Introdução a Orientação a Objetos em Java (II)

Grupo de Linguagens de Programação

Departamento de Informática
PUC-Rio

Sobrecarga

- Um recurso usual em programação OO é o uso de *sobrecarga* de métodos.
- Sobrecarregar um método significa prover mais de uma versão de um mesmo método.
- As versões devem, necessariamente, possuir listas de parâmetros diferentes, seja no tipo ou no número desses parâmetros (o tipo do valor de retorno pode ser igual).

Sobrecarga de Construtores

- Como dito anteriormente, ao criarmos o construtor da classe **Point** para inicializar o ponto em uma dada posição, perdemos o construtor padrão que, não fazendo nada, deixava o ponto na posição (0,0).
- Nós podemos voltar a ter esse construtor usando sobrecarga.

Sobrecarga de Construtores: Exemplo de Declaração

```
class Point {
  int x = 0;
  int y = 0;
  Point() {
  }
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  ...
}
```

Sobrecarga de Construtores: Exemplo de Uso

- Agora temos dois construtores e podemos escolher qual usar no momento da criação do objeto.

```
Point p1 = new Point(); // p1 está em (0,0)
Point p2 = new Point(1,2); // p2 está em (1,2)
```

Encadeamento de Construtores

- Uma solução melhor para o exemplo dos dois construtores seria o construtor vazio chamar o construtor que espera suas coordenadas, passando zero para ambas.
- Isso é um *encadeamento* de construtores.
- Java suporta isso através da construção **this (...)**. A única limitação é que essa chamada seja a primeira linha do construtor.

Exemplo revisitado

```
class Point {
  int x, y;
  Point() {
    this(0,0);
  }
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  ...
}
```

Sobrecarga de Métodos

- Pode ser feita da mesma maneira que fizemos com os construtores.
- Quando sobrecarregamos um método, devemos manter a semântica: não é um bom projeto termos um método sobrecarregado cujas versões fazem coisas completamente diferentes.

Sobrecarga de Métodos: Exemplo de Uso

- A classe **Math** possui vários métodos sobrecarregados. Note que a semântica das várias versões são compatíveis.

```
int a = Math.abs(-10);     // a = 10;
double b = Math.abs(-2.3); // b = 2.3;
```

Herança & Polimorfismo

Herança

- Como vimos anteriormente, classes podem ser compostas em hierarquias, através do uso de *herança*.
- Quando uma classe herda de outra, diz-se que ela a *estende* ou a *especializa*, ou os dois.
- Herança implica tanto herança de interface quanto herança de código.

Interface & Código

- Herança de interface significa que a classe que herda recebe todos os métodos declarados pela superclasse que não sejam *privados*.
- Herança de código significa que as *implementações* desses métodos também são herdadas. Além disso, os campos que não sejam privados também são herdados.

Herança em Java

- Quando uma classe *B* herda de *A*, diz-se que *B* é a *sub-classe* e *estende A*, a *superclasse*.
- Uma classe Java estende apenas uma outra classe—a essa restrição damos o nome de *herança simples*.
- Para criar uma sub-classe, usamos a palavra reservada **extends**.

Exemplo de Herança

- Podemos criar uma classe que represente um pixel a partir da classe **Point**. Afinal, um pixel é um ponto colorido.

```
public class Pixel extends Point {
  int color;
  public Pixel(int x, int y, int c) {
    super(x, y);
    color = c;
  }
}
```

Herança de Código

- A classe **Pixel** herda a interface e o código da classe **Point**. Ou seja, **Pixel** passa a ter tanto os campos quanto os métodos (com suas implementações) de **Point**.

```
Pixel px = new Pixel(1,2,0); // Pixel de cor 0
px.move(1,0); // Agora px está em (2,2)
```

super

- Note que a primeira coisa que o construtor de **Pixel** faz é chamar o construtor de **Point**, usando, para isso, a palavra reservada **super**.
- Isso é necessário pois **Pixel** é uma extensão de **Point**, ou seja, ela deve inicializar sua parte **Point** antes de inicializar sua parte estendida.
- Se nós não chamássemos o construtor da superclasse explicitamente, a linguagem Java faria uma chamada ao construtor padrão da superclasse automaticamente.

Árvore × Floresta

- As linguagens OO podem adotar um modelo de hierarquia em *árvore* ou em *floresta*.
- Árvore significa que uma única hierarquia compreende todas as classes existentes, isto é, existe uma superclasse comum a todas as classes.
- Floresta significa que pode haver diversas árvores de hierarquia que não se relacionam, isto é, não existe uma superclasse comum a todas as classes.

Modelo de Java

- Java adota o modelo de árvore.
- A classe **Object** é a raiz da hierarquia de classes à qual todas as classes existentes pertencem.
- Quando não declaramos que uma classe estende outra, ela, implicitamente, estende **Object**.

Superclasse Comum

- Uma das vantagens de termos uma superclasse comum é termos uma funcionalidade comum a todos os objetos.
- Por exemplo, a classe **Object** define um método chamado **toString** que retorna um texto descritivo do objeto.
- Um outro exemplo é o método **finalize** usado na destruição de um objeto, como já dito.

Especialização × Extensão

- Uma classe pode herdar de outra para *especializá-la* redefinindo métodos, sem ampliar sua interface.
- Uma classe pode herdar de outra para *estendê-la* declarando novos métodos e, dessa forma, ampliando sua interface.
- Ou as duas coisas podem acontecer simultaneamente...

Polimorfismo

- Polimorfismo é a capacidade de um objeto tomar diversas formas.
- O capacidade polimórfica decorre diretamente do mecanismo de herança.
- Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.

Polimorfismo de **Pixel**

- A sub-classe de **Point**, **Pixel**, é compatível com ela, ou seja, um pixel, além de outras coisas, *é* um ponto.
- Isso implica que, sempre que precisarmos de um ponto, podemos usar um pixel em seu lugar.

Exemplo de Polimorfismo

- Podemos querer criar um array de pontos. O array de pontos poderá conter pixels:

```
Point[] pontos = new Point[5]; // um array de pontos

pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0); // um pixel é um ponto
```

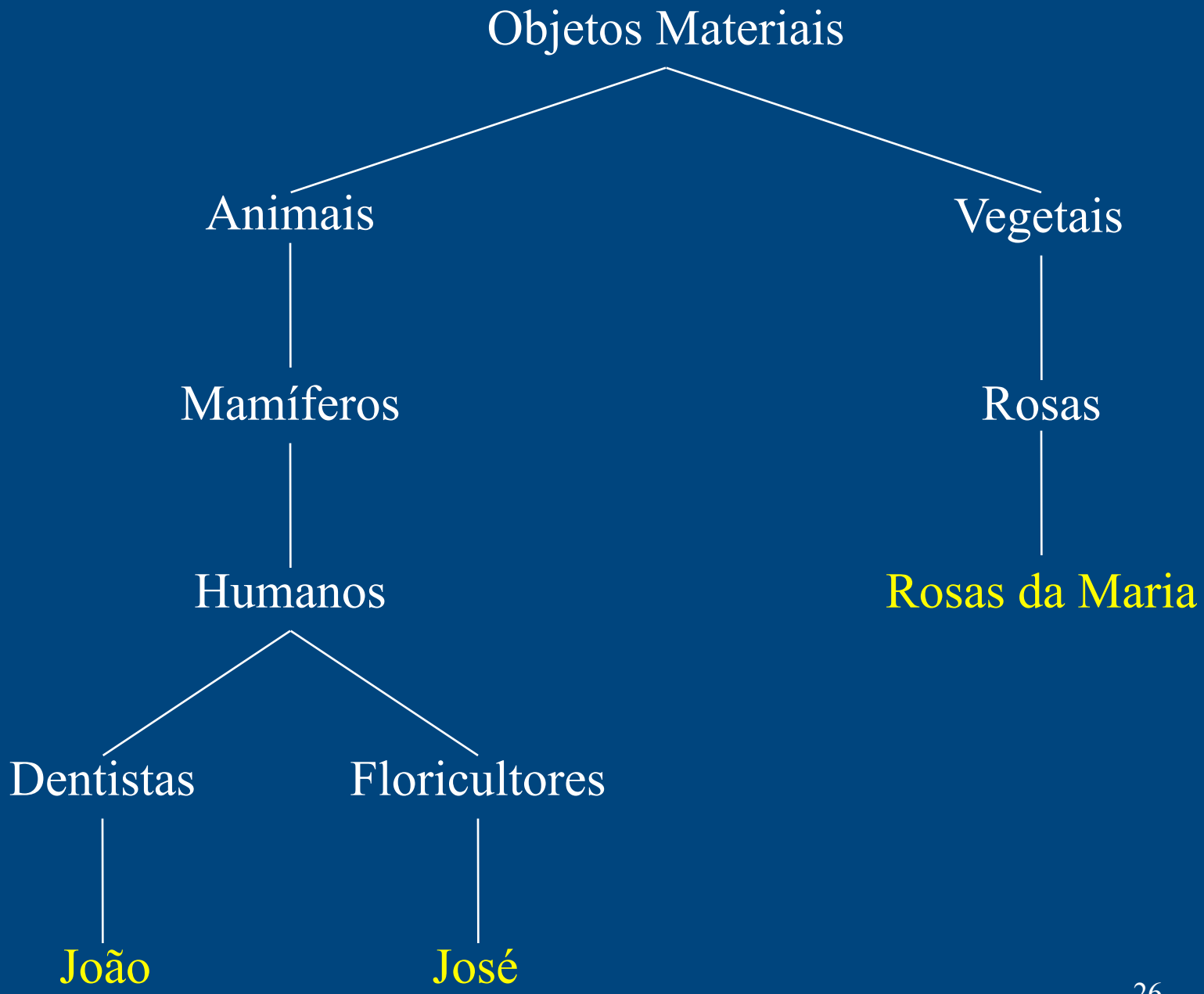
Mais sobre Polimorfismo

- Note que um pixel pode ser usado sempre que se necessita um ponto. Porém, o contrário não é verdade: não podemos usar um ponto quando precisamos de um pixel.

```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto.
Pixel px = new Point(0,0); // ERRO! ponto não é pixel.
```

Conclusão

Polimorfismo é o nome formal para o fato de que quando precisamos de um objeto de determinado tipo, podemos usar uma versão mais especializada dele. Esse fato pode ser bem entendido analisando-se a árvore de hierarquia de classes.


Objetos Materiais
Animais
Vegetais
Mamíferos
Rosas
Humanos
Rosas da Maria
Dentistas
Floricultores
João
José

Ampliando o Exemplo

- Vamos aumentar a classe **Point** para fornecer um método que imprima na tela uma representação textual do ponto.

```
public class Point {
   ...
   public void print() {
      System.out.println("Point ("+x+","+y+")");
   }
}
```

Ampliando o Exemplo (cont.)

- Com essa modificação, tanto a classe **Point** quanto a classe **Pixel** agora possuem um método que imprime o *ponto* representado.

```
Point pt = new Point();       // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)


pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Point (0,0)"
```

Ampliando o Exemplo (cont.)

- Porém, a implementação desse método não é boa para um pixel pois não imprime a cor.
- Vamos, então, *redefinir* o método em **Pixel**.

```
public class Pixel extends Point {
   ...
  public void print() {
    System.out.println("Pixel ("+x+","+y+","+color+")");
  }
}
```

Ampliando o Exemplo (cont.)

- Com essa nova modificação, a classe **Pixel** agora possui um método que imprime o *pixel* de forma correta.

```
Point pt = new Point();       // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)


pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Pixel (0,0,0)"
```

Late Binding

Voltando ao exemplo do array de pontos, agora que cada classe possui sua própria codificação para o método **print**, o ideal é que, ao correrrmos o array imprimindo os pontos, as versões corretas dos métodos fossem usadas. Isso realmente acontece, pois as linguagens OO usam um recurso chamado *late binding*.

Late Binding na prática

- Graças a esse recurso, agora temos:

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);

pontos[0].print(); // Imprime: "Point (0,0)"
pontos[1].print(); // Imprime: "Pixel (1,2,0)"
```

Definição de Late Binding

Late Binding, como o nome sugere, é a capacidade de adiar a resolução de um método até o momento no qual ele deve ser efetivamente chamado. Ou seja, a resolução do método acontecerá em tempo de execução, ao invés de em tempo de compilação. No momento da chamada, o método utilizado será o definido pela classe *real* do objeto.

Late Binding × Eficiência

O uso de late binding pode trazer perdas no desempenho dos programas visto que a cada chamada de método um processamento adicional deve ser feito. Esse fato levou várias linguagens OO a permitir a construção de métodos *constantes*, ou seja, métodos cujas implementações não podem ser redefinidas nas sub-classes.

Valores Constantes

- Java permite declarar um campo ou uma variável local que, uma vez inicializada, tenha seu valor fixo. Para isso utilizamos o modificador **final**.

```
class A {
  final int ERR_COD1 = -1;
  final int ERR_COD2 = -2;
  ...
}
```

Métodos Constantes em Java

- Para criarmos um método constante em Java devemos, também, usar o modificador **final**.

```
public class A {
  public final int f() {
    ...
  }
}
```

Classes Constantes em Java

- Uma classe inteira pode ser definida **final**. Nesse caso, em particular, a classe não pode ser estendida.

```
public final class A {
   ...
}
```

Conversão de Tipo

Como dito anteriormente, podemos usar uma versão mais especializada quando precisamos de um objeto de certo tipo mas o contrário não é verdade. Por isso, se precisarmos fazer a conversão de volta ao tipo mais especializado, teremos que fazê-lo explicitamente.

Type Casting

- A conversão explícita de um objeto de um tipo para outro é chamada *type casting*.

```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto.
Pixel px = (Pixel)pt; // OK! pt agora contém um pixel.
pt = new Point();
px = (Pixel)pt; // ERRO! pt agora contém um ponto.
pt = new Pixel(0,0,0);
px = pt; // ERRO! pt não é sempre um pixel.
```

Mais Type Casting

Note que, assim como o late binding, o type casting só pode ser resolvido em tempo de execução: só quando o programa estiver rodando é que poderemos saber o valor que uma dada variável terá e, assim, poderemos decidir se a conversão é válida ou não.

instanceof

- Permite verificar a classe real de um objeto

```
if (pt instanceof Pixel) {
   Pixel px = (Pixel)pt;
   …
}
```

Compatibilidade de Tipos

- *tipo* = interface de um objeto
  (métodos e atributos públicos)
- *Substitution Principle*
  - um tipo B é compatível com um tipo A sempre que em qualquer trecho código um objeto do tipo A pode ser substituido por um objeto do tipo B

Verificação de Compatibilidade

- Abstratamente, B *é-subtipo de* A se para cada operação e atributo público de A, existe um *compatível* em B

Compatibilidade em Java

- uma interface B é compatível com uma interface A, se B *estender* A (direta ou indiretamente)
- uma classe B é compatível com uma interface A, se B explicitamente implementar a interface A ou estender uma classe que o faça
- uma classe B é compatível com uma classe A, se B estender a classe A (direta ou indiretamente)

Redefinição de Métodos

- O que é necessário para um método de uma classe poder redefinir um método de herdado de uma super-classe ou implementar o método de uma interface?
  - deve possuir o mesmo nome
  - deve ter um tipo de valor de retorno do mesmo tipo ou de um sub-tipo do retorno do método original (regra da ***co-variância***)
  - deve ter uma lista de parâmetros com a mesma aridade, e cada parâmetro deve ser do mesmo tipo ou de um super-tipo do parâmetro do método original (regra da ***contra-variância***)
  - deve ter uma lista de exceções sinalizadas igual ao do método original, ou com menos tipos listados, ou com sub-tipos dos tipos de exceções originais, ou ambos.

Exercício

```
public class A {
   A calc (A obj, double f) throws E1 {...}
}
public class B extends A {
   Object calc (B obj, double f) throws E1 {...}
}
public class C extends A {
   C calc (A obj, double f) {...}
}
public class D extends A {
   A calc (D obj, double f) throws E1, E2 {...}
}
public class E extends A {
   A calc (A obj, double f) throws E1 {...}
}
```

Exercício

```
public class A {
   A calc (A obj, double f) throws E1 {...}
}
public class B extends A {
   Object calc (B obj, double f) throws E1 {...}
}
public class C extends A {
   C calc (A obj, double f) {...}
}
public class D extends A {
   A calc (D obj, double f) throws E1, E2 {...}
}
public class E extends A {
   A calc (A obj, double f) throws E1 {...}
}
```

Compatibilidade de Arrays

- Em Java, se B é sub-tipo de A, então B[] é sub-tipo de A[]
- Mas CUIDADO!!

```
public class Test {
   static void buggy (int i, A[] a) {
      a[i] = new A();
   }
   static public void main (String [] args) {
      A[] a = new A[5];
      buggy(0,a);
      buggy(1,a);

      B[] b = new B[10]; //B é sub-classe de A
      buggy(0,b);        //exceção!!
      buggy(1,b);        //exceção!!
   }
}
```

Herança: Simples × Múltipla

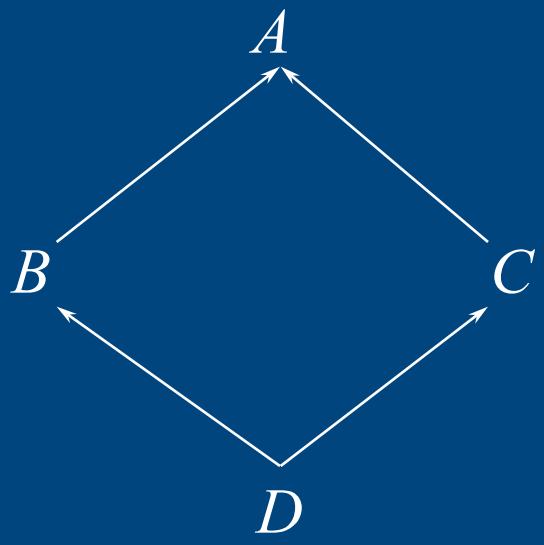
- O tipo de herança que usamos até agora é chamado de *herança simples* pois cada classe herda de apenas uma outra.
- Existe também a chamada *herança múltipla* onde uma classe pode herdar de várias classes.

Herança Múltipla

- Herança múltipla não é oferecida em todas as linguagens OO.
- Esse tipo de herança apresenta um problema quando construímos hierarquias de classes onde uma classe herda duas ou mais vezes de uma mesma super-classe. O que, na prática, torna-se um caso comum.

Problemas de Herança Múltipla

- O problema de herdar duas vezes de uma mesma classe vem do fato de existir uma herança de código.



Compatibilidade de Tipos

- Inúmeras vezes, herança múltipla é usada apenas para tornar uma classe *compatível* com as classes herdadas, sem aproveitar a herança de código.
  - para esses esses casos, Java oferece herança múltipla de interfaces
- Mas existem exceções...
  - exemplos clássicos em C++: persistência, distribuição, ...
  - para esses casos Java dá um tratamento especial

Visibilidade & Herança

Pelo que foi dito, membros públicos são herdados, enquanto membros privados não são. Às vezes precisamos algo intermediário: um membro que não seja visto fora da classe mas que possa ser herdado. As linguagens OO tipicamente dão suporte a esse tipo de acesso.

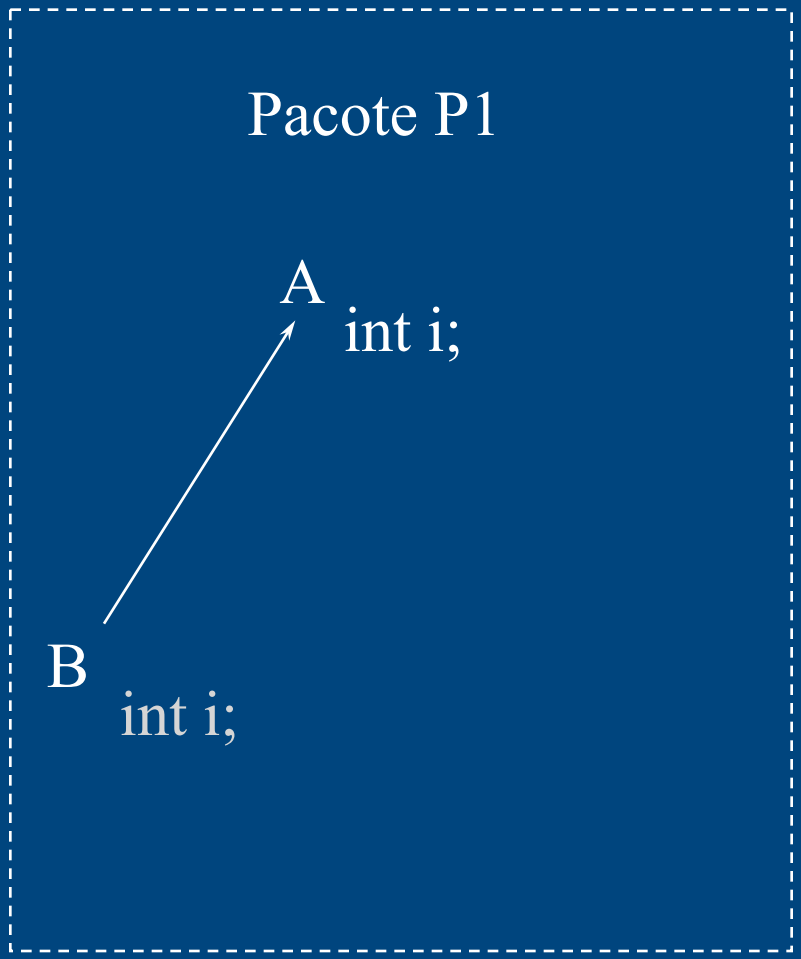
Mais Visibilidade em Java

- Java permite declararmos um membro que, embora não seja acessível por outras classes, é herdado por suas sub-classes.
- Para isso usamos o modificador de controle de acesso **protected**.

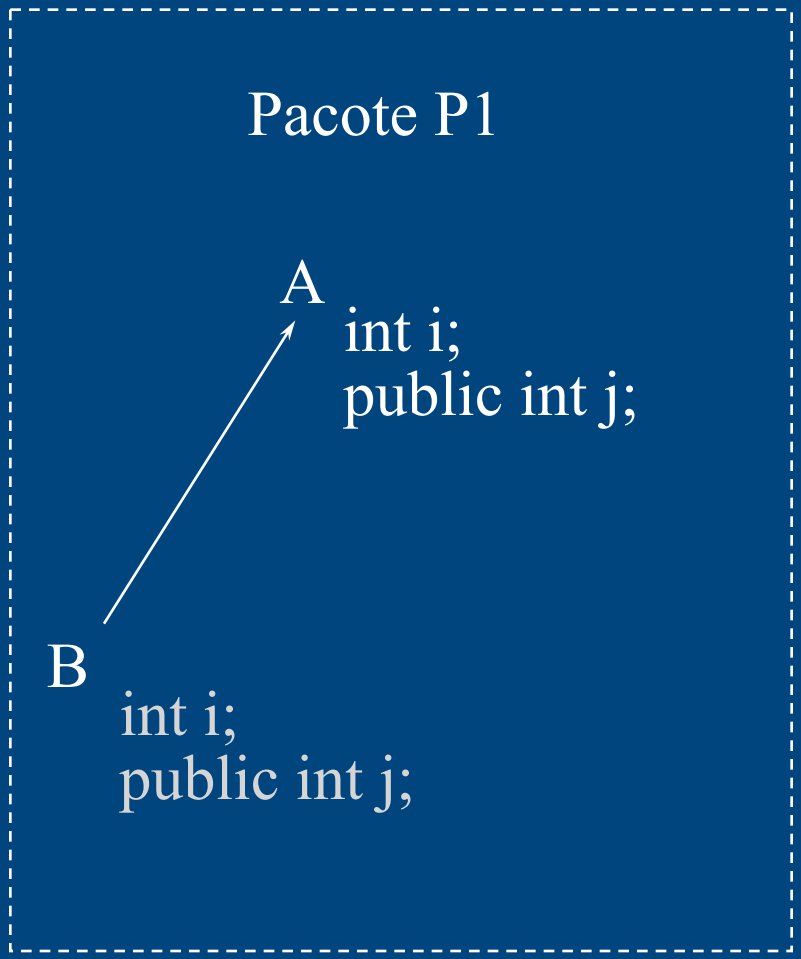
Resumo de Visibilidade em Java

- Resumindo todos os tipos de visibilidade:
  - **private**: membros que são vistos só pelo própria classe e não são herdados por nenhuma outra;
  - *package*: membros que são vistos e herdados pelas classes do pacote;
  - **protected**: membros que são vistos pelas classes do pacote e herdados por qualquer outra classe;
  - **public**: membros são vistos e herdados por qualquer classe.

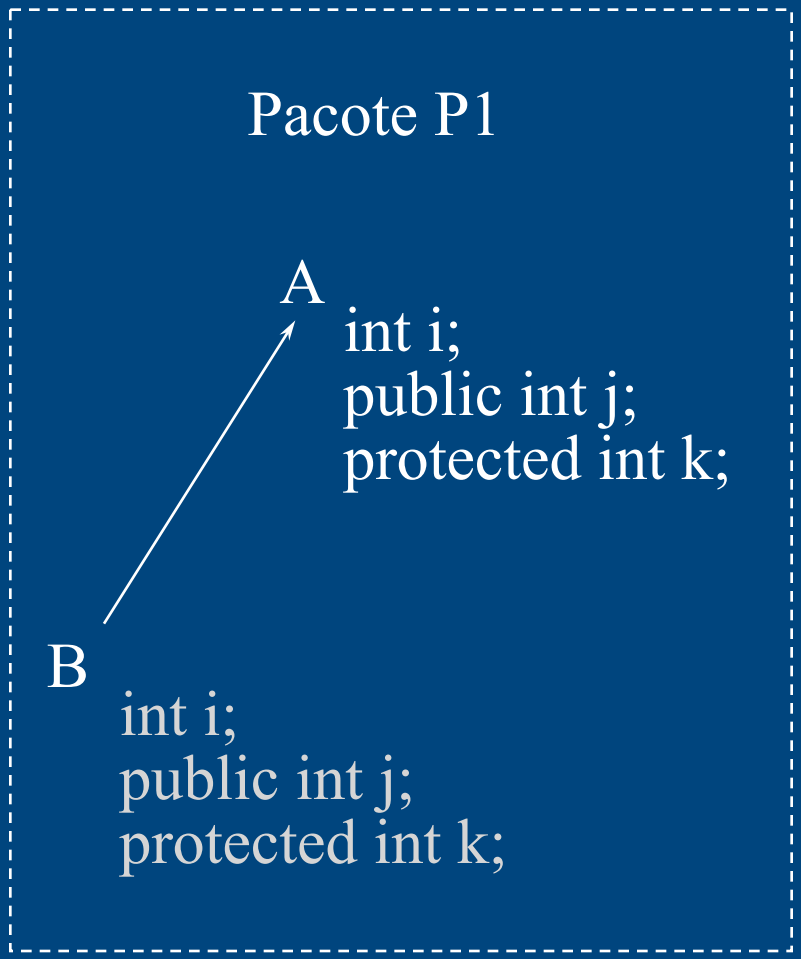
Herança de Membros



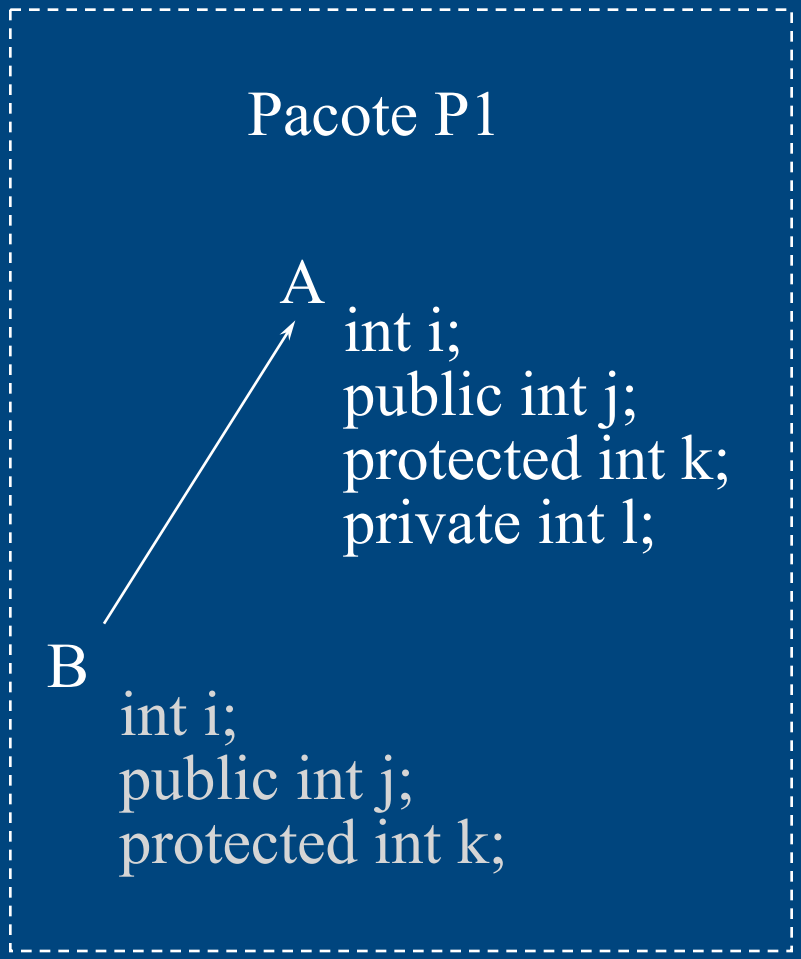
Herança de Membros



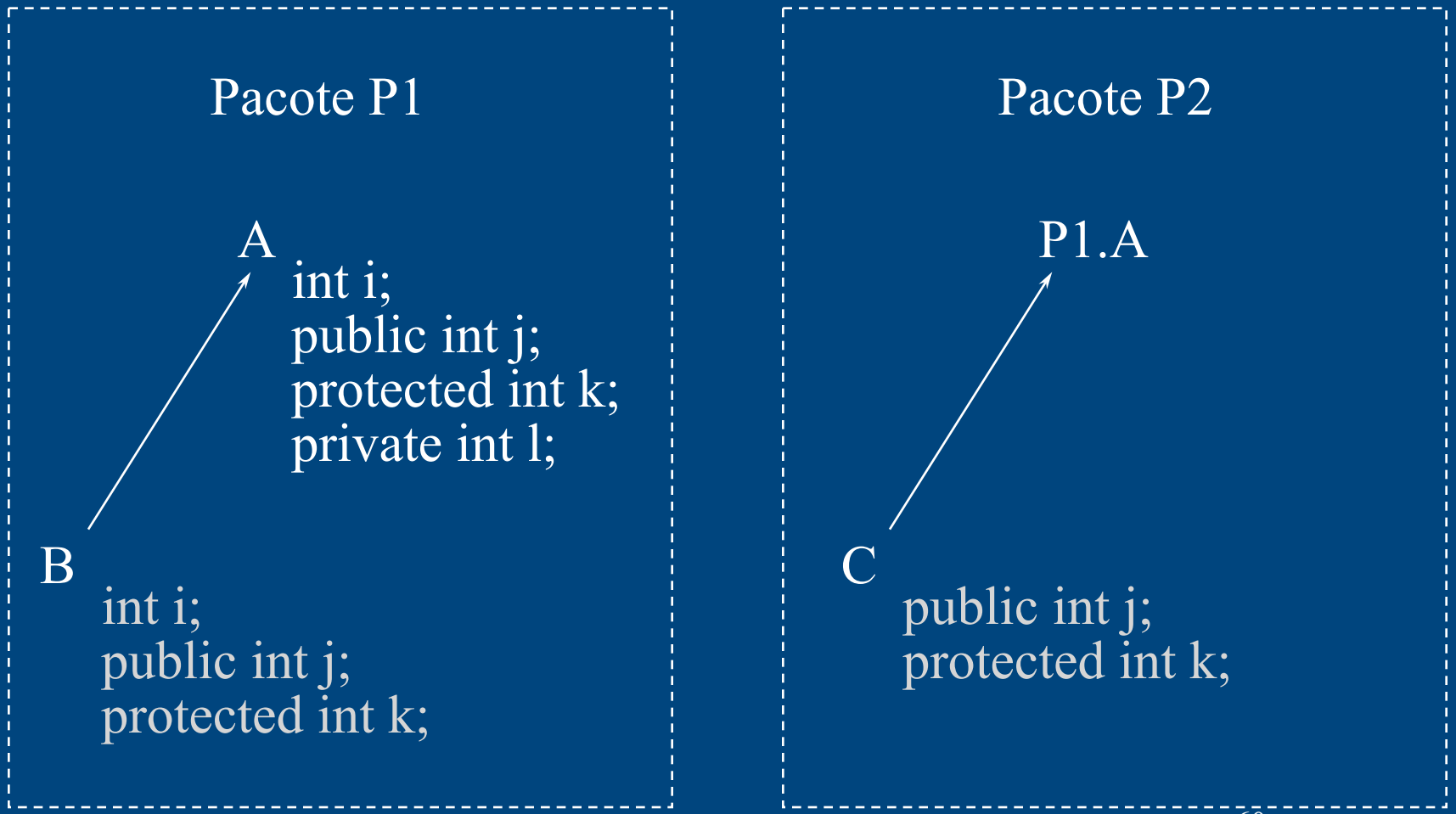
Herança de Membros



Herança de Membros



Herança de Membros



Classes Abstratas

- Ao criarmos uma classe para ser estendida, às vezes codificamos vários métodos usando um método para o qual não sabemos dar uma implementação, ou seja, um método que só sub-classes saberão implementar.
- Uma classe desse tipo não deve poder ser instanciada pois sua funcionalidade está incompleta. Tal classe é dita *abstrata*.

Classes Abstratas em Java

- Java suporta o conceito de classes abstratas: podemos declarar uma classe abstrata usando o modificador **abstract**.
- Além disso, métodos podem ser declarados abstratos para que suas implementações fiquem adiadas para as sub-classes. Para tal, usamos o mesmo modificador **abstract** e omitimos a implementação.

Exemplo de Classe Abstrata

```
public abstract class Drawing {
  public abstract void draw();
  public abstract BBox getBBox();
  public boolean contains(Point p) {
    BBox b = getBBox();
    return (p.x>=b.x && p.x<b.x+b.width &&
            p.y>=b.y && p.y<b.y+b.height);
  }
  ...
}
```