

CENTRO UNIVERSITÁRIO FACENS

CURSOS TECNOLÓGICOS – AS014TGN1



PROGRAMAÇÃO ORIENTADA A OBJETOS

CONTEXTO

Observe a figura abaixo, conforme o código temos um problema quando utilizamos a operação aritmética de divisão. Os valores de uma divisão podem ser imprevisíveis. Como resolver?

The image shows a Java IDE interface. On the left, a code editor displays a Java program. The code imports `java.util.*`, `java.lang.*`, and `java.io.*`. It defines a `Main` class with a `main` method. Inside the `main` method, variables `a` and `b` are declared and initialized to 9 and 0 respectively. Then, `c` is calculated as `a / b`, and the result is printed. The code is as follows:

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 // The main method must be in a class named "Main".
6 class Main {
7     public static void main(String[] args) {
8
9         int a = 9;
10        int b = 0;
11
12        int c = a / b;
13
14        System.out.println("c = " + c);
15    }
16 }
```

On the right side of the IDE, there are two buttons: "Execução" (Execution) and "Salvar" (Save). Below these buttons is a text input field labeled "Entrada do programa". Underneath that is a section titled "Saída do programa" (Program Output). This section contains the following text:

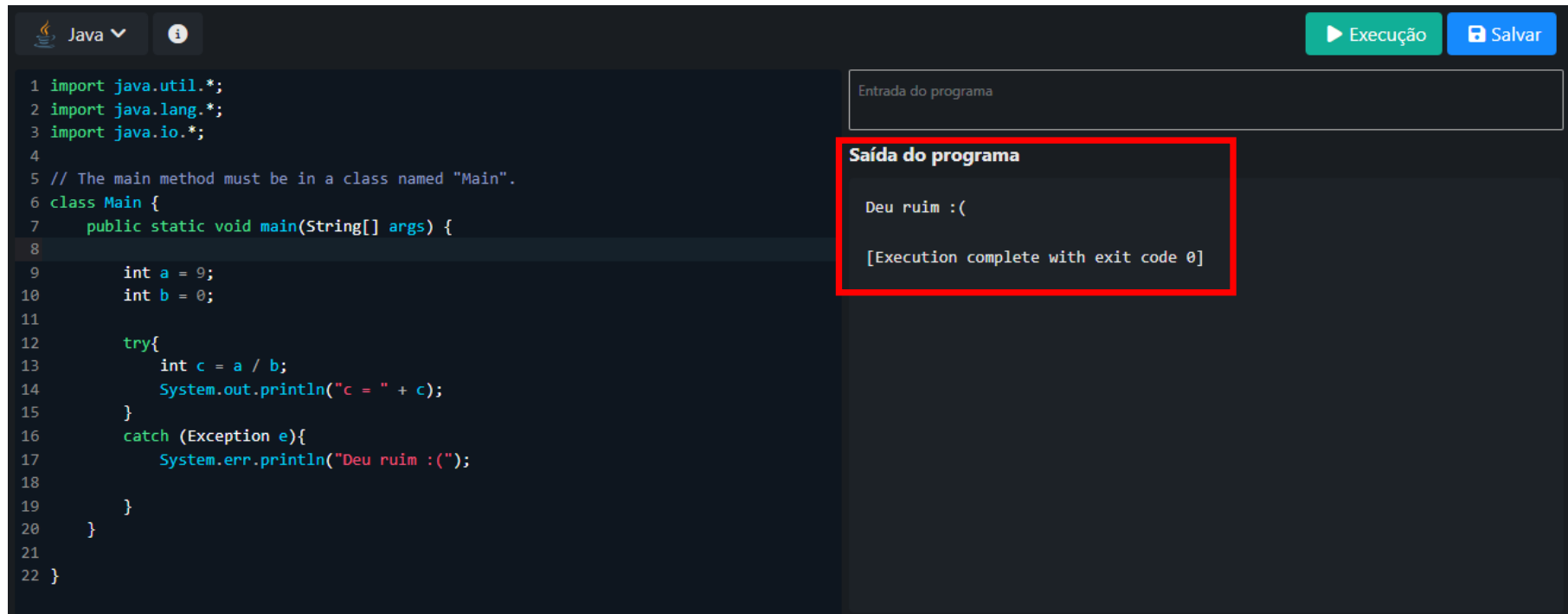
```
Exception in thread "main" java.lang.ArithmeticExcepti
    at Main.main(Main.java:12)

[Execution complete with exit code 1]
```

The output section is highlighted with a red rectangle, indicating the runtime error.

CONTEXTO

A instrução **try-catch** tenta executar um trecho de código e, caso falhe, executa o bloco **catch**.



The screenshot shows a Java IDE with a dark theme. On the left, a code editor displays a Java program. The code imports `java.util.*`, `java.lang.*`, and `java.io.*`. It defines a `Main` class with a `main` method. Inside the `main` method, two integers `a` and `b` are declared and initialized to 9 and 0 respectively. A `try` block contains the calculation `int c = a / b;` and the print statement `System.out.println("c = " + c);`. A `catch` block for `Exception e` contains the print statement `System.err.println("Deu ruim :(");`. On the right, there are two buttons: a green "Execução" button and a blue "Salvar" button. Below these buttons is a text input field labeled "Entrada do programa". Below the input field is a red-bordered box titled "Saída do programa" which contains the output: "Deu ruim :(" followed by "[Execution complete with exit code 0]".

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 // The main method must be in a class named "Main".
6 class Main {
7     public static void main(String[] args) {
8
9         int a = 9;
10        int b = 0;
11
12        try{
13            int c = a / b;
14            System.out.println("c = " + c);
15        }
16        catch (Exception e){
17            System.err.println("Deu ruim :(");
18        }
19    }
20 }
21
22 }
```

Entrada do programa

Saída do programa

Deu ruim :(

[Execution complete with exit code 0]

CONTEXTO

A instrução **try-catch** tenta executar um trecho de código e, caso falhe, executa o bloco **catch**.



The screenshot shows a Java IDE with a code editor on the left and a console on the right. The code in the editor is a Java class named 'Main' with a 'main' method. It uses a try-catch block to handle a division by zero exception. The try block contains a division of 'a' by 'b', and the catch block prints an error message. The console on the right shows the output of the program, which is 'Deu ruim :('. The 'Saída do programa' section is highlighted with a red box.

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 // The main method must be in a class named "Main".
6 class Main {
7     public static void main(String[] args) {
8
9         int a = 9;
10        int b = 0;
11
12        try{
13            int c = a / b;
14            System.out.println("c = " + c);
15        }
16        catch (Exception e){
17            System.err.println("Deu ruim :(");
18        }
19    }
20 }
21
22 }
```

Entrada do programa

Saída do programa

Deu ruim :(

[Execution complete with exit code 0]

CONTEXTO

Na POO, **encapsulamento** significa **esconder** atributos da classe de forma que somente ela pode acessá-lo. Essa classe pode fornecer **métodos** que permitem **acesso controlado** ao atributo. Esse recurso é utilizado para **proteger** dados de alterações indevidas, mas permitindo **acesso** por **intermédio** de métodos.

Uma classe pode controlar quais **atributos** e **métodos** são **acessíveis** a **outras classes**. Para isso, definimos **níveis** de **visibilidade**. Em Java, os modificadores são palavras-chave usadas para definir o comportamento, o nível de acesso e as propriedades das classes, métodos e variáveis. Eles podem ser divididos em dois grupos principais:

CONTEXTO

Modificadores de Acesso

Esses modificadores controlam a visibilidade de classes, métodos e atributos. São eles:

public: Permite o acesso de qualquer parte do código, de qualquer classe, em qualquer pacote.

protected: Permite o acesso dentro da própria classe, de subclasses (mesmo fora do pacote) e das classes do mesmo pacote.

private: Restringe o acesso apenas à própria classe. Sem modificador (default): O acesso é permitido apenas às classes dentro do mesmo pacote.

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 public class Exemplo {
6     public int publico;        // Acessível em qualquer lugar
7     protected int protegido;  // Acessível no mesmo pacote e subclasses
8     int padrao;                // Acessível no mesmo pacote
9     private int privado;       // Acessível apenas dentro da classe
10 }
11
```

CONTEXTO

Modificadores Não Acessíveis

Esses modificadores adicionam propriedades especiais ao comportamento de classes, métodos e variáveis.

static: Indica que o membro pertence à classe, e não à instância. Pode ser aplicado a métodos e variáveis.

Exemplo: static int contador;

final: Indica que a variável não pode ser modificada após a sua inicialização, o método não pode ser sobrescrito e a classe não pode ser estendida.

Exemplo: final int numeroConstante = 10;

abstract: Define classes ou métodos abstratos, que devem ser implementados por classes derivadas.

Exemplo: abstract void desenhar();

synchronized: Usado em métodos para controlar o acesso concorrente de múltiplas threads.

Exemplo: synchronized void metodoSincronizado() { /* código */ }

CONTEXTO

volatile: Garante que o valor da variável será sempre lido da memória principal e não de um cache.

Exemplo: `volatile int contador;`

transient: Indica que o atributo não deve ser serializado.

Exemplo: `transient int idTemporario;`

native: Indica que um método é implementado em outra linguagem (como C ou C++).

Exemplo: `native void metodoNativo();`

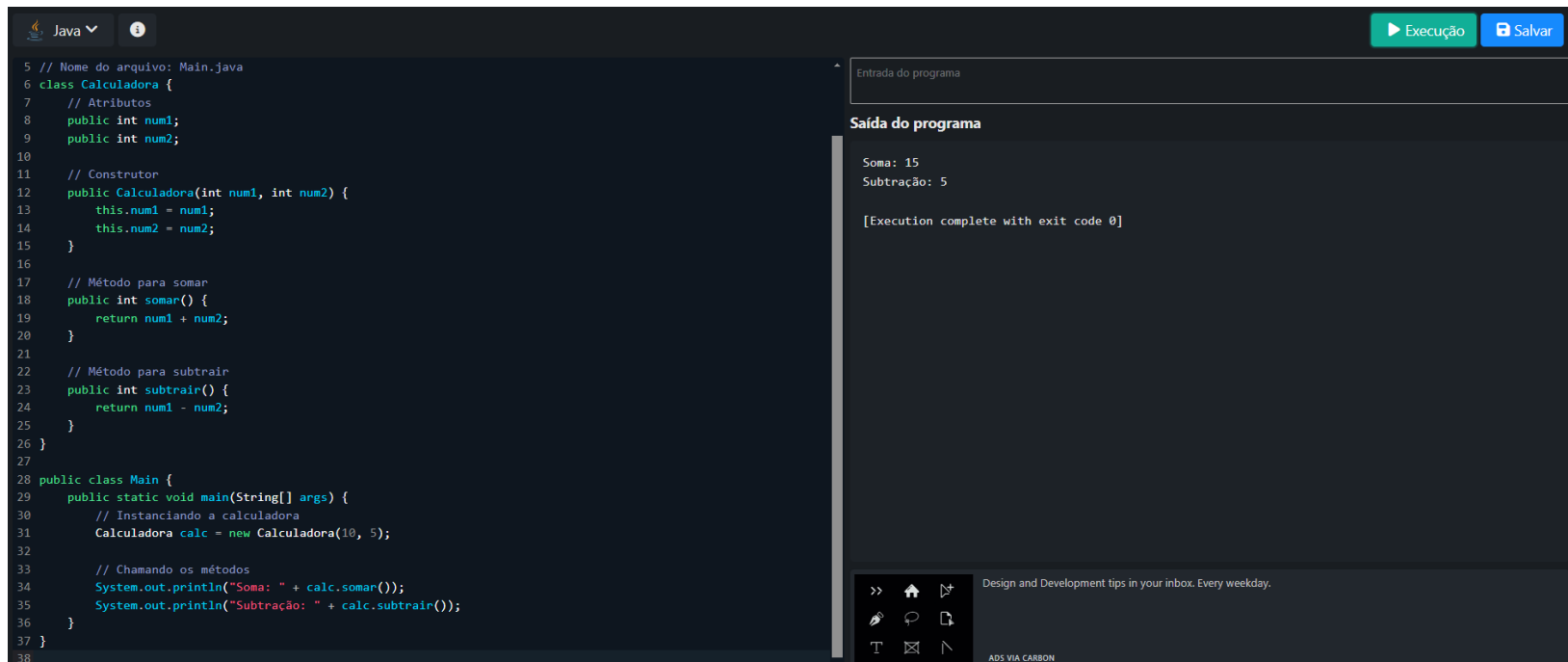
strictfp: Garante que os cálculos em ponto flutuante sigam o padrão IEEE 754, tornando o comportamento consistente em diferentes plataformas.

Exemplo: `strictfp class ExemploFP { /* código */ }`

Esses modificadores ajudam a controlar o design e o comportamento de código em Java.

CONTEXTO

Atributos **públicos** podem ser **acessados** livremente de **fora da classe**, observe o exemplo abaixo, lembrando que a linha 6 apresenta a classe Calculadora como private. Caso seja necessário colocar como public será importante que o arquivo ou projeto tenha o nome Calculadora.java.



The screenshot shows an IDE with a Java file named Main.java. The code defines a `Calculadora` class with two public attributes (`num1` and `num2`), a constructor, and two public methods (`somar` and `subtrair`). A `Main` class contains a `main` method that instantiates a `Calculadora` object and calls its methods. The IDE interface includes a top bar with 'Java' and a dropdown, and buttons for 'Execução' (Execution) and 'Salvar' (Save). On the right, there is a panel for 'Entrada do programa' (Program Input) and 'Saída do programa' (Program Output). The output shows the results of the calculations: 'Soma: 15' and 'Subtração: 5', followed by '[Execution complete with exit code 0]'. At the bottom right, there is a footer with 'Design and Development tips in your inbox. Every weekday.' and 'ADS VIA CARBON'.

```
5 // Nome do arquivo: Main.java
6 class Calculadora {
7     // Atributos
8     public int num1;
9     public int num2;
10
11     // Construtor
12     public Calculadora(int num1, int num2) {
13         this.num1 = num1;
14         this.num2 = num2;
15     }
16
17     // Método para somar
18     public int somar() {
19         return num1 + num2;
20     }
21
22     // Método para subtrair
23     public int subtrair() {
24         return num1 - num2;
25     }
26 }
27
28 public class Main {
29     public static void main(String[] args) {
30         // Instanciando a calculadora
31         Calculadora calc = new Calculadora(10, 5);
32
33         // Chamando os métodos
34         System.out.println("Soma: " + calc.somar());
35         System.out.println("Subtração: " + calc.subtrair());
36     }
37 }
38
```

Entrada do programa

Saída do programa

Soma: 15
Subtração: 5

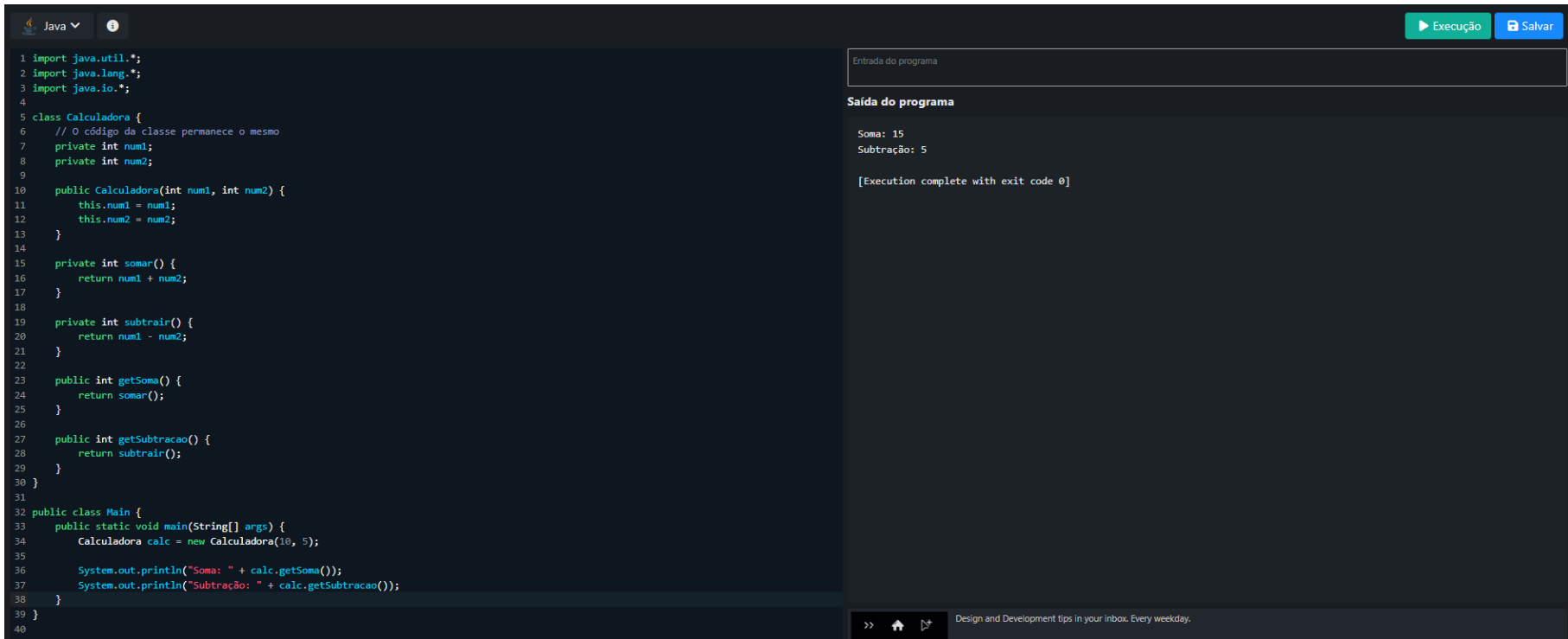
[Execution complete with exit code 0]

Design and Development tips in your inbox. Every weekday.

ADS VIA CARBON

CONTEXTO

Atributos **privados**, por outro lado, **não** podem ser acessados fora da classe. Ou seja, **somente a própria classe** tem domínio sobre um atributo privado.



The screenshot shows an IDE interface with a Java code editor on the left and a console/output window on the right. The code defines a `Calculadora` class with private attributes `num1` and `num2`, and methods for addition, subtraction, and getters. A `Main` class tests the calculator with values 10 and 5.

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 class Calculadora {
6     // O código da classe permanece o mesmo
7     private int num1;
8     private int num2;
9
10    public Calculadora(int num1, int num2) {
11        this.num1 = num1;
12        this.num2 = num2;
13    }
14
15    private int somar() {
16        return num1 + num2;
17    }
18
19    private int subtrair() {
20        return num1 - num2;
21    }
22
23    public int getSoma() {
24        return somar();
25    }
26
27    public int getSubtracao() {
28        return subtrair();
29    }
30 }
31
32 public class Main {
33     public static void main(String[] args) {
34         Calculadora calc = new Calculadora(10, 5);
35
36         System.out.println("Soma: " + calc.getSoma());
37         System.out.println("Subtração: " + calc.getSubtracao());
38     }
39 }
40
```

The right panel shows the execution output:

Entrada do programa

Saída do programa

Soma: 15
Subtração: 5

[Execution complete with exit code 0]

At the bottom right, there is a footer: Design and Development tips in your inbox. Every weekday.

CONTEXTO

No primeiro exemplo, os atributos `num1` e `num2`, o construtor e os métodos `somar()` e `subtrair()` são todos públicos, ou seja, podem ser acessados e modificados de qualquer lugar. No `main`, criamos uma instância da calculadora e acessamos os métodos diretamente.

Agora, ao adotarmos as alterações do segundo exemplo, os atributos `num1` e `num2`, e os métodos `somar()` e `subtrair()` foram modificados para `private`, ou seja, não podem ser acessados diretamente de fora da classe. Criamos os métodos públicos `getSoma()` e `getSubtracao()` para acessar de forma controlada os métodos privados.

Essa abordagem melhora o encapsulamento, evitando que os valores e operações internas sejam diretamente manipulados de fora da classe.

CONTEXTO

Com o uso da segunda abordagem temos uma alteração importante ao acesso a atributos e métodos de uma classe. Contudo utilizamos um comando que nos dá acesso a recursos privados e iremos conhecer melhor agora sobre esses comandos.

Em Programação Orientada a Objetos (POO), acessores são métodos utilizados para acessar ou modificar o valor de atributos (ou propriedades) de uma classe. Esses métodos garantem o encapsulamento, ou seja, protegem os atributos internos da classe e permitem o controle sobre como os dados são lidos ou modificados. Os dois principais tipos de acessores são:

1. **Getters (Métodos de Acesso)**
2. **Setters (Métodos de Modificação)**

CONTEXTO

Os **getters** são métodos que permitem acessar o valor de um atributo privado. Eles seguem a convenção de iniciar com o prefixo **get** seguido do nome da propriedade com a primeira letra em maiúsculo. A função é obter o valor de um atributo privado. Conforme a figura abaixo:

```
public class Pessoa {  
    private String nome;  
  
    // Método Setter  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

CONTEXTO

Os **getters** são métodos que permitem acessar o valor de um atributo privado. Eles seguem a convenção de iniciar com o prefixo **get** seguido do nome da propriedade com a primeira letra em maiúsculo. A função é obter o valor de um atributo privado. Conforme a figura abaixo a esquerda.

Os **setters** são métodos que permitem modificar o valor de um atributo privado. Eles seguem a convenção de iniciar com o prefixo **set** seguido do nome da propriedade com a primeira letra em maiúsculo. A função é alterar o valor de um atributo privado, permitindo que a lógica de verificação ou validação seja aplicada ao modificar o dado. Veja o exemplo abaixo a direita:

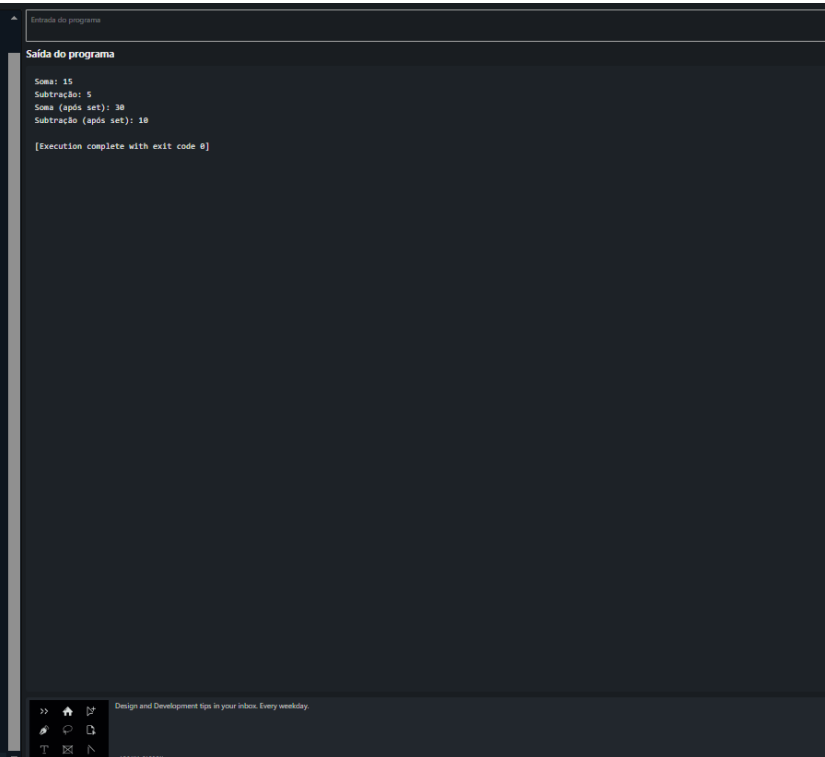
```
public class Pessoa {  
    private String nome;  
  
    // Método Getter  
    public String getNome() {  
        return nome;  
    }  
}
```

```
public class Pessoa {  
    private String nome;  
  
    // Método Setter  
    public void setName(String nome) {  
        this.nome = nome;  
    }  
}
```

CONTEXTO

No código abaixo, havíamos aplicados o uso do **getters**, agora vamos aplicar o **setter** para permitir modificações nos valores dos atributos privados.

```
1 import java.io.*;
2
3 class Calculadora {
4     // Atributos privados
5     private int num1;
6     private int num2;
7
8     // Construtor público
9     public Calculadora(int num1, int num2) {
10         this.num1 = num1;
11         this.num2 = num2;
12     }
13
14     // Método privado para somar
15     private int somar() {
16         return num1 + num2;
17     }
18
19     // Método privado para subtrair
20     private int subtrair() {
21         return num1 - num2;
22     }
23
24     // Getter para a soma
25     public int getSoma() {
26         return somar();
27     }
28
29     // Getter para a subtração
30     public int getSubtracao() {
31         return subtrair();
32     }
33
34     // Setter para num1
35     public void setNum1(int num1) {
36         this.num1 = num1;
37     }
38
39     // Setter para num2
40     public void setNum2(int num2) {
41         this.num2 = num2;
42     }
43 }
44
45 public class Main {
46     public static void main(String[] args) {
47         // Criando a calculadora com valores iniciais
48         Calculadora calc = new Calculadora(10, 5);
49
50         // Exibindo os resultados de soma e subtração
51         System.out.println("Soma: " + calc.getSoma());
52         System.out.println("Subtração: " + calc.getSubtracao());
53
54         // Modificando os valores usando os setters
55         calc.setNum1(20);
56         calc.setNum2(10);
57
58         // Exibindo os resultados novamente após a modificação
59         System.out.println("Soma (após set): " + calc.getSoma());
60         System.out.println("Subtração (após set): " + calc.getSubtracao());
61     }
62 }
63 }
```



CONTEXTO

Setters (setNum1 e setNum2):

Foram adicionados os métodos `setNum1(int num1)` e `setNum2(int num2)`, que permitem modificar os valores dos atributos `num1` e `num2` depois que o objeto `Calculadora` foi instanciado.

Uso dos setters no main:

Após a criação do objeto `Calculadora`, os valores de `num1` e `num2` são modificados usando os setters (`calc.setNum1(20)` e `calc.setNum2(10)`). Em seguida, os resultados de soma e subtração são recalculados com os novos valores.

Agora, os métodos **setters** permitem modificar os valores dos atributos de forma controlada, mantendo o encapsulamento e garantindo que os atributos sejam acessados e alterados de forma apropriada.

CONTEXTO

Além dos itens citados, em Java sempre que algo **inadequado** for **detectado**, se faz necessário a criação de um objeto da classe **Exception** e a lançamos como o comando **throw**.

O uso de **Exception** está ligado a uma **exceção** reportada derivada a uma situação **adversa** onde a **classe não é capaz de funcionar**. Vamos aplicar um modelo de exceção no código Calculadora.

CONTEXTO

```
5 // Custom exception para valores inválidos
6 class ValorInvalidoException extends Exception {
7     public ValorInvalidoException(String message) {
8         super(message);
9     }
10 }
11
12 class Calculadora {
13     // Atributos privados
14     private int num1;
15     private int num2;
16
17     // Construtor público
18     public Calculadora(int num1, int num2) throws ValorInvalidoException {
19         setNum1(num1);
20         setNum2(num2);
21     }
22
23     // Método privado para somar
24     private int somar() {
25         return num1 + num2;
26     }
27
28     // Método privado para subtrair
29     private int subtrair() {
30         return num1 - num2;
31     }
32
33     // Getter para a soma
34     public int getSoma() {
35         return somar();
36     }
37
38     // Getter para a subtração
39     public int getSubtracao() {
40         return subtrair();
41     }
42 }
```

CONTEXTO

```
42
43 // Setter para num1 com validação
44 public void setNum1(int num1) throws ValorInvalidoException {
45     if (num1 < 0) {
46         throw new ValorInvalidoException("O valor de num1 não pode ser negativo.");
47     }
48     this.num1 = num1;
49 }
50
51 // Setter para num2 com validação
52 public void setNum2(int num2) throws ValorInvalidoException {
53     if (num2 < 0) {
54         throw new ValorInvalidoException("O valor de num2 não pode ser negativo.");
55     }
56     this.num2 = num2;
57 }
58 }
59
60 public class Main {
61     public static void main(String[] args) {
62         try {
63             // Criando a calculadora com valores iniciais válidos
64             Calculadora calc = new Calculadora(10, 5);
65             System.out.println("Soma: " + calc.getSoma());
66             System.out.println("Subtração: " + calc.getSubtracao());
67
68             // Modificando os valores usando os setters
69             calc.setNum1(20);
70             calc.setNum2(10);
71
72             System.out.println("Soma (após set): " + calc.getSoma());
73             System.out.println("Subtração (após set): " + calc.getSubtracao());
74
75             // Tentando definir valores inválidos
76             calc.setNum1(-1); // Isso vai lançar uma exceção
77         } catch (ValorInvalidoException e) {
78             System.err.println("Erro: " + e.getMessage());
79         }
80     }
81 }
82
```

CONTEXTO

Saída do programa

```
Soma: 15  
Subtração: 5  
Soma (após set): 30  
Subtração (após set): 10  
Erro: O valor de num1 não pode ser negativo.  
  
[Execution complete with exit code 0]
```

Classe `ValorInvalidoException`: Uma classe de exceção personalizada que estende **`Exception`**. Ela é usada para lançar erros específicos relacionados a valores inválidos.

Modificação nos Setters: O método **`setNum1(int num1)`** e **`setNum2(int num2)`** agora lançam uma exceção **`ValorInvalidoException`** se o valor fornecido for negativo.

Construtor da Calculadora: O construtor também pode lançar uma exceção se os valores iniciais forem inválidos.

Bloco `try-catch` no `main`: Quando você tenta definir um valor inválido, o programa captura a exceção e imprime uma mensagem de erro apropriada.

EXEMPLO

```
5 class Veiculo {
6     private String modelo;
7     private int potencia;
8     private Combustivel combustivel;
9
10    public String getModelo() {
11        return modelo;
12    }
13
14    public void setModelo(String modelo) throws Exception {
15        if (modelo == null || modelo.trim().isEmpty()) {
16            throw new Exception("O modelo é obrigatório.");
17        }
18        this.modelo = modelo;
19    }
20
21    public int getPotencia() {
22        return potencia;
23    }
24
25    public void setPotencia(int potencia) throws Exception {
26        if (potencia < 0) {
27            throw new Exception("A potência deve ser positiva.");
28        }
29        this.potencia = potencia;
30    }
31
32    public Combustivel getCombustivel() {
33        return combustivel;
34    }
35
36    public void setCombustivel(Combustivel combustivel) {
37        this.combustivel = combustivel;
38    }
39 }
40 }
41
```

```
42 enum Combustivel {
43     GASOLINA, ETANOL
44 }
45
46 public class Main {
47     public static void main(String[] args) {
48         Veiculo v = new Veiculo();
49
50         try {
51             v.setModelo("Comodoro");
52             v.setPotencia(134);
53             v.setCombustivel(Combustivel.GASOLINA); // Utilizando o enum para garantir a validade
54
55             System.out.println("Veículo criado com sucesso!");
56             System.out.println("Modelo: " + v.getModelo());
57             System.out.println("Potência: " + v.getPotencia());
58             System.out.println("Combustível: " + v.getCombustivel());
59         } catch (Exception e) {
60             System.out.println("Erro ao criar veículo: " + e.getMessage());
61             // Aqui você pode implementar um tratamento de exceção mais robusto,
62             // como solicitar ao usuário que insira um novo valor para o combustível
63         }
64     }
65 }
66
```

Saída do programa

```
Veículo criado com sucesso!
Modelo: Comodoro
Potência: 134
Combustível: GASOLINA
```

[Execution complete with exit code 0]

EXEMPLO

Observe que no exemplo dado trabalhamos com dois comandos que não vimos antes, o **Exception** de forma diferente aplicada na Calculadora e o **Enum**.

No primeiro caso temos uma outra opção do uso do **Exception**, as exceções são classificadas em diferentes tipos, cada um com suas características e formas de tratamento. Essa classificação é importante para entender como lidar com diferentes tipos de erros que podem ocorrer durante a execução de um programa. Vamos conhecer algumas dessas exceções:

Checked Exceptions (Exceções Verificadas): São exceções que o compilador obriga você a tratar. Isso significa que você precisa usar um bloco ***try-catch*** ou declarar a exceção no método com a palavra-chave ***throws***. Representam condições que podem ser previstas e tratadas pelo programador, como problemas de E/S (entrada e saída), erros de rede, etc. Os exemplos são:

- **IOException:** Ocorre quando há um problema de entrada ou saída de dados.
- **SQLException:** Ocorre quando há um erro ao acessar um banco de dados.
- **ClassNotFoundException:** Ocorre quando uma classe não é encontrada.

EXEMPLO

Unchecked Exceptions (Exceções Não Verificadas): São exceções que não precisam ser tratadas explicitamente pelo programador. O compilador não obriga você a usar um bloco ***try-catch*** para elas. Geralmente representam erros de programação, como erros de lógica, acesso a índices inválidos de arrays, etc. Os exemplos são:

- **NullPointerException:** Ocorre quando você tenta acessar um objeto nulo.
- **ArithmeticException:** Ocorre quando você tenta dividir um número por zero.
- **ArrayIndexOutOfBoundsException:** Ocorre quando você tenta acessar um elemento de um array fora dos limites.

Errors: Representam erros graves que geralmente indicam problemas no sistema e não podem ser recuperados. Normalmente são causados por problemas de memória, erros do sistema operacional ou bugs no próprio ambiente de execução da JVM. Os exemplos são:

- **OutOfMemoryError:** Ocorre quando a memória disponível é insuficiente.
- **StackOverflowError:** Ocorre quando a pilha de chamadas fica muito grande.

EXEMPLO

Tipo de Exceção	Verificada?	Causas comuns	Exemplos
Checked	Sim	Problemas de E/S, banco de dados, etc.	IOException, SQLException
Unchecked	Não	Erros de programação, lógica	NullPointerException, ArithmeticException
Error	Não	Problemas graves no sistema	OutOfMemoryError, StackOverflowError

Por que a distinção entre os tipos é importante?

Tratamento: As checked exceptions exigem tratamento explícito, enquanto as unchecked podem ser tratadas ou não.

Prevenção: Entender as causas comuns de cada tipo de exceção ajuda a escrever código mais robusto e prevenir erros.

Design: A escolha de lançar uma checked ou unchecked exception pode influenciar o design da sua aplicação.

EXEMPLO

No caso segundo comando, em Java, um enum (abreviação de "enumeration") é um tipo de dado especial que define um conjunto de constantes nomeadas. Essas constantes representam valores únicos e imutáveis dentro de um determinado contexto. No caso do enum Combustivel, ele define os tipos de combustível possíveis para um veículo.

```
41  
42 enum Combustivel {  
43     GASOLINA, ETANOL  
44 }  
45
```

Essa linha de código declara um novo tipo de dado chamado Combustivel e define duas constantes: GASOLINA e ETANOL. Essas constantes representam os únicos valores válidos para o atributo combustivel de um objeto Veiculo.

EXEMPLO

Ao instanciar um novo objeto podemos atribuir `Combustivel.GASOLINA` ao atributo `combustivel` do objeto `meuCarro`, estamos definindo que o veículo utiliza gasolina como combustível.

```
Veiculo meuCarro = new Veiculo();  
meuCarro.setCombustivel(Combustivel.GASOLINA);
```

Vantagens de usar Enums:

Ao usar um enum, você garante que o valor atribuído a um atributo seja sempre um dos valores definidos no enum. Isso evita erros de digitação e torna o código mais seguro. Os nomes das constantes são claros e autoexplicativos, tornando o código mais fácil de entender e manter.

Um enum pode ser reutilizado em diferentes partes do código, promovendo a consistência e evitando a duplicação de valores. Se você precisar adicionar um novo tipo de combustível, basta adicionar uma nova constante ao enum.

EXEMPLO

Alguns pontos adicionais sobre o uso do **ENUM** é que podemos adicionar valores associados a cada constante do enum, como um código numérico ou uma descrição. É possível definir métodos dentro de um enum para realizar operações específicas. Os **enums** podem ser usados em instruções **switch-case** para realizar diferentes ações com base no valor do **enum**.

```
enum Combustivel {  
    GASOLINA(1), ETANOL(2);  
  
    private final int codigo;  
  
    Combustivel(int codigo) {  
        this.codigo = codigo;  
    }  
  
    public int getCodigo() {  
        return codigo;  
    }  
}
```

Neste exemplo, cada constante do enum possui um código associado, que pode ser utilizado para outras finalidades.

EXERCÍCIO

Crie um classe Filme que contém os atributos título, duração em minutos e gênero. Essa classe deve encapsular e validar todos os atributos.

- O título não pode estar vazio.
- A duração deve ser maior que zero.
- O gênero de ser Romance, Terror ou Comédia;

Crie uma classe conta bancária da qual o atributo saldo é encapsulado e seu valor será acessado por meio dos métodos, depósito, saque e consulta.

- Para cada depósito deve ser cobrado uma taxa de 1%.
- Para cada saque deve ser cobrado uma taxa de 0,5%.
- A cada 5 consultas, será cobrado uma taxa de 0,10 centavos.

MUITO OBRIGADO!!!!



daniel.ohata@facens.br