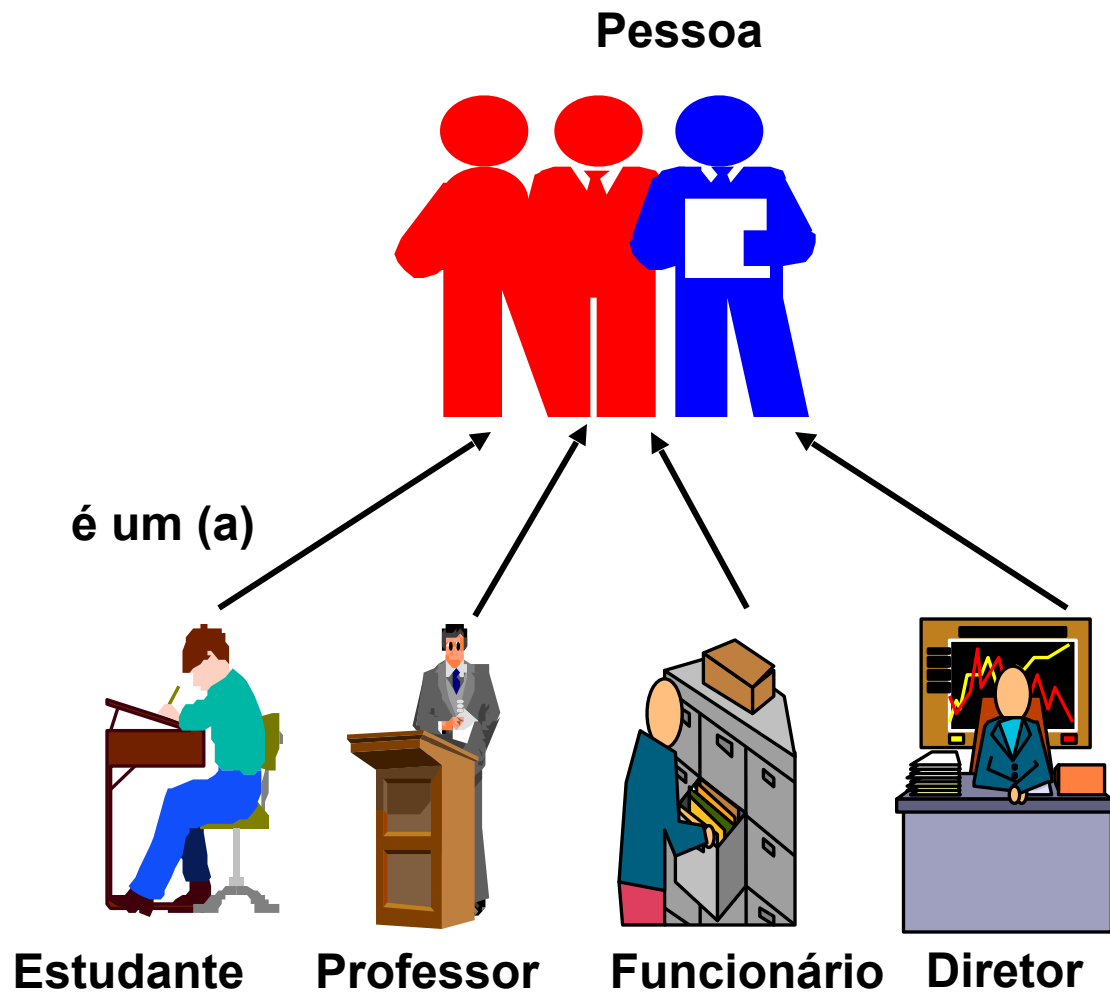


Material IsiFLIX para uso exclusivo de
Nelson de Campos Nolasco
nelsonnolasco@gmail.com

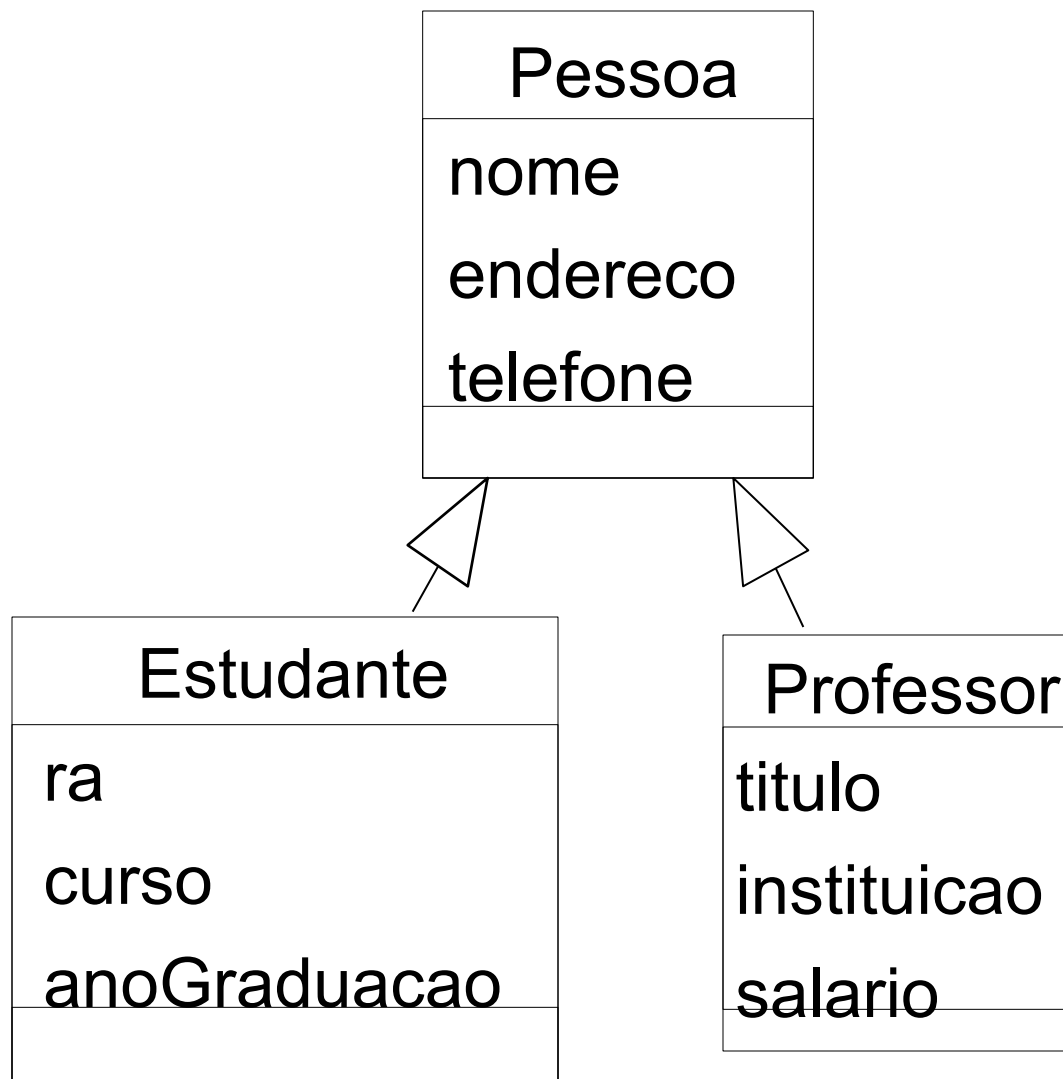


Programação Orientada a Objetos em Java

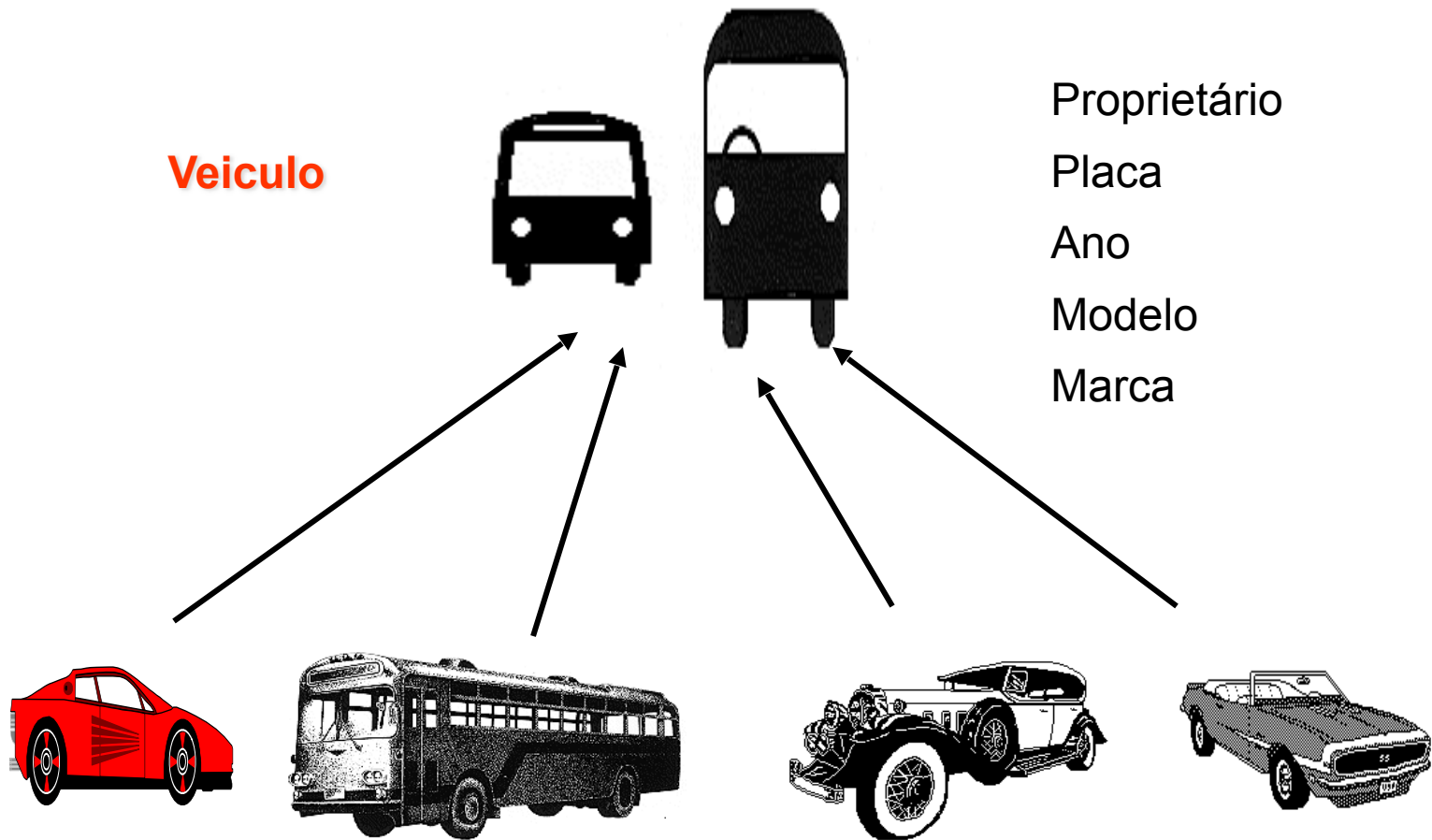
Aula 4 - Herança e Polimorfismo



[isi] Herança



[isi] Herança



[isi] Herança em Java

- O que torna a Orientação a Objetos única é o conceito de herança.
- Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas de uma classe base, ou superclasse.
- A herança é uma forma de reutilização de software em que novas classes são criadas a partir das classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos que as novas classes exigem



[isi] Herança em Java

- A partir de uma classe base, outras classes podem ser especificadas.
- Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da superclasse e acrescenta a elas o que for definido de particularidade para ela.
- Cada subclasse se torna uma candidata a ser uma superclasse para alguma subclasse futura



[isi] Herança em Java

- **Sintaxe:**

```
[modificador] class NomeDaSuperclasse  
{ // corpo da superclasse... }
```

```
[modificador] class NomeDaSubclasse extends  
NomeDaSuperclasse  
{ // corpo da subclasse... }
```

- **extends** - indica que está sendo criada uma nova classe que deriva de uma classe existente
- **Classe existente** - superclasse, classe base ou classe progenitora
- **Nova classe** - subclasse, classe derivada ou classe filha
 - Em geral, as subclasses têm mais funcionalidade que sua superclasse.



[isi] Observe bem

- O termo que Java usa para herança é a palavra reservada "**extends**"
- Se você traduzir ao pé da letra, **extends** significa "**ampliar**"
- Pense numa definição de classe que amplia uma definição já existente
- A classe herdada sempre tem "mais coisas" que a classe-base





Exemplo de Herança (Superclasse- Empregado.java)

```
import java.text.*;

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

    public Empregado (String n, double s) {
        this.setNome(n);
        this.setSalario(s);
    }

    public void setNome(String n){
        nome = n;
    }

    public void setSalario(double sal){
        salario = sal;
    }
}
```





Exemplo de Herança (Superclasse- Empregado.java)

```
public String getNome(){  
    return nome;  
}  
  
public double getSalario(){  
    return salario;  
}  
  
public void aumentarSalario (double percentual){  
    salario *= 1 + percentual / 100;  
}
```



Exemplo de Herança (Superclasse- Empregado.java)

```
public String formatarMoeda () {  
    NumberFormat nf = NumberFormat.getCurrencyInstance();  
    nf.setMinimumFractionDigits(2);  
    String formatoMoeda = nf.format(salario);  
    return formatoMoeda;  
}  
  
public void imprimir () {  
    System.out.println(nome + " " + "salario " + this.formatarMoeda());  
}
```





Exemplo de Herança (Superclasse- EmpregadoTeste.java)

```
public class EmpregadoTeste{  
    public static void main (String args []) {  
        Empregado [] lista = new Empregado [3];  
        lista[0] = new Empregado ("Harry Hacker", 3500);  
        lista[1] = new Empregado ("Carl Cracker", 7500);  
        lista[2] = new Empregado ("Tony Tester", 3800);  
        for (Empregado em: lista)  
            em.imprimir();  
        System.out.println("*****");  
        for (Empregado em: lista){  
            em.aumentarSalario(10);  
            em.imprimir();  
        }  
    }  
}
```





Exemplo de Herança (Subclasse- Gerente.java)

```
import java.text.*;

public class Gerente extends Empregado {
    private String nomeSecretaria;

    public Gerente (String n, double s, String nSec) {
        super (n, s);
        this.setNomeSecretaria(nSec);
    }

    public String getNomeSecretaria () {
        return nomeSecretaria;
    }

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





Exemplo de Herança (Subclasse-Gerente.java)

```
public void aumentarSalario (double percentual) {  
    //adiciona bonus de 20% ao valor do salario  
  
    double bonus = 20;  
  
    super.aumentarSalario(percentual + bonus);  
  
}  
  
} // fim da classe Gerente
```



[isi] Herança em Java

- Palavra-chave **super** refere-se a uma superclasse
 - indica a chamada ao construtor da superclasse
 - se a superclasse não contiver o construtor padrão e o construtor da subclasse não chamar nenhum outro construtor da superclasse explicitamente → compilador java vai informar um erro
- Em um relacionamento de herança:
 - é necessário apenas indicar as diferenças entre a subclasse e superclasse → o reuso é automático
 - é necessário redefinir métodos → um dos primeiros motivos para usar herança



[isi] Herança em Java

- Exemplo da **subclasse** `Gerente`:
 - redefinição do método `aumentarSalario()`
 - para que ele funcione diferente para gerentes e empregados comuns
 - esse método **não tem acesso direto** às variáveis de instância privados da **superclasse**, ou seja, esse método não pode alterar diretamente a variável de instância `salario`, embora cada objeto `Gerente` tenha uma variável de instância `salario`
 - **somente os métodos** da classe `Empregado` têm acesso aos atributos de instância privados
 - Resultado dessa redefinição para objetos da classe `Gerente`:
 - Quando se dá a todos os empregados um aumento de 5%, os gerentes vão receber um aumento maior automaticamente





Exemplo de Herança

(Classe Principal-GerenteTeste.java)

```
import java.text.*;

public class GerenteTeste {

    public static void main (String args[]){

        Gerente g = new Gerente ("Carl Cracker", 7500, "Harry Hacker");

        Empregado [] lista = new Empregado [3];

        lista[0] = g;

        lista[1] = new Empregado ("Harry Hacker", 3500);

        lista[2] = new Empregado ("Tony Tester", 3800);

        for (Empregado em: lista)
            em.aumentarSalario(10);

        for (Empregado em: lista)
            e[i].imprimir();

        System.out.println("O nome da secretaria do depto e:" +g.getNomeSecretaria());
```



[isi] Subclasses

- Para saber se a herança é adequada para um programa
 - ter em mente que qualquer objeto que seja uma instância de uma subclasse precisa ser utilizável no lugar de um objeto que seja uma instância de superclasse
 - objetos de subclasse são utilizáveis em qualquer código que use a superclasse
 - um objeto de subclasse pode ser passado como argumento para qualquer método que espera um parâmetro de superclasse
 - um objeto de superclasse não pode geralmente ser atribuído a um objeto de subclasse
`g = e[i]; //erro`
 - Os atributos só podem ser adicionados, e não removidos, os objetos de uma subclasse herdados têm, pelo menos, tantos atributos de dados quanto os objetos de superclasse



[isi] Subclasses

- Uma das regras fundamentais da herança:
 - um método definido em uma subclasse com o mesmo nome e mesma lista de parâmetros que um método em uma de suas classes antecessoras oculta o método da classe ancestral a partir da subclasse

1. Coloque métodos e atributos comuns na superclasse
2. Use herança para modelar uma relação de “estar contido em” (um objeto da subclasse é um (a) objeto da superclasse)

Exemplo: classe `Empreiteiro`

3. Não use herança a menos que todos os métodos herdados façam sentido

Exemplo: classe `Feriado`

```
class Feriado extends Day { . . . }
```

- Um dos métodos públicos da classe `Day` é `avancarData ()`, que permite transformar dias feriados em dias normais, de modo que não é um método apropriado para se fazer com dias feriados.

```
Feriado natal;
```

```
natal.avancarData(10); // neste caso, um feriado é um dia mas  
não um Day (objeto)
```

4. Use polimorfismo, não informação de tipo

- Sempre que você encontrar código do tipo

```
if (x é o tipo 1) acao1(x);  
else if (x é o tipo 2) acao2(x);
```

- Pense em **Polimorfismo**

- acao1 e acao2 representam um conceito comum?
- Caso afirmativo, faça o conceito virar um método de uma superclasse. Assim poderá simplesmente chamar:
`x.acao()`;
- o ponto a ser observado aqui é que o código para usar métodos polimórficos é muito mais fácil de se manter e estender que um código que use múltiplos testes de tipos de dados

[isi] Polimorfismo

- O polimorfismo permite escrever programas de uma forma geral para tratar uma ampla variedade de classes relacionadas existentes e ainda a serem especificadas
- **Polimorfismo:**
 - é a capacidade de um objeto decidir que método aplicar a si mesmo.
 - embora a mensagem possa ser a mesma - os objetos podem responder diferentemente
 - aplicado a qualquer método que seja herdado de uma superclasse



[isi] Polimorfismo

- Comunicação entre objetos: envio de mensagens
 - ao enviar uma mensagem que pede para uma subclasse aplicar um método usando certos parâmetros
 - a subclasse verifica se ela tem ou não um método com esse nome e com exatamente os mesmos parâmetros. Se tiver, usa-o.
 - caso contrário: a superclasse torna-se responsável pelo processamento da mensagem e procura por um método com esse nome e esses parâmetros. Se encontrar, chama esse método.
- Exemplo:
 - o método `aumentarSalario()` da classe `Gerente` é chamado em vez do método `aumentarSalario()` da classe `Empregado` quando se envia uma mensagem `aumentarSalario` ao objeto `Gerente`

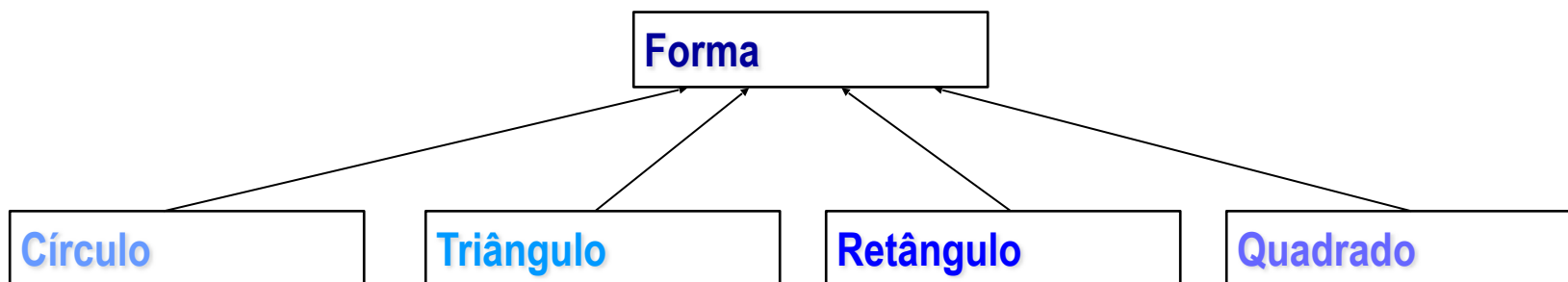


[isi] Polimorfismo

- a chave para fazer polimorfismo:
 - ligação tardia (late binding) ou ligação dinâmica:
 - o compilador não gera o código para chamar um método em tempo de compilação
 - em vez disso, cada vez que se aplica um método a um objeto, o compilador gera código para calcular que método deve ser chamado, usando informações de tipo do objeto
 - o mecanismo de chamada de método tradicional é chamado ligação estática: determinada em tempo de compilação

[isi] Polimorfismo

- Vinculação dinâmica de objeto
 - Em programação OO, cada uma dessas classes pode ser dotada da capacidade de desenhar a si própria.
 - Cada classe tem seu próprio método `desenhar` e a implementação do método `desenhar` é bem diferente para cada forma
 - Ao desenhar uma forma, qualquer que seja essa forma, seria ótimo se poder tratar genericamente como objetos da superclasse `Forma`



[isi] Polimorfismo

- Assim, para desenhar qualquer forma, poderíamos simplesmente chamar o método `desenhar` da superclasse `Forma` e deixar o programa determinar dinamicamente (durante a execução), com base no tipo de objeto real, qual é o método `desenhar` de subclasse que deve ser utilizado.
- Para permitir esse tipo de comportamento, declaramos `desenhar` na superclasse e então sobrescrevemos `desenhar` em cada uma das subclasses para `desenhar` a forma apropriada
- Se utilizarmos uma referência para a superclasse para fazer referência a um objeto da subclasse e invocarmos o método `desenhar`, o programa escolherá o método `desenhar` correto da subclasse dinamicamente.



[isi] Classes Abstratas

- Classes abstratas são classes que devem ser definidas com o propósito de criar apenas um modelo de implementação
- As classes abstratas não podem ter objetos instanciados
- Uso: apenas para usar uma referência genérica
- Exemplo: Classe Forma Geométrica (Aula 4)

[isi] Classes Abstratas

- As classes abstratas podem ser tornar ferramentas poderosas para a construção de sistemas complexos e que envolvam vários níveis de hierarquia de classes
- Exemplo de uso de classes abstratas
 - Sistema de folha de pagamento de instituição de ensino
 - Modo de calcular o salário de um professor não é o mesmo de um funcionário administrativo ou um estagiário
 - Entretanto todos podem ser classificados como “Funcionários”



[isi] Polimorfismo

- Mesmo com classes abstratas, o polimorfismo continua válido
- Muitas classes podem usar a referência à classe abstrata e utilizar os métodos de cada instância específica
- Algumas características:
 - Classes abstratas podem ter métodos não abstratos
 - Classes abstratas podem ter métodos abstratos
 - A classe que herda de uma classe abstratas com métodos abstratos DEVE redefinir o corpo do método
 - Classes não-abstratas não podem ter métodos abstratos
 - Se uma classe tiver métodos abstratos, a classe DEVE ser abstrata



[isi] Herança Múltipla

- Herança múltipla é um caso específico de polimorfismo
 - Uma classe possui a relação “é-um” com mais de um antecessor
 - Exemplo: Instituição de Ensino
 - Classe Funcionário
- Java não suporta Herança Múltipla!
 - Na verdade, não é possível realizar a operação **extends** com mais de uma classe

[isi] Como fazer?

- Utilizar Interfaces
- Interface é um “contrato” no qual o objeto “compromete-se” a implementar todos os métodos
- Interface = classe abstrata com métodos abstratos
 - Programador define interface e compilador “enxerga” uma classe abstrata com métodos abstratos
- Interface NÃO DEVE ter corpo de métodos
- Como realizar herança com interfaces?



[isi] Interfaces

- Declaração de Interface

```
public interface nome_da_interface  
{  
    cabeçalho_do_metodo_1 (parametros);  
    cabeçalho_do_metodo_2 (parametros);  
    cabeçalho_do_metodo_3 (parametros);  
}
```



[isi] Herança com Interface

- Implementando uma interface

```
public class nome_classe implements nome_da_interface
{
    cabeçalho_do_metodo_1(parametros) {
        corpo do método 1
    }
    cabeçalho_do_metodo_2(parametros) {
        corpo do método 2
    }
}
```





Mas e a Herança Múltipla?

- A classe filha deve estender a classe pai e implementar as demais interfaces
- Neste caso é possível implementar mais de uma interface
- Lembre-se:
 - TODOS os métodos definidos nas interfaces devem ser redefinidos

[isi] Sintaxe

```
public class Classe extends classe_pai  
                implements nome_da_interface1,  
                            nome_da_interface2,...  
  
{  
    ....  
}
```



[isi] Exemplos

- Objeto que deve ter as características de um objeto gráfico e também de uma thread
 - Interface nativa Runnable
- Objeto de armazenamento que deve ter as características de um objeto de negócios e também ferramentas para armazenamento em bancos de dados
 - Interface DAO definida pelo usuário
- Objeto de negócios e também deve ser um elemento “serializável”
 - Interface nativa Serializable

