Instituto de Computação

Técnicas de Programação Avançada

TCC-00.174

Prof.: Anselmo Montenegro

www.ic.uff.br/~anselmo

anselmo@ic.uff.br

Conteúdo: Polimorfismo

Orientação a Objetos

Revisão ...

A classe provê a estrutura para a construção de objetos

Um objeto é uma instância de uma classe

Possui características (atributos) e um estado (valores de seus atributos)

Expõe o seu comportamento através de métodos (funções)

Orientação a Objetos

Revisão ...

Um programa OO é um conjunto de objetos que colaboram entre si para a solução de um problema

Objetos colaboram através de trocas de mensagens

A troca de mensagem é realizada através da chamada de um método

Encapsulamento:

Princípio pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação

Ocultação da Informação:

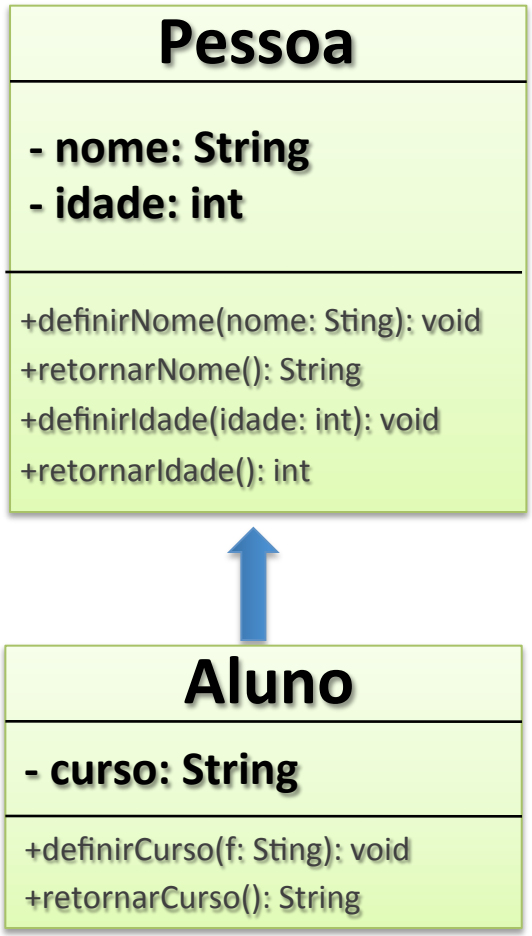
Princípio pelo qual cada componente deve manter oculta sob sua guarda uma decisão de projeto única.

Para a utilização desse componente, apenas o mínimo necessário para sua operação deve ser revelado (tornado público).

Diferentes classes podem ter diversas semelhanças:
duas ou mais classes poderão compartilhar os mesmos atributos e/ou os mesmos métodos.

Herança: Permite a uma classe herdar características (atributos) e o comportamento (métodos) de outra classe

- Superclasse
- Subclasse
- Ancestral
- Descendente



Instâncias de Aluno

João
25
Sistemas de Informação

Maria
20
Sistemas de Informação

Orientação a Objetos

Herança

```
//SuperClass.java
public class SuperClass
{
   // Atributos e métodos
}

//SubClass.java
public class SubClass extends SuperClass
{
   // Atributos e métodos
}
```

Orientação a Objetos

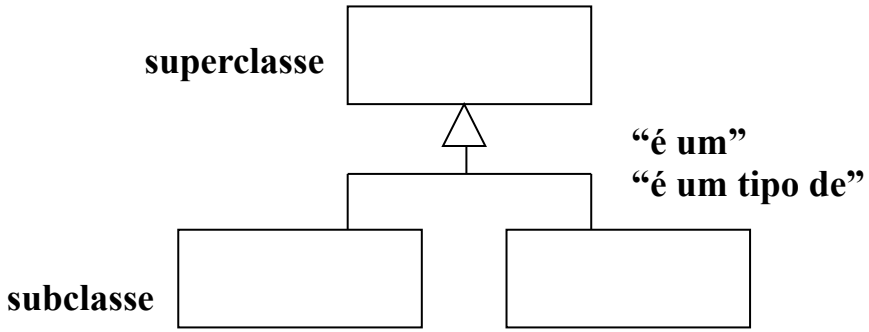
Herança

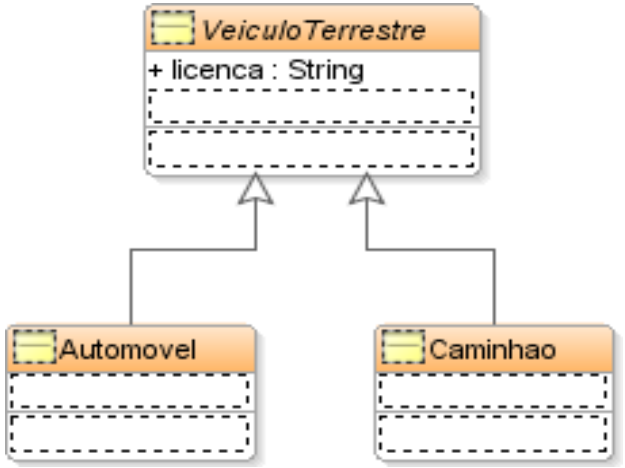
```
class Pessoa {
    String  nome;
    int    idade;

    void definirNome(String valor) {
            nome = valor;
      }

    String retornarNome() {
            return nome;
      }

    void definirIdade(int valor) {
            idade = valor;
      }

    int retornarIdade() {
            return idade;
      }
}
```

```
class Aluno extends Pessoa {
    String   curso;

    void definirCurso(String valor) {
            curso = valor;
      }

    String retornarCurso() {
            return curso;
      }

}
```

Uma generalização é um relacionamento entre itens gerais (superclasse) e itens mais específicos (subclasses)

É representada por uma linha sólida com um triângulo vazado apontando para o item mais geral.





Classes **Abstratas** Vs Classes **Concretas**

Uma *classe abstrata* é uma classe que *não tem instâncias diretas*, mas cujas classes descendentes podem ter instâncias diretas

Uma *classe concreta* é uma *classe que pode ser instanciada*

Classes **Abstratas** Vs **Interfaces**

A *classe abstrata* pode possuir métodos não abstratos, bastando ter apenas um método abstrato para ser considerada como tal

Um interface *apenas propõe os métodos que devem ser implementados* pelas classes que desejarem

Uma interface define um tipo


Identidade
Classificação
Encapsulamento
Hereditariedade
Ligação Dinâmica
Polimorfismo

Tipo – conjunto de valores associado a um conjunto de operações que podem ser aplicados sobre aqueles valores

Objeto de dados – valor de um certo tipo armazenado em uma área de memória

As operações definidas para um tipo são a única forma de manipular objetos de dados

Qualquer outra tentativa de manipulação causa um erro de tipo

Um programa é seguro com relação a tipos se não contém erros de tipo

Tipos de dados – componentes semânticos fundamentais de uma linguagem de programação que procuram capturar a natureza dos dados manipulados pelo programa

Linguagens de programação diferem em relação a forma com a qual manipulam os tipos

Sistema de tipos – conjunto de regras de uma linguagem que devem ser seguidas para definir e manipular os dados do programa

Objetivo: evitar a escrita de programas não seguros em relação aos tipos considerados

Sistema de tipos forte – garante que programas escritos segundo as regras de tipagem não geram erros de tipo

Linguagem fortemente tipada – linguagem que possui um sistema de tipos forte subjacente

Linguagem fracamente tipada – não possui um sistema de tipos forte

Linguagem tipada estaticamente – os tipos de toda a expressão da linguagem são conhecidos em tempo de compilação (usam um sistema de tipos estático)

Linguagem tipada dinamicamente – os tipos das expressões podem vir a ser conhecidos somente em tempo de execução

Sistema de tipo estrito – se uma operação espera um operando de tipo ***T***, então somente pode ser invocada sobre um parâmetro de tipo ***T***

Compatibilidade de tipos – algumas linguagens fornecem maior flexibilidade definindo condições nas quais um operando de outro tipo, dito compatível, pode ser utilizado sem violar a segurança de tipos

Quando compatibilidade de tipos é usada, é necessário algum procedimento de checagem de tipos

A compatibilidade de tipos pode ser nominal (por nome) ou estrutural (por estrutura)

Compatibilidade por nome – somente tipos com mesmo nome são compatíveis

Compatibilidade estrutural – dois tipos T_1 e T_2 são compatíveis se:

- T_1 é compatível por nome com T_2;
- T_1 e T_2 são definidos aplicando o mesmo construtor de tipos sobre componentes correspondentes estruturalmente compatíveis;

```
Type s1 is  struct{
int y;
int w;
}
Type s2 is  struct{
int y;
int w;
}
Type s3 is  struct{
int y;
}
S3 func (s1 z){
...
}
```

```
s1 a,x;
s2 b;
s3 c;
int d;

a = b; -- (1)
x = a; -- (2)
c= func(b); -- (3)
d = func(a); -- (4)
```

Considerando compatibilidade de nomes – instruções (1), (3) e (4) apresentam erro de tipo

Considerando compatibilidade estrutural – instruções (1), (2) e (3) são corretas

Um subtipo ST de um tipo T é um subconjunto dos valores associados as mesmas operações de T

Assume-se que as operações de T são herdadas automaticamente por ST

Uma linguagem de programação munida de subtipos deve poder definir:

Um modo de definir subconjuntos de um dado tipo

Regras de compatibilidade entre um subtipo e seu supertipo

Sistemas de tipos Monomórficos vs Polimórficos

Em linguagens tipadas estaticamente, uma entidade de programa (constante, variável ou rotina) tem um tipo especifico que é definido por uma declaração

Toda operação nessas linguagens requer operandos de tipo exatamente igual ao especificado na definição da operação

Esse sistema de tipos é definido como monomórfico

Polimorfismo é uma palavra grega que significa múltiplas formas

É uma característica de linguagens de programação em que valores de diferentes tipos de dado podem ser manipulados por uma interface comum

Em uma linguagem de programação polimórfica objetos podem pertencer a mais de um tipo

Rotinas (funções) podem aceitar parâmetros de diferentes tipos como parâmetros formais

Praticamente todas as linguagens modernas desviam-se do monorfismo puro

Exemplos de desvio de monomorfismo puro:

- Coerção
- Sobrecarga de operadores
- Compatibilidade de tipos
- Subtipagem

Coerção – denota uma conversão implícita durante o tempo de compilação ou execução (*run time*) de um subtipo para um supertipo

Exemplo: expressões que misturam tipos numéricos na linguagem C

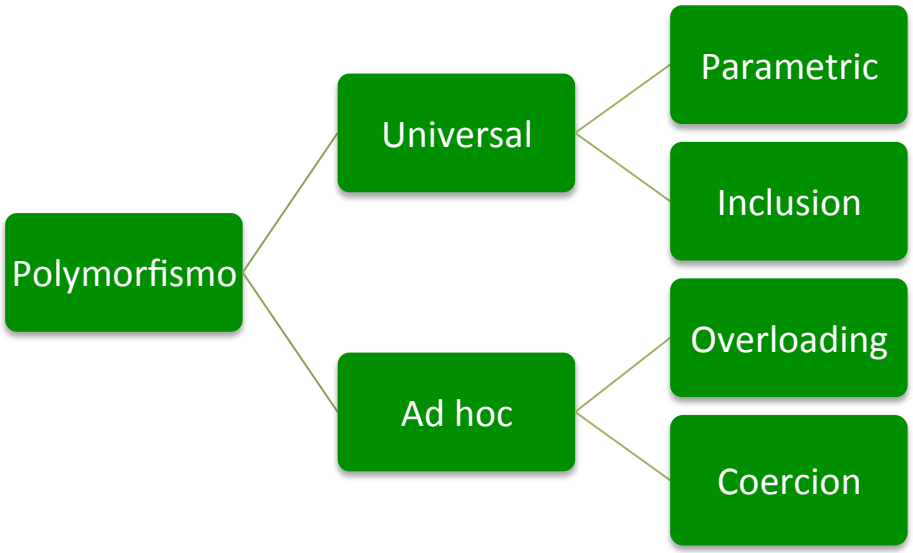
```
double d; long l; int i;
if (d > i) d = i;
if (i > l) l = i;
if (d == l) d *= 2;
```

Desvios de monomorfismo - Sobrecarga

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.2);
System.out.println("Result of ob.test(123.2): "
+ result);
}
}
```

Linguagens diferem pelos diferentes tipos de polimorfismo (considera-se aqui polimorfismo funcional)



Polimorfismo Ad Hoc – trabalha em um conjunto finito e pequeno de tipos e pode se comportar de forma diferente para cada tipo

Polimorfismo Universal – trabalha uniformemente em um conjunto infinito de tipos os quais tem estrutura comum

Exemplos de polimorfismo ad hoc

Coerção – é a operação que converte o argumento de uma função para o tipo que a função espera

Sobrecarga (overload) de métodos – o mesmo nome de função é usado para denotar diferentes funções e em cada contexto seu nome é unicamente determinado

Ex.:
```
println();
    void println(int arg);
    void println(float arg);
    void println(String arg);
```

Orientação a Objetos

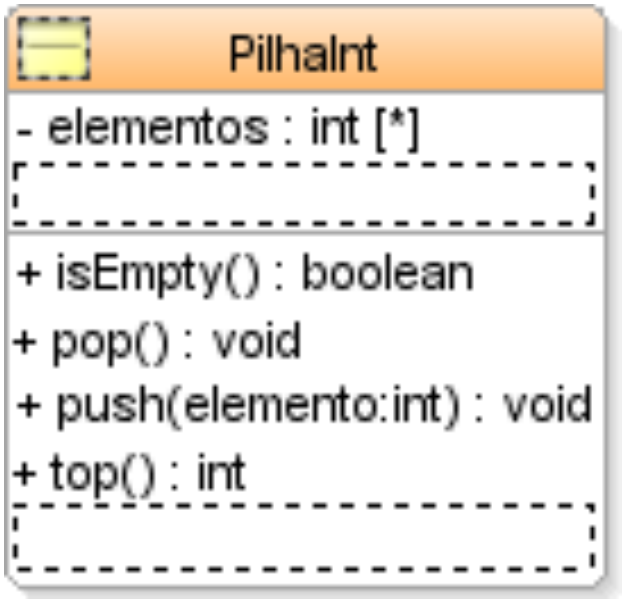
Polimorfismo Paramétrico

Tipo de polimorfismo universal mais genuíno – trabalh uniformemente sobre uma gama de tipos

No polimorfismo paramétrico um tipo de dados (ou função) pode ser escrito genericamente, de modo que ele(ou ela) possa lidar com diferentes valores de forma idêntica sem depender de seu tipo

O Polimorfismo paramétrico é uma forma de tornar a linguagem mais expressiva mantendo porém a segurança estática
de tipos

Exemplo: Classe Pilha




```
public class PilhaInt {

    private int[] elementos = new int[100];
    private int topo = -1;

    public void push(int elemento) {
        elementos[++topo] = elemento;   }

    public void pop() {
        topo--;    }

    public int top() {
        return elementos[topo];    }

    public boolean isEmpty() {
        return topo < 0;    }
}
```

Classe Pilha usando tipos genéricos

```
public class Pilha<T> {

    private Object[] elementos = new Object[100];
    private int topo = -1;

    public void push(T elemento) {
        elementos[++topo] = elemento;    }

    public void pop() {
        topo--;    }

    public T top() {
        return (T)elementos[topo];    }

    public boolean isEmpty() {
        return topo < 0;    }
}
```

É uma melhoria no sistema de tipos introduzida na J2SE 5.0 que permite um método ou tipo operar em objeto de vários tipos

Permite segurança de tipos em tempo de compilação

Adiciona segurança de tipos em tempo de compilação ao Framework de coleções e elimina a necessidade *de casting*

Uso da classe genérica Pilha

```
public class MainPilha {

    public static void main(String[] args) {

        Pilha<Integer> pilha = new Pilha<Integer>();
        pilha.push(new Integer(5));
        System.out.println(pilha.top());

    }
}
```

Problemas com tipos genéricos

```
class List<T>{
}

List<String> ls = new ArrayList<String>();

List<Object> lo = ls; // se fosse permitido

lo.add(new Object());

String s = ls.get(0); // erro em execução
```

Não podemos assumir que

`class<T1>` é supertipo de `class<T2>`

quando `class T2 extends T1`

Problemas com tipos genéricos

```
List<String> ls = new ArrayList<String>();

ls.add("string");

List<?> lo = ls;

Object o = lo.get(0);

System.out.println(o.toString());

lo.add(new Object()); // erro de compilação

String s = ls.get(0);
```

class<T> é subtipo de class<?>

Orientação a Objetos

Polimorfismo Paramétrico

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}

public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) {
  ... }
}
public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) {
  ... }
}

public class Canvas {
  public void draw(Shape s) {
    s.draw(this);  }
  public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes)
      s.draw(this);
}
```

Orientação a Objetos

Polimorfismo Paramétrico

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}

public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) {
  ... }
}
public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) {
  ... }
}

public class Canvas {
  public void draw(Shape s) {
    s.draw(this);  }
  public void drawAll(List<? extends Shape> shapes) {
    for (Shape s: shapes)
      s.draw(this);
}
```

Métodos genéricos

```
static <T> T fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) {
    c.add(o);
   }
}
```

Orientação a Objetos

Polimorfismo de inclusão

Polimorfismo de inclusão (de subtipos) permite que uma rotina (função ou método) seja aplicada a um tipo e a seus subtipos

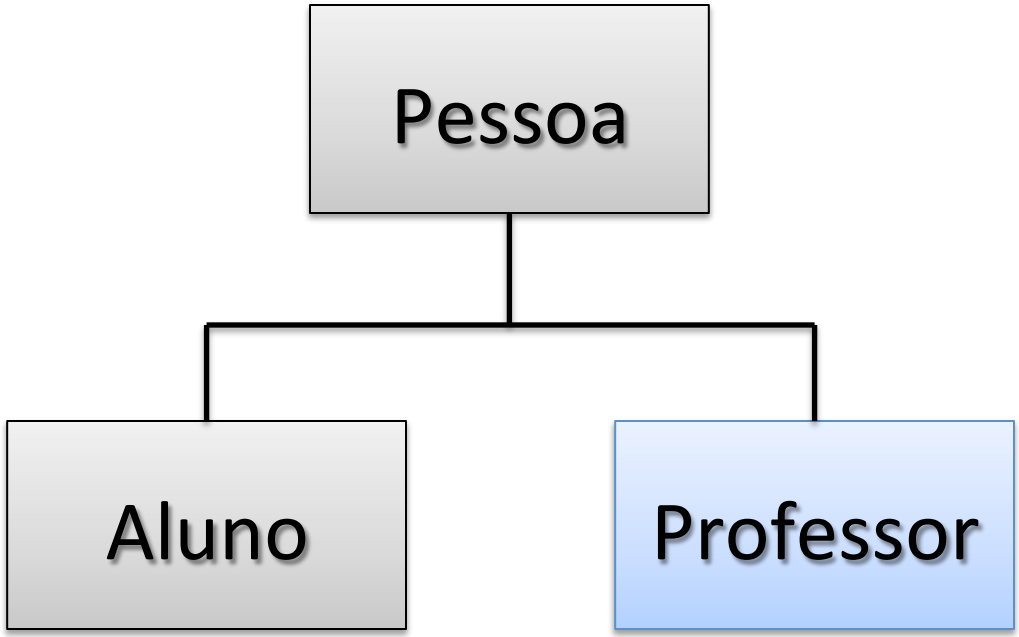
Linguagens de programação O.O proveem variáveis polimórficas que podem se referir a objetos de diferentes tipos

Linguagens fortemente tipadas restringem o polimorfismo a de tais variáveis: uma variável de uma classe ***T*** somente pode se referir a objetos de um tipo ***T*** ou de classes derivadas de ***T***

Polimorfismo de subtipos nos permite escrever programas tais que objetos que compartilham a mesma superclasse sejam tratados como se fossem objetos da superclasse.

Pode simplificar a programação

Dada uma superclasse **Pessoa**, com uma subclasse **Aluno**, desejamos adicionar outra subclasse chamada **Professor**.



Exemplo 01

```
public static main( String[] args ) {

    Aluno objetoAluno = new Aluno();
    Professor objectoProfessor = new Professor();

    // Referência da classe Pessoa recebe um objeto da classe Aluno.
    Pessoa ref = objetoAluno;

    // Chamada para o método getName() da classe Aluno.
    ref.getName();
}
```

Suponha que exista um método *getName* na superclasse Pessoa, que foi reimplementado nas subclasses Aluno e Professor.

```
public class Aluno{
    public String getName(){
        System.out.println("Nome do Aluno:" + name);
        return name;
    }
}

public class Professor{
    public String getName(){
        System.out.println("Nome do Professor:" + name);
        return name;
    }
}
```

Orientação a Objetos

Polimorfismo

```
public static main( String[] args ) {
   Aluno objetoAluno = new Aluno();
   Professor objetoProfessor = new Professor();

   // Referência da Pessoa recebe um objeto da classe Aluno.
   Pessoa ref = objetoAluno;

   // Chamada para o método getName() da classe Aluno.
   ref.getName();

   // Referência da Pessoa recebe um objeto da classe Professor.
   ref = objetoProfessor;

   // Chamada para o método getName() da classe Professor.
   ref.getName();
}
```

Exemplo 02

```
public static main( String[] args ){
    Aluno objetoAluno = new Aluno();
    Professor objetoProfessor = new Professor();

    printInfo( objetoAluno    );
    printInfo( objetoProfessor);
}
```

```
public static printInfo( Pessoa p ){
    // Chama o método getName() do objeto
    // da classe Pessoa passado por argumento
    p.getName();
}
```

```
public static main( String[] args ){
    Aluno objetoAluno = new Aluno();
    Professor objetoProfessor = new Professor();

    printInfo( objetoAluno    );
    printInfo( objetoProfessor);
}
```

Determinam se atributos e métodos poderão ser acessados por outras classes.

public (público)

private (privado)

protected (protegido)

modificador **não explícito** (package-private)

Uma classe pode ser:

- ✓ **public**
  acessada por qualquer outra classe

- ✓ **nenhum modificador** (package-private)
  acessada somente dentro do seu pacote

Atributos e Métodos podem ser:

- ✓ **public**
  acessados por qualquer outra classe.
- ✓ **nenhum modificador** (package-private)
  acessados somente dentro do seu pacote.
- ✓ **private**
  acessados somente dentro de suas próprias classes.
- ✓ **protected**
  acessados somente dentro do seus pacotes e por suas subclasses.