OO – Encapsulamento

Este material foi construído pelo Professor **Valdemar Lorenzon Junior** (Lorenzon), trazendo adaptações de material de própria autoria ao longo de anos de dedicação e trabalhos com pesquisa, desenvolvimento e docência com OO, recuperação de cursos certificados SUN e Oracle.

É um material teórico/prático enriquecido com atualizações de conteúdo e certificações de updates, podendo trazer alguma incompatibilidade em algum momento que pode ser corrigida quando observada. O membro pode ser acessado por classes no mesmo pacote.

Paralelo ao material didático teórico há o programa Apostila.java em anexo final, que é um conteúdo prático em código fonte Java e na sequência lógica de apresentação do conteúdo didático. Em sala de aula será trabalhado com o conteúdo prático orientando o conteúdo teórico.

Sumário

OO – Encapsulamento 1
Uma classe e o Objeto 3
O Encapsulamento 3
Visibilidade de classes 5
Visibilidade para atributos e métodos: 6
Atributos 9
Métodos 10
Modificadores adicionais 11
Modificadores adicionais de Classes 11
Modificadores adicionais de atributos 12
Modificadores adicionais de métodos 12
Inicialização de Atributos 13
Classes sem Instância 13
Reusabilidade 14
Manutenibilidade 15
Exercícios de fixação 16

Uma classe e o Objeto

Antes de se definir um objeto, é necessário que seus componentes (atributos e métodos – variáveis membro e funções membro) sejam especificados.

A classe é o mecanismo que provê meios para encapsulamento e ocultação de informação necessária para definição de um objeto (ou, mais precisamente de uma categoria de objetos).

Uma classe (simples) em JAVA é declarada utilizando a palavra reservada **class** da seguinte maneira:

```
class <NomeClasse>  {

     <atributos>;
     <construtores>;
     <getters e setters>;
     <comportamento>;

}
```

Procure construir sua classe seguindo a ordem acima de orientação de ordem: atributos, construtores, getters e setters e funções membro de comportamento.

O Encapsulamento

Quando executamos um método, podemos dizer que estamos enviando uma mensagem. Isso pode ser entendido como uma chamada de função, porém a função pertence a alguém, este alguém é um objeto ou uma classe.

O modo como ele vai cumprir a ação para responder a esta mensagem é transparente ao usuário, isto é, ele não precisa necessariamente saber como ele vai executá-la. Por exemplo, quando pisamos no acelerador, estamos enviando uma mensagem a uma instância da classe Automóvel, um objeto, solicitando que ele execute o seu método acelerar e ninguém precisa entender de mecânica para realizar essa operação. A essa transparência do objeto começamos a compreender o encapsulamento.

Encapsulamento: Proteção que o objeto dá as suas propriedades (atributos) e métodos de maneira que nenhum agente externo tenha acesso a elas sem solicitá-las.

Cada linguagem implementa o encapsulamento de maneira diferente. A mais comum é a possibilidade de definir métodos e propriedades através de **escopo** para os mesmos.

Este escopo define a **visibilidade** do método em relação ao ambiente onde o mesmo está inserido.

O encapsulamento é dito como uma muralha em torno do objeto, impedindo de o cliente enxergar os detalhes de sua implementação, o que denominamos ocultação de informação. O objetivo é separar o usuário da classe de seu implementador.

No entanto o encapsulamento não é apenas este simples conceito, queremos mais do que apenas ocultação de informação, nós precisamos de objetos que sejam independentes da aplicação.

Portanto, uma visão mais completa do encapsulamento deve incluir: Uma interface abstrata e bem definida, que permita ao cliente usar o objeto ignorando a forma como está organizada a sua estrutura interna e como as operações são realizadas.

Para que isso seja cumprido, apresentamos as seguintes regras do encapsulamento:

Regra 1: Apenas os métodos devem ser capazes de modificar o estado interno de um objeto;

Regra 2: A resposta a uma mensagem deve ser completamente determinada pelo estado interno do objeto receptor.

De acordo com a regra 1, não deve ser possível acessar o valor armazenado em um atributo de instância diretamente. Toda mudança de estado interno de um objeto deve ser consequência de uma mensagem – protegendo assim o estado interno do objeto.

Isto não significa que toda mensagem deva mudar o estado interno de um objeto. Um método pode apenas retornar um valor obtido a partir do conteúdo dos atributos de um objeto, sem modificar nenhum deles.

De acordo com a regra 2, se duas instâncias de uma mesma classe possuem o mesmo estado interno, então devem responder a mensagens iguais de forma idêntica. Isto garante que a resposta a uma mensagem não dependa de fatores externos ao objeto.

Portanto, não devemos acessar variáveis globais da aplicação no interior de um método. A existência de estruturas globais compartilhadas causa todos os problemas de interdependência entre módulos, que é justamente o que se quer evitar.

O escopo mencionado deverá ser obtido através da aplicação de palavras reservadas da linguagem de programação que definem a visibilidade do método.

Visibilidade de classes

A tabela a seguir define a visibilidade através de seus modificadores de acesso e em relação ao ambiente classes:

```
public class....
```

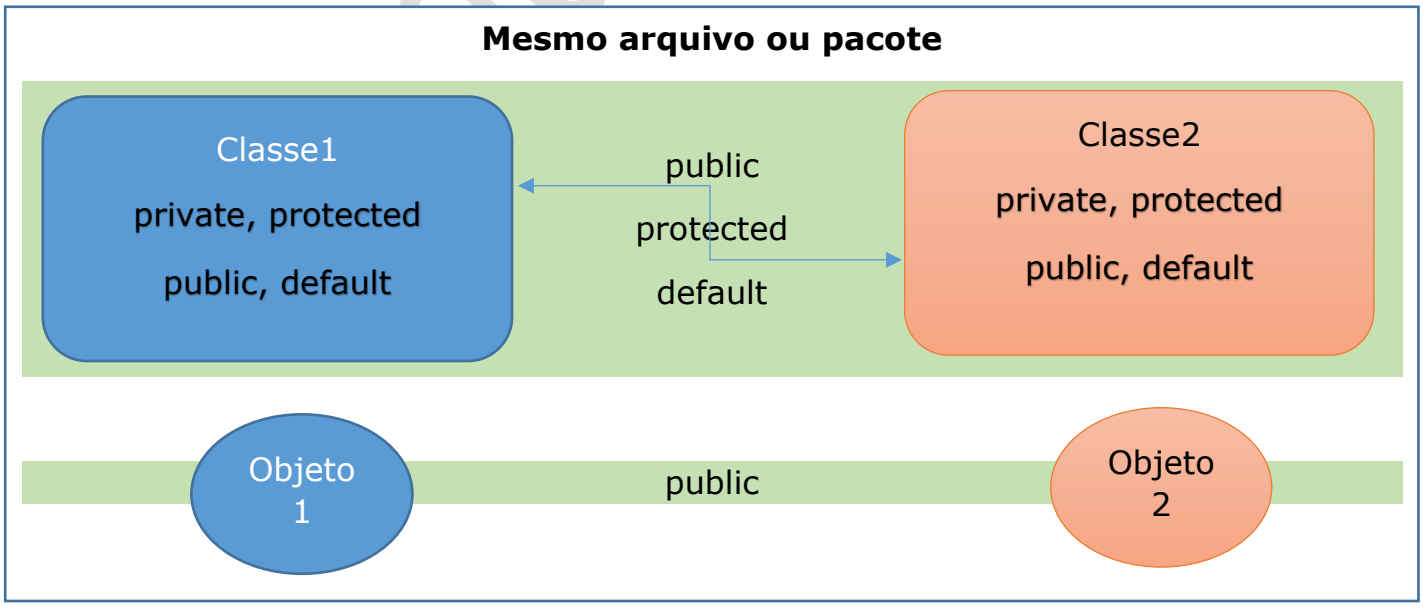
Escopo	Palavra reservada	Definição	Ambiente	Símbolo
Default		A classe pode ser acessada pelo mesmo pacote ou arquivo onde está declarada.	Pacote	
Público	***public***	A classe é pública, pode ser acessada dentro e fora do pacote onde foi criada. Uma classe pública exige ser única classe pública no arquivo onde está definida. Uma classe interna pode ser public.	Objeto	+
Privado	***private***	A classe privada pode existir se for uma classe interna, ou seja, criada e usada dentro de uma outra classe ou método. Será demonstrado no material de pacotes.	Classe	-

Visibilidade para atributos e métodos:

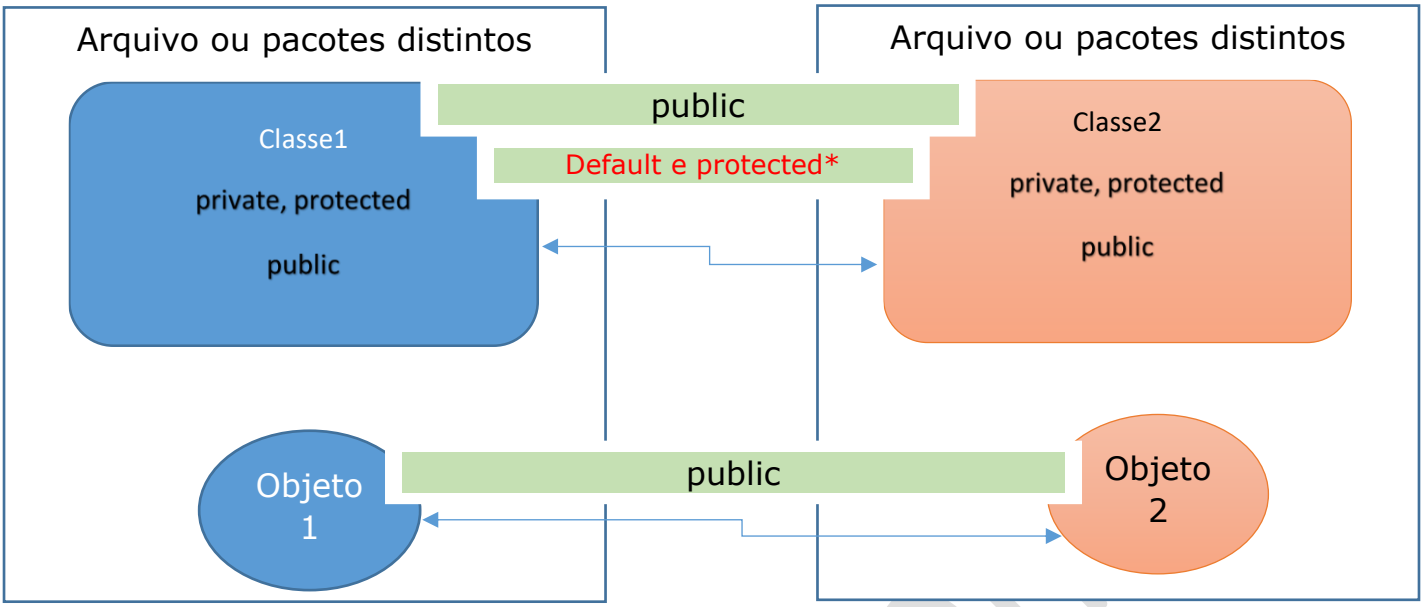
Escopo	Palavra reservada	Definição	Ambiente	Símbolo
Default		O item pode ser acessado pelo mesmo pacote ou arquivo onde está declarada.	Classe e Pacote	
Público	***public***	O item é público, pode ser acessado dentro e fora do pacote onde foi criado.	Classe, pacote e Objeto	+
Privado	***private***	O item privado só pode ser acessado na classe onde foi criado. Exige-se gettres e setters se necessário para acesso fora da classe (estes públicos).	Classe	-
Protegido	***protected***	O item poderá ser acessado pelas classes pertencentes ao mesmo arquivo, pacote ou derivações por herança.	Pacote, Classe pai e derivadas	#

Um esboço para entendimento do caso pode ser observado como a seguir:

Para entender melhor esta ideia imagina-se 2 cenários:



Public, protected e default neste cenário são públicos entre classes.



* Apenas se, Classe2 é filha de Classe 1, exceto se mesmo pacote, assim então como default continua valendo para o mesmo arquivo ou pacote, mas evita-se.

Os objetos não enxergam nada diretamente das classes, a não ser a partir de referências ou se as classes forem estáticas – abordado a seguir neste material.

Para aplicar o encapsulamento, geralmente declaramos os atributos de uma classe como **private** e fornecemos métodos **public** para acessá-los (**getters**) e modificá-los (**setters**).

Isso permite que a classe mantenha o controle sobre como seus dados são acessados e modificados, podendo, por exemplo, validar dados antes de modificá-los.

Nem todo atributo precisa de getter e setter.

Um setter é exigido quando para o atributo há uma validação a ser garantida ou a alteração do atributo interfere no estado interno do objeto alterando novos atributos.

Um getter pode ser um retorno formatado ou um conjunto de informações do estado interno do objeto, não apenas o valor isolado do atributo.

Tanto getters e setters podem ser desenhados com visibilidade public, protected ou privite.

Dentro da classe, não importará a visibilidade, tudo é acessível, mas a abstração deve levar em conta as regras do encapsulamento.

Em próprios construtores, para alimentar o estado interno do objeto já se deve usar os setters e também em todos os métodos ou funções membro da classe, garantindo coesão de código para garantir as regras do encapsulamento.

A seguir um exemplo de classe com atributos e o uso de getters e setters:

```
public class Pessoa {
            private String    nome;  // Atributo encapsulado
            private int       idade; // Atributo encapsulado
            // Construtor
            public Pessoa(String nome, int idade) { //sem retorno
                   setNome(nome);
                   setIdade(idade);
            }
            // Getter para idade
            public int getIdade() {
                   return idade;
            }
            // Getter para nome
            public String getNome() {
                   return nome;
            }
            // Setter para nome
            public void setNome(String nome) {
                   this.nome = nome;
            }
            // Setter para idade com validação
            public void setIdade(int idade) {
                   if (idade >= 0) {
                       this.idade = idade; //
                   } //else ou tratativas de exceções
            }
        }
```

Classe de aplicação

```
public class App {
     public static void main (string s[]) {

          Pessoa p1;
          p1 = new Pessoa ("Junior", 23);
          System.out.println (p1.getNome());
          p1.idade = -1; // Impossível, por quê?
          P1.setIdade (-1); //Argumento será validado como inválido
}
```

Atributos

A sintaxe para a criação de um atributo dá-se a seguir:

```
[especificador de acesso]*
[modificador adicional]**
<tipo> <nome> [= inicialização]**;

* Opcional, mas deve-se praticar pois é encapsulamento
** Opcional.
```

As declarações de atributos podem (às vezes devem) incluir inicializações. Os modificadores de atributos podem ser assim descritos:

As variáveis de classe: Um único exemplar para todos os objetos da classe. São utilizadas sem instanciação, ou seja, através da classe. São declarados com modificador **static**.

As variáveis de instância: Cada objeto tem sua cópia. Pode ser usado apenas a partir de um objeto, são o padrão, sem uso de modificador.

Métodos

A sintaxe para criação de um método é dada a seguir:

```
[especificador de acesso]*
[modificador adicional]**
<tipo de retorno>
<nome> (<tipo1> <argumento1>, ... <tipon> < argumentoN>***) {
       <instruções  e declarações de variáveis locais>;
}


* Opcional, mas deve-se praticar pois é encapsulamento.
** Opcional.
*** Opcional, porém poucos métodos seriam sem parâmetros.
```

A passagem de parâmetros, argumentos, é apenas por valor/cópia. Exceto se o parâmetro for uma referência para um objeto, o próprio objeto será modificado.

Usamos este artifício para modificar o valor de uma variável de algum tipo primitivo usa-se uma classe wrapper, inclusive interna, por exemplo.

O retorno de um método pode ser um tipo primitivo ou uma referência (Classe). A recursividade é aceita sem problemas.

A sobrecarga de métodos pode ser feita da seguinte forma:

- Métodos de uma mesma classe podem ter o mesmo nome desde que suas listas de parâmetros/argumentos sejam diferentes em número ou em tipos;
- Os métodos não podem diferir apenas em tipos de retorno.

Exemplo:

```
class Classe {
    public  void    umMetodo  (double x) {...}
    public  void    umMetodo  (int x)    {...}    // é
    public  double  umMetodo  (double x) {...}    // não é
}
```

Modificadores adicionais

A seguir são apresentados os modificadores que são combinados com os modificadores de acesso, visibilidade. São combinados com os outros tipos de itens que compõem uma classe, método ou atributo.

Modificadores adicionais de Classes

Modificador	Como a classe pode ser usada
public	Pode ser usada por qualquer pacote.
abstract	Classe incompleta; não pode ser instanciada (i.e., não pode ter objetos). Para ser útil, ela deve ser derivada, o abstract precisa ser codificado. Necessário conhecimentos de herança e polimorfismo.
final	Não pode ter nenhuma derivação, não pode ser a classe pai de ninguém.
	Pode ser sem e será visível apenas no pacote aonde ela é definido.

Modificadores adicionais de atributos

Modificador	Significado
static	Existe apenas um exemplar que é compartilhado por todos os objetos da classe e existe mesmo que a classe não tenha nenhum objeto.
final	Deve ser inicializado, mas não pode alterado posteriormente. Uma constante! • Se ***static***: deve ser inicializado em sua declaração; • Se não static: pode ser incializado fora ou dentro de um construtor.
transient	*Previne a serialização de um atributo. Quando um objeto é serializado, os campos marcados como transient não são incluídos no processo de serialização.*
volatile	*Indica que um atributo é acessado por múltiplas threads. O volatile garante que qualquer thread que lê o campo receberá a versão mais recente escrita por qualquer thread. Todas as leituras e escritas são feitas diretamente na memória principal, garantindo a visibilidade das mudanças entre as threads.*

Modificadores adicionais de métodos

Modificador	Significado
abstract	Não possui implementação. Não é necessário escrever corpo de código para o método. As derivações da classe serão obrigadas a implementar ou não poderá ter-se objetos a partir da derivação até que não se escreva código para o método em questão.
final	O método não pode ser oculto nem sobrescrito.
static	Pode ser acionado através de uma classe, sem a necessidade de uma instância da mesma. Pertence à classe.
synchronized	*Bloqueia o objeto enquanto não retorna; usado com threads.*
native	*Implementado com código dependente de plataforma (i.e., não portável).*

Inicialização de Atributos

Se um atributo é inicializado em sua declaração e ele é	***A atribuição de valor ocorre***
static	Exatamente uma vez
Não static	Sempre que um objeto é criado

A inicialização de variável de classe (i.e., um campo static) não pode envolver nenhuma variável de instância (i.e., um campo não static), this ou super.

Classes sem Instância

Às vezes não faz sentido permitir que uma classe tenha instâncias. Por exemplo, não faz sentido a existência de instâncias da classe **Math** pois os métodos nesta são do tipo static. Pode-se impedir que uma classe tenha qualquer instância declarando um construtor default como sendo private.

Reusabilidade

A proposta da reusabilidade é a diminuição dos custos dos projetos através do reaproveitamento dos componentes, tornando o desenvolvimento mais rápido sem diminuir os padrões de qualidade.

Com o encapsulamento, o funcionamento de cada classe é independente do resto da aplicação. Os objetos se tornam mais reusáveis, isto é, eles podem ser reaproveitados diversas vezes sem a necessidade de realizar mudanças na implementação.

Um objeto reusável possui valor próprio, enquanto um módulo dependente da aplicação não possui valor algum fora daquela aplicação. Além disso, projetar para reuso permite a adoção de testes com critérios muito mais rígidos e a elaboração da documentação segundo padrões mais elevados.

Dessa forma, classes de objetos projetadas para reuso permitem o aumento de produtividade conjuntamente com uma elevação qualitativa, devido ao fato de os objetos já terem sido exaustivamente testados e documentados.

No entanto, para alcançar essa meta plenamente, um alto grau de planejamento é essencial. Uma classe planejada para ser reusável deve ser mais flexível e, portanto, mais complexa do que seria uma solução convencional. Isso ocorre porque não desejamos um objeto que satisfaça os requisitos de uma aplicação particular, mas sim uma classe genérica encapsulando funcionalidade útil em uma grande diversidade de contextos.

Esse ganho de flexibilidade implica inevitavelmente um aumento da complexidade do objeto em nível de implementação/, o que significa que o projetista deve dedicar uma parcela considerável de tempo ao planejamento da nova classe se realmente deseja um resultado satisfatório.

Portanto, para a adoção plena da metodologia de programação orientada para objetos, não basta assimilar novos conceitos e aprender a utilizar novas ferramentas, é preciso também abandonar certos hábitos muito bem estabelecidos, do tipo “sentar em frente ao computador e deixar a imaginação correr solta”.

Para que classes de objetos possam ser efetivamente reusadas é preciso que elas estejam disponíveis em unidades (arquivos, units ou bibliotecas), e não junto ao restante do código da aplicação. Cada Unidade deve ser dedicada a uma classe ou a um conjunto de classes muito relacionadas. Não se deve juntar diversas classes em uma mesma Unidade apenas porque elas são utilizadas na mesma aplicação. A proposta do reuso é justamente desenvolver classes genéricas que sejam independentes de aplicação.

Unidades ou bibliotecas devem ser distribuídas aos clientes já compiladas reforçando o encapsulamento. A documentação necessária para que o cliente possa usar as classes é constituída basicamente da interface e da descrição das ações que cada método executa. Não é preciso revelar nenhum detalhe da implementação.

Manutenibilidade

Um benefício adicional do encapsulamento é o aumento da manutenibilidade das aplicações. Um sistema é mais ou menos manutenível dependendo do impacto que as modificações podem causar em seu funcionamento.

Graças ao encapsulamento, os detalhes de implementação permanecem ocultos. Assim, o projetista mantém a liberdade de fazer modificações da parte privada do objeto, provavelmente visando a um aumento de eficiência com a garantia de não prejudicar nenhum de seus clientes.

A forma como são projetadas as interfaces faz com que qualquer necessidade de mudança tenha um efeito mínimo e muito bem localizado, impedindo a propagação descontrolada de alterações por todo o sistema, que é o pesadelo de todo o programador.

Vale lembrar que as melhorias só chegam aos clientes caso os objetos distribuídos como unidades pré-compiladas o que não aconteceria se simplesmente fossem trechos de código copiados para dentro das aplicações.

Note que as duas regras do encapsulamento devem ser respeitadas para que se obtenha a unidade a mudanças. A regra 1 garante que as modificações da parte interna do objeto não se propaguem através do sistema, no entanto, caso a segunda regra seja desrespeitada, uma mudança externa ao objeto poderia gerar resultados imprevistos, com consequências desastrosas.

É importante também ressaltar a diferença entre manutenção corretiva e evolução. A manutenção corretiva se caracteriza por mudanças realizadas esporadicamente visando eliminar erros encontrados no sistema. A evolução é contínuo desenvolvimento para atingir os requisitos dos usuários, que estão em constante mutação.

Assim, a evolução dos sistemas é inevitável, e a adaptação a mudanças, ou manutenibilidade, é uma necessidade fundamental. O encapsulamento permite a construção de sistemas que possam ser facilmente modificados durante seu tempo de vida, vem sendo o desafio da sobrevivência às mudanças.

Exercícios de fixação

Vamos revisar nossas soluções dos estudos de casos propostos na abstração aplicando agora conceitos de encapsulamento.

É importante o pensamento de proteger e responder as mensagens apenas a partir do estado interno do objeto, sem dependências externas da classe.

É importante imaginar que a alteração de um comportamento pode incidir em vários atributos do estado interno do objeto.

1- Aplicar os conceitos de encapsulamento para o estudo de caso 1 de Pessoa Física e Jurídica com associação a Endereço e PessoaJurídica com associação à Pessoa Física;

2- Desenvolva os estudos de caso 2 e 3, descartando suas criações sem encapsulamento e abstraia novamente pensando em encapsulamento e garantia coesa do estado interno do objeto a partir e observações de que uma alteração pode incidir em mudança no estado interno do objeto e que a proteção do mesmo deve ser sempre garantida.