Análise e Projeto de Desenvolvimento de Sistemas

APDS

Aula 4

Universidade Federal do Rio Grande do Norte
Unidade Acadêmica Especializada em Ciências Agrárias
Escola Agrícola de Jundiaí
Técnico em Informática
Profa. Alessandra Mendes

Revisando...

Revisando Objetos

- Um **objeto**
  - É uma entidade do mundo real que tem uma identidade e, por este motivo, será sempre distinto e outro objeto mesmo que eles apresentem exatamente as mesmas características.
  - Pode representar uma entidade concreta (elevador, bicicleta) ou conceitual (estratégia de jogo, conta bancária).
  - É uma instância de uma classe – que determina qual informação o objeto contém e como ele pode manipulá-la.

Desta forma...

Um programa desenvolvido com uma linguagem de programação orientada a objetos manipula estruturas de dados através dos ***objetos*** *da mesma forma que um programa em linguagem estruturada utiliza* ***variáveis****.*

Revisando Classes

- Uma **classe**
  - É uma é uma estrutura que abstrai um conjunto de objetos com características similares.
  - Define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos, ou seja, descreve os serviços providos por seus objetos e quais informações eles podem armazenar.
  - É uma fôrma capaz de produzir objetos.
  - É responsável pela criação de seus objetos via método construtor (chamado pelo *new*)
    - Mesmo nome da classe
    - Sem tipo de retorno

Revisando Classes

- Uma **classe**

```
public class Carro {
 private int velocidade;
 public Carro(int velocidadeInicial) {
  velocidade = velocidadeInicial;
 }
 public void acelera() {
  velocidade++;
 }
 public void freia() {
  velocidade--;
 }
}
```

Os atributos (características) são variáveis globais acessíveis por todos os métodos da classe.

O construtor só é executado uma vez quando o objeto é instanciado.

Os métodos são os comportamentos.

Criação de objetos

- Os objetos devem ser instanciados antes de serem utilizados
  - *new* instancia um objeto, chamando o seu construtor
  - O valor *null* é utilizado para representar um objeto não inicializado ou para descartar um objeto previamente instanciado

```
Carro fusca = new Carro(10);
Carro bmw = new Carro(15);
fusca.freia();
bmw.acelera();
fusca = bmw;
fusca.acelera();
fusca = null;
```

Qual a velocidade de cada carro?

O que acontece aqui?

O que acontece aqui?

Pacotes

- Utilizados para agregar classes relacionadas
  - O pacote é indicado na primeira linha da classe
  - Declaração *package*
- Se uma classe não declara seu pacote, o interpretador assume que a classe pertence a um pacote *default*
  - Modificadores de acesso (*protected* ou *acesso de pacote*) permitem que determinadas classes sejam visíveis apenas para outras classes do mesmo pacote.

```
package exemplos;

public class Carro {
  ...
}
```

Pacotes

- Sempre que for usar uma classe de outro pacote, é necessário importar utilizando a palavra-chave *import* seguida do nome da classe desejada.
  - As importações são apresentadas antes da declaração da classe mas depois da declaração do pacote .
  - A importação de um pacote não importa os subpacotes recursivamente.

```
package exemplos;

import java.util.Scanner;

public class Carro {
 ...
}
```

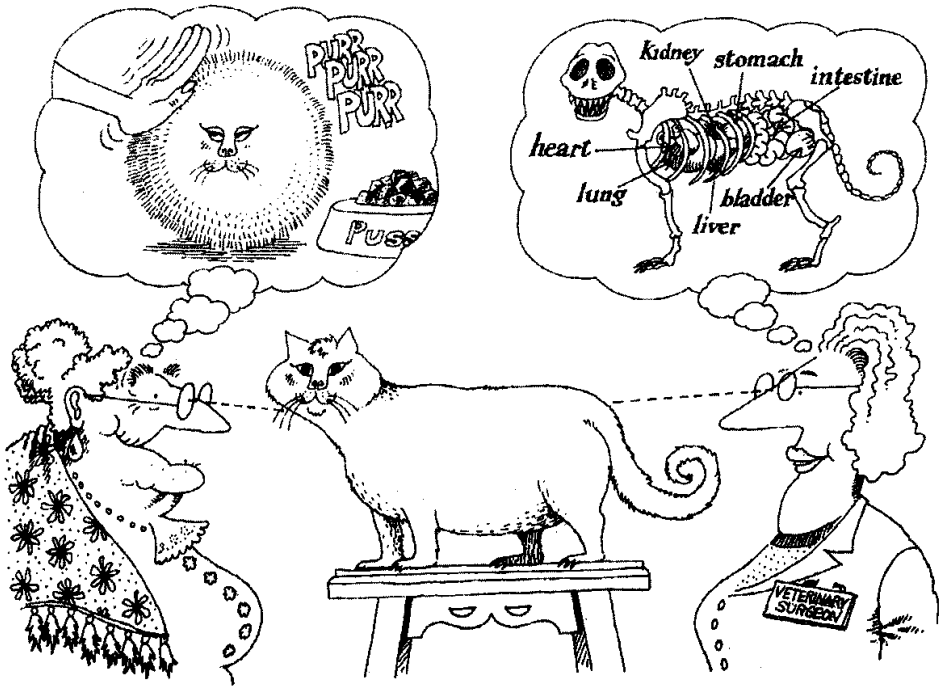
Paradigma OO - Conceitos

Profa. Alessandra Mendes – UFRN/EAJ/TI/APDS

Princípios do Paradigma OO

Abstração

- A **representação** computacional do objeto real deve se concentrar nas **características** que são **relevantes** para o problema.



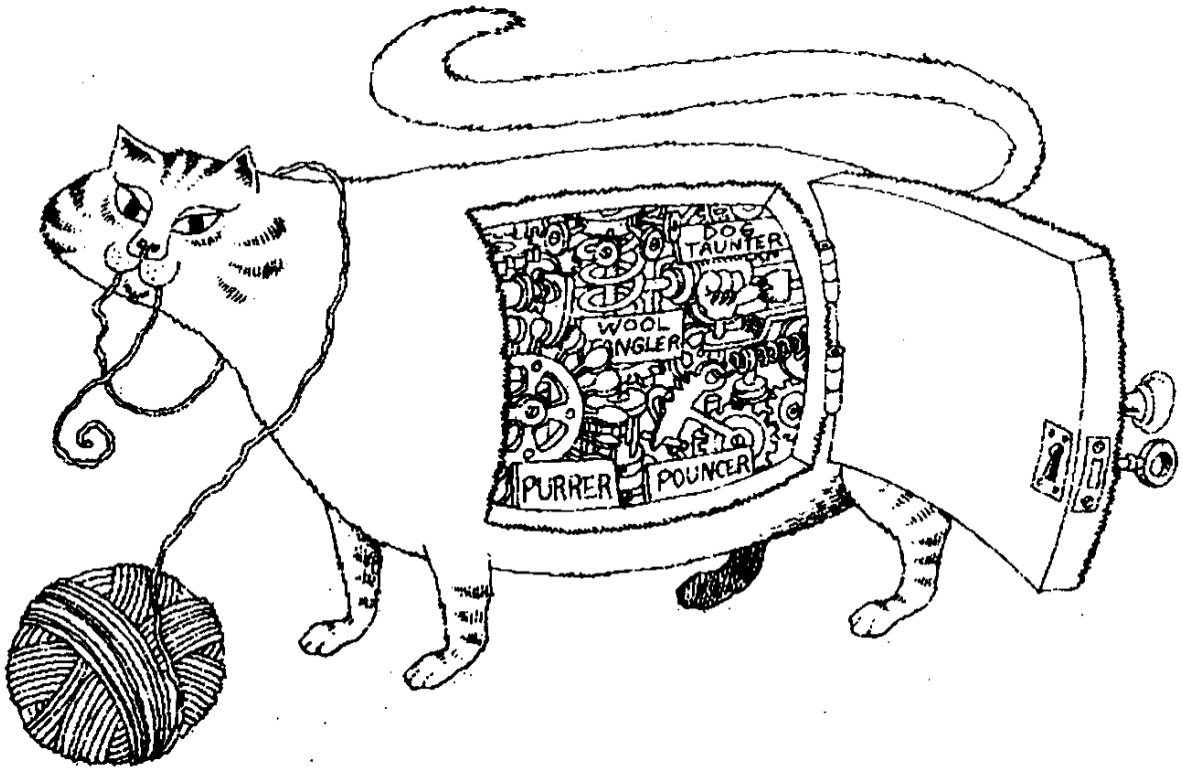
Fonte: livro "Object-Oriented Analysis and Design with Applications"

Abstração

- São criados somente os atributos e métodos necessários para o problema que está sendo analisado.
- Exemplo: quais seriam os atributos e métodos para o objeto Carro em cada uma das situações seguintes?
  - Sistema de uma locadora de carros
  - Sistema de uma revendedora de carros
  - Sistema de uma oficina mecânica
  - Sistema do DETRAN

Encapsulamento

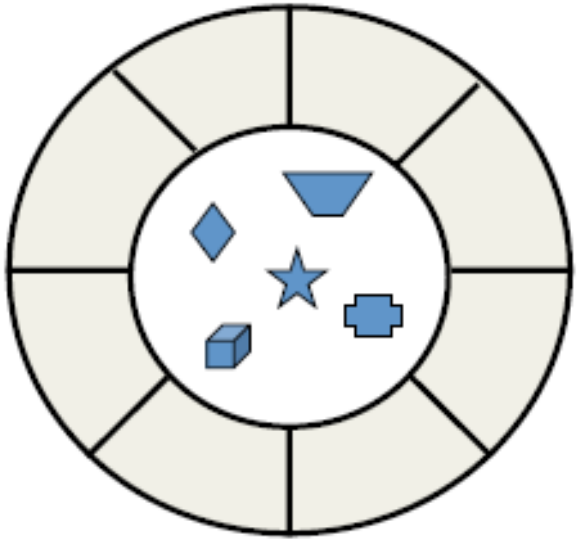
- O objeto deve **esconder** seus dados e os detalhes de sua implementação.



Fonte: livro “Object-Oriented Analysis and Design with Applications”

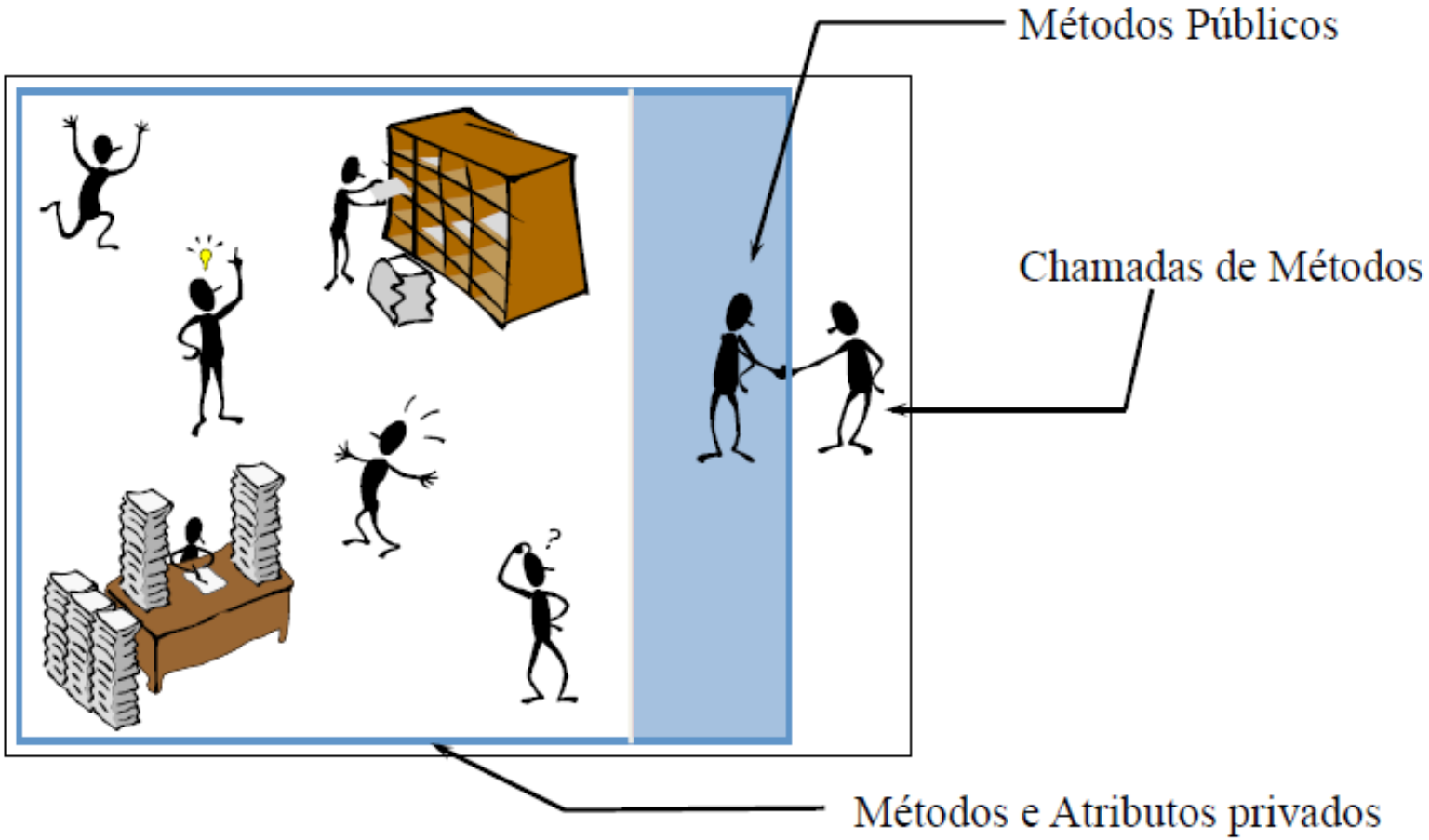
Encapsulamento

- Atributos e Métodos
  - Os métodos formam uma "cerca" em torno dos atributos;
  - Os atributos não devem ser manipulados diretamente;
  - Os atributos somente devem ser alterados ou consultados através dos métodos do objeto.



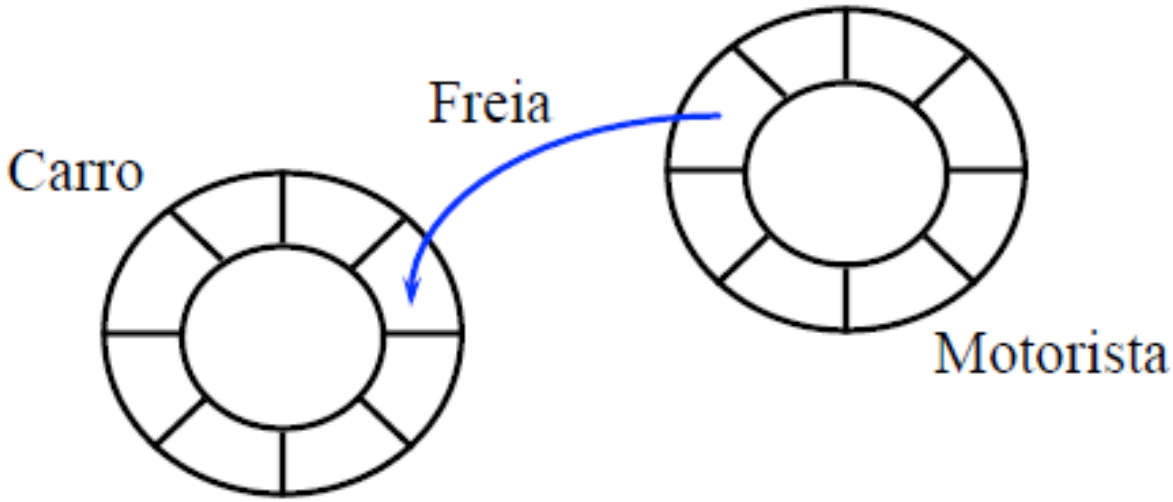
Encapsulamento

- Modificadores de acesso



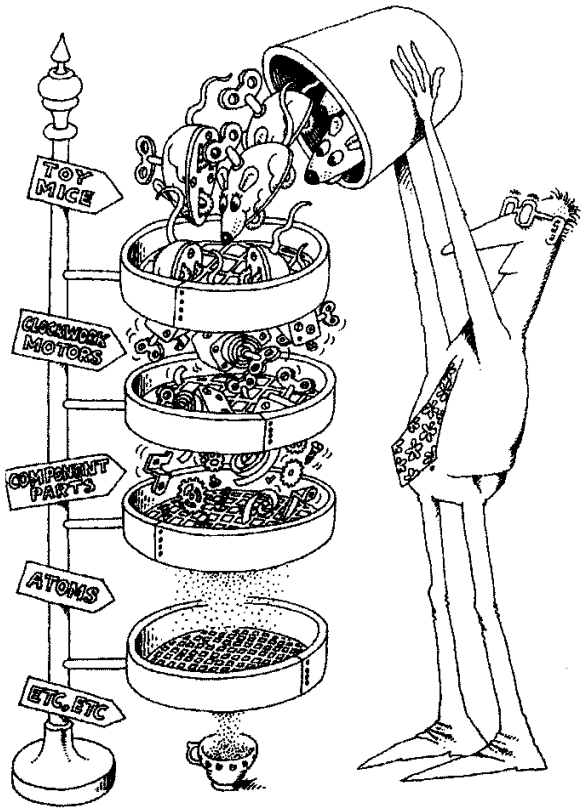
“Responsabilidade”

- Um programa OO é um conjunto de objetos que colaboram entre si para a solução de um problema.
  - Objetos colaboram através de chamadas de métodos uns dos outros.
  - Onde cada método deve ser implementado?



Hierarquia

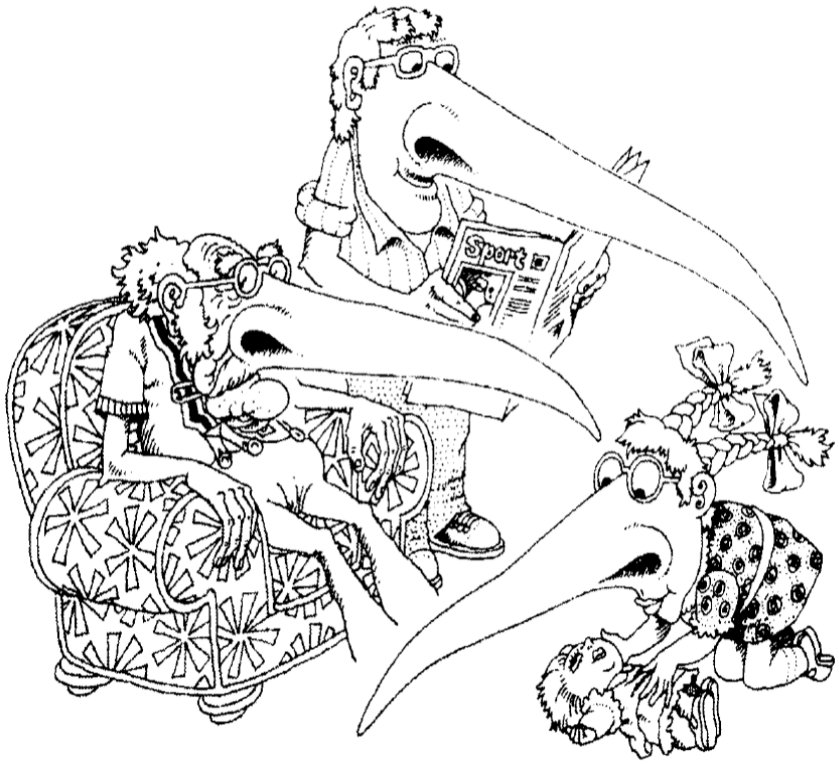
- Os objetos devem ser organizados no sistema de forma hierárquica.



Fonte: livro “Object-Oriented Analysis and Design with Applications”

Hierarquia

- Objetos herdam atributos e métodos dos seus ancestrais na hierarquia.

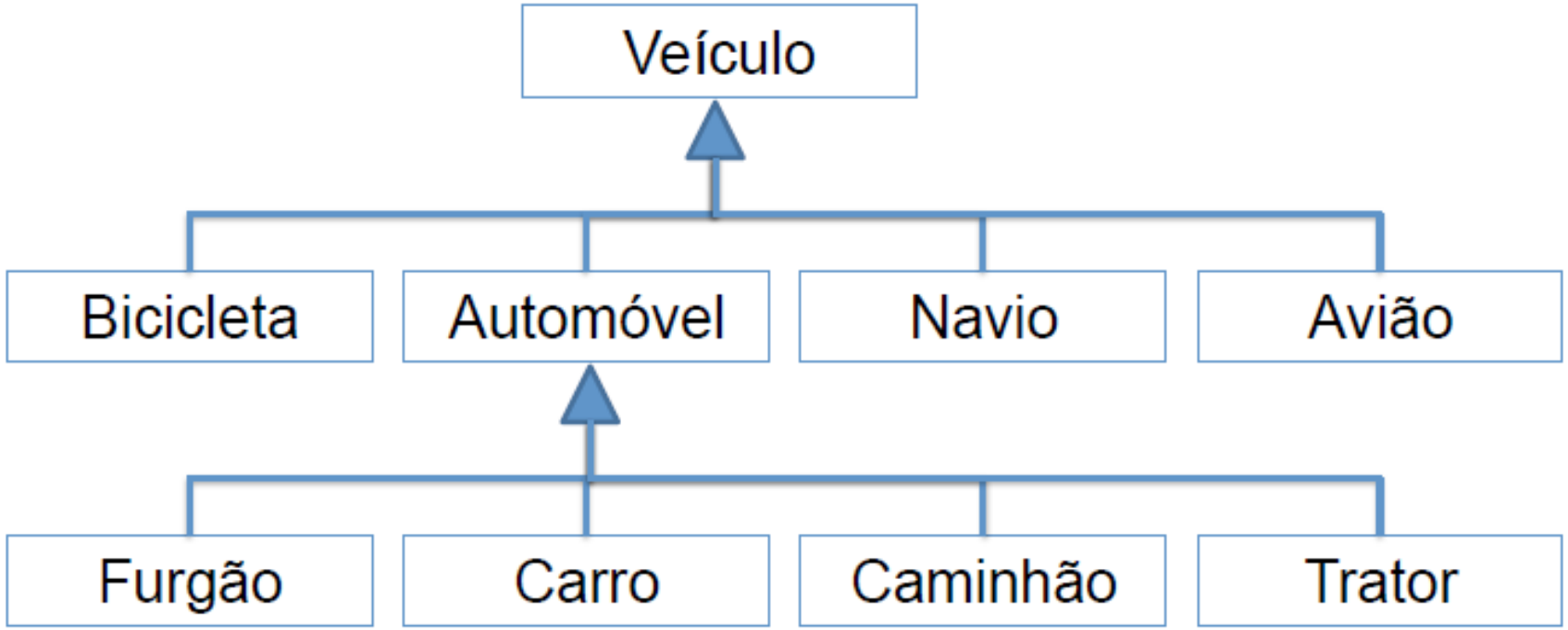


Fonte: livro “Object-Oriented Analysis and Design with Applications”

Herança

- Para viabilizar a hierarquia entre objetos, as classes são organizadas em estruturas hierárquicas.
  - A classe que forneceu os elementos herdados é chamada de superclasse;
  - A classe herdeira é chamada de subclasse;
  - A subclasse pode herdar os métodos e atributos de suas superclasses;
  - A subclasse pode definir novos atributos e métodos específicos;
  - As subclasses são mais especializadas do que as suas superclasses, mais genéricas.

Exemplo de Herança



Teste da Leitura: "*subclasse* é **um** *superclasse*"
Ex.: Carro é um Automóvel; Trator é um Veículo; ...

Exemplo de Herança

- Relembrando a classe Carro...

```
public class Carro {
 private int velocidade;
 public Carro(int velocidadeInicial) {
  velocidade = velocidadeInicial;
 }
 public void acelera() {
  velocidade++;
 }
 public void freia() {
  velocidade--;
 }
}
```

Exemplo de Herança

- Criando um carro inteligente...

```
public class CarroInteligente extends Carro {
	public CarroInteligente(int velocidadeInicial) {
		super(velocidadeInicial);
	}
	public void estaciona() {
		// código mágico para estacionar sozinho
	}
}
```

- Usando um carro inteligente...

```
CarroInteligente tiguan = new CarroInteligente(10);
for (int i = 10; i > 0; i--) {
	tiguan.freia();
}
tiguan.estaciona();
```

De onde veio isso? :o

Exemplo de Herança

- Criando um carro de corrida...

```
public class CarroCorrida extends Carro {
	public CarroCorrida(int velocidadeInicial) {
		super(velocidadeInicial);
	}
	public void acelera() {
		for(int i = 1; i <=5; i++)
			super.acelera();
	}
}
```

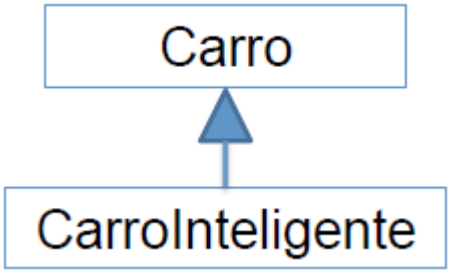
- Usando um carro de corrida...

```
CarroCorrida f1 = new CarroCorrida(10);
f1.acelera();
```

Qual a velocidade agora: :/

Compatibilidade

- Qualquer subclasse é compatível com a sua superclasse. Contudo, a reciproca não é verdadeira...




```
Carro c = new CarroInteligente(20);
c.acelera();
c.freia();
```

✓

```
CarroInteligente c = new Carro(20);
c.acelera();
c.freia();
c.estaciona();
```

✗

Herança em Java

- Uma classe só pode herdar de uma outra classe (herança simples).
- Caso não seja declarada herança, a classe herda da classe *Object*.
  - Ela define o método *toString*(), que retorna a representação em *String* do objeto.
  - Qualquer subclasse pode sobrescrever o método *toString*() para retornar o que ela deseja.
  - Veja os demais métodos da classe *Object* em https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Polimorfismo

- Uma subclasse pode redefinir (sobrescrever) um método herdado. Este mecanismo é chamado de **polimorfismo**.
- O polimorfismo se realiza através da recodificação de um ou mais métodos herdados por uma subclasse.
  - Em tempo de execução, o Java saberá qual implementação deve ser usada.
- Tipos de polimorfismo: sobrecarga (*overload*) e sobreposição (*override*).

Polimorfismo

- A **sobrecarga** de métodos consiste em criar métodos distintos com nomes iguais em uma mesma classe, contanto que suas listas de argumentos sejam diferentes.
  - Se o programa encontrar dois métodos com argumentos iguais ele não saberá qual chamar e haverá um erro em seu programa.
  - A sobrecarga é muito utilizada em construtores.
  - Exemplo:

```
public class Calculadora{
   public int soma(int a, int b){
      return a+b;
    }
    public double soma(double a, double b){
       return a+b;
    }
}
```

Polimorfismo

- A **sobreposição** permite que os métodos da classe pai sejam reescritos nas classes filhas, transformando métodos genéricos em específicos e implementando outras funcionalidades.
  - Os métodos que serão sobrepostos devem possuir o mesmo nome, tipo de retorno e quantidade de parâmetros do método inicial.
  - Exemplo:

```
public class Calculadora{
   public int soma(int a, int b){
      return a+b;     }
}
public class CalculadoraCientifica extends Calculadora{
    public int soma(int a, int b){
       if((a>0)&&(b>=0)))
           System.out.println("Numeros positivos");
       return a+b;     }
}
```

Exercícios

Mais exemplos... (1)

- Em um sistema de loja, há 3 tipos de usuário: gerente, funcionário e cliente. Todo usuário tem nome e senha. O cliente possui, além do nome e senha, outros dados cadastrais. O funcionário possui métodos relacionados a venda de produtos. O gerente pode fazer tudo que o funcionário pode e também fechamento do caixa. Como é a hierarquia de herança desse sistema no que se refere a controle de usuários?

Mais exemplos... (2)

- O que está definido no código abaixo?

```
import java.util.Date;
public class Pessoa {
        public String nome;
        public String cpf;
        public Date data_nascimento;

        public Pessoa(String _nome, String _cpf, Date _data) {
                this.nome = _nome;
                this.cpf = _cpf;
                this.data_nascimento = _data;
        }
}
```

Mais exemplos... (2)

- O que está definido no código abaixo?

```
import java.util.Date;
public class Aluno extends Pessoa {
        public Aluno(String _nome, String _cpf, Date _data) {
                super(_nome, _cpf, _data);
        }
        public String matricula;
}
public class Professor extends Pessoa {
        public Professor(String _nome, String _cpf, Date _data) {
                super(_nome, _cpf, _data);
        }
        public double salario;
        public String disciplina;
}
public class Funcionario extends Pessoa {
        public Funcionario(String _nome, String _cpf, Date _data) {
                super(_nome, _cpf, _data);
        }
        public double salario;
        public Date data_admissao;
        public String cargo;
}
```

Mais exemplos... (3)

- O que está definido no código abaixo?

```
public class Pessoa {
        public String nome;
        public String cpf;
        public Date data_nascimento;
        public Pessoa(String _nome, String _cpf) {
                this.nome = _nome;
                this.cpf = _cpf;
        }
        public double tirarCopias(int qtd) { //Preço para tirar copias
                return 0.10 * (double) qtd;
        }
}
public class Aluno extends Pessoa {
        public Aluno(String _nome, String _cpf) {
                super(_nome, _cpf);
        }
        public String matricula;
        public double tirarCopias(int qtd) { //Preço para alunos
                return 0.07 * (double) qtd;
        }
}
```

Classes e métodos abstratos

Profa. Alessandra Mendes – UFRN/EAJ/TI/APDS

Classes e métodos abstratos

- Servem apenas como modelos para classes concretas;
- Não podem ser instanciadas diretamente com o *new*, devem ser herdadas por classes concretas.
- Podem conter ou não métodos abstratos, ou seja, pode implementar ou não um método.
  - Um método abstrato não tem corpo.
  - Os métodos abstratos definidos em uma classe abstrata devem obrigatoriamente ser implementados em uma classe concreta.
  - Se uma classe abstrata herdar outra classe abstrata, a classe que herda não precisa implementar os métodos abstratos.

Exemplo

```
public abstract class Eletrodomestico {
    private boolean ligado;
    private int voltagem;
    // métodos abstratos não possuem corpo */
    public abstract void ligar();
    public abstract void desligar();
    // método construtor - Classes Abstratas também podem ter métodos
    // construtores, mas não podem ser usados para instanciar um objeto
    public Eletrodomestico(boolean ligado, int voltagem) {
        this.ligado = ligado;
        this.voltagem = voltagem;
    }
    // métodos concretos - Uma classe abstrata pode possuir métodos concretos
    public void setVoltagem(int voltagem) {
        this.voltagem = voltagem; }
    public int getVoltagem() {
        return this.voltagem; }
    public void setLigado(boolean ligado) {
        this.ligado = ligado; }
    public boolean isLigado() {
        return ligado; }
}
```

Exemplo

```
public class TV extends Eletrodomestico {
     private int tamanho;
     private int canal;
     private int volume;
     public TV(int tamanho, int voltagem) {
          super (false, voltagem); // construtor classe abstrata
          this.tamanho = tamanho;
          this.canal = 0;
          this.volume = 0;
     }
     /* implementação dos métodos abstratos */
     public void desligar() {
          super.setLigado(false);
          setCanal(0);
          setVolume(0);
     }
     public void ligar() {
          super.setLigado(true);
          setCanal(3);
          setVolume(25);
     }
     // abaixo teríamos todos os métodos construtores get e set...
}
```

Exemplo

```
public class Radio extends Eletrodomestico {
    public static final short AM = 1;
    public static final short FM = 2;
    private int banda;
    private float sintonia;
    private int volume;
    public Radio(int voltagem) {
        super(false, voltagem);
        setBanda(Radio.FM);
        setSintonia(0);
        setVolume(0);
    }
    /* implementação dos métodos abstratos */
    public void desligar() {
        super.setLigado(false);
        setSintonia(0);
        setVolume(0);
    }
    public void ligar() {
        super.setLigado(true);
        setSintonia(88.1f);
        setVolume(25);
    }
    // abaixo teríamos todos os métodos construtores get e set...    }
```

Exemplo

```
public class Main {
    public static void main(String[] args) {
        TV tv1 = new TV(29, 110);
        Radio radio1 = new Radio(110);
        /**
        chamando os métodos abstratos implementados
        * dentro de cada classe (TV e Radio)
        */
        tv1.ligar();
        radio1.ligar();
        System.out.print("Neste momento a TV está ");
        System.out.println(tv1.isLigado() ? "ligada" : "desligada");
        System.out.print("e o Rádio está ");
        System.out.println(radio1.isLigado() ? "ligado." : "desligado.");
    }
}
```

Concluindo...

- As classes abstratas servem de base para codificação de uma classe inteira, diferentemente das interfaces que são apenas assinaturas dos métodos.
- Sumarizando, quando temos que definir variáveis, constantes, regras, e pequenas ações definidas devemos usar classes abstratas. Mas, se formos apenas criar a forma como objetos devem realizar determinadas ações (métodos) devemos optar por interfaces.

Interfaces

Profa. Alessandra Mendes – UFRN/EAJ/TI/APDS

Por que padronizar?

- Padronização
  - No dia-a-dia lidamos com diversos aparelhos elétricos e as empresas fabricam aparelhos elétricos com plugues;
  - E se cada empresa decidisse por conta própria o formato dos plugues ou tomadas que fabricará?
  - Essa falta de padrão pode gerar problemas de segurança, aumentando o risco de uma pessoa levar um choque...
  - O governo estabelece padrões para plugues e tomadas, facilitando a utilização para os consumidores e aumentando a segurança.
  - **Padronizar pode trazer grandes benefícios, inclusive no desenvolvimento de aplicações**.

Por que padronizar?

- Padrões de plugues

Contratos

- Podemos dizer que os objetos se “encaixam” através dos métodos públicos assim como um plugue se encaixa em uma tomada através dos pinos.
  - Para os objetos de uma aplicação “conversarem” entre si mais facilmente é importante padronizar o conjunto de métodos oferecidos por eles.

- Um padrão é definido através de especificações ou contratos.
  - Em orientação a objetos, um contrato é chamado de interface
  - Um interface é composta basicamente por métodos abstratos

Interfaces

- Os métodos de uma interface não possuem corpo (implementação) pois serão implementados nas classes vinculadas a essa interface.
  - São abstratos
- Todos os métodos de uma interface devem ser públicos e abstratos
  - Os modificadores *public* e *abstract* são opcionais
- As classes concretas que implementam uma interface são obrigadas a possuir uma implementação para cada método declarado na interface

Interfaces

- Vantagens:
  - Padronizar as assinaturas dos métodos oferecidos por um determinado conjunto de classes
  - Garantir que determinadas classes implementem certos métodos
- Exemplo:

```
1  interface Conta {
2    void deposita(double valor);
3    void saca(double valor);
4  }
```

```
1  class ContaPoupanca implements Conta {
2    public void deposita(double valor) {
3      // implementacao
4    }
5    public void saca(double valor) {
6      // implementacao
7    }
8  }
```

```
1  class ContaCorrente implements Conta {
2    public void deposita(double valor) {
3      // implementacao
4    }
5    public void saca(double valor) {
6      // implementacao
7    }
8  }
```

Modificadores *static* e *final*

Profa. Alessandra Mendes – UFRN/EAJ/TI/APDS

Modificador *static*

- Modifica o escopo de um método ou atributo, pois estes passam a **pertencer à classe** e não à instância do objeto.
- É usado para a criação de uma variável que poderá ser acessada por todas as instâncias de objetos desta classe como uma variável comum, ou seja, a variável criada será a mesma em todas as instâncias e quando seu conteúdo é modificado numa das instâncias, a modificação ocorre em todas as demais.

Métodos *static*

- Às vezes, um método realiza uma tarefa que não depende de um objeto. Esse método se aplica à classe em que é declarado como um todo e é conhecido como método *static* ou método de classe.
- Os métodos *static* podem ser chamados sem uma instância pois ajudam no acesso direto à classe.
- Desta forma, não é necessário instanciar um objeto para acessar o método.

```
NomeDaClasse.nomeDoMétodo(argumentos)
```

Atributos *static*

- Os atributos *static* possuem o mesmo valor para todas as instâncias de um objeto;
- Exemplo:

```
public class MinhaClasse {
      static int valorGlobal = 1;
      public static int getValorGlobal() {
             return valorGlobal;
      }
}
       // classe principal
       MinhaClasse c1 = new MinhaClasse();
       MinhaClasse c2 = new MinhaClasse();
       MinhaClasse.valorGlobal = 2;
       System.out.println(c1.getValorGlobal()); //imprime 2
       System.out.println(c2.getValorGlobal()); //imprime 2
```

Classes *final*

- Uma classe com este modificador **não pode ser estendida**, isto é, não pode ter classes que herdam dela.
- Usa-se para garantir que uma determinada implementação não tenha seu comportamento modificado (imutabilidade).
- Exemplo:

```
public final class minhaClasse
  { ... }
```

Métodos *final*

- É um método que não pode ser sobrescrito nas subclasses.
- Usa-se para garantir que um determinado algoritmo não possa ser modificado pelas subclasses.
- Exemplo:

```
public class Xadrez{
    int jogador;
    final int getJogador() {
        return jogador;
    }
    ...
}
```

Atributos *final*

- É um atributo que só pode ter seu valor atribuído uma única vez, seja na própria declaração ou no construtor.
- Usa-se para garantir que um atributo de objeto não vai mudar.
- Exemplo:

```
public class MinhaClasse {
    final int x = 1;
    final int y;
    public MinhaClasse(int y) {
        this.y = y;
    }
}
```

Variáveis *final*

- É uma variável que só pode ter seu valor atribuído uma única vez.
- Use para garantir que você não está modificando o valor indevidamente.
- Exemplo:

```
final boolean a = lerInputUsuario();
final boolean b = lerInputUsuario();
```

Resumindo o *final*...

- Quando é aplicado na classe, não permite estendê-la;
- Quando é aplicado nos métodos, impede que o mesmo seja sobrescrito (*overriding*) na subclasse;
- Quando é aplicado nos atributos ou valores de variáveis, não pode ser alterado depois que já tenha sido atribuído um valor.

static e *final* juntos

- Utilizados juntos para constantes, pois indicam que o mesmo valor vai ser visto para todas as instâncias da classe (*static*) e nunca vai poder ser modificado depois de inicializado (*final*).

Vamos à prática...

Listas para estudo

- Lista 2
- Lista 3

- Implementar e/ou acompanhar as resoluções em vídeo dos exercícios das listas 2 e 3.
- Na próxima aula poderão ser tiradas dúvidas das questões.

Dúvidas?