Módulo 2

No módulo anterior configuramos o ambiente de desenvolvimento e geramos um projeto vazio. Criamos também uma classe com a definição de alguns testes. Neste módulo implementaremos os testes, a solução e conheceremos algumas funcionalidades do Demoiselle.

Se você ainda não tomou nota, não esqueça de guardar o link para as documentações de referência nos favoritos do seu *browser*: http://demoiselle.sourceforge.net/docs. Vale a pena cadastrar-se na lista de usuários também: https://lists.sourceforge.net/lists/listinfo/demoiselle-users. Nos próximos módulos não haverá mais citação a estes *links*. Quem avisa, amigo é!

Matrícula

Vamos começar pelo mais simples, a matrícula do aluno. Para facilitar, consideraremos que só existe uma turma.

1. Crie a classe *Turma* na pasta *src/**main**/java* no pacote *br.gov.serpro.inscricao*.
2. Crie os métodos *public void matricular(String aluno)* e *public boolean estaMatriculado(String aluno)*.
3. Apenas corrija os erros de compilação, faça o método *estaMatriculado* retornar *false*.

Pelo nome dos métodos você já imaginou para que servem, não é? Sua classe vai ficar assim:

```
public class Turma {

	public void matricular(String aluno){
	}

	public boolean estaMatriculado(String aluno){
		return false;
	}
}
```

Antes de elaborar a solução, vamos fazer o teste. Abra a classe *TurmaTest* e implemente o método de teste *matricularAlunoComSucesso*. Utilizaremos as assertivas do JUnit para verificar se o aluno foi matriculado. Seu teste vai ficar assim:

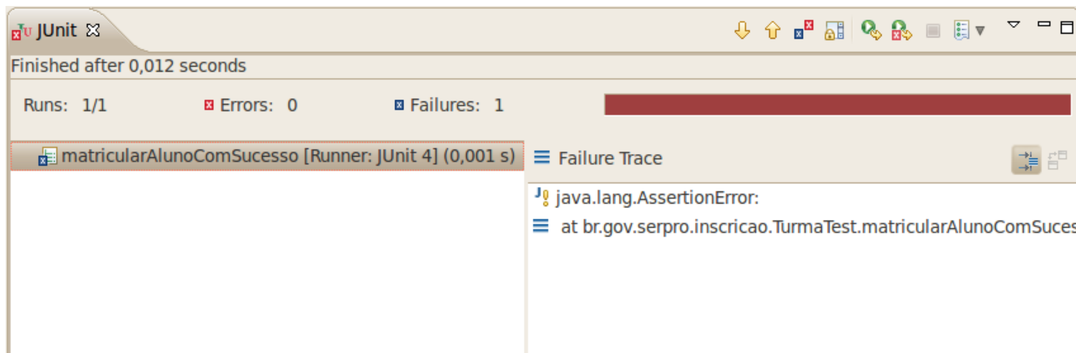
```
@Test
public void matricularAlunoComSucesso(){
	Turma turma = new Turma();

	turma.matricular("Santos Dumont");
	Assert.assertTrue(turma.estaMatriculado("Santos Dumont"));
}
```

Importe a classe *org.junit.Assert.* Na linha *Assert.assertTrue(turma.estaMatriculado("Santos Dumont"))* utilizamos a o método *assertTrue* do JUnit que verifica se o retorno da chamada *turma.estaMatriculado("Santos Dumont")* é *true*. Caso não seja, o teste falhará. Rode o teste e veja o resultado:

1. Clique com o botão direito na classe *TurmaTest*.
2. No menu que apareceu acesse *Run As / JUnit Test*.

O resultado será este:



Agora implemente a sua solução na classe *Turma*. Para facilitar, vamos armazenar os alunos matriculados numa lista.

1. Crie o atributo *private List<String> alunosMatriculados = new ArrayList<String>()*.
2. Implemente o método *matricular* que inclui o aluno na lista de matriculados.
3. Implemente o método *estaMatriculado* que verifica se o aluno está na lista.

A classe *Turma* vai ficar assim:

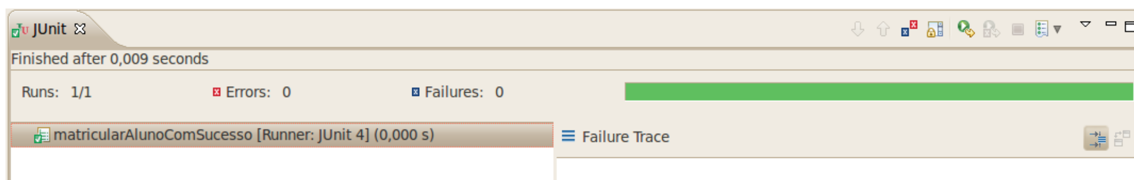
```
public class Turma {

	private List<String> alunosMatriculados = new ArrayList<String>();

	public void matricular(String aluno){
		alunosMatriculados.add(aluno);
	}

	public boolean estaMatriculado(String aluno){
		return alunosMatriculados.contains(aluno);
	}
}
```

Rode o teste novamente e perceba que agora vai passar:



Você deve estar se perguntando: “Onde está o Demoiselle neste código?”. Por enquanto, em lugar nenhum!

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-2-video-1

Logger

Vamos incrementar nossa aplicação. Sempre que um aluno for matriculado o sistema deve gerar um *log* no console. Se você pensou em usar *System.out.println()*, esqueça! Esta solução gera problemas catastróficos de performance, por isso inventaram o *Logger*.

As vantagens de utilizar *Logger* são várias: desde melhoria de desempenho em relação ao preemptivo *System.out* até flexibilidade no formato do *log*. Para utilizar, você precisaria criar o seguinte atributo na classe *Turma*:

```
private Logger logger = LoggerFactory.getLogger(br.gov.serpro.inscricao.Turma.class);
```

Seria melhor assim:

```
@Inject
private Logger logger;
```

Com o Demoiselle você poderá fazer da segunda forma. O que você precisa saber agora é que as duas maneiras são equivalentes, porém na segunda você está utilizando um recurso importante do Framework: a Injeção de Dependências.

Então mãos à obra:

1. Inclua o atributo ***@Inject private Logger logger*** na classe *Turma*.
2. Corrija o erro de compilação, importe a interface ***org.slf4j.Logger***.
3. Acrescente no método *matricular* a geração do *log* utilizando ***logger.info(. . .)***.

As modificações na classe são estas:

```
...

@Inject
private Logger logger;

public void matricular(String aluno){
      ...
      logger.info("Aluno matriculado com sucesso");
}

...
```

Rode novamente o teste. Desta vez vai falhar com o seguinte erro: *java.lang.NullPointerException* na linha que contém *logger.info()*, na classe *Turma*. Ou seja, a injeção não funcionou! Em outras palavras, ninguém criou uma instância de *org.slf4j.Logger* e a atribuiu à propriedade *logger* de *Turma*. Quem deveria fazer isto?

Vamos corrigir isto!

1. Na classe *TurmaTest* a turma não pode ser instanciada com *new*. Crie o atributo ***@Inject private Turma turma***.
2. Anote a classe de teste com ***@RunWith(DemoiselleRunner.class)***.

Injeções nos casos de teste não funcionam, pois o JUnit não oferece suporte nativo ao CDI. Por isto criamos o componente ***demoiselle-junit*** com a classe *br.gov.frameworkdemoiselle.junit.DemoiselleRunner*. O componente demoiselle-junit cria um ambiente CDI utilizando o projeto Weld (Implementação de referência da especificação do CDI) tornando possível a injeção de dependências.

Feitos todos os ajustes, *TurmaTest* ficará assim:

```
@RunWith(DemoiselleRunner.class)
public class TurmaTest {

	@Inject
	private Turma turma;

	@Test
	public void matricularAlunoComSucesso(){
		turma.matricular("Santos Dumont");
		Assert.assertTrue(turma.estaMatriculado("Santos Dumont"));
	}

	...
}
```

Execute o teste novamente, agora passará. Observe o *log* gerado no console:

```
18:17:57,187 INFO [inscricao.Turma] Aluno matriculado com sucesso
```

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-2-video-2

Injeção de Dependências

Injeção de dependência é um padrão de desenvolvimento utilizado para manter o baixo nível de acoplamento entre os módulos da aplicação. Em Java, o conceito foi popularizado por diversos *frameworks*, tais como: Spring, JBoss Seam, Google Guice e a primeira versão do Demoiselle.

Dentre as novidades do JavaEE 6, o mecanismo de injeção de dependências se tornou padrão através da JSR-299, comumente referenciada como CDI (*Contexts and Dependency Injection*). Uma das principais vantagens desse recurso é o gerenciamento do ciclo de vida dos objetos pelo container. A anotação *@javax.inject.Inject* é utilizada para solicitar ao container a criação dos seus objetos. Caso seu objeto seja criado com *new*, o CDI não tomará conhecimento e o gerenciamento do ciclo de vida será de sua responsabilidade.

O Demoiselle 2 permite injetar classes que que não foram originalmente criadas de acordo com a JSR-299, como é o caso do *org.slf4j.Logger*. Se você estivesse utilizando CDI sem o Demoiselle, você teria que resolver por conta própria.

Resource Bundle

Também conhecido como I18N, o recurso de internacionalização permite que a aplicação suporte diversos idiomas. Você deve estar pensando: "Nunca precisei disto na vida!". Então responda, onde você guarda as mensagens da aplicação? No próprio código-fonte, em constantes ou em arquivo de mensagens? Sem sombra de dúvidas, o pior cenário é espalhar as mensagens pelo código-fonte, o melhor é guardar em arquivo.

A plataforma Java provê a classe *java.util.ResourceBundle* que possibilita a manipulação de arquivos de mensagens, porém não há integração nativa com CDI. Vejamos como o Demoiselle oferece esta facilidade:

1. Acrescente o atributo ***@Inject private ResourceBundle bundle*** na classe *Turma*.
2. Corrija o erro de compilação, importe a classe ***br.gov.frameworkdemoiselle.util.ResourceBundle***.
3. Modifique o método *matricular* para buscar a mensagem utilizando ***bundle.getString("matricula.sucesso")***.
4. Crie o arquivo ***messages.properties*** na pasta *src/main/**resources***.
5. Abra o arquivo e inclua a linha ***matricula.sucesso=Aluno matriculado com sucesso***.

A classe *Turma* vai ficar assim:

```
...

@Inject
private ResourceBundle bundle;

public void matricular(String aluno) {
      ...
      logger.info(bundle.getString("matricula.sucesso"));
}

...
```

Rode o teste e verifique que a mensagem continua aparecendo no console. Seria interessante que o *log* indicasse o aluno matriculado?

1. Abra o arquivo ***messages.properties***.
2. Modifique a linha para ficar assim: ***matricula.sucesso=Aluno {0} matriculado com sucesso***.
3. Abra a classe *Turma*
4. Exclua o ***import java.util.ResourceBundle*** .
5. Acrescente o ***import br.gov.frameworkdemoiselle.util.ResourceBundle***.
6. Passe o parâmetro na chamada ***bundle.getString("matricula*.sucesso*", aluno)***.

A classe *Turma* vai ficar assim:

```
...

@Inject
private ResourceBundle bundle;

public void matricular(String aluno) {
      ...
      logger.info(bundle.getString("matricula.sucesso", aluno));
}

...
```

A classe *java.util.ResourceBundle* não suporta a passagem de parâmetros, por isso o Framework disponibiliza o utilitário *br.gov.frameworkdemoiselle.util.ResourceBundle*. O símbolo "{0}" será substituído pelo primeiro parâmetro. É possível utilizar "chave={1} {2} {3}" desde que os parâmetros sejam passados no *getString*, ex: *bundle.getString("chave", param1, param2, param3)* e assim por adiante.

Mas se você precisar de uma aplicação multi-idiomas, o quefazer? Recorra ao guia de referência, pois não vamos aprofundar nesta funcionalidade no curso básico. Vamos ver outras situações agora.

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-2-video-3

Aluno duplicado ou sala cheia

Vamos partir para a implementação dos testes do fluxo de exceção, ou seja, quando a nossa aplicação deve falhar propositalmente. No *falhaAoTentarMatricularAlunoDuplicado*, espera-se que o método *matricular* da classe *Turma* lance *java.lang.RuntimeException*. Se a exceção não ocorrer o teste falhará automaticamente.

```
@Test(expected = RuntimeException.class)
public void falhaAoTentarMatricularAlunoDuplicado() {
      turma.matricular("Orville Wright");
      turma.matricular("Orville Wright");
}
```

Convencionando que a turma lota com 5 alunos, vamos implementar o teste *falhaAoTentarMatricularAlunoNaTurmaCheia*. Seguindo o mesmo raciocínio do teste anterior, o teste falhará caso a matrícula do 6º aluno não gere a exceção esperada.

```
@Test(expected = RuntimeException.class)
public void falhaAoTentarMatricularAlunoNaTurmaCheia() {
      for (int i = 1; i <= 5; i++) {
            turma.matricular("Aluno " + i);
      }

      turma.matricular("Aluno 6");
}
```

Rode o teste e observe que irá falhar.

A propriedade ***expected*** utilizada na anotação @Test indica que para este teste ser executado com sucesso, ele terá que receber uma exceção daquele tipo. Neste caso RuntimeException.

Chegou a hora de ajustar o método *matricular* da classe *Turma*. O método ficará assim, após implementar as validações:

```
public void matricular(String aluno) {
	if (estaMatriculado(aluno) || alunosMatriculados.size() == 5) {
		throw new RuntimeException();
	}

	...
}
```

Rode novamente o teste, agora tem que passar!

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-2-video-4

Retrospectiva

Implementamos os testes da nossa aplicação. Conhecemos o componente *demoiselle-junit* que possibilita a injeção nos cases de teste. Experimentamos as facilidades do Demoiselle para manipulação de arquivos de *log* e internacionalização do projeto.

Começamos praticar o lançamento de exceções. No próximo módulo aprenderemos como capturá-las utilizando as funcionalidades do Framework. Veremos também como parametrizar a aplicação com arquivos de configuração.