# PROGRAMAÇÃO ORIENTADA A OBJETOS

AULA 6

Prof. Leonardo Gomes

# CONVERSA INICIAL

Nesta aula, vamos abordar a princípio o tratamento de erros e exceções na linguagem Java, como um programa deve se portar diante de situações inesperadas utilizando os comandos ***try*** e ***catch***. Na sequência, iremos debater formas de criar as próprias exceções e, por fim, vamos discutir outras palavras reservadas importantes dentro do Java.

**Objetivos da aula**: ao final desta aula, esperamos atingir os seguintes objetivos, que serão avaliados ao longo da disciplina da forma indicada.

Quadro 1 – Objetivos

| Objetivos | Avaliação |
|---|---|
| 1 - Entendimento do conceito de exceção e seu devido tratamento utilizando os *try catch*. | Questionário e questões dissertativas |
| 2 - Ser capaz de desenvolver as próprias exceções. | Questionário e questões dissertativas |
| 3 - Dominar o conceito de alguns métodos padrões importantes dentro do Java como *toString* e *equals*. | Questionário e questões dissertativas |

# TEMA 1 – TRATAMENTO DE EXCEÇÕES

Neste tema, vamos abordar a questão das exceções e a forma de lidar com elas na linguagem Java. Primeiramente, vamos definir algumas questões. Dentro do Java, temos o conceito do erro e da exceção.

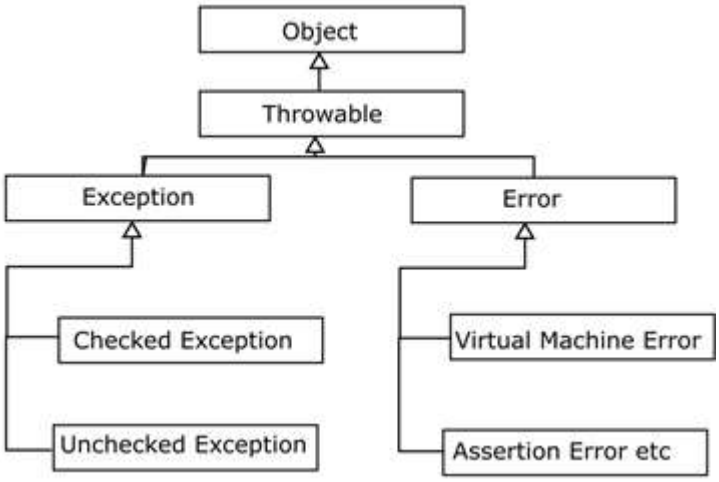
**Erro (Error)**: é um problema sério que ocorre em tempo de execução, impossível para o compilador detectar e que geralmente não tem tratamento para ele. No geral, são problemas na plataforma que está rodando o programa Java.

Por exemplo, a falta de memória para alocar recursos necessários do programa gera um erro que, na maioria das aplicações, não existirá meio de se contornar e, na maioria das vezes, o programa deverá simplesmente ser interrompido. A solução real seria realmente agir na plataforma em que o *software* está rodando e cabe ao programa gerar um *log* (relatório), ou alerta descrevendo o evento. O Java, ao detectar esse problema, lança esse erro no formato ***java.lang.OutOfMemoryError.***

**Exceção (Exception)**: também é um problema, mas que geralmente pode ser manejado pelo programa e tratado de alguma forma. Por exemplo, ao tentar ler dados em um arquivo que foi apagado e não existe mais, irá gerar a exceção chamada pelo Java de ***FileNotFoundException***. Nesta situação, geralmente é possível em vez de simplesmente encerrar o programa repentinamente, apresentar uma mensagem para o usuário e solicitar alguma intervenção e seguir com o programa funcionando.

Tanto os erros quanto as exceções no Java, ao ocorrerem, são identificadas e lançadas de uma forma que determinados comandos conseguem recuperar o controle do programa e impedir que ele seja encerrado subitamente. Erros e exceções geram subclasses de uma superclasse ***Throwable*** e respeitam a hierarquia da figura a seguir.

Figura 1 – Representação da hierarquia das classes de erros e exceções do Java




Quando ocorre uma exceção/erro em um método, ele gera um objeto do tipo específico e envia para a máquina virtual Java (JVM). O objeto *Exception* contém o nome, uma descrição e o estado do programa no momento que ocorreu o problema. A ordenação das chamadas dos métodos é conhecida como ***Call Stack*** (pilha de chamadas). Depois isso, ocorre a seguinte sequência de passos.

- A JVM busca na *call stack* um método que contenha um bloco de código capaz de tratar a exceção, vamos chamar de **bloco tratador de exceção**. A busca inicia no método que gerou a exceção e segue ordem reversa de chamada.
- Ao encontrar um bloco tratador apropriado, ele recebe o controle do fluxo de código junto ao objeto exceção. Por apropriado, entendemos que ele trata um tipo de exceção compatível com a que foi gerada.
- Se a JVM não encontrar um bloco tratador de exceções na *call stack*, então a JVM utiliza um bloco tratador de exceções padrão que irá encerrar o programa imediatamente e imprimir uma mensagem no console dando todas as informações da exceção, nome, descrição e a *call stack*.

## 1.1 LIDANDO COM EXCEÇÕES

Na linguagem Java, assim como nas principais linguagens de programação, existem os comandos ***try*** (tentar) e ***catch*** (capturar). O comando *try* é associado a um bloco de código que será executado e caso um erro ou exceção aconteça o programa não encerrará automaticamente com mensagem de alerta no console, mas, sim, o controle será devolvido ao programa dentro do bloco *catch* adequado. Opcionalmente, é possível colocar o comando ***finally*** (finalmente) que também é associado a um bloqueio de código que será executado independente de o código ter entrado ou não no bloco *catch*.

Vamos ver um exemplo de código a seguir.

```
01.     public class MinhaClasse {
02.     public static void main(String[] args) {
03.      try {
04.         int[] meusNumeros = {1, 2, 3};
05.         System.out.println(meusNumeros[10]);
06.         } catch (Exception e) {
07.           System.out.println("Problema = " + e);
08.         } finally {
09.           System.out.println("Terminado o try catch.");
10.         }
11.       }
12.     }
```

Na linha 3, temos o início do bloco *try*, observe que, na linha 5, acessamos a posição 10 de um *array* com apenas três itens, isso gera uma ArrayIndexOutOfBoundsException sem o *try*, nesse momento, o programa iria interromper imediatamente, porém, com o *try*, a exceção gerada pelo sistema será capturada pelo *catch* e em vez de interromper a mensagem problema, será impressa na tela. Observe que é possível imprimir o objeto exceção, o que trará nome e descrição dela. Por fim, o bloco existente dentro do *finally* (linha 8) será executado.

## TEMA 2 – CRIANDO AS PRÓPRIAS EXCEÇÕES

Além de tratar as exceções e erros que o sistema lança por padrão, podemos lançar exceções e até mesmo criar nossas próprias exceções. A ideia por trás de lançar exceções está em evitar que o programa continue executando mediante situações anormais e devolver o controle para o método tratador adequado.

Por exemplo, se o seu programa falhar em conectar com o banco de dados, o sistema lançará uma exceção de forma automática, provavelmente do tipo java.sql.SQLException. No entanto, se o banco de dados conectar adequadamente, mas um dado crítico para o funcionamento do seu programa não estiver presente na base dados, então não existirá uma exceção e, caso a situação não

seja devidamente identificada e tratada, o programa continuará em execução com um comportamento imprevisível.

Este é o cenário adequado para se lançar uma exceção própria, identificando que o dado crítico não está presente na base de dados e isso pode ser feito por meio do comando ***throw.*** Esse comando é capaz de explicitamente lançar uma exceção qualquer. Confira o código a seguir.

```
01.     class Principal{
02.         static void funcao(){
03.             try{
04.               throw new
   NullPointerException("Problema!");
05.             }
06.             catch(NullPointerException e){
07.                 System.out.println("funcao() :" + e);
08.                 throw e; // jogando a exceção novamente
09.             }
10.         }
11.         public static void main(String args[]){
12.             try{
13.                 funcao();
14.             }
15.             catch(NullPointerException e){
16.                 System.out.println("main() :" + e);
17.             }
18.
19.         }
20.     }
```

Nesse código, na linha 4, explicitamente criamos um objeto do tipo exceção *NullPointerException*, que geralmente é associado a um ponteiro nulo em uma situação inapropriada.

No construtor do ponteiro, colocamos a mensagem *Problema*, podendo associar a mensagem que desejarmos nas exceções.

Na linha 6, a exceção será capturada, e uma mensagem será impressa na tela e o erro lançado novamente. Como não existe outro bloco *try* englobando esse *throw*, a exceção sobe na pilha de chamadas de função até encontrar um *try*, no caso na *main* que chamou o método funcao() (linha 13) e, por estar em um bloco *try/catch*, também capturou na linha 15 e imprime uma mensagem final.

## 2.1 TIPOS DE EXCEÇÕES

Na Figura 1, vemos que temos duas subclasses de *Exception* e são os dois tipos diferentes de exceções no Java:

1) ***Checked***: são exceções verificadas em tempo de compilação, *checked* em inglês significa checada. Isso significa que ao lançar uma exceção desse tipo o método deve tratar ela com *try/catch* ou explicitamente anunciar na assinatura do método que pode lançar esse tipo de exceção com a palavra *throws*, caso não faça nenhuma das duas coisas, o compilador irá gerar um erro. Vamos analisar o código a seguir:

```
01.     class Principal {
02.         public static void main(String[] args) {
03.             FileReader arquivo = new
04.     FileReader("teste.txt");
05.             BufferedReader entrada = new
06.     BufferedReader(file);
07.             // Ler e imprimir 3 linhas de teste.txt"
08.             for (int i = 0; i < 3; i++)
09.                 System.out.println(entrada.readLine());
10.             fileInput.close();
11.      }
12.     }
```

No código acima, três linhas são lidas e impressas na tela do arquivo “teste.txt”. No entanto, o código não funcionará, pois a classe FileReader lança uma exceção IOException do tipo **checked**. Para o compilador aceitar o código, será necessário utilizar bloco *try/catch* ou na assinatura do método explicitar que a exceção pode ser lançada.

```
public static void main(String[] args) throws IOException {
```

Todo o método A que explicitamente lance uma *exception* na assinatura obriga que, ao ser invocado por um outro método B, ele deva ou tratar com *try/catch* ou ele também lançar a *exception* na sua assinatura. No caso da *main*, se ela lançar a exceção para quem a invocou, o programa será interrompido imediatamente com mensagem no console informando a exceção.

- ***Unchecked***: são exceções que não são checadas em tempo de compilação, dispensam o uso de *throws* na assinatura da função quando não tratadas, fica livre ao programador decidir se precaver com o uso de *try/catch* ou não. Problemas do tipo *Error* também não são checados, na hierarquia de classes presente na Figura 1, as exceções não checadas também são chamadas de ***RuntimeException***.

```
01.     class Principal {
02.         public static void main(String args[]) {
03.             int dividendo = 0;
04.             int divisor = 10;
05.             int resultado = dividendo/divisor;
06.         }
07.     }
```

Esse código irá gerar uma *RunTimeException* do tipo java.lang.ArithmeticException.

## 2.2 DESENVOLVENDO SUA PRÓPRIA *EXCEPTION*

Além de lançar as exceções existentes, é possível criar a sua própria exceção subclasse de *Exception* ou *RuntimeException* e até mesmo *Error*. Ao criar uma exceção que estende *Exception,* ela será do tipo checada e, se ela estender *RuntimeException*, será do tipo não checada. Basta criar uma subclasse e ela poderá ser utilizada para descrever uma situação inesperada.

No exemplo a seguir, temos um código que personaliza uma *Exception* para a situação de não encontrar um usuário em um banco de dados.

```
01.     public class UsuarioInexistenteException extends
   Exception {
02.       public UsuarioInexistenteException(String
   mensagem){
03.         super(mensagem);
04.       }
05.     }
```

*Classe* possui o método *buscar* que, caso não encontre o usuário, lança a exceção personalizada.

```
01.     public class Gerenciador {
02.         public Usuario buscar(String usuarioID) throws
03.     UsuarioInexistenteException {
04.             if (usuarioID.equals("123456")) {
05.                 return new Usuario();
06.             } else {
07.                 throw new
   UsuarioInexistenteException("Nao existe usuario " +
   studentID);
08.             }
09.         }
10.     }
```

Método principal aplicando o tratamento com *try/catch*.

```
01.     public class Teste {
02.       public static void main(String[] args) {
03.         Gerenciador gerenciador= new Gerenciador();
04.          try {
05.             Usuario usr = gerenciador.buscar("0000001");
06.         } catch (UsuarioInexistenteException ex) {
07.             System.err.print(ex);
08.         }
09.       }
10.     }
```

Dessa forma, outros problemas que no contexto de uma aplicação seja conveniente serem tratados como exceção também poderão ter suas exceções personalizadas.

## TEMA 3 – IGUALDADE

Neste tema, vamos debater questões relativas à igualdade no Java. Quando desejamos comparar se dois elementos são iguais no Java, é comum pensarmos no comando ==. Por exemplo:

x == 10  'a' == 'b'

No entanto, esse comando embora compare adequadamente primitivas (que são variáveis básicas: *int*, *char*, *float* etc.), ele tem uma funcionalidade diferente ao comparar objetos, o comando retornará verdadeiro apenas se o endereço dos objetos for o mesmo, ou seja, se são exatamente a mesma instância e não se o conteúdo é o mesmo. Por exemplo, no Java as *Strings* são objetos e não primitivas, portanto, não é possível comparar o conteúdo com ==, para isso utilizamos um método chamado *equals*. Vejamos o código a seguir.

```
public class Teste {
    public static void main(String[] args)
    {
        String s1 = new String("Ola");
        String s2 = new String("Ola");
        String s3 = s1;
        System.out.println(s1 == s2); //falso
        System.out.println(s1.equals(s2)); //verdadeiro

        System.out.println(s1 == s3); //verdadeiro
    }
}
```

Nesse exemplo, quando comparamos s1 com s2 utilizando ==, observe que o resultado é falso justamente pelo fato de serem instâncias diferentes. Agora s1 e s3 são iguais, pois tratam da mesma instância, o exato mesmo objeto, se realizarmos uma mudança em s3, vai afetar s1 e vice-versa, pois ambos apontam para a mesma região na memória.

O método ***equals*** é um padrão dentro do Java e serve justamente para comparar conteúdos, não é obrigatório implementar o método *equals* para todas as classes, mas é uma convenção para aquelas classes que faça sentido a comparação entre objetos. Essa padronização garante a compatibilidade com certas classes do sistema também. A seguir, criamos uma classe simples para representar um usuário de um sistema e um método *equals* que compara ambos:

```
01.     public class Usuario {
02.          int id;
03.          String nome;
04.          String cpf;
05.          public Usuario(int id, String nome, String
   cpf) {
06.               super();
07.               this.id = id;
08.               this.nome = nome;
09.               this.cpf = cpf;
10.          }

11.          public boolean equals(Object outro) {
12.             // Compara consigo mesmo
13.             if (outro == this) {
14.                 return true;
15.             }

16.             //Objeto outro é uma instância de Usuario?
17.             if (!(outro instanceof Usuario)) {
18.                 return false;
19.             }

20.             // type cast para Usuario
21.             Usuario o = (Usuario) outro;

22.             //Compara os atributos são iguais
23.                if( this.id == o.id &&
24.                     this.nome.equals(o.nome) &&
25.                     this.cpf.equals(o.cpf)) {
26.                          return true;
27.                }
28.                return false;
```

```
29.          }
30.     }

31.     public class Principal {
32.          public static void main(String[] args) {

33.               Usuario m1 = new
34.     Usuario(1,"Mario","111.222.333-44");
35.               Usuario m2 = new
   Usuario(1,"Mario","111.222.333-44");
36.          Usuario m3 = new
   Usuario(2,"Luigi","555.666.777-88");
37.
38.               //verdadeiro
39.               System.out.println(  m1.equals(m2)  );
40.               //falso
41.               System.out.println(  m1 == m2  );
42.               //falso
43.               System.out.println(  m1.equals(m3)  );

44.          }
45.     }
```

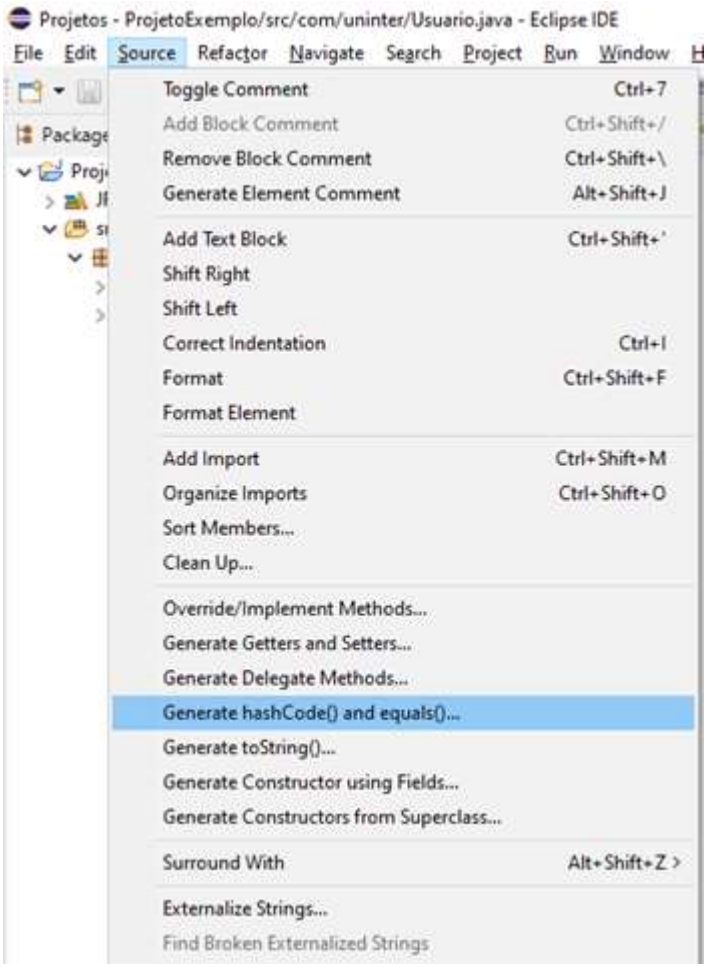
Na linha 11, repare que o método *equals* não recebe um Usuário como parâmetro, mas, sim, um ***Object***, que é a classe mais geral de todas em Java. Por padrão todas as classes que não tenham uma superclasse são subclasses de *Object*, o que garante que todo o objeto instanciado pode ser entendido como um *Object*.

Em outras palavras, é possível, dessa forma, comparar uma instância de Usuário com qualquer coisa. Com isso, até mesmo uma instância de uma subclasse de Usuário poderia ser comparada a uma outra instância de Usuário e se tiverem os mesmos valores será retornado verdadeiro.

Na linha 21, observe o uso da técnica chamada de ***type cast***, quando temos um objeto de uma classe e queremos indicar aquele objeto, pode ser interpretado como uma outra classe, a colocamos entre parênteses da forma que foi feita para evitar que ocorra um erro de compilação. Observe que isso não executa nenhuma transformação nos dados, apenas evita o erro de compilação, se a transição funcionará é responsabilidade do programador que realizou o comando.

Diversas IDE Java, incluindo o Eclipse, conseguem gerar o código do *equal* automaticamente na opção ***Source/Generate hashCode() and equals()...***

Figura 2 – Menu Eclipse para criação de código automático




## TEMA 4 – MÉTODOS ESPECIAIS: *TOSTRING*

Neste tema, vamos apresentar mais um método padrão que podemos sobrescrever no Java e que apresenta facilidades ao lidar com texto o método ***toString()***.

Supondo a situação em que temos um objeto e desejamos descrevê-lo em formato texto para realizar um *log*, depuração ou mesmo apresentar o conteúdo de um objeto diretamente para o usuário do programa. O comum seria acessar cada atributo individualmente, no entanto, podemos acessar o objeto diretamente no formato *string* e especificar por meio do método *toString*(), como essa string será, indicando até mesmo atributos e resultados de métodos.

Como já foi discutido anteriormente, todo o objeto Java é diretamente ou indiretamente subclasse de *Object*. E a classe *Object* possui o método *toString*() que, por padrã,o retorna uma *String* com o endereço na memória daquele objeto. Esse método *toString*() é implicitamente acessado sempre que tentamos ler o objeto no formato de uma *string*, por exemplo, ao tentar imprimir um objeto.

```
Usuario usr = new Usuario();
System.out.println(usr);
```

Ao executarmos esse código, o endereço de memória no qual foi instanciado o objeto usr aparecerá na tela. Para que, ao fazermos isso, apareçam os dados do objeto usuário, podemos sobrescrever o método *toString*, como no exemplo a seguir.

```
01.     public class Usuario {
02.          int id;
03.          String nome;
04.          String cpf;

05.          @Override
06.          public String toString() {
07.               return "Usuario [id=" + id + ", nome=" +
   nome
   + ", cpf=" + cpf + "]";
08.          }
09.     }
```

//Na main()

```
01.     Usuario m1 = new Usuario(1,"Mario","111.222.333-
   44");
02.     System.out.println(m1);
```

No caso geral, ao tentarmos imprimir um objeto da classe Usuário, teríamos uma mensagem como Usuario@232204a1, o nome da classe seguido de um endereço de memória em que o objeto foi instanciado. Mas ao criarmos o método *toString,* essa mensagem é substituída pelo resultado do método.

No código anterior, na linha 6, vemos o método *toString*, que retorna uma *string* que condiz a como desejamos representar o objeto no formato *String*.

Na linha 5, temos o comando **@Override**, que serve para indicar que o método está sobrescrevendo outro de sua superclasse, esse comando não é necessário, no entanto, ele possui dois papéis: indicar aos programadores que lerão o código que aquele é um método sobrescrevendo outro e fazer com que o compilador verifique a adequação da assinatura do método.

Também é possível em diversas IDEs Java, incluindo eclipse, gerar o código automaticamente para o método *toString,* indo na opção *Source/Generate toString*().

## TEMA 5 – MODIFICADORES JAVA (*SINGLETON*)

Neste tema, vamos abordar um padrão de desenvolvimento muito importante dentro das linguagens de programação orientadas a objeto, o padrão *Singleton*.

Certos problemas dentro da programação, de tão clássicos e recorrentes, possuem soluções que se tornam referência e são usadas de forma padrão em projetos profissionais, na literatura, essas soluções padronizadas de problemas recorrentes são chamadas de ***design pattern***.

Um problema recorrente é a necessidade de termos um objeto que seja instanciado uma única vez e sempre que seja solicitado pelas classes de um projeto seja direcionado ao mesmo objeto. Um conceito semelhante ao de uma variável global, porém, utilizando uma instância.

Dentro da literatura de *design pattern*, a solução para essa situação é a utilização do que chamamos de ***Singleton***. Alguns exemplos comuns de uso do padrão *Singleton* é quando temos algum tipo de recurso compartilhado. Por exemplo, diversas classes requisitando acesso a uma impressora, banco de dados ou um mesmo arquivo não é interessante que esses recursos sejam acessados paralelamente por múltiplas instâncias, poderia causar conflitos entre as requisições, nesse tipo de situação, *Singleton* seria bem aplicado.

Outro exemplo comum é a classe responsável por efetuar *log* de atividades. Essa tarefa é constantemente invocada em diferentes etapas de um sistema e ter uma única instância realizando-a minimiza o *overhead* (trabalho extra de gestão) da classe, visto que essa operação é realizada de forma constante e relativamente independentemente do restante do projeto.

Na prática, para criar uma classe *Singleton*, são necessários dois passos:

1. Ter um construtor privado.

2. Criar um método estático que retorna um objeto da classe em questão instanciado. Conceito conhecido como *Lazy initialization* (inicialização preguiçosa), em que o objeto é criado depois de declarado.

A seguir, o código apresenta uma classe *Singleton* que pode ser instanciada uma única vez:

```
class Singleton
{
    // variável estática que armazenara a nossa única
instancia
    private static Singleton instancia = null;

    // variavel para teste
    public int numero;

    //construtor privado
    private Singleton()
    {
        numero =20;
    }

    // método estático para criar a instancia
    public static Singleton getInstance()
    {
        if (instancia == null)
            instancia = new Singleton();
        return instancia;
    }
}

class Principal
{
    public static void main(String args[])
    {
        // instantiating Singleton class with variable x
        Singleton x = Singleton.getInstance();

        // instantiating Singleton class with variable y
```

```
        Singleton y = Singleton.getInstance();

        // instantiating Singleton class with variable z
        Singleton z = Singleton.getInstance();

        // changing variable of instance x
        x.numero+=10;

        System.out.println("x: " + x.numero);
        System.out.println("y: " + y.numero);
        System.out.println("z: " + z.numero);
        System.out.println("\n");
        //Todos imprimem 30

        z.numero-=5;

        System.out.println("x: " + x.numero);
        System.out.println("y: " + y.numero);
        System.out.println("z: " + z.numero);
        System.out.println("\n");
        //Todos imprimem 25

    }
}
```

Nesse código, o método *getInstance*() é utilizado para recuperar a instância, já que o construtor é privado e, portanto, inacessível de fora da classe. Observe que a variável número é a mesma para as variável x, y, z, afinal, trata-se da mesma instância, mesma posição de memória internamente.

O método *getInstance* verifica se o objeto já foi instanciado. Caso positivo, ele retorna o objeto e, caso contrário, ele efetivamente cria a instância antes de retorná-la.

## FINALIZANDO

Nesta aula, abordamos temas diversos, o tópico principal foi a gestão de erros e exceções. Como evitar que uma situação imprevisível faça perder o controle do programa e como gerar as próprias exceções para informar situações fora da normalidade.

Na sequência, discutimos dois métodos importantes que existem na classe *Object*, tradicionalmente sobrescrita, sendo eles o método *equals*, que compara o conteúdo de objetos distintos, e o método *toString*, que permite de forma prática tratar objetos como *strings* personalizadas.

Por fim, debatemos o conceito de *Singleton*, um *design pattern* bastante popular, aplicado a diversas linguagens de programação e que serve para garantir a existência de uma única instância.

## REFERÊNCIAS

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

DEITEL, P.; DEITEL, H. **Java**: como programar. 10. ed. São Paulo: Pearson, 2017

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2007.

MEDEIROS, E. S. de. **Desenvolvendo *software* com UML 2.0**: definitivo. São Paulo: Pearson Makron Books, 2004.

PAGE-JONES, M. **Fundamentos do desenho orientado a objetos com UML**. São Paulo: Makron Book, 2001

PFLEEGER, S. L. **Engenharia de *software***: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

SINTES, T. **Aprenda programação orientada a objetos em 21 dias**. 5. reimpressão. São Paulo: Pearson Education do Brasil, 2014.

SOMMERVILLE, I. **Engenharia de *software***. 9. ed. São Paulo: Pearson, 2011.