**Disciplina: Programação Orientada a Objetos**
**Professor: Antonio Henrique Pinto Selvatici**

**AULA 3 – Classes e objetos com encapsulamento**

ROTEIRO:

- Rever programa da aula anterior
- Encapsulamento (continuação)
- Ciclo de vida do objeto
- Uso em arrays
- Diagrama de classes
- Exercício: especificação e implementação do sistema bancário

1. **Programa da aula anterior**
2. **Encapsulamento**

Como vimos na aula anterior, o encapsulamento em POO corresponde ao fato de que as informações (atributos) estão encapsuladas junto das tarefas que usam essas informações dentro do objeto. Dessa forma conseguimos reduzir o acoplamento entre as partes do sistema, pois apenas os métodos dentro da classe precisariam conhecer os atributos daquela classe. Para reforçar a ideia do encapsulamento, usamos a ocultação de informações, ou seja, evitamos que métodos de outras classes tenham acesso direto aos atributos.

Por enquanto, vamos considerar apenas os modificadores de visibilidade public e private. Quando usamos public, indicamos que métodos de qualquer classe podem acessar aquele atributo ou método, enquanto private restringe o acesso ao atributo ou método aos métodos da própria classe. Por exemplo, seja a definição de classe abaixo:

```
public class Pessoa {
    private String nome;
    private int idade;

    public void apresentar() {
        System.out.println("Olá, sou " + nome + " e tenho " +
idade + "anos");
    }
}
```

Neste exemplo, temos os atributos todos privados, ou seja, não podem ser acessados de fora da classe. Apenas o método apresentar é visível de fora da classe.

Agora vamos supor que outra classe deseja saber o nome da pessoa. Isso não será possível no exemplo acima, a não ser que criemos um método específico para isso dentro da classe:

```
public String getName() {
        return nome;
}
```

Alguns atributos de uma classe são, **em alguns casos**, naturalmente requisitados fora da classe que os contêm, às vezes sendo necessário apenas a leitura do seu valor, e às vezes sendo necessário inclusive a modificação desse atributo fora da classe proprietária. Esses atributos são conhecidos em várias linguagens orientadas a objeto como “propriedades”. Porém, mesmo que o acesso à propriedade consista apenas em ler ou gravar o atributo de forma simples, o acesso a ela sempre se dará por meio de métodos específicos para esse fim. Em Java, esses métodos são os famosos “getters” e “setters”, e devem seguir o seguinte padrão:

Propriedade: inicia com letra minúscula, escrita em Camel Case. Ex: nomeDaMae.

Getter: get<Propriedade>(), retornando o seu valor.

Ex: `String getNomeDaMae();`

Setter: set<Propriedade>(<tipo> valor), retornando void.

Ex: `void setNomedaMae(String nome);`

Por que são necessários os getters e setters, mesmo que a propriedade seja de leitura e escrita? Porque esses métodos controlam a forma como o valor atributo do objeto será acessado. Por exemplo, caso a propriedade **nome** possa ter apenas até 11 caracteres, devemos criar o setter de forma que:

```
public setNome(String valor) {
    if(valor.length() > 11) {
        nome = valor.substring(0,11);
    } else {
        nome = valor;
    }
}
```

E o que fazer quando não podemos gravar o valor de um ou mais atributos? De onde eles virão? Esses valores poderão ser passados no momento de criação do objeto, como veremos a seguir.

Além da visibilidade dos métodos e atributos, o encapsulamento em Java é afetado pelo seu escopo. O escopo, ou seja, o alcance de métodos e de atributos pode ser:

- **De instância (dinâmico):** indica que o atributo tem um valor diferente para cada instância da classe (objeto). Os atributos definidos na classe são naturalmente dinâmicos, não cabendo nenhum modificador. Os métodos com escopo de

instância têm acesso direto aos atributos de instância, além da referência para o objeto que o invocou através da palavra reservada **this**.

- **De classe (estático):** indica que o atributo tem o mesmo valor, valendo para a classe inteira, independente do objeto que o invoca. Os métodos estáticos e dinâmicos também acessam diretamente os atributos estáticos, porém os métodos com escopo de classe não têm acesso direto aos atributos dinâmicos, mas apenas através de uma instância específica da classe.

## 3. Ciclo de vida do objeto

Variáveis são trechos de memória com posições e tamanho predefinidos, que são invocadas no programa através de um nome. Porém, diferentemente de uma variável numérica, um objeto não é guardado inteiro dentro da variável: ela apenas guarda uma referência para outra posição de memória, onde o objeto é alocado no processo de criação do mesmo. O conteúdo dessas variáveis são conhecidos como objetos dinâmicos, uma vez que a posição de memória em que estão alocados pode mudar ao longo da execução programa.

Um objeto pode inclusive ter o valor **null**, indicando que aquela variável não aponta para nenhum objeto. E quando uma variável do tipo objeto recebe o valor de outra variável do mesmo tipos, ambas passam a referenciar o mesmo objeto.

Quando uma classe é instanciada para gerar o objeto, um método especial denominado construtor é invocado. Um construtor pode ter nenhum ou vários parâmetros, pode ter diferentes níveis de visibilidade (public ou private, por exemplo), porém o tipo de retorno não é nunca especificado. Um construtor é definido da forma:

<visibilidade> <NomeDaClasse> ( <parâmetros> ) { <corpo do construtor> }

Assim, um possível construtor da classe Pessoa acima é:

```
public Pessoa( String nome, int idade ) {
    this.nome = nome;
    this.idade = idade;
}
```

Primeiramente, observe que o construtor acima evita a necessidade de criar setters para os atributos nome e idade, forçando que esses dados sejam especificados na criação do objeto (a não ser que haja outros construtores).

Segundo, o que significa **this**? É a palavra reservada que nos permite referenciar a posição de memória ocupada pelo objeto que estamos criando ou manipulando, tendo acesso aos seus atributos. Neste caso, usamos o this para diferenciar entre o atributo do próprio objeto sendo criado (`this.nome`, por exemplo) do parâmetro do construtor.

E quando não especificamos nenhum construtor? Nesse caso o compilador Java cria automaticamente o construtor padrão (default constructor), com visibilidade pública e nenhum parâmetro, e que não modifica os atributos do objeto.

Em algum momento, o objeto que foi criado deixará de ser usado. Isso acontecerá quando todas as variáveis e atributos que o referenciam deixam de existir na memória do computador. Nesse momento o Java marca esse objeto para remoção, que ocorre quando um processo em paralelo chamado “Garbage Collector” devolve a memória usada pelo objeto, fechando quaisquer recursos que o mesmo possam ter utilizado.

## 4. Usando arrays com objetos

Em primeiro lugar, um array é tratado pelo Java como um objeto especial (observe que também é instanciado com **`new`**). Quando criamos um array de um tipo objeto, todas as suas posições são na verdade referências para objetos, e são inicializadas com null.

Assim, caso queiramos preencher posições desse array, temos instanciar um objeto individual para cada posição. Por exemplo:

```
Pessoa[] pessoas = new Pessoa[50]; // array com 50 posições
pessoas[0] = new Pessoa("João", 30); // primeira posição
System.out.println("Nome da primeira pessoa: " +
     pessoas[0].getNome());
pessoas[1].getIdade(); // Null Pointer Exception
```

## 5. Exercício

Transformar o sistema bancário desenvolvido na última aula em um sistema com programação orientada a objetos. Vamos pensar nas classes necessárias, quais os métodos e atributos mais adequados, focando no encapsulamento dos métodos e atributos.

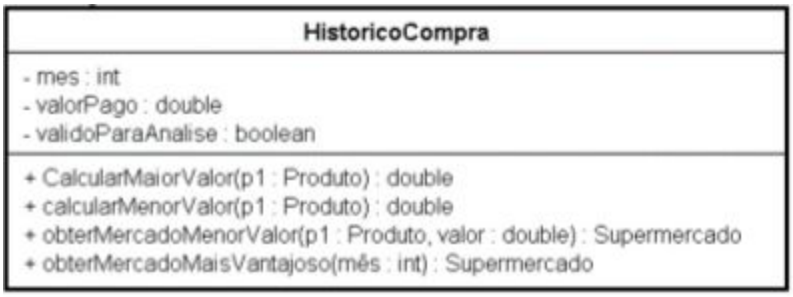
Para armazenar os objetos referentes a clientes e contas bancárias, use um

## 6. Diagrama de classes

A UML define uma forma de visualizarmos a principais classes a serem implementadas no sistema. Dentro da chamada visão lógica, que explica como os diferentes componentes se conectam, temos o diagrama de objetos e o diagrama de classes. Enquanto o diagrama de objetos mostra uma “fotografia” do sistema, exemplificando que objetos poderiam estar instanciados e quais os valores de seus atributos, o diagrama de classes mostra as classes que devem ser implementadas, quais são seus métodos e atributos e como elas se associam

Uma classe no diagrama de classes é representada por três divisões: a do nome da classe, em cima, a dos atributos no meio e a dos métodos embaixo. Junto aos métodos e atributos é indicada a visibilidade de cada um com um símbolo de “+” para visibilidade pública e “-” para visibilidade privada.

Os atributos e parâmetros dos métodos são indicados da forma `<nome>: <tipo>`, onde `<nome>` é o nome do atributo/método, e `<tipo>` é seu tipo ou classe.

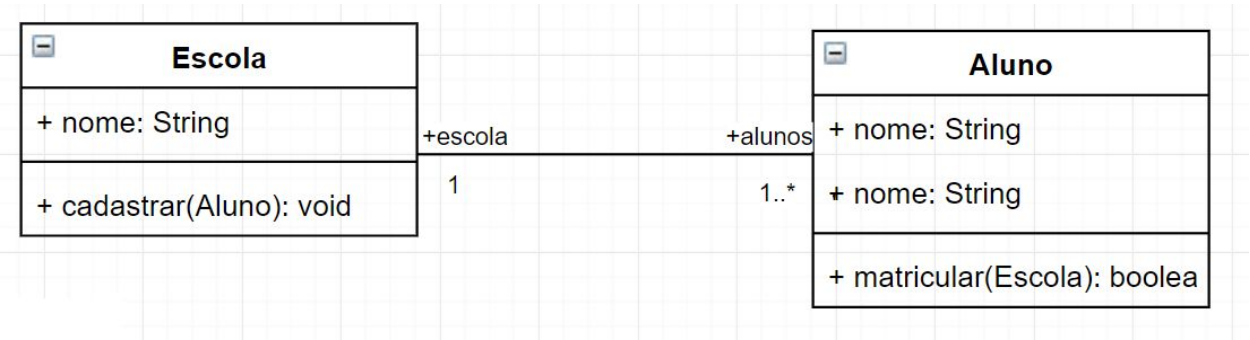



Exemplo de representação da classe **HistoricoCompra**.

**Associação**

Quando os objetos de uma classe necessitam se comunicar com objetos de outra (ou até da mesma) classe, temos uma associação entre elas. Nesse caso, indicamos a associação com um traço ligando as classes. Essa ligação pode ainda indicar os nomes de papéis em cada extremidade, a cardinalidade, direção e restrições.

Vamos analisar o exemplo das classes Escola e Aluno. Elas precisam se conversar para atingir o objetivo do sistema (cadastro dos alunos e gerenciamento das aulas). Enquanto o aluno está matriculado apenas em uma escola, a escola pode ter um ou mais alunos. O uso de papéis evidencia, em geral, que essas informações estão armazenadas nos próprios objetos na forma de atributos.




Exemplo de associação com cardinalidade e papéis

Como exercício, vamos criar o diagrama de classes do nosso sistema bancário, denotando os métodos, atributos, visibilidade, papéis e cardinalidade.