

Encapsulamento



Prof. Jeferson Souza, MSc. (thejefecomp)

jeferson.souza@udesc.br



UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

JOINVILLE
CENTRO DE CIÊNCIAS
TECNOLÓGICAS

Afinal, o que é Encapsulamento?

Definição de Encapsulamento

Encapsulamento não é:

- ▶ uma forma de embalar medicamentos;
- ▶ uma forma de empacotar programas;
- ▶ uma forma de virar borboleta;
- ▶ uma forma de proteger-se do frio :-D.

Final, o que é Encapsulamento?

Definição de Encapsulamento

Portanto, **Encapsulamento** pode ser definido como uma característica intrínseca presente nas linguagens de programação, onde seus elementos (e.g. variáveis, atributos, métodos, funções, classes, etc...) fazem parte de um contexto bem definido, o qual estabelece os limites da existência e do acesso aos referidos elementos.

Afinal, o que é Encapsulamento?

Definição de Encapsulamento

Encapsulamento também pode ser visto como uma boa prática de programação, a definir e restringir a existência e o acesso a elementos específicos, por meio de sua declaração em um contexto escolhido.

Pense na Violação do Encapsulamento Como...

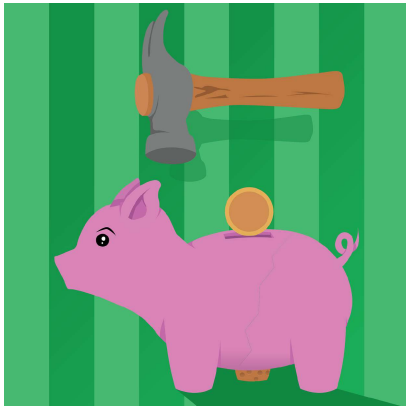


Imagem: Pixabay (<https://pixabay.com>)

A violação do **Encapsulamento** pode ser representada como a quebra do porquinho sem permissão.

Pense no Acesso de um Elemento Encapsulado Como...



Imagem: Pixabay (<https://pixabay.com>)

A abertura consciente e autorizada do porquinho para acesso às suas moedinhas por meio da interface exposta.

Qual é a definição de Escopo?

Definição de Escopo

Dentro do **Encapsulamento** o **Escopo** representa o contexto e as restrições associadas ao mesmo. Diz-se que um elemento está dentro do **Escopo** quando encontra-se dentro do contexto de sua definição; e diz-se que um elemento está fora do **Escopo** caso contrário.

Qual é a definição de Escopo?

Estou fora do Escopo, então...

Diz-se ainda que um elemento a tentar ser acessado fora do **Escopo** representa a violação do **Encapsulamento**.

Escopo de Variáveis na Orientação à Objetos

O que é uma variável local?

Uma **variável local** é uma variável definida dentro de um método.

Escopo de Variáveis na Orientação à Objetos

O que é uma variável local?

Uma **variável local** é uma variável definida dentro de um método.

O que é uma variável de instância?

Uma **variável de instância** é uma variável que representa um atributo da classe, o qual é instanciado juntamente com o objeto para poder ser utilizado.

O que é uma variável de classe?

Uma **variável de classe** é uma variável (i.e. atributo) que é partilhada por múltiplas instâncias da classe, a implicar que sua instanciação não ocorre com cada um dos objetos.

Exemplo de Variável Local em Java

```
public void executarAcao(){
```

```
    Boolean iniciaAcao = true; //A variável iniciaAcao representa uma  
    variável local.
```

```
}
```

Exemplo de Variável de Instância em Java

```
public class Pessoa {
```

```
    String nome; //A variável nome representa uma variável de instância,  
    i.e., um atributo da classe.
```

```
}
```

Exemplo de Variável de Classe em Java

```
public class Quadrado {
```

```
    static Double AREA_MAXIMA = 100.00; // A variável  
    AREA_MAXIMA representa uma variável de classe, i.e., um atributo  
    estático da classe.
```

```
}
```

Exemplo de Encapsulamento de Variáveis em múltiplas camadas em Java

```
public void executarAcao(){
```

```
    Boolean iniciaAcao = true; //A variável iniciaAcao representa uma  
    variável local.
```

```
    {
```

```
        boolean variavelEncapsuladaEmCamadas = true;
```

```
    }
```

```
    //Neste ponto a variável variavelEncapsuladaEmCamadas está fora de  
    escopo, mesmo a estar dentro do método.
```

```
}
```


Modificadores de Acesso em Java

O que são os Modificadores de Acesso?

Os modificadores de acesso estabelecem restrições ao *Encapsulamento* dos elementos presentes na linguagem. Cada modificador de acesso permite, ou não, o acesso de um elemento fora de seu escopo de definição.

Pense nos Modificadores de Acesso Como...



Imagem: Pixabay (<https://pixabay.com>)

Modificador de Acesso como a autorização para acessar as moedinhas do porquinho por meio da interface exposta.

Pense nos Modificadores de Acesso Como...



Imagem: Pixabay (<https://pixabay.com>)

Modificador de Acesso como o estado do pote de biscoitos. O pote de biscoitos representa o **Encapsulamento**, enquanto que o **Modificador de Acesso** define se o pote está **Fechado** para quem não tem autorização, ou **Aberto** caso contrário.

Tipos de Modificadores de Acesso em Java

Podem ser de quatro tipos

- ▶ **public**: elemento pode ser acessado por qualquer classe;
- ▶ **private**: elemento pode ser acessado somente dentro da classe;
- ▶ **protected**: elemento pode ser acessado de classes do mesmo pacote e/ou subclasses;
- ▶ **Acesso padrão (privado no pacote)**: elemento pode ser acessado por qualquer classe ou subclasse dentro do mesmo pacote. Não existe modificador de acesso, basta omitir.

Exemplo de acesso a Elemento com Modificador de Acesso (**public**)

```
public class Principal {  
    public static void main(String ...args){  
        Animal animal = new Animal();  
        animal.setDescricao("Golfinho-Chileno (Cephalorhynchus eutropia)");  
    }  
    public class Animal {  
        private String descricao;  
        public void setDescricao(String descricao){  
            this.descricao = descricao;  
        }  
    }  
}
```

Os método público **setDescricao()** da classe **Animal** está a ser acessado dentro da classe **Principal**.

Exemplo de acesso a elemento com Modificador de Acesso (**private**)

```
public class Pessoa {  
    private String nome;  
    public String getNome(){  
        return this.nome;  
    }  
    public void setNome(String nome){  
        this.nome = nome;  
    }  
}
```

O atributo **nome** está a ser acessado dentro da classe nos métodos **getNome()** e **setNome()**.

Exemplo de acesso a Elemento com Modificador de Acesso (**protected**)

```
public class Principal {  
    public static void main(String ...args){  
        Pessoa pessoa = new Pessoa();  
        pessoa.nome = "Odete";  
        System.out.println(pessoa.nome);  
    }  
    class Pessoa {  
        protected String nome;  
    }  
}
```

O atributo **nome** da classe **Pessoa** está a ser acessado dentro da classe **Principal** (acesso **protected**).

Exemplo de acesso a Elemento sem Modificador de Acesso [**Acesso padrão (privado no pacote)**]

```
public class Principal {
    public static void main(String ...args){
        Carro carro = new Carro();
        carro.denominacao = "Fusca Bola";
        System.out.println(carro.denominacao);
    }
}

class Carro {
    String denominacao;
}
```

O atributo **denominacao** da classe **Carro** está a ser acessado dentro da classe **Principal** [**Acesso padrão (privado no pacote)**].

Classes Aninhadas (Nested Classes) em Java

O que são Classes Aninhadas (Nested Classes)?

As **Classes Aninhadas (Nested Classes)** são classes que podem ser declaradas dentro de outras classes. São de quatro tipos:

- ▶ **Classe Membro (Member Inner Class):** são declaradas como variáveis de instância da classe;
- ▶ **Classe Local (Local Inner Class):** são declaradas dentro de métodos;
- ▶ **Classe Anônima (Anonymous Class):** tipo especial de classe local que não possui nome;
- ▶ **Classe Estática (Static Nested Class):** são declaradas como variáveis de classe.

Exemplo de Classe Membro (Member Inner Class)

```
public class Veiculo {  
    private String nome;  
    private class Interior {  
        private Boolean comercial;  
        public void setComercial(Boolean comercial){  
            this.comercial = comercial;  
        }  
    }  
    private Interior interior;  
    public void criaInterior(Boolean comercial){  
        this.interior = new Interior();  
        this.interior.setComercial(comercial);  
    }  
}
```

Neste exemplo a classe *Interior* é uma **Classe Membro** da classe **Veiculo**.

Exemplo de Classe Local (Local Inner Class) [Boyarsky&Selikoff, 2015]

```
public class Outer {  
    private int tamanho = 5;  
    public void calcular() {  
        final int largura;  
        class Inner {  
            public void multiplicar(){  
                System.out.println(tamanho * largura);  
            }  
        }  
        Inner inner = new Inner();  
        inner.multiplicar();  
    }  
    public static void main(String[] args){  
        Outer outer = new Outer();  
        outer.calcular();  
    }  
}
```

Neste exemplo a classe *Inner* é uma **Classe Local** do método **calcular()**.

Exemplo de Classe Anônima (Anonymous Class) [Boyarsky&Selikoff, 2015]

```
public class AnonInner {  
    abstract class VendaSomenteHoje {  
        abstract int descontoDollar();  
    }  
    public int admissao(int precoBase){  
        VendaSomenteHoje venda = new VendaSomenteHoje(){  
            int descontoDollar() { return 3;}  
        };  
        return precoBase - venda.descontoDollar();  
    }  
}
```

Neste exemplo a declaração de uma subclasse da classe *VendaSomenteHoje* é uma **Classe Anônima** do método **admissao()**.

Exemplo de Classe Estática (Static Nested Class)

```

public class Principal {
    private static class Carro {
        private String denominacao;
    }

    public static void main(String ...args){
        Carro carro = new Carro();
        carro.denominacao = "Fusca Bola";
        System.out.println(carro.denominacao);
    }

```

Neste exemplo a classe *Carro* é uma **Classe Estática** da classe **Principal**.

A Magia dos Métodos em Java

Existe muita coisa além do **public static void main()**...

Não é só do método **public static void main()** que vive o mundo que descreve o comportamento dos programas em Java. Existe muito mais além deste método principal e crucial para a execução de programas. Portanto, vamos desbravar o mundo dos métodos e sua relação com o encapsulamento.

A Magia dos Métodos em Java - Continuação

O desbravar da assinatura...

Um método Java pode ser definido da seguinte forma:

```
< mod_acesso > < opt_classif > < retorno > < nome > ( < lista_param > ) < opt_exceção > { < corpo > }
```

A Magia dos Métodos em Java - Continuação

O desbravar da assinatura...

Onde:

`< mod_acesso >` é o modificador de acesso;

`< opt_classif >` é o classificador (opcional) [PS: podem existir vários separados por espaços.];

`< retorno >` é o tipo de retorno;

`< nome >` é o nome do método;

`< lista_param >` é a lista de parâmetros;

`< opt_exceção >` é a exceção que pode ser disparada pelo método (opcional) [PS: podem existir várias separadas por vírgula];

`< corpo >` é o corpo do método.

A Magia dos Métodos em Java - Continuação

O desbravar da assinatura...

Os elementos que caracterizam a assinatura singular de cada um dos métodos são: o **nome do método** (`< nome >`) e a **lista de parâmetros** (`< lista_param >`). A mudança de todos os outros elementos não caracteriza uma mudança de sua assinatura. A implicação desta característica está relacionada com o conceito de Polimorfismo (Veremos maiores detalhes juntamente com o conceito de Herança).

A Magia dos Métodos em Java - Continuação

Relação com encapsulamento

Cada método representa um contexto único dentro de um contexto mais abrangente definido pela classe^{a,b}. Portanto, a declaração de um método, por si só, representa um elemento a fornecer um encapsulamento adicional à definição e execução dos programas.

^aOu pela Interface, tal como veremos quando os conceitos de Herança e Polimorfismo forem abordados.

^bEnumerados e Anotações, que são elementos presentes na linguagem Java, são tipos especiais de classes e interfaces, respectivamente.

A Magia dos Métodos em Java - Continuação

Exemplo de método 1

```
public String getNome () {
return this.nome; //Corpo do método
}
```

Onde:

public é o modificador de acesso;

String é o tipo de retorno;

getNome é o nome do método;

O método não recebe parâmetros, e portanto não existe nada entre os parênteses [()]. As chaves [{}] são obrigatórias para delimitar o início e o fim do corpo do método.

A Magia dos Métodos em Java - Continuação

Exemplo de método 2

```
public final void setNome (String nome) {  
this.nome = nome; //Corpo do método  
}
```

Onde:

public é o modificador de acesso;

final é o classificador;

void é o tipo de retorno (i.e. sem retorno);

setNome é o nome do método;

String nome é a lista de parâmetros (com um só elemento).

A Magia dos Métodos em Java - Continuação

Exemplo de método 3

```
public final void sorrir () throws POOException {  
    //Dê um sorriso :-).  
}
```

Onde:

public é o modificador de acesso;

final é o classificador;

void é o tipo de retorno (i.e. sem retorno);

sorrir é o nome do método;

throws é a palavra reservada que indica o disparo da exceção **POOException** (Acontece quando vocês não dão um sorriso com as minhas piadas o/).

A Magia dos Métodos em Java - Continuação

O desbravar do Varargs...

Exemplo: **public static void** main (String ...args) { }

Alguns de vocês já podem ter se perguntado: Afinal, para que servem os três pontinhos (...) na lista de parâmetros dos métodos?

Os três pontinhos (...) indicam a presença de uma construção da linguagem chamada de **Varargs** (i.e. argumentos de tamanho variável). A **Varargs** comportam-se como se fosse um *array* (i.e. vetor), mas possui uma característica peculiar: só pode ser utilizada se for o último elemento da lista de parâmetros de um método. No exemplo acima o tradicional **String[] args** pode ser substituído pela **Varargs** por ser um parâmetro singular.

A Magia dos Métodos em Java - Continuação

Exemplo de método com Varargs

```
public final void recebeFruta (String nomeVendedor, String
...frutas) {
```

//O acesso a cada uma das frutas pode ser feito pelo índice do array frutas, a começar pelo valor inicial de zero (0). O tipo associado à Varargs pode ser qualquer tipo válido na linguagem de programação Java, a incluir os tipos (i.e. classes) definidos pelo desenvolvedor.

```
}
```

Ao realizar a chamada:

```
recebeFruta(“Jeferson”, “Banana”, “Laranja”, “Limão”);
```

```
frutas[0] -> “Banana”;
```

```
frutas[1] -> “Laranja”;
```

```
frutas[2] -> “Limão”.
```

Pacotes em Java

Afinal, aonde está o meu pacote?

Um **pacote** é um elemento em orientação a objetos que permite inserir um nível adicional de encapsulamento aos programas. Em Java, os pacotes podem ser utilizados por meio da palavra reservada **package**. Entretanto, existe um “pacote padrão” (*default package*), e portanto mesmo sem realizar a declaração explícita de um pacote, a definição da classe acaba por fazer parte do “pacote padrão”.

Pense nos Pacotes Como...



Imagem: Pixabay (<https://pixabay.com>)

Pacote como um saco de papel, daqueles utilizados em supermercado. Pode-se ter elementos (i.e. Classes e Interfaces) e outros pacotes dentro. Cada pacote representa um nível de encapsulamento adicional, a caracterizar unicamente os elementos internos.

Criação de Pacotes em Java

Um Pacote representa um diretório no sistema de arquivos...

Em termos do sistema operacional, a criação de um **pacote** representa a simples criação de um diretório no sistema de arquivos. Não existe magia, nem muito menos esquizofrenia. Basta utilizar a simples criação de um diretório para começar a “empacotar”.

Criação de Pacotes em Java - Continuação

Exemplo de classe com pacote 1

package poo; //Deve ser a primeira linha do arquivo.

```
public class Pessoa {
    private String nome;
    public String getNome(){
        return this.nome;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
}
```

Criação de Pacotes em Java - Continuação

Exemplo de classe com pacote 2

package org.poo; //Deve ser a primeira linha do arquivo.

```
public class Carro {
    private String nome;
    public String getNome(){
        return this.nome;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
}
```

Criação de Pacotes em Java

Nas declarações de pacotes...

Cada nível (i.e. **pacote**) é separado por um ponto (.), o qual indica a entrada em um novo diretório no sistema de arquivos. Portanto, a declaração:

package *poo*;

refere-se ao diretório *poo* no sistema de arquivos. Enquanto que a declaração:

package *org.poo*;

refere-se aos diretórios *org/poo* no sistema de arquivos.

Nome dos Elementos com Pacote

O ponto não é usado só na declaração do pacote...

Elementos (i.e. classes e interfaces^a) são identificados unicamente pelo seu nome composto, o qual inclui o nome de todos os pacotes que fizerem parte de sua hierarquia de pacotes. Nos exemplos anteriores as classes *Pessoa* e *Carro* são identificadas unicamente da seguinte forma:

poo.Pessoa

org.poo.Carro

Portanto se existirem duas classes com o mesmo nome, mas com declaração de pacote distinta, essas classes são vistas pela Máquina Virtual Java como classes distintas.

^aInterfaces serão introduzidas juntamente com os conceitos de Herança e Polimorfismo.

O uso do **import**

Abreviar é sempre bom...

A palavra reservada **import** serve para que não seja necessário utilizar o nome composto cada vez que um elemento (i.e. classe ou interface) for utilizado no código-fonte. É possível “importar” o pacote todo, ou uma classe específica para que somente o seu nome simples (nome declarado no arquivo .java) possa ser utilizado.

Exemplo de uso do **import** - Classe Específica

```
package poo;
import java.util.Scanner;
public class Principal {
    public static void main(String ...args){
        Scanner scanner = new Scanner(System.in);
    }
}
```

Caso o **import** não fosse realizado, a classe Scanner só poderia ser utilizada com o seu nome composto java.util.Scanner.

Exemplo de uso do **import** - Pacote Completo

```
package poo;  
import java.util.*;  
public class Principal {  
    public static void main(String ...args){  
        Scanner scanner = new Scanner(System.in);  
    }  
}
```

Todas as classes do pacote java.util podem ser utilizadas somente pelo seu nome simples. Isso é possível graças ao uso do asterísco (*) na declaração da importação.

Exemplo de uso do **import** - Importação Estática

```

package poo;
import static java.util.Arrays.asList;
import java.util.List;
import java.util.Scanner;
public class Principal {
    public static void main(String ...args){
        Scanner scanner = new Scanner(System.in);
        List<String> listaNomes = asList("Jeferson", "José", "Zeca");
    }

```

A classe `java.util.Arrays` possui um método estático `asList()`, o qual está a ser importado no exemplo acima. Não é permitido importar todos os métodos estáticos de uma classe, como é feito com as classes de um dado pacote.

Curiosidade acerca do uso do **import**

Com nomes simples iguais, importa uma vez e não mais...

Quando existirem duas classes com nome simples iguais tais como **org.poo.Pessoa** e **poo.Pessoa**, somente uma delas poderá ser importada. A outra, obrigatoriamente, precisa de seu nome composto para ser utilizada.

Como compilar e executar com pacotes

Exemplo de compilação

```
javac -classpath . org/poo/Pessoa.java
```

Exemplo de execução

```
java -classpath . org.poo.Pessoa
```

O desenvolvedor deve estar no diretório acima do primeiro diretório a representar o primeiro pacote.

Módulos em Java

O que são Módulos?

O conceito de **módulo** remete a uma estrutura adicional de encapsulamento, a qual permite organizar o código-fonte, e seus artefatos de *software* derivados, em um conjunto de pacotes e recursos com políticas de acesso associadas.

História dos Módulos em Java

Módulos foram introduzidos a partir da versão 9...

O conceito de **módulo** só foi disponibilizado nativamente na linguagem de programação Java a partir do Java 9, a dar um passo importante na evolução do desenho de programas de forma mais compartimentada.

História dos Módulos em Java

Módulos foram introduzidos a partir da versão 9...

O conceito de **módulo** só foi disponibilizado nativamente na linguagem de programação Java a partir do Java 9, a dar um passo importante na evolução do desenho de programas de forma mais compartimentada.

Módulos têm origem no projeto Jigsaw...

O projeto Jigsaw (<https://openjdk.java.net/projects/jigsaw/>) é o precursor do **Java Platform Module System (JPMS)**, que em bom Português significa **Sistema de Módulos da Plataforma Java**. O projeto teve início em 2008, e sua integração no Java 9 começou no ano de 2014.

História dos Módulos em Java

A especificação do JPMS está disponível na JSR 376...

A *Java Specification Request (JSR) 376* (<https://www.jcp.org/en/jsr/detail?id=376>), i.e., uma solicitação formal para construir especificações presentes na linguagem de programação Java, contém a especificação do **Sistema de Módulos da Plataforma Java**, a descrever em detalhes o que foi concretizado na implementação de referência realizada na versão 9 da linguagem.

Pense nos Módulos como...



Módulos como uma caixa de organização com domínio bem definido. No exemplo ilustrado pela imagem acima somente colocamos dentro da caixa pacotes, e recursos associados ao domínio da reciclagem.

Declaração de Módulos em Java

Um módulo a declarar é uma caixa a carregar...

Ao declarar um módulo em Java o desenvolvedor deve especificá-lo por meio de um arquivo denominado **module-info.java**.

Declaração de Módulos em Java

Um módulo a declarar é uma caixa a carregar...

Ao declarar um módulo em Java o desenvolvedor deve especificá-lo por meio de um arquivo denominado **module-info.java**.

module-info.java é o boiadeiro da boiada...

O arquivo **module-info.java** fica no diretório raiz onde encontra-se o código-fonte de todas classes pertencentes ao módulo, a especificar as dependências necessárias para a sua compilação, e execução por meio de classes a declarar o método **public static void main()**. Além disso, o arquivo **module-info.java** declara quais os pacotes são disponibilizados para uso externo ao módulo.

Declaração de Módulos em Java - Sintaxe Base

A sintaxe base para declaração de um módulo em Java é a seguinte:

Arquivo **module-info.java**

```
module <nome_do_modulo> {
    <declaração_das_dependências> (opcional)
    <declaração_dos_artefatos_de_software_oferecidos> (opcional)
    <declaração_dos_serviços_que_usa>* (opcional)
    <declaração_dos_serviços_que_oferece>* (opcional)
}
```

* Não serão abordados a utilização e/ou fornecimento de serviços por meio de módulos, a considerar a necessidade de introdução de conceitos associados à Engenharia de Software e Sistemas Distribuídos que estão fora do escopo da disciplina.

Declaração de Módulos em Java - Nome do Módulo

Convenção do nome do módulo

A declaração do nome de um módulo (`<nome_do_modulo>`) segue as convenções da declaração de nomes de pacotes. Entretanto, caso o nome do módulo seja especificado com múltiplos nomes separados por ponto (`.`), os diferentes nomes não são materializados em diretórios distintos no sistema de arquivos. Ao invés disso, um único diretório é criado com o conjunto de caracteres a representar o nome do módulo. Exemplo:

```
module modulo.poo.sensacional { }
```

A declaração acima especifica um módulo em Java de nome composto **modulo.poo.sensacional**, módulo este a ser materializado como um diretório chamado **modulo.poo.sensacional** no sistema de arquivos.

Declaração de Módulos em Java - Dependências

Declaração de dependências

A declaração de dependências de um módulo é realizada por meio da palavra reservada **requires**. As dependências são representadas por outros módulos, e portanto, são indiretamente compostas pelos pacotes que são disponibilizados pelos mesmos. A sintaxe é a seguinte:

requires <modificadores> <nome_do_modulo>;

Onde:

<modificadores> são classificações opcionais à dependência. Exemplos: **transitive** e **static**;

<nome_do_modulo> é o nome do módulo a ser declarado como dependência.

Declaração de Módulos em Java - Dependências: Exemplo

Declaração de dependências

```
module modulo.poo.sensacional {  
  
    requires modulo.poo.entidade;  
  
}
```

A declaração acima especifica um módulo em Java de nome composto **modulo.poo.sensacional**, módulo este possui como dependência o módulo **modulo.poo.entidade**.

PS: As pessoas (i.e. entidades) tornam POO sensacional :-D...

Declaração de Módulos em Java - Dependências Transitivas

Dependências transitivas

Dependências transitivas são dependências indiretas entre módulos que podem ser classificadas por meio da palavra reservada **transitive**.

$$A \longrightarrow B \longrightarrow C$$

(Vocês nem imaginam o que vos espera em Teoria dos Grafos :-D)

Imagine que o **módulo A** possui dependência no **módulo B**, que por sua vez disponibiliza classes a conter métodos com tipo de retorno com dependência no **módulo C**. Logo, essa dependência torna-se transitiva, já que o **módulo A** precisa ter conhecimento de classes presentes no **módulo C** utilizadas pelo **módulo B**. Para evitar que todos os módulos dependentes no **módulo B**, também tenham que declarar dependência no **módulo C**, usa-se o modificador **transitive**.

Declaração de Módulos em Java - Dependências Estáticas

Dependências estáticas

Dependências estáticas são dependências necessárias no momento da compilação, porém, que podem ser opcionais em tempo de execução. As bibliotecas a disponibilizar implementações dos famosos *drivers* de acesso à bases de dados^a são exemplos clássicos de dependências estáticas. As dependências estáticas são declaradas por meio da palavra reservada **static**.

^aAo trabalhar com desenvolvimento de *software*, vocês nunca mais vão deixar de ter contato, direto ou indireto, com as bases de dados depois de conhecê-las :-D...

Compilação de Código-Fonte com Módulos

Para compilar não é precisa mudar de ferramenta :-D...

A compilação de código-fonte encapsulado em um módulo também é realizada por meio do **javac**. A sintaxe base para compilação de um módulo é a seguinte:

```
javac -d <diretorio_destino> -module-source-path <diretorio_origem_fontes>
-m <nome_do_modulo>
```

Onde:

<diretorio_destino> é o diretório onde serão gerados e armazenados os arquivos com extensão .class;

<diretorio_origem_fontes> lista de diretórios a conter os fontes do(s) módulo(s) a ser(em) compilado(s);

<nome_do_modulo> lista de nomes dos módulos a serem compilados. Cada nome é separado por vírgula.

Compilação de Código-Fonte com Módulos: Exemplo

Exemplo de compilação do módulo `org.poo.sensacional`

```
javac -d ../classes -module-source-path . -m modulo.poo.sensacional
```

Bibliografia



BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer I: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2015.



BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer II: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2016.

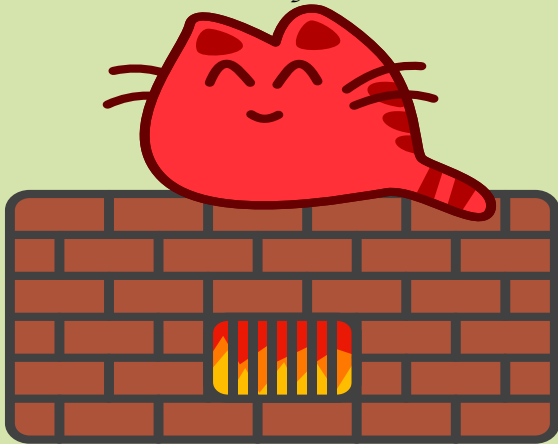


CHENG, F. **“Exploring Java 9”**. Apress. Auckland, New Zealand. 2018.



SANAULLA, M. and SAMOYLOV, N. **“Java 9 Cookbook”**. Packt Publishing Ltd. Birmingham, UK. 2017.

That's it folks!



Thank you for your attention!