Aula 11 - Tópicos Extras em Java

Docupedia Export

Author:Silio Leonardo (SO/OPM-TS21-BR)
Date:16-May-2024 14:52

Table of Contents

- Tratamento de Erros
- Componentes Estáticos
- Enums
- Interfaces
- Pacotes

1 Tratamento de Erros

O sistema de erros do Java é um pouco mais complexo do que em outras linguagens. Como você pode ver na figura nas heranças entre as classes. No Java existe um sistema de Checked e Unchecked exceptions. Basicamente, as exceções que herdam de RuntimeException são Unchecked Exceptions, ou seja, são erros inesperados que não deveriam acontecer e que vão explodir a aplicação. Já as Checked Exceptions são quaisquer outras exceções que herdem de Exception e filhos, exceto é claro RuntimeException. Essas exceções representam erros que são esperados e indicam que algo ocorreu, mas trata-se de um fluxo alternativo do problema. Exemplo: O usuário tentar carregar um arquivo que não existe é um erro esperado e o sistema deve tratar apresentando mensagens. Já o código acessar fora de um vetor indica que o algoritmo está errado e o software está inconsistente.
Você pode fazer seus próprios erros pensando neste conceito apenas herdando da classe correta.

Você usará **throw** para lançar seus próprios erros. Camadas superiores da aplicação podem lidar com este erro da forma que acreditarem ser melhor. Também pode filtrar erros usando o bloco **try-catch**, mas vamos compreender toda essa dinâmica a seguir.

```java
public class ErroInesperado extends RuntimeException { }
```

```java
class ErroEsperado extends Exception
{
    String title;
    ErroEsperado(String title)
    {
        this.title = title;
    }
}
```

```java
import java.util.Random;

class Main
{
    public static void main(String[] args)
    {
        // Tenta executar, se ocorreu um erro, o bloco catch é executado
        try
        {
            Random random = new Random();
            int valor = random.nextInt() % 3;
            if (valor == 0)
            {
```

```
14                  funcA();
15              }
16              else if (valor == 1)
17              {
18                  // Se funcB não fosse chamado aqui, o catch
19                  // de ErroEsperado não poderia existir
20                  // da mesma forma, se o catch não existisse
21                  // a funcB não poderia ser chamada aqui.
22                  funcB();
23              }
24              return;
25          }
26          // Catch é executado a depender do erro encontrado
27          catch (ErroEsperado ex)
28          {
29              System.out.println(ex.title);
30          }
31          catch (ErroInesperado ex)
32          {
33              System.out.println("erro inesperado aconteceu");
34          }
35          // O finally é SEMPRE executado após o Try ou Catch,
36          // mesmo que tenha um return no meio dos catchs ou try
37          finally
38          {
39              System.out.println("Fim!");
40          }
41      }
42
43      static void funcA()
44      {
45          // Criamos objetos para estourar erros
46          throw new ErroInesperado();
47      }
48
49      // Checked Exceptions precisam ser sinalizados na função se não tratados
50      static void funcB() throws ErroEsperado
51      {
52          // Podemos mandar dados pra exceção se ela tiver variávies/Construtores
```

```
53            throw new ErroEsperado("erro na funcB");
54        }
55
56    }
```

2 Componentes Estáticos

Em Java como em muitas outras OO podemos fazer objetos globais que são chamados de estáticos. Também podemos por apenas alguns métodos ou váriaveis como estáticas e chama-lás globalmente:

```java
public class TicTacToe
{
    // Conta quantos objetos de jogo da velha foram criados
    private static int gameCount = 0;
    static int getTotalGameCount() {
        return gameCount;
    }

    private int[] values;
    TicTacToe() {
        gameCount++;
        values = new int[9];
    }

    // implementações aqui

    // Ao invés de um construtor, podemos fazer métodos para criação de objetos.
    // A ideia filosófica por trás disso é indicar ao usuário que a criação do
    // objeto requer uma operação complexa e isso deve ser levado em conta pois
    // garante que nenhum programador abusará da criação de objetos achando
    // que é um processo simples como apenas inicializar dados.
    static TicTacToe CreateRandom() {
        // Cria e retorna um jogo da velha
        // numa posição aleatória
    }
}

// Na Main.java
TicTacToe game = TicTacToe.CreateRandom();
System.out.println(TicTacToe.getTotalGameCount());
```

Note que a função Main é estática e por isso é global para que o programa java possa executá-la de fato.
Você também pode fazer classes inteiramente estáticas privando seu contrutor e garantindo que não existirão classes bases:

```
final class GameOptions
{
    private GameOptions() { }

    private static int difficultLevel = 2;
    static setDifficultLevel(int newDifficult) {
        this.difficultLevel = newDifficult;
    }
}
```

3 Enums

Vamos supor que você queria descrever uma váriável categórica. Por exemplo, o estado de um aluno na disciplina entre cursando, aprovado e reprovado. Você poderia representar usando uma classe EstadoCurso:

```
class EstadoCurso
{
    static EstadoCurso Aprovado = new EstadoCurso(true, false);
    static EstadoCurso Reprovado = new EstadoCurso(false, true);
    static EstadoCurso Cursando = new EstadoCurso(false, false);

    private boolean aproved;
    private boolean reproved;

    EstadoCurso(boolean aproved, boolean reproved)
    {
        this.aproved = aproved;
        this.reproved = reproved;
    }

    boolean IsAproved() {
        return this.aproved;
    }

    boolean IsReproved() {
        return this.reproved;
    }
}
```

É uma classe muito útil e interessante para diversos cenários:

```
class Main
{
    public static void main(string[] args)
    {
        EstadoCurso estado = EstadoCurso.Aprovado;
        if (estado == EstadoCurso.Aprovado) // Ou simplismente estado.IsAproved()
            System.out.println("Parabéns!");
    }
```

```
}
```

Isso é feito em várias linguagens e é a ideia básica de Enum, mas em Java podemos reescrever o primeiro código de forma simplificada e mais bonita:

```
enum EstadoCurso
{
    Aprovado(true, false), Reprovado(false, true), Cursando(false, false);

    private boolean aproved;
    private boolean reproved;

    EstadoCurso(boolean aproved, boolean reproved)
    {
        this.aproved = aproved;
        this.reproved = reproved;
    }

    boolean IsAproved() {
        return this.aproved;
    }

    boolean IsReproved() {
        return this.reproved;
    }
}
```

4 Interfaces

Interfaces não são telas nem nada visual, na verdade são como classes abstratas. A diferença é que elas não podem obrigar as classes filhas a terem implementações e dados específicos, ou seja, você não pode por funções já implementadas ou variávies o que tornaria as classes filhas obirgatoriamente mais pesadas. Interfaces são ótimas escolhas quando você só quer exigir que uma classe tenha uma determinada funcionalidade. Enquanto uma classe A que herda da classe B indica que A **é** B, por exemplo, um gato é um aimal, a interface representa qualidades funcionais de coisas não correlacionadas. Por exemplo, a classe gato implementa a interface PodeAndar bem como carro, e ambas as classes não tem nada haver uma com a outra. Note que quando uma classe herda da interface dizendo que ela implementa a interface, indicando ainda mais que é mais uma questão contratual do que uma herança em si. Também, enquanto uma classe só pode herdar de uma única classe, ela pode implementar várias interfaces. Vamos ver um exemplo de sintaxe para poder começar a explorar o recurso:

```
1  // Constumamos usar adjetivos para nomes de interfaces. Addable é 'adicionável' em inglês.
2  interface Addable
3  {
4      // Especificamente em java, podemos pedir a existência de variávies,
5      // embora seja mais recomendado adicionar gets como um getCount() aqui
6      int count = 0;
7
8      // Não definimos corpo nem abstract aqui.
9      // A depender da versão você pode adicionar uma
10     // implementação padrão que será usada caso não
11     // exista uma implementação para ela
12     void add(int num);
13 }
14
15 // Podemos ter interfaces genéricas também
16 interface Comparable<T>
17 {
18     boolean Compare(T obj);
19 }
20
21 // Exemplo incompleto de class implementando inerfaces
22 class MyList implements Addable, Comparable<int[]>
23 {
24     void add(int num)
25     {
26         // implmentação aqui
27     }
28
29     boolean Comparable(int[] data)
```

```
30      {
31          // implmentação aqui
32      }
33  }
34
35  // Usando interfaces
36  void adicionarPrimos(Addable addable)
37  {
38      addable.add(2);
39      addable.add(3);
40      addable.add(5);
41      addable.add(7);
42  }
43
44  Addable myList = new MyList();
45  adicionarPrimos(myList);
46  int[] array = new int[] { 2, 3, 5, 7 };
47  if (myList.Compare(array))
48      System.out.println("Ok!");
```

5 Pacotes

Você pode organizar seu código usando pacotes. Pacotes são estruturas refletidas nas pastas do projeto que lhe permitiram esconder objetos e organizá-los:

```java
// myproject/datastructures/MyList.java
package myproject.datastructures;

// public = Pode ser vista fora do pacote (por isso a main é publica)
public class MyList
{
    // protected = Pode ser vista fora do pacote, somente se, por uma classe filha
    // ou seja, classes que herdam de MyList.
    protected int size = 0;
}

// myproject/Main.java
package myproject;

import myproject.datastrucutures.MyList;
// ou 'import myproject.datastrucutures.*;' para importar tudo

public class Main
{
    public static void main(String[] args)
    {
        MyList list = new MyList();
    }
}
```