

TI-220 Java Orientado a Objetos

ANTONIO CARVALHO - TREINAMENTOS

A solid blue horizontal bar spanning the width of the slide, located at the bottom.

Módulos

Módulos

O sistema de módulos do Java conhecido como Java Platform Module System (JPMS) foi uma grande mudança gerada no Java na versão 9 e permitiu uma melhor integridade da plataforma, facilidade para escalar, e um encapsulamento melhor.

Com o uso de módulos é possível criar versões da JRE mais compactas usando apenas os módulos necessários.

O Java JDK possui um diretório chamado **jmods** que contém os módulos em formato **jmod** (arquivos zipados) contendo uma estrutura de diretórios **bin**, **legal**, **lib**

Módulos (Comandos da JVM)

```
java --list-modules
```

Lista todos os módulos instalados neste Java Runtime

```
java --describe-module <module name>
```

Descreve um modulo específico

```
jlink --module-path ../jmods --add-modules java.base --output <caminho>
```

Cria uma JRE específica contend apenas os modulos selecionados em **--add-modules**

Neste caso cria uma JRE com aproximadamente **35Mb**, bem menor que uma JRE padrão contendo **200Mb**

Módulos (Tipos)

Os tipos de módulos no novo sistema são:

- **System Modules**, aqui são os módulos que vem junto do Java SE e do JDK, estes módulos podem ser vistos por meio do comando **java -list-modules**
- **Application Modules**, este tipo abriga os módulos criados pelos desenvolvedores
- **Automatic Modules**, quando adicionamos um arquivo **.JAR** de um módulo à lista de módulos, cada arquivo **.JAR** será tratado como um novo módulo e todos eles terão acesso uns aos outros automaticamente.
- **Unnamed Modules**, são criados pelas bibliotecas definidas no classpath, todas as bibliotecas que estiverem no classpath estarão automaticamente em um módulo chamado unnamed, onde todas as classes possuirão acesso umas às outras como se estivessem no mesmo módulo

Módulos (Arquivo de configuração)

As classes de um módulo somente são acessíveis dentro do módulo, a não ser que sejam explicitadas em um arquivo de configuração ou seja não é possível acessar classes que sejam públicas, seus métodos ou atributos se não forem explicitados.

Para criar um módulo é preciso definir um arquivo conhecido como **module descriptor** com o nome **module-info.java** no diretório raiz dos pacotes, no caso do **Eclipse** o arquivo **module-info.java** deve ficar na pasta **src**

O conteúdo do arquivo ficara da seguinte forma:

```
module <nomeDoModulo> {  
    ◦ // informações das diretivas  
}
```

Módulos (Diretivas)

As diretivas que podem ser colocadas no arquivo de declaração do módulo são:

requires, informa a dependência de outro modulo, como exemplo
requires javafx.controls;

requires static, informa uma dependência opcional, tal dependência existem em tempo de compilação apenas

requires transitive, assegura as dependências de um módulos sejam repassadas para outro sem que haja uma declaração explícita. Por exemplo se o módulo A depende do módulo B e um outro módulo C tenha dependência do módulo A, a dependência do módulo B **não será automática** a não ser que o modulo A tenha declarado a dependência como **transitive**

Módulos (Diretivas)

As diretivas que podem ser colocadas no arquivo de declaração do módulo são:

exports, por padrão as APIs do módulo não são exportadas para outros módulos, por meio desta diretiva todos os elementos públicos do módulo serão exportados

exports ... to, similar ao exports, esta diretiva pode exportar um módulo a outro(s) módulo(s) em específico

uses, especifica um serviço por meio de uma classe abstrata ou interface que será deixada como explícita para as classes consumidoras deste serviço, a vantagem é a velocidade em que o serviço é disponibilizado no grafo dos módulos e dependências, tornando disponível para a classe consumidora sem necessitar de uma análise prévia pelo compilador.

provides ... with, define o módulo como um provedor de serviço onde outros módulos podem consumir, a sintaxe consiste em

provides <Interface> with <Classe de Implementação>

Módulos (Diretivas)

As diretivas que podem ser colocadas no arquivo de declaração do módulo são:

open, permite que outros módulos possam fazer reflection sobre o módulo definido como open. Por padrão o reflection para elementos privados não será mais autorizados entre módulos a não ser que seja feito de maneira explícita por meio do modificador open.

```
open module <nome do modulo> {  
  
}
```







opens, funciona de maneira similar ao open porém é aplicável para alguns pacotes específicos do modulo. Exemplo:

```
module <nome do modulo> {  
    opens <pacote específico>;  
}
```

opens ... to, funciona de maneira similar ao opens porém o pacote poderá ter o reflection visível para alguns módulos em específico. Exemplo:

```
module <nome do modulo> {  
    opens <pacote específico> to <modulo 1>, <modulo 2>, <modulo N>;  
}
```

Exemplo de criação de módulo

▼  > TesteModulo [fatec-2019-1s master]
 >  JRE System Library [jdk-11.0.1]
 ▼  > src
 ▼  > edu.teste.modulo
 >  TesteA.java
 >  module-info.java

```
module teste.modulo {  
    exports edu.teste.modulo;  
}  
  
package edu.teste.modulo;  
public class TesteA {  
    public void fazAlgo() {  
        System.out.println("Teste de classe");  
    }  
}
```

Exemplo de uso do modulo

```
▼ [?] > TesteModuloCliente [fatec-2019-1s master]
  > [?] JRE System Library [jdk-11.0.1]
  ▼ [?] > src
    ▼ [?] > edu.teste.modulo.cliente
      > [?] TesteModuloClienteA.java
      > [?] module-info.java
```

```
module teste.modulo.cliente {
    requires teste.modulo;
}

package edu.teste.modulo.cliente;
import edu.teste.modulo.TesteA;
public class TesteModuloClienteA {
    public static void main(String[] args) {
        TesteA a = new TesteA();
        a.fazAlgo();
    }
}
```

Nota: Será preciso adicionar o projeto **TesteModulo** no caminho de compilação do projeto **TesteModuloCliente**

Exemplo de criação de módulo com serviço

```
▼ [?] > TesteServico [fatec-2019-1s master]
  > [?] JRE System Library [jdk-11.0.1]
  ▼ [?] > src
    ▼ [?] > edu.curso
      > [?] ServicoA.java
      > [?] ServicoAImpl1.java
      > [?] ServicoAImpl2.java
      > [?] module-info.java
```

```
module teste.app {
    exports edu.curso;
    provides edu.curso.ServicoA
    with edu.curso.ServicoAImpl1;
}

package edu.curso;
public interface ServicoA {
    void fazAlgo(String text);
}

package edu.curso;
public class ServicoAImpl1 implements ServicoA {
    @Override
    public void fazAlgo(String text) {
        System.out.println("Provido pelo Servico1: " + text);
    }
}
```

Exemplo de consumo de modulo com serviço

```
▼ [?] > TesteConsumoServico [fatec-2019-1s master]
  > [?] JRE System Library [jdk-11.0.1]
  ▼ [?] > src
    ▼ [?] > edu.curso.consumo
      > [?] UserServiceicoA.java
      > [?] module-info.java
```

```
module teste.app.consumidor {
    requires teste.app;
    uses edu.curso.ServicoA;
}

package edu.curso.consumo;
import java.util.Iterator;
import java.util.ServiceLoader;
import edu.curso.ServicoA;
public class UserServiceicoA {
    public static void main(String[] args) {
        ServiceLoader<ServicoA> a =
            ServiceLoader.load(ServicoA.class);
        Iterator<ServicoA> it = a.iterator();
        while (it.hasNext()) {
            ServicoA srva = it.next();
            srva.fazAlgo("Teste de Texto");
        }
    }
}
```

Nota: Será preciso adicionar o projeto **TesteServio** no caminho de compilação do projeto **TesteConsumoServico**

Bibliografia

FRANKLIN CHRISTOPHER, A Guide to Java 9 Modularity, Baeldung, 2018, disponível em: <https://www.baeldung.com/java-9-modularity>

PAUL, DEITEL. Understanding Java 9 Modules – What they are and how to use them, Java Magazine ed. Sep-Oct, 2017, disponível em: <http://www.javamagazine.mozaicreader.com/SeptOct2017#&pageSet=18&page=0>

RO TSAERT, GUNTER, Java 9 Modules (Part 1): Introduction, Dzone, 2018, disponível em: <https://dzone.com/articles/java-9-modules-introduction-part-1>