Programação Orientada a Objetos em

Flávio Miguel Varejão

Departamento de Informática

UFES

Conteúdo

- Aula 1
  - Discussão do Trabalho
  - Polimorfismo
  - RTTI e Interfaces
  - Herança Múltipla

Conteúdo

- Aula 2
  - Classes Internas
  - Exceções
  - Arquivos e Streams

Conteúdo

- Aula 3
  - Coleções
  - Novo for
  - Enumerações
  - Propriedades
  - Polimorfismo Paramétrico

Trabalho

- Scanner
  - Leitura de Arquivo

```
Scanner sc = new Scanner(new File(“mestre.txt"));
while (sc.hasNextLine()) {
    String linha = sc.nextLine();
    int i = Integer.parseInt(linha);
}
```

Trabalho

- Formatter e printf
  - Escrita em Arquivo com printf

    PrintStream ps = new PrintStream (new File(“relat.txt"));

    ps.printf (“idade = %d;%n”, idade);
  - Escrita em Arquivo com Formatter

    Formatter f = new Formatter (new File(“relat.txt"));

    f.format (“idade = %d;%n”, idade);

Trabalho

- Professor DE
  - Método na classe MembroUFES

    ```
    boolean professorDE() {return false;}
    ```
  - Método sobrescrito na classe Professor

    ```
    boolean professorDE() {
        return de.compareToIgnoreCase("DE") == 0;
    }
    ```
  - Solução Deselegante!!!

Trabalho

- Ordenação Única
  - Classe abtrata com método compara

```
public abstract class ComparaUFES {
    public abstract boolean compara (MembroUFES x,
    MembroUFES y);
}
```

  - Classes concretas com implementação de compara

```
public class ComparaMatricula extends ComparaUFES {
  public boolean compara (MembroUFES x, MembroUFES y) {
     return x.menorMatric(y);
  }
}
public class ComparaIdade extends ComparaUFES {
  public boolean compara (MembroUFES x, MembroUFES y) {
     return x.menorIdade(y);
  }
}
```

Trabalho

- Ordenação Única
  - Método ordena na classe GrupoUFES

```
void ordena(ComparaUFES c, boolean crescente) { ...
    if (crescente) {
        if (c.compara(p[i],p[i+1])) { ...
}
```

  - Chamada de ordena passando comparador apropriado em Cadastro

```
gp.ordena (new ComparaMatricula(), true);
gp.ordena (new ComparaMatricula(), false);
gp.ordena (new ComparaIdade(), false);
```

Trabalho

- Inclusão de novo MembroUFES
  - Deveria alterar apenas nos locais onde ocorre criação de objetos membros da UFES
  - Uso de Padrão Fábrica para criação de objetos
- Mudança para Cadastro de Produtos
  - Necessário reescrever quase todo o código
  - Ideal
    - Trocar as classes membros da UFES pelas classes Produto
    - Alterações nas listagens/relatórios

O Padrão Fábrica

```
public abstract class FabricaUFES {
  public static MembroUFES fabrica(String tipo) {
    if(tipo.equals("prof")) return new Professor();
    if(tipo.equals("serv")) return new Servidor();
    if(tipo.equals("pos")) return new Posgraduando();
    if(tipo.equals("grad")) return new Graduando();
    System.out.println(tipo + "desconhecido");
    return null;
  }
}

public class TestaricaFab {
  public static void main(String args[]) {
    MembroUFES f1 = FabricaUFES.fabrica("prof");
    MembroUFES f2 = FabricaUFES.fabrica("pos");
  }
}
```

O Método toString

- Qualquer objeto não primitivo tem toString
  - herdado de Object
- Chamada pelo compilador pode ser implícita
  - System.out.println (“fonte = “ + fonte);
- Classes devem implementar sua própria toString

toString de Object

```
// HMS.java
class A { int i; }
public class HMS {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] { new Float(3.14), new A(),
       new Integer(47), new HMS(), new Double(11.11) });
    f(new Object[] {new A(), new A(), new A()});
  }
}
```

Downcast

- Upcast
  - conversão da classe derivada para base
  - sempre segura pois derivada possui interface de base
- Downcast
  - conversão da classe base para derivada
  - sistema de tipos pode ser violado porque classe derivada pode estender classe base

Downcast

- Usos
  - estruturas de dados genéricas
    - Exemplo das coleções
  - comportamento dinâmico determinado em função da categoria
    - Imprimir quantidade de itens no acervo de cada produto
  - OO Puro X Downcast
    - Colocar métodos na superclasse X identificar o tipo

RTTI

- Identificação de Tipo em Tempo de Execução
  - objetiva impedir que se faça downcast para um objeto de tipo errado
  - operação de cast é checada durante a execução
  - dispara exceção ClassCastException caso seja incorreta

RTTI

```
// RTTI.java
// Downcasting & RTTI

class Util {
  public void f() {}
  public void g() {}
}

class MuitoUtil extends Util {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}
```

RTTI

```
public class RTTI {
  public static void main(String[] args) {
    Util[] x = {
      new Util(),
      new MuitoUtil()
    };
    x[0].f();
    x[1].g();
    // Erro de compilacao: nao e' metodo de Util
    //! x[1].u();
    ((MuitoUtil)x[1]).u(); // Downcast/RTTI
    ((MuitoUtil)x[0]).u(); // Erro de execucao
  }
}
```

RTTI

- É possível usar mecanismo de RTTI antes da operação de downcast
  - uso do operador instanceof

```
if(x instanceof Cachorro)
  ((Cachorro)x).late();
```

RTTI

```
class Gato {
  private int numeroGato;
  Gato(int i) {
    numeroGato = i;
  }
  void miar() {
    System.out.println("Gato #" +
            numeroGato + " miou!");
  }
  void brincar() {
    System.out.println ("Novelo");
  }
}
```

RTTI

```
class Cao {
  private int numeroCao;
  Cao(int i) {
    numeroCao = i;
  }
  void latir() {
    System.out.println("Cao  #" +
            numeroCao + " latiu!");
  }
  void brincar() {
    System.out.println ("Osso");
  }
}
```

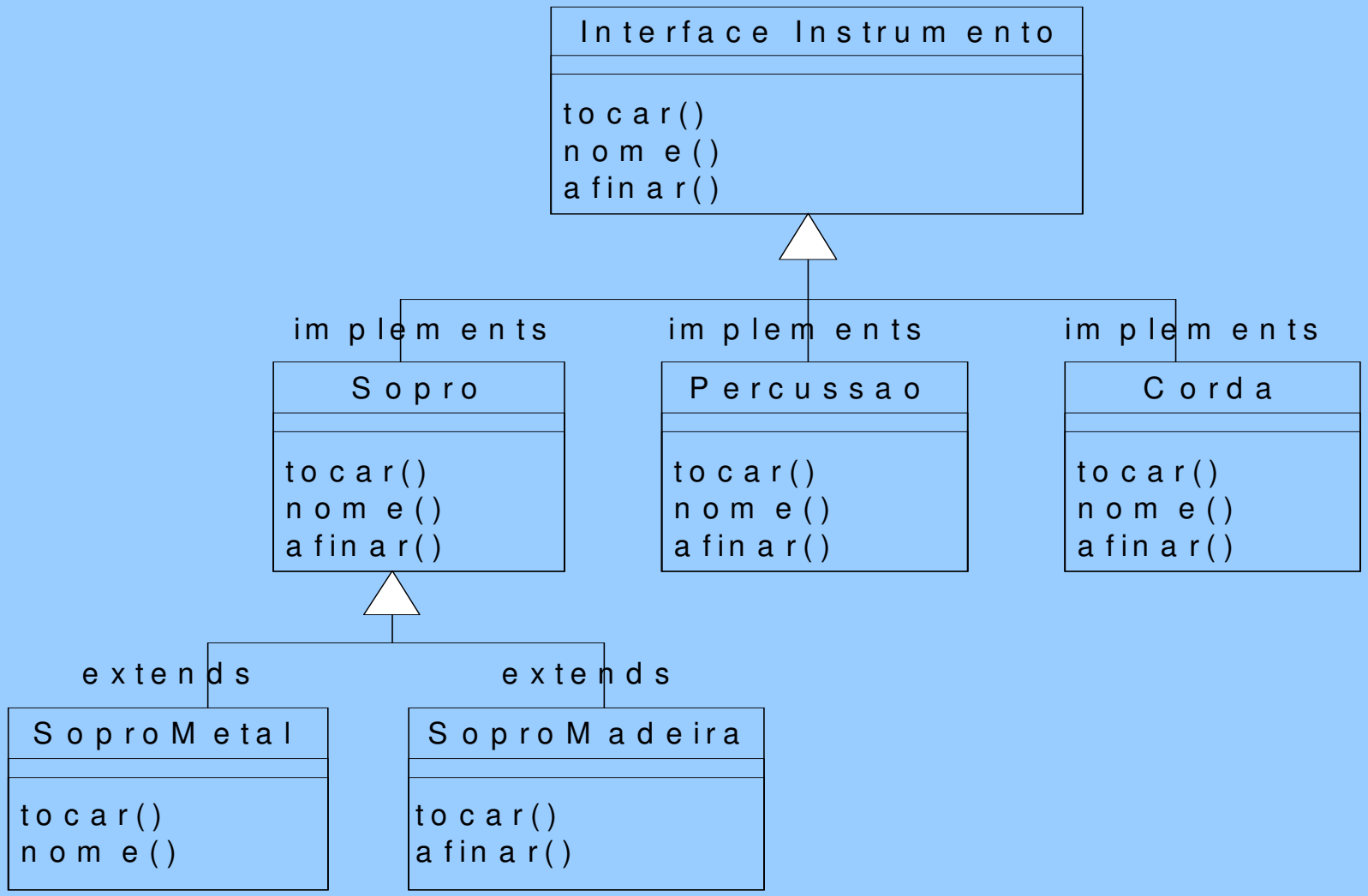
RTTI

```
public class GatosECaes {
  public static void main(String[] args) {
    Object [] Domesticos = new Object [10];
    for(int i = 0; i < 10; i = i + 2)
      Domesticos[i] = new Gato(i);
    for(int i = 1; i < 10; i = i + 2)
      Domesticos[i] = new Cao(i);
    for(int i = 0; i < Domesticos.length; i++) {
      if (Domesticos[i] instanceof Gato)
        ((Gato)Domesticos[i]).miar();
      if (Domesticos[i] instanceof Cao)
        ((Cao)Domesticos[i]).latir();
    }
    //! ((Cao) Domesticos[0]).brincar();
  }
}
```

Interfaces

- Classe Abstrata Pura
  - só possui atributos de classe (static e final)
  - todos os métodos públicos e abstratos
  - uso da palavra interface
- Classes implementam interfaces
  - devem declarar métodos como públicos
  - uso de implements

Interfaces


InterfaceInstrumento
tocar()
nome()
afinar()
implements
implements
implements
Sopro
tocar()
nome()
afinar()
Percussao
tocar()
nome()
afinar()
Corda
tocar()
nome()
afinar()
extends
extends
SoproMetal
tocar()
nome()
SoproMadeira
tocar()
afinar()

Interfaces

```
interface Instrumento {
 int i = 5; // static & final
 // não pode haver implementação dos métodos:
 void tocar();  // automaticamente publicos
 String nome();
 void afinar();
}

class Sopro implements Instrumento {
 public void tocar() {
  System.out.println("Sopro.tocar()");
 }
 public String nome() { return "Sopro"; }
 public void afinar() {}
}
```

Interfaces

```
class Percussao implements Instrumento {
 public void tocar() {
  System.out.println("Percussao.tocar()");
 }
 public String nome() { return "Percussao"; }
 public void afinar() {}
}
class Corda implements Instrumento {
 public void tocar() {
  System.out.println("Corda.tocar()");
 }
 public String nome() { return "Corda"; }
 public void afinar() {}
}
```

Interfaces

```
class SoproMetal extends Sopro {
 public void tocar() {
  System.out.println("SoproMetal.tocar()");
 }
 public void afinar() {
  System.out.println("SoproMetal.Afinando()");
 }
}
class SoproMadeira extends Sopro {
 public void tocar() {
  System.out.println("Madeira.tocar()");
 }
 public String nome() { return "SoproMadeira"; }
}
```

Interfaces

```
public class MusicaInterface {
 static void melodia(Instrumento i) {
  // ...
  i.tocar();
 }
 static void sinfonia(Instrumento[] e) {
  for(int i = 0; i < e.length; i++)
  melodia(e[i]);
 }
```

Interfaces

```
public static void main(String[] args) {
  Instrumento[] orquestra = new Instrumento[5];
  int i = 0;
  // Upcasting:
  orchestra[i++] = new Sopro();
  orchestra[i++] = new Percussao();
  orchestra[i++] = new Corda();
  orchestra[i++] = new SoproMetal();
  orchestra[i++] = new SoproMadeira();
  sinfonia(orquestra);
 }
}
```

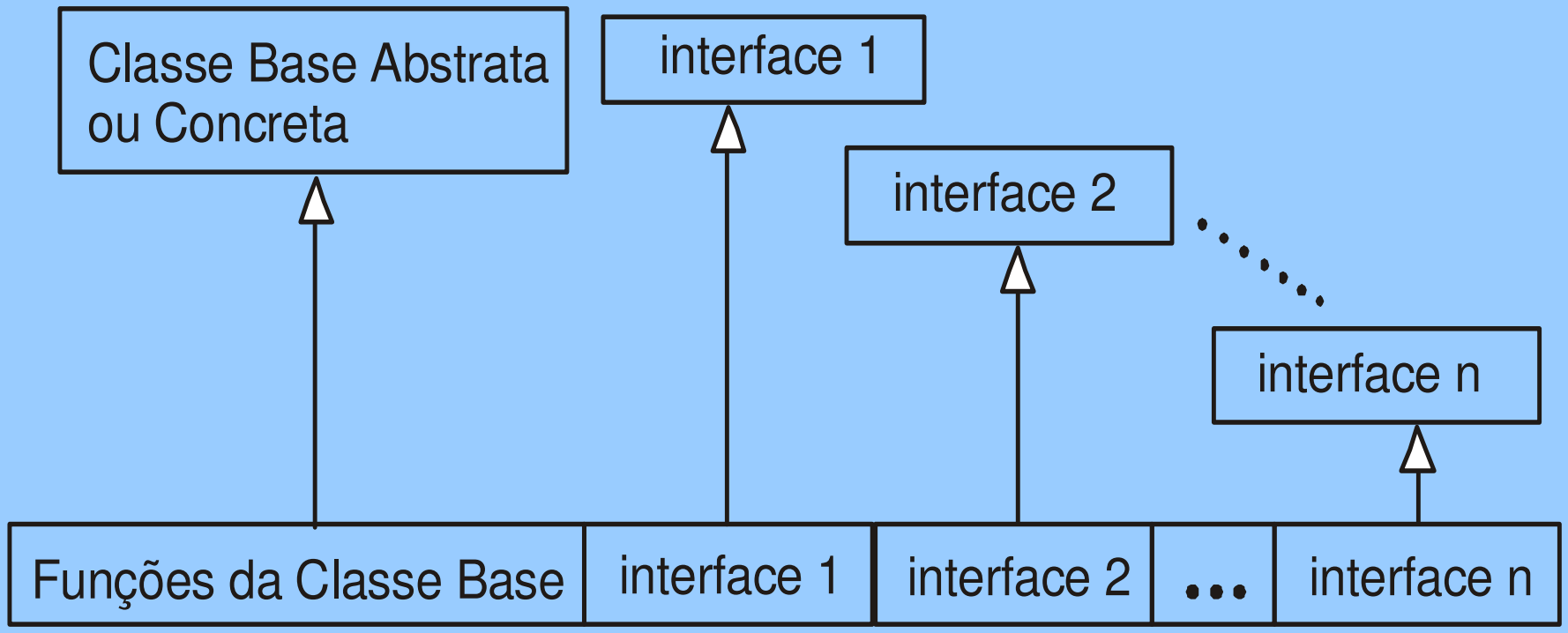
Herança Múltipla

- Herança Simples
  - pode ser restritiva
    - Exemplo de MembroUniv, Estudante, Assalariado e AssalariadoEstudante
- Herança Múltipla
  - traz problemas
    - conflito de nomes
    - herança repetida

Herança Múltipla

- Java
  - Herança simples de Classes
  - Herança múltipla de Interfaces
- Evita em grande parte problema de conflitos
- Permite múltiplos supertipos
- Não fornece reuso de código

Herança Múltipla



Herança Múltipla

```
interface PoderLutar {
 void lutar();
}
interface PoderNadar {
 void nadar();
}
interface PoderVoar {
 void voar();
}
class PersonagemDeAcao {
 public void lutar() {}
}
class Heroi extends PersonagemDeAcao
 implements PoderLutar, PoderNadar, PoderVoar {
 public void nadar() {}
 public void voar() {}
}
```

Herança Múltipla

```
public class Aventura {
 static void t(PoderLutar x) { x.lutar(); }
 static void u(PoderNadar x) { x.nadar(); }
 static void v(PoderVoar x) { x.voar(); }
 static void w(PersonagemDeAcao x) { x.lutar(); }

 public static void main(String[] args) {
  Heroi h = new Heroi();
  t(h); // é tratado como PodeLutar
  u(h); // é tratado como PodeNadar
  v(h); // é tratado como PodeVoar
  w(h); // é tratado como PersonagemDeAcao
 }
}
```

Colisão de Nomes

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 10; } // sobrecarga
}
class C3 extends C implements I2 {
  public int f(int i) { return 100; } // sobrecarga
}
class C4 extends C implements I3 {
  public int f() { return 1000; } // sobrescrita
}
// Problemas:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {}
```

Herança de Interfaces

```
interface Monstro {
 void ameaçar();
}
interface MonstroPerigoso extends Monstro {
 void destruir();
}
interface Letal {
 void matar();
}
class DragonZilla implements MonstroPerigoso {
 public void ameaçar() {}
 public void destruir() {}
}
```

Herança de Interfaces

```
interface Vampiro
 extends MonstroPerigoso, Letal {
 void beberSangue();
}
class ShowDeHorror {
 static void u(Monstro b) { b.ameaçar(); }
 static void v(MonstroPerigoso d) {
  d.ameaçar();
  d.destruir();
 }
 public static void main(String[] args) {
  DragonZilla if2 = new DragonZilla();
  u(if2);
  v(if2);
 }
}
```

Inicialização de Atributos de Interfaces

```
// ValoresAleatorios.java
import java.util.*;
public interface ValoresAleatorios {
 int rint = (int)(Math.random() * 10);
 long rlong = (long)(Math.random() * 10);
 float rfloat = (float)(Math.random() * 10);
 double rdouble = Math.random() * 10;
}
```

A Interface Comparable

```
import java.util.*;
public class TipoComp implements Comparable {
  int i;
  int j;
  public TipoComp(int n1, int n2) {
    i = n1;
    j = n2;
  }
  // sobrecarga do método toString:
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
```

A Interface Comparable

```
// implementação do método compareTo
  public int compareTo(Object rv) {
    TipoComp t = (TipoComp) rv;
    int rvi = t.i;
    int rvj = t.j;
    return (i < rvi ? -1 :
              (i == rvi ?
                 (j < rvj ? -1 :
                    (j == rvj ? 0 : 1) ):
                       1));
  }
```

A Interface Comparable

```
  public static void main(String[] args) {
    TipoComp[] a = new TipoComp[6];
    // preenche o vetor com objetos TipoComp:
    a[0] = new TipoComp(5,1);
    a[1] = new TipoComp(3,2);
    a[2] = new TipoComp(9,4);
    a[3] = new TipoComp(3,1);
    a[4] = new TipoComp(11,5);
    a[5] = new TipoComp(8,3);
    Arrays.sort(a);
    for (int n=0; n<=5; n++){
      // imprime o vetor depois de ordenado
      System.out.println("a["+n+"]:"+a[n]+" ");
    }
    System.out.println (Arrays.binarySearch(a,a[3]));
  }
}
```

A Interface Comparator

```
import java.util.*;
public class TipoComp implements Comparator {
  int i;
  int j;
  public TipoComp(int n1, int n2) {
    i = n1;
    j = n2;
  }
  // sobrecarga do método toString:
  public String toString() {
    return "[i = " + i + ", j = " + j + "]";
  }
```

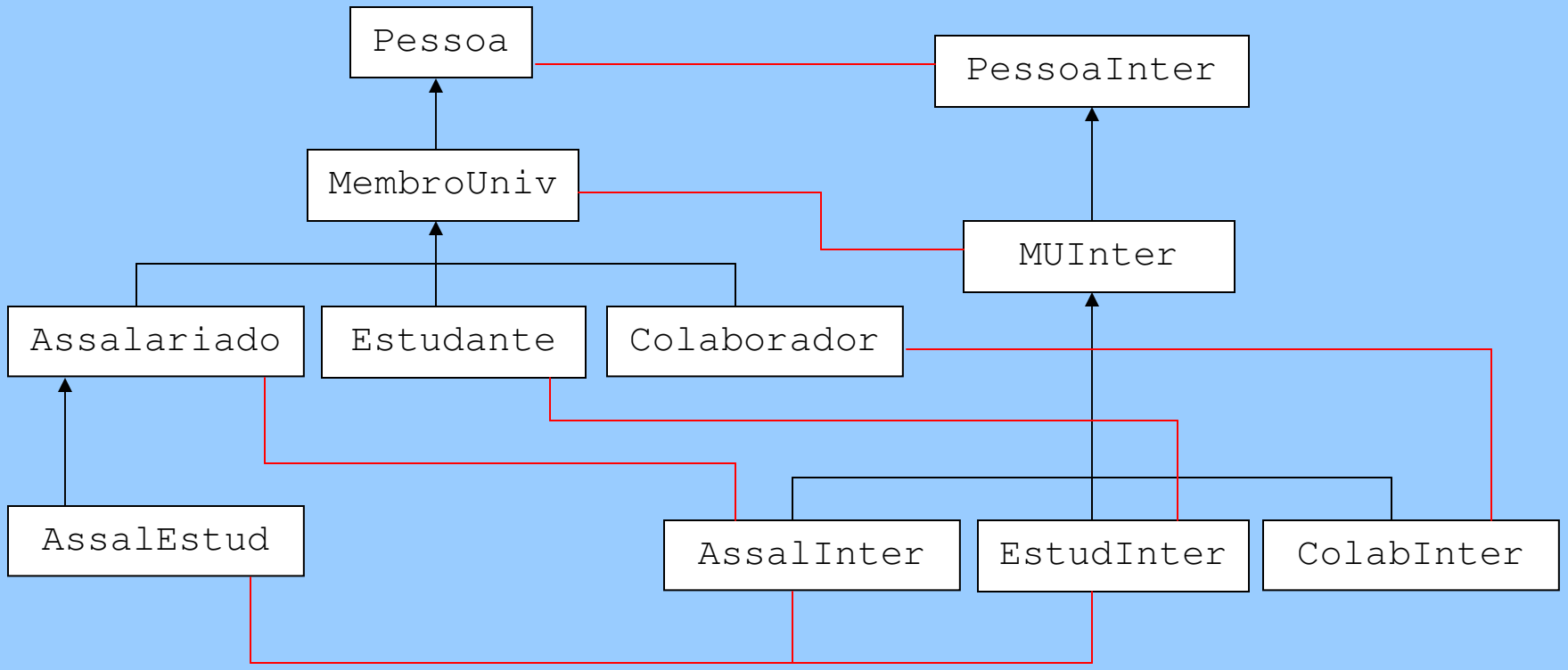
A Interface Comparator

```
// implementação do método compare
public int compare (Object rv1,
                    Object rv2) {
  TipoComp t = (TipoComp) rv1;
  TipoComp c = (TipoComp) rv2;
  return (t.i < c.i ? -1 :
            (t.i == c.i ?
              (t.j < c.j ? -1 :
                (t.j == c.j ? 0 : 1)):
                  1));
}
```

A Interface Comparator

```
    public static void main(String[] args) {
     TipoComp[] a = new TipoComp[6];
     // preenche o vetor com objetos TipoComp:
     a[0] = new TipoComp(5,1);
     a[1] = new TipoComp(3,2);
     a[2] = new TipoComp(9,4);
     a[3] = new TipoComp(3,1);
     a[4] = new TipoComp(11,5);
     a[5] = new TipoComp(8,3);
     TipoComp m = new TipoComp(0,0);
     Arrays.sort(a, m);
     for (int n=0; n<=5; n++){
       // imprime o vetor depois de ordenado
       System.out.println("a["+n+"]:"+a[n]+" ");
     }
     System.out.println (
          Arrays.binarySearch(a,a[3],m));
   }
}
```

Exemplo dos Membros da Universidade



Membro de Universidade

```
Data hoje = new Data();
for (int j=0; j<gp.tamanho(); j++) {
  PessoaInter p = (PessoaInter) gp.obtem(j);
  if(hoje.igualDiaMes(p.retornaData())){
       System.out.println (p.retornaNome()+
            " e' aniversariante!");
  }
  if (p instanceof EstudInter) {
     EstudInter a = (EstudInter) p;
     if (a.bomAluno()) {
        System.out.println(a.retornaNome());
  }
}
```