# Modificador *final*

Fazendo uma revisão sobre *override*, que é quando mudamos o comportamento do método na sua própria classe, definido de forma diferente que foi definido na sua superclasse. Agora iremos falar sobre um modificador que afeta diretamente à herança, que é o modificador final.

Este modificador tem várias utilidades dependendo da onde usa. Podemos usá-lo em métodos, classes e/ou atributos.

## A) Atributos:

Primeiro criemos uma classe chamada carro:

|  |
| --- |
| **public** **class** Carro  { **private** **double** velocidadeLimite;  **private** String nome;  **private** String marca;    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |

Digamos que nosso carro possui um limite de velocidade, e este parâmetro não pode mudar, independente do que aconteça. São conhecidas como constantes, e no java as definimos com o fator *static*. Como sabemos que nossa *double* velocidade limite, por legislação, não deve variar independente da instância da classe carro, então simplesmente colocamos este fator antes do tipo da variável declarada e, além disso, devemos já inicializa-la com um valor.

|  |
| --- |
| **public** **class** Carro  { **private** **static** **double** velocidadeLimite;  .  .  .  .  } |

Mas ainda assim, não temos uma forma de impedir que não mude. Então, como queremos que nosso atributo seja constante, independente da instância da classe, devemos complementar nosso atributo estático com o fator ***final* e ainda inicializá-lo com um valor**.

|  |
| --- |
| **public** **class** Carro  { **private** **static** **final** **double** ***velocidadeLimite*** = 130;  **private** String nome;  **private** String marca;    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;} prá  **public** **void** setMarca(String marca) {**this**.marca = marca;}    **public** **static** **double** getVelocidadeLimite(){**return** ***velocidadeLimite***;}  **public** **static** **void** setVelocidadeLimite(**double** velocidadeLimite)  {Carro.***velocidadeLimite*** = velocidadeLimite;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |

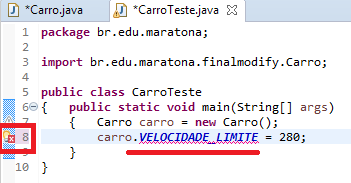
Uma boa prática é sempre nomear essas variáveis com letras maiúsculas, e se tiver que separar palavras, usar “­\_”.

|  |
| --- |
| **public** **class** Carro  { **public** **static** **final** **double** ***VELOCIDADE\_LIMITE*** = 130;  **private** String nome;  **private** String marca;    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |

**OBS: Caso não declararmos nosso atributo constante com fator *static*, podemos inicializa-la através de um construtor. Ou, quando não declaramos como *final*, podemos fazer por bloco de inicialização.**

|  |
| --- |
| **public** **class** Carro  { **public** **final** **double** VELOCIDADE\_LIMITE;  **private** String nome;  **private** String marca;    **public** Carro() {**this**.VELOCIDADE\_LIMITE = 130;}    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |
| **public** **class** Carro  { //private static final double VELOCIDADE\_LIMITE = 130;  **public** **static** **double** *VELOCIDADE\_LIMITE*;  **private** String nome;  **private** String marca;    **static** {*VELOCIDADE\_LIMITE* = 130;}    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |

Vamos utilizar o primeiro bloco de código e testar em uma nova classe de teste chamada CarroTeste.java com método *main*. Se tentarmos definir uma velocidade limite para nossa instância de carro, perceba que teremos já um erro de compilação, pois o java já irá entender que estamos tentando alterar algo que não deve.

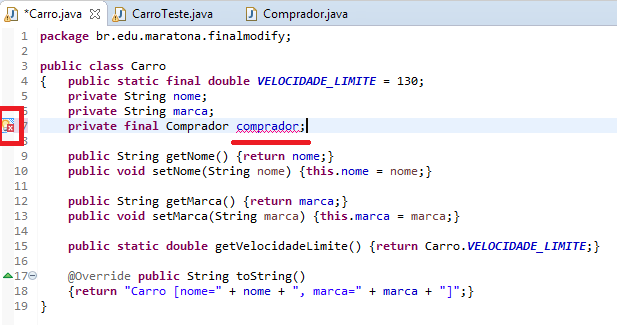


**Figura – Erro de compilação. Alteração de uma constante.**

Mas isso quando temos uma variável do tipo primitivo, agora o que será que acontece quando temos uma variável do tipo Referência, ou seja, uma classe. Criemos uma nova classe do tipo Comprador.java.

|  |
| --- |
| **public** **class** Comprador  { **private** String nome;    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  } |

Se declararmo que essa classe é um atributo da classe Carro, como *final*, veja que teremos um erro novamente de compilação.

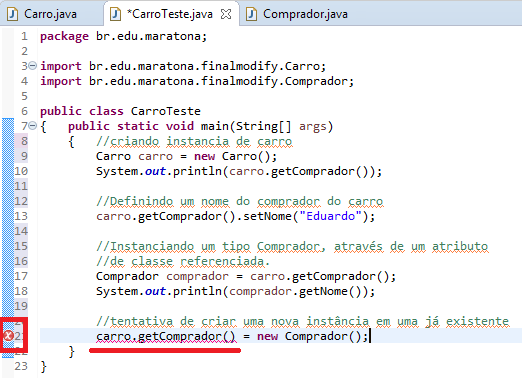


**Figura – Erro de compilação. Declaração de classe, como atributo, tipo final.**

Se quisermos corrigir isso, já vimos em momentos anteriores que basta inicializa-lo, por exemplo:

|  |
| --- |
| **public** **class** Carro  { **public** **static** **final** **double** ***VELOCIDADE\_LIMITE*** = 130;  **private** String nome;  **private** String marca;  **private** **final** Comprador comprador = **new** Comprador();    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |

**Obs: outro erro comum é criar uma nova referência em algo que já está referenciado.**

****

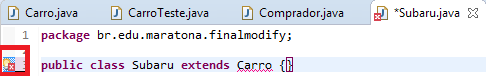
**Figura – Erro de compilação. Atribuindo uma nova instância em um atributo já referenciado e como *final*.**

## B) Classes

Agora iremos definir que nossa classe Carro é do tipo *final*. Isso implica que essa classe não pode ser extendida, ou seja, não poderá ter classes filhas pois é a última das classes.

|  |
| --- |
| **public** **final** **class** Carro  {.  .//bloco de códigos  .  } |

Se quisermos tentar criar uma classe que herda dessa classe final, teremos mais um erro de compilação, pois uma classe definida como final, não podemos ter classes que herde da mesma.

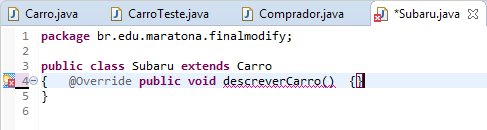


**Figura – Erro de compilação. Classe herdando de uma classe do tipo *final*.**

## C) Métodos

Para fecharmos esse assunto, podemos usar esse fator *final* em métodos também. Isso tem a ver com POO também, pois não podemos sobrescrever um método desse tipo. Se voltarmos a classe Carro para não ser como final, ou seja, que possa ter classes herdeiras, e tiver um método do tipo *final*, significa que suas classes irão herdar mas não poderão modifica-las.

|  |
| --- |
| **public** **class** Carro  { **public** **static** **final** **double** ***VELOCIDADE\_LIMITE*** = 130;  **private** String nome;  **private** String marca;  **private** **final** Comprador comprador = **new** Comprador();    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** String getMarca() {**return** marca;}  **public** **void** setMarca(String marca) {**this**.marca = marca;}    **public** Comprador getComprador() {**return** comprador;}    **public** **final** **void** descreverCarro() {**this**.toString();}    @Override **public** String toString()  {**return** "Carro [nome=" + nome + ", marca=" + marca + "]";}  } |



**Figura – Erro de compilação. Tentativa de sobreescrita de método do tipo *final*.**

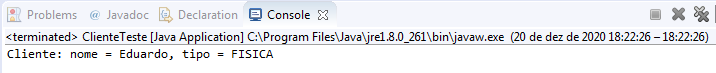
# ENUM

Enumeração foi um recurso criado a partir da versão 5 do JAVA e iremos explicar o por que. Imagine que temos a seguinte classe de cliente.

|  |
| --- |
| **public** **class** Cliente  { **private** String nome;  **public** **static** **final** **int** ***PESSOA\_FISICA*** = 1;  **public** **static** **final** **int** ***PESSOA\_JURIDICA*** = 2;  **private** **int** tipo;    **public** Cliente(String nome, **int** tipo)  { **this**.nome = nome;  **this**.tipo = tipo;  }    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** **int** getTipo() {**return** tipo;}  **public** **void** setTipo(**int** tipo) {**this**.tipo = tipo;}  @Override **public** String toString()  { String tipoPessoa = "";  **if**(**this**.getTipo() == 1) {tipoPessoa = "FISICA";}  **else** {tipoPessoa = "JURIDICA";}  **return** "Cliente: nome = " + **this**.nome +  ", tipo = " + tipoPessoa;  }  } |

Agora iremos, além de criar uma classe de tipo ClienteTeste.java, instanciar um tipo de cliente pessoa física.

|  |
| --- |
| **public** **class** ClienteTeste  { **public** **static** **void** main(String[] args)  { //definindo um cliente tipo pessoa fisica  Cliente cliente = **new** Cliente("Eduardo",  Cliente.***PESSOA\_FISICA***);    System.***out***.println(cliente.toString());  }  } |



**Figura – Printando um Cliente.**

Pode acontecer de uma pessoa não passar mais utilizar essas variáveis estáticas constantes. A variável inteira “tipo” da classe Cliente é do tipo int, então, o que impediria um programador de digitar e passar um valor 10 por exemplo. No nosso caso, o método *toString()* da classe Cliente irá imprimir o cliente como Pessoa Jurídica, mas se tivermos que salvar essa informação no banco de dados? Esse valor inteiro poderia trazer muitos problemas. Para evitar esse tipo de situação que foi criado o *enum*, ou seja, enumerações. Nas enumerações, tudo é considerado constante.

Temos duas formas de criar uma enumeração:

## A) Criando classes:

Criar classe de forma separada, que muitos consideram a melhor opção.

|  |
| --- |
| **public** **enum** EnumCliente  {***PESSOA\_FISICA***, ***PESSOA\_JURIDICA***;} |
| **public** **class** Cliente  { **private** String nome;  //public static final int PESSOA\_FISICA = 1;  //public static final int PESSOA\_JURIDICA = 2;  **private** EnumCliente tipo;    **public** Cliente(String nome, EnumCliente tipo)  { **this**.nome = nome;  **this**.tipo = tipo;  }    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** EnumCliente getTipo() {**return** tipo;}  **public** **void** setTipo(EnumCliente tipo) {**this**.tipo = tipo;}  @Override **public** String toString()  { String tipoPessoa = "";  **if**(**this**.getTipo() == EnumCliente.***PESSOA\_FISICA***)  {tipoPessoa = "FISICA";}  **else** {tipoPessoa = "JURIDICA";}  **return** "Cliente: nome = " + **this**.nome +  ", tipo = " + tipoPessoa;  }  } |
| **import** br.edu.maratona.enumeracao.EnumCliente;  **public** **class** ClienteTeste  { **public** **static** **void** main(String[] args)  { //definindo um cliente tipo pessoa fisica  Cliente cliente = **new** Cliente("Eduardo",  EnumCliente.***PESSOA\_FISICA***);    System.***out***.println(cliente.toString());  }  } |

Dessa forma, ao invés de um programador ou qualquer profissional da área que tentasse implementar algo, o impediria de escrever qualquer coisa, pois ao invés de permiti-lo de escrever um valor inteiro, obrigará escrever um tipo constante.

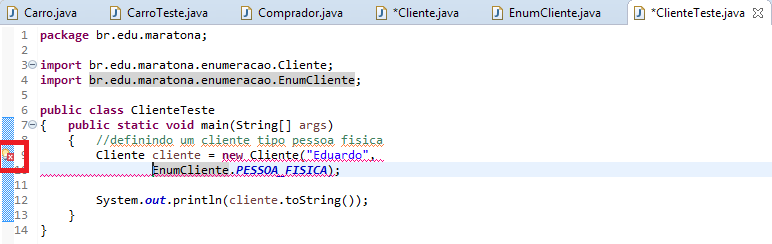
## B) Criar como Atributo de classe:

Como um *enum* é um tipo constante, basta então criarmos uma variável local de classe do tipo *enum*, definindo valores em sequência. Então podemos simplesmente fazer uma cópia do que escrevemos na classe *EnumCliente* e colar na classe Cliente.

|  |
| --- |
| **public** **class** Cliente  { **private** String nome;  //public static final int PESSOA\_FISICA = 1;  //public static final int PESSOA\_JURIDICA = 2;  **public** **enum** EnumCliente {***PESSOA\_FISICA***, ***PESSOA\_JURIDICA***;}  **private** EnumCliente tipo;      **public** Cliente(String nome, EnumCliente tipo)  { **this**.nome = nome;  **this**.tipo = tipo;  }    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** EnumCliente getTipo() {**return** tipo;}  **public** **void** setTipo(EnumCliente tipo) {**this**.tipo = tipo;}  @Override **public** String toString()  { String tipoPessoa = "";  **if**(**this**.getTipo() == EnumCliente.***PESSOA\_FISICA***)  {tipoPessoa = "FISICA";}  **else** {tipoPessoa = "JURIDICA";}  **return** "Cliente: nome = " + **this**.nome +  ", tipo = " + tipoPessoa;  }  } |

O problema de fazer dessa maneira, é que toda vez que tivermos que chamar na classe de execução, além de chamar o tipo EnumCliente, primeiro temos que chamar a classe Cliente para depois chamar o tipo *enum*. Caso contrário, teremos um erro de compilação.

|  |
| --- |
| **public** **class** ClienteTeste  { **public** **static** **void** main(String[] args)  { //definindo um cliente tipo pessoa fisica  Cliente cliente = **new** Cliente("Eduardo",  Cliente.EnumCliente.***PESSOA\_FISICA***);    System.***out***.println(cliente.toString());  }  } |



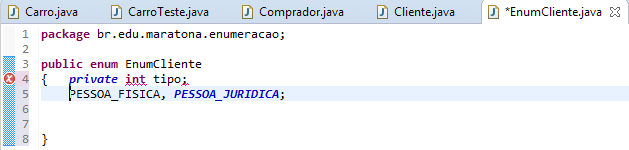
**Figura – Erro de compilação: Atributo tipo *enum* numa classe.**

**OBS: De maneira alguma é permitido criar um *enum* em um método no java.**

Enumeração é um tipo especial de classe, pois podemos até mesmo adicionar métodos dentro de um *enum*. Então voltando para o nosso EnumCliente, lembra que antes de tê-la criada, tínhamos somente um inteiro “tipo” que só admitiam valores 1 e 2? Imagine que estamos fazendo modificação em um sistema, onde banco de dados já estava salvando esses valores e precisamos continuar salvando. Para resolver esse problema, podemos fazer com que a nossa enumeração tenha um construtor e tenha atributos de classe e métodos também. Nunca poderemos chamar ou inicializar um construtor de uma enumeração, pois ele próprio será chamado.

Então podemos criar atributos, porém existe uma regra: não podemos criar antes da própria enumeração.

|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***, ***PESSOA\_JURIDICA***;    **private** **int** tipo;  } |

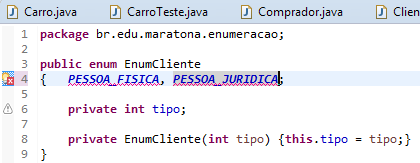


**Figura – Erro de compilação: Atributos em enum.**

Para criar um construtor, jamais podemos defini-lo como *public*, pois nunca devemos instanciar um tipo *enum*, pois ele mesmo será chamado. Além disso, devemos definir um numero para os parâmetros:

|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***(1), ***PESSOA\_JURIDICA***(2);    **private** **int** tipo;  **private** EnumCliente(**int** tipo) {**this**.tipo = tipo;}  } |

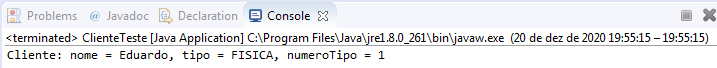
Caso contrário teremos um erro de compilação, como vemos na figura abaixo.



**Figura – Erro de compilação: Construtor em *enum*.**

Agora iremos inserir um método, por exemplo, tipo GETTER somente, pois se um tipo SETTER fosse possível, ou seja métodos modificadores de acesso, deixaria de ser um *enum*.

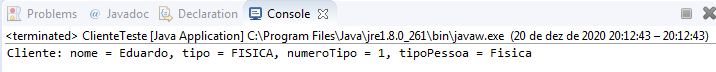
|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***(1), ***PESSOA\_JURIDICA***(2);    **private** **int** tipo;  **private** EnumCliente(**int** tipo) {**this**.tipo = tipo;}  **public** **int** getTipo() {**return** tipo;}  } |
| **public** **class** Cliente  { **private** String nome;  //public static final int PESSOA\_FISICA = 1;  //public static final int PESSOA\_JURIDICA = 2;  //public enum EnumCliente {PESSOA\_FISICA, PESSOA\_JURIDICA;}  **private** EnumCliente tipo;    **public** Cliente(String nome, EnumCliente tipo)  { **this**.nome = nome;  **this**.tipo = tipo;  }    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** EnumCliente getTipo() {**return** tipo;}  **public** **void** setTipo(EnumCliente tipo) {**this**.tipo = tipo;}  @Override **public** String toString()  { String tipoPessoa = "";  **if**(**this**.getTipo() == EnumCliente.***PESSOA\_FISICA***)  {tipoPessoa = "FISICA";}  **else** {tipoPessoa = "JURIDICA";}  **return** "Cliente: nome = " + **this**.nome +  ", tipo = " + tipoPessoa + ", numeroTipo = " +  **this**.tipo.getTipo();  }  } |



**Figura – Resultado do método tipo GETTER em um enum.**

Podemos também, no caso dos construtores, inserir parâmetros de outro tipo, por exemplo string, para indicar que valor real possui o atributo “tipo” definido.

|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***(1, "Fisica"), ***PESSOA\_JURIDICA***(2, "Juridica");    **private** **int** tipo;  **private** String tipoPessoa;    **private** EnumCliente(**int** tipo, String tipoPessoa)  { **this**.tipo = tipo;  **this**.tipoPessoa = tipoPessoa;  }  **public** **int** getTipo() {**return** tipo;}  **public** String getTipoPessoa() {**return** tipoPessoa;}  } |
| **public** **class** Cliente  { **private** String nome;  //public static final int PESSOA\_FISICA = 1;  //public static final int PESSOA\_JURIDICA = 2;  //public enum EnumCliente {PESSOA\_FISICA, PESSOA\_JURIDICA;}  **private** EnumCliente tipo;    **public** Cliente(String nome, EnumCliente tipo)  { **this**.nome = nome;  **this**.tipo = tipo;  }    **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}    **public** EnumCliente getTipo() {**return** tipo;}  **public** **void** setTipo(EnumCliente tipo) {**this**.tipo = tipo;}  @Override **public** String toString()  { String tipoPessoa = "";  **if**(**this**.getTipo() == EnumCliente.***PESSOA\_FISICA***)  {tipoPessoa = "FISICA";}  **else** {tipoPessoa = "JURIDICA";}  **return** "Cliente: nome = " + **this**.nome +  ", tipo = " + tipoPessoa + ", numeroTipo = " +  **this**.tipo.getTipo() + ", tipoPessoa = " +  **this**.tipo.getTipoPessoa();  }  } |



**Figura – Resultado de um *enum* com outro tipo de atributo.**

Resumindo, as enumerações são muito úteis quando queremos definir constantes, em que os valores que não serão e nem deverão serem mudados.

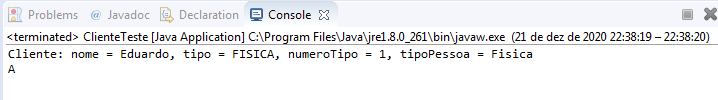
Existe um conceito na enumeração chamada *constant especific class body*, ou seja, corpo de classe constante específica. Imagine um exemplo de classe chamado Cliente e que temos nessa classe um método que retornará uma constante.

*public String getId()*

E o ID para todos eles, tanto de PF e PJ será o mesmo.

*public String getId() {return “A”;}*

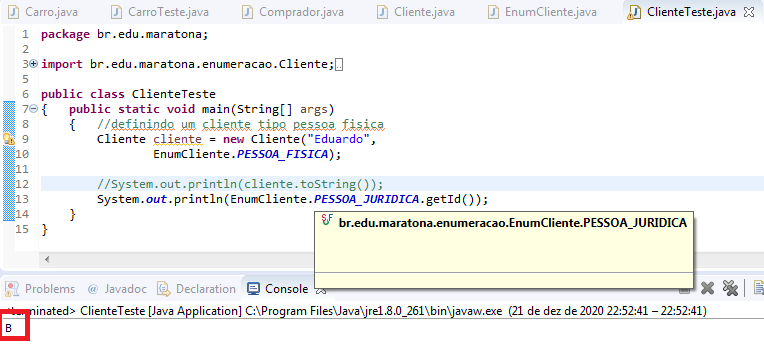
|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***(1, "Fisica"), ***PESSOA\_JURIDICA***(2, "Juridica");    **private** **int** tipo;  **private** String tipoPessoa;    **private** EnumCliente(**int** tipo, String tipoPessoa)  { **this**.tipo = tipo;  **this**.tipoPessoa = tipoPessoa;  }    **public** String getId() {**return** "A";}  **public** **int** getTipo() {**return** tipo;}  **public** String getTipoPessoa() {**return** tipoPessoa;}  } |
| **public** **class** ClienteTeste  { **public** **static** **void** main(String[] args)  { //definindo um cliente tipo pessoa fisica  Cliente cliente = **new** Cliente("Eduardo", EnumCliente.***PESSOA\_FISICA***);    System.***out***.println(cliente.toString());  System.***out***.println(EnumCliente.***PESSOA\_JURIDICA***.getId());  }  } |



**Figura – Método de retorno para *enum.***

Porém, podemos precisar em algum momento sobrescrever esse método *getId()*, supondo que nosso *enum* possui mais de duas constantes e em apenas um deles, tenha valor diferente de ID. O que podemos fazer é criar exatamente esse corpo específico de classe constante. Para fazer isso, imagine que queremos para a PJ.

|  |
| --- |
| **public** **enum** EnumCliente  { ***PESSOA\_FISICA***(1, "Fisica"),  //para essa constante iremos sobrescrever esse método  ***PESSOA\_JURIDICA***(2, "Juridica")  { **public** String getId()  {**return** "B";}  };/\*esse ponto e vírgula é obrigatório ou teremos erro  de compilação\*/    **private** **int** tipo;  **private** String tipoPessoa;    **private** EnumCliente(**int** tipo, String tipoPessoa)  { **this**.tipo = tipo;  **this**.tipoPessoa = tipoPessoa;  }    **public** String getId() {**return** "A";}  **public** **int** getTipo() {**return** tipo;}  **public** String getTipoPessoa() {**return** tipoPessoa;}  } |



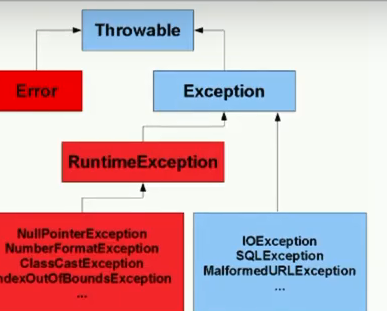
**Figura – Método de retorno para *enum* de constante específica*.***

Se mandarmos imprimir para a PF, apenas nos retornará o valor “A”, que é o *default* neste caso.

# Exceptions

Uma exception acontece quando o código não está fluindo naturalmente, logo acontece que pode estar previsto no seu código ou não. Imagine a seguinte situação, em que estamos tentando abrir um arquivo e durante seu programa sempre abrimos o mesmo arquivo, mas por algum motivo alguém mexeu nesse arquivo e o colocou como somente de leitura, ou seja, não podemos mais fazer nenhuma alteração nesse mesmo arquivo. O que se espera caso tentemos escrever mais coisa nele? Isso é um caso no java que representa exceções, ou seja, fluxos que não seguem o caminho que deveriam ser seguidos.

Outro exemplo disso é quando tentamos comunicar com banco de dados, porém em um determinado momento, por algum motivo externo ou não, não é possível conectar, o que esperemos que aconteça com seu código, se para funcionar precisa da conexão com banco de dados? Essas exceções então foram justamente criadas para tratarmos todos esses problemas que podem acontecer nesses imprevistos no seu código.



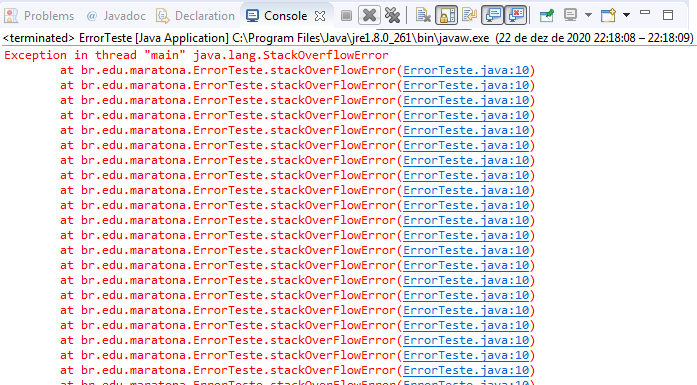
**Figura – Árvore de exceções.**

Todas essas exceções são herdeiras da classe *Throwable*. Quando ocorre alguma exceção no Java, dizemos que determinado método lançou uma exceção. Na figura acima podemos ver que temos 2 tipos, que são filhos diretos de *throwable*:

## A) Error

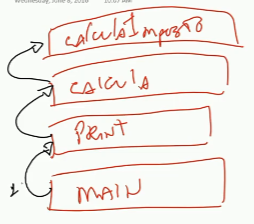
Este possui conceitos um pouco diferente de exceção, pois exceções podem ser tratadas, como foi explicado anteriormente. Porém, quando trabalhamos com erro, seu programa muito provavelmente irá parar de funcionar. Por exemplo, imagine que temos pouca memória em nosso servidor e seu sistema acaba fazendo mal uso dos objetos e simplesmente estoura a quantidade de memória disponível. Então, será lançado um erro chamado *out of memory* *error*, ou seja, a quantidade de memória disponível para a JVM não é suficiente para execução do programa. Exemplo:

|  |
| --- |
| **public** **class** ErrorTeste  { **public** **static** **void** main(String[] args)  {*stackOverFlowError*();}    /\*Isso é uma recursividade infinita, até  que seja lançado um erro chamado stack\*/  **public** **static** **void** stackOverFlowError()  {*stackOverFlowError*();}  } |



**Figura – Error recursivo.**

Para entender uma stack, devemos entender primeiro como funciona uma chamada de método. Supondo o nosso método *main*, imagine que este está chamando um outro método *print()*, que por sua vez chamará o método *calcula()*, que por sua vez irá chamar um outro método por exemplo *calculaImposto()*.



**Figura – Chamadas de múltiplos métodos.**

Se repararmos bem estamos montando uma pilha, ou mais conhecido como *stack*. Quando temos um problema de *stackOverflow*, temos um método *main* que chama outros métodos e que uma hora esta pilha estoura o espaço de memória. Nesses casos não há muito o que possamos fazer, simplesmente o programa irá parar de executar a JVM e parar a execução do programa.

## B) Exception

São fluxos anormais, que não seguem corretamente o que deveriam acontecer no seu código.

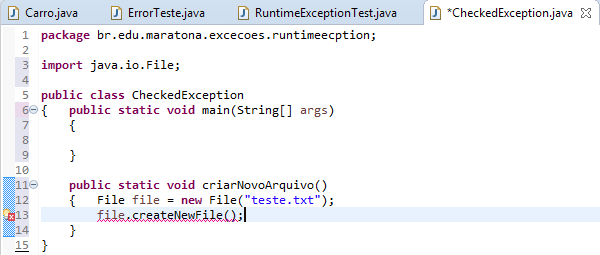
A única semelhança entre erros e exceções é que ambos são lançados. Isso significa que quando temos um método, por exemplo, o último da nossa pilha, e caso este método lançar uma exceção e não for tratada, ou até mesmo um erro, o método irá jogar uma exceção no método que o chamou e assim por diante até o primeiro método.

É muito comum as pessoas confundirem error com exceção, mas tecnicamente não é, pois não é filha de uma exception, porém é uma exceção que não deveria ocorrer normalmente. Iremos focar agora mais no assunto de exceção. Nós temos duas palavras que são extremamente importantes quando estamos trabalhando com exceções:

## C) Checked

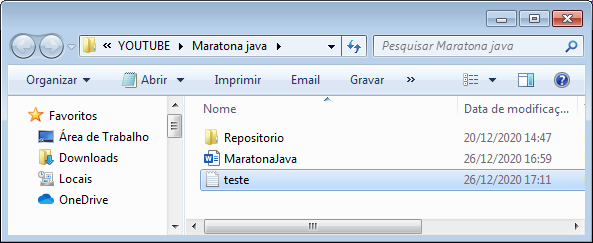
Neste caso, sempre seremos obrigado a criar algum tipo de tratamento para aquela exceção. Quando falamos que uma exceção é do tipo checked, o compilador já está esperando que criemos um tratamento antes mesmo de um problema seja executado. Se não for criado um tratamento, nem conseguiremos compilar seu programa. Exemplo: erro de conexão com banco de dados. Isso está além de um programador.

Se queremos ver uma exceção do tipo checked sem criarmos um código, devemos usar algum código da JVM que já tenha feito isso para gente. Entre as classes que temos na JVM do java que lança uma exceção, provavelmente quase todas que trabalham com arquivos lançam algum tipo de exceção. Por exemplo a classe *File* do java e pegar um método como o *createNewFile()*, que tem o *throws* do tipo *IOException*. Isso significa que todas as vezes que tentarmos usar este método, existe a possibilidade de lançar uma exceção de entrada/saída, o que não significa que irá acontecer, mas devemos trata-lo caso ocorra.



**Figura – Erro de compilação por falta de tratamento de exceção.**

|  |
| --- |
| **public** **class** CheckedException  { **public** **static** **void** main(String[] args)  {*criarNovoArquivo*();}  **public** **static** **void** criarNovoArquivo()  { File file = **new** File("F:/Users/eduardowmu/Desktop/meusdoc/estudo/FATEC/"  + "7 - OUTROS SEMESTRES/YOUTUBE/Maratona java/teste.txt");    //isso recomenda tentar executar algo que pode gerar uma exceção.  **try** {System.***out***.println("Arquivo criado: " + file.createNewFile());}    //caso ocorra a exceção, o java criará o objeto "e" poderá usa-lo para  //retornar algo como o motivo da ocorrencia da exceção.  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |



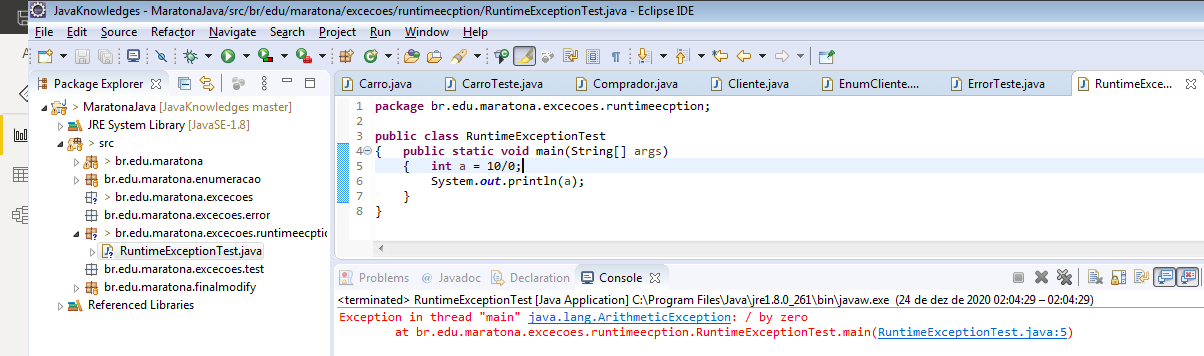
**Figura – Arquivo novo gerado.**

O *try-catch* pode ser usado tanto para exceções do tipo checked como exceções do tipo unchecked. É recomendável sempre usar a exceção mais específica possível, para quando ocorrer, descobrir mais rapidamente a exceção ocorrida.

## D) Unchecked

São exceções que não precisa criar nenhum tipo de tratamento. Geralmente a maioria das exceções desse tipo são erros de lógica, também conhecido como erros de programador. Exemplo, um looping que vai além de um array de tamanho definido. Exceções do tipo RuntimeException são consideradas unchecked, pois não é necessário nenhum tipo de tratamento para seu código compilar. E todas as que são incluindo exception, são do tipo Checked.

|  |
| --- |
| **package** br.edu.maratona.excecoes.runtimeecption;  **public** **class** RuntimeExceptionTest  { **public** **static** **void** main(String[] args)  { **int** a = 10/0;  System.***out***.println(a);  }  } |

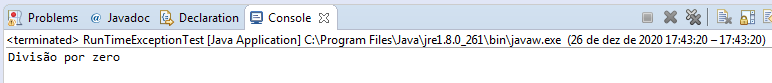


**Figura – RuntimeException.**

Outro tipo de erro muito comum do tipo Unchecked é *NullPointerException*. Temos também, como já dito, um looping ser executado mais vezes que um array suporta. Então esses são tipo unchecked, ou seja, não existira nenhum tratamento que possa tratar uma exceção gerada, pois o programa irá ser executado de qualquer jeito e causará um impacto no programa.

Agora será mostrado como podemos usar os argumentos try-catch em um erro de execução, do tipo, divisão por zero (unchecked exception):

|  |
| --- |
| **public** **class** RunTimeExceptionTest  { **public** **static** **void** main(String[] args)  { **try**{*divisao*(10, 0);}  **catch**(IllegalArgumentException e)  {System.***out***.println(e.getMessage());}  }  **private** **static** **void** divisao(**int** num1, **int** num2)  { **if**(num2 == 0)  { **throw** **new** IllegalArgumentException(  "Divisão por zero");  }  System.***out***.println(num1/num2);  }  } |



**Figura – Tratamento da exception IllegalArgumentException.**

Existe uma outra maneira para criar o bloco *try-catch*, que seria apenas lançar a exceção do primeiro método de origem da exceção, usando o *throws* no nome do método e criar o bloco no último método da pilha que será executado. Caso não seja feito no último da pilha, o código simplesmente irá parar de ser executado e não retornará nada que possa identificar o que aconteceu para o código ter parado de ser executado.

Iremos agora falar sobre uma estrutura que podemos usar no *try-catch* que é o *try with resources*. Até a versão 6 do java, teríamos que montar algo desse tipo de try catch múltiplos:

|  |
| --- |
| **public** **class** TryWithResources  { **public** **static** **void** main(String[] args)  { **try** {*lerArquivo*();}    //tem preferencia dobre o segundo catch  **catch** (FileNotFoundException e)  {System.***out***.println(e.getMessage());}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **public** **static** **void** lerArquivo() **throws** IOException  { Reader reader = **null**;  reader = **new** BufferedReader(  **new** FileReader("../Maratona java/teste.txt"));  reader.close();  }  } |

E dependendo teríamos que fazer até mesmo um try-catch dentro de um bloco *finally*. O *try with resources* nos permite criar uma espécie de método com parâmetro inicializado. A regra é que as variáveis de referência que vão dentro do *try* obrigatoriamente precisam implementar a interface *AutoCloseble*, que apenas possui um método que é o *close()*.

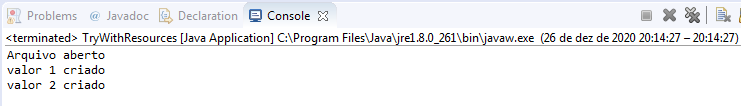
|  |
| --- |
| **public** **class** TryWithResources  { **public** **static** **void** main(String[] args)  { **try** {*lerArquivo*();}  **catch** (IOException e) {}  }  **public** **static** **void** lerArquivo() **throws** IOException  { //podemos inserir mais de um parâmetro  **try**(Reader reader = **new** BufferedReader(  **new** FileReader("../Maratona java/teste.txt")))  {}  }  } |

Perceba que dessa maneira, dentro do método lerArquivo(), não precisamos usar o *catch* mesmo tendo o *try*, desde que tenha criado o *throws* no nome do método, embora seja recomendável implementar o *catch*, conforme mostrado abaixo.

|  |
| --- |
| **public** **class** TryWithResources  { **public** **static** **void** main(String[] args)  {*lerArquivo*();}  **public** **static** **void** lerArquivo()  { //podemos inserir mais de um parâmetro  **try**(Reader reader = **new** BufferedReader(  **new** FileReader("../Maratona java/teste.txt")))  {}  **catch**(IOException e)  {System.***out***.println(e.getMessage());}  }  } |

**OBS:** **Quando tentamos criar mais de um recurso dentro de dos parâmetros do *try(...)*, as inicializações dos recursos ocorrem na ordem contrária do que foi escrita.**

|  |
| --- |
| **public** **class** Valor **implements** AutoCloseable  { **private** String nome;    **public** Valor(String nome)  { **this**.nome = nome;  System.***out***.println(**this**.nome + " criado");  }  @Override **public** **void** close() **throws** Exception {}  } |
| **public** **class** TryWithResources  { **public** **static** **void** main(String[] args)  { *lerArquivo*();  *criaValor*();  }  **public** **static** **void** lerArquivo()  { //podemos inserir mais de um parâmetro  **try**(Reader reader = **new** BufferedReader(  **new** FileReader("F:/Users/eduardowmu/Desktop/"  + "meusdoc/estudo/FATEC/7 - OUTROS SEMESTRES/"  + "YOUTUBE/Maratona java/teste.txt")))  { System.***out***.println("Arquivo aberto");  reader.close();  }  **catch**(IOException e)  {System.***out***.println(e.getMessage());}  }    **public** **static** **void** criaValor()  { **try**(Valor valor1 = **new** Valor("valor 1");  Valor valor2 = **new** Valor("valor 2"))  { valor1.close();  valor2.close();  }  **catch**(Exception e)  {System.***out***.println(e.getMessage());}  }  } |

****

**Figura – bloco try com vários recursos.**

O java possui milhares de classes já pré definidas de exceções, então provavelmente para a maioria dos problemas podemos resolver usando uma dessas classes. Mas quando estamos desenvolvendo um software sua regra de negócio pode ser única. Por exemplo, caso estejamos fazendo um método e este realiza um login, este método vai no mínimo pegar um nome de usuário e uma senha. Caso as informações não baterem com as informações de autenticação, significa que a pessoa não pode logar e podemos lançar uma *exception*. Iremos então criar essa *exception*.

|  |
| --- |
| /\*Exceção do tipo checked, pois está sendo diretamente uma  \*subclasse de exception\*/  **public** **class** LoginInvalidoException **extends** Exception  { //exceção customizada para o nosso problema  **public** LoginInvalidoException()  {**super**("Usuario ou Senha invalida");}  } |
| **public** **class** CustomExceptionTest  { **public** **static** **void** main(String[] args)  { **try** {*logar*();}  **catch** (LoginInvalidoException e)  {System.***out***.println(e.getMessage());}  }  **private** **static** **void** logar() **throws** LoginInvalidoException  { //dados de autenticação  String userBD = "123";  String senhaBD = "111";    //dados digitado pelo usuario  String usuarioDig = "Goku";  String senhaDig = "123";    **if**(!userBD.equals(usuarioDig) ||  !senhaBD.equals(senhaDig))  {**throw** **new** LoginInvalidoException();}    **else** {System.***out***.println("Logado");}  }  } |

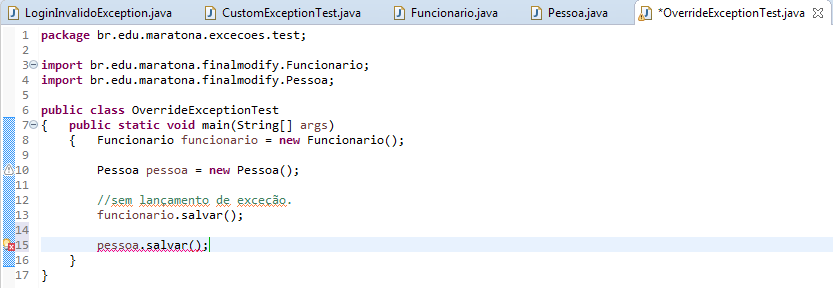
**OBS: Nunca coloquem nenhum tipo de tratamento nas classes de *Exception*. Sempre que criarmos uma classe de exceções customizadas, não devemos colocar métodos ou outra coisa para tratar as exceptions, pois o ciclo de vida de software é mais novo do que imaginamos. Então se fizermos algum tratamento hoje, daqui uns 6 meses o tratamento pode não ser mais válido.**

# Regras de sobreescrita em Exceptions

Quando sobre escrevemos um método, podemos dizer que o método criado não lança nenhuma exceção, pois o código pode mudar completamente e o novo pode não lançar nenhuma exceção, como acontece no seguinte teste com duas classes.

|  |
| --- |
| **public** **class** Pessoa  { **public** **void** salvar()  **throws** LoginInvalidoException,  FileNotFoundException  {}  } |
| **public** **class** Funcionario **extends** Pessoa  {**public** **void** salvar() {}} |
| **public** **class** OverrideExceptionTest  { **public** **static** **void** main(String[] args)  { Funcionario funcionario = **new** Funcionario();    Pessoa pessoa = **new** Pessoa();    funcionario.salvar();  }  } |

Mas se fizermos o mesmo para um objeto da classe Pessoa, ou seja, da classe mãe que lança uma *exception*, devemos tratar com *try-catch*.



**Figura – Exceção lançada.**

Podemos tratar da seguinte maneira, quando lançamos múltiplas *exceptions*.

|  |
| --- |
| **public** **class** OverrideExceptionTest  { **public** **static** **void** main(String[] args)  { Funcionario funcionario = **new** Funcionario();  Pessoa pessoa = **new** Pessoa();  //sem lançamento de exceção.  funcionario.salvar();    **try** {pessoa.salvar();}  **catch** (FileNotFoundException | LoginInvalidoException e)  {System.***out***.println(e.getMessage());}  }  } |

Podemos declarar mais de uma *exception* em um método, exceto o *IOException*, pois foi uma exceção que não foi declarada na classe pessoa e essa exceção é uma super classe. Se um método que estamos tentando sobrescrever tentar lançar uma exceção de uma super classe ou uma exceção do tipo checked que não foi lançada lá, teremos um erro de compilação. Não podemos lançar nenhuma exceção que não esteja importada, a num ser, que seja do tipo *RunTimeException* ou suas subclasses.

|  |
| --- |
| **public** **class** Pessoa  { **public** **void** salvar()  **throws** LoginInvalidoException,  FileNotFoundException  {}  } |
| **public** **class** Funcionario **extends** Pessoa  { **public** **void** salvar()  **throws** RuntimeException,  FileNotFoundException  {}  } |
| **public** **class** OverrideExceptionTest  { **public** **static** **void** main(String[] args)  { Funcionario funcionario = **new** Funcionario();    Pessoa pessoa = **new** Pessoa();    //sem lançamento de exceção.  **try** {funcionario.salvar();}  **catch** (FileNotFoundException | RuntimeException e1)  {System.***out***.println(e1.getMessage());}    **try** {pessoa.salvar();}  **catch** (FileNotFoundException | LoginInvalidoException e)  {System.***out***.println(e.getMessage());}  }  } |

**Resumo:**

Em uma sobre escrita de método, podemos lançar apenas *exceptions* do tipo *RuntimeException* ou suas subclasses.

# Assertivas

Sabemos que quando estamos desenvolvendo não podemos presumir absolutamente nada, ou seja, não podemos pensar em “eu acho que tal valor nunca vá chegar a tanto...”. No caso do método da classe Funcionario da aula anterior, com o lançamento das exceções, não podemos pensar que nunca será lançada. Isso é uma coisa natural e por causa disso, a partir da versão 4 do java, foi introduzido as assertivas, que servem para verificar valores e caso esse valor não seja o que estás esperando, vá lançar um erro e não uma exceção.

Vamos criar uma classe de teste chamada AssertsTeste e criar um método de calculo de salario. Sabemos que salário nunca deve ser negativo. Então, podemos lançar um *throws* para caso retorne um valor negativo.

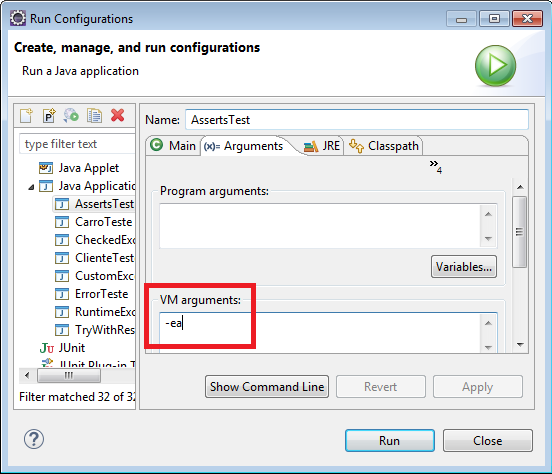
|  |
| --- |
| **public** **class** AssertsTest  { **public** **static** **void** main(String[] args)  {*calculaSalario*(1000);}  **private** **static** **void** calculaSalario(**double** salario)  { **if**(salario > 0)  {System.***out***.println("Salario calculado " + salario);}    **else**  {System.***out***.println("Calculo de salario errado");}  }  } |

Esse *else* do método é algo muito ruim, pois forçará sempre ter salario > 0. Então, para isso foi criado o *Assert*, pois ela sempre irá garantir que sempre iremos passar esse valor que estamos esperando.

|  |
| --- |
| **public** **class** AssertsTest  { **public** **static** **void** main(String[] args)  {*calculaSalario*(1000);}  **private** **static** **void** calculaSalario(**double** salario)  { //estamos falando que a comparação dentro do assert  //deve ser verdadeiro, pois caso contrário, nos dirá  //que nosso código tem um erro muito sério e não estávemos  //preparados para esse tipo de erro. Com isso podemos  //garantir, em tempo de fase de testes que salario sempre  //vá ser maior que zero.  **assert**(salario > 0);  /\*if(salario > 0)  {System.out.println("Salario calculado " + salario);}    else  {System.out.println("Calculo de salario errado");}\*/  }  } |

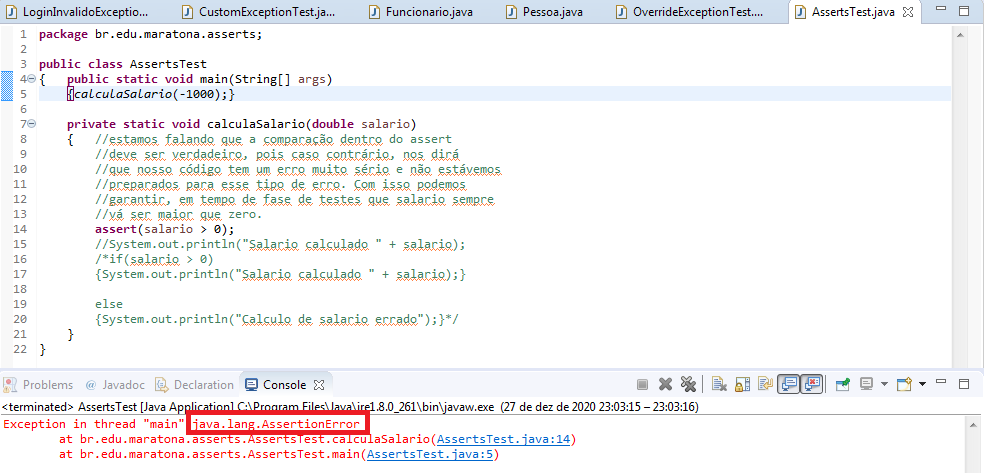
**OBS: perceba que foi usado um método tipo *private*, pois é um péssimo habito usá-lo em métodos públicos, pois qualquer pessoa que estiver usando seu código pode fazer a chamada do método, no caso, calcularSalario() se fosse público.**

O *assert* geralmente é usado durante a fase de desenvolvimento. Quando jogamos para produção o *assert* é ignorado.



**Figura – Habilitando asserts.**

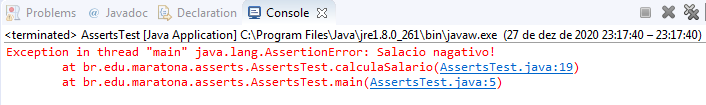
Se alterarmos o valor do salario para -1000 e executarmos o programa, isso nos retornará um erro.



**Figura – Erro gerado pelo assert.**

Quando colocamos isso em produção, este assert estão sempre desativadas, as JVMs simplesmente irá ignorar a linha de código com *asserts*. Se caso quiséssemos criar em um método público, o melhor que podemos fazer é lançar uma exceção.

|  |
| --- |
| **public** **class** AssertsTest  { **public** **static** **void** main(String[] args)  { *calculaSalario*(-1000.0);    **try** {*calculaSalario*(-10);}  **catch**(IllegalArgumentException e)  {System.***out***.println(e.getMessage());}  }  **private** **static** **void** calculaSalario(**double** salario)  { //estamos falando que a comparação dentro do assert  //deve ser verdadeiro, pois caso contrário, nos dirá  //que nosso código tem um erro muito sério e não estávemos  //preparados para esse tipo de erro. Com isso podemos  //garantir, em tempo de fase de testes que salario sempre  //vá ser maior que zero.  **assert**(salario > 0): "Salacio nagativo!";  }    **public** **static** **void** calculaSalario(**int** salario)  **throws** IllegalArgumentException  { **if**(salario < 0)  {**throw** **new** IllegalArgumentException("Salario negativo!");}  }  } |



**Figura – Erro com indicação por mensagem.**

O *assert* sempre retornará um valor booleano, e é um meio de garantir que seu código está seguindo o que presumimos o que deveria acontecer. Isso economiza código, pois evita de usar if-else, e por padrão vem desabilitado.

Outra informação importante é que os *asserts* podem ser usadas em métodos públicos mas existe apenas um caso, em que tenhamos certeza absoluta de que nossa variável em questão não possa assumir outro valor qualquer.

# Wrapper Classes

Wrapper é uma das classes utilitárias do Java, que são classes que facilitam nosso trabalho no dia a dia. Ao invés de crirar um código do zero para resolver um problema, muitas ou uma dessas classes já resolvem seus problemas, com muito menos linhas de código.

Para criar a classe Wrapper, só precisamos criar tipos primitivos, exceto os tipo *char* e *int*, com o nome da variável primitiva com letra maiúscula, e assim estaremos criando um Wrapper do tipo primitivo. Por exemplo:

|  |
| --- |
| **public** **class** WrappersTest  { **public** **static** **void** main(String[] args)  { //os 8 tipos primitivos do menor para o maior numericos  **byte** byteValue = 1;  **short** shortValue = 1;  **int** intValue = 10;  **long** longValue = 10L;  **float** floatValue = 10F;  **double** doubleBlue = 100.0;    //não numericos  **char** charValue = 'A';  **boolean** booleanValue = **true**;    //classes Wrapper que encapsulam os tipos primitivos  Byte byteWrapper = 1;  Short shortWrapper = 1;  Integer intWrapper = 10;  Long longWrapper = 10L;  Float floatWrapper = 10F;  Double doubleWrapper = 100.0;  Character charWrapper = 'A';  Boolean booleanWrapper = **true**;  }  } |

Apenas precisamos, nos casos dos wrappers, que não estaremos trabalhando mais com tipos primitivos, ou seja, agora estamos trabalhando com objetos e as regras de herança e polimorfismo funcionam para essas classes. Podemos remover o “L” e “F” dos valores primitivos *long* e *float*.

Mas quando trabalhamos com classes Wrapper, no java, as coisas não parecem ser tão simples assim. Trabalhando com Byte, Short e Integer não teremos tantos problemas, pois o java consegue identificar que, um byte de 1 a 127, seja um tipo Byte. Um valor maior que 127, será considerado int. Mas as coisas começam a ficar mais complicadas quando começamos a trabalhar com Long, Float e Double, pois todos os seus valores numéricos são filhos da super classe Number. Não podemos pegar o valor intWrapper e atribuir a variável longWrapper por exemplo.

Agora falaremos sobre AutoBoxing e Unboxing, sendo o primeiro, que nada mais é do que estávamos fazendo, ou seja, pegando um tipo primitivo e colocando em um wrapper. O Unboxing é por exemplo, declarar um primitivo e atribuí-lo um valor de um wrapper.

*int valor = integerWrapper;*

Além disso, quando por exemplo queremos instanciar um tipo *Integer*, podemos inserir um valor numérico String, que desde que seja um valor numérico, o java irá automaticamente convertê-lo em um valor tipo *Integer*.

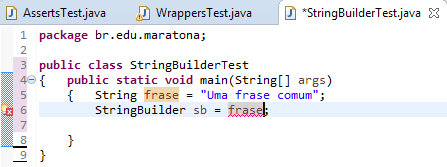
*Integer integerValue = new Integer(“10”);*

E isso é válido para qualquer classe Wrapper, desde que seja inicializado com valores convenientes a suas origens. No caso dos tipo *Boolean*, poderá ser escrito com letras maiúscula ou minúscula, pois nesse caso não é case sensitive. Cada classe wrapper é conhecida como utilitária, pois facilitam nossas vidas, com seus métodos. O *Character* por exemplo tem *isDigit(‘1’)*, que retorna um booleano se o valor inserido for tipo numérico.

# StringBuilder e StringBuffer

Essas classes possuem uma performance que a classe String, pois esta classe todas as vezes que a criamos, ela é imutável, por causa disso todas as vezes que fazemos uma alteração, uma nova string é alocada no pool de strings no reap de memória iniciado pela JVM.

A StringBuilder funciona da seguinte maneira, esta não vai criar uma string no pool de strings, a StringBuilder não é uma String, são classes diferentes. Não podemos diretamente atribuir uma String em um StringBuilder pois ambas são classes diferentes.



**Figura – Erro de compilação: String X StringBuilder.**

|  |
| --- |
| **public** **class** StringBuilderTest  { **public** **static** **void** main(String[] args)  { String frase = "Uma frase comum";  //criação de uma String com 10 caracteres  StringBuilder sb = **new** StringBuilder(10);  //método com sobrecarga, mas normalmente usamos  //a que recebe String  sb.append(frase);  //isso funciona pois automaticamente e indiretamente  //o java estará usando o método toString() da classe  //StringBuilder  System.***out***.println(sb);  }  } |

## reverse()

Outro método que demonstra uma performance melhor que o String, é o Reverse.

|  |
| --- |
| **public** **class** StringBuilderTest  { **public** **static** **void** main(String[] args)  { String frase = "Uma frase comum";    //criação de uma String com 10 caracteres  StringBuilder sb = **new** StringBuilder(10);    //método com sobrecarga, mas normalmente usamos  //a que recebe String  sb.append(frase);    //isso funciona pois automaticamente e indiretamente  //o java estará usando o método toString() da classe  //StringBuilder  System.***out***.println(sb);    //metodo reverse, reverte a ordem da string  System.***out***.println(sb.reverse());  }  } |

## delete()

Funciona como o método substring da classe String.

|  |
| --- |
| **Public** **class** StringBuilderTest  { **public** **static** **void** main(String[] args)  { String frase = “Uma frase comum”;    //criação de uma String com 10 caracteres  StringBuilder sb = **new** StringBuilder(10);    //método com sobrecarga, mas normalmente usamos  //a que recebe String  sb.append(frase);    //isso funciona pois automaticamente e indiretamente  //o java estará usando o método toString() da classe  //StringBuilder  System.***out***.println(sb);    //étodo reverse, reverte a ordem da string  //System.out.println(sb.reverse());    //delete  System.***out***.println(sb.delete(0, 4));  }  } |

## insert()

É o oposto do delete().

|  |
| --- |
| **public** **class** StringBuilderTest  { **public** **static** **void** main(String[] args)  { String frase = "Uma frase comum";    //criação de uma String com 10 caracteres  StringBuilder sb = **new** StringBuilder(10);    //método com sobrecarga, mas normalmente usamos  //a que recebe String  sb.append(frase);    //isso funciona pois automaticamente e indiretamente  //o java estará usando o método toString() da classe  //StringBuilder  System.***out***.println(sb);    //metodo reverse, reverte a ordem da string  //System.out.println(sb.reverse());    //delete  System.***out***.println(sb.delete(0, 4));    //insert  System.***out***.println(sb.insert(11, " hello world"));  }  } |

O *StringBuffer* é uma classe muito similar com o *StringBuilder*, com a diferença que na primeira, os métodos são sincronizados, mas possui os mesmos métodos do segundo.

# Classe Date e Calendar

Data sempre foi algo um pouco complicado de trabalhar, pois existem datas do computador, do servidor, datas de diferentes países, então o java oferece uma customização para trabalharmos com internacionalização de datas de uma forma bem simples, usando a classe *Locale*. Mas por enquanto vamos trabalhar com o básico, a princípio com a primeira classe a se trabalhar com Data, que é o *Date*.

A classe *Date* nunca foi muito boa para se trabalhar com internacionalização, por isso foi adicionado uma nova classe chamada *Calendar*, que até o java 7, é a classe responsável para se trabalhar com internacionalização de datas. A partir da versão 8 é que foi criado mais um pacote de datas chamada *Time*, que oferece muito mais funcionalidades do que ambas as classes anteriores. A classe *Date* já está praticamente com todos os seus métodos obsoletos.

## Date:

Na classe *Date*, todas as datas são passadas em milissegundos.

|  |
| --- |
| **import** java.util.Date;  //pegando a hora atual  **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  System.***out***.println(date);  }  } |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date(1000000000L);  System.***out***.println(date);  }  } |

Se quisermos saber, no primeiro caso, quanto ficaria essa data em milissegundos, basta chamar o método *getTime()*.

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  System.***out***.println(date.getTime());  }  } |

Perceba que com a classe *Date* é muito complicado trabalhar com datas, pois por exemplo, quiséssemos trabalhar com uma hora, teríamos que pegar o valor do *getTime()* e somar com equivalente a uma hora em milissegundos.

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);  }  } |

## Calendar

*Calendar* é uma classe Abstrata, portanto não podemos criar uma instância do modo convencional usando *Calendar*. Para podermos criar um objeto do tipo, podemos fazer da seguinte maneira:

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);    Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar.getTime());  }  } |

Se mandarmos imprimir apenas o objeto *calendar*, irá imprimir um monte de informações úteis, o que nos mostra que o *Calendar* é bem mais completo que o *Date*. Se quisermos fazer com que o esta classe receba um *Date*, podemos fazer da seguinte maneira:

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);    Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar.getTime());    //ojeto calendar recebendo um tipo date  calendar.setTime(date);  System.***out***.println(calendar.getTimeInMillis());  System.***out***.println(calendar.getTimeZone().getDisplayName());  System.***out***.println(calendar.getTime());}  } |

O método *getTime()* da classe *Calendar* retorna um tipo *Date*, então podemos fazer o modo inverso também.

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);    Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar.getTime());    //ojeto calendar recebendo um tipo date  calendar.setTime(date);  System.***out***.println(calendar.getTimeInMillis());  System.***out***.println(calendar.getTimeZone().getDisplayName());  System.***out***.println(calendar.getTime());  System.***out***.println(calendar.getFirstDayOfWeek());  System.***out***.println(calendar.***DAY\_OF\_WEEK***);    //um Date recebendo um tipo Calendar através do método  //getTime()  Date date2 = calendar.getTime();  System.***out***.println(date2);  }  } |

Se quisermos adicionar duas horas:

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);    Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar.getTime());    //ojeto calendar recebendo um tipo date  calendar.setTime(date);  System.***out***.println(calendar.getTimeInMillis());  System.***out***.println(calendar.getTimeZone().getDisplayName());  System.***out***.println(calendar.getTime());  System.***out***.println(calendar.getFirstDayOfWeek());  System.***out***.println(calendar.***DAY\_OF\_WEEK***);    //um Date recebendo um tipo Calendar através do método  //getTime()  Date date2 = calendar.getTime();  System.***out***.println(date2);    calendar.add(Calendar.***HOUR***, 2);  System.***out***.println(calendar.getTime());  }  } |

Outro método interessante é o *roll()*.

|  |
| --- |
| **public** **class** ManipulacaoDatasTeste  { **public** **static** **void** main(String[] args)  { Date date = **new** Date();  date.setTime(date.getTime() + 3600000);  System.***out***.println(date);    Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar.getTime());    //ojeto calendar recebendo um tipo date  calendar.setTime(date);  System.***out***.println(calendar.getTimeInMillis());  System.***out***.println(calendar.getTimeZone().getDisplayName());  System.***out***.println(calendar.getTime());  System.***out***.println(calendar.getFirstDayOfWeek());  System.***out***.println(calendar.***DAY\_OF\_WEEK***);    //um Date recebendo um tipo Calendar através do método  //getTime()  Date date2 = calendar.getTime();  System.***out***.println(date2);    //add horas no tempo  calendar.add(Calendar.***HOUR***, 2);  System.***out***.println(calendar.getTime());    //este mostra os meses do ano de 0 a 11  calendar.roll(Calendar.***MONTH***, 11);  Date date3 = calendar.getTime();  System.***out***.println(date3);  }  } |

# Formatação de datas

Iremos nesta aula fazer formatação de datas usando a classe *DateFormat*. Esta classe é bem simples pois temos 6 formas de formatarmos datas.

|  |
| --- |
| **public** **class** DateFormatTeste  { **public** **static** **void** main(String[] args)  { Calendar calendar = Calendar.*getInstance*();  DateFormat[] df = **new** DateFormat[6];    //1º formato  df[0] = DateFormat.*getInstance*();    //2º formato  df[1] = DateFormat.*getDateInstance*();    //3º formato  df[2] = DateFormat.*getDateInstance*(DateFormat.***SHORT***);    //4º formato  df[3] = DateFormat.*getDateInstance*(DateFormat.***MEDIUM***);    //5º formato  df[4] = DateFormat.*getDateInstance*(DateFormat.***LONG***);    //6º formato  df[5] = DateFormat.*getDateInstance*(DateFormat.***LONG***);    **for**(**int** i = 0; i < df.length; i++)  {System.***out***.println(df[i].format(calendar.getTime()));}  }  } |

A classe *DateFormat* é uma classe que já tem alguns tipos pré definidos para imprimirmos. Agora iremos ver a classe *SimpleDateFormat*, que nos permite formatar datas da forma que quisermos. É uma classe concreta cujos construtores podem receber como parâmetro a forma de como queremos formatar as datas, porém é necessário seguir um padrão.

<https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>

|  |
| --- |
| **public** **class** SimpleDateFormatTeste  { **public** **static** **void** main(String[] args)  { Calendar calendar = Calendar.*getInstance*();  DateFormat df = **new** SimpleDateFormat("dd/MM/YYYY");  System.***out***.println(df.format(calendar.getTime()));  }  } |

Podemos formatar, junto com a data, o horário.

|  |
| --- |
| **public** **class** SimpleDateFormatTeste  { **public** **static** **void** main(String[] args)  { Calendar calendar = Calendar.*getInstance*();  DateFormat df = **new** SimpleDateFormat("dd/MM/YYYY");  System.***out***.println(df.format(calendar.getTime()));    //tudo que estiver dentro de aspas simples,  //será desconsiderado  DateFormat df2 = **new** SimpleDateFormat(  "dd/MM/YYYY 'at' HH:mm:ss");    System.***out***.println(df2.format(calendar.getTime()));  }  } |

**OBS: Atenção que o método *format()* não retorna um *Date*  e sim uma String.**

# Internacionalização com a classe Locale

Nesta aula iremos trabalhar com a classe *Locale* para formatarmos, por exemplo, datas de acordo com países que desejarmos.

Esta classe *Locale* é muito simples de se utilizar. Primeiro precisamos chamar a própria classe, do pacote *java.util* e instanciamos um objeto do tipo. No construtor, podemos passar até dois parâmetros: a língua e/ou país. O Java segue um padrão da ISO639.

|  |
| --- |
| **public** **class** LocaleTeste  { **public** **static** **void** main(String[] args)  { //idioma inglês  Locale locale = **new** Locale("en");    //idioma inglês americano  Locale locale2 = **new** Locale("pt", "BR");    //instanciando um calendar  Calendar calendar = Calendar.*getInstance*();    //instanciando um DateFormat  DateFormat df = DateFormat.*getDateInstance*(DateFormat.***FULL***, locale2);    System.***out***.println(df.format(calendar.getTime()));    DateFormat df2 = DateFormat.*getDateInstance*(DateFormat.***FULL***, locale);    System.***out***.println(df2.format(calendar.getTime()));  }  } |

Podemos usar outros métodos dessa classe *Locale*

|  |
| --- |
| **public** **class** LocaleTeste  { **public** **static** **void** main(String[] args)  { //idioma inglês  Locale locale = **new** Locale("fr", "FR");    //idioma inglês americano  Locale locale2 = **new** Locale("pt", "BR");    //instanciando um calendar  Calendar calendar = Calendar.*getInstance*();    //instanciando um DateFormat  DateFormat df = DateFormat.*getDateInstance*(DateFormat.***FULL***, locale2);    System.***out***.println(df.format(calendar.getTime()));    DateFormat df2 = DateFormat.*getDateInstance*(DateFormat.***FULL***, locale);    System.***out***.println(df2.format(calendar.getTime()));  System.***out***.println(locale.getDisplayLanguage() + ", " +  locale.getDisplayLanguage(locale));  System.***out***.println(locale.getDisplayCountry());  }  } |

# Formatação de Números com NumberForma

Assim como a Classe *DateFormat*, a *NumberFormat* também é uma classe abstrata.

|  |
| --- |
| **public** **class** NumberFormatTest  { **public** **static** **void** main(String[] args)  { **double** valor = 123.4567;    //definindo tipos de moedas pelo país  Locale franca = **new** Locale("fr", "FR");  Locale port = **new** Locale("pt", "PT");  Locale brasil = **new** Locale("pt", "BR");  Locale eua = **new** Locale("en", "US");    NumberFormat[] nf = **new** NumberFormat[4];  nf[0] = NumberFormat.*getCurrencyInstance*(franca);  nf[1] = NumberFormat.*getCurrencyInstance*(port);  nf[2] = NumberFormat.*getCurrencyInstance*(brasil);  nf[3] = NumberFormat.*getCurrencyInstance*(eua);    **for**(NumberFormat n:nf)  {System.***out***.println(n.format(valor));}  }  } |

Um método muito utilizado dessa classe é o *parse()*.

|  |
| --- |
| **import** java.text.DecimalFormat;  **import** java.text.NumberFormat;  **import** java.text.ParseException;  **import** java.util.Locale;  **public** **class** NumberFormatTest  { **public** **static** **void** main(String[] args)  { **double** valor = 123.4567;    System.***out***.println(valor);    //definindo tipos de moedas pelo país  Locale franca = **new** Locale("fr", "FR");  Locale port = **new** Locale("pt", "PT");  Locale brasil = **new** Locale("pt", "BR");  Locale eua = **new** Locale("en", "US");    NumberFormat[] nf = **new** NumberFormat[4];  nf[0] = NumberFormat.*getCurrencyInstance*(franca);  nf[1] = NumberFormat.*getCurrencyInstance*(port);  nf[2] = NumberFormat.*getCurrencyInstance*(brasil);  nf[3] = NumberFormat.*getCurrencyInstance*(eua);    **for**(NumberFormat n:nf)  {System.***out***.println(n.format(valor));}    NumberFormat df = **new** DecimalFormat();    String valorString = "123,4567";  //convertendo valor String numerico em double  **try**  { System.***out***.println(df.parse(valorString));  df.setParseIntegerOnly(**true**);  System.***out***.println(df.parse(valorString));  }  **catch** (ParseException e)  {System.***out***.println(e.getMessage());}  }  } |

# Regex – Expressões Regulares

*Regex* é como se fosse uma linguagem dentro de outra linguagem. Várias linguagens oferecem suporte à expressões regulares, como o java no caso. *Regex* é muito poderosa quando precisamos usar padrões para buscar determinadas informações dentro de grandes arquivos de texto. Por exemplo: suponha que temos um grande arquivo de texto e queremos pegar nesse arquivo todos os CEPs que estão disponíveis lá ou todos os números de telefone. Se estiverem seguindo o mesmo padrão, conseguimos facilmente pegar todos esses dados através de *Regex*.

No java precisaremos de duas classes para se trabalhar com *Regex*:

## A) Pattern

Classe que procura padrões.

## B) Matcher

Encontra os padrões.

|  |
| --- |
| **import** java.util.regex.Matcher;  **import** java.util.regex.Pattern;  **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "abaaaba";    //queremos encontrar "ab" dentro da variável texto  String regex = "ab";    Pattern pattern = Pattern.*compile*(regex);  Matcher matcher = pattern.matcher(texto);    System.***out***.println("texto: " + texto);  System.***out***.println("indice: 0123456789");  System.***out***.println("expressao: " + matcher.pattern());  System.***out***.println("posições encontradas");    //enquanto o matcher encontra padrão desse texto  **while**(matcher.find())  {System.***out***.println(matcher.start() + " ");}  }  } |

Isso significa que nas posições 0 e 4 foram encontradas a expressão padrão “ab”. Mas se alterarmos o texto para “abababa” e a expressão padrão para “aba”?

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "abababa";    //queremos encontrar "ab" dentro da variável texto  String regex = "aba";    Pattern pattern = Pattern.*compile*(regex);  Matcher matcher = pattern.matcher(texto);    System.***out***.println("texto: " + texto);  System.***out***.println("indice: 0123456789");  System.***out***.println("expressao: " + matcher.pattern());  System.***out***.println("posições encontradas");    //enquanto o matcher encontra padrão desse texto  **while**(matcher.find())  {System.***out***.println(matcher.start() + " ");}  }  } |

Isso aconteceu pois o *Regex* irão sempre começar da esquerda para a direita, e uma vez que tal expressão em determinada expressão for usada no meio do texto, ou seja, no caso, a expressão “aba” da posição 0 ao 2 não será utilizado para a procura do próximo padrão. O Regex não utiliza um caractere que já foi encontrado em expressão anterior.