



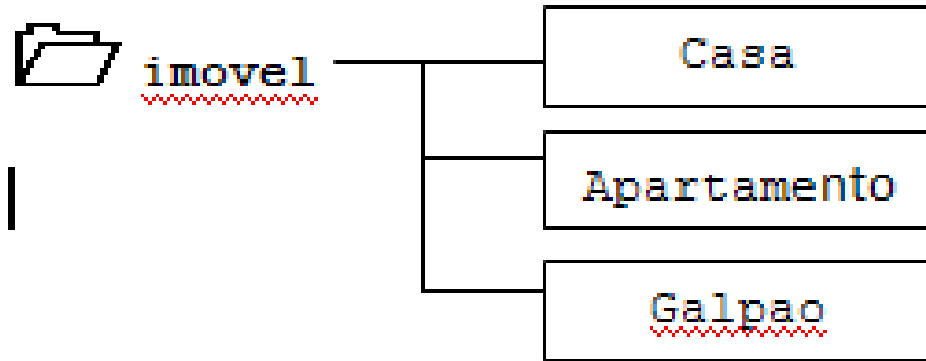
Pacotes e Modificadores de Acesso

Prof. Msc Gustavo Molina

CONCEITO DE PACOTE (*PACKAGE*)

- Grupos de classes que mantêm uma relação entre si.
- Quando não se declara um pacote para uma classe, dizemos que o pacote associado é o *default*.
- Estes conjuntos de classes são determinados através da codificação de uma linha no topo de cada arquivo, indicando a qual pacote pertencem as classes ali declaradas. Se nenhuma linha é inserida assume-se que todas as classes pertencem a um pacote só.

Exemplo 1



Será criado um pacote chamado `imovel` que conterá as classes que representam diversos tipos de imóveis (Casa, Apartamento e Galpao).

O pacote chamado `imovel` fica armazenado em uma pasta com o mesmo nome.

E as demais classes devem estar dentro deste diretório.

Exemplo 1

Imagine que a classe `imovel` está localizada no pacote `nacional.saopaulo.imovel` → está dentro do diretório `nacional/saopaulo/imovel`

Na classe `Casa`, que está dentro de `imovel`, devemos codificar:

```
package nacional.saopaulo.imovel;
```

```
class Casa {
```

```
•
```

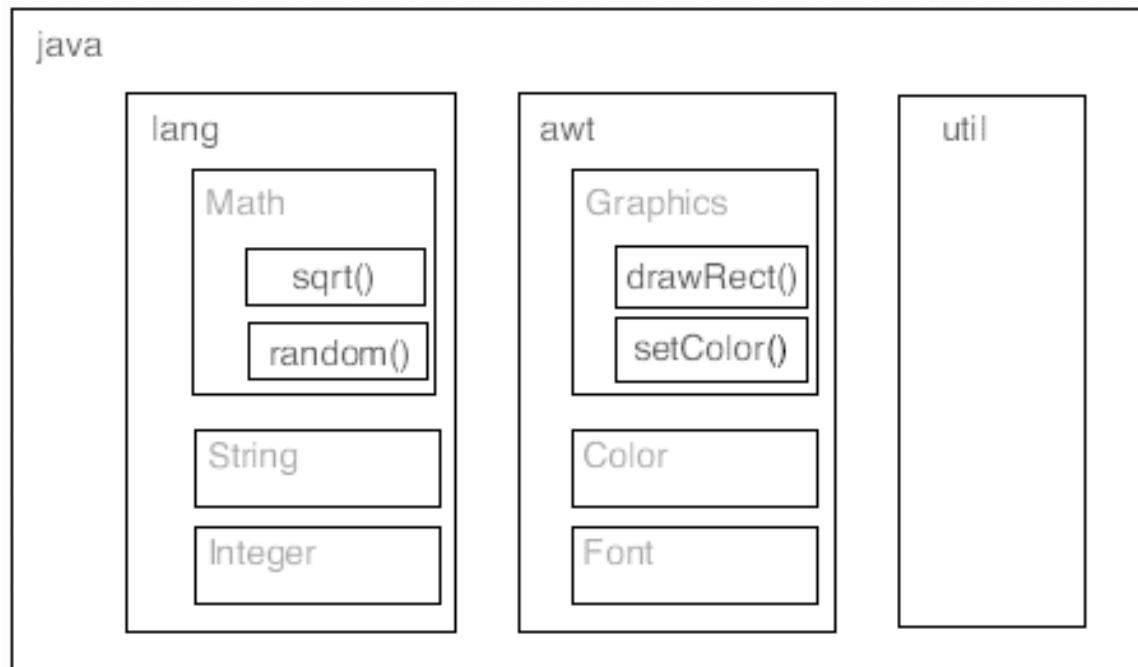
```
•
```

```
•
```

```
}
```

PADRÃO *SUN* PARA NOMENCLATURA DOS PACOTES

- Devem ser formados apenas por letras minúsculas.
- A linguagem Java é totalmente organizada em pacotes.
- A versão SE do Java tem 165 pacotes diferentes e 3278 classes.



ACESSO A UMA CLASSE FORA DO PACOTE

- Uso do nome longo:

```
java.lang.Math.sqrt()
```

- ou usar a instrução `import` no início da classe:

```
import java.lang.Math;
```

- Não importar um pacote não significa que não se podem usar as classes de outro pacote. Significa que a forma de se chamar as classes deverá ser feita da forma longa, como no exemplo ilustrado acima.

ORDEM

- ➔ Primeiro, aparece uma (ou nenhuma) vez o `package`;
- ➔ depois, pode aparecer um ou mais `import`;
- ➔ e, por último, as declarações de classes.

ENCAPSULAMENTO

→ o encapsulamento “protege” os membros declarados em uma classe e com isso permite a restrição de acesso a certas partes de uma classe.

→ **Situação 1:** o atributo `saldo` só é alterado pelos métodos `saque()` e `deposito()`.

```
class ContaCorrente {
    String nome;
    double saldo;

    ContaCorrente (double saldo){
        this.saldo = saldo;
    }

    void deposito (double valor) {
        saldo = saldo + valor;}

    void saque (double valor) {
        saldo = saldo - valor;}
}
```


NO MÉTODO `main()`

```
public static void main(String [] args) {  
    ContaCorrente cta;  
    cta = new Contacorrente(1000);  
  
    cta.deposito(200);  
    cta.saque (100);  
}
```

- o saldo só é alterado através do construtor e dos métodos da classe `ContaCorrente`.
- Porém, é possível alterar o valor do `saldo`, fora da classe `ContaCorrente`. Veja:

SITUAÇÃO 2

```
class ContaCorrente {
    String nome;
    double saldo;


    ContaCorrente (double saldo){
        this.saldo = saldo;
    }

    void deposito (double valor) {
        saldo = saldo + valor;}

    void saque (double valor) {
        saldo = saldo - valor;}
}

.
.
public static void main(String [] args) {
    ContaCorrente cta;
    cta = new Contacorrente(1000);

    cta.deposito(200);
    cta.saque (100);
    cta.saldo = 5000;
}
```



CARACTERÍSTICAS DO ENCAPSULAMENTO

➔ Programar pensando sempre na interface!! Pensar em como os usuários irão utilizar os métodos e não em como os métodos funcionam.

ANALOGIA: Quando você dirige um carro, o que importa são os pedais e o volante (**interface**) e não o motor que você está usando (**implementação**). É claro que um motor diferente pode dar melhores resultados, mas **o que ele faz** é a mesma coisa que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem os usando da mesma maneira).

CARACTERÍSTICAS DO ENCAPSULAMENTO

- ➔ Programar tendo em vista a **interface** é também chamado de “*programming in the large*”.
- ➔ Programar tendo em vista **implementação** (codificação) é chamado de “*programming in the small*”.
- ➔ Com encapsulamento você será capaz de criar componentes de software reutilizáveis, seguros, fáceis de modificar.

MODIFICADORES DE ACESSO

public

Atributos e métodos declarados como `public` são visíveis e, portanto, acessíveis a qualquer classe. Esta é a forma menos rígida de encapsulamento, que na verdade significa não encapsular.

private

Atributos e métodos declarados como `private` são visíveis e, portanto, acessíveis apenas na classe onde foram declarados. Ou seja, só podem ser acessados, modificados e/ou executados dentro da própria classe. Esta é a forma mais rígida de encapsulamento.

MODIFICADORES DE ACESSO

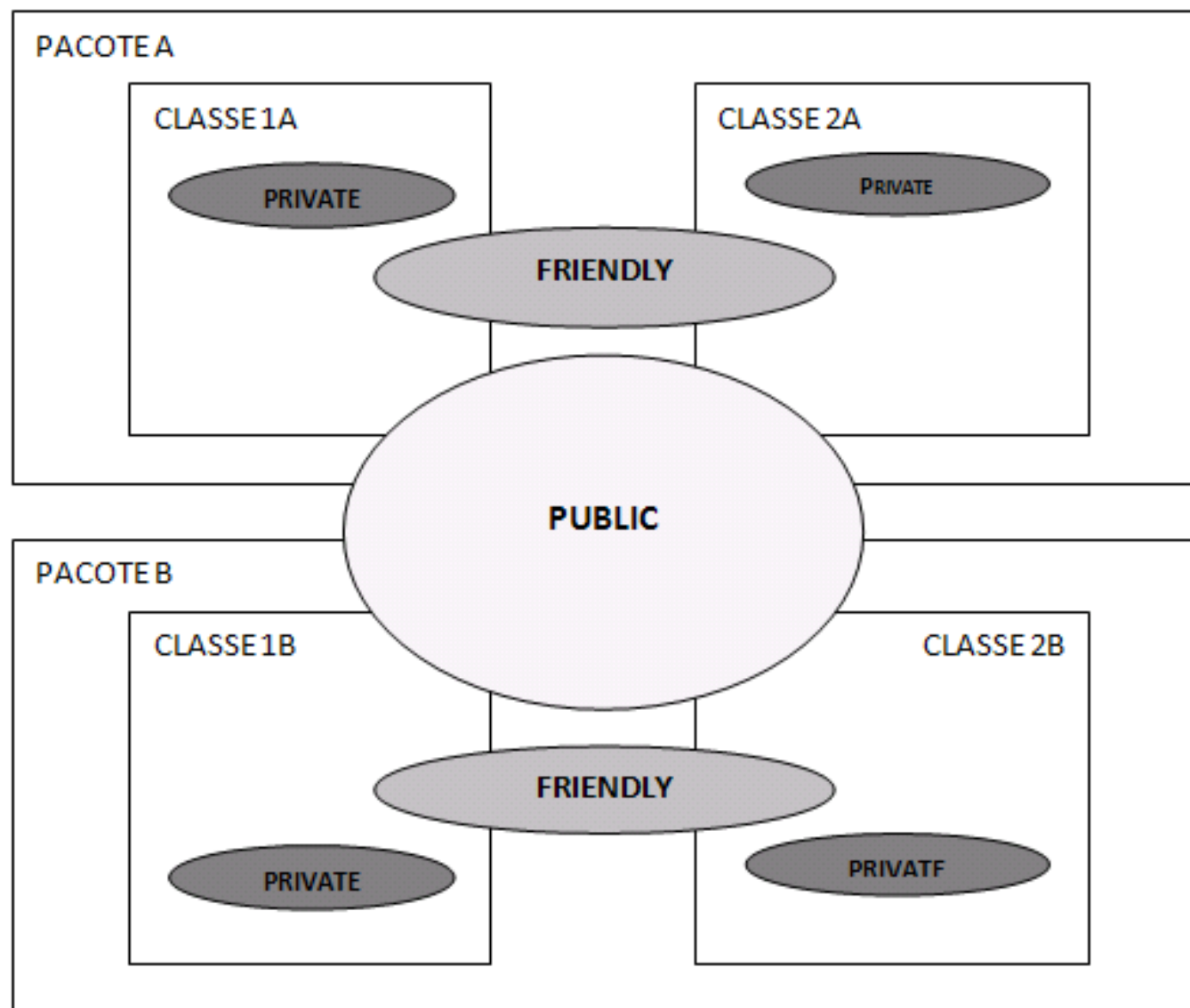
protected

Atributos e métodos declarados como `protected` são visíveis e, portanto, acessíveis na classe onde foram declarados e nas suas subclasses. O conceito de subclasse será estudado em Herança.

Sem modificador de acesso

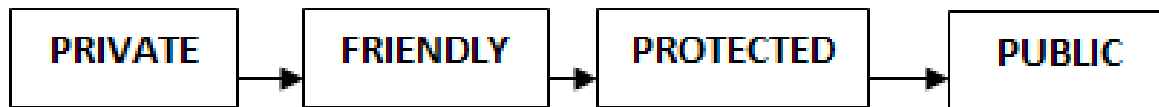
Chamados de “*package*” ou “*friendly*”, os membros declarados sem nenhum modificador de acesso são visíveis e, portanto, acessíveis apenas às classes pertencentes ao mesmo pacote.

Graficamente:



MODIFICADORES DE ACESSO

Assim, temos do modificador de acesso MAIS restritivo para o MENOS restritivo:



SITUAÇÃO 3 – COM ENCAPSULAMENTO

```
class ContaCorrente {  
    private String nome;  
    private double saldo;  
  
    ContaCorrente (double saldo){  
        this.saldo = saldo;  
    }  
  
    void deposito (double valor) {  
        saldo = saldo + valor;}  
  
    void saque (double valor) {  
        saldo = saldo - valor;}  
}
```

➔ O saldo e o nome passaram a ser do tipo `private` – só são visíveis e acessíveis à classe `ContaCorrente`.

➔ Os métodos `deposito()` e `saque()` são do tipo `public` – visíveis e acessíveis à qualquer classe de qualquer pacote

SITUAÇÃO 3 – COM ENCAPSULAMENTO

→ seria impossível realizar a alteração do `saldo` fora da classe `ContaCorrente`

```
.  
public static void main(String [] args) {  
    ContaCorrente cta;  
    cta = new Contacorrente(1000);  
  
    cta.deposito(200);  
    cta.saque (100);  
    cta.saldo = 5000;  
}
```



ERRO!!

Setters & Getters

➔ São métodos públicos para acessar atributos privados

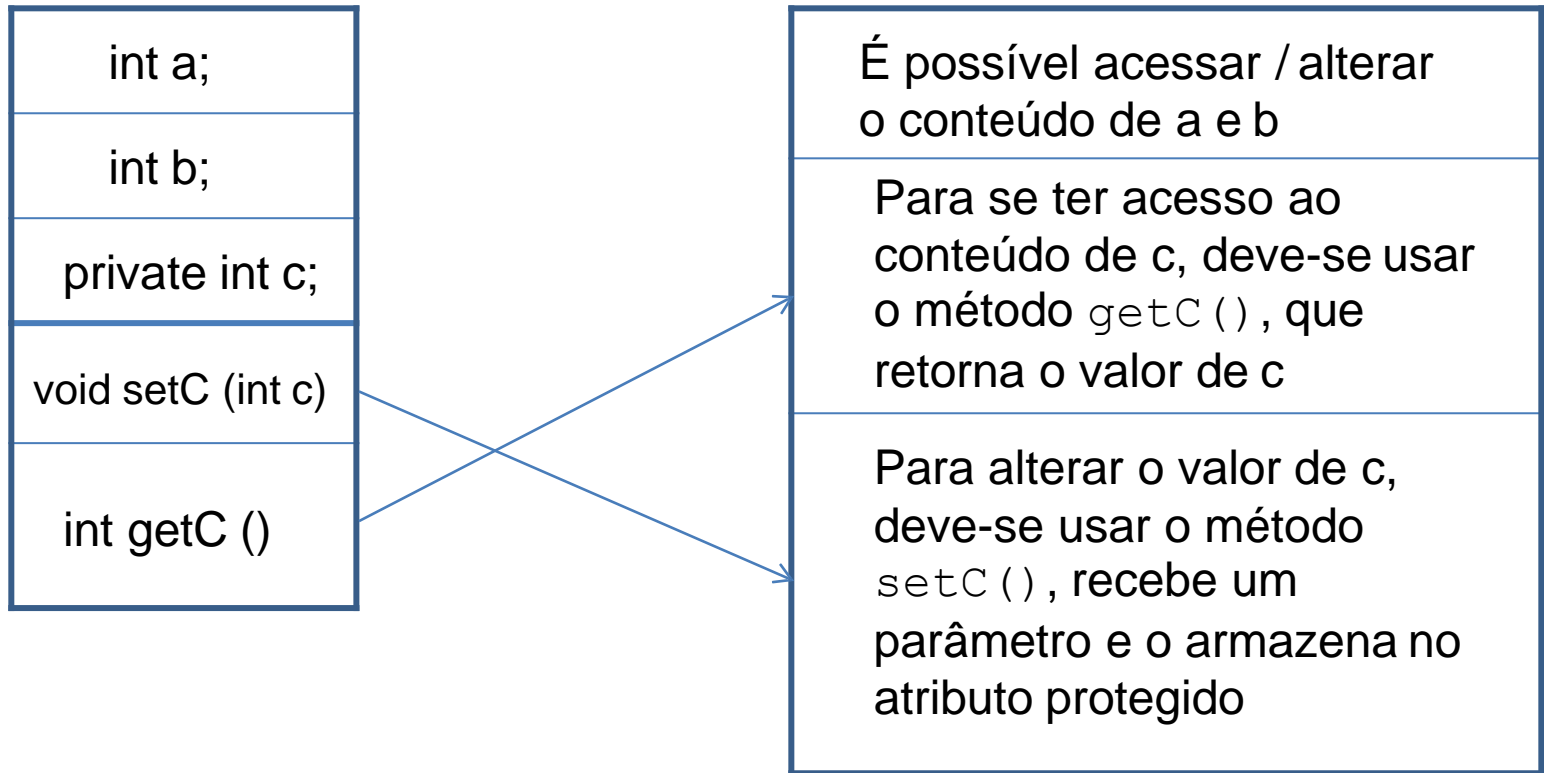
```
public class Teste {
    int a;
    int b;
    private int c;

    void setC (int c){
        this.c = c;}

    int getC (){
        return c;}
}

public class TestaTeste {
    public static void main(String[] args) {
        Teste obj = new Teste();
        obj.a = 10;
        obj.b = 20;
        // obj.c = 30; não pode ser acessado diretamente
        obj.setC(30);
        System.out.println("a="+ obj.a + " b=" + obj.b+ " c="+
obj.getC());
    }
}
```

Setter & Getter



Setters & Getters

Métodos de acesso → `getXXX()` – permitem o acesso à algum atributo de uma classe

Métodos modificadores → `setXXX()` – alteram algum atributo de uma classe

Padrão adotado, pelos programadores em Java, para estes métodos é `setNomeAtributo()` e `getNomeAtributo()`.

Exercício 1

Crie uma classe em Java que:

- a) contenha os atributos nome, idade e altura;
- b) encapsule os atributos;
- c) crie um construtor para armazenar os dados passados como parâmetro nos atributos da classe (todos os atributos)
- d) crie Getters para todos os atributos
- e) crie um método main() que crie o objeto da classe, já armazenando algum valor inicial nos seus atributos através do construtor e mostre os valores que estão nos atributos.

Exercício 2

Crie uma classe Retangulo. A classe tem atributos largura e altura, ambos sendo do tipo double. A classe deve ter métodos que calculam o perímetro (perimetro()) e a área (area()) do retângulo. A classe tem métodos set e get para a largura (largura) e a altura (altura). Escreva um programa em Java para testar a classe Retangulo, de forma que os valores da altura e da largura sejam fornecidos pelo teclado e a área e o perímetro sejam exibidos.