Modularização

- **Tópicos:** modularização/acoplamento, coesão, pacotes em Java, uso de pacotes, nomes qualificados, importação e visibilidade de pacotes.
- **Objetivos:** exercitar a definição de pacotes e a construção de programas modulares que acessam elementos definidos em pacotes externos
- **Pré-requisitos:** classes e objetos, conhecimento básico sobre programação estruturada, herança.

Roteiro

- Modularização / Acoplamento, Coesão
- Pacotes em Java - *Packages*
- Importação de Pacotes
- Pacotes Anônimos
- Visibilidade de Pacotes
  - Visão Interna do Pacote
  - Visão Externa do Pacote
- Variável de ambiente - *classpath*
- Utilitário Jar
- Exemplos de Aplicação

Modularização (I)

- Dois princípios fundamentais regem a construção de sistemas computacionais complexos: abstração e modularização.
- Abstração nos permite entender e analisar o problema concentrando-se nos aspectos relevantes e ignorando detalhes irrelevantes.
- Modularização nos permite projetar e construir um sistema computacional a partir de partes menores chamados módulos ou pacotes.

Modularização (II)

- Durante o projeto de construção de um sistema, nós criamos uma estrutura modular para o programa.
- Se os módulos que implementam o programa correspondem às abstrações descobertas durante a análise do problema, então o sistema será mais facilmente compreendido e gerenciado.

Modularização (III)

- O objetivo do projeto de um sistema é encontrar uma **decomposição modular** que seja adequada para a sua implementação.
- Uma decomposição modular é considerada boa quando ela é composta por módulos que sejam independentes o máximo possível uns dos outros (isto é, fraco acoplamento entre os módulos).
- Cada módulo deve "esconder" uma decisão particular de projeto; de forma que, se essa decisão for mudada, apenas o módulo que "conhece" a decisão tomada será modificado. Os outros módulos permanecem inalterados.

Modularização (IV)

- Uma decomposição modular (ou fatoração) de boa qualidade deve permitir:
  1. Fácil compreensão da estrutura do sistema como um todo e de cada uma de suas partes isoladamente;
  2. Desenvolvimento, uso e manutenção dos pacotes com o máximo de independência.
- Recomenda-se os critérios de fraco acoplamento entre os diferentes pacotes e alta coesão entre os elementos de um mesmo pacote.

Modularização (V)

- **Acoplamento:** representa uma medida das conexões entre grupos distintos. Grupos fracamente acoplados conectam-se apenas através de interfaces que sejam simples, visíveis e bem definidas.

- **Coesão:** representa a intensidade dos relacionamentos entre os elementos de um mesmo grupo. Todos os elementos de um grupo altamente coeso colaboram intensamente entre si para realizar uma tarefa; com um mínimo de dependência com relação a elementos estranhos ao grupo.

Modularização (VI)

- Se o projeto é composto por módulos altamente independentes então:
  1. Os módulos formam a base para a atribuição de trabalho para membros do time de programação.
  2. O sistema pode ser mais facilmente evoluído e reparado porque os efeitos das modificações são restritos a módulos individuais.

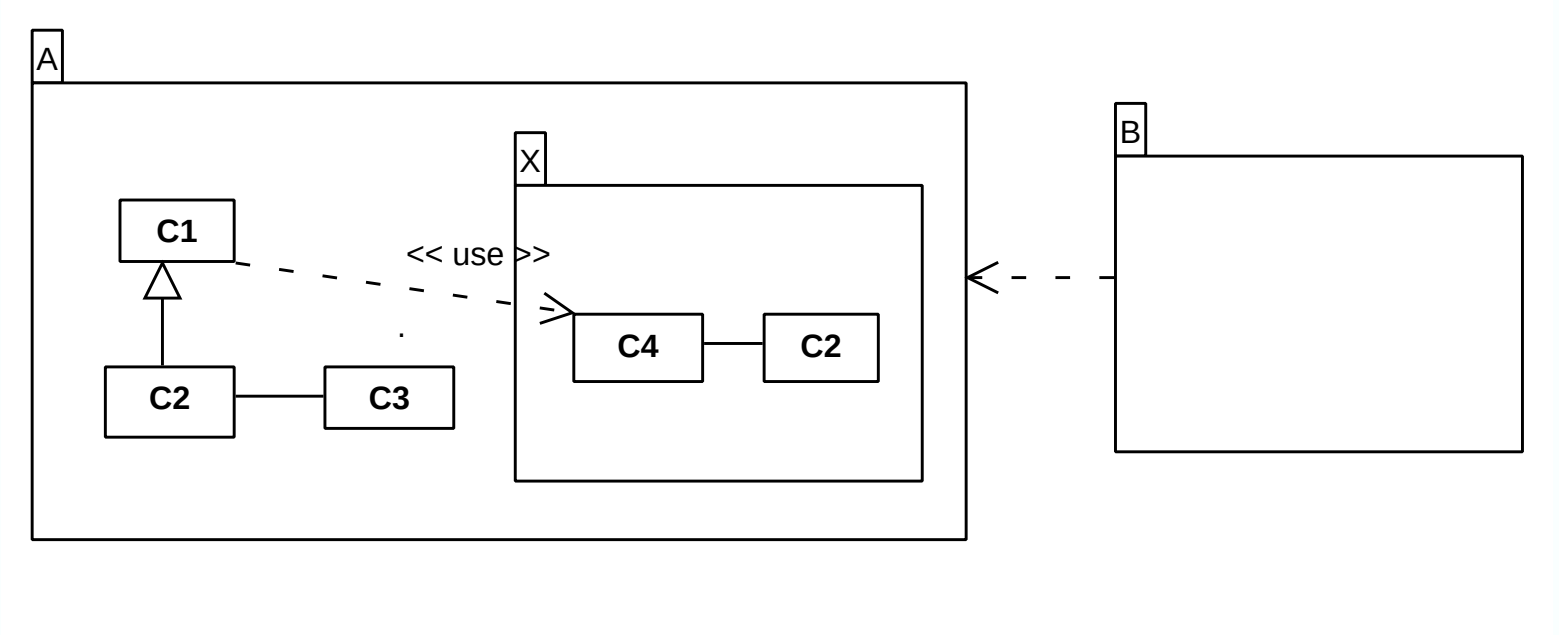
Pacotes em Java (I)

- Pacote (**package**) é um mecanismo para organizar classes e interfaces relacionadas em grupos.
- Um pacote pode estar contido dentro de outro pacote e assim por diante.
- Um sistema computacional pode ser visto como o único pacote de alto nível que contém todos os elementos do sistema, inclusive outros pacotes.
- Um pacote é um encapsulamento lógico (não físico), isto é, qualquer classe pode especificar que ela pertence a um pacote em particular.

Pacotes em Java (II)

- Uma classe que não especifica o pacote ao qual ela pertence é colocada num pacote anônimo.
- Um pacote cria um espaço de nomes.
- Em Java, um pacote é representado por um diretório.
- O escopo de pacotes é uma alternativa mais elegante para a solução de "friends" dada por C++.
- A visibilidade "friends" de C++ fornece acesso a atributos e métodos privados de uma classe, implicando em quebra de encapsulamento.

Exemplo de Uso de Pacotes (I)

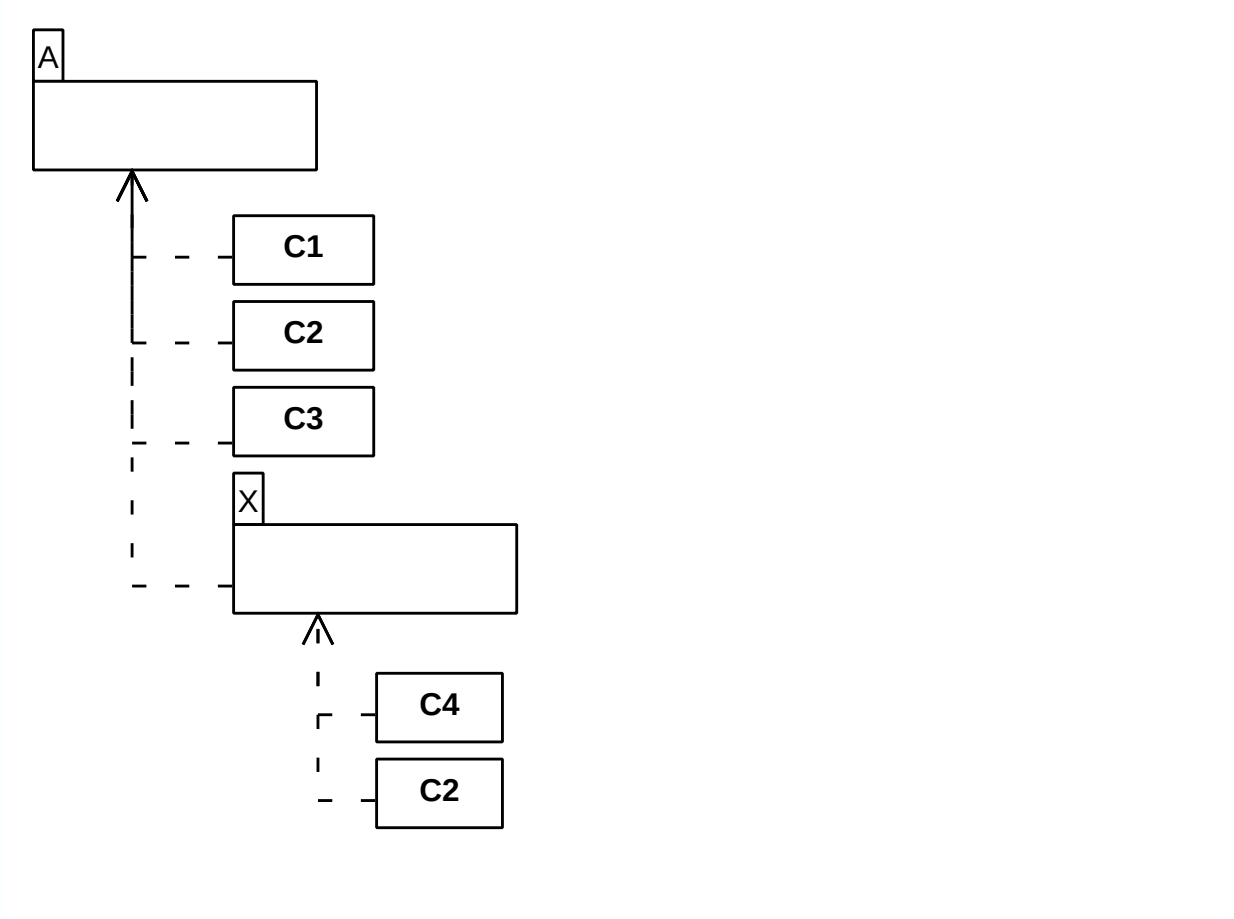


- OBS1.: C1 depende de C4.
- OBS2.: O pacote B depende do pacote A.

Exemplo de Uso de Pacotes (II)

- O pacote A contém as classes C1, C2 e C3; e o pacote X, que é usado pela classe C1.
- O pacote X contém as classes C2 e C4.
- Sobre o pacote B, sabemos apenas que ele depende do pacote A.

Estrutura em Árvore do Pacote A (I)



Nomes Qualificados

- O nome completo (**qualificado**) da classe C2, do pacote X, pertencente ao pacote A, é: **A.X.C2**.
- Note que o nome C2 é usado dentro do pacote A e dentro do pacote X.
- Como um novo espaço de nomes é criado, pode haver nomes repetidos de classes em pacotes diferentes.

Definição de Pacotes em Java (I)

- Exemplo : Package A

```
//arquivo A/c1.java
   package A;
   public class c1 {...}

//arquivo A/c2.java
   package A;
   public class c2 {...}

//arquivo A/c3.java
   package A;
   class c3 {...}
```

Definição de Pacotes em Java (II)

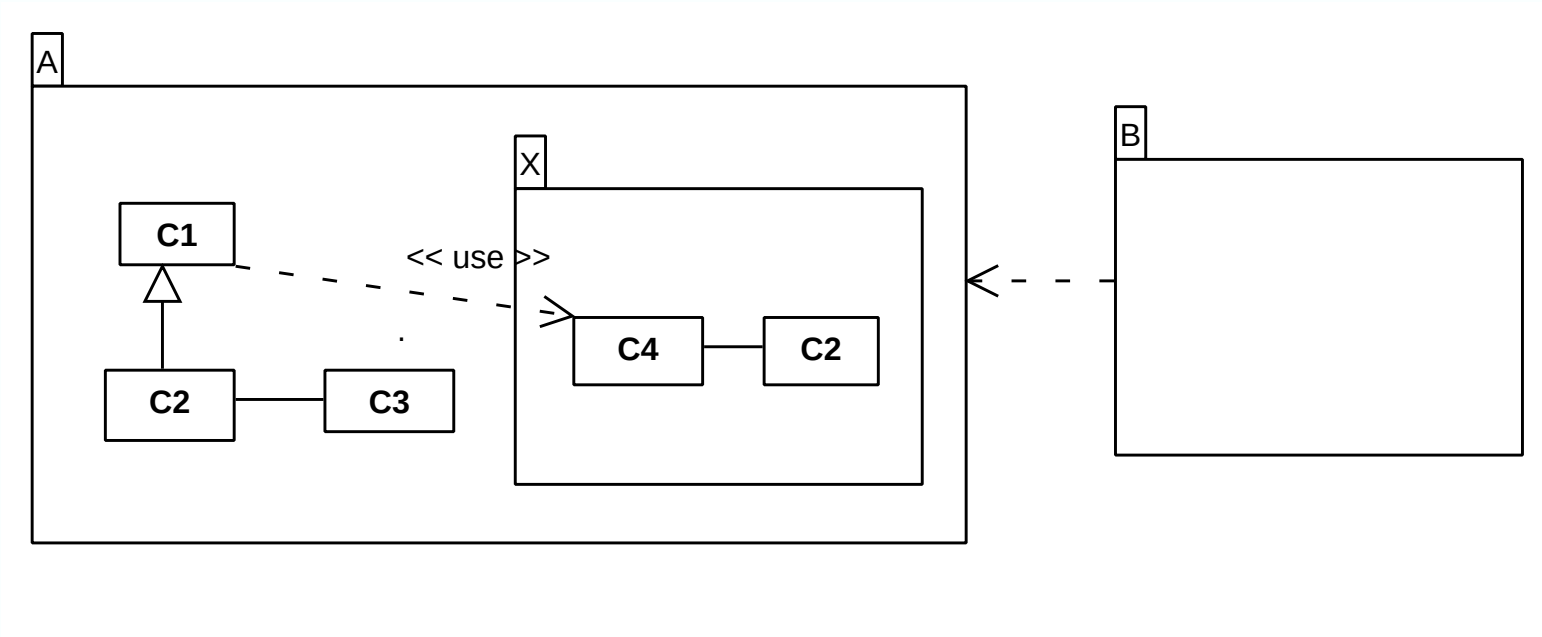
- Exemplo : Package A.X

```
//arquivo A/X/c4.java
   package A.X;
   public class c4 {...}

//arquivo A/X/c2.java
   package A.X;
   public class c2 {...}
```

Importação de Pacotes (I)

- Uma classe pode referenciar diretamente qualquer outra classe definida dentro do seu próprio pacote.



Importação de Pacotes (II)

- A classe C2 do pacote A pode instanciar um objeto do tipo C3 assim:
  `C3 atr = new C3 ( );`
- O pacote B pode criar um objeto do tipo C2 pelo menos de três formas alternativas.

1^a Alternativa

```
A.X.C2 atr = new A.X.C2( );
```

- Desvantagens:
  1. O programa fica "poluído" com a manipulação de nomes longos com várias referências para classes do mesmo pacote.
     Ex : `A.X.C2` e `A.X.C4`
  2. Intensifica-se o acoplamento entre as classes com várias referências repetidas à localização da definição de um mesmo tipo.

2^a Alternativa

- - Declara-se o nome completo do tipo através do comando import.

  ```
  import A.X.C2;
  C2 atr = new C2 ( );
  ```

- Dentro do pacote B pode-se usar o tipo C2 do pacote A.X
- Note que esta forma só pode ser usada quando o nome do tipo que estamos importando não coincide com outro nome de pacote corrente ou de outro comando import da mesma definição.
- Essa solução é a mais recomendada pois elimina as desvantagens citadas na 1a.Alternativa.

3^a Alternativa

- Usar o comando *import* para declarar um pacote do qual iremos importar um ou mais tipos:

```
import A.X.*; // todas as classes do pacote A.X
são importadas
C2 atr = new C2( );
```

- Desvantagens:
  1. Oculta-se a informação de que a classe C2 pertence ao pacote A.X (ela pode pertencer ao pacote B!).
  2. Aumenta-se o acoplamento entre os pacotes, já que a definição faz com que um pacote dependa de todo o conteúdo de um pacote importado.

Importação de Pacotes (III)

- A sintaxe `import A.*` não se aplica a subpacotes de A (no caso, o pacote X).
- Se você importar o pacote A, você deve se referir à classe C4 através do seu nome completo A.X.C4 (ou usar o comando `import A.X.*`).
- Se a classe C1 do pacote A precisa criar um objeto do tipo C4 do pacote X então ela deve usar o nome completo da classe:
  A.X.C4 atr = new A.X.C4( );

Pacotes Anônimos

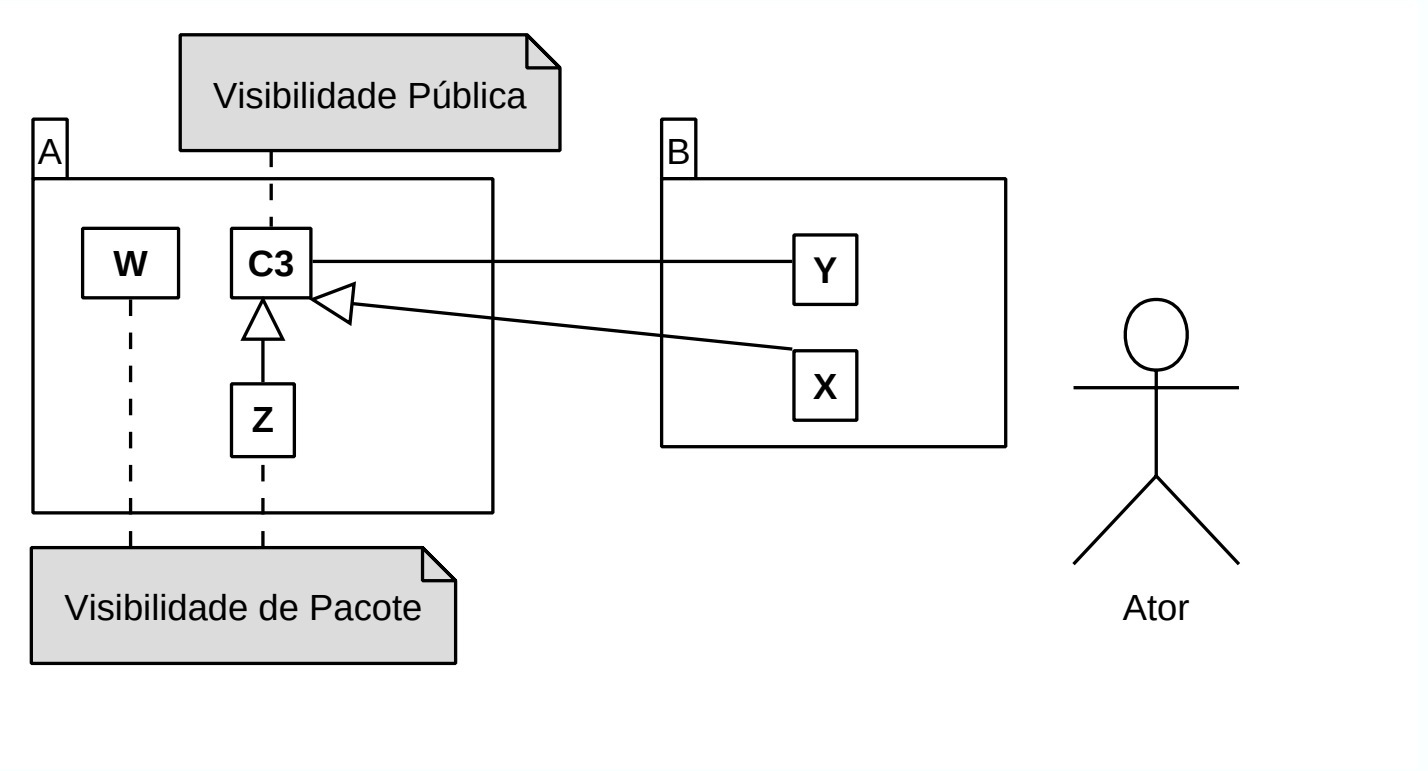
- Uma definição de classe que não contenha um comando **package** é considerada parte de um pacote anônimo.
- Um pacote anônimo contém apenas os tipos definidos no diretório corrente do ambiente de execução que não usam o comando **package**.
- Uma definição de tipo de um pacote anônimo pode importar tipos de outros pacotes.
- Não se aconselha importar tipos de um pacote anônimo pois a sua interpretação depende da implementação específica de Java de cada ambiente de execução.

Visibilidade

- Dois aspectos importantes devem ser considerados:
  1. Visão externa do pacote, e
  2. Visão interna ao pacote.
- Existem quatro graus de visibilidade em Java:
  1. modificador **public**: aplicado a classes (ou interfaces), métodos e atributos.
  2. modificador **private**: aplicado apenas a métodos e atributos.
  3. modificador **protected**: aplicado apenas a métodos e atributos.
  4. modificador **package**: aplicado a classes (ou interfaces), métodos e atributos. Essa visibilidade é assumida por Java quando nenhum modificador (public, private ou protected) é explicitamente empregado.

Visão Externa do Pacote (I)

- *Para que um tipo contido num pacote possa ser utilizado por classes ou interfaces de outros pacotes, sua definição deve ser feita usando o modificador* **public**.



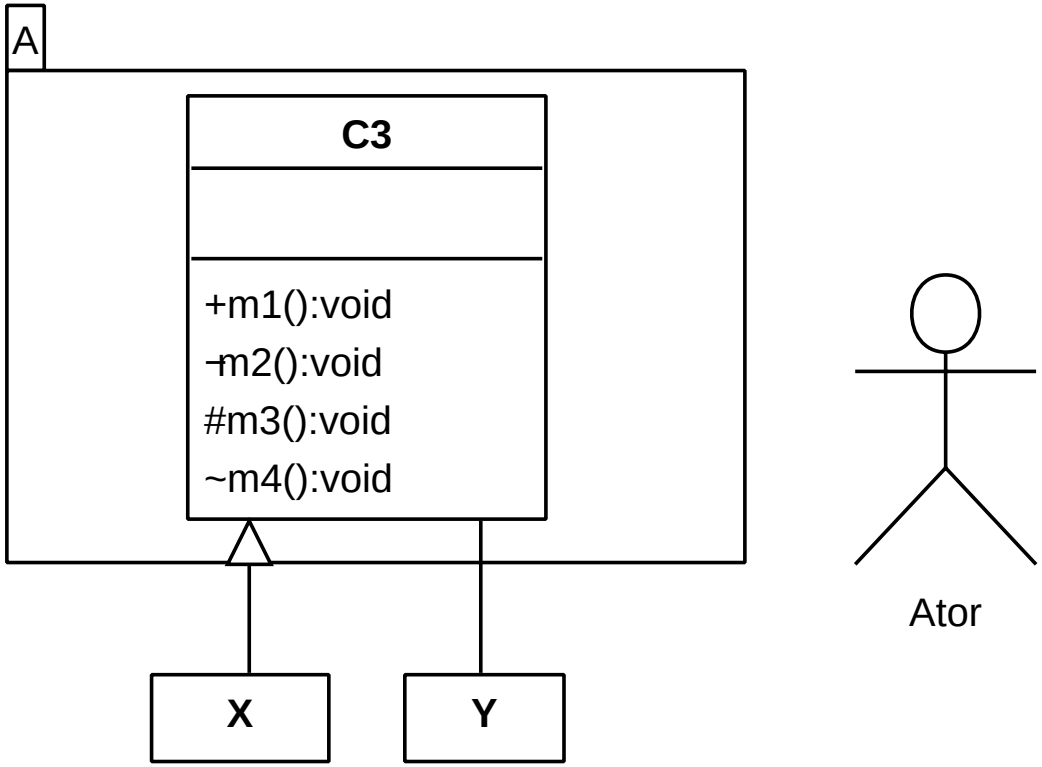
Visão Externa do Pacote (II)

- Exemplo :

```
public class C3 {...}
class W {..} // visibilidade de pacote
class Z extends C3 {...} // visibilidade
de pacote
```

- A classe C3 é visível externamente ao pacote A enquanto as classes **W** e **Z** são internas ao pacote A e, portanto, não são visíveis fora dele.

Visão Externa do Pacote (III)



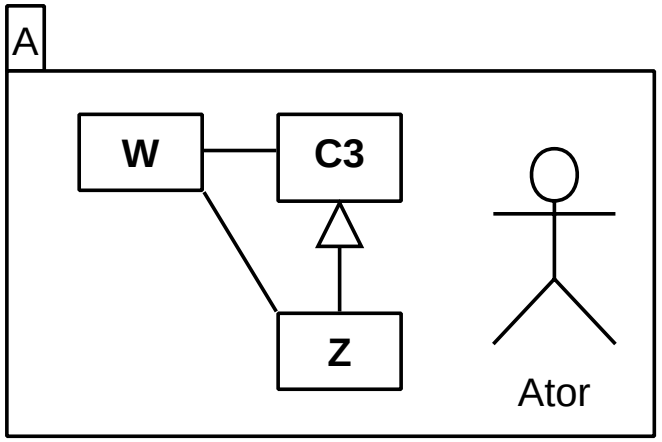
Visão Externa do Pacote (IV)

- Para clientes externos ao pacote A, e.g. classes $\mathbf{X}$ e $\mathbf{Y}$, a interface pública de $\mathbf{C3}$ é composta pelo método $\mathbf{m1}(\ )$ que tem visibilidade pública.
- Classe $\mathbf{X}$ é um cliente por herança (ou seja, $\mathbf{X}$ é filha de $\mathbf{C3}$), e, portanto, além de ter acesso ao método $\mathbf{m1}(\ )$, ela também "enxerga" o método $\mathbf{m3}(\ )$ que tem visibilidade protegida.
- Classe $\mathbf{Y}$ não é filha de $\mathbf{C3}$ e, portanto, enxerga apenas o método $\mathbf{m1}(\ )$.

Visão Externa do Pacote (V)

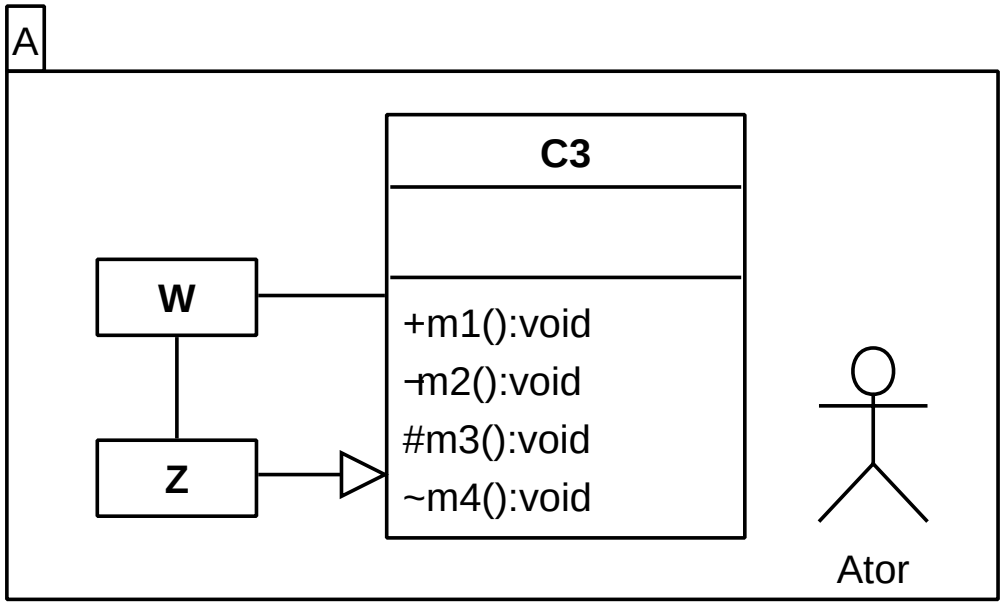
- O método $\mathbf{m2}(\ )$ é privado e, portanto, é visto apenas dentro da classe $\mathbf{C3}$ (implementação dos seus métodos).
- O método $\mathbf{m4}(\ )$ tem visibilidade de pacote e, portanto, não é visto por clientes fora do pacote (apenas por clientes dentro do pacote).

Visão Interna do Pacote (I)



- Todas as classes pertencentes a um mesmo pacote enxergam-se umas às outras.

Visão Interna do Pacote (II)



Visão Interna do Pacote (III)

- Classe **Z** (filha da classe C3) enxerga:
  - **m1( )** que é público, visível dentro e fora do pacote;
  - **m4( )** que é declarado com visibilidade de pacote e visto, portanto, por todas as classes dentro do pacote;
  - **m3( )** que é protegido;
- A visibilidade protegida não tem o efeito esperado para classes de dentro do pacote. Tanto a classe **Z** (que é filha de C3) quanto a classe **W** (que não é filha de C3) enxergam o método **m3( )** declarado como protegido!
- A visibilidade protegida de **m3( )** afeta as classes filhas de **C3** que estão fora do pacote A, e.g., a classe X.

Visão Interna do Pacote (IV)

- As semânticas das visibilidades protegidas de Java e C++ não são equivalentes.
- A Classe $\mathbf{W}$ (que não é filha de $\mathbf{C3}$) enxerga:
  - $\mathbf{m1}(\ )$ visibilidade pública;
  - $\mathbf{m4}(\ )$ visibilidade de pacote;
  - $\mathbf{m3}(\ )$ visibilidade protegida, implicando que todas as classes internas ao pacote "enxergam" $\mathbf{m3}(\ )$.
- O método $\mathbf{m2}(\ )$ é privado e, portanto, não é visto nem por $\mathbf{W}$, nem por $\mathbf{Z}$, nem por nenhuma classe fora do pacote.

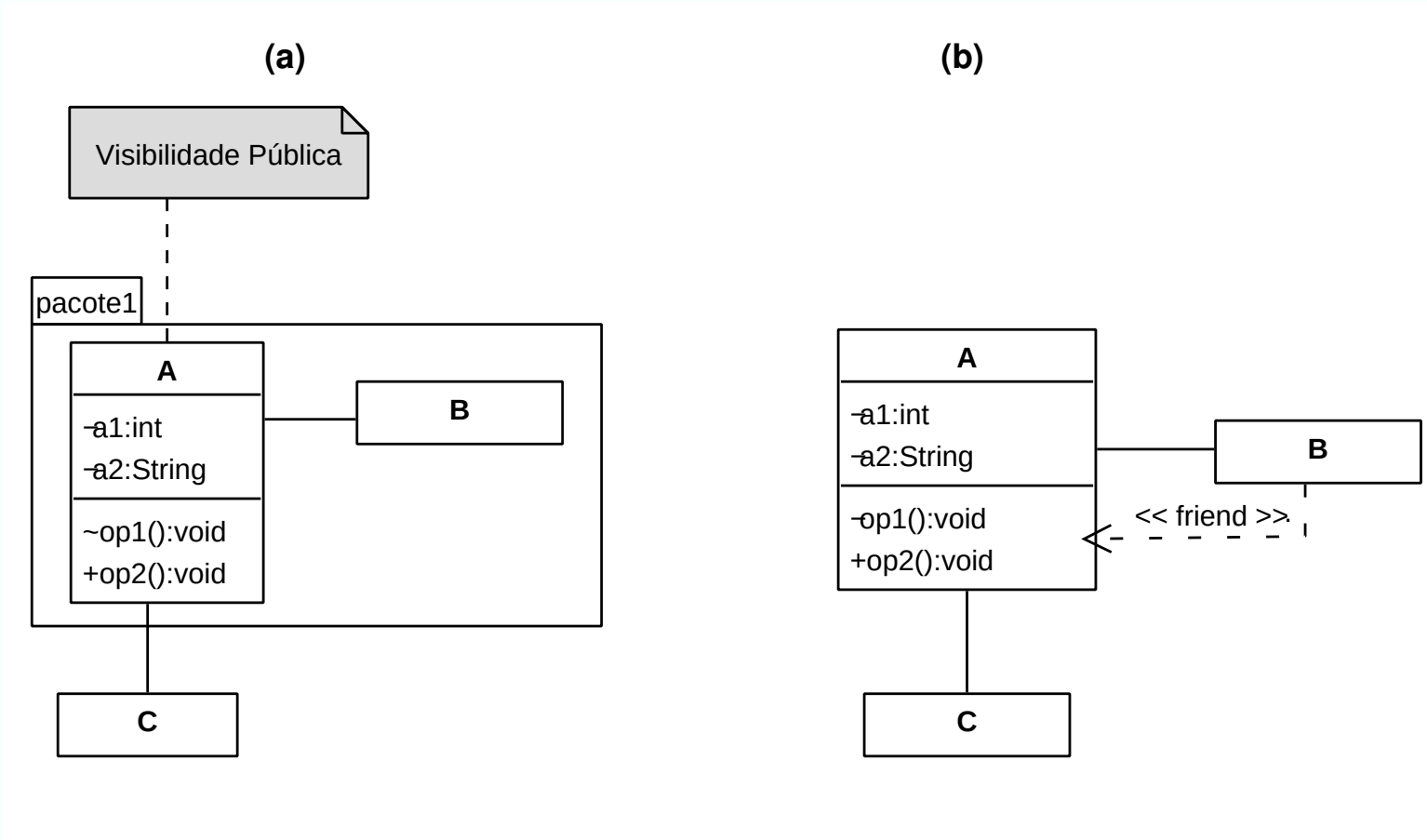
Visibilidade de Pacote (I)

- Métodos e atributos de uma classe que não são públicos e que devem ser feitos visíveis para classes derivadas **fora do pacote** são declarados como protegidos.
- A visibilidade protegida de Java oculta métodos e atributos de classe não-filhas fora do pacote.
- Todos os atributos e métodos não-privados (isto é: públicos, protegidos ou com visibilidade de pacote) de todas as classes dentro de um pacote são "amigos", isto é, todas as classes enxergam uma às outras e também aos atributos e métodos públicos, protegidos e com visibilidade de pacote.

Visibilidade de Pacote (II)

- C++ é uma linguagem OO cujo único mecanismo de encapsulamento é o conceito de classe (embora recentemente tenha sido adicionado o conceito de "name space" a ela).
- Java, além de classes, possui o mecanismo de pacotes que permite que classes inter-relacionadas enxerguem membros não-privados umas das outras.

Exemplo de Pacotes em C++ (I)



Exemplo de Pacotes em C++ (II)

- No item (a), a classe B "enxerga" todos os métodos da A, mas a classe C só "enxerga" os métodos públicos.
- Uma maneira de implementar isso em C++ é apresentado no item (b). Porém, há quebra de encapsulamento e os atributos de A também podem ser acessados por B.

Variável de Ambiente CLASSPATH (I)

- Essa variável contém a relação dos diretórios onde a máquina virtual Java deve procurar os arquivos das classes (.class) em tempo de execução.
- Após uma busca mal sucedida da classe nas bibliotecas padrões do Java, a máquina virtual busca em cada um dos diretórios especificados no CLASSPATH, na ordem definida nessa variável.
- Por exemplo, supondo que essa variável contenha o seguinte valor:

```
C:\dir1; C:\dir2; C:\dir3
```

Caso a máquina virtual não encontre a classe "`A.C`" nas suas bibliotecas, ela tentará encontrar em `dir1`, caso não encontre, continuará a busca em `dir2`. Caso contrário, a busca é encerrada e a classe é instanciada a partir de `dir1`.

Variável de Ambiente CLASSPATH (II)

- Suponha que os diretórios **A** e **B** correspondam aos pacotes de nível mais alto de sua aplicação, e que eles pertençam à estrutura de diretórios do disco C:

```
C:\
    CursoJava
    |          Exemplos
    |          |        A
    |          |        |   X
    |          |        |   | C4
    |          |        |   | C2
    |          |        |   C1
    |          |        |   C2
    |          |        |   C3
    |          |        B
```

Variável de Ambiente CLASSPATH (III)

```
C:\> set  CLASSPATH = %CLASSPATH%; C:\CursoJava\Exemplos
```

- Devemos acrescentar esse novo caminho à variável de ambiente CLASSPATH.
- O %CLASSPATH% significa que o conteúdo anterior da variável de ambiente será mantido.
- Os diferentes caminhos são separados por “;” .

O Utilitário JAR (I)

- Para facilitar o transporte de uma aplicação entre diferentes computadores temos o comando jar:

```
C:\jar -cf A.jar C:\CursoJava\Exemplos\A\*.*
C:\jar -cf B.jar C:\CursoJava\Exemplos\B\*.*
```

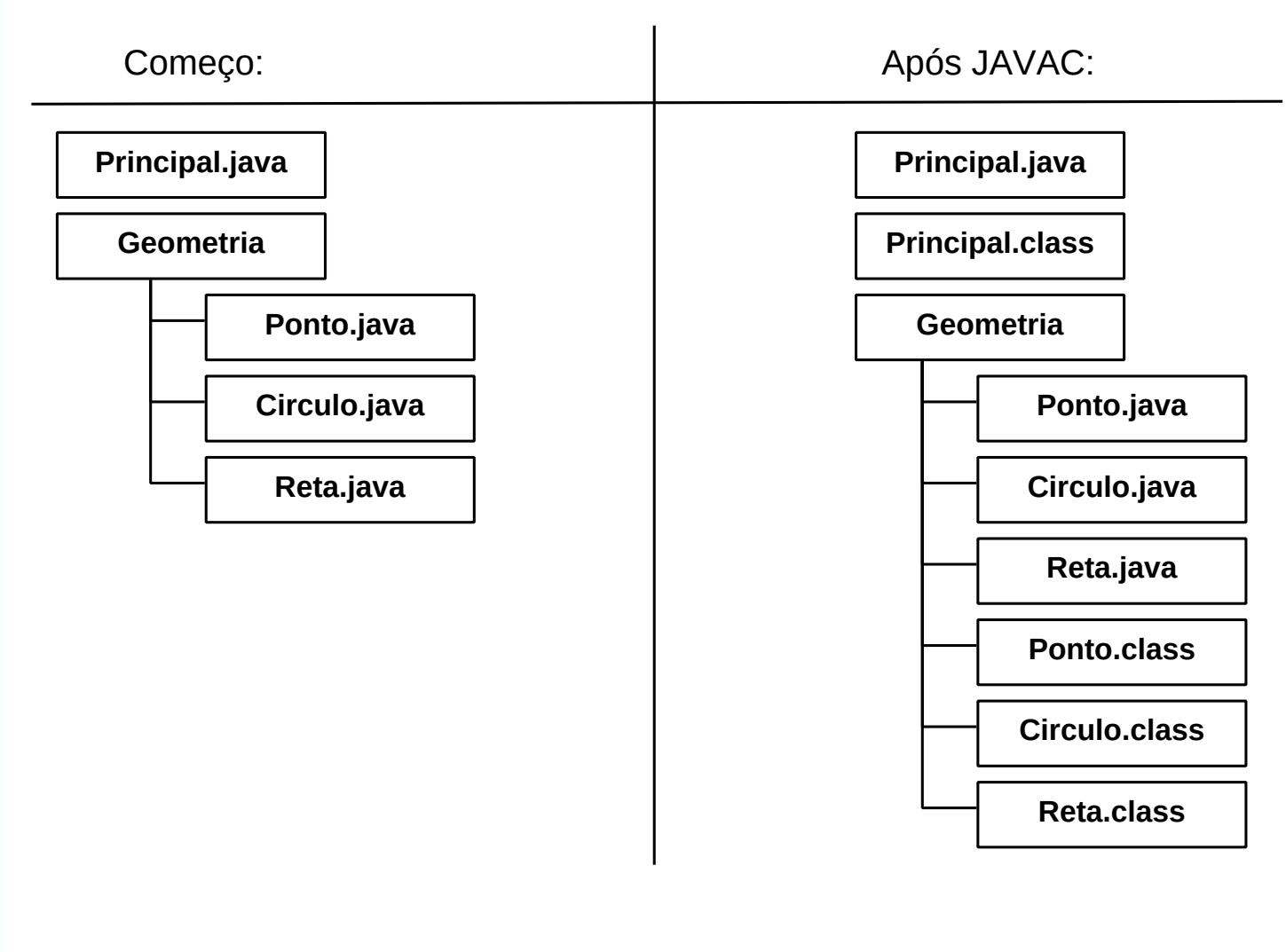
- Dois arquivos(`A.jar` e `B.jar`) são criados contendo, respectivamente, todos os pacotes e definições das subárvores das aplicações A e B.

O Utilitário JAR (II)

- Para usar essas aplicações você deve:
  1. Garantir que exista uma máquina virtual Java instalada.
  2. Incluir na variável `CLASSPATH` o nome completo dos arquivos transferidos.
- Supondo que os arquivos sejam mantidos no diretório raiz do drive C da nova máquina:

```
C:\> set  CLASSPATH = %CLASSPATH%; C:\A.jar; C:\B.jar
```

Visibilidade de Pacote - Exemplo de Pacote



Exemplo - Arquivo Ponto.java

```
//Ponto.java
package Geometria;
public class Ponto {
  float x,y; // visibilidade de pacote
  public Ponto(float ax, float ay){
     this.x=ax; this.y =ay;
  }
  public void move(float dx, float dy){
     this.x+=dx; this.y+=dy;
  }
  public void mostra( ){
     System.out.println(``(''+this.x+``,''+this.y+``)'');
  }
}// fim da classe Ponto
```

Exemplo - Arquivo Circulo.java (I)

```
//Circulo.java
package Geometria;
public class Circulo {
    float raio,x,y; // visibilidade de pacote
    public Circulo(float ax, float ay, float ar){
        this.x=ax; this.y =ay;this.raio=ar;
    }
    public void altera_raio(float a){
        this.raio=a;
    }
    public void move(float dx, float dy ){
        this.x+=dx; this.y+=dy;
    }
...
```

Exemplo - Arquivo Circulo.java (II)

```
public float distancia(Ponto ap) {
  float dcp; // distância do centro do círculo ao ponto
  dcp =(float)Math.sqrt(
      (double) ((x-ap.x)*(x-ap.x)+(y-ap.y)*(y-ap.y)) );
  // acesso direto aos atributos do objeto pois as classes
  // pertencem ao mesmo pacote
  if (dcp<raio) {return raio -dcp;}
  else{return dcp-raio;}
}
public void mostra ( ) {
  System.out.println(``(''+this.x+``,''+this.y+ ``,''+
                     this.raio+``)''); }
} // fim da classe Circulo
```

Exemplo - Arquivo Reta.java (I)

```
//Reta.java
package Geometria;
public class Reta {
    Ponto a, b; // visibilidade de pacote
    public Reta(float ax, float ay, float bx, float by){
        a = new Ponto(ax,ay);
        b=new Ponto(bx,by);
    }
    public float distancia (Ponto ap){
        // método não implementado, acesso livre
        // aos atributos do objetoap  para
        // calcular a distância de ponto a reta.
        return 0.0f;
    }
...
```

Exemplo - Arquivo Reta.java (II)

```
...
    public void mostra( ) {
        a.mostra( );
        b.mostra( );
    }
} // fim da classe Reta
```

Exemplo - Arquivo Principal.java

```
//Principal.java
import Geometria.Circulo;
import Geometria.Ponto;
public class Principal {
  public static void main(String args[ ]){
    Circulo acirc;
    // acirc.x=(float)10.0; ERRO! visibilidade de pacote
    Ponto apto;
    acirc = new Circulo((float)0.0,(float)0.0,(float)1.0);
    acirc.mostra( );
    apto=new Ponto((float)4.0, (float)3.0);
    apto.mostra( );
    System.out.println(``Dist:'' + acirc.distancia(apto));
  }
} // fim da classe Principal
```