unesp

Linguagens de Programação

Aula 12

Celso Olivete Júnior

olivete@fct.unesp.br

Na aula passada

- **Implementando subprogramas**

unesp

Na aula de hoje

- Suporte para a programação orientada a objetos

unesp

Roteiro

- Introdução
- Programação orientada a objetos
- Questões de projeto para programação orientada a objetos
- Suporte para programação orientada a objetos em C++
- Suporte para programação orientada a objetos em Java
- Implementação de construções orientadas a objetos

Introdução

- Muitas linguagens de programação orientadas a objeto
  - Algumas suportam programação procedural e orientada a objetos (por exemplo, Ada 95 e C++)
  - Linguagens mais novas não suportam outros paradigmas, mas usam suas estruturas imperativas (por exemplo, Java e C#)
  - Algumas são linguagens de programação orientadas a objetos puras (por exemplo, Smalltalk e Ruby)

Programação orientada a objeto

- Uma linguagem orientada a objetos deve fornecer suporte aos três recursos chave de linguagem:

1. Tipos de dados abstratos
   - Inclui apenas a representação de dados de um tipo de dados específico e os subprogramas que fornecem as operações para esse tipo. Exemplo: **classe**
2. Herança
   - Herança é o tema central da programação orientada a objetos e das linguagens que a suportam
3. Vinculação dinâmica de chamadas a métodos
   - Polimorfismo

Evolução do paradigma

- Procedural – 1950 a 1970:
  - Abstração procedural.
- Orientada a dados – início dos anos 80:
  - Orientada a dados.
- POO – final dos anos 80:
  - Herança e *binding* dinâmico.

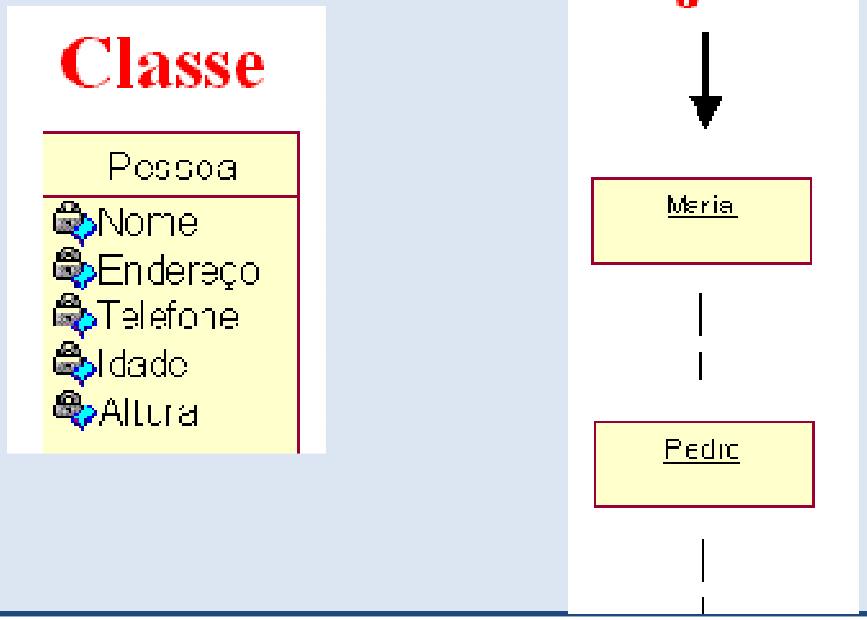
Origem da herança – anos 80

- Aumento de produtividade pode ocorrer com o reuso
  - Tipos de dados abstratos são difíceis de reusar –sempre precisam de mudanças
  - Definições de tipos de dados abstratos são todas independentes e no mesmo nível

- Herança permite novas classes definidas nos termos das já existentes

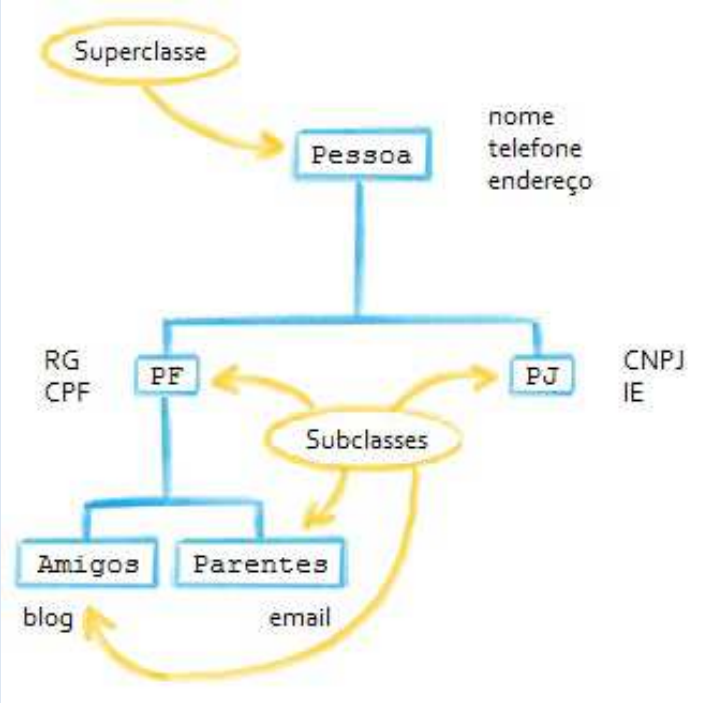
Conceitos de orientação a objetos

- Tipos de dados abstratos são geralmente chamados de *classes*
- Instâncias de classes são *objetos*



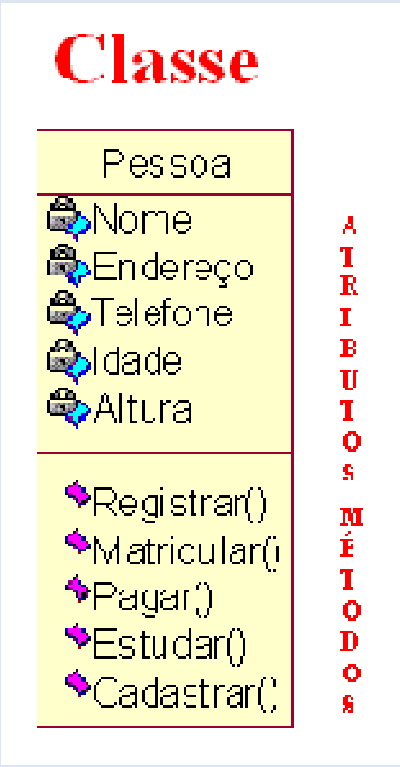
Conceitos de orientação a objetos

- Uma classe derivada por meio de herança de outra classe é uma *classe derivada* ou uma *subclasse*
- A classe da qual a nova classe é derivada é sua *classe pai* ou *superclasse*



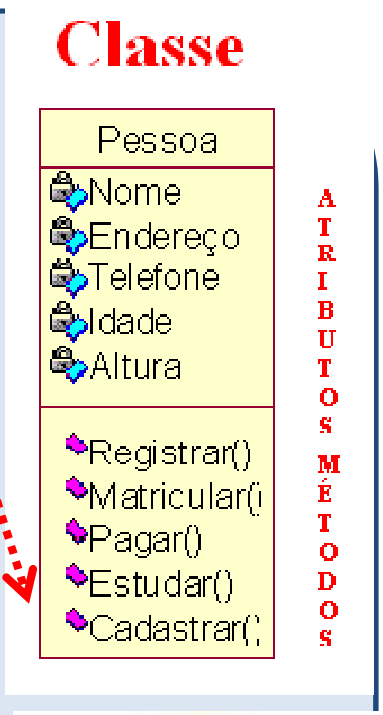
Conceitos de orientação a objetos

- Subprogramas que definem as operações em objetos de uma classe são chamados de *métodos*



Conceitos de orientação a objetos (cont...)

- Chamadas aos métodos são *mensagens*
  - Ex: objPessoa.Registrar();
- A coleção completa de métodos de um objeto é chamada de *protocolo de mensagens* ou *interface de mensagens*
- Mensagens têm duas partes: objeto e um nome de método
- Se uma nova classe é uma subclasse de uma única classe pai, então o processo de derivação é chamado de herança simples



Conceitos de orientação a objetos (cont...)

- A herança pode ser complicada por controles de acesso às entidades encapsuladas
  - Uma classe pode esconder entidades de suas subclasses
  - Uma classe pode esconder entidades de seus clientes
  - Uma classe também pode ocultar entidades para seus clientes, mas permitir às suas subclasses vê-los

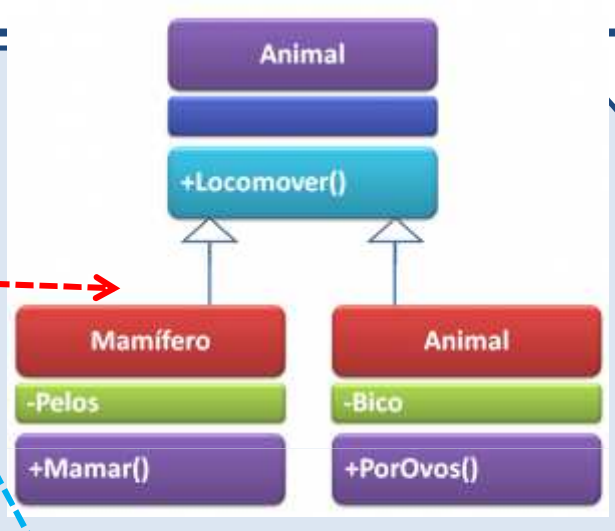
- Uma classe pode modificar um método herdado
  - O novo método sobrescreve o método herdado
  - Este, então, é chamado de método sobrescrito

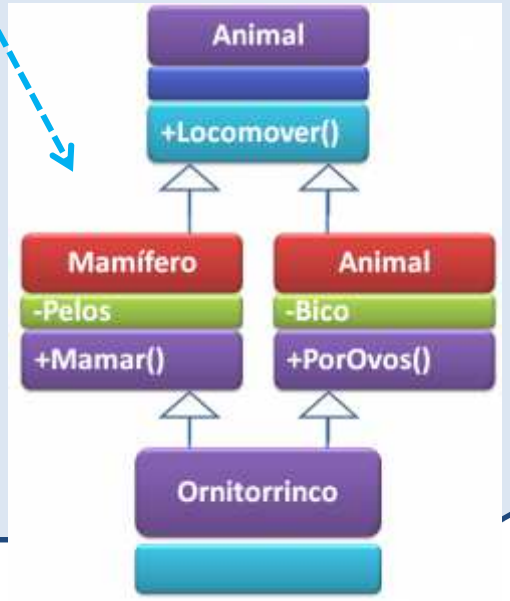
Conceitos de orientação a objetos (cont...)

- Há dois tipos de variáveis:
  - *Variáveis de classe*
  - *Variáveis de instância*
- Há dois tipos de métodos:
  - *Métodos de classe*
  - *Métodos de instância*

Conceitos de orientação a objetos (cont...)

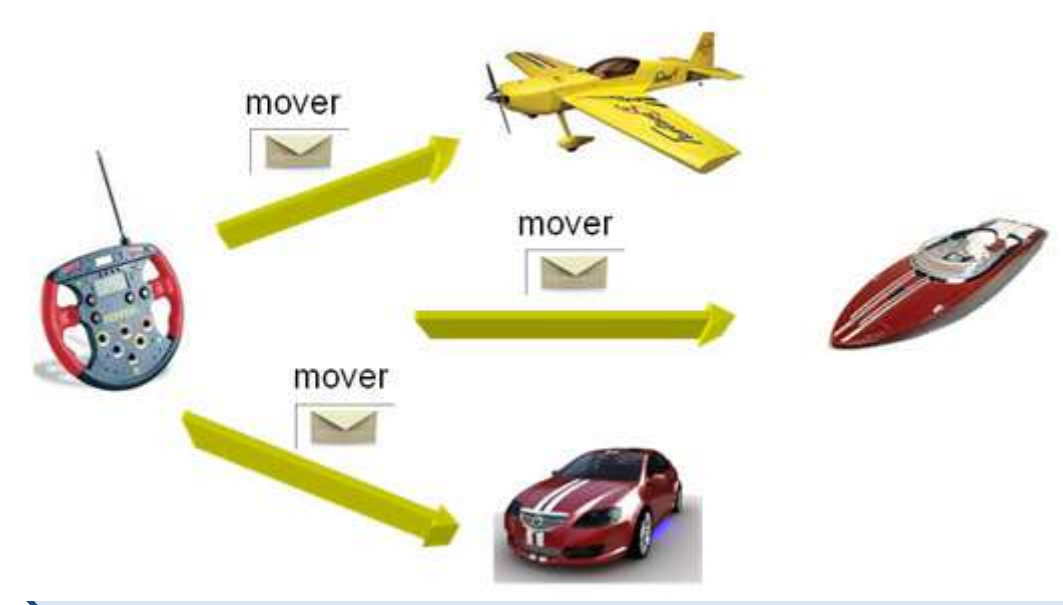
- Herança simples × herança múltipla
- Uma desvantagem da herança como uma forma de aumentar a possibilidade de reuso é que ela cria dependências entre classes em uma hierarquia





Vinculação dinâmica

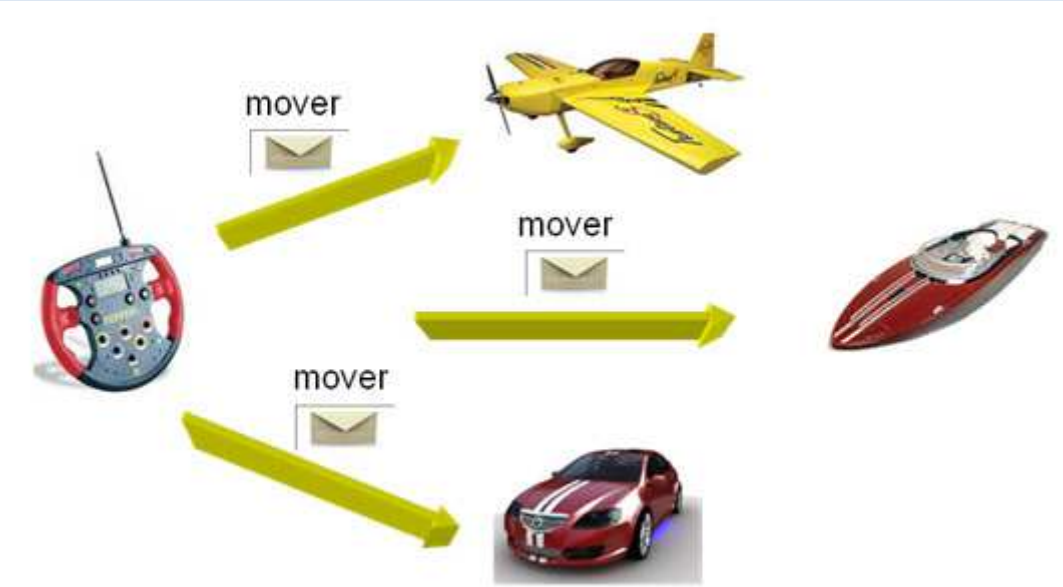
- Terceiro recurso chave de uma LP: *polimorfismo*
- Uma *variável polimórfica* pode ser definida em uma classe que é capaz de referenciar (ou apontar) os objetos da classe e objetos de qualquer dos seus descendentes



O **Polimorfismo** permite que diferentes objetos (avião, barco, automóvel) respondam uma mesma mensagem (mover) de formas diferentes (voar, navegar e correr).

Vinculação dinâmica

- Terceiro recurso chave de uma LP: *polimorfismo*
- Quando uma hierarquia de classe inclui as classes que sobrescrevem métodos e esses métodos são chamados por uma variável *polimórfica*, a ligação para o método correto será dinâmica



O **Polimorfismo** permite que diferentes objetos (avião, barco, automóvel) respondam uma mesma mensagem (mover) de formas diferentes (voar, navegar e correr).

Conceitos de vinculação dinâmica

- Um *método abstrato* é um que não inclui uma definição (apenas define um protocolo)
  - Subprograma sem o corpo
  - Subclasse que herdar esse método abstrato é a responsável por definir o corpo

- Uma *classe abstrata* inclui pelo menos um método virtual. Um classe abstrata não pode ser instanciada
  - Uma classe abstrata serve de molde para outras classes
  - Geralmente apresenta métodos incompletos, o que pode ser sobrescrito na classe filha

Exemplo de classe abstrata

- Java

```
public abstract class Funcionario
{
    public string Nome;
    public decimal Salario;
    public abstract void Reajustar();
    ...
}
```

```
public class Programador extends Funcionario
{
    public void Reajustar() //implementação o método abstrato
    {
        Salario += 1000;
    }
}
```

Objetivos da vinculação dinâmica

- Permitir que os sistemas de software sejam melhor entendidos durante o desenvolvimento e a manutenção.

Questões de projeto

1. A exclusividade dos objetos
2. As subclasses são subtipos?
3. Verificação de tipos e polimorfismo
4. Herança simples e múltipla
5. Alocação e liberação de objetos
6. Vinculação estática e dinâmica
7. Classes aninhadas
8. Inicialização de objetos

1. A exclusividade dos objetos

- Tudo é um objeto
  - Vantagem - elegância e pureza
  - Desvantagem - operações lentas para objetos simples (troca de mensagens)

2. As subclasses são subtipos?

- Um *relacionamento* "é-um(a)" se mantém entre uma classe derivada e sua classe pai?
  - Se uma classe derivada é um(a) classe pai, então os objetos da classe derivada devem expor todos os membros que são expostos por objetos da classe pai

- Uma classe derivada é um subtipo se tiver um relacionamento "é-um(a)" com sua classe pai
  - Os métodos da subclasse que sobrescrevem métodos da classe pai devem ser compatíveis em relação ao tipo com seus métodos sobrescritos correspondentes

3. Verificação de tipos e polimorfismo

- O polimorfismo pode exigir uma verificação de tipo dinâmico de parâmetros e o valor de retorno
  - Verificação de tipos dinâmica custa tempo de execução e posterga a detecção de erros de tipo

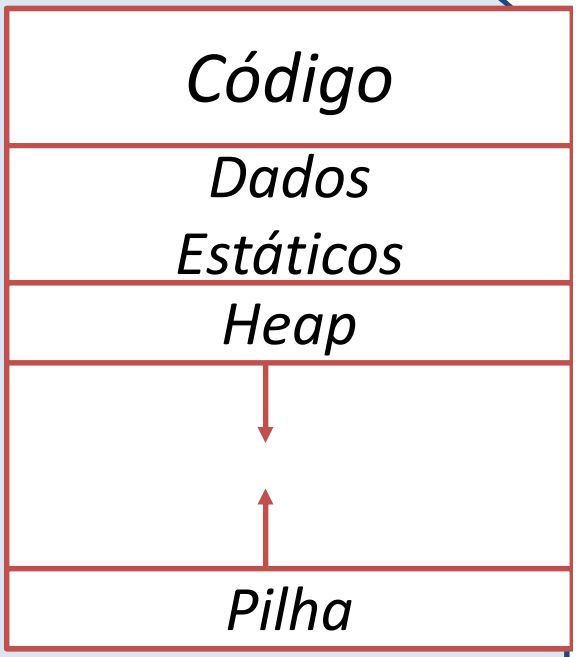
- Se o método sobrescrevedor tiver o mesmo número de tipos de parâmetros e de retorno do que o método sobrescrito, a verificação pode ser estática

4. Herança simples e múltipla

- Herança múltipla permite uma nova classe herdar de duas ou mais classes
- Desvantagens de herança múltipla:
  - Complexidade de linguagem e implementação
  - Potencial ineficiência
- Vantagem:
  - Às vezes, é bastante conveniente

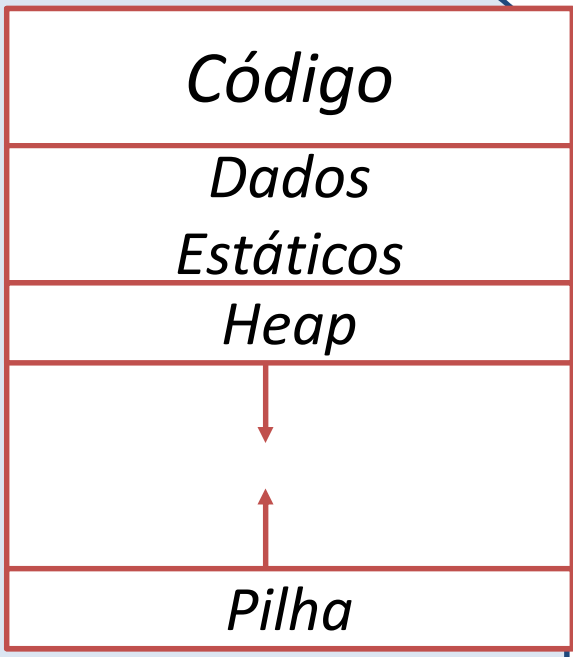
5. Alocação e liberação de objetos

- Local onde os objetos são alocados?
  - Se eles se comportam como tipos de dados abstratos, então talvez eles possam ser alocados de qualquer lugar
    - ✓ Alocados da pilha de tempo de execução
    - ✓ Explicitamente criados no monte com um operador ou função, como **new**



5. Alocação e liberação de objetos

- Local onde os objetos são alocados?
  - Se eles são todos dinâmicos do monte, existe a vantagem de ter um método de criação e acesso uniforme por meio de ponteiros ou variáveis de referência
  - Se os objetos são dinâmicos da pilha, existe um problema relacionado aos subtipos. Exemplo: se a classe B é filha da classe A e B é um subtipo de A, um objeto do tipo B pode ser atribuído a variável do tipo A.
- A liberação é explícita ou implícita?



6. Vinculação estática e dinâmica

- Todas as vinculações de mensagens a métodos são dinâmicas?
  - Todas as mensagens para métodos deveriam ser dinâmicas

7. Classes aninhadas

- Se uma nova classe é necessária em apenas uma classe, não há razão para defini-la de forma que possa ser vista por outras classes
  - Pode a nova classe ser aninhada dentro da classe que a usa?
  - Em alguns casos, a nova classe está aninhada em um subprograma, em vez de em outra classe

- Questões de projeto:
  - Quais recursos da classe aninhadora são visíveis para a classe aninhada e vice-versa?

7. Classes aninhadas
Exemplo

- Definição: classes internas (aninhadas) são classes dentro de classes.
- Vantagens:
  - agrupar classes que são usadas somente em um lugar.
  - aumenta o *encapsulamento*.
  - classes aninhadas podem tornar o código mais legível e fácil de consertar.

```
class Aninhadora{
...
      class Aninhada{
      ...
      }
}
```

7. Classes aninhadas

- Uma classe aninhada é um membro que está envolvendo uma classe e, como tal, tem acesso a outros membros da classe envolvida, mesmo que eles sejam declarados privados
  - **aumenta o *encapsulamento***

```
class Aninhadora{
...
   static class AninhadaEstatica {
     ...
   }
   class AninhadaNaoEstatica {
   ...
   }
}
```

7. Classes aninhadas

- São divididas em duas categorias: estáticas e não estáticas
  - Classes aninhadas que são declaradas *static* são simplesmente chamadas *static nested classes* (classes aninhadas estáticas).
  - Classes aninhadas não estáticas são chamadas *inner classes* (classes internas).

```
class Aninhadora{
...
   static class AninhadaEstatica {
     ...
   }
   class AninhadaNaoEstatica {
   ...
   }
}
```

7. Classes aninhadas estáticas

- Classes aninhadas estáticas são acessadas usando o nome da classe envolvida:
- Aninhadora.AninhadaEstatica

```
class Aninhadora{
...
    static class AninhadaEstatica {
      ...
    }
    class AninhadaNaoEstatica {
    ...
    }
}
```

7. Classes aninhadas estáticas

- Para criar um objeto para a classe aninhada estática

```
Aninhadora.AninhadaEstatica objAninhado
  = new Aninhadora.AninhadaEstatica();
```

```
class Aninhadora{
...
   static class AninhadaEstatica {
     ...
   }
   class AninhadaNaoEstatica {
   ...
   }
}
```

8. Inicialização de objetos

- Quando um objeto de uma subclasse é criado, a inicialização associada do membro herdado da classe pai é implícita ou o programador deve lidar explicitamente com ela?

Suporte para programação orientada a objetos em C++

- Características gerais:
  - Evolui de C e SIMULA 67
  - Está entre as linguagens de programação orientadas a objeto mais usadas
  - Sistema de tipos misto
  - Construtores e destrutores
  - Elabora os controles de acesso para entidades de classe

Suporte para programação orientada a objetos em C++

- Herança
  - Uma classe não precisa ser a subclasse de alguma classe
  - Controles de acesso aos membros podem ser
    - ✓ private (visível apenas na classe e "amigos").
    - ✓ public (visível nas subclasses e clientes).
    - ✓ protected (visível na classe a nas subclasses).

Suporte para programação orientada a objetos em C++

- Adicionalmente, o processo de subclasses pode ser declarado com controles de acesso (privado ou público), que define potenciais mudanças no acesso a subclasses
  - Derivação privada – herança pública e membros protegidos são privados nas subclasses
  - Derivação pública membros protegidos são também públicos e protegidos nas subclasses

Suporte para programação orientada a objetos em C++

Exemplo

```
class classe1 {
   private:
      int a;
      float x;
   protected:
      int b;
      float y;
   public:
      int c:
      float z;
};
```

```
class subclasse1: public classe1
{...}
b e y: protegidos
c e z: públicos
a e x: inalcançáveis
```

```
class subclasse2: private classe1
{...}
b, c, y e z: privados
a e x: inalcançáveis
```

Reexportação em C++

- Um membro que não é acessível em uma subclasse (por causa da derivação privada) pode ser declarado para ser visível usando um operador de resolução de escopo (::), por exemplo,

```
class classe1 {
   private:
     int a;
     float x;
   protected:
     int b;
     float y;
   public:
     int c:
     float z;
};
```

Instâncias de subclasse3 podem acessar **c**

```
class subclasse3: private classe1
{
    classe1 :: c; //c passa ser público
}
```

Operador de resolução de escopo → especifica a classe onde a entidade seguinte é definida

Reexportação (cont...)

- Uma motivação para usar derivação privada
  - Uma classe fornece membros que devem ser visíveis, então eles são definidos para serem membros públicos.
  - Uma classe derivada adiciona alguns novos membros, mas não quer que seus clientes vejam os membros da classe pai, mesmo que tivessem de ser públicos na definição da classe pai

Suporte para programação orientada a objetos em C++ (cont...)

- Herança múltipla é suportada
  - Se houver dois membros herdados com o mesmo nome, ambos podem ser referenciados usando o operador de resolução de escopo

Suporte para programação orientada a objetos em C++ (cont...)

- Vinculação dinâmica
  - Um método pode ser definido para ser `virtual`, o que significa que pode ser chamado por meio de variáveis polimórficas e vinculação dinâmica a mensagens
  - Um função virtual pura não tem corpo e não pode ser chamada
  - Qualquer classe que inclua uma função virtual pura é uma *classe abstrata*

Suporte para programação orientada a objetos em C++ (cont...)

- Avaliação
  - C++ fornece herança múltipla
  - Em C++, o programador deve decidir em tempo de design que métodos serão estaticamente vinculados e que deve ser dinamicamente vinculado
  - Vinculação estática é mais rápida!

Suporte para programação orientada a objetos em Java

- O projeto de classes, herança e métodos em Java é similar ao de C++

- Características gerais
  - Todos os dados são objetos, exceto valores dos tipos primitivos escalares
  - Todos os objetos em Java são dinâmicos do monte explícitos. A maioria é alocada com o operador new, mas não existe um operador de liberação explícito
  - Um método `finalize` é implicitamente chamado quando o coletor de lixo está prestes a recuperar o armazenamento ocupado pelo objeto

Suporte para programação orientada a objetos em Java (cont...)

- Herança
  - Java suporta diretamente apenas herança simples, mas um tipo de classe abstrata, chamada de `interface`, fornece suporte parcial para herança múltipla

Suporte para programação orientada a objetos em Java (cont...)

- Interface – algumas propriedades:
  a) Não pode ser instanciável (não pode criar objetos com **new**)
  b) Só pode possuir assinaturas de métodos de instância, públicos e abstratos (sem corpo).Não pode possuir métodos concretos (com corpo), nem métodos estáticos.
  c) Não pode conter variáveis de instância ou de classe (static);
  d) Pode conter declarações de constantes (com prefixo **final** e inicializadas para um valor) – nesse caso essa variável funcionará como uma constante de classe.
  e) Pode ser criada como **subinterface** de outra interface já existente, usando **extends**, como as classes.

Suporte para programação orientada a objetos em Java (cont...)

- Exemplo de interface

```
public interface InterfaceExemplo{
    public final String PALAVRA = “TESTE”;
    public void metodo1(int x);
    public String metodo2 (String s);
}
public interface InterfaceExemplo2 extends InterfaceExemplo{
   public void metodo3();
}
```

Subinterface da primeira – extensão
Herda as definições da superinterface

Suporte para programação orientada a objetos em Java (cont...)

- Dynamic Binding
  - Em Java, todas as chamadas a métodos são dinamicamente vinculadas a menos que o método chamado tenha sido declarado como **`final`**
  - A vinculação estática é usada também se o método for estático (`static`) ou privado (`private`), em que ambos os modificadores não permitem sobrescrita

Suporte para programação orientada a objetos em Java (cont...)

- Diversas variedades de classes aninhadas
- Todas possuem a vantagem de serem ocultas de todas as classes em seus pacotes, exceto para a classe aninhadora
- Classes não estáticas aninhadas diretamente em outras são *classes internas*
- Classes aninhadas podem ser anônimas
- Uma classe aninhada local é definida em um método de sua classe aninhadora

Suporte para programação orientada a objetos em Java (cont...)

- Avaliação
  - O projeto de Java para suporte à programação orientada a objetos é similar ao de C++
  - Java não suporta programação procedural
  - Não permite classes sem pais
  - Usa vinculação dinâmica como a maneira "normal" de vincular chamadas a métodos às definições de métodos
  - Usa interfaces para fornecer uma forma simples de suporte para herança múltipla

Implementação de construções orientadas a objetos

- Duas partes interessantes e desafiadoras
  - Estruturas de armazenamento para variáveis de instância
  - Vinculações dinâmicas de mensagens a métodos

Armazenamento de dados de instâncias

- **Registro de instância de classe (RIC)** armazena o estado de um objeto
  - Estático (construído em tempo de compilação)

- Cada classe possui seu próprio RIC. Quando uma derivação ocorre, o RIC da subclasse é uma cópia do RIC da superclasse, com entradas para as novas variáveis de instância adicionadas no final

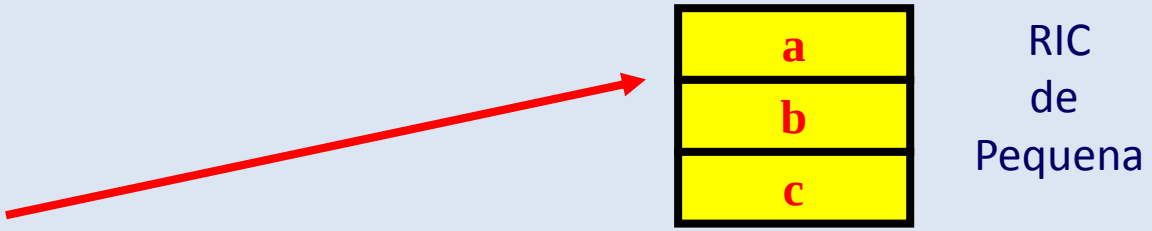
- Dado que a estrutura do RIC é estática, o acesso a todas as variáveis de instância pode ser feita como é feito nos registros (**deslocamento local**)
  - Eficiente

Armazenamento de dados de instâncias

❖**Cada classe possui seu próprio RIC.**

- Exemplo

```
class Pequena
{
  public int a,b,c;
}
class Grande extends Pequena
{
  public int d,e;
}
```



Armazenamento de dados de instâncias

❖**Cada classe possui seu próprio RIC.**

- Exemplo

```
class Pequena
{
  public int a,b,c;
}
class Grande extends Pequena
{
  public int d,e;
}
```

Armazenamento de dados de instâncias

❖**Cada classe possui seu próprio RIC.**

❑ Exemplo

```
class Pequena
{
  public int a,b,c;
}
class Grande extends Pequena
{
  public int d,e;
}
```

Quando uma derivação ocorre, o RIC da subclasse é uma cópia do RIC da superclasse, com entradas para as novas variáveis de instância adicionadas no final

Vinculação dinâmica de chamadas a métodos

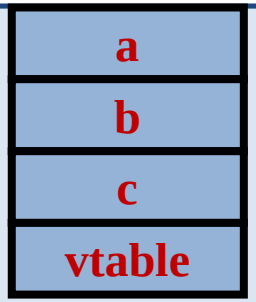
- Métodos em uma classe que são estaticamente vinculados não precisam se envolver no RIC para a classe
- Métodos que serão vinculados dinamicamente devem ter entradas nessa estrutura
  - Chamadas a um método podem então ser conectadas ao código correspondente por meio desse ponteiro no RIC
  - A estrutura de armazenamento para a lista é frequentemente chamada de uma tabela de métodos virtual (vtable)
  - As chamadas a métodos podem ser representadas como deslocamentos a partir do início da vtable

Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```

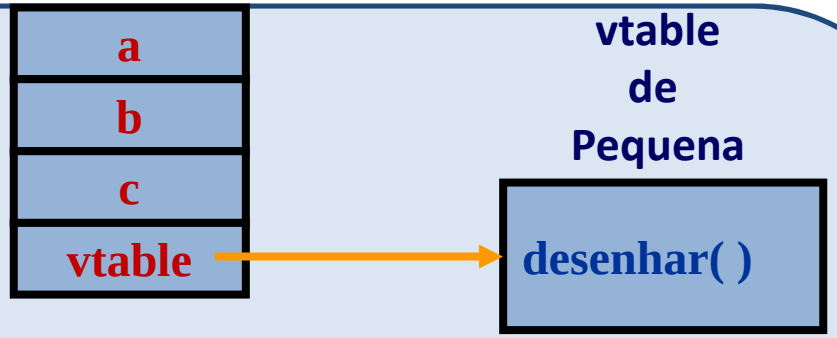


Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```

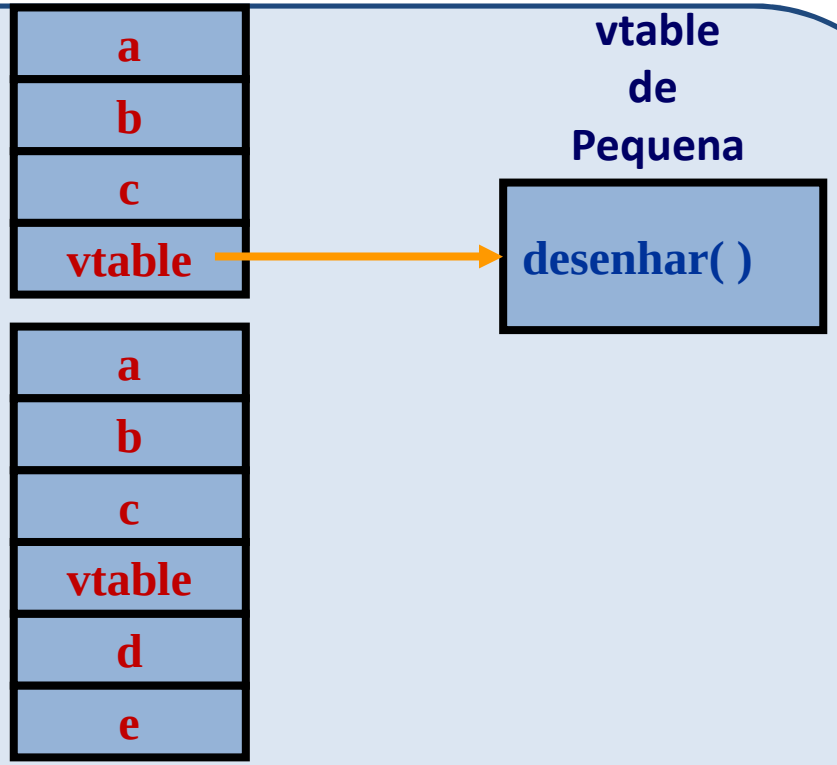


Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```

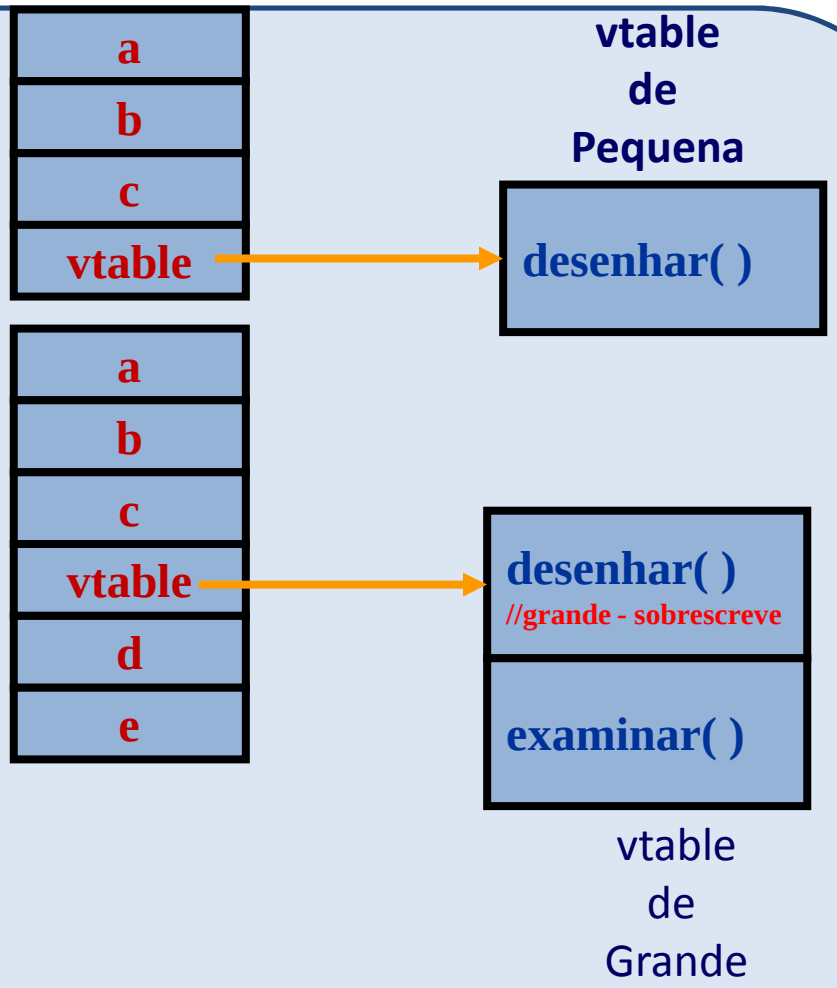


Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```

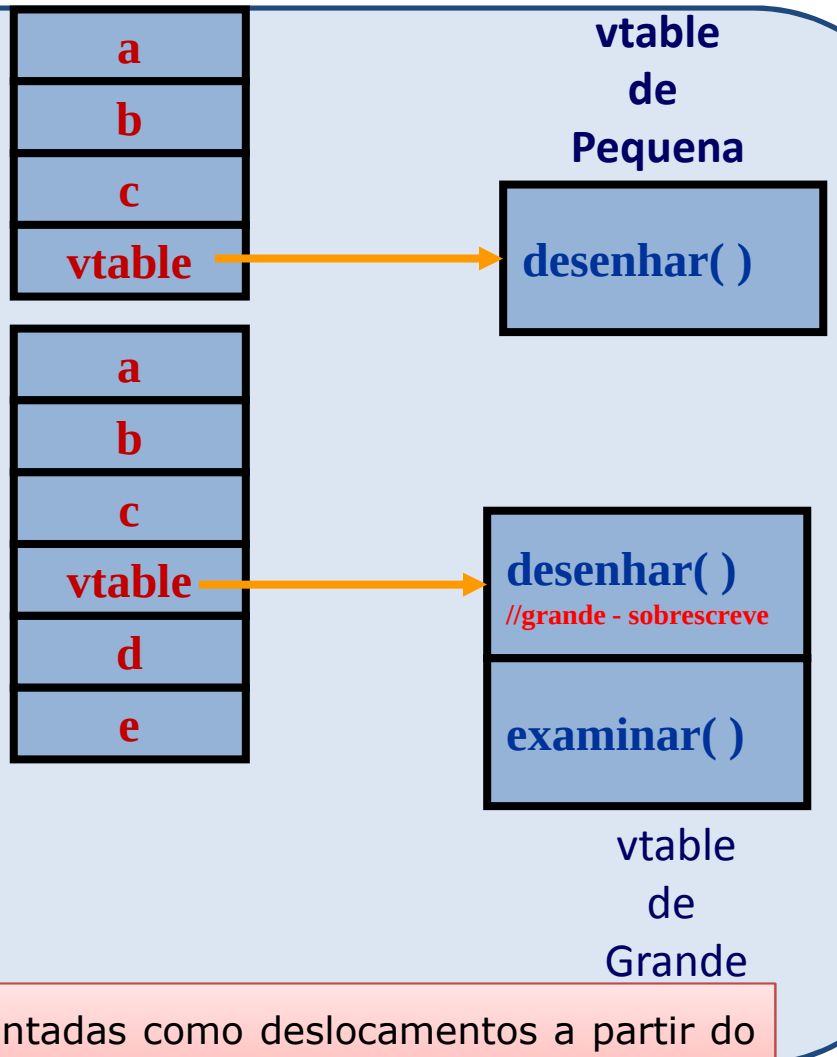


Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```



CHAMADA POLIMÓRFICA

objeto.desenhar( );

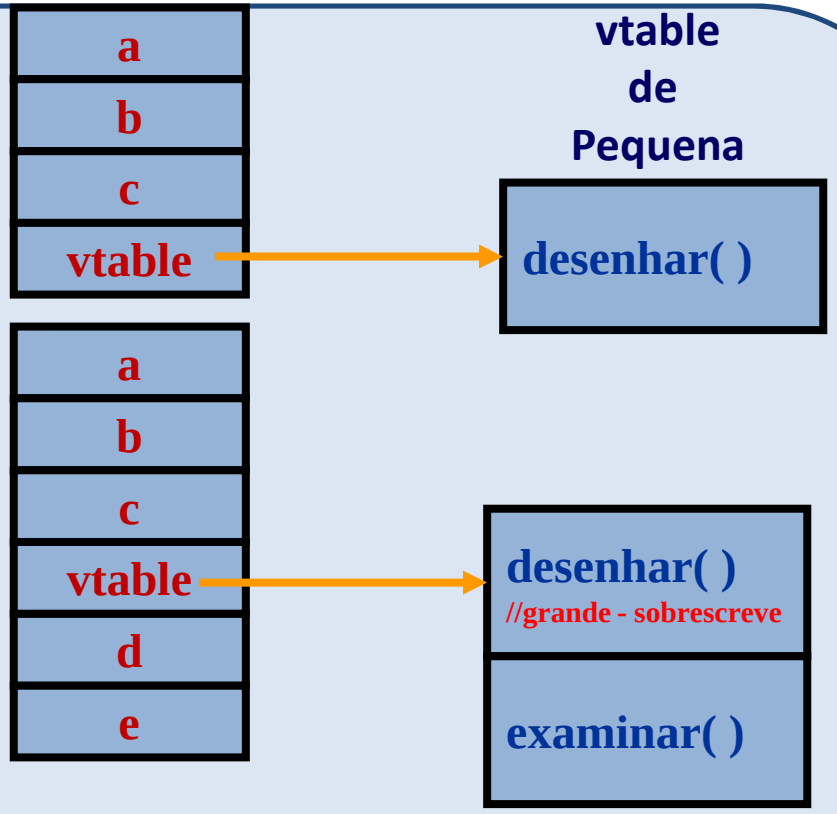
As chamadas a métodos podem ser representadas como deslocamentos a partir do início da vtable

Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( )
  public void examinar( )
}
```



Instância de Pequena

CHAMADA POLIMÓRFICA

objeto.desenhar( );

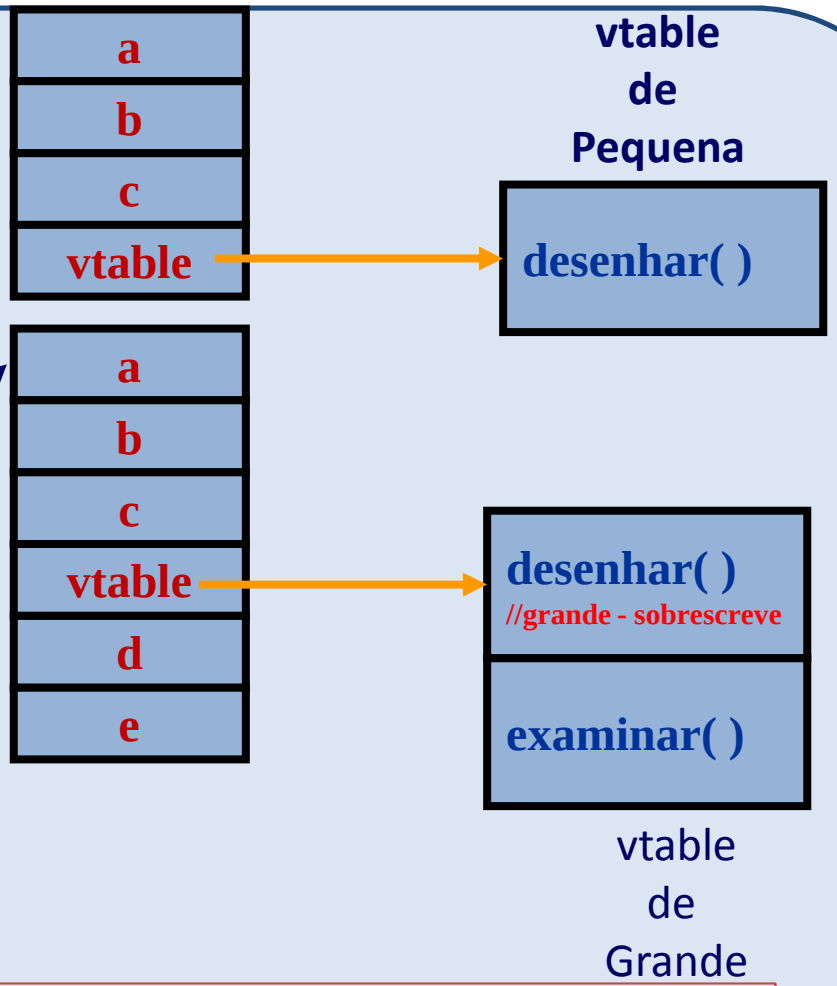
As chamadas a métodos podem ser representadas como deslocamentos a partir do início da vtable

Vinculação dinâmica de chamadas a métodos

- Exemplo

```
class Pequena
{
  public int a,b,c;
  public void desenhar( ) {...}
}

class Grande extends Pequena
{
  public int d,e;
  public void desenhar( ) {...}
  public void examinar( ) {...}
}
```



CHAMADA POLIMÓRFICA

objeto.desenhar( );

As chamadas a métodos podem ser representadas como deslocamentos a partir do início da vtable

Resumo

- A programação orientada a objetos envolve três conceitos fundamentais: tipos de dados, abstratos, herança e vinculação dinâmica
- Questões de projeto: exclusividade de objetos, subclasses e subtipos, verificação de tipo e polimorfismo, herança simples e múltipla, vinculação dinâmica, liberação explícita ou implícita de objetos e classes aninhadas
- C++ tem dois sistemas de tipos (híbrida)

Resumo

- Java não é uma linguagem híbrida como C++; suporta apenas programação orientada a objetos
- Implementar linguagens de programação orientada a objetos envolve novas estruturas de dados

Exercício

- Implemente uma classe que contenha duas classes aninhadas (uma estática e uma interna) em C++ e Java
- Faça um exemplo de herança múltipla em C++
- Faça um exemplo de interface (Java)