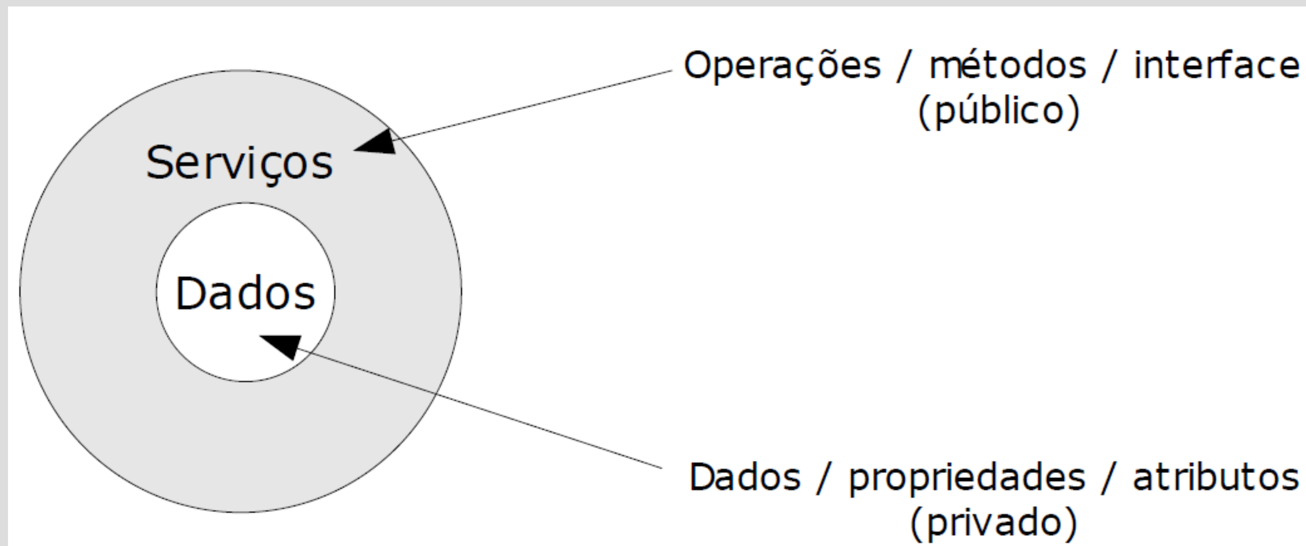
- Os dados são protegidos (**encapsulados**)
- São acessíveis apenas através de **interfaces** bem definidas.
- A definição de interfaces para acesso a dados permite o controle sobre como eles são modificados.

Encapsulamento

Programação procedimental:



Programação Orientada a Objetos:



Encapsulamento

Encapsulamento:

- Ocultação de dados.

- Garante a transparência de utilização dos componentes do software, facilitando:

  - Entendimento
  - Re-uso
  - Manutenção

Encapsulamento

- **<u>Minimiza as interdependências</u>** entre módulos, através da definição de interfaces externas (serviços).

- Classe como “caixa preta”,

- Não se conhece seu funcionamento internamente, apenas como utilizar.

- A <u>interface</u> (métodos públicos) de uma classe declara todas as operações acessíveis a outras classes.

Encapsulamento

- Todo o acesso aos dados é feito através de chamada a serviços conhecidos como *getters e setter*.

- As mudanças na implementação interna do objeto (que preservem a sua interface externa) não afetam o resto do Sistema.

Encapsulamento

Benefícios

a) Segurança: protege os objetos de terem seus atributos corrompidos por outros objetos.

b) Independência: “escondendo” seus detalhes de implementação, uma classe evita que outras fiquem dependentes de sua estrutura interna.

Encapsulamento

Visão do
Cliente

Dados e Estruturas

Operações e
Algoritmos

Visão do
Programador

Encapsulamento

- Elevado grau de abstração.
  - Abstração → esconder e não se preocupar com detalhes!
  - Resultado: uma maior facilidade na:
    - Compreensão;
    - Correção;
    - Manutenção;
    - Reutilização
    - Extensão.

Encapsulamento

Tipos de Visibilidade:

- Em sistemas orientados a objetos, há diferentes tipos de visibilidade de atributos e métodos. Os principais são:
  - Público – objetos de outras classes possuem acesso direto.
  - Privado – o acesso é restrito ao interior da classe.
  - Protegido – o acesso é restrito às classes do mesmo pacote, e às subclasses em qualquer pacote.
  - Pacote – o acesso é restrito às classes do mesmo pacote.

Encapsulamento

- Em java:
  - público: palavra reservada - *public*
  - privado: palavra reservada - *private*
  - protegido: palavra reservada - *protected*
  - package: (default) – sem modificador

  Em UML:
  - público: +
  - privado: -
  - protegido: #
  - pacote: ~

Encapsulamento

- Exercício – Teste a diferença entre as quatro formas de visibilidade.

Encapsulamento

- Para inserir uma classe em um pacote (existente ou não):

```
package teste;
import java.util.*;
class Teste {
    ...

 }
```

Encapsulamento

- Para importar as classes do pacote:

```
package Teste2;
import teste.*; OU import teste.Teste;
class Horario {
   ...
}
```

Encapsulamento

Exemplo:

- Getters & Setters (interface de acesso)

```
public class Funcionario extends Pessoa{
        private float salario;
        private int horasExtras;
        public boolean setSalario(float salario) {
                if (salario > 0) {
                        this.salario = salario;
                        return true;
                }
                else return false;
        }
        public float getSalario() {
                return this.salario;
        }
}
```

Encapsulamento

- Utilização dos getters & setters

```
...
Funcionario f;
f = new Funcionario();
f.setSalario(1000);
…
System.out.println(f.getSalario());
```

Encapsulamento

- Os atributos *private* não são visíveis na subclasse, apesar de serem herdados.
- Eles devem ser acessados através de *getters* e *setters* herdados.

Encapsulamento

- A visibilidade dos métodos sobrescritos pode mudar, mas, apenas para dar mais acesso.

- Por exemplo:

  - um método declarado na superclasse como *protected* pode ser redefinido *protected* ou *public*, mas não *private* ou com visibilidade de pacote.

Encapsulamento

- Implementação de **relações** com encapsulamento

- Exemplo
  - Considere a seguinte relação **bidirecional** de 1 para 1.

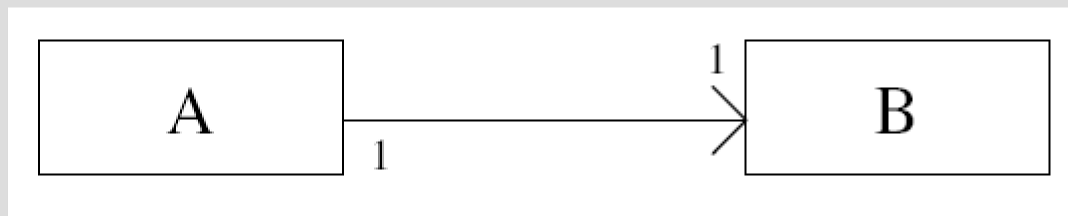
Encapsulamento

```
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

Relações 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

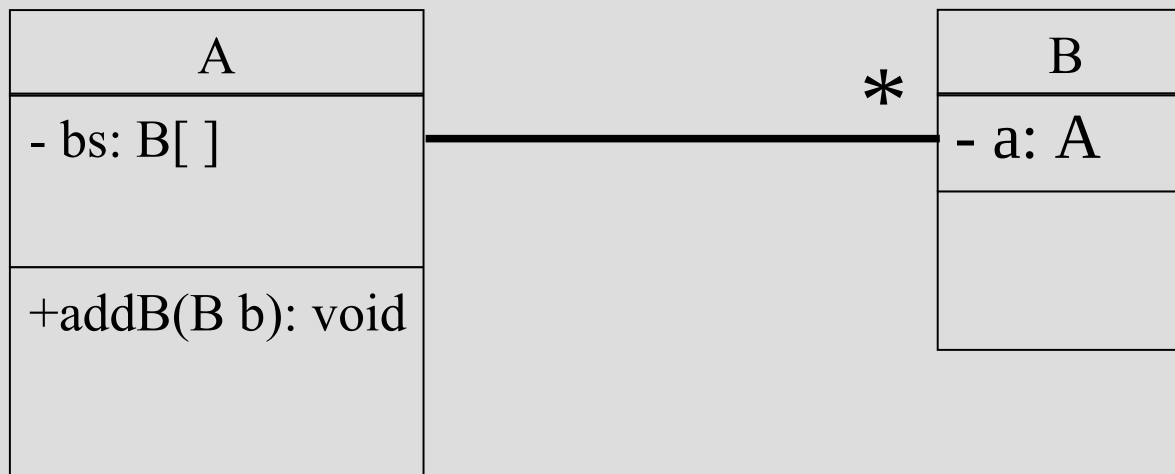
```
class Principal{
    public static void main(String args[]){
        A a = new A();
        B b1 = new B();
        B b2 = new B();
        ...
        a.setB1(b1);
        a.setB2(b2);
        ...
        b1.setA(a);
        b2.setA(a);
        …
    }
}
```

Encapsulamento

- 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:

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

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

void clear() remove todos os elementos

boolean contains(Object element)retorna *true* se o *Vector* 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 actual do *Vector*

Encapsulamento

- Alteração da implementação usando a classe ArrayList

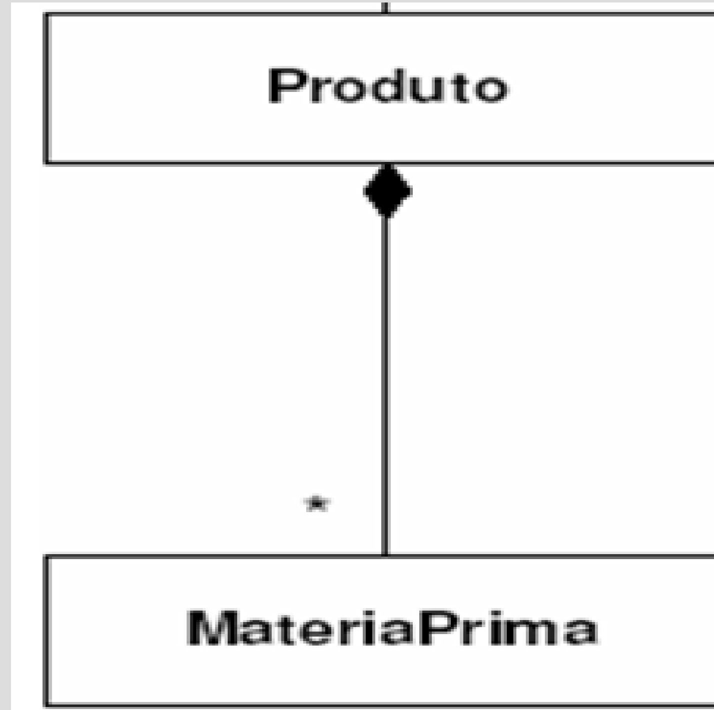
```
class A {
    private ArrayList<B> bs = new ArrayList<B>();

    public void addB(B b);
        bs.add(b);
    }

}
```

Encapsulamento

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

Encapsulamento

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



Referências

BOOCH, G. Object-Oriented Analysis and Design with Applications, 3a Edição. Addison-Wesley, 2007.

BOOCH, G., RUMBAUGH, J., JACOBSON, I. UML, Guia do Usuário. Rio de Janeiro: Campus, 2000.

DEITEL, H. M.; DEITEL P. J. Java: Como Programar, 6a. Edição. Pearson, 2005. (Livro Texto)

FOWLER, M. UML Essencial, 2a Edição. Bookmann, 2000.

HORSTMANN, C.; CORNELL, G. Core Java 2 - Fundamentals, 7a. Edição. Prentice Hall, 2004.

LARMAN, C. Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos. Porto Alegre: Bookmann, 2001.

RUMBAUGH, J.; BLAHA, M. Modelagem e Projetos Baseados em Objetos com UML 2, 1a Edição. Editora Campus, 2006