MC0037 – Programação para Web

Aula 1: Revisão de POO

2 Q / 2015

Aula de hoje: Roteiro

- Programação Orientada a Objetos em Java
  - Classes, objetos
  - Membros de instância e membros estáticos
  - Construtores
  - Herança
  - Polimorfismo
- Convenções do Java

Conceitos

- POO é um **paradigma de programação** cuja essência é resolver problemas através da modelagem do mundo real através de objetos.

- Quando dizemos mundo real, queremos dizer uma porção limitada, ou algum ambiente que de fato existe, envolvendo seres vivos e objetos.

- Paradigma, no dicionário Aurélio, significa “padrão” ou “metodologia”.

Conceitos

- Na POO, o programador é **responsável pela criação de modelos computacionais** que representem elementos do mundo real: pessoas, animais, plantas, veículos, edifícios, documentos, máquinas, conta bancária, etc.

- Estes modelos computacionais são representações simplificadas dos elementos reais, que buscam descrever principalmente:
  - Propriedades mais relevantes
  - Comportamentos mais relevantes
  - Interação entre os elementos

Exemplo

- Um programador está interessado em criar uma aplicação para gerenciar as diversas mesas de um restaurante.

- Um modelo para "Mesa de restaurante"
  - **Propriedades** mais relevantes
    - Número da mesa
    - Número de lugares
    - Estado (ocupada?)
    - Pedidos entregues
    - Pedidos Pendentes
  - **Comportamento** (ações)
    - Adicionar pedido
    - Entregar pedido
    - Fechar a conta

Classe

- Uma classe é o principal elemento de um programa OO.
- Em geral, uma classe representa um objeto do mundo real.
- Uma classe é um *template* (molde) para definir e criar uma categoria de objetos.
- Uma propriedade de um objeto é definida por um atributo que é uma variável que pode armazenar o valor que representa a propriedade.
- O comportamento de um objeto é definido por métodos.
- Os atributos e métodos da classe são conhecidos como os membros da classe.
- Declaração de classe em Java:

```
public class Exemplo {
   // declaracao de atributos e métodos
}
```

Objeto

- Um objeto é uma instância de uma classe.
- O objeto é criado usando a classe como modelo (este processo é chamado de instanciação).
- Quando um objeto é instanciado, uma referência para o objeto é retornada e representa o objeto criado.
- Uma referência de objeto é uma variável que armazena um valor de referência e é um identificador para um objeto.
- A criação de um objeto consiste em declarar uma variável de referência (do tipo da classe desejada) e criar o objeto usando a palavra-chave **new**.

```
Exemplo var1;    // declaracao do objeto var1 do tipo Exemplo
var1  = new Exemplo();  // criação do objeto
```

Exemplo de classe

- Exemplo de classe em Java:

```
class  MesaRestaurante {

  int numero;
  String[] pedidosPendentes;
  String[] pedidosEntregues;
  boolean ocupada;


  void adicionarPedido(String pedido){..}

  void fecharConta() {...}

}
```

Propriedades / Atributos

Comportamentos / Métodos

Criando um objeto

- Exemplo de criação de objeto

```
public class Test {
    public static void main(String[] args) {
       MesaRestaurante m1; // declaração da variável de
                  // referência (ainda não foi criado objeto!)

      m1 = new MesaRestaurante();  // criação do objeto


      m1.numero = 10;  // acessando um campo do objeto
      m1.fechaConta();  // chamando um método do objeto
    }
 }
```

Variáveis em Java

- As variáveis em Java armazenam valores de tipos de dados primitivos (int, boolean, char, double, etc.) e também podem armazenar referências de objetos.
- Variáveis que armazenam referências de objetos são chamadas variáveis de referência.

- Declaração de variáveis:

```
int a, b, c;        // a, b e c são variaveis inteiras
boolean flag;       // flag é uma variável booleana
int i = 10;         // i é uma variável inteira com valor
                    // inicial 10
Exemplo var1;       // var1 é uma variável de referência
```

Membros de instância

- Cada objeto criado em Java possui suas próprias cópias de atributos (definidos em sua classe) e são chamados de variáveis de instância.
- Os valores das variáveis de instância de um objeto compõem o estado do objeto.
- Os métodos de um objeto definem o seu comportamento (métodos de instância).
- As variáveis de instância e os métodos de instância, que pertencem a objetos são chamados de membros de instância (que são diferentes dos membros estáticos, que pertencem às classes).

```
class Exemplo {
    int i;   // variavel de instancia

    void metodo1() { // metodo de instancia
    }
}
```

Membros estáticos

- Membros estáticos, que são declarados com a palavra-chave **static**, são aqueles pertencem à classe e não a um objeto específico.
- Uma classe pode ter variáveis estáticas e métodos estáticos.
- Variáveis estáticas são inicializadas quando a classe é carregada, na execução.

```
class Exemplo {
    static int i;   // variável estática

    static void metodo1() { // método estático
        ...
    }
}
```

Variáveis estáticas

- Uma variável estática é aquela que é compartilhada por todas as instâncias (objetos) de uma classe.
- Há apenas uma cópia da variável para uma classe em vez de uma cópia para cada instância.

- Utilidades de campos estáticos em classes:
  - Manter uma informação (ou estado), para todas as instâncias de uma classe de maneira compartilhada, de modo que esta possa ser modificada ou acessada por qualquer uma das instâncias. (Ex.: uma variável contadora)

  - Armazenar valores que não serão modificados por instâncias da classe: constantes.

Variáveis estáticas

- Armazenar valores que não serão modificados por instâncias da classe: constantes

Convenção de nomeação de constantes: maiúsculas e *underline* separando as palavras

```
public class ConstantesMatematicas {
    public static final double PI = 3.14159265358979323846264433;
    public static final double RAIZ_DE_2 = 1.41421356237309504880;
    public static final double RAIZ_DE_3 = 1.73050807568772935;
    public static final double RAIZ_DE_5 = 2.23606797749978969640;
}
```

O modificador **final** faz com que os valores dos campos, depois de inicializados, não possam mais ser modificados.

Exemplo de utilização da classe:

```
area =  ConstantesMatematicas.PI * raio * raio;
```

Observe que não é preciso criar a instância da classe, embora seja possível.

Métodos estáticos

- Se um método é estático significa que o mesmo não requer uma instância (objeto) da classe para ser executado, ou seja, o seu comportamento não depende de um objeto específico.
- Assim como as variáveis estáticas, os métodos estáticos podem ser chamados a partir do nome da classe, sem que seja necessário criar um objeto daquela classe.

- Exemplo: Para calcular a raiz quadrada de um número x:

```
double raiz = Math.sqrt(x);
```

Métodos estáticos

- Principal aplicação: Criar bibliotecas de métodos
- Exemplo: uma biblioteca para conversão de unidades de comprimento

```
public class ConversaoDeUnidades {
    public static double polegadasParaCentimetros(double polegadas){
        return (polegadas * 2.54);
    }

    public static double pesParaCentimetros(double pes) {
        return (pes * 30.48);
    }
}
```

- Exemplo:

```
double cent = ConversaoDeUnidades.polegadasParaCentimetros(5.4);
```

Sobrecarga de métodos

- Cada método possui um nome e uma lista de parâmetros.
- O nome do método e a lista de parâmetros constituem a assinatura do método.
- Pode-se ter mais de um método com o mesmo nome, desde que as assinaturas sejam diferentes, isso é chamado de sobrecarga de métodos.
- Logo, métodos sobrecarregados são aqueles que tem o mesmo nome mas possuem uma diferente lista de parâmetros.
- Por exemplo, qual o erro na declaração dos métodos abaixo?

```
1. void metodoA (int a, double b) { }
2. int metodoA(int a) { return a; }
3. int metodoA() { return 1; }
4. long metodoA(double a, int b) { return b; }
5. long metodoA(int c, double d) { return c; }
```

Sobrecarga de métodos

- As 4 primeiras implementações de métodos estão sobrecarregadas corretamente, uma vez que cada método possui uma diferente lista de parâmetros, e portanto uma assinatura diferente.
- O método na linha 5 possui a mesma assinatura do método na linha 1, muda apenas o tipo de retorno. Mudar apenas o tipo de retorno de um método não é suficiente para sobrecarregar o método, portanto está incorreto.

```
1. void metodoA (int a, double b){ }
2. int metodoA(int a) { return a; }
3. int metodoA() { return 1; }
4. long metodoA(double a, int b) { return b; }
5. long metodoA(int c, double d) { return c; }  // erro
```

Construtor

- Quando um objeto é criado usando o operador **new**, o construtor é chamado para atribuir o estado inicial do objeto.
- O construtor deve ser declarado com o mesmo nome da classe.
- Construtores não retornam valor. Exemplo:

```
class ExemploA {
    int i;

    ExemploA (int i) {  // construtor
       this.i = i; // atribui um valor inicial para i
    }
}
```

- Quando nenhum construtor é especificado na classe, um construtor padrão (sem parâmetros) é gerado pelo compilador, que faz uma chamada para o construtor da superclasse.

Construtor padrão implícito

- Exemplo:

```
class ExemploA { // classe sem construtor explícito
    int i;
}
```

```
class ExemploB {
    ExemploA var1 = new ExemploA();    // chama o construtor
                                       // padrão
}
```

```
// construtor padrão implícito (não aparece na classe ExemploA,
// é criado pelo compilador):
class ExemploA () {
   super();
}
```

Construtor

Se a classe define um construtor explícito, o compilador não vai gerar o construtor padrão.

```
class ExemploA {
    int i;

    ExemploA (int i) {  // construtor
       this.i = i;
    }
}
```

```
class ExemploB {
    //  ...
    ExemploA var1 = new ExemploA(2);// chama o construtor

    ExemploA var2 = new ExemploA();  // erro de compilação
 }
```

Construtores sobrecarregados

- Construtores podem ser sobrecarregados, assim como os métodos, apenas as listas de parâmetros devem ser diferentes entre si.

```
class ExemploA {
    int i;
    ExemploA () {          // construtor sem parâmetros
      i = 0;
    }
    ExemploA (int i) {  // construtor com um parâmetro
          this.i = i;
    }
 }
```

```
class ExemploB {
    //  ...
    ExemploA var1 = new ExemploA(3);
    ExemploA var2 = new ExemploA();
 }
```

Encapsulamento

- É a ocultação das características internas de um objeto, de forma que estas não possam ser vistas ou modificadas externamente.
- O encapsulamento “protege” os dados que estão dentro do objeto, impedindo que os mesmos sejam acessados ou modificados de forma direta por um outro objeto, evitando que sejam alterados erroneamente.
- Em termos de implementação, o encapsulamento pode ser alcançado tornando os atributos de uma classe como privado (*private*).
- Os dados só podem ser acessados ou alterados através dos métodos públicos do objeto, que controlam as operações que podem ser feitas sobre eles.

- Obs.: *private* é um modificador de acesso, outros possíveis são *public*, *protected* ou sem modificador (default).

Encapsulamento

- Exemplos no mundo real:
  - Celular, carro, televisão, etc.
  - Usuários finais tem acesso a apenas alguns comandos e informações.

- Vantagens:
  - O usuário terá acesso somente àquilo que lhe interessa.
  - Ele não precisa ter conhecimento do funcionamento do objeto (apenas assume que o objeto “funciona”, e faz uso do mesmo - caixa preta).

  - Facilidade de manutenção.
  - Segurança da informação (controle na atribuição de novos valores).

Encapsulamento

- Exemplo:

```
class ContaCorrente {
    private String nome;
    private double saldo;

 }
```

```
class Teste {
    public static void main(String args[]) {
        ContaCorrente conta1 = new ContaCorrente();
        conta1.saldo = 1000;   // erro de compilação!
    }
}
```

Encapsulamento

- Escrevendo métodos para acessar/modificar os atributos:

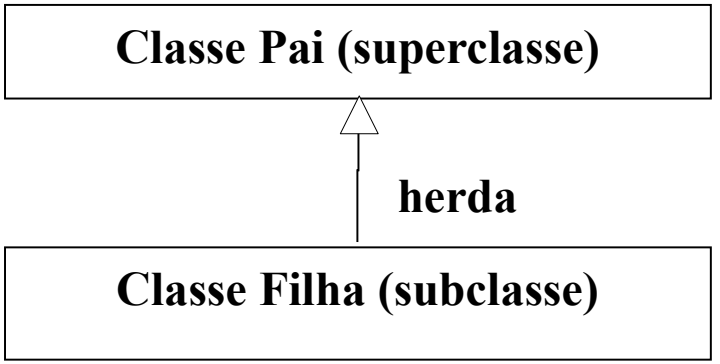
```
class ContaCorrente {
    private String nome;
    private double saldo;

    public String getNome() {
       return nome;
    }
    public void setNome(String nome) {
       this.nome = nome;
    }
    public double getSaldo() {
       return saldo;
    }
 }
```

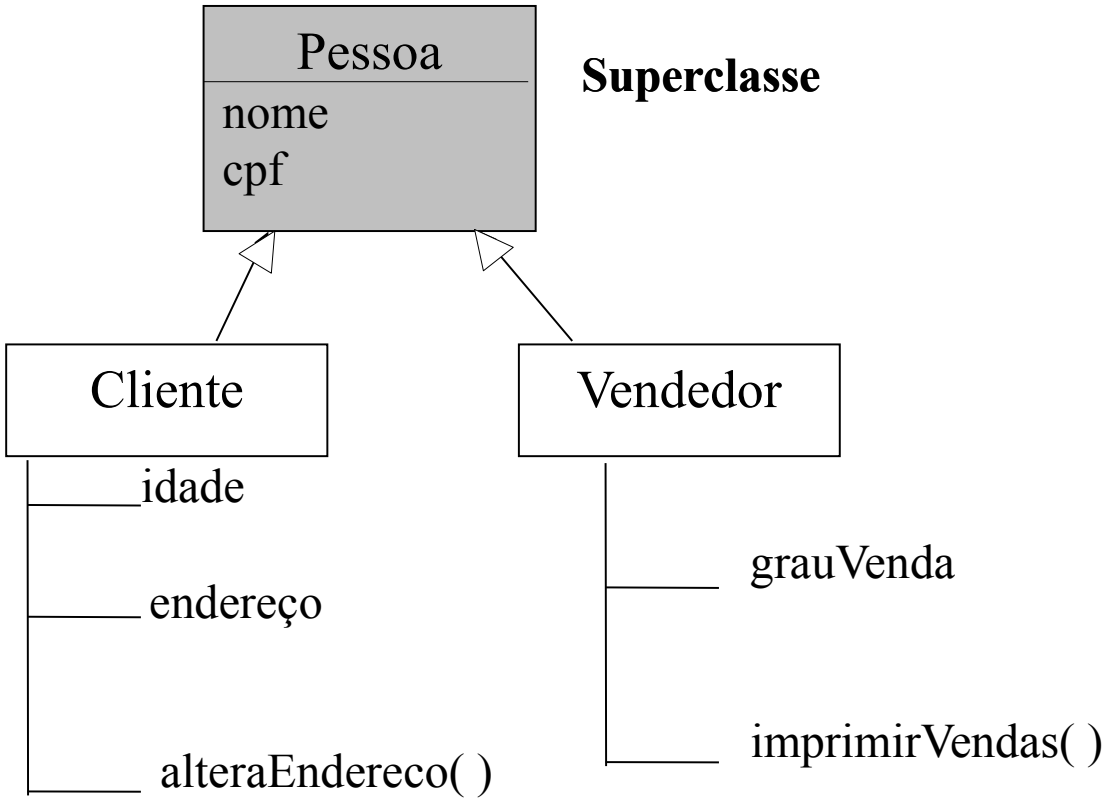
  - Observe que nem sempre queremos ter métodos para acessar/modificar todos os atributos da classe, isso dependerá da aplicação, das regras de negócio.

Herança

- Herança é um dos conceitos fundamentais em programação OO.
- Utilizando herança, uma classe herda as mesmas propriedades e métodos de uma outra classe, possibilitando criar especializações desta.
- Proporciona relacionamentos do tipo Pai-Filho, criando uma estrutura hierárquica de classes.
- A classe “pai” (superclasse) agrupa as características mais genéricas, enquanto que a classe “filha” (subclasse) herda tais características, possivelmente especializando algumas destas e até mesmo adicionando novas.



Exemplo

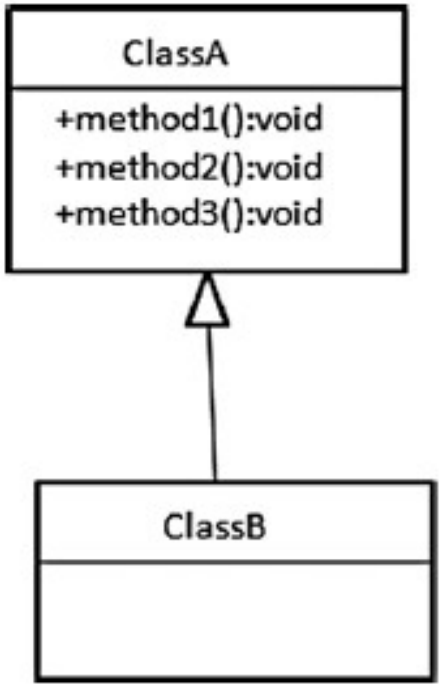


Em Java, todas as classes estendem por padrão a classe **java.lang.Object**.

Uma classe estende outra usando a palavra-chave **extends**.

- Quando a classe Vendedor é instanciada, a classe se comportará como Vendedor ou Pessoa.

Exemplo de código



- Exemplo:

```
public class ClasseA {

    public void metodo1() {
        System.out.println(" classeA - metodo1");
    }
    private void metodo2() {
        System.out.println(" classeA - metodo2");
    }
    public static void metodo3() {
        System.out.println(" classeA - metodo3");
    }
}
```

```
public class ClasseB extends ClasseA {
    ...
}
```

Exemplo de código

- Mesmo que não existam métodos declarados na classe ClasseB, os métodos da classe ClasseA estão disponíveis na ClasseB (foram herdados) e podem ser invocados usando uma referência para um objeto da ClasseB.
- Métodos privados não são herdados.

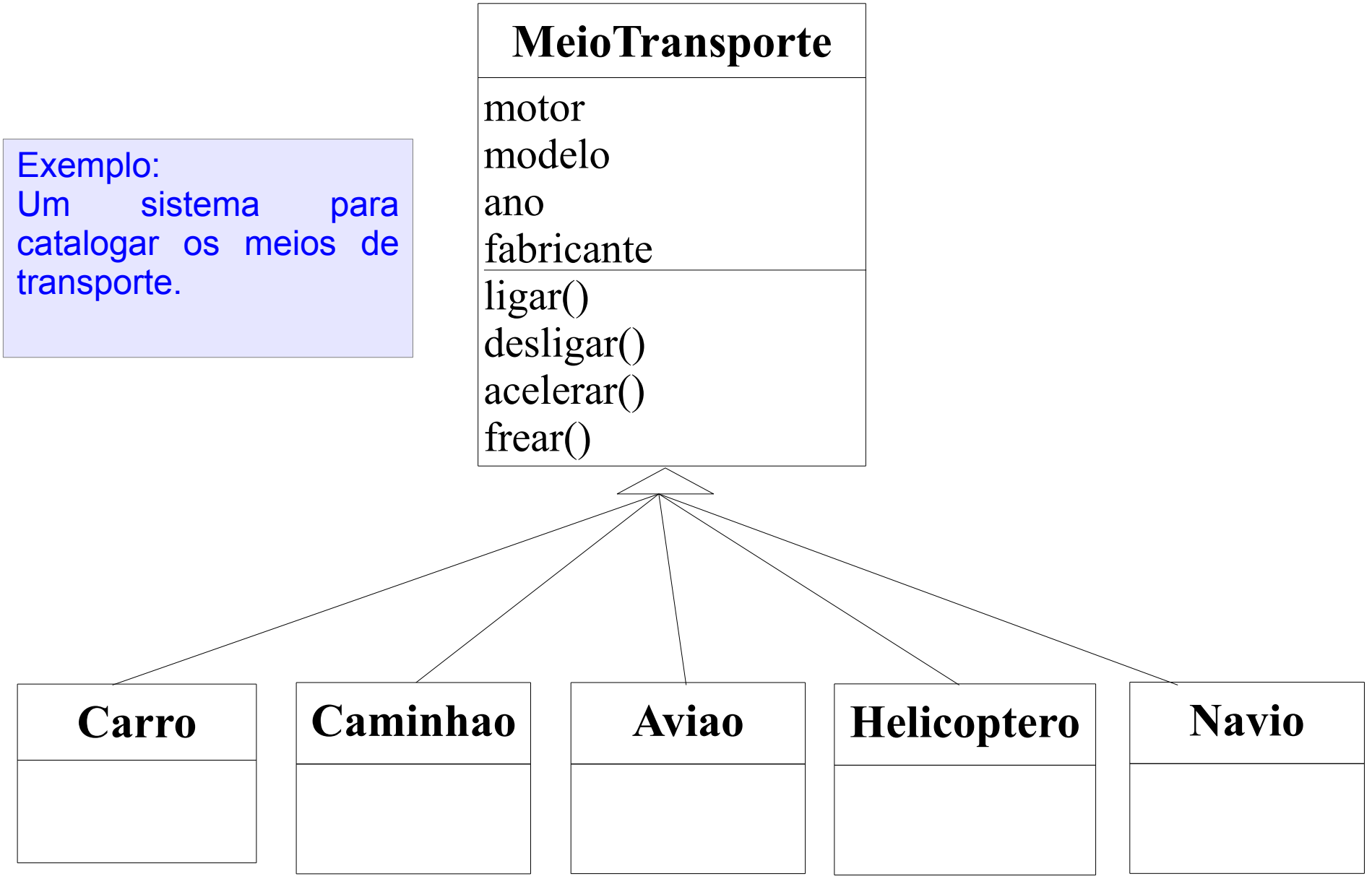
```
public class Test {
    public static void main(String[] args) {
        ClasseB var1 = new ClasseB();
        var1.metodo1();
        var1.metodo2();  // Erro de compilação
                         // método privado não é herdado
        ClassB.metodo3();  // acessando o método estático
    }
}
```

Saída:

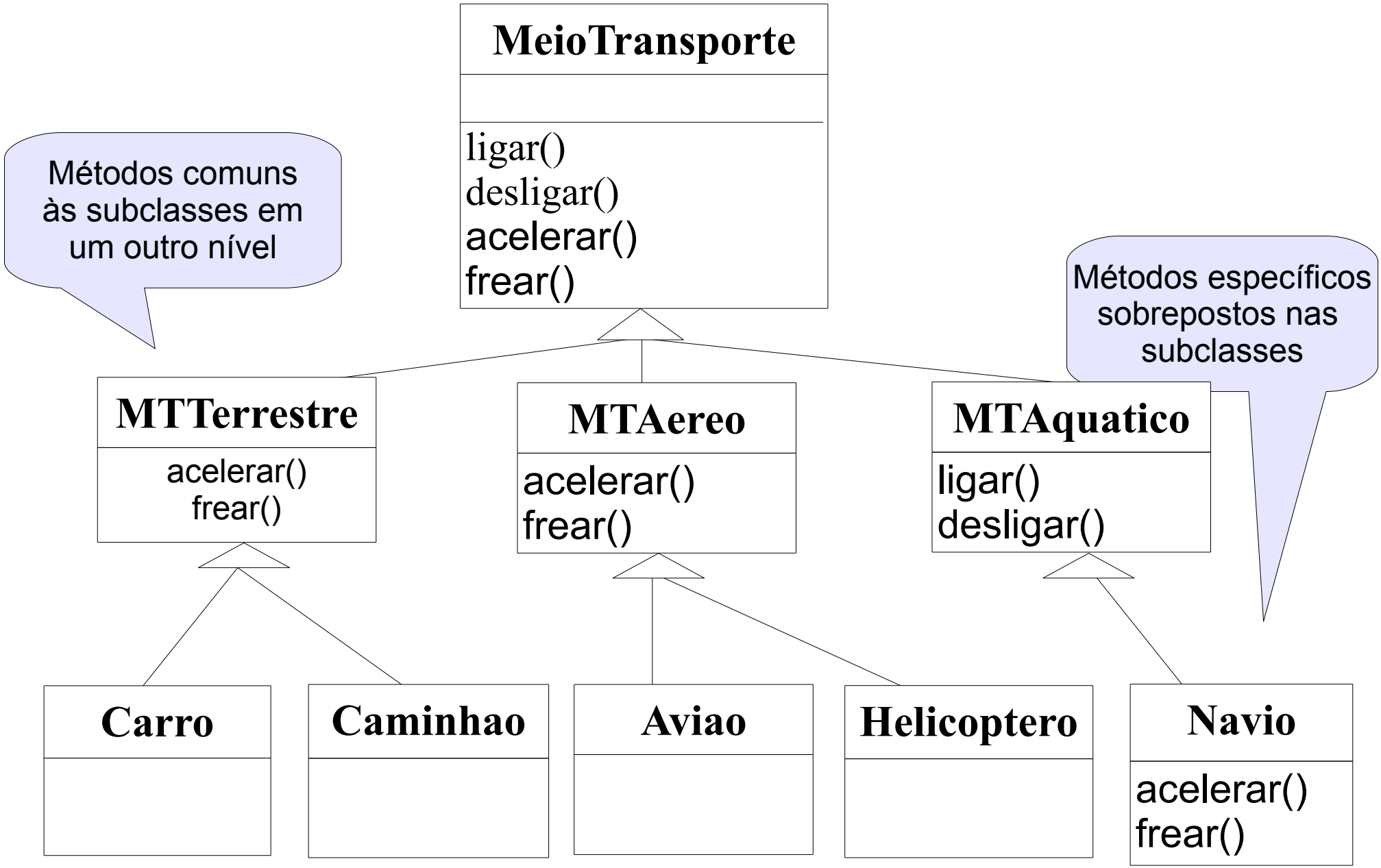
```
classeA - metodo1
classeA - metodo3
```

Outro exemplo: Meios de transporte

Exemplo:
Um sistema para catalogar os meios de transporte.



Meios de transporte



Herança

- Os métodos herdados podem ser sobrepostos para adicionar ou alterar o comportamento.
  - Exemplo de sobreposição adicionando funcionalidade ao método da subclasse (por exemplo, Navio):

```
public void acelerar()    {
     super.acelerar(); // chama o método da superclasse

        … // codigo adicional
}
```

Quando usar herança

- Use herança quando tiver uma classe que é de um tipo mais específico que uma outra
  - Ex.: Uma Macieira é um tipo específico de Árvore, portanto, faz sentido Macieira estender Árvore

- Considere a herança quando tiver um comportamento (código implementado) que possa ser compartilhado entre várias classes do mesmo tipo (→ funcionalidades na superclasse)
  - No exemplo anterior, Carro e Caminhao devem acelerar e frear da mesma forma

- Obs.: Não usar herança somente para reutilizar código, i.e., se as classes não possuem qualquer relacionamento.

Polimorfismo

- Do grego *poli* (=muitas) *morphus* (=formas)
- É a capacidade que um objeto possui de se comportar de maneiras diferentes de acordo com o tipo instanciado.

- Em Java o tipo é importante:

```
Carro c = new Carro(); // atribui um objeto Carro à variável de referência c (tipo da variável de referência é igual ao tipo do objeto)
Carro c = new Aviao(); //  erro de compilação
```

- Mas isso é possível:

```
MeioTransporte mt = new Carro();
```

  - Com o polimorfismo, o tipo da variável de referência pode ser diferente do tipo do objeto
  - Desde que o tipo da variável de referência seja uma superclasse do tipo do objeto

Polimorfismo

- Exemplo: Vetor “Polimórfico”

```
MeioTransporte mt = new MeioTransporte[3];
mt[0] = new Carro();
mt[1] = new Aviao();
mt[2] = new Navio();


for(int i = 0; i < mt.length; i++) {
    mt[i].ligar();
    mt[i].acelerar();
}
```

Vetor que conterá referências de objetos do tipo MeioTransporte

Qualquer objeto (subclasse de MeioTransporte) pode ser inserido no vetor

Chama o método no objeto correspondente.
Ex.: quando i=0, chama ligar() e acelerar() do objeto Carro.

Polimorfismo

Qualquer objeto (subclasse de MeioTransporte) pode ser passado como parâmetro.

```
public class Motorista {
    public void acionarMotor(MeioTransporte mt) {
        mt.ligar();
    }
}
```

Chama o método ligar() do objeto que foi passado como parâmetro.

```
class Teste {
    public static void main(String[] args) {
        Motorista mot = new Motorista();
        Carro c = new Carro();
        Aviao a = new Aviao();
        mot.acionarMotor(c);
        mot.acionarMotor(a);
    }
}
```

Passando por parâmetros objetos dos tipos Carro e Aviao que são subclasses de MeioTransporte

Polimorfismo

- Com o polimorfismo, novos comportamentos (sobreposição de métodos) podem ser definidos nas novas subclasses → no momento da chamada, o método utilizado será aquele definido pela classe real do objeto.

- Métodos com argumentos “polimórficos” não precisam ser alterados → qualquer classe que estenda a superclasse (definida como tipo de parâmetro no método) pode ser passada como argumento.

Convenções do Java

- **nomes de variáveis:**
  - primeira letra minúscula;
  - se tiver mais de uma palavra, as demais palavras começam com maiúsculas. **Exemplos**: idadeDoCliente, nomeDoCliente, hojeEstaNublado, etc.

- **nomes de classes:**
  - primeira letra maiúscula;
  - se tiver mais de uma palavra, as demais palavras começam com maiúsculas. **Exemplos**: Pessoa, ImpostoDeRenda, TestaIdade, Cliente, TestaAcesso, etc.
  - nome da classe é o nome do arquivo;

Convenções do Java

- **nomes de métodos:**
  - primeira letra minúscula;
  - se tiver mais de uma palavra, as demais palavras começam com maiúsculas.
  - sempre no imperativo: calculaImposto(), testaIdade(), testaAcesso(), saca(), validaCPF(), etc.

- **nomes de pacotes:**
  - primeira letra minúscula;
  - se tiver mais de uma palavra, as demais palavras começam com maiúsculas. **Exemplos**: criptografia, usuarios, conexaoDeBancoDeDados, etc.
  - nome do pacote deve ser o mesmo nome da pasta;

Convenções do Java

- nomes de constantes:
  - todas as letras são maiúsculas;
  - para separar as palavras, usamos underline. **Exemplos**: TAMANHO, PI, QUANTIDADE, PARAR_DE_EXECUTAR, etc;

Fonte:
http://www.oracle.com/technetwork/java/codeconventions-135099.html#367

Java – o que não esquecer

- **classes public** PRECISAM ter o nome de arquivo igual ao nome da classe;
- **private:** acessível somente dentro da classe. Utilizado para atributos, construtores e métodos;
- **public**: acessível de qualquer lugar mesmo de outras pastas (neste caso, precisa de import). Utilizado para classes, atributos, construtores e métodos;
- **sem private nem public**: acessível somente a arquivos na mesma pasta;
- **protected**: pode ser acessado por todas as classes do mesmo pacote e por todas as classes que o estendam, mesmo que não estejam no mesmo pacote. Utilizado para atributos, construtores e métodos;

Java – o que não esquecer

- **pacotes (packages)**: agrupam classes de funcionalidades similares ou relacionadas. Padrão da Sun/Oracle: relativo ao nome da empresa que desenvolveu. Exemplo:

br.edu.ufabc.progweb.subpacote1;

br.edu.ufabc.progweb.subpacote2;

onde "ufabc" é a "empresa"; "progweb" é o projeto; "subpacote1" e "subpacote2" são dois diretórios dentro do projeto progweb;

- **@Override**: notifica o compilador que estamos sobreescrevendo um método da classe mãe.

Referências

- Deitel, H. M. e Deitel, P. J.; *JAVA – Como Programar*; 6ª edição, Editora Pearson Prentice-Hall, 2005.

- Santos, R. Introdução à Programação Orientada a Objetos usando Java, Ed. Campus, 2003.