Programação Orientada à Objetos (POO)

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Especialização, Herança e Polimorfismo**
  * Especialização
  * Herança
  * Sobreposição de métodos
  * Polimorfismo
  * Coleções heterogêneas
  * Sobrecarga de métodos e construtores

Especialização, Herança e Polimorfismo

* **Especialização**
  * Generalização e especialização referem-se a estratégias de agrupar indivíduos com atributos comuns, manipulando de diferentes formas as diferenças que existem entre eles.

  * Exemplos:
    * Lagosta e Sardinha são especializações de “Frutos do Mar”.
    * Fusca e Caminhonete são especializações de “Veículo”.

Especialização, Herança e Polimorfismo

* **Herança**
  * Herança denota especialização.
  * Herança permite a criação de novas classes com propriedades adicionais a classe original.
  * Em geral, se uma classe B estender (herdar) uma classe A, a classe B vai herdar métodos e atributos da classe A e implementar alguns recursos a mais.
  * Herança implementa o relacionamento É-UM.

Especialização, Herança e Polimorfismo

* **Herança**
  * Exemplo: classes Pedido e PedidoExpresso

    * PedidoExpresso É-UM Pedido
    * PedidoExpresso é uma especialização de Pedido. Estende Pedido.
    * PedidoExpresso herda todas as definições (atributos e métodos) já implementadas na classe Pedido.

Especialização, Herança e Polimorfismo

* **Herança**
  * Exemplos:
    * Classes Mamífero e Cachorro
    * Cachorro É-UM Mamífero

    * Classes Veículo e Carro
    * Carro É-UM Veículo

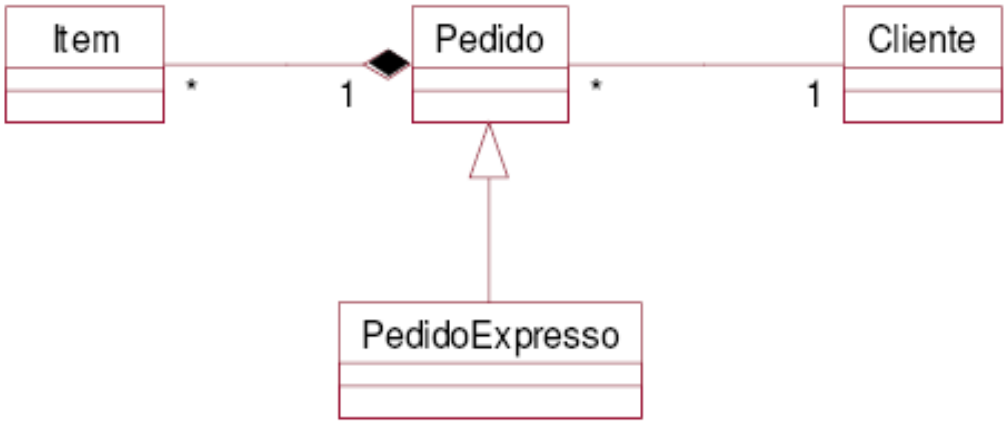
    * Classes Empregado e Gerente
    * Gerente É-UM Empregado

Especialização, Herança e Polimorfismo

* **Herança**
  * A classe herdeira é chamada de subclasse (ou classe derivada) e a classe herdada é chamada de superclasse (ou classe base).
  * Carro é subclasse de Veículo
  * Veículo é superclasse de Carro
  * Carro é a classe derivada da classe base Veículo

Especialização, Herança e Polimorfismo

* **Herança, Composição e Associação**



* Herança: PedidoExpresso É-UM Pedido
* Composição: Pedido TEM-UM (tem vários) Item
* Associação: Pedido está associado a um Cliente

Especialização, Herança e Polimorfismo

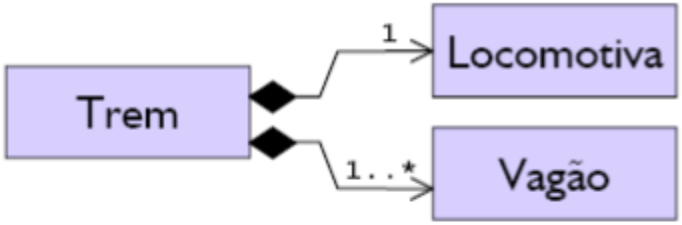
* **Composição, Agregação e Associação**

- *Composição:* *a "é parte essencial de" b* 
- *Agregação:* *a "é parte de" b*
- *Associação:* *a "é usado por" b* 

* Notações muito utilizadas em Análise e Projeto de Sistemas.
* Quanto melhor o modelo de requisitos, melhor será o sistema depois de codificado.

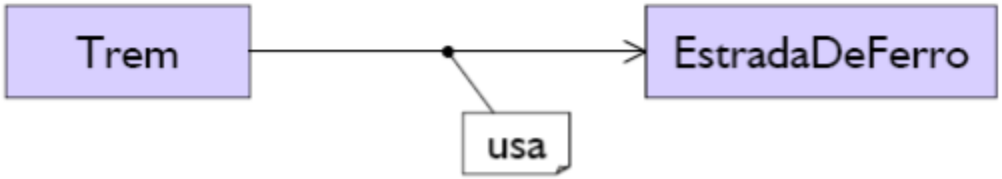
Composição, Agregação e Associação

- ***Composição***: *um trem é formado por locomotiva e vagões*



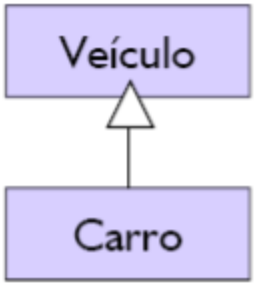
- ***Agregação***: *uma locomotiva tem um farol (mas não vai deixar de ser uma locomotiva se não o tiver)*

- ***Associação***: *um trem usa uma estrada de ferro (não faz parte do trem, mas ele depende dela)*



Herança

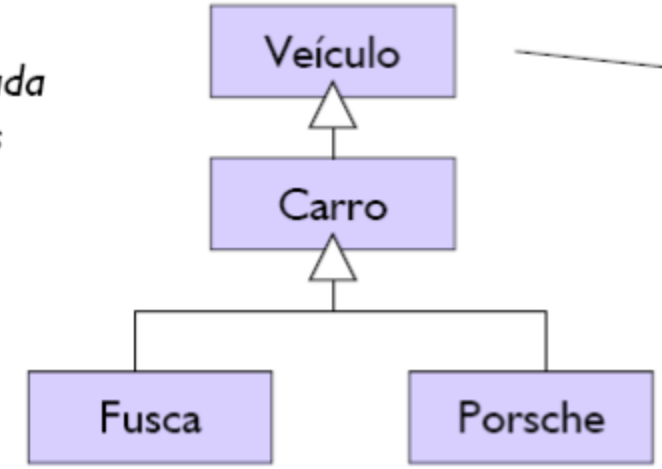
- *Um carro é um veículo*

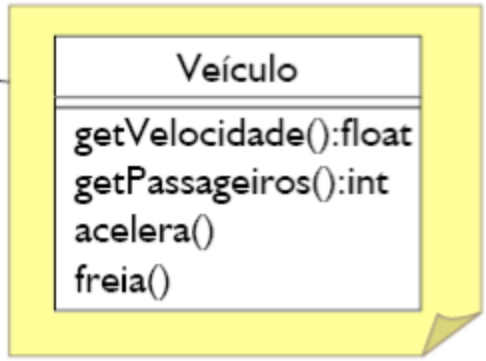


representação UML

- *Fuscas e Porsches são carros (e também veículos)*

representação UML simplificada (não mostra os métodos)





representação UML detalhada de 'Veículo'

Especialização, Herança e Polimorfismo

* **Herança**

  Sintaxe:

  -Palavra-chave **extends**.

  -Utiliza-se na definição da subclasse a palavra-chave extends seguida pelo nome da super-classe.

```
class SuperClasse {
...
}
class SubClasse  extends  SuperClasse {
...
}
```

Exercício

Implemente a hierarquia de classes mostrada abaixo:

Classe: **Pessoa**

Atributos: nome (tipo String) e telefone (tipo String). Todos atributos privados;

Método construtor para inicializar os atributos;

Métodos *get* e *set* para obter e alterar cada um dos atributos; implementar o método toString().

Classe: **PessoaFisica** (subclasse de Pessoa)

Atributos: sexo (tipo char: M-Masculino, F-Feminino), identidade e cpf (tipo String). Atributos privados;

Método construtor para inicializar os atributos;

Métodos *get* e *set* para obter e alterar o atributo; implementar o método toString().

Classe: **PessoaJuridica** (subclasse de Pessoa)

Atributos: cnpj (tipo String) e razaoSocial (tipo String). Todos atributos privados;

Método construtor para inicializar os atributos;

Métodos *get* e *set* para obter e alterar cada um dos atributos; implementar o método toString().

Classe: **Funcionario** (subclasse de PessoaFisica)

Atributos: carteiraTrabalho (tipo String), dataContratacao (tipo GregorianCalendar) e salario (tipo double). Todos atributos privados;

Método construtor para inicializar os atributos;

Métodos *get* e *set* para obter e alterar cada um dos atributos; implementar o método toString().

Classe: **Gerente** (subclasse de Funcionario)

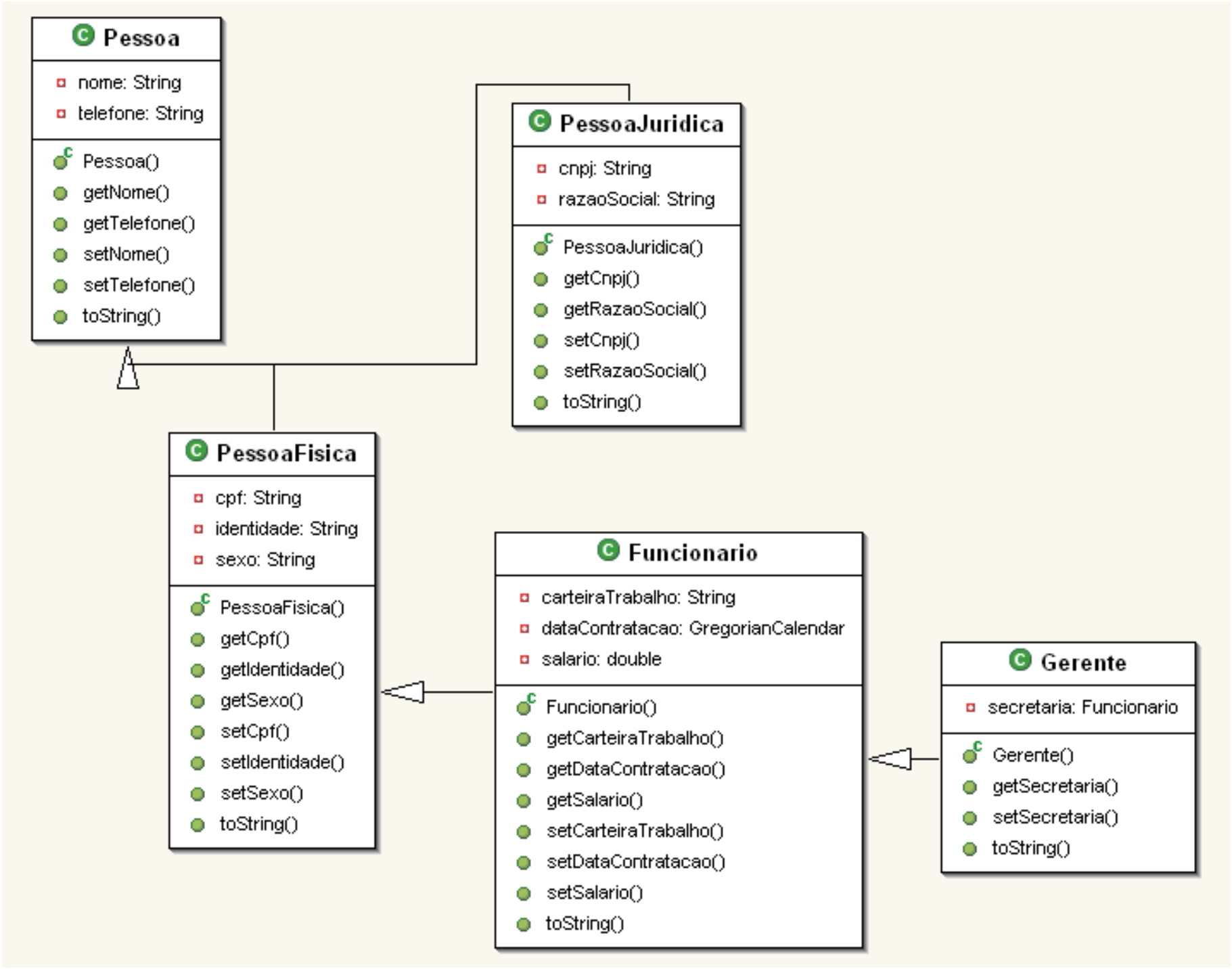
Atributo: secretaria (tipo Funcionario). Atributo privado;

Método construtor para inicializar os atributos;

Métodos *get* e *set* para obter e alterar o atributo; implementar o método toString().

Classe: **Principal**

Classe com um método ***main*** para testar a hierarquia. Crie objetos de cada uma das classes e execute cada um de seus métodos.


Pessoa
nome: String
telefone: String
Pessoa()
getNome()
getTelefone()
setNome()
setTelefone()
toString()
PessoaJuridica
cnpj: String
razaoSocial: String
PessoaJuridica()
getCnpj()
getRazaoSocial()
setCnpj()
setRazaoSocial()
toString()
PessoaFisica
cpf: String
identidade: String
sexo: String
PessoaFisica()
getCpf()
getIdentidade()
getSexo()
setCpf()
setIdentidade()
setSexo()
toString()
Funcionario
carteiraTrabalho: String
dataContratacao: GregorianCalendar
salario: double
Funcionario()
getCarteiraTrabalho()
getDataContratacao()
getSalario()
setCarteiraTrabalho()
setDataContratacao()
setSalario()
toString()
Gerente
secretaria: Funcionario
Gerente()
getSecretaria()
setSecretaria()
toString()

Especialização, Herança e Polimorfismo

* **Herança**
  * SubClasse pode ser usada sempre que a SuperClasse é esperada (contrário não vale).
  * A subclasse possui todos os métodos e atributos da superclasse. Subclasse É-UM superclasse.

Especialização, Herança e Polimorfismo

* **Herança**
  * SuperClasse sp;
  * SubClasse sb = new SubClasse();
  * sp = sb; // correto
  * SuperClasse sp2 = new SubClasse(); // correto
  * sb = sp2; // incorreto

  * Todos os métodos na SuperClasse são herdados sem modificação na SubClasse.
  * Todos os atributos que formam a SuperClasse formam também a SubClasse.

Especialização, Herança e Polimorfismo

* **Herança – Atributos herdados**
  * Subclasses podem adicionar novos atributos.
  * Subclasses não podem remover atributos herdados.
  * Subclasses podem criar atributos com os mesmos nomes dos atributos originais (herdados).
  * Neste caso, os atributos originais são escondidos (shadow).

Especialização, Herança e Polimorfismo

* **Herança – Métodos herdados**
  * Subclasses podem adicionar novos métodos.
  * Subclasses não podem remover métodos herdados.
  * Subclasses podem redefinir (override) métodos originais (herdados), usando a mesma assinatura.
  * Subclasses não podem reduzir o nível de acesso de métodos herdados. O contrário é válido.
  * Exemplo:
    * de public para private // incorreto
    * de public para protected // incorreto
    * de protected para private // incorreto
    * de protected para public // correto

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**
  * A Sobreposição de métodos (override) é um conceito do polimorfismo que nos permite reescrever um método, ou seja, podemos reescrever nas classes filhas métodos criados inicialmente na classe pai, os métodos que serão sobrepostos, diferentemente dos sobrecarregados, devem possuir o mesmo nome, tipo de retorno e quantidade de parâmetros do método inicial, porém o mesmo será implementado com especificações da classe atual, podendo adicionar algo a mais ou não.

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**
  * Enquanto a sobrecarga da vida a variação de métodos, a sobreposição possibilita a extensibilidade dos mesmos, pois com ela podemos reescrever métodos criados anteriormente, permitindo assim a criação de versões mais específicas deles.

  * Com a sobreposição podemos pegar um método genérico e transformá-lo em específico, implementando novas funcionalidades pertinentes da classe à qual ele está.

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**
  * Diferente da sobrecarga, a sobreposição funciona por meio do sistema de herança, e para a mesma funcionar o nome e lista de argumentos dos métodos devem ser totalmente iguais aos da classe herdada.

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**
  * Exemplo:

```
// Suponha que estes métodos estão em classes separadas

public void exibeMensagem(){
System.out.println("Mensagem classe pai"); }

public void exibeMensagem(){
System.out.println("Mensagem classe filho"); }

// Ao invocar os métodos:
classePai.exibeMensagem(); //Imprime: "Mensagem classe pai"
classeFilho.exibeMensagem(); //Imprime: "Mensagem classe filho"
```

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**
  * Para que acidentalmente você não mude o nome do método da classe Pai sem alterar também o nome do método da classe Filha, use a anotação **@Override** no topo do método da classe Filha.

  * Dessa forma, ao modificar o nome do método da classe Pai, o código NÃO irá compilar até que o método da classe Filha também tenha seu nome alterado.

Especialização, Herança e Polimorfismo

* **Sobreposição de métodos**

```
public class SuperClasse {
    public void imprime() {
        System.out.println("imprime");
    }
}

public class MinhaClasse extends SuperClasse {
    @Override
    public void imprime() {
        System.out.println("imprime diferente");
    }
}
```

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus "ancestrais".

  * Entretanto, em alguns casos, é necessário que as ações para um mesmo método sejam diferentes.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Em outras palavras, o *polimorfismo* consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

  * Como um exemplo, temos um objeto genérico "Eletrodoméstico". Esse objeto possui um método, ou ação, "Ligar()". Temos dois objetos, "Televisão" e "Geladeira", que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método "Ligar()".

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Polimorfismo significa muitas formas. Em termos de programação, significa que um único nome de classe ou de método pode ser usado para representar comportamentos diferentes dentro de uma hierarquia de classes.
  * A decisão sobre qual comportamento utilizar é tomada em tempo de execução.
  * As linguagens que suportam polimorfismo são chamadas de linguagens polimórficas. As que não suportam são chamadas de linguagens monomórficas. Nestas, cada nome é vinculado estaticamente ao seu código.

Especialização, Herança e Polimorfismo

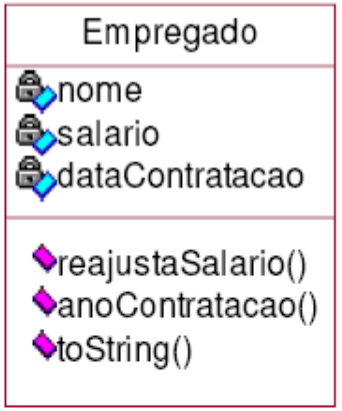
* **Polimorfismo**
  * Clientes podem ser implementados genericamente para chamar uma operação de um objeto sem saber o tipo do objeto.
  * Se são criados novos objetos que suportam uma mesma operação, o cliente não precisa ser modificado para suportar o novo objeto.
  * O polimorfismo permite que os clientes manipulem objetos em termos de sua superclasse comum.
  * O polimorfismo torna a programação orientada por objetos eficaz, permitindo a escrita de código genérico, fácil de manter e de estender.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Quando o método a ser invocado é definido durante a compilação do programa, o mecanismo de ligação prematura (early binding) é utilizado.
  * Para a utilização de polimorfismo, a linguagem de programação orientada por objetos deve suportar o conceito de ligação tardia (late binding), onde a definição do método que será efetivamente invocado só ocorre durante a execução do programa. O mecanismo de ligação tardia também é conhecido pelos termos ligação dinâmica (dynamic binding) ou ligação em tempo de execução (run-time binding).

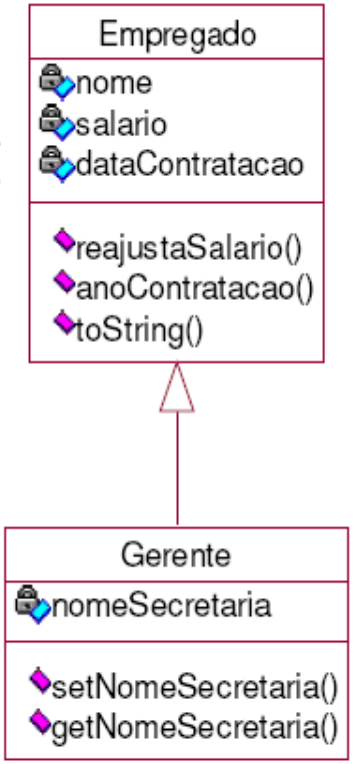
Exemplo – Classe Empregado

```
import cursojava.*;

public class Empregado {
   private String nome;
   private double salario;
   private Day dataContratacao;

   public Empregado(String nome, double salario, Day dataContr) {
      this.nome = nome;
      this.salario = salario;
      dataContratacao = dataContr;      }

   public void reajustaSalario(double percentagem) {
      salario *= 1 + percentagem / 100;      }

   public int anoContratacao() {
      return dataContratacao.getYear();     }

   public String toString() {
      return nome + " " + salario + " " + anoContratacao()   }
}
```



Exemplo – Classe Gerente

```
import cursojava.*;

public class Gerente extends Empregado {
  private String nomeSecretaria;

  public Gerente(String nome, double salario, Day dataContr) {
    super(nome, salario, dataContr);
    nomeSecretaria = "";    }

  public void reajustaSalario(double percentagem) {
    // adiciona 1/2% de bônus para cada ano de serviço
    Day hoje = new Day();
    double bonus = 0.5 * (hoje.getYear() - anoContratacao());
    super.reajustaSalario(percentagem + bonus);    }

  public void setNomeSecretaria(String nome) {
    nomeSecretaria = nome;     }

  public String getNomeSecretaria() {
    return nomeSecretaria;     }
}
```



Exemplo – Classe GerenteTeste

```
import cursojava.*;

public class GerenteTeste {
   public static void main(String[] args) {
      Gerente chefe = new Gerente("Pedro Paulo", 7500, new Day(1998,12,15));
      chefe.setNomeSecretaria("Patricia Moreira");
      Empregado[] emps = new Empregado[4];
      emps[0] = chefe;
      emps[1] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
      emps[2] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
      emps[3] = new Empregado("Joao Barros", 300, new Day(2003,3,15));
       for (int i = 0; i < 4; i++) {
          emps[i].reajustaSalario(5);
          System.out.println(emps[i].toString());
      }
      System.out.println("A Secretaria do Departamento e' " +
                                 chefe.getNomeSecretaria());     }
```

Polimorfismo

Exemplo explicado

- Polimorfismo no exemplo anterior:
  - O objeto emps é do tipo Empregado[], portanto uma posição do vetor pode receber um objeto do tipo Empregado ou do tipo Gerente (ou do tipo de qualquer subclasse de Empregado que venha a existir).
  - O método reajustaSalario() existe tanto na classe Empregado quanto na classe Gerente. No código

```
for (int i = 0; i < 4; i++) {
    emps[i].reajustaSalario(5);
```

  o compilador executa o código de reajustaSalario() da classe Empregado se o objeto da posição emps[i] for do tipo Empregado ou o código da classe Gerente se o objeto emps[i] for do tipo Gerente.

Exemplo explicado

- Se não houvesse polimorfismo:
  - O vetor emps só receberia objetos do tipo Empregado e portanto em tempo de compilação já se saberia qual código de reajustaSalario() seria executado. A classe GerenteTeste mudaria para:

```
. . .
emps[0] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
emps[1] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
emps[2] = new Empregado("Joao Barros", 300, new Day(2003,3,15));
for (int i = 0; i < 3; i++) {
    emps[i].reajustaSalario(5);   // executa código da classe Empregado
    System.out.println(emps[i].toString()); }
chefe.reajustaSalario(5);       // executa código da classe Gerente
System.out.println(chefe.toString());
. . .
```

Outro exemplo

Considere o seguinte trecho de código:

```
public class Empresa {
    private static final int MAXEMPS = 1000;
    private Empregado[] emps = new Empregado[MAXEMPS];
    private numEmps = 0;  // número de empregados na lista
    . . .
    public void adicionaEmpregado (Empregado emp) {
            if (numEmps = MAXEMPS)  return;  // lista cheia
            emps[numEmps] = emp;
            numEmps++;
    }
    public void reajustaSalarios(double perc) {
        for (int i=0; i<numEmps; i++)
        emps[i].reajustaSalario(perc);
    }
}
```

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Com o uso de **polimorfismo**, o código da classe Empresa é genérico:
  - O método adicionaEmpregado() pode receber como parâmetro tanto um objeto do tipo Empregado quanto do tipo Gerente e adicioná-lo à lista emps que é do tipo Empregado[].
  - O método reajustaSalarios() chama o método reajustaSalario() da classe Empregado ou da classe Gerente dependendo do tipo do objeto na posição emps[i].
  - Se for criada uma nova subclasse de Empregado, por exemplo, a classe Diretor, o código da classe Empresa não precisaria ser modificado para inserir e reajustar salários dos objetos do tipo Diretor.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Se não houvesse polimorfismo, a classe Empresa seria algo do tipo:

```
public class Empresa {
    private Empregado[] emps = new Empregado[1000];
    private Gerente[] gers = new Gerente[50];
    . . .
    public void adicionaEmpregado (Empregado emp) { . . . }
    public void adicionaGerente (Gerente ger) { . . . }
    public void reajustaSalariosEmps(double perc) { . . . }
    public void reajustaSalariosGers(double perc) { . . . }
}
```

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Se fosse criada a classe Diretor, a classe Empresa também deveria ser alterada, incluindo uma lista de diretores e métodos para adicionar e reajustar salários de diretores.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * O tipo de polimorfismo mostrado nos exemplos anteriores é conhecido como Polimorfismo de Inclusão ou Polimorfismo Puro.
  * A Sobrecarga de Métodos também é um tipo de polimorfismo (ad-hoc). Ela permite que vários métodos tenham o mesmo nome, desde que o tipo e/ou o número de parâmetros sejam diferentes. Exemplo: os métodos da classe Math:
    * static int max (int a, int b)
    * static long max (long a, long b)
    * static float max (float a, float b)
    * static double max (double a, double b)

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Se não houvesse sobrecarga cada método teria de ter um nome diferente, mesmo tendo o mesmo significado. Exemplo:
    * static int maxInt (int a, int b)
    * static long maxLong (long a, long b)
    * static float maxFloat (float a, float b)
    * static double maxDouble (double a, double b)
  * Com polimorfismo, pode-se chamar simplesmente max() e passar os parâmetros. O compilador chamará o método correto internamente.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Quanto mais polimorfismo for utilizado, mais organizado o código ficará.
  * Evite utilizar conversão de tipo explícita. Vejamos a seguir do que se trata.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Considere o seguinte trecho de código do exemplo da classe GerenteTeste:

```
Gerente chefe = new Gerente("Pedro Paulo", 7500, new Day(1998,12,15));
chefe.setNomeSecretaria("Patricia Moreira");
Empregado[] emps = new Empregado[4];
emps[0] = chefe;
emps[1] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
```

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Neste exemplo, o objeto chefe é do tipo Gerente (subclasse) e o objeto emps é do tipo Empregado[] (superclasse). Portanto a atribuição abaixo é legal:

    * emps[0] = chefe // atribuição legal
    * Porém, o verdadeiro tipo está sendo subestimado.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Ao objeto emps[i] somente se aplicam os métodos da classe Empregado. Assim,

    * emps[0].setNomeSecretaria(“Patricia Ferreira”); // erro
    * resulta em erro, mesmo sendo o objeto emps[0] do tipo Gerente.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Para converter novamente o objeto para sua classe original, usa-se a conversão de tipo explícita (cast), semelhante ao que foi feito com a conversão de tipos primitivos:

    * Gerente chefe2 = (Gerente) emps[0]; // legal
    * ((Gerente)emps[0]).setNomeSecretaria("Patricia Ferreira"); // legal

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Converter um objeto de uma superclasse para uma subclasse pode resultar em erro:

    * Gerente chefe3 = (Gerente) emps[1]; // erro em tempo de execução
    * // emps[1] não é um Gerente!

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Pode-se fazer uma conversão de tipo explícita somente dentro de uma hierarquia de classes.

    * Ponto2D p1 = (Ponto2D) emps[1]; // erro de compilação
    * // resulta em erro de compilação, pois Ponto2D não é subclasse de Empregado.

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Para descobrir se um objeto é ou não instância de uma classe, usa-se o operador **instanceof**.
    * Exemplo:

```
for (int i = 0; i < 4; i++) {
emps[i].reajustaSalario(5);
System.out.println(emps[i].toString());
if (emps[i] instanceof Gerente)
System.out.println("A Secretaria do Gerente e' " +
((Gerente)emps[i]).getNomeSecretaria());
}
```

Especialização, Herança e Polimorfismo

* **Polimorfismo**
  * Conversões de tipo explícita
    * Advertência: não use conversão de tipo explícita de forma indiscriminada! Sempre que possível, coloque os métodos genéricos nas superclasses e use polimorfismo.
    * No exemplo anterior, a única razão para se fazer um cast é usar um método que seja único para os gerentes como getNomeSecretaria(). Para os demais, como reajustaSalario() e toString(), deve-se usar o polimorfismo.

Especialização, Herança e Polimorfismo

* **Coleções heterogêneas**
  * Perceba que a lista de empregados:
  * `Empregado[] emps = new Empregado[4];`
  * é uma coleção heterogênea, ou seja, uma coleção com tipos diferentes de objetos.
  * A lista possui objetos do tipo **Empregado** e objetos do tipo **Gerente**.

Especialização, Herança e Polimorfismo

* **Coleções heterogêneas x homogêneas**
  * Em Java também é possível criar coleções de objetos que tem uma classe em comum. Essas coleções são chamadas de coleções **homogêneas**.

  * Exemplo:

```
Agenda[] nome = new Agenda[3];
nome[0] = new Agenda("Carina");
nome[1] = new Agenda("Cristina");
nome[2] = new Agenda("Juliana");
```

Especialização, Herança e Polimorfismo

* **Coleções heterogêneas**
  * A linguagem de programação Java tem uma classe chamada **Object** e todas as classes em Java extendem essa classe Object. Assim você pode fazer coleções de todos os tipos de elementos devido ao polimorfismo. Essas coleções de vários tipos são chamadas coleções **heterogêneas.**

  * Exemplo:

```
Animal[] a = new Animal[3];
a[0] = new Peixe();
a[1] = new Cavalo();
a[2] = new Aguia();
```

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * A sobrecarga de métodos (overload) é um conceito do polimorfismo que consiste basicamente em criar variações de um mesmo método, ou seja, a criação de dois ou mais métodos com nomes totalmente iguais em uma classe.

  * A Sobrecarga permite que utilizemos o mesmo nome em mais de um método contanto que suas listas de argumentos sejam diferentes para que seja feita a separação dos mesmos.

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * Para entender melhor a sobrecarga, vamos pensar que estamos implementando uma calculadora simples que some apenas dois valores do mesmo tipo por vez.

  * Nela teremos o método calcula que será sobrecarregado com variações de tipos de soma conforme mostrado a seguir.

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**

```
public class calculadora{
  public int calcula( int a, int b){
    return a+b;
  }
  public double calcula( double a, double b){
    return a+b;
  }
  public String calcula( String a, String b){
    return a+b;
  }
```

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * A classe calculadora possui três métodos que somam dois valores do mesmo tipo, porém, os mesmos possuem o mesmo nome, então, como vamos saber se o programa principal vai chamar o método correto ao convocarmos o calcula()?

  * O programa, ao receber o calcula() com os parâmetros passados, verificará na classe calculadora no tempo de execução qual dos seguintes métodos está implementado para receber o parâmetro e convocará o mesmo.

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * Faça o teste:

```
public static void main(String args[]){
    calculadora calc = new calculadora();
    System.out.println(calc.calcula(1,1));
    System.out.println(calc.calcula(2.0,6.1));
    System.out.println(calc.calcula("vi","ram?"));
  }
```

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * A sobrecarga é muito utilizada em construtores, pois esses consistem em linhas de código que serão sempre executadas quando uma classe for instanciada. Entende-se instanciada quando criamos um objeto a partir dela.

  * Por regra, o programa cria um construtor sem implementação para cada classe criada que, no caso, será o construtor padrão, porém podemos criar quantos construtores acharmos necessários.

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * Podemos entender os construtores como uma base inicial que os objetos terão ao serem criados. Por base os construtores devem possuir o mesmo nome que a classe na qual eles estão. Na classe calculadora temos um construtor padrão sem implementação, porém não o vemos.

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * Vamos implementar a nossa classe calculadora com atributos e a sobrecarga de construtores:

```
public class calculadora{

    private String modelo;
    private String marca;
    private String uso;

    //Sobrecarga de construtores.
    public calculadora(){
      // esse é o construtor padrão que o programa cria para todas as classes.
    }
    public calculadora(String marca,String modelo){
             this.marca=marca;
             this.modelo=modelo;
    }

    public calculadora(String marca,String modelo,String uso){
             this.marca=marca;
             this.modelo=modelo;
             this.uso=uso;
    }
    public int calcula(int a,int b){
         return a+b;
    }
    public double calcula(double a,double b){
          return a+b;
    }
    public String calcula(String a,String b){
           return a+b;
    }
```

```
 public static void main(String args[]){
     calculadora calc= new calculadora("optpex","N110","Empresarial");
     calculadora cald= new calculadora("Zion","Neo1");
     System.out.println(calc.calcula(900,1000));
     System.out.println(calc.calcula(99.0,100.1));
     System.out.println(calc.calcula("Sobrecarga de "," construtores"));
     System.out.println("calculadora 1 Marca: "+calc.marca+" Modelo:
"+calc.modelo+" Uso: "+calc.uso);
     System.out.println("calculadora 2 Marca: "+cald.marca+" Modelo:
"+cald.modelo);
    }
  }
```

Especialização, Herança e Polimorfismo

* **Sobrecarga de métodos e construtores**
  * A sobrecarga de construtores tem muito em comum com a sobrecarga de métodos; podemos dizer que o conceito de sobrecarga é sempre o mesmo.

  * Portanto entendemos que a sobrecarga é conceito poderoso do polimorfismo, e a mesma permite ao programador mais facilidade na criação de variações de códigos já criados, poupando-o assim de inventar nomes para cada operação que compõem um mesmo escopo.

Resumo de características OO

- *Abstração de conceitos*
  - *Classes, definem um tipo separando interface de implementação*
  - *Objetos: instâncias utilizáveis de uma classe*
- *Herança: "é um"*
  - *Aproveitamento do código na formação de hierarquias de classes*
  - *Fixada na compilação (inflexível)*
- *Associação "tem um"*
  - *Consiste na delegação de operações a outros objetos*
  - *Pode ter comportamento e estrutura alterados durante execução*
  - *Vários níveis de acoplamento: associação, composição, agregação*
- *Encapsulamento*
  - *Separação de interface e implementação que permite que usuários de objetos possam utilizá-los sem conhecer detalhes de seu código*
- *Polimorfismo*
  - *Permite que objeto seja usado no lugar de outro*

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos