



Orientação a Objetos

Cleudson Souza - cdesouza@ufpa.br

Departamento de Informática

Universidade Federal do Pará

Agenda

■ Introdução a OO

- Histórico
- Motivos que influenciaram
- Vantagens da OO
- Áreas de aplicação
- O que é a Orientação a Objetos?
- Preconceitos sobre a OO
- Desenvolvimento OO
- Linguagens de Programação OO

■ Paradigma de Objetos

- Classe e Objetos
- Encapsulamento
- Herança
- Classes Abstratas
- Polimorfismo
- Delegação
- Acoplamento Dinâmico
- MetaClasses

■ Bibliografia

Histórico

■ Linguagens de Programação

- Simula 67 : linguagem projetada para simulação.
- Família de linguagens Smalltalk. Smalltalk-72 e 80.
- Maior divulgação a partir de 1986 com o 1º Workshop em Programação Orientada a Objetos e a conferência *Object-Oriented Programming Languages, Systems and Applications* (OOPSLA).
- C++
- Java

Histórico(2)

■ Metodologias

- Surgiram devido a incompatibilidade das abordagens estruturadas com a Programação Orientada a Objetos.
- Exemplo de metodologias:
 - Coad - Yourdon (1990)
 - OMT (1991)
 - OOSE (1992)
 - Fusão (1995)
 - UML (1996)

■ Banco de Dados

■ Sistemas Operacionais

■ ...

Motivos que influenciaram

- Avanços na tecnologia de arquiteturas de computadores, suportando sofisticados ambientes de programação e interfaces homem-máquina.
- Avanços na área de linguagens de programação como modularização, ocultamento de informação, etc.
- Crise do Software: termo utilizado para descrever problemas associados ao modo como o software é desenvolvido, como é feita a manutenção e como acompanhar a demanda por mais software [Pressman, 1995].

Vantagens da Tecnologia de Objetos

- Facilita a reutilização de código
- Os modelos refletem o mundo real de maneira mais aproximada:
 - Descrevem de maneira mais precisa os dados.
 - A decomposição é baseada em um particionamento natural.
 - Mais fáceis de entender e manter.
- Pequenas mudanças nos requisitos não implicam em alterações massivas no sistema em desenvolvimento.

O que é a Orientação a Objetos ?

- É um paradigma para o desenvolvimento de software que baseia-se na utilização de componentes individuais (objetos) que colaboram para construir sistemas mais complexos. A colaboração entre os objetos é feita através do envio de mensagens.
- Um paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver problemas dentro desta fronteira. Um paradigma ajuda-nos a organizar a e coordenar a maneira como olhamos o mundo.

O que é a Orientação a Objetos ?

- O paradigma de objetos baseia-se nos seguintes conceitos:
 - Classes
 - Objetos
 - Herança
 - Polimorfismo e
 - *Binding* Dinâmico

- Cada um destes conceitos será abordado neste curso.

“Preconceitos” sobre Orientação a Objetos

- A orientação a objetos não é uma metodologia para o desenvolvimento de interfaces gráficas amigáveis, ou seja, o paradigma de objetos não está necessariamente relacionada a programação visual.
- A orientação a objetos não elimina a necessidade de implementar os sistemas, e nem está relacionadas apenas a fase de implementação.
- A orientação a objetos não garante a reutilização, ela oferece mecanismos para que isso ocorra, mas sempre será função do desenvolvedor garantir isso.

Algumas áreas de aplicação da OO

- Sistemas baseados em interfaces gráficas.
- Sistemas de tempo real.
- Software Básico (sistemas operacionais, protocolos de comunicação, etc.)
- Software comercial
- Sistemas de Banco de Dados
- outros

Desenvolvimento Orientado a Objetos

- **Análise Orientada a Objetos:** É o processo de construção de modelos do domínio do problema, identificando e especificando um conjunto de objetos que interagem e comportam-se conforme os requisitos estabelecidos para o sistema.
- **Projeto Orientado a Objetos:** é o processo de geração de uma especificação detalhada do software a ser desenvolvido, de tal forma que esta especificação possa levar a direta implementação no ambiente alvo.

Desenvolvimento Orientado a Objetos

- **Programação Orientada a Objetos:** é um modelo de programação que baseia-se em conceitos como classes, objetos, herança, etc. Seu objetivo é a resolução de problemas baseada na identificação de objetos e o processamento requerido por estes objetos, e então na criação de simulações destes objetos.
- A programação é obtida através da definição de classes e criação de hierarquias, nas quais propriedades comuns são transmitidas das superclasses para as subclasses através do mecanismo de herança.

Desenvolvimento Orientado a Objetos

- Objetos destas classes são instanciados tal que a execução do programa é vista como um conjunto de objetos relacionados que se comunicam enviando mensagens uns para os outros.

Linguagens Orientadas a Objetos

- Na literatura existe uma distinção entre linguagens baseadas em objetos e linguagens orientadas a objetos:
 - Uma linguagem é *baseada em objetos* quando ela fornece apoio somente ao conceito de objetos. Exemplo: Ada e Visual Basic
 - Uma linguagem é *orientada a objetos* quando ela fornece apoio a objetos, e requer que objetos sejam instâncias de classes. Além disso, um mecanismo de herança deve ser oferecido. Ex: C++, Java e Smalltalk.

Linguagens Orientadas a Objetos

- Além da distinção entre linguagens baseadas em objetos e linguagens orientadas a objetos, existe uma outra distinção que classifica as linguagens orientadas a objetos em:
 - *Híbridas*: São linguagens que originalmente não foram projetadas orientadas a objetos, mas que passaram a incorporar os conceitos deste paradigma. Ex: C++ e Object Pascal
 - *Puras*: São linguagens que foram projetadas originalmente orientadas a objetos. Ex: Smalltalk e Java.

Linguagens Orientadas a Objetos

■ Podemos identificar similaridades entre a programação procedural (ou imperativa) e a programação orientada a objetos.

■ P a r a d i g m a	■ Paradigma	de
Procedural	Objetos	

■ Tipos de Dados

■ Variável

■ Função / Procedimento

■ Chamada de Função

■ Classes

■ Objeto / Instância

■ Operação / Método Serviço

■ Envio de Mensagem

Objetos

- Informalmente um objeto representa uma entidade, tanto física quanto conceitual ou de software.

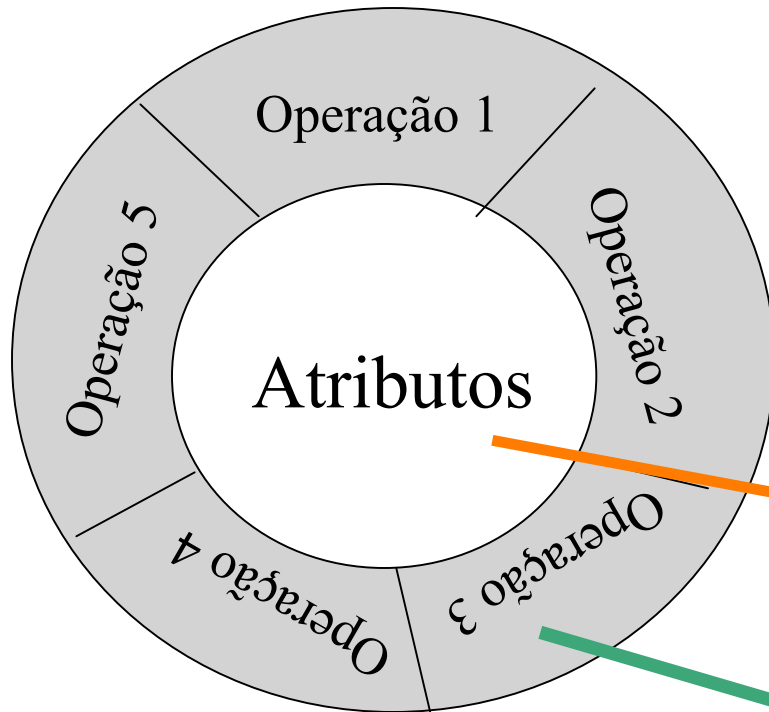
Exemplos:

- Entidade Física: caminhão, carro, bicicleta, etc.
 - Entidade Conceitual: processo químico, matrícula, etc
 - Entidade de Software: lista encadeada, arquivo, etc.
-
- Podemos afirmar que um objeto é um conceito, abstração, ou entidade com limites bem definidos e um significado para a aplicação.

Objetos

- Objetos são implementações de Tipos Abstratos de Dados (TAD's). TAD's, um conceito da área de Linguagens de Programação, são entidades que encapsulam dados e operações associadas que manipulam esses dados.
- Evolução de Tipos de Dados
 - Assembler: não possui tipos,
 - Fortran: primeiro tipos primitivos,
 - Pascal: tipos agregados de dados
 - Simula: associação de tipos e operações.

Objetos



Exemplo

Objeto geométrico

cor:
posição:

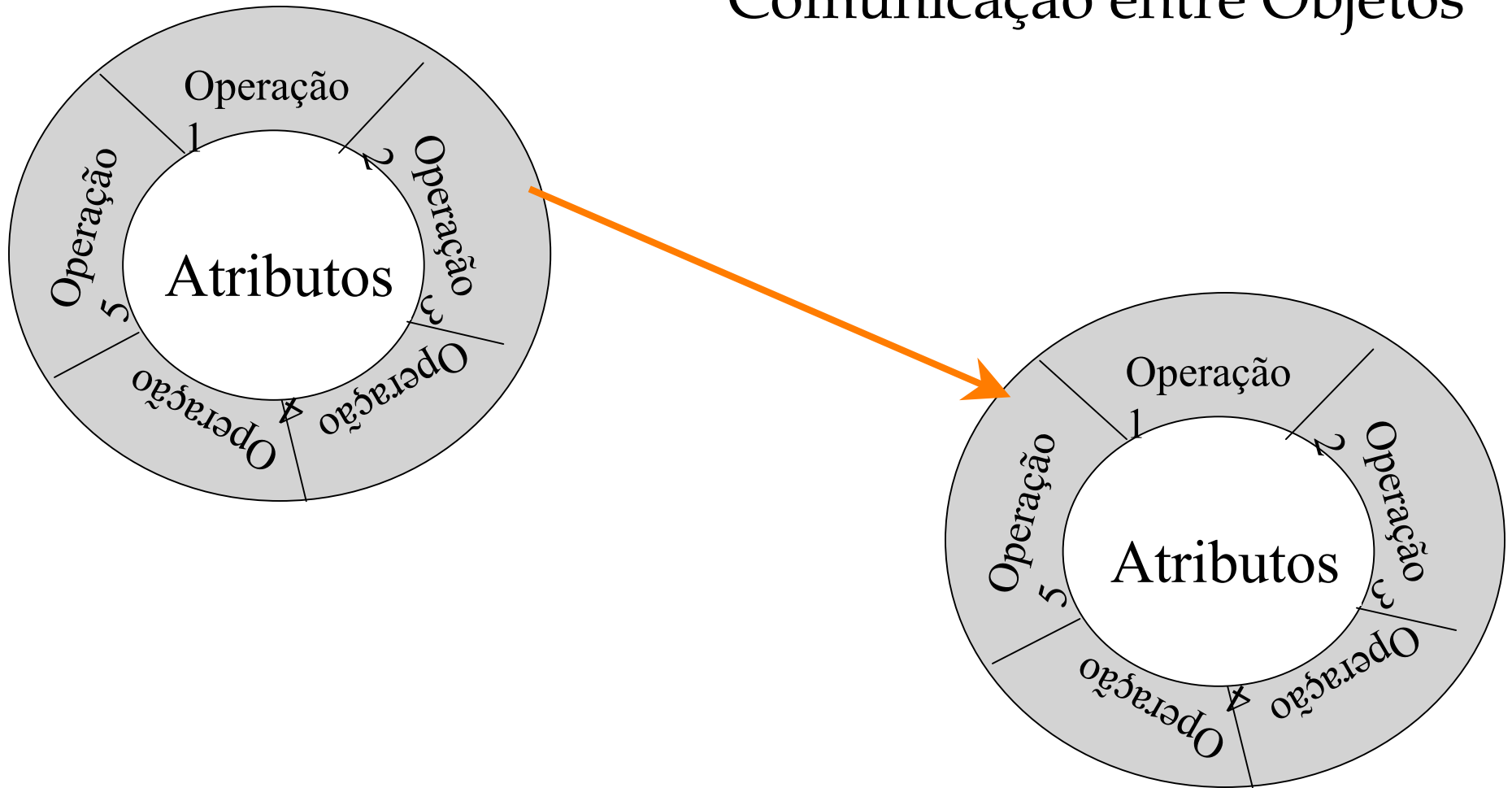
selecionar(p: Ponto): boolean
girar(Ângulo: real)
mover(delta: coord)

Objetos - Características

- Os dados de um objeto são totalmente escondidos e protegidos de outros objetos. A única maneira de acessá-los é através da invocação de uma operação declarada na interface pública do objeto. A interface pública de um objeto consiste no conjunto de operações que um cliente do objeto pode acessar.
- Variáveis representando o estado interno do objeto são chamadas variáveis de instância. As operações são chamadas de métodos.
- Um objeto comunica-se com outro através de mensagens que identificam operações a serem realizadas no segundo objeto.

Objetos - Características

Comunicação entre Objetos



Encapsulamento

- Esconder os detalhes da implementação de um objeto é chamado Encapsulamento.
- A capacidade de um objeto possuir uma parte privada, acessível somente através dos métodos definidos na sua classe.
- Benefícios:
 - O código cliente pode usar apenas a interface para a operação.
 - A implementação do objeto pode mudar, para corrigir erros, aumentar performance, etc sem que seja necessário modificar o código do cliente.
 - A manutenção é mais fácil e menos custosa.
 - Cria um programa legível e bem estruturado

Exemplo de Encapsulamento

- Implementação de uma Lista sem usar Encapsulamento:

```
public class Lista{  
    public int dados[];  
    public int tamanho;  
    public Lista(int n) {  
        dados = new int[n];  
        tamanho = 0;  
    }  
}
```

```
public static void  
    main(String args[]) {  
    Lista al = new Lista(5);  
    al.dados[0]=0;  
    al.tamanho++;  
    al.dados[1]=1;  
    al.tamanho++;  
    if (al.tamanho>=2) {  
  
        System.out.println("Imprime c  
1 elemento "+ al.dados[0]);  
  
        System.out.println("Imprime c  
2 elemento "+ al.dados[1]);  
  
    }  
}
```

Exemplo de Encapsulamento

- Implementação de uma Lista com Encapsulamento:

```
public class ArrayLista{
    private int dados[];
    private int tamanho;
    public ArrayLista(int n) {
        dados = new int[n];
        tamanho = 0;
    }
    public void add(int a) {
        dados[tamanho] = a;
        tamanho++;
    }
    public int remove(int posicao) {
        if ((posicao>0) &&
            (posicao<tamanho))
            return dados[posicao];
        return -1;
    }
}
```

```
public static void main(String args[])
{
    ArrayLista al=new ArrayLista(5);
    al.add(1);
    al.add(2);
    System.out.println("Imprime o 1
    elemento " +al.remove(0));
    System.out.println("Imprime o 2
    elemento " +al.remove(1));
}
}
```

Exemplo de Encapsulamento

■ Implementação de uma Lista com Encapsulamento:

```
import java.util.Vector;
public class VectorLista{
    private int dados[];
    public VectorLista(int n) {
        dados = new Vector(n);
    }
    public void add(int a) {
        dados.addElement(new Integer(a));
    }
    public int remove(int posicao) {
        if ((posicao>0) && (posicao<dados.size()))
            return ((Integer)
                dados.elementAt(posicao)).intValue();
        return -1;
    }
}
```

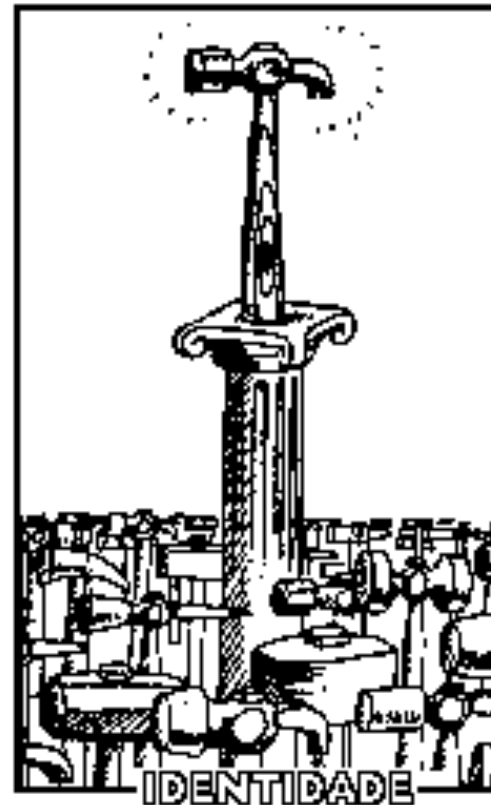
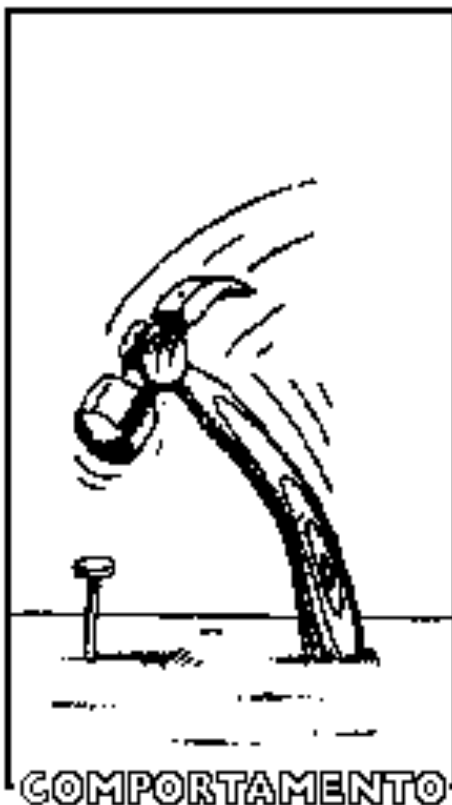
```
public static void main(String args[])
{
    VectorLista al = new VectorLista(5);
    al.add(1);
    al.add(2);
    System.out.println("Imprime o 1
    elemento "+ al.remove(0));
    System.out.println("Imprime o 2
    elemento "+ al.remove(1));
}
}
```

Objetos

- Formalmente, um objeto é algo que possui:
 - um estado, que é normalmente implementado através de seu conjunto de propriedades (denominadas atributos), com os valores das propriedades, mais as ligações que o objeto pode ter com outros objetos;
 - uma identidade única. Identidade é a propriedade de um objeto que distingue-o de outros objetos. Identidade não é o nome do objeto, nem o endereço de memória onde ele está armazenado, é um conceito de linguagens de programação que não é visível para os “usuários”. e
 - um comportamento. O comportamento define como um objeto reage às requisições de outros objetos, em termos de mudanças de estados e passagem de mensagens.

Objetos

- Um objeto possui um estado, exibe um comportamento bem-definido e possui uma identidade única.

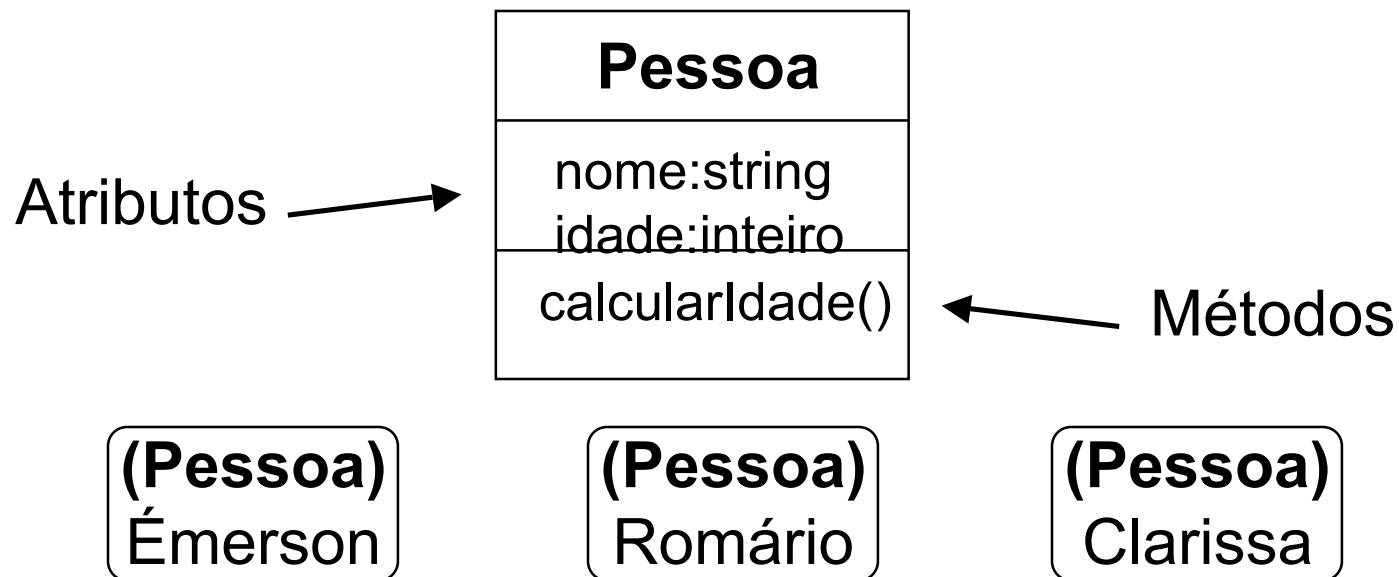


Classe

- É a descrição de um grupo de objetos com propriedades similares (atributos), comportamento comum (operações), relacionamentos com outros objetos e semânticas idênticas.
 - Todo objeto é instância de uma classe.
 - Exemplo: int a;
 - Pessoa p;
- Enquanto um objeto individual é uma entidade concreta que executa algum papel no sistema, uma classe captura a estrutura e comportamento comum a todos os objetos que estão relacionados.

Classes

- Uma classe define a estrutura e o comportamento de qualquer objeto da classe, atuando como um padrão para a construção de objetos.
- Objetos podem ser agrupados em classes.

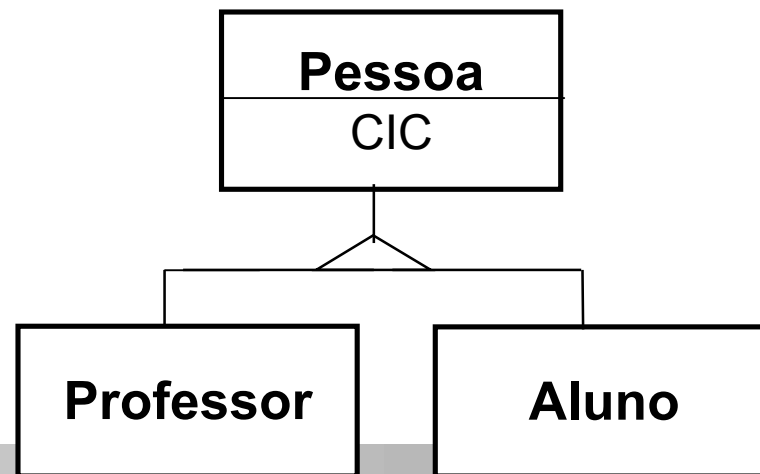


Classes

- A definição da classe consiste na definição dos atributos e operações dos objetos desta classe.
- Um atributo é uma característica de uma classe. Atributos não apresentam comportamento, eles definem a estrutura da classe.
- Operações caracterizam o comportamento de um objeto, e são o único meio de acessar, manipular e modificar os atributos de um objeto.

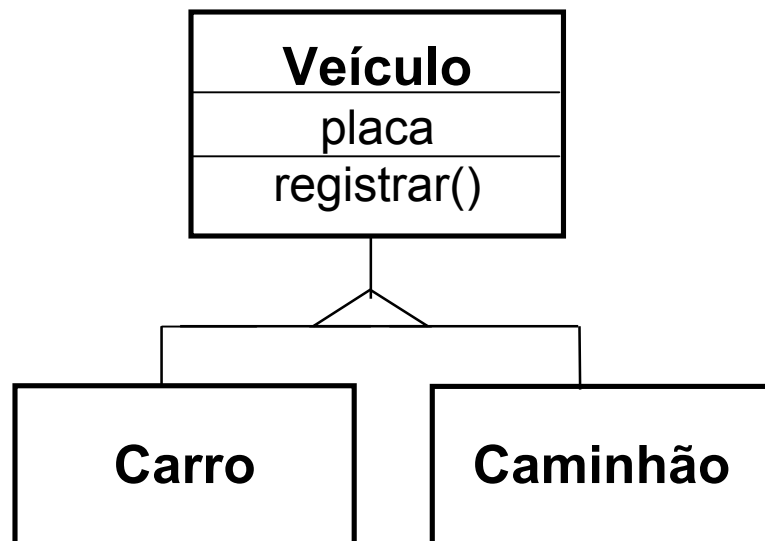
Herança

- É um mecanismo existente no paradigma orientado a objetos que permite a reutilização da estrutura e do comportamento de uma classe ao se definir novas classes.
- A herança também é conhecida como relac. “é-um”.
- A classe que herda o comportamento é chamada de subclasse e a que definiu o comportamento superclasse.



Herança

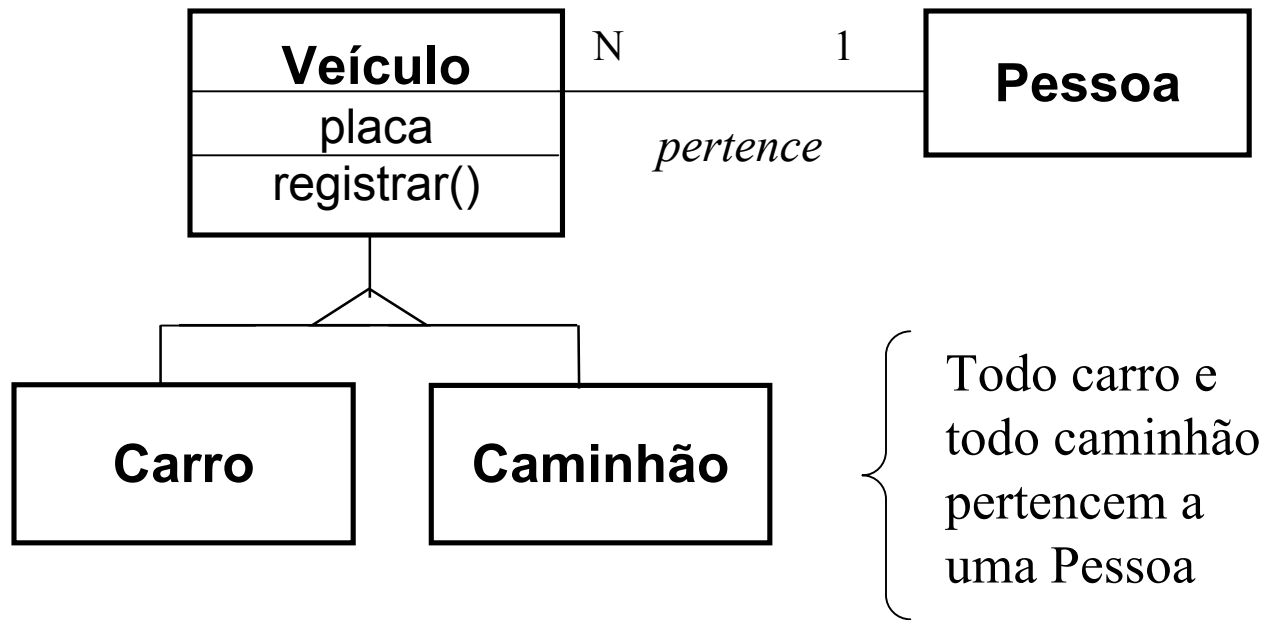
■ Herança de Atributos



Todo carro
e todo
caminhão
possuem
uma placa
e uma
operação
`registrar()`

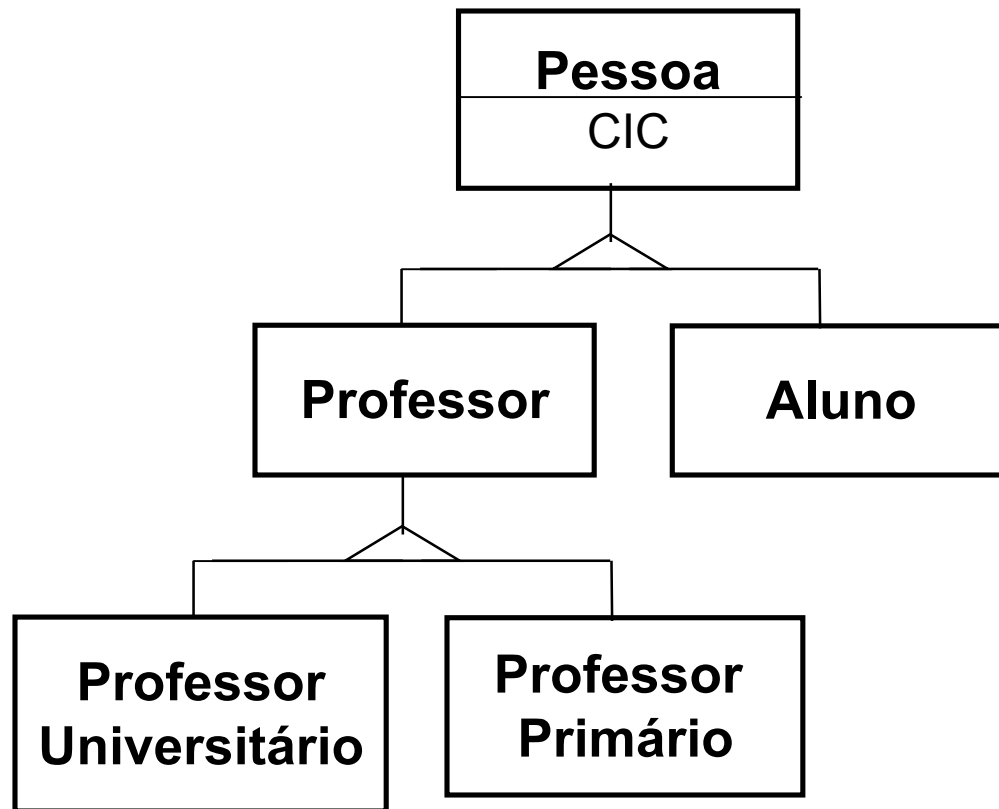
Herança

■ Herança de Relacionamentos



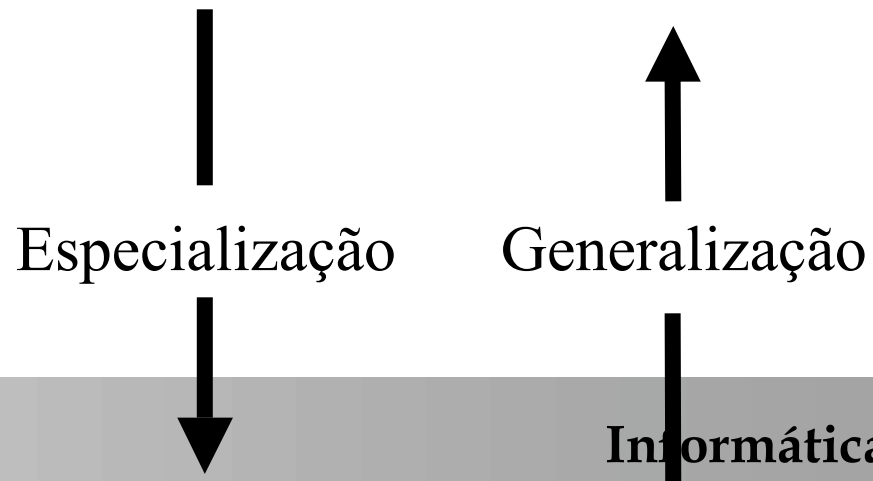
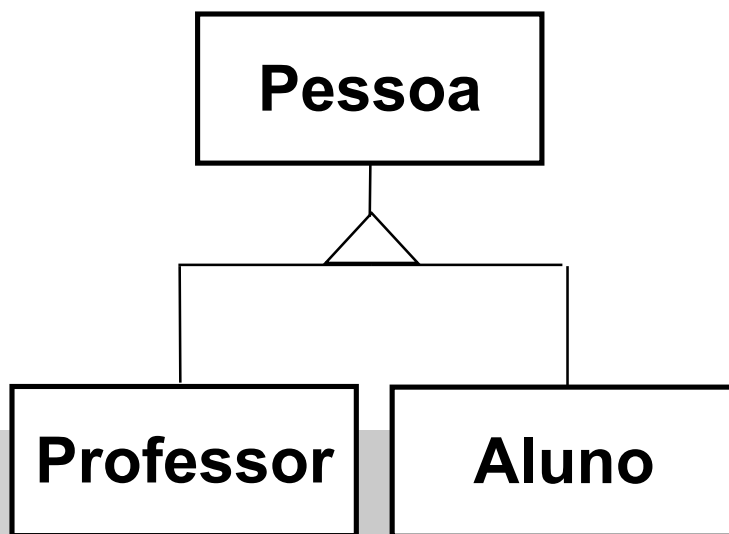
Herança

- Não há limites no número de níveis na hierarquia de herança.



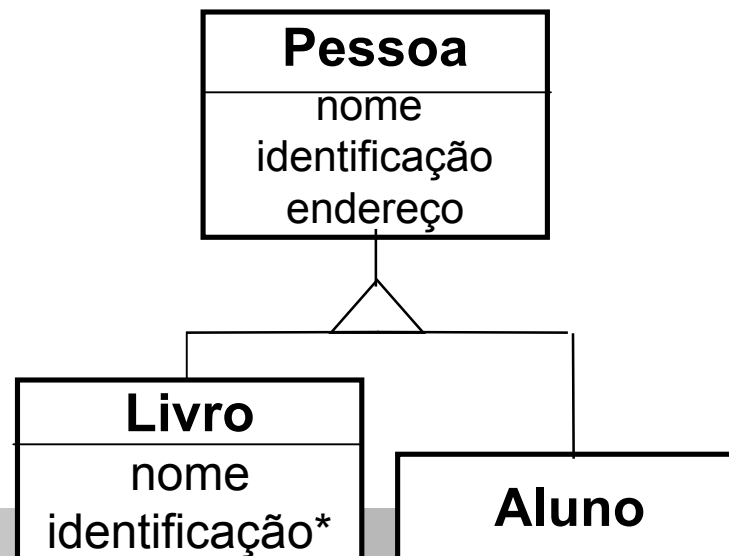
Herança

- A utilização da herança aumenta a reutilização de código porque o código definido na superclasse pode ser utilizado automaticamente na subclasse.
- Através da herança é possível representar a relação de generalização/especialização entre duas classes:
 - a superclasse é uma generalização da(s) subclasse(s), e
 - a subclasse é uma especialização da(s) superclasse(s).



Tipos de Herança

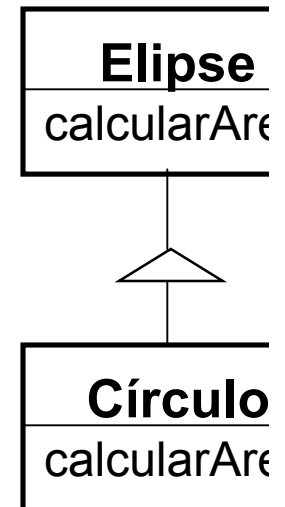
- Herança estrita: as subclasses podem redefinir ou excluir propriedades herdadas da superclasse.
- Herança não estrita: as mudanças acima são permitidas.
 - Observação: se uma mudança de um tipo e/ou nome é feita, pode caracterizar a subclasse de maneira diferente da superclasse.



Tipos de Herança

■ A subclasse pode :

- adicionar novas operações:
- Exemplo: na classe Professor a operação `ministrarAula`.
- redefinir uma operação existente.
- Exemplo: Um círculo é um tipo especial de elipse cujo método para calcular área é πr^2 .
- remover um comportamento. (pouco frequente)
- uma combinação das três anteriores.



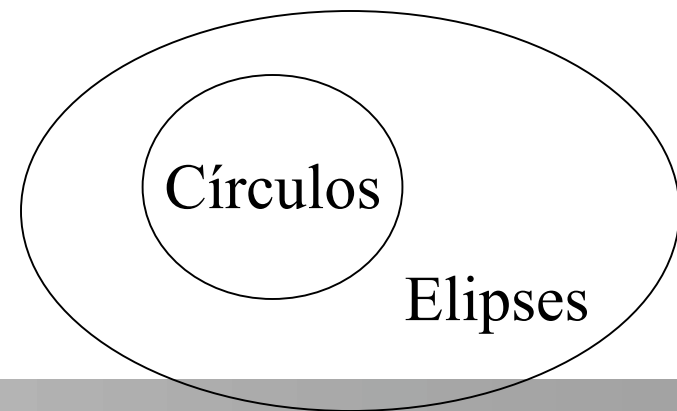
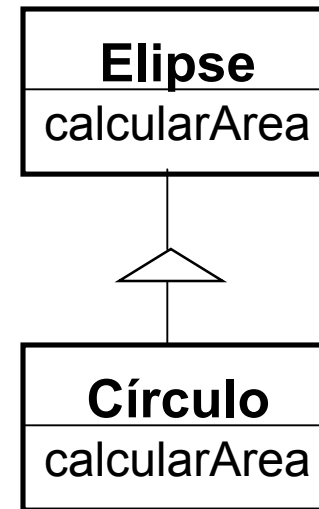
Tipos de Herança

■ Observações:

- Quando uma operação ou atributo é redefinida em uma subclasse ele é chamado de sobrecarregado (overloaded).
- Algumas linguagens de programação podem implementar a exclusão de propriedades, no entanto, isto ocasiona um problema. Não se sabe até onde a propriedade está sendo transmitida em uma hierarquia de generalização.
- Na realidade em todas as linguagens onde existe alguma facilidade de cancelamento, a única forma de verificar se alguma propriedade de uma classe ainda é aceita em uma subclasse é conferir cada uma das subclasses[TAK90].

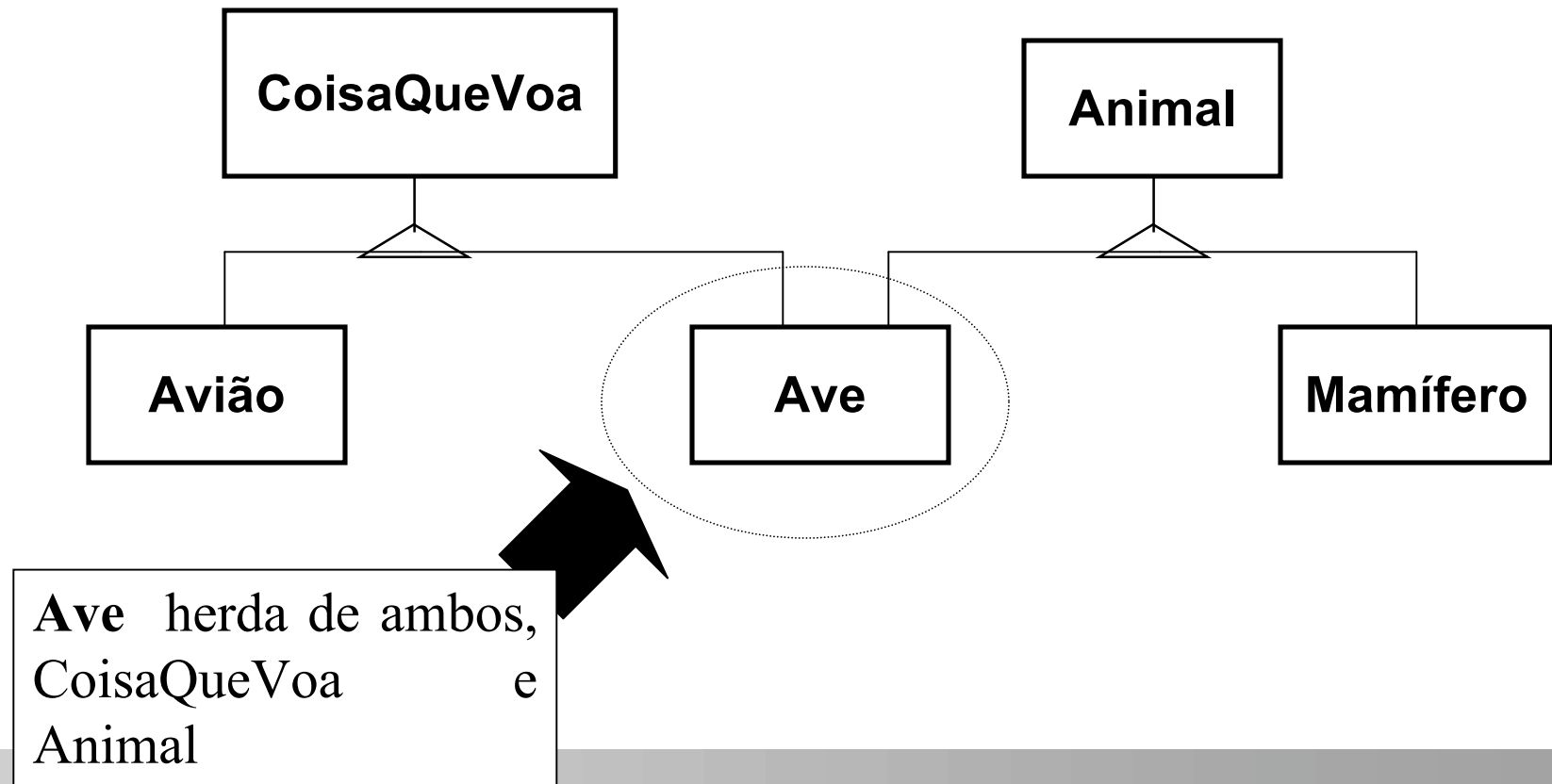
Tipos de Herança

- A herança deve ser utilizada visando a reutilização de comportamento. Ou seja, as classes derivadas devem se comportar como as superclasses. Para isso sempre deve existir uma hierarquia de Generalização /Especialização entre as classes.
- Um objeto do tipo círculo pode ser usado no lugar de um objeto do tipo elipse porque todo círculo é um tipo especial de elipse.



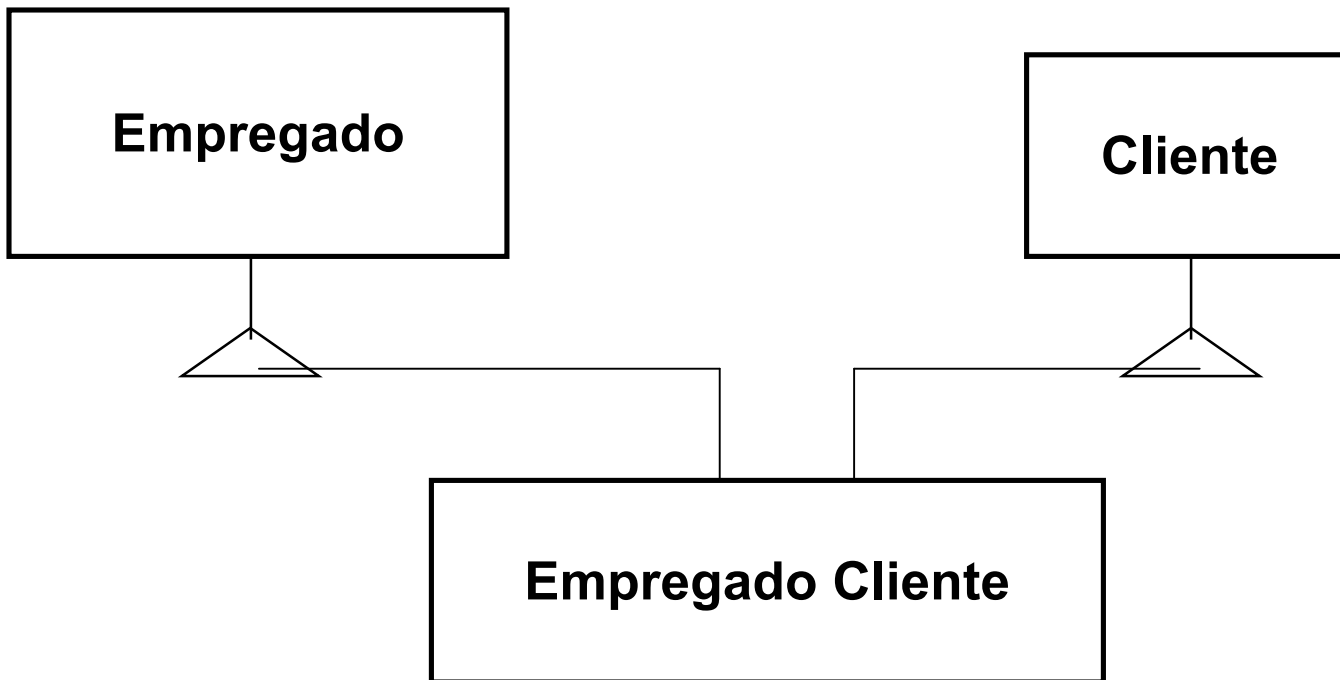
Herança Múltipla

- Herança múltipla é a possibilidade de se definir uma subclasse com mais de uma superclasse.



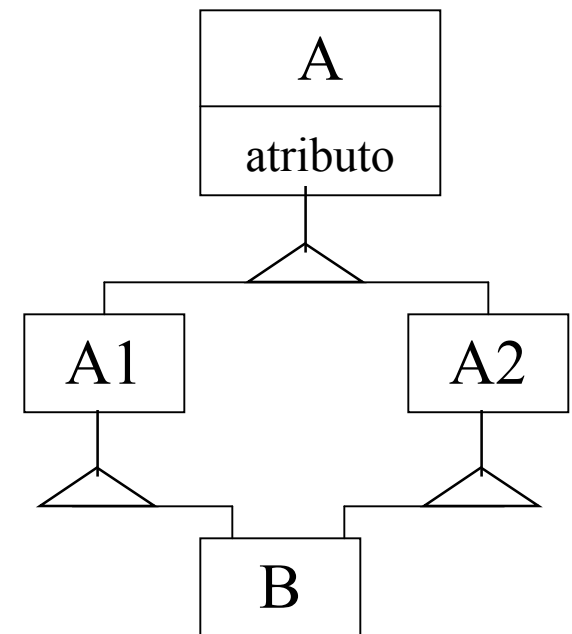
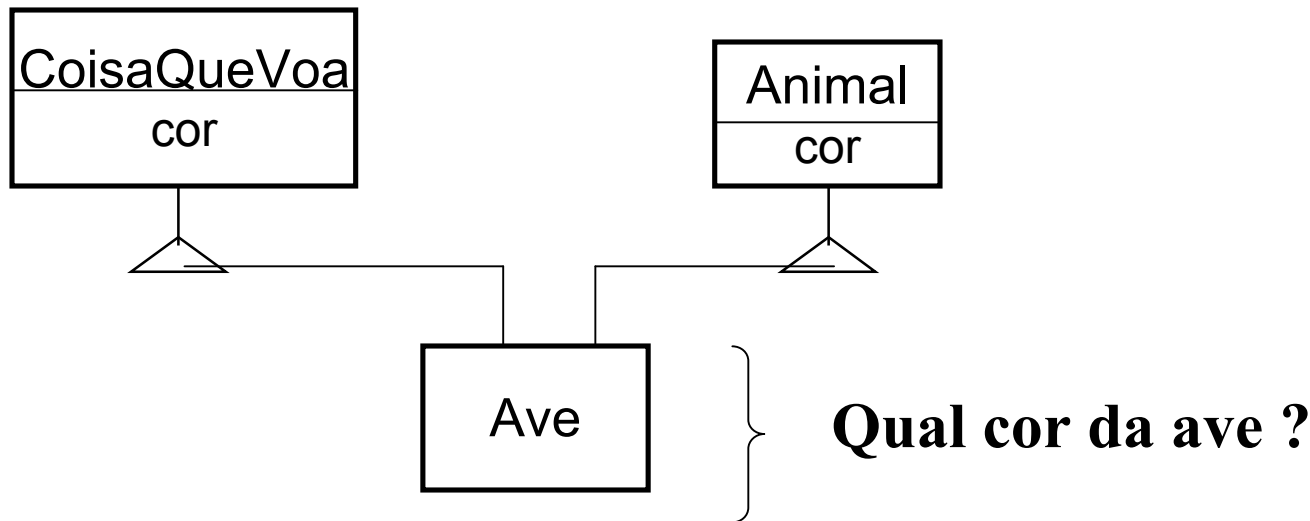
Herança Múltipla

■ Outro Exemplo de Herança Múltipla



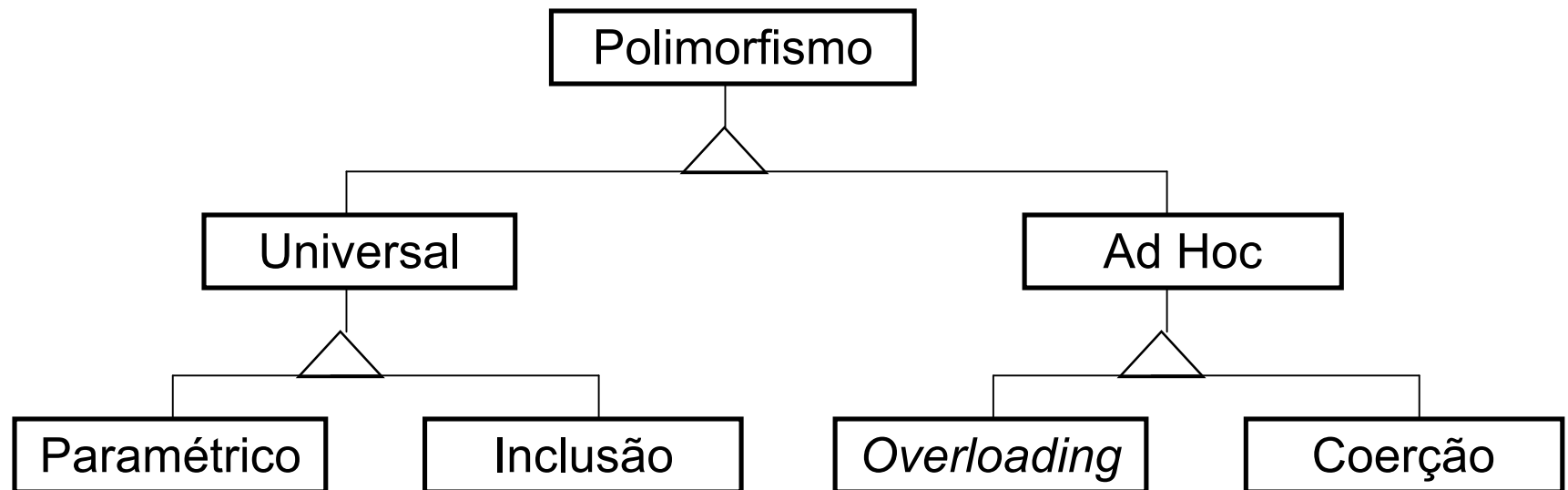
Herança Múltipla: Observações

- Conceitualmente, a herança múltipla é necessária para modelar o mundo real de maneira mais precisa.
- Na prática, ela pode levar a problemas na implementação pois nem todas as linguagens de programação orientadas a objetos suportam herança múltipla.



Polimorfismo

- É a habilidade de variáveis terem “mais de um tipo”. Funções são ditas polimórficas quando seus operandos podem ter mais de um tipo.
- Classificação de [Cardelli & Wegner]:

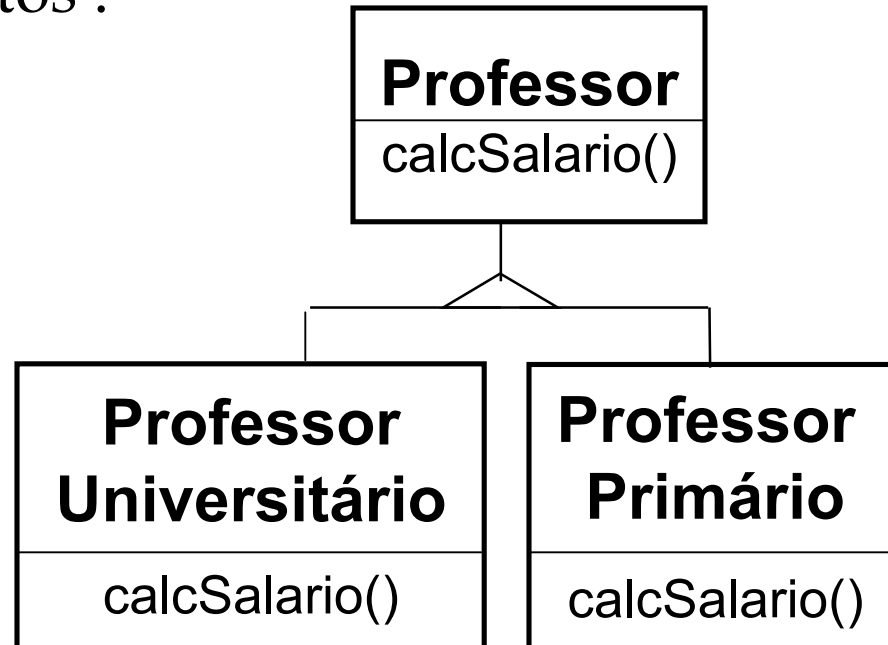


Polimorfismo

- **Coerção:** a linguagem de programação tem um mapeamento interno entre tipos.
 - Exemplo: se o operador `+` é definido para somar dois números reais e um inteiro é passado como parâmetro então o inteiro é “coargido” para real.
- **Overloading (sobrecarga):** permite que um “nome de função” seja usado mais de uma vez com diferentes tipos de parâmetros. O compilador automaticamente chama a função “correta” que deve ser utilizada.
 - Exemplo: o operador `+` que pode ter 2 parâmetros inteiros, 2 parâmetros reais, 2 cadeias de caracteres (concatenação), etc. A instrução `read(x)` em Pascal onde `x` pode ser inteiro, real ou string.

Polimorfismo

- Em Orientação a Objetos :



- Assim, pode-se adicionar um comportamento específico (implementação) às subclasses de uma hierarquia de generalização / especialização.

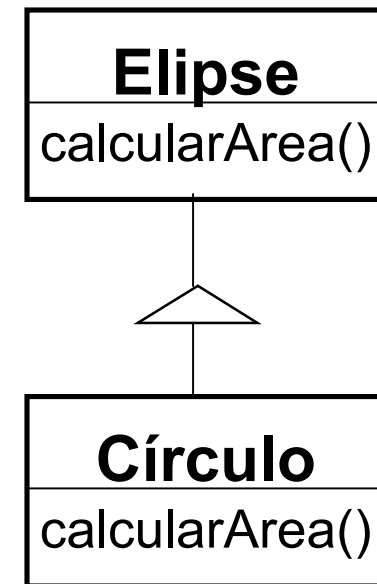
Polimorfismo

- Polimorfismo paramétrico ou parametrização: a partir de uma única definição de uma função ela pode trabalhar uniformemente. Também conhecido como *genericity*.
- Exemplo:
 - `/* Definição da Classe */`
 - `template class pilha <Tipo T> {`
 - `/* Estrutura de dados ... */`
 - `void empilhar(T);`
 - `T desempilhar();`
 - `}`
 - `/* Utilização no código Cliente */`
 - `pilha<int> pilhaInt;`
 - `pilha<float> pilhaFloat;`

Polimorfismo

- Polimorfismo de Inclusão: tipo de polimorfismo encontrado em linguagens orientadas a objetos. Todo objeto de uma subclasse pode ser usado no contexto de um superclasse.
- Exemplo: todo objeto do tipo círculo pode ser usado no lugar de um objeto do tipo elipse.

```
/* Código polimórfico */  
void main( ) {  
    Elipse e;  
    Circulo c;  
    imprimir(e);  
    imprimir(c);  
}
```



```
47 void imprimir(Elipse eli) {..... }
```

Acoplamento (*Binding*)

- É uma associação feita pelo compilador (interpretador) entre um atributo e uma entidade.
 - Exemplo: acoplamento entre tipos e variáveis.
- O acoplamento pode ser:
 - Estático: se ocorre antes do tempo de execução e permanece inalterado durante a execução do programa.
 - Exemplo: quando você declara em Pascal que uma variável é do tipo integer;
 - Dinâmico ou Atrasado: se ocorre durante o tempo de execução ou muda durante a execução do programa.

Acoplamento

■ Exemplo (Java):

- 1 `Elipse e;`
- 2 `Circulo c;`
- 3 `e := c;`
- 4 `e.calcularArea();`

■ A atribuição da linha 3 é dinamicamente acoplada, pois acopla o objeto *e* à um tipo diferente de seu tipo originalmente declarado.

■ Em princípio, seria uma violação atribuir um objetos de um tipo diferentes tipo a variável *e*, no entanto, como *c* (Circulo) é subclasse de *e* (Elipse) essa atribuição é válida. Qual o método executado ? Círculo ou

Acoplamento

- A operação executada é a de `Circulo`, porque o compilador em tempo de execução verifica que a variável aponta para um objeto desta classe.
- Em algumas linguagens o programador deve pedir explicitamente o acoplamento dinâmico para uma mensagem em particular. Por exemplo, em C++ a operação deve ser declarada *virtual* na superclasse e redefinida na subclasse.
- Importante: Todas as operações sobrecarregadas na classe derivada devem proporcionar **semanticamente** os mesmos serviços oferecidos pela superclasse.

Classes Abstratas

- Uma classe abstrata é uma classe que não tem instâncias diretas, mas cujas subclasses podem ter instâncias. Uma classe concreta é uma classe que pode ter instâncias. Em outras palavras se X é uma classe abstrata vc não pode executar o código a seguir: `X objeto = new X();`
- Apesar disso, você pode criar construtores de uma classe abstrata para que eles sejam chamados pelos construtores das subclasses. (Reutilização)
- Em Java utiliza-se a palavra *abstract* para indicar uma classe abstrata:

Classes Abstratas

- O objetivo de criarmos classes abstratas é para encapsular outras classes com comportamento comum. Elas podem surgir naturalmente na modelagem ou serem criadas para promover o reuso.
- Além disso, uma classe abstrata pode definir um protocolo para uma operação sem definir a implementação do método.
 - `public abstract class Figure { // inicioFigure`
 - `public abstract double area();`
 - `public abstract double perimetro();`
 - `} // fimFigure`

Classes Abstratas

■ Assim, você pode declarar métodos abstratos em uma classe abstrata apenas para especificar um protocolo comum de operações. Toda subclasse concreta da classe abstrata deve fornecer uma implementação para TODOS os métodos abstratos:

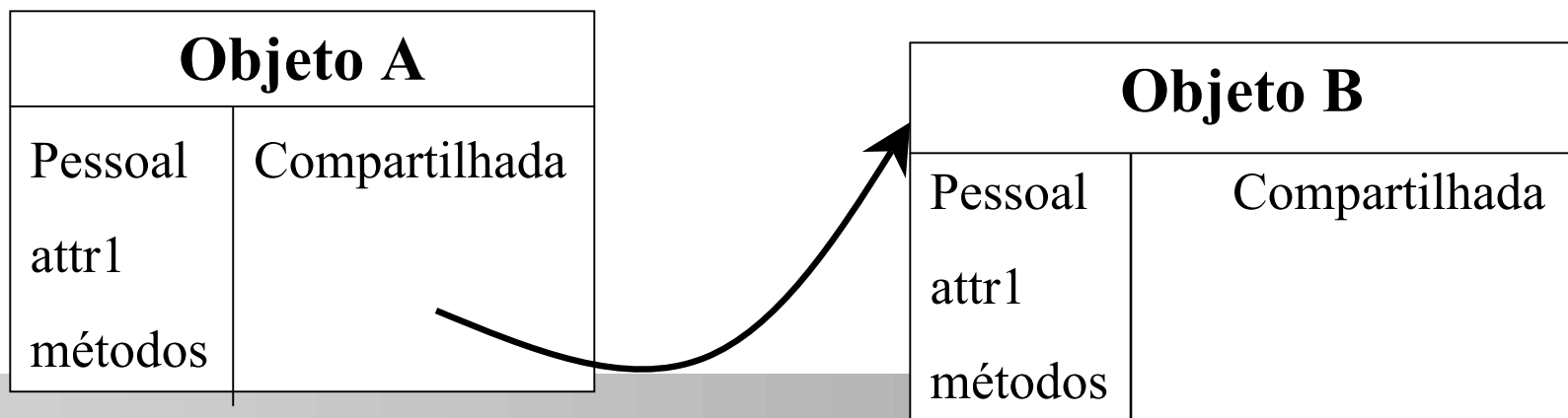
- `class Circle extends Figure {`
- `protected double raio;`
- `public Circle(double r) { raio = r;}`
- `public double area() { return PI*raio*raio;}`
- `public double perimetro() { return 2*PI*raio;}`
- `}`

Classes Abstratas

- Qualquer classe com pelo menos um método *abstract* é automaticamente abstrata e deve ser declarada como tal.
- Se uma subclasse de uma classe abstrata não implementa todos os métodos abstratos então ela também é abstrata.
- Uma classe abstrata também pode ter métodos concretos. Frequentemente, faz sentido mover o máximo de funcionalidade possível para uma superclasse, seja ela abstrata ou não.
- Uma classe pode ser abstrata sem ter métodos abstratos. Isto evita que a classe seja instanciada.

Delegação

- É um outro mecanismo para compartilhamento de código. Também conhecido como composição.
- Normalmente ele é utilizado em linguagens que não utilizam o conceitos de classes. Nestas linguagens os objetos são chamados protótipos (prototypes) e cada um deles implementa um comportamento específico.



Delegação

- Se o objeto A não implementa uma determinada mensagem ele delega (repassa) a mensagem para o objeto B. Se o objeto B implementa aquela mensagem então ele a executa com os dados de A, senão ele a delega para seus “delegatee’s”
- Pode existir uma lista de objetos para os quais mensagens são delegadas, o que causa conflito assim como na herança múltipla. A solução é uma lista sequencial de objetos.
- Exemplos de linguagens: Actor e Self.

Delegação

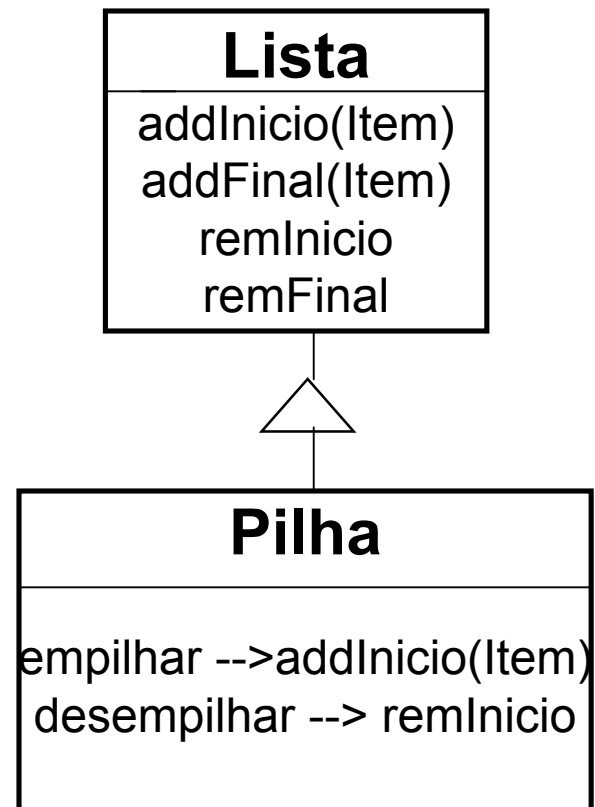
- A herança permite o compartilhamento baseado em classes, enquanto que a delegação permite o compartilhamento baseado em objetos.
- A delegação é um modo mais geral que a herança de estender a funcionalidade de uma classe, visto que existem muitas ocasiões em que não é conveniente usar a herança. Além disso, a delegação permite que um objeto possa alterar suas respostas para pedidos de serviços em tempo de execução.

Delegação

- Segundo [Grand98], qualquer extensão de uma classe que pode ser feita utilizando-se a herança, também pode ser feita com a delegação. Entretanto, a recíproca não é verdadeira.
- Em linguagens baseadas em classes este mecanismo também pode ser obtido. Isto é feito através do repasse da mensagem para o outro objeto. Para isso é necessário que o objeto delegador do serviço contenha uma referência para o objeto responsável pela execução.

Delegação: quando usar

- 1. Se uma subclasse precisar esconder um método ou variável herdado da subclasse. Na maioria das linguagens orientadas a objetos não existe nenhum mecanismo para permitir isto, logo deve-se evitar a herança.



Delegação: quando usar

```
public class Lista {
    private Vector _dados = new Vector();
    .....
    public void addInicio(int x) {...}
    public void addFinal(int x) {...}
    public int remFinal(      ) {...}
    public int remInicio(     ) {...}
}

public class Pilha {
    private Lista _lista;
    // A implementacao das operacoes especificas da pilha
    // são simplesmente DELEGADAS para a lista.
    public void push(int x) { _list.addInicio(x); }
    public int pop(      ) { return _list.remInicio( ); }
}
```

Delegação: quando usar

- 2. Quando você tem um código a ser reutilizado entre duas ou mais classes e não existe herança entre elas:

```
public class ClienteJuridico extends Cliente {
    private Endereco _endereco;
    public void setEndereco(numero, rua, cidade, estado, cep) {
        _endereco.setData(numero, rua, cidade, estado, cep) ;}
}

public class Fornecedor {
    private Endereco _end;
    public void setEndereco(numero, rua, cidade, estado, cep) {
        _end.setData(numero, rua, cidade, estado, cep) ; }
}
```

Delegação: quando usar

```
public class Endereco {  
    private numero, rua, cidade, estado, cep;  
}
```

Delegação: quando usar

3. Para acomodar modificações dos objetos que mudam de Subclasses.

```
public abstract class Aluno {  
    // Atributos comuns de um Aluno}  
public class Bolsista extends Aluno {}  
public class Monitor extends Aluno {}  
public class Estagiario extends Aluno {}
```

- A herança implica em uma acomodação fraca dos objetos que mudam de classe ao longo do tempo: Um aluno está frequentemente mudando de subclasses, ou seja, hoje ele é um Bolsista, depois de um mês ele é um Estagiário, e assim por diante.

Delegação: quando usar

- Para modelarmos isso, seria necessário que o objeto Bolsista existente fosse destruído do sistema e um novo objeto Estagiário fosse criado, entretanto deveríamos ter antes copiado os atributos comuns de um objeto para outro.
- Além disso, quando um objeto se transmuta, ele perde todo o seu histórico. Isso torna a mudança mais complexa.
- A solução é usar a delegação, visto que ela se ajusta melhor à mudanças frequentes. Quando um objeto precisa de responsabilidades adicionais específicas do papel, basta adicionar um novo papel.

Delegação: quando usar

- Desta forma, estaremos usando a composição (uma pessoa e seus papéis) e a herança (papéis especializados desempenhados por um aluno).

```
public class Aluno {  
    // Atributos comuns de um Aluno  
    //Um novo atributo é adicionado que corresponde  
    // a um apontador para uma lista de papéis  
    // desempenhados pelo aluno  
    Vector _papeis;  
}
```

Delegação: quando usar

```
public abstract class PapelAluno {  
    // Não possui nada, apenas a definição da  
    classe...  
}
```

```
public class Bolsista extends PapelAluno {}  
public class Monitor extends PapelAluno {}  
public class Estagiario extends PapelAluno {}
```

MetaClasses

- São classes cujas instâncias também são classes.
- Uma classe contém informação sobre os objetos, enquanto que uma metaclasses contém as informações referentes à classe.
- Dependendo da linguagem orientada a objetos :
 - Existe suporte explícito a metaclasses. Ex: ObjVLisp.
 - Existe suporte implícito a metaclasses. Ex: Smalltalk.
 - Não existe suporte a metaclasses. Ex: C++ e Java.

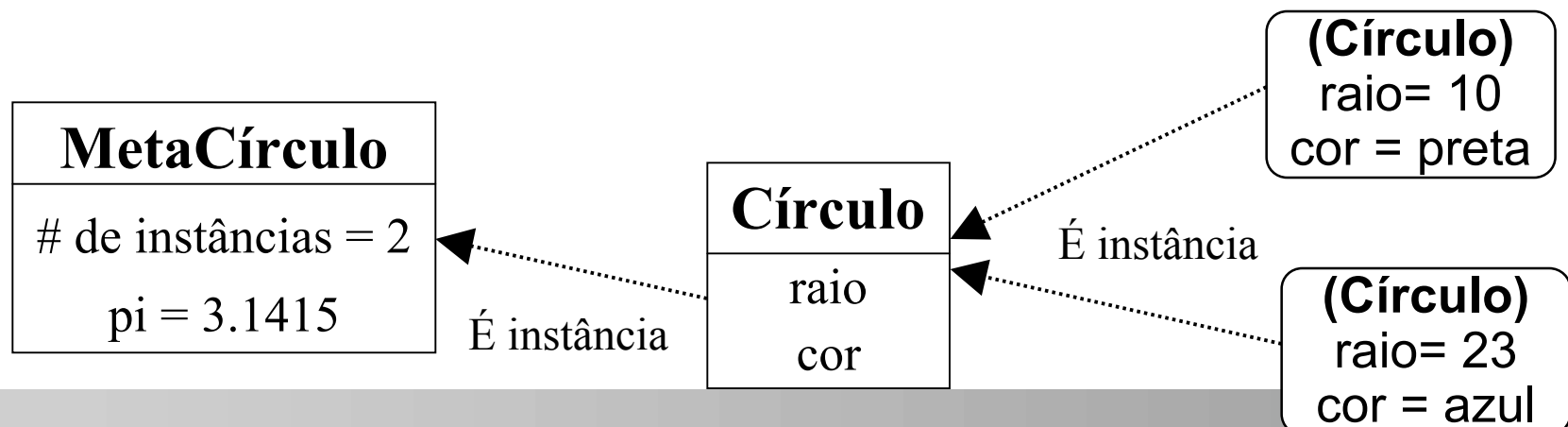


MetaClasses

■ Vantagens

- 1. As classes podem ser utilizadas para armazenar a informação em relação ao grupo de objetos.

► Exemplo: o número de objetos instanciados a partir da classe, o valor médio de uma propriedade calculada a partir dos valores específicos desta propriedade nas instâncias, uma propriedade comum a todas as classes.



MetaClasses

- 2. São utilizadas no processo de criação (e inicialização) das novas instâncias de uma classe.
 - Exemplo:
 - - Uma classe passa a ser tratada como um objeto que recebe uma mensagem solicitando a criação de objetos. Assim, o código abaixo corresponderia a :
 - `Circulo c = new Circulo();`
 - enviarmos uma mensagem (new) a um objeto `Círculo`. Logo, assim com a classe específica as operações da interface pública dos objetos, necessita-se de uma metaclasses para especificar as operações da interface pública das classes.

MetaClasses

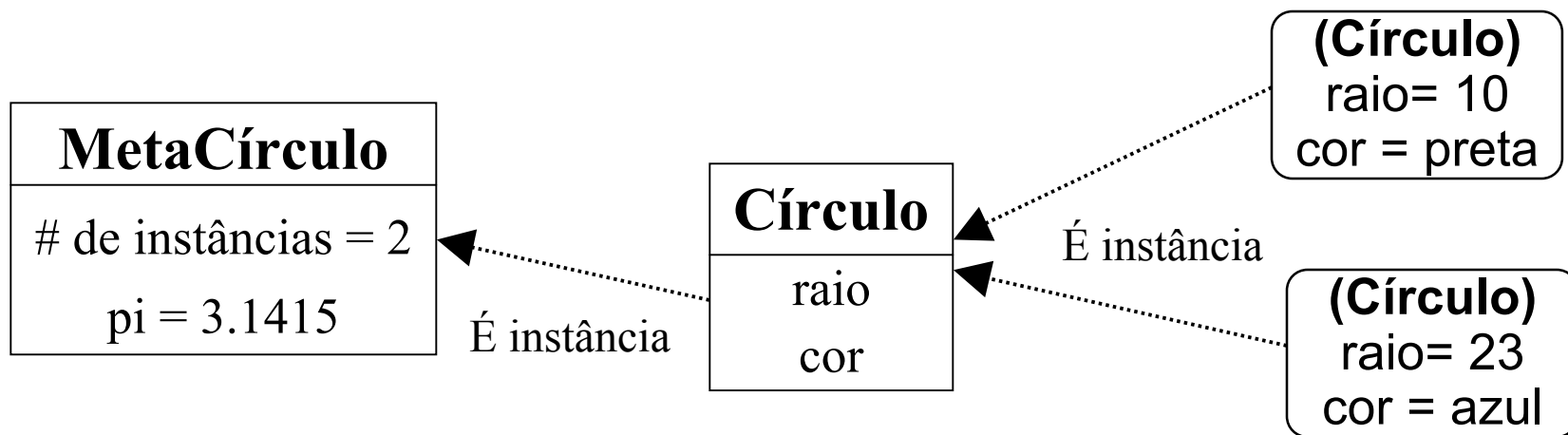
- É importante frisar que o conceito de metaclasses na implementação tende a desaparecer. O ponto a destacar é que o nível da metaclasses é, frequentemente irrelevante para o programador típico [TAK90].
- O conceito de metaclasses é utilizado em algumas metodologias de Análise e Projeto Orientado a Objetos.

MetaClasses em Java

- Em Java o conceito de constructor e atributos e métodos static, corresponde a união do conceito de classes e metaclasses.
- Atributos Static: este modificador em atributos torna-os atributos da classe.
- Métodos Static: da mesma forma, métodos static pertencem a uma classe e não operam sobre instâncias de uma classe.
- A sintaxe para acessar metodo *static* é:
 - NomeDaClasse . MetodoStatic(Parametros)
 - Por exemplo: `double valor = Circulo.getPi();`

MetaClasses em Java

```
public class Circulo {  
    static final double Pi = 3.14;  
    int raio; Color cor;  
    public Circulo(int raio) {this.raio = raio; }  
    public static double getPi() { return Pi;}  
    public double getArea() { return Pi*raio*raio;}  
}
```



MetaClasses em Java

- Um método static pode ser chamado sem que seja necessário criar um objeto da classe.
- Exemplo:

```
public class TestRan {
    static public void main(String[] args) {
        double d, soma = 0;
        int i=0;
        while (i<100) {
            d = Math.random(); soma += d; i++;
        }
        System.out.println(" A media e "+(soma/100));}
}
```

MetaClasses em Java

- Um método *static* só pode acessar atributos *static*.
- Como o método *main* é *static* ele não pode acessar atributos não *static* da classe. Por isso, no método *main* frequentemente você deverá criar objetos da própria classe para que assim você possa referenciar atributos desta classe.

```
class Application{  
    public static void main(String[] args) {  
        Application a = new Application();  
    }  
}
```

Bibliografia

- Blair, G. et al. (Editors) *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, 1991.
- Buzato, L. E., Rubira, C. M. F. *Construção de Sistemas Orientados a Objetos Confiáveis*, Décima Primeira Escola de Computação, Rio de Janeiro, Julho de 1998.
- Cornell, G., Horstmann, C. S. *Core Java*. Makron Books, São Paulo, 1998.
- Oliva, Alexandre. *Programação em Java*. II Simpósio Brasileiro de Linguagens de Programação, Campinas, setembro de 1997.
- Cesta, A. A., Rubira, C. M. F. *Tutorial: A Linguagem de Programação Java*, Campinas, junho de 1997.
- Flanagan, D. *Java in a Nutshell, Second Edition*, O'Reilly Associates, 1997.

Bibliografia

- Grand, M. *Patterns in Java*, John Wiley & Sons, 1998.
- Meyer, B., *Object-oriented software construction*, Prentice-Hall, 1988.
- Flanagan, D. *Java Examples in a Nutshell*, First Edition, O'Reilly Associates, 1997.
- Takahashi, T. *Programação Orientada a Objetos*, Escola de Computação, São Paulo, 1990.
- Rubira, C. M. F. *Tópicos Especiais em Engenharia de Software II*, Universidade Estadual de Campinas, notas de aula, 1996.