Módulo 2

Introdução à Programação II

Lição 2

Exceções e Assertivas

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

O conceito básico sobre tratamento de exceções foi mostrado no Módulo 1 – Introdução a Programação I. Esta lição fornecerá um entendimento aprofundado sobre exceções e também sobre assertivas.

Ao final desta lição, o estudante será capaz de:

- Tratar exceções pelo uso de *try*, *catch* e *finally*
- Diferenciar entre o uso de *throw* e *throws*
- Utilizar as classes de exceções existentes
- Diferenciar entre exceções verificadas e não verificadas
- Definir suas próprias classes de exceção
- Explicar os benefícios do uso de assertivas
- Utilizar declarações assertivas

2. O que são exceções?

2.1. Introdução

Bugs ou erros ocorrem freqüentemente na execução dos projetos, mesmo quando estes são escritos por programadores hábeis e experientes. Para evitar perder mais tempo na verificação do erro do que na resolução do problema em si, Java nos fornece um mecanismo para o tratamento de exceções.

As exceções são, resumidamente, eventos excepcionais. São erros que ocorrem durante a execução de um determinado trecho de instrução, alterando o seu fluxo normal. Os erros podem ocorrer por diferentes motivos, por exemplo: erros de divisão por zero, acessar elementos em um array além de seu próprio tamanho, entrada inválida, erro de acesso ao disco rígido, abertura de arquivo inexistente e estouro de memória.

2.2. As classes Error e Exception

Todas as exceções são sub-classes, direta ou indiretamente, da classe *Throwable*. Imediatamente abaixo desta classe encontram-se as duas categorias gerais de exceções: as classes *Error* e *Exception*.

A classe *Exception* lida com as condições que os usuários podem tratar. Em geral, estas condições são o resultado de algumas falhas no código. Exemplo de exceções são: erro de divisão por zero e erro de índice em um array.

Por outro lado, a classe *Error* é utilizada pela JVM para manipular os erros ocorridos no ambiente de execução. Geralmente, estes erros estão além do controle do programador, desde que sejam causados dentro do ambiente de execução. Por exemplo: erro de falta de memória e erro de acesso ao disco rígido.

2.3. Exemplo de Exceção

Considere a seguinte classe:

```
class DivByZero {
   public static void main(String args[]) {
      System.out.println(3/0);
      System.out.println("Pls. print me.");
   }
}
```

Executando o código, a seguinte mensagem de erro será apresentada:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
DivByZero.main(DivByZero.java:3)
```

A mensagem fornece a informação do tipo de exceção que ocorreu e a linha do código que a originou. Esta é a forma como o manipulador padrão trata as exceções não capturadas. Quando não há código do usuário tratando as exceções, o manipulador padrão de exceções entra em ação. Primeiramente, a descrição da exceção que ocorreu é apresentada. Além disso, também é apresentado o caminho que indica a hierarquia dos métodos até onde a exceção aconteceu. Por fim, o manipulador padrão de exceções faz com que o código termine sua execução.

Ao necessitar tratar as exceções de uma maneira diferente? Felizmente, a linguagem Java possui três importantes palavras-chaves para o tratamento de exceções *try*, *catch* e *finally*.

3. Capturando Exceções

3.1. As instruções try-catch

Como mencionado na seção anterior, as palavras-chaves *try*, *catch* e *finally* são usadas para tratar diferentes tipos de exceções. Estas três palavras-chaves são usadas juntas mas o bloco *finally* é opcional. Primeiramente, focaremos no bloco *try-catch* e mais tarde voltaremos ao bloco *finally*.

Encontra-se abaixo a sintaxe geral de uma instrução try-catch.

```
try {
   <código a ser monitorado para exceções>
} catch (<ClasseExceção1> <nomeObj>) {
   <tratar se ClasseExceção1 ocorrer>
}
...
} catch (<ClasseExceçãoN> <NomeObj>) {
   <tratar se ClasseExceçãoN ocorrer>
}
```

Dicas de programação:

1. O bloco catch começa depois do fechamento do bloco precedente, seja este try ou catch.
2. Instruções dentro do bloco devem ser recuadas de modo a conseguir uma melhor visualização.

Aplicando isto a classe *DivByZero* teremos*:*

```
class DivByZero {
   public static void main(String args[]) {
      try {
         System.out.println(3/0);
         System.out.println("Please print me.");
      } catch (ArithmeticException exc) {
         //reação ao evento
         System.out.println(exc);
      }
      System.out.println("After exception.");
   }
}
```

O erro de divisão por zero é um exemplo de um tipo de exceção encontrado na classe *ArithmeticException*. O código trata o erro simplesmente apresentando uma descrição do problema.

Como saída para esta classe, teremos:

```
java.lang.ArithmeticException: / by zero
After exception.
```

Um código específico monitorado no bloco *try* pode causar a ocorrência de mais de um tipo de exceção. Neste caso, os diferentes tipos de erro podem ser tratados pelo uso de diversos blocos *catch*. Observe que o código no bloco *try* pode acionar apenas uma exceção por vez, mas pode provocar a ocorrência de diferentes tipos de exceção em momentos diferentes.

Aqui temos um exemplo de uma classe que trata mais de um tipo de exceção:

```
class MultipleCatch {
   public static void main(String args[]) {
      try {
         int den = Integer.parseInt(args[0]);  //linha 4
         System.out.println(3/den);       //linha 5
      } catch (ArithmeticException exc) {
         System.out.println("Divisor was 0.");
      } catch (ArrayIndexOutOfBoundsException exc2) {
         System.out.println("Missing argument.");
      }
      System.out.println("After exception.");
   }
}
```

Neste exemplo, a linha 4 pode retornar uma *ArrayIndexOutOfBoundsException* quando o usuário se esquecer de entrar com um argumento enquanto a linha 5 retorna uma *ArithmeticException* se o usuário entrar 0 como argumento.

Veja o que acontece ao executar a classe quando os seguintes argumentos abaixo são informados pelo usuário:

1. Se nenhum argumento for passado, a seguinte saída será apresentada:

```
Missing argument.
After exception.
```

2. Passando o valor **1** como argumento, a seguinte saída será apresentada:

```
3
After exception.
```

3. Passando o valor **0**, a seguinte saída será apresentada:

```
Divisor was 0.
After exception.
```

Também é possível aninhar blocos *try* em Java.

```
class NestedTryDemo {
   public static void main(String args[]){
      try {
         int a = Integer.parseInt(args[0]);
         try {
            int b = Integer.parseInt(args[1]);
            System.out.println(a/b);
         } catch (ArithmeticException e) {
            System.out.println("Divide by zero error!");
         }
      } catch (ArrayIndexOutOfBoundsException exc) {
         System.out.println("2 parameters are required!");
      }
   }
}
```

Vejamos o que acontece a classe quando os seguintes argumentos são informado:

a) Nenhum argumento

```
2 parameters are required!
```

b) Valor **15**

```
2 parameters are required!
```

c) Valor **15 3**

```
5
```

d) Valor **15 0**

```
Divide by zero error!
```

O código abaixo possui um try aninhado mascarado com o uso de métodos:

```
class NestedTryDemo2 {
   static void nestedTry(String args[]) {
      try {
         int a = Integer.parseInt(args[0]);
         int b = Integer.parseInt(args[1]);
         System.out.println(a/b);
      } catch (ArithmeticException e) {
         System.out.println("Divide by zero error!");
      }
   }
   public static void main(String args[]){
      try {
         nestedTry(args);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("2 parameters are required!");
      }
   }
}
```

Qual é a saída dessa classe quando testado com os seguintes argumentos?

a) Nenhum argumento
b) 15
c) 15 3
d) 15 0

A saída esperada para a classe *NestedTryDemo2* é similar a classe *NestedTryDemo*.

3.2. A palavra-chave finally

Iremos agora incorporar a palavra-chave *finally* à instrução try-catch. Observe como as palavras reservadas são organizadas no bloco:

```
try {
   <código a ser monitorado para exceções>
} catch (<ClasseExceção1> <nomeObj>) {
   <tratar se ClasseExceção1 ocorrer>
} ...
} finally {
   <código a ser executado antes do bloco try ser finalizado>
}
```

Dicas de programação:

1. A mesma convenção de codificação se aplica ao bloco *finally* como no bloco *catch*. O bloco *finally* começa depois do fechamento do bloco *catch* precedente.
2. Instruções dentro desse bloco devem ser também adiantadas de forma a conseguir melhor clareza no código.

O bloco *finally* contém o código para a finalização após um *try* ou *catch*. Este bloco de código é sempre executado independentemente de uma exceção acontecer ou não no bloco *try*. Isto permanece verdadeiro mesmo que instruções *return*, *continue* ou *break* sejam executadas.

Vejamos um exemplo completo com o bloco *try-catch-finally*:

```
class FinallyDemo {
   public static void main(String args[]) {
      for (int i = 1; i > -1; i--) {
         try {
            System.out.println(2/i);
         } catch (ArithmeticException e) {
            System.out.println("Divide by zero error!");
         } finally {
            System.out.println("Finally forever!");
         }
      }
   }
}
```

A saída esperada para esta classe, será:

```
2
Finally forever!
Divide by zero error!
Finally forever!
```

4. Lançamento de Exceções

4.1. A palavra-chave throw

Além de capturar exceções, Java também permite que os métodos lancem exceções (por exemplo, faz com que um evento excepcional ocorra). A sintaxe para o lançamento de exceções é:

```
throw <objetoExceção>;
```

Considere este exemplo:

```
class ThrowDemo {
   public static void main(String args[]){
      try {
         throw new RuntimeException("throw demo");
      } catch (RuntimeException e) {
         System.out.println("Exception caught here.");
         System.out.println(e);
      }
   }
}
```

Esta classe mostrará a seguinte saída:

```
Exception caught here.
java.lang.RuntimeException: throw demo
```

4.2. A palavra-chave throws

No caso de um método causar uma exceção mas não capturá-la, deve-se utilizar a palavra-chave *throws* para repassar esta para quem o chamou. Esta regra se aplica apenas para exceções verificadas. Veremos mais sobre exceções verificadas e não-verificadas mais a frente.

Esta é a sintaxe para o uso da palavra-chave *throws*:

```
<tipo>* <nomeMetodo> (<argumento>*) throws <listaExcecao> {
   <instrução>*
}
```

É necessário um método para cada *catch* ou lista de exceções que podem ser lançadas, contudo podem ser omitidas aquelas do tipo *Error* ou *RuntimeException*, bem como suas sub-classes.

Este exemplo indica que *myMethod* não trata *ClassNotFoundException*.

```
class ThrowingDemo {
   public static void myMethod() throws ClassNotFoundException {
      throw new ClassNotFoundException("just a demo");
   }
   public static void main(String args[]) {
      try {
         myMethod();
      } catch (ClassNotFoundException e) {
         System.out.println(e);
      }
   }
}
```

A classe apresentará a seguinte saída:

```
java.lang.ClassNotFoundException: just a demo
```

Existem quatro cenários diferentes num bloco *try-catch-finally*:

1. Uma saída forçada ocorre quando o fluxo de controle é forçado a sair do bloco *try* por uma instrução *return*, *continue* ou *break*.
2. As instruções dentro do bloco *try-catch-finally* executam normalmente sem que erro algum ocorra.
3. O código pode ter um bloco *catch* específico para tratar a exceção que foi lançada.
4. Temos o oposto do terceiro cenário, onde a exceção não é tratada. Nesse caso, a exceção lançada não foi tratada por nenhum bloco *catch*.

Estes cenários são vistos na classe a seguir:

```
class FourDemo {
   static void myMethod(int n) throws Exception{
      try {
         switch(n) {
            case 1: System.out.println("first case"); return;
            case 3: System.out.println("third case");
                    throw new RuntimeException("third case demo");
            case 4: System.out.println("fourth case");
                    throw new Exception("fourth case demo");
            case 2: System.out.println("second case");
         }
      } catch (RuntimeException e) {
            System.out.print("RuntimeException caught: " + e.getMessage());
      } finally {
            System.out.println("try-block is entered.");
      }
   }
   public static void main(String args[]){
      for (int i=1; i<=4; i++) {
         try {
            FourDemo.myMethod(i);
         } catch (Exception e){
            System.out.print("Exception caught: ");
            System.out.println(e.getMessage());
         }
         System.out.println();
      }
   }
}
```

As seguintes linhas são esperadas como saída dessa classe:

```
first case
try-block is entered.

second case
try-block is entered.

third case
RuntimeException caught: third case demo
try-block is entered.

fourth case
try-block is entered.
Exception caught: fourth case demo
```

5. Categorias de Exceções

5.1. Hierarquia das Classes de Exceções

Conforme mencionado anteriormente, a classe de origem de todas as classes de exceção é *Throwable*. Abaixo encontra-se a hierarquia das classes de exceções. Todas essas exceções estão definidas no pacote *java.lang*.

<table>
<tr><th colspan="4">Hierarquia das Classes de Exceções</th></tr>
<tr><td rowspan="20">Throwable</td><td>Error</td><td colspan="2">LinkageError, ...</td></tr>
<tr><td></td><td colspan="2">VirtualMachineError, ...</td></tr>
<tr><td rowspan="19">Exception</td><td colspan="2">ClassNotFoundException,</td></tr>
<tr><td colspan="2">CloneNotSupportedException,</td></tr>
<tr><td colspan="2">IllegalAccessException,</td></tr>
<tr><td colspan="2">InstantiationException,</td></tr>
<tr><td colspan="2">InterruptedException,</td></tr>
<tr><td rowspan="3">IOException,</td><td>EOFException,</td></tr>
<tr><td>FileNotFoundException,</td></tr>
<tr><td>...</td></tr>
<tr><td rowspan="10">RuntimeException,</td><td>ArithmeticException,</td></tr>
<tr><td>ArrayStoreException,</td></tr>
<tr><td>ClassCastException,</td></tr>
<tr><td>IllegalArgumentException,
(IllegalThreadStateException e
NumberFormatException como sub-classes)</td></tr>
<tr><td>IllegalMonitorStateException,</td></tr>
<tr><td>IndexOutOfBoundsException,</td></tr>
<tr><td>NegativeArraySizeException,</td></tr>
<tr><td>NullPointerException,</td></tr>
<tr><td>SecurityException</td></tr>
<tr><td colspan="2">...</td></tr>
</table>

Tabela 1: Hierarquia das Classes de Excessões

Agora que estamos familiarizados com diversas classes de exceção, devemos conhecer a seguinte regra: Múltiplos blocos *catch* devem ser ordenados da sub-classe para a super-classe.

```
class MultipleCatchError {
   public static void main(String args[]){
      try {
         int a = Integer.parseInt(args [0]);
         int b = Integer.parseInt(args [1]);
         System.out.println(a/b);
         } catch (Exception e) {
            System.out.println(e);
         } catch (ArrayIndexOutOfBoundsException e2) {
            System.out.println(e2);
         }
         System.out.println("After try-catch-catch.");
   }
}
```

A compilação desse código produzirá a mensagem de erro abaixo, já que a classe *Exception* é super-classe da classe *ArrayIndexOutOfBoundsException*.

```
MultipleCatchError.java:9: exception
java.lang.ArrayIndexOutOfBoundsException has already been caught
         } catch (ArrayIndexOutOfBoundsException e2) {
```

5.2. Exceções Verificadas e Não-verificadas

Uma exceção pode ser verificada ou não-verificada.

Uma exceção verificada é aquela que é verificada pelo compilador Java. O compilador se certifica que cada *catch* ou lista de exceções encontram-se dentro da cláusula *throws*. Se a exceção verificada não for capturada nem listada, ocorre um erro de compilação.

Ao contrário das exceções verificadas, as exceções não-verificadas não são condicionadas à verificação para o tratamento de exceções no momento da compilação. As classes de exceção não-verificadas são: *Error*, *RuntimeException*, e suas sub-classes. Desse modo, estes tipos de exceção não são verificadas porque o tratamento de todas as exceções pode complicar o código o que causaria um enorme transtorno.

5.3. Exceções Definidas pelo Usuário

Apesar de muitas classes de exceção já existirem no pacote *java.lang*, as classes de exceção embutidas não são suficientes para cobrir todas possibilidades de exceções que podem ocorrer. Por essa razão, é provável criar nossas próprias exceções.

Para criar nossa própria exceção, teremos que criar uma classe que estenda a classe *RuntimeException* ou *Exception*. Deste modo, devemos customizar a classe de acordo com o problema a ser resolvido. Atributos de objeto e construtores podem ser adicionados na sua classe de exceção.

Segue um exemplo:

```
class ExplodeException extends RuntimeException{
   public ExplodeException(String msg) {
      super(msg);
   }
}
class ExplodeDemo {
   public static void main(String args[]) {
      try {
         throw new ExplodeException("Explode Message");
      } catch (ExplodeException e) {
         System.out.println("Message: " + e.getMessage());
      }
   }
}
```

Aqui está a saída esperada para a classe:

```
Message: Explode Message
```

6. Assertivas

6.1. O que são Assertivas?

Assertivas permitem ao programador descobrir se uma suposição foi encontrada. Por exemplo, uma data onde o mês não está no intervalo entre 1 e 12 deveria ser considerada inválida. O programador pode afirmar que o mês deve se encontrar neste intervalo. Contudo, é possível utilizar outros construtores para simular a funcionalidade das assertivas, mas seria difícil fazer isto de alguma maneira com o recurso da assertiva desabilitado. A boa notícia sobre assertivas é que o usuário tem a opção de habilitá-la ou não na execução.

Assertivas podem ser consideradas como uma extensão de comentários em que a assertiva informa à pessoa que lê o código que uma condição específica deve ser sempre satisfeita. Com assertivas não há necessidade de ler cada comentário para descobrir suposições feitas no código. Em vez disso, ao executar a classe, a mesma informará se as assertivas feitas são verdadeiras ou não. No caso de uma assertiva ser falsa, um *AssertionError* será lançado.

6.2. Habilitando e Desabilitando Assertivas

Para utilizar as declarações assertivas não se faz necessário a importação do pacote *java.util.assert*. Declarações assertivas são utilizadas para verificar os argumentos de métodos não-públicos, pois métodos públicos podem ser acessados diretamente por quaisquer outras classes. É possível que os autores dessas outras classes não estejam atentos que eles terão assertivas habilitadas. Nesse caso, a classe poderá agir incorretamente. Já os métodos não públicos, geralmente, são chamados por meio de códigos escritos por pessoas que tem acesso aos métodos. Desse modo, eles devem estar cientes que ao executar seu código, as assertivas devem estar habilitadas.

Para compilar arquivos que usam assertivas, um parâmetro extra na linha de comando é necessário como mostrado abaixo:

```
javac –source 1.4 MyProgram.java
```

Para executar a classe sem o recurso da assertiva, basta executá-la normalmente:

```
java MyProgram
```

Todavia, para habilitar o recurso das assertivas, é necessário utilizar os argumentos *–ea* ou *-enableassertions*:

```
java –enableassertions MyProgram
```

Para habilitar a verificação de assertivas em tempo de execução no NetBeans IDE, siga os seguintes passos:

1. Procure o nome do projeto no painel *Projects* à esquerda. Pressione com o botão direito no nome do projeto
2. Selecione *Properties*
3. À esquerda, no painel *Categories*, selecione *run*
4. No campo *VM Options* insira *-ea*
5. Clique no botão *OK*
6. Compile e execute o projeto normalmente.

As telas abaixo irão ajudá-lo a entender cada passo:

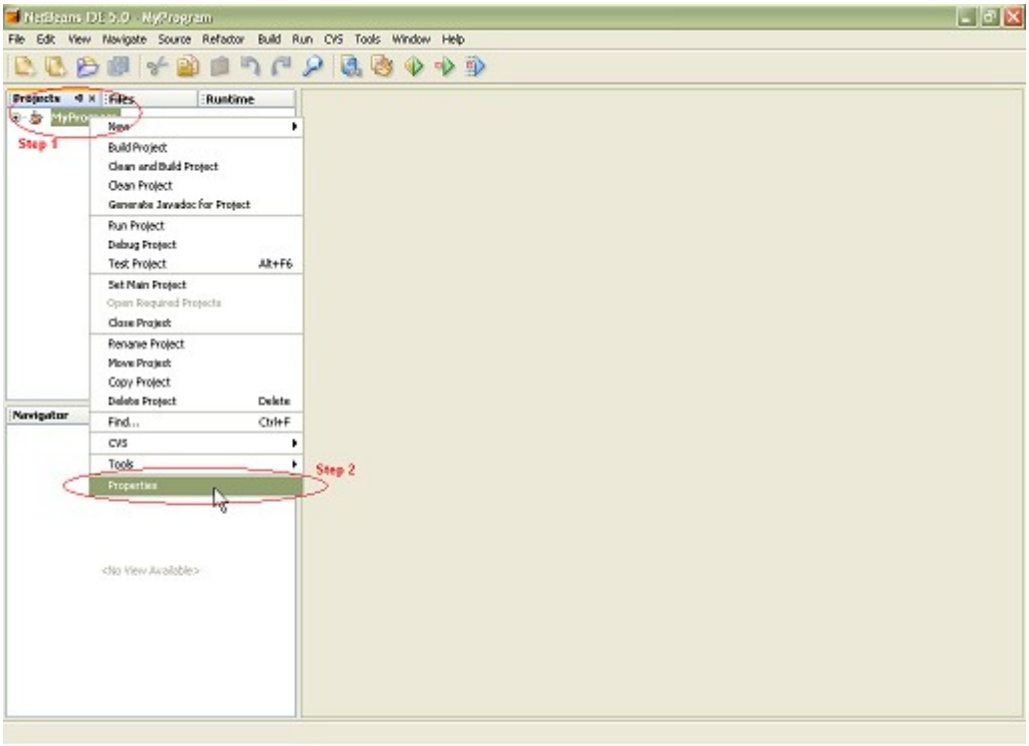

Figura 1: Habilitando assertivas no NetBeans – Passos 1 e 2

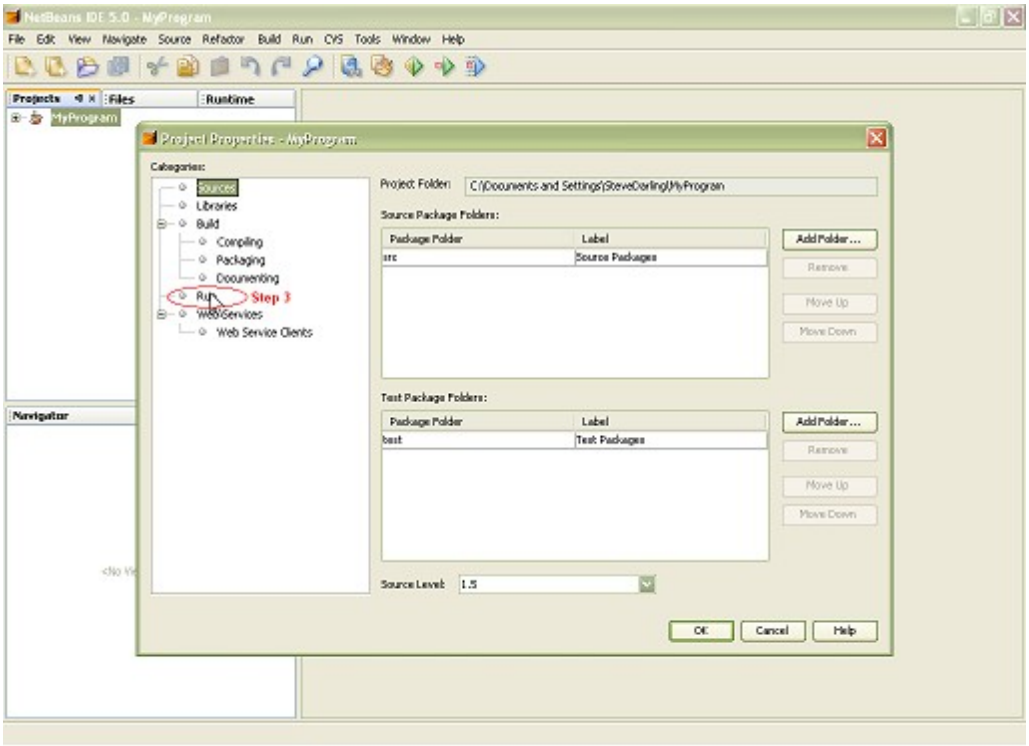

Figura 2: Habilitando assertivas no NetBeans – Passo 3

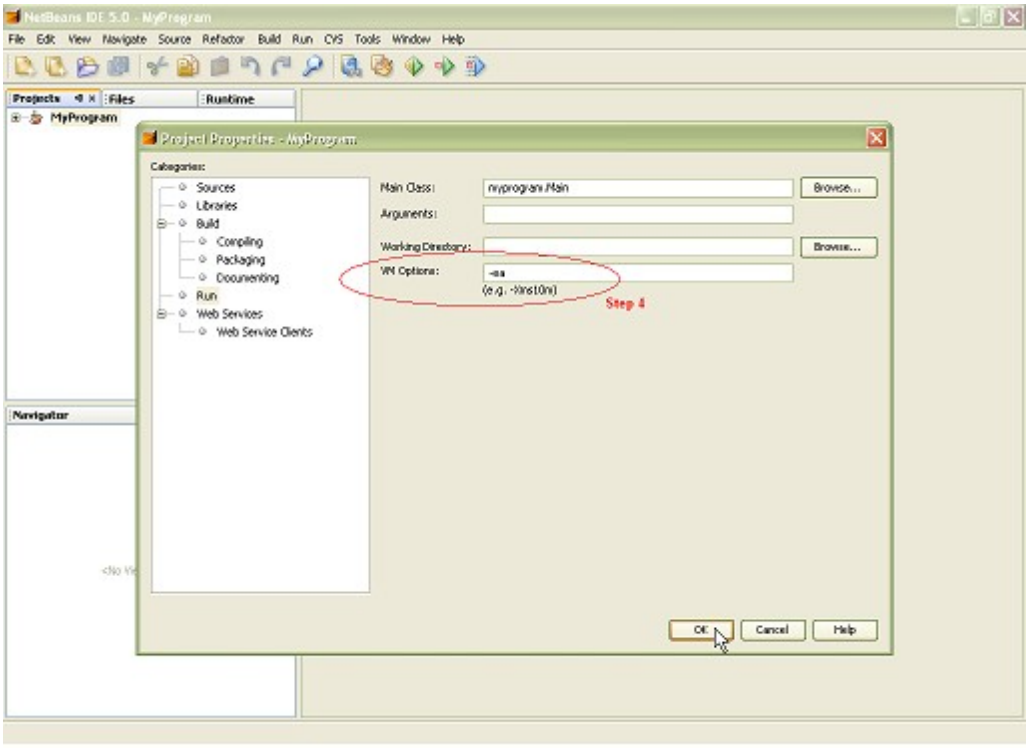

Figura 3: Habilitando assertivas no NetBeans – Passo 4

6.3. Sintaxe das Assertivas

A sintaxe das assertivas pode ser feita de duas formas.

A forma mais simples tem a seguinte sintaxe:

```
assert <expressãoLógica>;
```

onde <*expression1*> é a condição que é afirmada para ser verdadeira.

A outra forma utiliza duas expressões como demostra a sintaxe abaixo:

```
assert <expressãoLógica> : <mensagem>;
```

onde <*expression1*> é a condição que é afirmada para ser verdadeira e <*expression2*> é alguma informação útil para diagnosticar o motivo da instrução ter falhado.

Aqui temos um exemplo da forma mais simples de utilizar assertivas:

```
class AssertDemo {
   public static void main(String args[]) {
      assert(args == null);
      int age = Integer.parseInt(args[0]);
      if (age >= 18) {
         System.out.println("Congrats! You're an adult! =)");
      }
   }
}
```

A execução desta classe lança um *AssertionError* quando não existe argumento passado para a classe. Nesse caso, a seguinte mensagem de erro é apresentada na execução.

```
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:4)
Java Result: 1
```

Passando como argumento um valor maior que 0 e menor que 18, a classe não apresentará nenhuma saída.

Para um argumento com valor maior ou igual a 18, será mostrada a seguinte saída:

```
Congrats! You're an adult! =)
```

As instruções seguinte é similar ao exemplo anterior exceto por apresentar uma informação adicional sobre a exceção ocorrida. Dessa forma utiliza-se a segunda sintaxe das assertivas.

```
class AssertDemo {
   public static void main(String args[]) {
      int age = Integer.parseInt(args[0]);
      assert(age>0) : "age should at least be 1";
      /* se a idade é válida (ex. age>0) */
      if (age >= 18) {
         System.out.println("Congrats! You're an adult! =)");
      }
   }
}
```

Quando o usuário entra com um argumento menor do que 1, ocorre um *AssertionError* e são apresentadas informações adicionais da exceção. Nesse cenário, a seguinte saída é apresentada.

```
Exception in thread "main" java.lang.AssertionError: age should at
least be 1
        at AssertDemo.main(AssertDemo.java:4)
Java Result: 1
```

Para os outros casos, esta classe nos apresenta uma saída similar ao exemplo anterior.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.