

# ORIENTAÇÃO A OBJETOS

## AULA 7

### Classes Empacotadores, Herança e Janelas de Diálogo

Vandor Roberto Vilardi Rissoli



# APRESENTAÇÃO


- Classes Empacotadores
- Herança
- Sobreposição em *Object*
- Janelas de Diálogo
- Referências



# Classes Empacotadores (*wrappers*)

Na Programação poderão existir situações onde seja necessária a conversão (*cast*) de um tipo de dado primitivo em um objeto (empacotar o tipo primitivo).

Esta conversão não pode ser feita implicitamente (automática), sendo preciso o uso das classes referentes a cada tipo, que estão disponíveis na **java.lang**

<u>Classe</u>	<u>Tipo Primitivo</u>	
Byte	⇒ byte	 superclasse Number
Short	⇒ short	
<b>Integer</b>	⇒ int	
Long	⇒ long	
Float	⇒ float	
Double	⇒ double	

<u>Classe</u>	<u>Tipo Primitivo</u>
<b>Character</b>	⇒ char
Boolean	⇒ boolean
Void	⇒ void

# Classes Empacotadores (*wrappers*)

O uso destas classes permitem a referência a valores de tipos primitivos, o que não é possível para estes tipos de dados diretamente.

```
public static void main(String[] args) {  
    Integer preco = 10;  
    Float taxa = 0.25F;  
    System.out.print("Aumento= " + (preco * taxa));  
}
```

Por meio destas classes ainda são implementados métodos que manipulam, adequadamente, estes tipos primitivos, convertidos em objetos e vice-versa (objetos → tipos primitivos), sendo alguns deles relacionados a seguir:



# Classes Empacotadores (*wrappers*)

Para classe **Integer**:

<u>MÉTODO</u>	<u>FUNCIONALIDADE</u>
intValue()	retorna o valor do objeto Integer na forma de int
toString(int)	retorna objeto String representando o valor inteiro
parseInt(String)	retorna o valor inteiro referente a String
valueOf(String)	retorna um objeto Integer inicializado com o valor inteiro que será convertido a String (parâmetro)
:	:

Exemplo:

```
int idade = Integer.parseInt(str);    // str se torna int
```

Para cada um dos tipos primitivos existem métodos similares implementados em suas respectivas classes, como em:

```
double salario= Double.parseDouble(str); // str para double
```

⇒ Todos estes tipos possuem suas classes e métodos.

# Classes Empacotadores (*wrappers*)

Além de métodos estas classes também possuem alguns atributos, tais como:

MIN\_VALUE - menor valor de seu tipo

MAX\_VALUE - maior valor de seu tipo

```
int valor = Integer.MAX_VALUE; // atributo constante
System.out.print("Maior Inteiro = " + valor);
```

Exemplo de manipulações destas classes:

```
Integer anoCorrente = new Integer(2008); // mét. construtor
int ano = anoCorrente.intValue(); // converte Integer em int
```

```
String anos = "42"; // comum converter String em int
int idade = Integer.parseInt(anos);
```

```
String condicional = "false";
boolean condicao = Boolean.getBoolean(condicional);
```

# Classes Empacotadores (*wrappers*)

A partir do Java 5 foi implementado o *Autoboxing*, permitindo uma sintaxe mais elegante sobre os mesmos recurso. Isso possibilitou a seguinte codificação:

```
Integer anoCorrente = 2008;  
int ano = anoCorrente;
```

No entanto, é importante ressaltar que tipo primitivo e objeto não são a mesma coisa, mas a partir desta versão (Java 5) a codificação pôde ser facilitada, inclusive sobre a superclasse **Object**, passando um tipo primitivo para um método que recebe Object como argumento:

```
Object valor = 3;
```



# Exercícios de Fixação

- 1) Faça um programa orientado a objeto que armazene o ano, maior que 1900 e menor que o ano atual, e o nome completo do presidente brasileiro neste ano. Estes dados serão cadastrados enquanto o usuário quiser e indicarão anos onde movimentações nos astros por nossa galáxia foram expressivos, por exemplo: ano e presidente em que o cometa *Halley* passou e foi visto da Terra, eclipses, etc. Todos atributos de cada entidade em sua solução não poderão ser de tipos de dados primitivos, devendo ser usadas as classes empacotadoras. Toda interação com o usuário para coleta destes dados deverá acontecer de forma interativa e compreensível para o usuário, mas o relatório final será tabelar e mostrado na console com todos os registros feitos pelo usuário. Este programa só poderá ser encerrado se ao menos um cadastro válido for realizado. Use em sua solução todos os conteúdos estudados em Orientação a Objetos até o momento e use a parametrização para armazenar dados referenciáveis para diminuir a quantidade de conversões (*casting*) e tratando as classes de dados corretamente.



# Exercícios de Fixação

2) Faça um programa que permita o controle de uma coleção de quadros e de seus pintores para uma galeria internacional. Para cada **quadro** será fornecido um código de identificação (valor único que não repete e começa em 1), a identificação do pintor, preço (R\$) e ano de aquisição do quadro por esta galeria. Para cada **pintor** será cadastrado o nome, seu código de identificação pessoal (valor único ou **chave** para cada pintor iniciando também de 1) e o ano de nascimento do pintor, sempre maior que 1000 (definido na constante chamada ANOMINIMO). O programa apresentará um menu simples com as opções de funcionalidades que deverão estar disponíveis no programa para o usuário acionar até que escolha encerrar o programa:

- 1- Cadastra um novo Pintor;
- 2- Cadastra um novo Quadro;
- 3- Lista todos Quadros de um Pintor informado por qualquer parte de seu nome, mostrando ao final a soma total (R\$);
- 4- Mostra todos Quadros cadastrados até o momento, independente do Pintor;
- 0- Encerra o programa.

# Exercícios de Fixação

... continuar exercício 2

Identifique quais classes de dados serão necessárias nesta solução e despreze as diferenças entre letras maiúsculas e minúsculas na pesquisa do Pintor por nome (opção 3). Todos os dados de cada entidade (classe de dados) em sua solução não poderão ser de tipos de dados primitivos (usar as empacotadoras somente para os atributos das classes pois as empacotadoras não são usadas em variáveis de processamentos do programa).

Analise corretamente a solução apresentada que exige o relacionamento entre as entidades **Quadro** e **Pintor**, por meio dos seus códigos de identificação respectivos (chaves que nunca se repetem entre cada entidade específica). Por isso, sua solução não pode permitir que um quadro seja registrado antes do seu pintor,

pois não existiria o código chave do pintor para ser armazenado no objeto de dados do quadro. Utilize na sua solução a parametrização (*Generics*) para guardar os dados referenciáveis em coleções dinâmicas, diminuindo as conversões (*casting*) realizadas em JAVA.



# Herança

Em POO o termo **herança** faz referência a capacidade de derivação das classes já existentes na geração de novas classes que herdaram, direta ou indiretamente, os componentes (atributos e métodos) das primeiras classes.

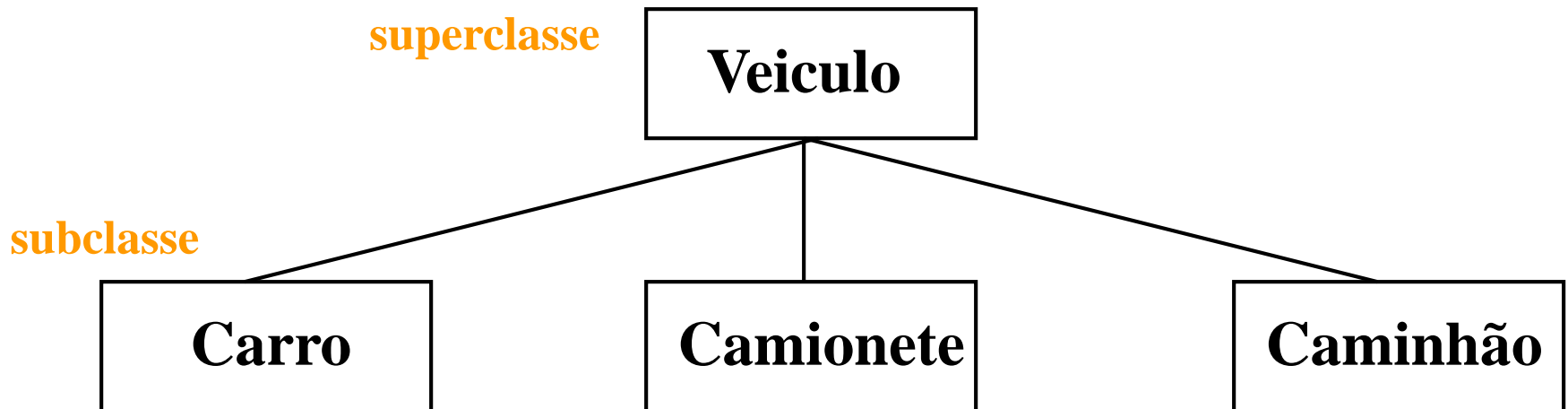


Esta organização na identificação e implementação das classes é extremamente relevante em POO, sendo a superclasse e a subclasse indicada por meio de sinônimos comuns, tais como:

- superclasse: base, progenitora, ancestral, pai, ...
- subclasse: filho, derivada, subtipo, ...

# Herança

Esta organização representa uma hierarquia existente entre as classes, como pode ser observada na representação gráfica:



# Herança

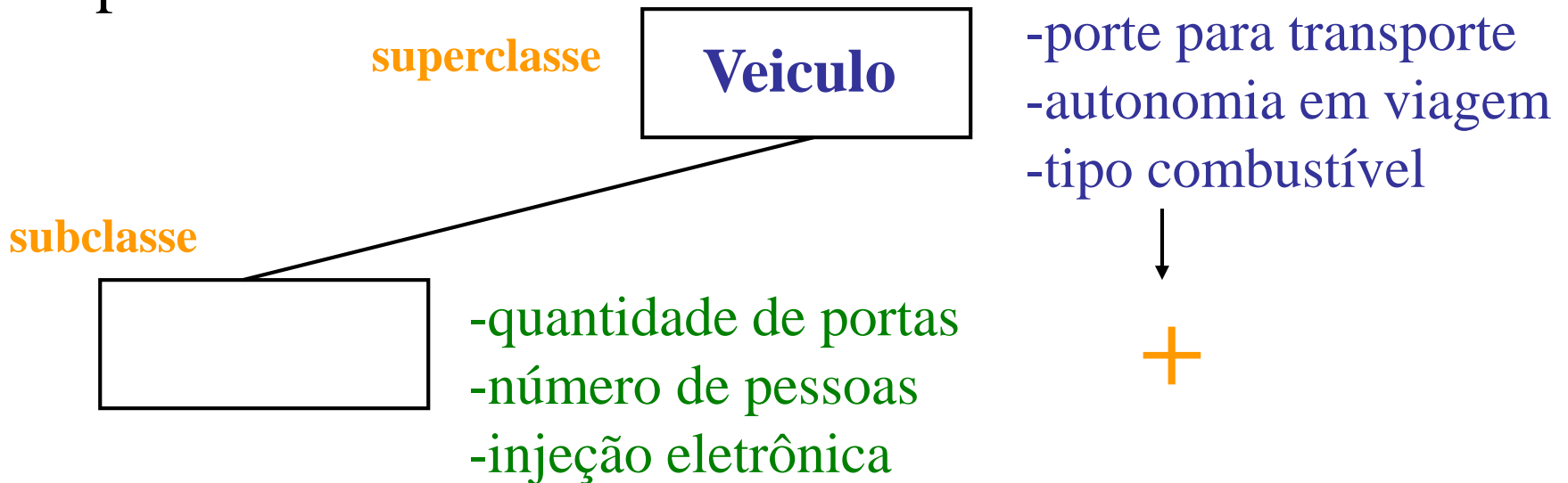
## Características da Herança

- Permite criar novas classes estendendo as classes já existentes
- As subclasses são **especializações** da superclasse e herdam todos os seus componentes (atributos e métodos)
- Cada subclasse pode criar novas características (atributos e métodos) além das provenientes de sua superclasse
- Os métodos da superclasse podem ser reescritos em suas subclasses (sobreposição)
- As subclasses podem ser superclasses para outros níveis na hierarquia das classes e na necessidade de manipulação de seus objetos
- Reaproveitamento e reuso de código já criado em outras classes progenitoras ou base



# Herança

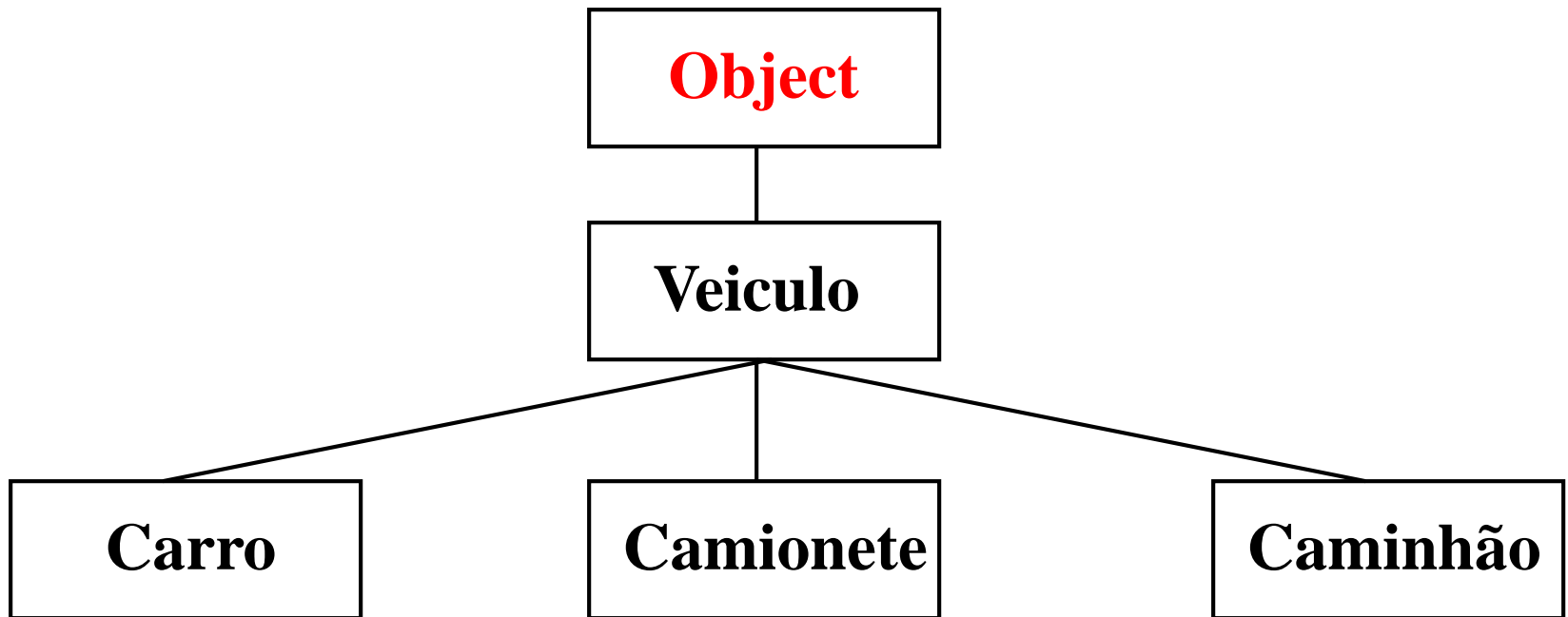
Estas características da herança envolvem a **especialização**, pois uma nova classe (subclasse) herda as características de uma outra (superclasse), podendo implementar partes específicas que não são contempladas na classe base, tornando-se mais especializada.



→ **Carro** (subclasse) consiste na especialização de **Veiculo**

# Herança

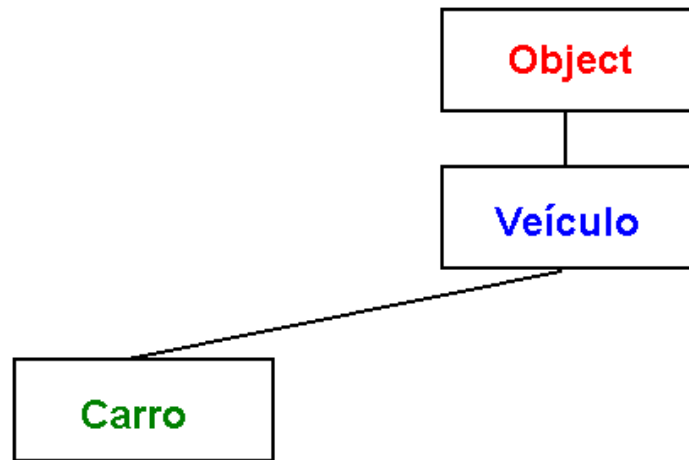
A implementação da herança na POO em Java consiste em uma característica natural, pois toda e qualquer classe elaborada é subclasse da superclasse **Object**, direta ou indiretamente.



# Herança

Observe na representação gráfica da hierarquia de herança anterior que:

- **Object** é uma classe base ou superclasse
- **Veículo** é subclasse **direta** da superclasse **Object**



- **Carro** é subclasse **indireta** de **Object**, mas subclasse **direta** de **Veículo**, que por sua vez é superclasse de **Carro**, apesar de **Veículo** ser subclasse **direta** de **Object**



# Herança

## Hierarquia de Herança

- Coleção de todas as classes que derivam de uma mesma superclasse comum.

## Sequência de Herança

- Percurso de uma classe específica até sua superclasse na hierarquia de herança.

## Propriedade da Transitividade

- Veiculo herda de Object
  - Carro herda de Veiculo
- ↓                      ↓
- Carro      herda de Object**

# Herança

Na linguagem Java a herança direta é realizada por meio da instrução **extends**, não sendo necessária a especificação da mesma para **Object**, que já é padrão.


```
/** Síntese
 * Métodos: frea(), acelera(), getMarca(),
 *          getVelocidade(), setMarca(String),
 *          setVelocidade(float)
 */
public class Veiculo {
    private String marca;
    private Float velocidade;

    public void frea() {
        if (velocidade > 0)
            velocidade--;
    }

    public void acelera() {
        if (velocidade <= 10)
            velocidade++;
    }
}
```

# Herança

```
// continuação do exemplo anterior
    public String getMarca() {
        return marca;
    }
    public void setMarca(String marca) {
        this.marca = marca;
    }
    public float getVelocidade() {
        return velocidade;
    }
    public void setVelocidade(float velocidade) {
        this.velocidade = velocidade;
    }
}
```




```
/** Síntese
 * Método: liga(), desliga(), getStatus(),
 *         setStatus(boolean)
 */
```

# Herança

// continuação do exemplo anterior

```
public class Carro extends Veiculo {  
    private Boolean status;  
  
    public void liga() {  
        status = true;  
    }  
  
    public void desliga() {  
        status = false;  
    }  
  
    public boolean getStatus() {  
        return status;  
    }  
  
    public void setStatus(boolean status) {  
        this.status = status;  
    }  
}
```



# Herança

```
/** Síntese
 *   Objetivo: criar um carro novo no sistema
 *   Entrada:  sem entrada (só atribuições)
 *   Saída:    registro do carro nvo
 */

public class RegistraCarro {
    public static void main(String[] args) {
        Carro auto1 = new Carro();
        // exemplo de inicialização no carro novo
        auto1.setMarca("FIAT");
        auto1.setVelocidade(0);
        auto1.setStatus(false);

        // Mostra carro novo
        System.out.println("Marca= " +
                           auto1.getMarca() );
        System.out.print("\tVelocidade= " +
                          auto1.getVelocidade() );
        System.out.println("\tSituação= " +
                            (auto1.getStatus() ? "ligado" : "desligado"));
    }
}
```

# Herança

Observando o programa anterior tem-se:

- A classe Veiculo possui 2 atributos (marca e velocidade) e 2 métodos (frea e acelera)
- Na classe Carro existe a necessidade de mais 1 atributo (status – situação de ligado ou não) e 2 métodos (liga e desliga)
- A instrução **extends** estende a classe Veiculo, criando uma nova classe (subclasse) Carro que junta os componentes de Veiculo e as necessidades da Carro
- Ao instanciar o **auto1** para **Carro** (ver no método **main**) ele possuirá todos os atributos de Veiculo e Carro (3) e todos os seus métodos (4)
- Qualquer alteração na superclasse irá refletir em suas subclasses, por exemplo criando o atributo combustível em Veiculo ela estará disponível para Carro

# Herança

## Por que SUPERclasse?

O uso da instrução **extends** estabelece a herança direta em Java, permitindo que subclasses sejam criadas por meio da incorporação de mais componentes (atributos e métodos) do que na classe base, especializando as novas classes derivadas.

Dessa forma, as subclasses possuem mais recursos (componentes) que suas superclasses, não sendo estas últimas (superclasses) tão SUPER assim.

Os prefixos **super** e **sub** são provenientes da linguagem matemática de conjuntos, onde o conjunto de todos os veículos contém o conjunto de todos os carros, sendo Veículo descrito como **superconjunto** do conjunto Carro, enquanto que Carro é **subconjunto** do conjunto de todos os veículos.



# Herança

## Instrução super

Todo construtor de uma subclasse precisa acionar o construtor de sua superclasse, podendo ser este acionamento

- **explícito**: uso da instrução **super** que aciona o construtor da superclasse
- **implícito**: não aciona, explicitamente, o construtor da superclasse que usará seu construtor padrão definido em Java

A instrução **super** possibilita herança de componentes da superclasse em suas subclasses, sem que tenham que ser redesenvolvidas em cada nova subclasse.

```
public Carro() {  
    super(); // aciona construtor da superclasse  
    :        // Veiculo  
}
```



# Herança

Geralmente, os métodos construtores são sobrecarregados, onde a instrução **super** pode possuir certa variação:

- sem parâmetros: inicia a superclasse com valores padrões
- com 1 parâmetro String: título de janela na Swing
- com parâmetros: inicia valores de instancia na superclasse

## Restrições da Instrução super

- Deve ser a primeira instrução em um método construtor
  - A instrução **this** e **super** não podem acontecer, simultaneamente, no mesmo construtor
- **super** – referencia os componentes da superclasse
- **this** – referencia os componentes de sua própria classe

# Herança

Observe também, no exemplo anterior, que os métodos definidos em **Veiculo** podem ser usados em **Carro**, sendo automática a disponibilização dos métodos da superclasse para suas subclasses.

Entretanto, pode ser necessária uma **nova definição** para um método herdado da superclasse, mas que tenha um tratamento especializado na subclasse e **oculte o método da superclasse** seguindo a hierarquia das classes.

Para isso é necessário **redefinir o método** no corpo da subclasse (**sobreposição**). Porém, este método não terá acesso aos atributos privados da superclasse, apesar de cada objeto da subclasse possuir seus próprios atributos herdados da superclasse. Essa alteração só será possível pela subclasse através do uso da interface adequada disponível pela superclasse (garantia de encapsular).

# Herança

Supondo a inclusão de um turbo na subclasse Carro, implemente o método `acelera()` que sobreponham o método da superclasse `Veiculo`.

```
/** Síntese
 * Método: liga(), desliga(), acelera(),
 *         getStatus(), setStatus(boolean)
 */
public class Carro extends Veiculo {
    private Boolean status;

    public void liga() {
        status = true;
    }

    public void desliga() {
        status = false;
    }

    public boolean getStatus() {
        return status;
    }
}
```

# Herança

```
// continuação do exemplo anterior

public void setStatus(boolean status) {
    this.status = status;
}

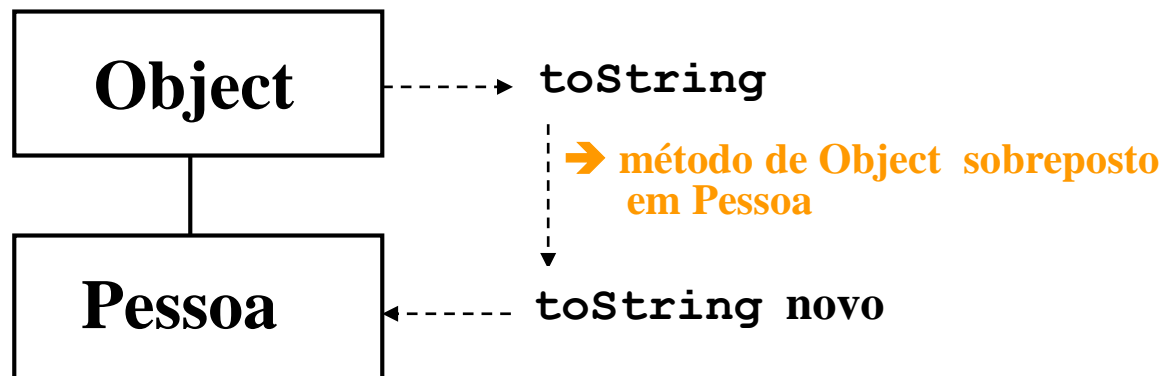
public void acelera() { // sobrepõem método de Veiculo
    final int turbo = 2; // potencia do turbo no carro
    float novaVelocidade = getVelocidade();
    if (getVelocidade() <= 10) {
        novaVelocidade += turbo;
        setVelocidade(novaVelocidade);
    }
}
}
```

Modifique também a classe executável RegistraCarro para acionar o método da subclasse Carro e observe como a aceleração é mais rápida quando usa o método com turbo, **ocultando** o mesmo método de Veiculo.

# Sobreposição de Métodos de *Object*

O recurso da sobreposição possibilita a especialização de métodos de qualquer classe que seja superclasse, direta ou indiretamente, de uma subclasse.

Sendo a classe **Object** superclasse de qualquer outra classe (é a classe “suprema”), seus métodos podem ser sobrepostos (redefinidos). Uma situação comum desta sobreposição ocorre com seu método **toString**, definido em **Object** e acionado sempre que uma instrução **print** for executada para mostrar dados de qualquer objeto.



# Sobreposição de Métodos de *Object*

## toString

Este método é definido na superclasse `Object`, sendo redefinido (sobreposto) por várias classes para mostrar a situação (valor dos atributos) de seus objetos.

- Pertence a classe suprema em Java - **Object**
- O acionamento da instrução `System.out.print` o aciona conforme redefinição feita em sua própria classe
- A sobreposição deste método geralmente mostra a situação do objeto que o está acionando, ou seja, faz a representação de todo objeto em uma string a ser mostrada
- Poder ser usado com uma `String` mutável com todos os valores contidos em seu objeto, inclusive para apresentação de vários objetos de uma mesma classe

# Janelas de Diálogo

## Janela de Dialogo

Geralmente, elaborar aplicativos necessita de comunicação entre o sistema e o usuário, em que algumas vezes o usuário fornece algum tipo de informação para que o sistema prossiga corretamente em sua execução.

Esse tipo de rápida interação é realizada através de caixas ou janelas de dialogo, sendo 4 abordadas neste material. A implementação da **Swing** fornece uma classe de fácil uso para este tipo de interação, chamada JOptionPane.

A JOptionPane possibilita uma variação de janelas de dialogo, por meio de métodos estáticos específicos, tais como:

**MessageDialog, ConfirmDialog, OptionDialog, InputDialog**

# Janelas de Diálogo

As características de cada uma das janelas de dialogo implementada pela **Swing** serão abordadas a seguir, porém uma importante propriedade, no ambiente de janelas (gráfico) consiste em:

- **modal** – não permite o acesso a outras janelas, enquanto esta permanecer em execução na aplicação. Exemplo: a necessidade da identificação do sexo do usuário para que o sistema possa prosseguir adequadamente em sua execução;
- **Não modal** (sem modo) – permite o acesso a janela não modal ou a outra janela qualquer em seu computador. Exemplo: janela como barra de ferramentas.

→ A janela de dialogo pode ser ou não modal, geralmente sendo modal, como será abordado neste material.



# Janelas de Diálogo

A partir de métodos da `JOptionPane` são acionadas 4 formas de diálogos diferentes, sendo cada uma destas janelas compostas por:

- a) Título na janela de dialogo;
- b) Mensagem destinada ao contato com usuário;
- c) Ícone representativo do tipo de mensagem:
  - 1. Informação;
  - 2. Pergunta;
  - 3. Alerta (ou advertência);
  - 4. Erro;
  - 5. Um outro definido pelo usuário.
- d) Um ou mais botões de interação com usuário.

→ Também é possível a ausência de qualquer ícone, sendo apresentada somente uma mensagem orientadora.

# Janelas de Diálogo

## MessageDialog





- Mostra somente uma informação para o usuário;
- Não retorna nenhum valor após sua utilização;
- Sua sintaxe geral consiste em:

`JOptionPane.showMessageDialog(Componete, Mensagem, Título da Mensagem, Tipo de Mensagem)`

- **Componete**: referente ao objeto do tipo contêiner que permite definir a posição na tela em que a janela de dialogo aparecerá. Normalmente seu valor é **null** para que a janela se apresente no centro da tela;
- **Mensagem**: mensagem que a janela mostrará ao usuário;
- **Título da Mensagem**: título que será mostrado na janela;
- **Tipo de Mensagem**: defini o ícone padrão que será apresentado na janela de dialogo, ou um outro ícone qualquer definido no sistema, ou ainda sem ícone algum.

# Janelas de Diálogo

Os ícones padrões são indicados pelas constantes:

- INFORMATION\_MESSAGE;  (informação)
- ERROR\_MESSAGE;  (Erro)
- WARNING\_MESSAGE;  (Advertência)
- QUESTION\_MESSAGE;  (Pergunta)
- PLAIN\_MESSAGE (sem ícone)

→ Para cada um dos tipos de dialogo existe também um método que permite trocar o ícone padrão por um outro ícone qualquer, definido pelo desenvolvedor do programa.

# Janelas de Diálogo

## ConfirmDialog

- Mostra uma mensagem e alguns botões pré-definidos (Yes, No, OK, Cancel), solicitando a confirmação do usuário;
  - Retorna um valor inteiro para identificação de qual botão foi pressionado (escolhido) pelo usuário;
  - Várias são as opções de indicação de botões para esta janela de dialogo, sendo as mais comumente usadas:
    - DEFAULT\_OPTION
    - YES\_NO\_OPTION
    - YES\_NO\_CANCEL\_OPTION
    - OK\_CANCEL\_OPTION
- :
- Conforme o botão pressionado é retornado um valor inteiro que inicia em zero, do botão mais a esquerda, e segue sequencialmente para 1 e 2 dos botões vindo para direita.

# Janelas de Diálogo

- Sua sintaxe geral consiste em:

`JOptionPane.showConfirmDialog`(**Componente**, **Mensagem**, **Título da Mensagem**, **Botões**, **Tipo de Mensagem**)

- **Componente**: referente ao objeto do tipo contêiner que permite definir a posição na tela em que a janela de dialogo aparecerá. Geralmente é deixado como **null** para janela ser apresentada no centro da tela;
- **Mensagem**: mensagem que a janela mostrará ao usuário;
- **Título da Mensagem**: título que será mostrado na janela;
- **Botões**: constante que defini os botões que estarão presente nesta janela de dialogo;
- **Tipo de Mensagem**: defini o ícone padrão que será apresentado na janela de dialogo, ou um outro ícone definido no sistema e ainda sem ícone algum.



# Janelas de Diálogo

## InputDialog

- Solicita uma informação ao usuário que deverá responder com um texto (String) na própria janela de dialogo;
- Retorna um valor String para ser utilizado;
- Sua sintaxe geral consiste em:

`JOptionPane.showInputDialog(Componete, Mensagem, Título da Mensagem, Tipo de Mensagem)`

- **Componete**: referente ao objeto do tipo contêiner que permite definir a posição da janela de dialogo na tela sendo comumente definido como **null** para esta janela ser mostrada no centro da tela;
- **Mensagem**: mensagem orientadora para o usuário;
- **Título da Mensagem**: título da janela de dialogo;
- **Tipo de Mensagem**: defini o ícone que será mostrado nesta janela, podendo ser os padrões, definido pelo usuário ou nenhum ícone (só a mensagem).

# Janelas de Diálogo

Algumas situações especiais são possíveis na interação com esta janela de dialogo, sendo elas:

- Pressionar o botão OK **sem** fornecer nenhuma resposta atribuirá vazio ("") na String de resposta;
- Pressionar o botão Cancel **sem** fornecer nenhuma resposta atribuirá nulo (**null**) na String de resposta;
- Pressionar o botão OK **com** alguma resposta atribui o valor informado na String indicada como resposta;
- Pressionar o botão Cancel **com** alguma resposta atribuirá nulo (**null**) na String de resposta;



# Janelas de Diálogo

## OptionDialog

- Janela de dialogo mais complexa de ser utilizada, pois é capaz de combinar todos os recursos destas outras janelas;
- Retorna um valor inteiro, conforme o botão escolhido;
- Sua sintaxe geral consiste em:

`JOptionPane.showOptionDialog(Componente, Mensagem, Título da Mensagem, Botões, Tipo de Mensagem, Ícone, Array de objetos, Seleção padrão)`





# Janelas de Diálogo

## Parâmetros da JOptionPane.showOptionDialog

- **Componente**: referente ao objeto do tipo contêiner, que sendo **null** posicionará esta janela no centro da tela;
- **Mensagem**: mensagem orientadora para o usuário;
- **Título da Mensagem**: título da janela de dialogo;
- **Botões**: usa os botões padrões ou novos botões;
- **Tipo de Mensagem**: umas das mensagens padrões que possibilitam indicar o ícone padrão em outras janelas;
- **Ícone**: objeto **ImageIcon** que permite a inclusão de outro ícone diferente do padrão, ou **null** para ficar sem ícone;
- **Array objetos**: array que indicará as possíveis escolhas nesta caixa de diálogo, caso YES\_NO\_OPTION, OK\_CANCEL\_OPTION, ... não estejam sendo usados;
- **Seleção padrão**: objeto de padrão inicial selecionado, quando as constantes padrões não estiverem em uso.

# Janelas de Diálogo

```
/** Síntese
 *   Objetivo: Identificar a faixa de idade do usuário
 *   Entrada:  faixa de idade
 *   Saída:    confirmação da faixa de idade indicada
 */
import javax.swing.*;
public class OpcaoDialogo {
    public static void main(String[] args) {
        String [] escolha = {"Entre 1 e 20 anos",
                             "Entre 21 e 40 anos", "Mais de 40 anos"};
        String idade = null;
        try {
            int resposta = JOptionPane.showOptionDialog(null,
                                                         "Qual sua idade?", "Idade", 0,
                                                         JOptionPane.INFORMATION_MESSAGE,
                                                         null, escolha, escolha[0]);
            idade = (escolha[resposta]);
            System.out.print("Faixa de Idade = " + idade);
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Programa encerrado.");
        }
    }
}
```

# Janelas de Diálogo

- Para cada uma destas janelas de dialogo existe um método que permite definir um outro ícone qualquer;
  - A organização da mensagem de orientação a ser apresentada por uma janela de dialogo pode ser melhor elaborado com saltos de linha '\n';
  - Apesar da facilidade de uso destas janelas de interação, seus tamanhos são limitados para manipulação de vários componentes gráficos, sendo as mesmas empregadas somente na atividade de “dialogo” ágil com os usuários.
- O estudo de outros recursos gráficos possíveis pela classe Swing e AWT será efetivado mais a frente em nossa disciplina, onde vários componentes serão usados.

# Janelas de Diálogo

Todas estas opções podem parecer confusas para interagir com os usuários, mas a prática mostrará que esta aparência está errada. Siga os passos para elaborar uma interação adequada e agradável com seu usuário.

1. Escolha o tipo de diálogo (mensagem, confirmação, entrada ou opção)
2. Selecione um ícone (informação, erro, advertência, pergunta, nenhum ou um outro qualquer personalizado)
3. Reforce o intuito do diálogo no título coerente desta janela
4. No diálogo de confirmação defina as opções possíveis (*Yes/No*, *Yes/No/Cancel*, *Ok/Cancel*, padrão, ...);
5. Para janela de opções defina os componentes desejados para interação ágil e adequada
6. Seja direto na janela de entrada para que usuário forneça o retorno esperado pelo programa
7. Encontre o método apropriado da `JOptionPane` a ser chamado

# Sobreposição de Métodos de *Object*

```
/* Síntese
 * Objetivo: cadastrar grupo de pessoas
 * Entrada: nome e idade das pessoas
 * Saída:   listar todas pessoas cadastradas
 */
import java.util.*;
import javax.swing.JOptionPane;
public class Principal {
    public static void main(String[] args)
    { ArrayList<Pessoa> pessoa = new ArrayList<Pessoa>();
      String nomeAux;
      int idadeAux = 0, aux = 0;
      boolean erro;
      do {
          nomeAux = JOptionPane.showInputDialog(null,
            "Pessoa "+(aux+1)+"\nDigite seu primeiro nome.",
            "Cadastros", JOptionPane.QUESTION_MESSAGE);
          if(nomeAux.length() != 0) {
              do {
                  erro = false;
                  try {
                      idadeAux = Integer.parseInt(
                        JOptionPane.showInputDialog(null,
                          "Pessoa "+(aux+1)+"\nDigite sua idade em anos.",
                          "Cadastros", JOptionPane.QUESTION_MESSAGE));
```

# Sobreposição de Métodos de *Object*

// continuação do exemplo anterior

```
        if(idadeAux <= 0) {
            JOptionPane.showMessageDialog(null,
            "Valor inválido. Informe um número positivo.",
            "Erro", JOptionPane.ERROR_MESSAGE);
            erro = true;
        }
    } catch (NumberFormatException ex) {
        erro = true;
        JOptionPane.showMessageDialog(null,
        "Valor inválido. Informe um número em anos.",
        "Erro", JOptionPane.ERROR_MESSAGE);
    }
} while (erro);
pessoa.add(new Pessoa(nomeAux, idadeAux));
aux++;
}
} while(nomeAux.length() != 0);
saltaLinha(10);
```

# Sobreposição de Métodos de *Object*

// continuação do exemplo anterior

```
        System.out.println("REGISTROS\n\nNOME\tIDADE");
        System.out.println("=====");
        for(aux = 0; aux < pessoa.size(); aux++)
            System.out.println(pessoa.get(aux));
    }
    public static void saltaLinha(int quantidade) {
        for(int aux = 0; aux < quantidade; aux++)
            System.out.println();
    }
}
```



```
/* Síntese
 *   Atributos: nome, idade
 *   Métodos:  getNome(), getIdade(), setNome(String),
 *             setIdade(int), toString(Pessoa)
 */
public class Pessoa {
    // atributos
    private String nome;
    private Integer idade;
```

# Sobreposição de Métodos de *Object*

```
// continuação do exemplo anterior

// método Construtor
Pessoa(String nomePar, int idadePar)
{ this.setNome(nomePar);
  this.setIdade(idadePar);
}

// métodos para encapsulamento
public void setNome(String nome) {
    this.nome = nome;    }
public String getNome() {
    return nome;        }
public void setIdade(int idade) {
    this.idade = idade;  }
public int getIdade() {
    return idade;        }

// método sobrescrito
public String toString()
{
    return(this.getNome() + "\t" + this.getIdade());
}
}
```



# Herança

## Restringir a Herança

Além da possibilidade de ocultar um método da superclasse, através da redefinição em sua subclasse, ainda é possível evitar a herança em Classes e Métodos.

O uso da instrução **final** garante que uma classe não pode ser superclasse ou progenitora de outras classes. Por exemplo: suponha que a classe Camionete seja final.

```
public final class Camionete {  
    :    // definição dos componentes desta classe  
}
```

Todos os métodos de uma classe final, automaticamente, são finais, e não permitem sua redefinição em suas classes derivadas (sobreposição).

# Herança

A necessidade de uma lógica em um projeto pode precisar da implementação de somente alguns métodos que não possam ser sobrepostos por suas subclasses. Para isso a palavra reservada **final** é usada na assinatura do método, evitando que ele seja redefinido. Por exemplo:

```
public final String getCombustivel() {  
    :    // definição do corpo do método de Veiculo  
}
```

Isso oferece segurança quanto a não permitir que um mesmo método, com mesma assinatura, possa ser implementado em uma subclasse com processamento inadequado (classe **String** em Java é **final**).

→ **Assinatura** de um método não compreende seu retorno.

# Herança

## Atenção com os Qualificadores

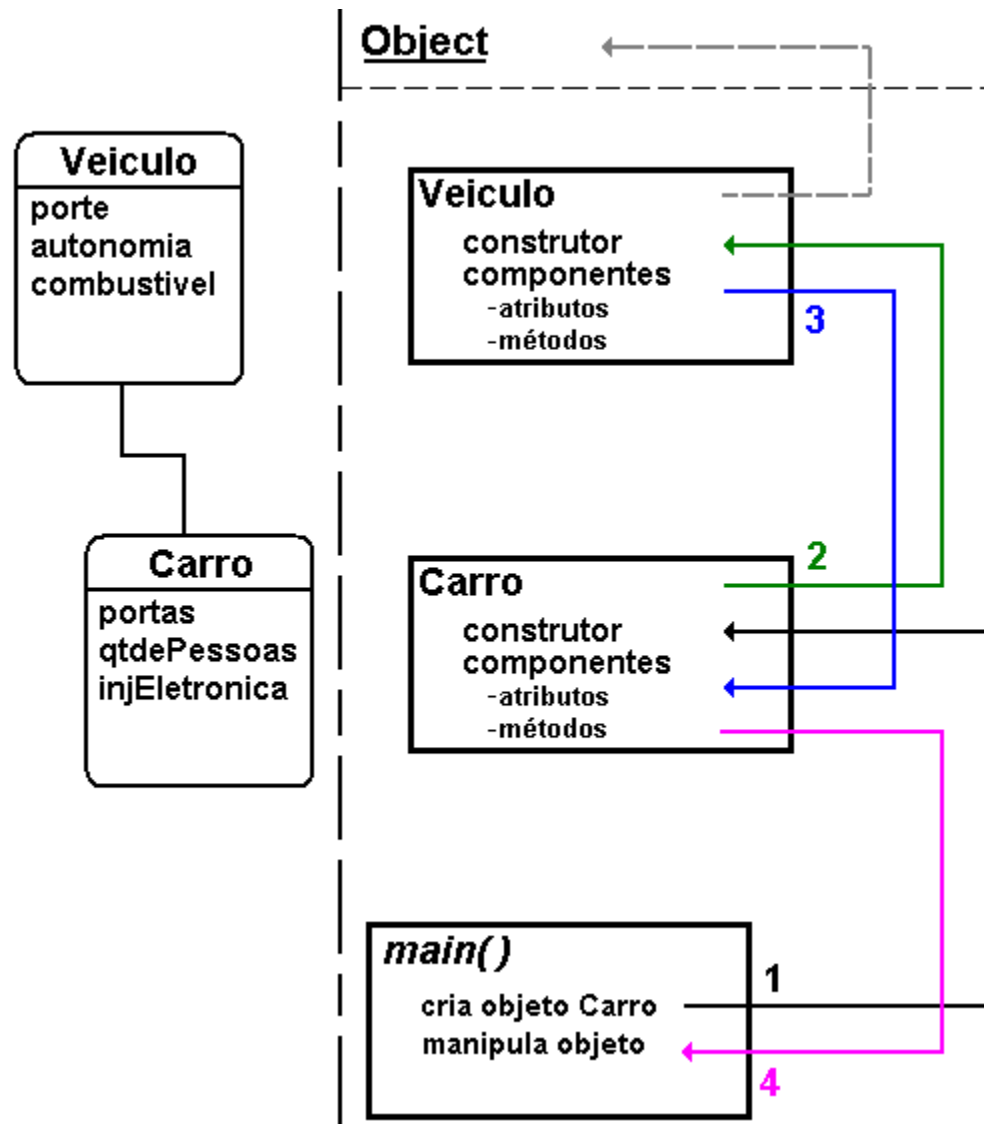
Um aspecto importante a ser analisado é a definição coerente dos qualificadores de acesso nas classes envolvidas em um projeto e adequadamente definidas em conformidade com estas novas características fundamentais na POO.

Assim como os componentes privados abordados no estudo do encapsulamento, agora também é relevante uma análise precisa com a herança, por exemplo:

- **private** permite acesso somente da própria classe, seja ela uma superclasse ou subclasse
- **protected** restringi acesso de qualquer classe, mas permite que este acesso aconteça somente pela própria classe e suas subclasses
- demais qualificadores permitem acesso das subclasses aos componentes da superclasse, mas de outras classes

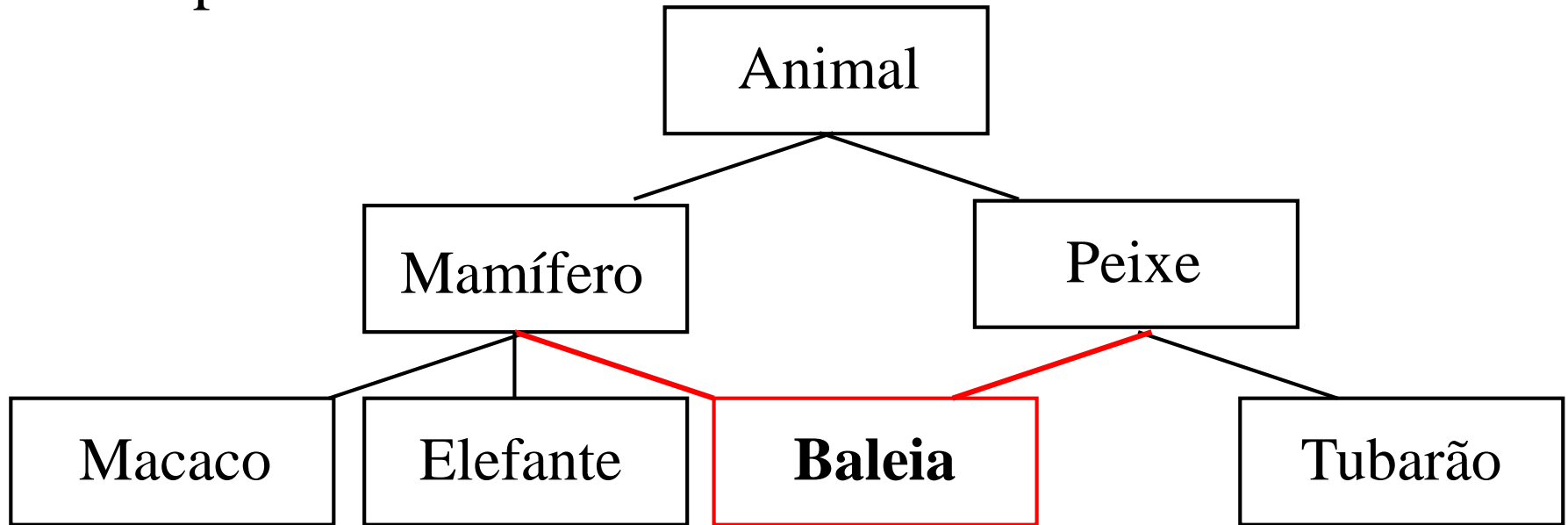
# Herança

## Representação do Processamento com Herança



# Herança

Dentre as várias possibilidades com a herança, a linguagem Java não aceita a implementação da **herança múltipla**, ou seja, uma nova classe derivada ser gerada por mais que uma progenitora (superclasse), por exemplo:



→ **Baleia** não pode ser implementada porque possui duas heranças diretas das superclasses **Mamífero** e **Peixe**.

# Herança

## Composição e Herança

A composição em uma classe ocorre quando esta possui como atributo um objeto de outra classe, enquanto que a herança identifica que a classe derivada consiste em uma especialização de sua superclasse.

Por exemplo:

A classe Carro possuirá como um de seus atributos um objeto da classe Pneu, sendo geralmente 4 elementos de pneu que são usados por um Carro. Mas é importante observar que **Carro não é Pneu** e por isso não pode estender o mesmo ou vice-versa (Pneu não é Carro e não pode estendê-la).

// objeto **pneus** compõe a classe **Carro**

```
public class Carro {  
    Pneu [ ] pneus = new Pneu[4];  
    :  
}
```

// classe **Carro** é derivada de **Pneu**

```
public class Carro extends Pneu {  
    :  
}
```

**situação real incorreta**

# Herança

Uma forma de identificar a herança e constatar se ela ocorre realmente é questionar:

A “minha classe” **possui** esta outra classe, ou ela é esta outra classe?

Outra forma de verificar se a herança é adequada seria averiguar se uma instancia da subclasse pode ser usada como um objeto instanciado da superclasse.

~~O Carro é um Pneu?~~

A Camionete é um Veiculo?

Um Carro **possui** Pneu?

~~Uma Camionete **possui** Veiculo?~~



# Herança

É importante destacar que o processo de derivação de uma classe, gerando suas subclasses, pode acontecer até que a abstração necessária para solução de um problema seja atingida. As características pertinentes a estas subclasses não teriam nada a ver com as suas classes irmãs (subclasses em um mesmo nível).

Por exemplo:

A geração das subclasses **Carro**, **Camionete** e **Caminhão**, da superclasse **Veiculo**, não possuem características especializadas que as relacione diretamente, não tendo nada a ver uma com a outra.

Apesar desta constatação, é possível que estas subclasses sejam usadas como sua superclasse na:

- atribuição da subclasse em variável da superclasse
- passagem de parâmetro da subclasse para o método que espera a superclasse como parâmetro



# Herança

## Vantagens com a Herança

- Inclusão de novas classes (subclasses) na hierarquia, porém sem qualquer necessidade de alteração no código, pois cada classe define suas próprias características
- Reaproveitamento do código criado em uma classe e usado em outra(s) classe(s) derivada(s) (subclasse)
- Facilidade na manutenção, que se realizará sobre a classe que possui tal implementação (componentes) e não em vários pontos do código que duplicariam a implementação desta lógica por vários objetos
- Incorporação de características especializadas as necessidades de novas classes e seus objetos (especialização das classes existentes)
- Promover a ALTA Coesão e o BAIXO Acoplamento

envolve recursos bem definidos e específicos (bem coeso) a um tratamento computacional

relação ou dependência de outra(s) classe(s)

# Herança

## Propriedades de Acoplamento e de Coesão

O **baixo acoplamento**, frequentemente, corresponde a um sinal do software elaborado ser bem estruturado e possuir bom design.

A combinação com a **alta coesão** fornece suporte aos objetivos gerais de maior legibilidade e facilidade de manutenção no software.

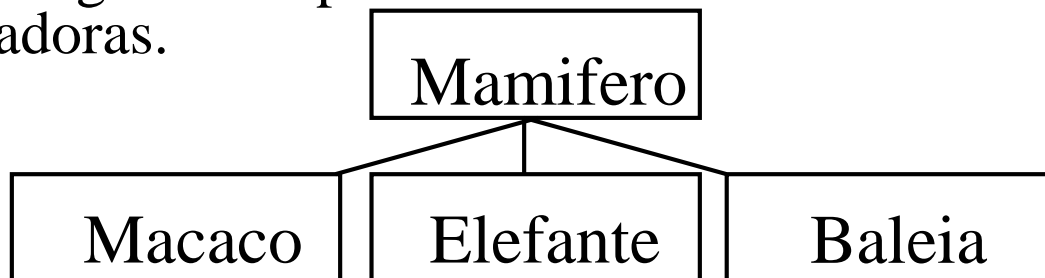
Assim, se é desejado para um software o

***baixo acoplamento com alta coesão***  
(e vice-versa)



# Exercícios de Fixação

- 3) Observando o digrama que representa a hierarquia de heranças abaixo, elabore uma implementação adequada para as classes Mamífero, Macaco, Elefante, Baleia. Implemente na superclasse os componentes (atributos e métodos) necessários para o acompanhamento da idade geral de amamentação materna e de vida do animal (ambos em anos), descrição de sua espécie e o tamanho normal de um na fase adulta (valor em metros), usando ambiente gráfico para leitura destes dados e classes empacotadoras.



Nos macacos indique o porte (pequeno, médio, grande), no elefante e baleia guarde o peso e somente para o elefante também armazene a descrição de seu habitat natural. Implemente os métodos necessários para cadastrar qualquer um destes animais enquanto o usuário quiser e não superar os 500 registros de animais. Quando o usuário não quiser mais fazer registros apresente todos os animais cadastrados como um relatório tabelar na console e encerre o programa.

# Exercícios de Fixação

- 4) Uma empresa contrata pessoas registrando seu nome, CPF e data de nascimento, onde todas recebem o mesmo piso salarial de R\$232,00. Elabore um programa que permita o cadastramento de várias pessoas como funcionário regular, prestação de serviços ou gerencia de equipe. Implemente para estas 3 possíveis subclasses derivadas de pessoa um método que sobreponha o método da superclasse *calculaSalario* que simplesmente atribui o piso salarial a cada pessoa contratada. Para funcionário regular o método *calculaSalario* deverá fornecer o piso salarial acrescido de 10%, enquanto que a prestação de serviço será calculado o pagamento através da quantidade de horas trabalhadas multiplicada por dois acrescido do próprio piso salarial. Para gerencia de equipe o salário a ser pago será obtido pela quantidade de projetos vezes 50% do piso salarial acrescido do próprio piso. Após o usuário encerrar o cadastro apresente um menu que possibilite ao usuário ter acesso ao total de funcionários cadastrados em cada uma das três categorias e o total salarial a ser pago para mesma. Permaneça neste menu até que o usuário escolha a opção que encerre o programa. Use somente classes empacotadoras nos atributos das classes e colete todos os dados de entrada por janelas gráficas de diálogo.

# Referência de Criação e Apoio ao Estudo

## Material para Consulta e Apoio ao Conteúdo

- CAMARÃO, C., FIGUEIREDO, L., Programação de Computadores em Java, Rio de Janeiro: LTC, 2003.
  - Capítulo 13
- HORSTMANN, C. S., CORNELL, G., Core Java2 , volume 1, Makron Books, 2001.
  - Capítulo 5 e 9
- FURGERI, S., Java 2: Ensino Didático: Desenvolvendo e Implementando Aplicações, São Paulo: Érica, 2002.
  - Capítulo 7 e 9
- Universidade de Brasília (UnB Gama)
  - <https://cae.ucb.br/conteudo/unbfga>  
(escolha a disciplina **Orientação a Objetos** no menu superior)