[Modificador *final* 4](#_Toc72700027)

[A) Atributos: 4](#_Toc72700028)

[B) Classes 8](#_Toc72700029)

[C) Métodos 9](#_Toc72700030)

[ENUM 9](#_Toc72700031)

[A) Criando classes: 11](#_Toc72700032)

[B) Criar como Atributo de classe: 12](#_Toc72700033)

[Exceptions 18](#_Toc72700034)

[A) Error 19](#_Toc72700035)

[B) Exception 21](#_Toc72700036)

[C) Checked 21](#_Toc72700037)

[D) Unchecked 23](#_Toc72700038)

[Regras de sobreescrita em Exceptions 27](#_Toc72700039)

[Assertivas 29](#_Toc72700040)

[Wrapper Classes 33](#_Toc72700041)

[StringBuilder e StringBuffer 35](#_Toc72700042)

[reverse() 35](#_Toc72700043)

[delete() 36](#_Toc72700044)

[insert() 37](#_Toc72700045)

[Classe Date e Calendar 38](#_Toc72700046)

[Date: 38](#_Toc72700047)

[Calendar 39](#_Toc72700048)

[Formatação de datas 43](#_Toc72700049)

[Internacionalização com a classe Locale 45](#_Toc72700050)

[Formatação de Números com NumberForma 46](#_Toc72700051)

[Regex – Expressões Regulares 48](#_Toc72700052)

[A) Pattern 48](#_Toc72700053)

[B) Matcher 48](#_Toc72700054)

[C) “\d” 50](#_Toc72700055)

[D) “\D” 51](#_Toc72700056)

[E) “\s” 51](#_Toc72700057)

[F) “\S” 52](#_Toc72700058)

[G) “\w” 53](#_Toc72700059)

[H) “\W” 53](#_Toc72700060)

[Quantificadores em Regex 57](#_Toc72700061)

[A) “?” 57](#_Toc72700062)

[B) “\*” 58](#_Toc72700063)

[C) “+” 58](#_Toc72700064)

[D) “[]” 58](#_Toc72700065)

[E) “()” 58](#_Toc72700066)

[F) “|” (pipe) 58](#_Toc72700067)

[G) “$” 58](#_Toc72700068)

[H) “^” 66](#_Toc72700069)

[Tokens e delimitadores com String e Scanner 67](#_Toc72700070)

[A) Tratamento de token e delimitadores por String 67](#_Toc72700071)

[B) Tratamento de token e delimitadores por Scanner 67](#_Toc72700072)

[Resources Bundles 69](#_Toc72700073)

[File Classes 71](#_Toc72700074)

[A) File 71](#_Toc72700075)

[B) FileReader 75](#_Toc72700076)

[C) BufferedReader 76](#_Toc72700077)

[D) FileWritter 77](#_Toc72700078)

[E) BufferedWritter 78](#_Toc72700079)

[Classe File para diretórios 81](#_Toc72700080)

[Classe Console 83](#_Toc72700081)

[NIO 83](#_Toc72700082)

[A) Paths: 84](#_Toc72700083)

[B) Files 84](#_Toc72700084)

[Normalizando Paths 89](#_Toc72700085)

[Resolvendo paths 89](#_Toc72700086)

[Relativizando Paths 91](#_Toc72700087)

[BasicFileAttributes e BasicFileAttributeView 92](#_Toc72700088)

[A) BasicFileAttributes 93](#_Toc72700089)

[B) PosixFileAttributes 93](#_Toc72700090)

[C) DosFileAttributes 94](#_Toc72700091)

[D) BasicFileAttributeView, PosyxFileAttributeView e DosFileAttributeView 95](#_Toc72700092)

[E) FileOwnerAttributeView 95](#_Toc72700093)

[F) AclFileAttributeView 95](#_Toc72700094)

[DosFileAttributes e DosFileAttributeView 96](#_Toc72700095)

[PosixFileAttributes e PosixFileAttributesView 97](#_Toc72700096)

[DirectoryStream e SimpleFileVisitor 98](#_Toc72700097)

[A) DirectoryStream 98](#_Toc72700098)

[B) FileVisitor 99](#_Toc72700099)

[PathMatcher 100](#_Toc72700100)

[InputStream e OutPutStream 103](#_Toc72700101)

[A) InputStream: 104](#_Toc72700102)

[B) OutPutStream 104](#_Toc72700103)

[C) Buffer 105](#_Toc72700104)

[Zipando arquivos com java 106](#_Toc72700105)

[Serialização 110](#_Toc72700106)

[Genérics 117](#_Toc72700107)

[Wild Card 118](#_Toc72700108)

[Classes Genéricas 119](#_Toc72700109)

[Métodos Genéricos 121](#_Toc72700110)

[Inner class 122](#_Toc72700111)

[Classes Anônimas 124](#_Toc72700112)

[Classes Internas Estáticas 126](#_Toc72700113)

[Threads 128](#_Toc72700114)

[A) Threads daimon 128](#_Toc72700115)

[B) Usuário 129](#_Toc72700116)

[Estados de threads 136](#_Toc72700117)

[Parando Execução com Sleep, Yeld e Join 137](#_Toc72700118)

[A) Sleep 137](#_Toc72700119)

[B) Yield 142](#_Toc72700120)

[C) Join 145](#_Toc72700121)

[Sincronismo de código 149](#_Toc72700122)

[Interação entre as threads (wait, notify e notifyAll) 155](#_Toc72700123)

[Concorrência 158](#_Toc72700124)

[Variáveis atômicas 158](#_Toc72700125)

[Lock e ReentrantLock 161](#_Toc72700126)

[Conditions 165](#_Toc72700127)

[**Concorrência: CopyOnWriteArrayList** 167](#_Toc72700128)

[Coleções Concorrentes 168](#_Toc72700129)

[Executors 169](#_Toc72700130)

[Interface Callable 177](#_Toc72700131)

[**Padrões de Projetos (*Design Patterns*)** 179](#_Toc72700132)

[Padrão Builder 179](#_Toc72700133)

[Padrão Factory 182](#_Toc72700134)

[**Parametrizando comportamentos** 183](#_Toc72700135)

[**Lambda** 185](#_Toc72700136)

[Lambdas com Metodos References 190](#_Toc72700137)

[**Novas regras para interfaces: default** 200](#_Toc72700138)

[**Optional** 201](#_Toc72700139)

[**Stream** 204](#_Toc72700140)

[Metodos *finding()* e *matching()* 212](#_Toc72700141)

[Stream especializados 215](#_Toc72700142)

[Reduce 215](#_Toc72700143)

# 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 não possui, 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*:

https://www.devmedia.com.br/conceitos-basicos-sobre-expressoes-regulares-em-java/27539

## 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.

Agora iremos tratar de um assunto importante de *Regex*, que são os meta caracteres das expressões regulares. Os meta caracteres são caracteres que facilitam a expressão regular, pois muitas vezes, quando estamos trabalhando com *Regex*, esta dará cerca de 20 a 30 caracteres, e acaba ficando bem complicado de entendermos. Os meta caracteres então foram feitas justamente para facilitar a busca de determinados caracteres. Por exemplo, digamos que num texto tenhamos um conjunto de caracteres do tipo “bab1278abbyhgaa92178” e queremos remover apenas os números. Os meta caracteres vão servir justamente para esse tipo de problema.

Existem vários tipos de meta caracteres, mas iremos tratar somente dos mais utilizados. Uma informação importante é que todos eles começam com um “\” e para ser inserido em código, é necessário dobrá-lo, pois somente um é uma outra coisa, portanto é necessário inserir “\\”.

## C) “\d”

Este meta caractere busca em uma frase todos os dígitos.

|  |
| --- |
| **import** java.util.regex.Matcher;  **import** java.util.regex.Pattern;  **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abbyhgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\d";    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***.print(matcher.start() + " ");}  }  } |

Perceba que apenas encontrou posições em que dígitos foram encontrados.

## D) “\D”

Encontra tudo o que não for dígito.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abbyhgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\D";    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***.print(matcher.start() + " ");}  }  } |

## E) “\s”

Este meta caracteres identifica espaços em branco.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abby hgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\s";    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***.print(matcher.start() + " ");}  }  } |

## F) “\S”

O que não for espaço em branco. Perceba que no resultado abaixo, foi o oposto da meta anterior.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abby hgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\S";    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***.print(matcher.start() + " ");}  }  } |

## G) “\w”

Procura tudo o que for letras de a-z, A-Z, dígitos e underline.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abby\_hgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\w";    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***.print(matcher.start() + " ");}  }  } |

## H) “\W”

É o oposto do meta caractere anterior. Perceba que a execução do código abaixo só identificou o espaço.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "bab1278abby\_ hgaa92178";    //queremos encontrar "ab" dentro da variável texto  String regex = "\\W";    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***.print(matcher.start() + " ");}  }  } |

Agora iremos ver como encontrar um range de caracteres, letras e assim por diante. Imagine que temos uma determinada frase por exemplo “cafeBASE” e queremos encontrar somente as letras A, B e C, ignorando se são maiúsculas ou não. Para isso, devemos usar os [ ] e dentro dela especificar os conteúdos desejados.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "cafeBABE";    //queremos encontrar "ab" dentro da variável texto  String regex = "[abcABC]";    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***.print(matcher.start() + " ");}  }  } |

Agora imagine que queremos encontrar as letras do alfabeto inteiro, sem importar com se é maiúsculo ou não, basta inserir no meio do intervalo de caracteres um “-“.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { String texto = "123A456a789b101112B";    //queremos encontrar "ab" dentro da variável texto  String regex = "[a-zA-Z]";    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***.print(matcher.start() + " ");}  }  } |

Este mesmo caso vale também para os números, incluindo os números hexadecimais.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { **int** numeroHex = 0xaff;  String texto = "123A456a789b101112B";    //queremos encontrar "ab" dentro da variável texto  String regex = "[a-zA-Z]";    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***.print(matcher.start() + " ");}    System.***out***.println(numeroHex);  }  } |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  String regex = "0[xX][0-9a-fA-F]";    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***.print(matcher.start() + " ");}    //System.out.println(numeroHex);  }  } |

Veja que podemos verificar quais são os valores identificado por essas posições.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  String regex = "0[xX][0-9a-fA-F]";    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

# Quantificadores em Regex

Se olharmos novamente o que fizemos na última aplicação, não apresenta exatamente o que queríamos, pois na nossa variável texto 0xF é um valor válido mas na verdade o numero hexadecimal é 0xFFABC e a mesma coisa temos em 0x10G mas isso não representa numero hexadecimal válido, por causa do “G”.

Portanto nesta aula iremos usar outro recurso para pegar exatamente o que queremos em nosso regex. Estamos procurando algo que comece com “0[xX]” e depois “[0-9a-fA-F]” só que isso uma vez, então a expressão 0x da variável texto casou com a primeira parte e logo depois o resto casou com a segunda parte. Isso acontece porque não temos quantificadores.

Os quantificadores é uma forma de dizer que devemos procurar por uma ou mais ocorrências de determinada expressão. Temos basicamente 4 tipos de quantificadores.

## A) “?”

Significa zero ou um.

## B) “\*”

Significa zero ou mais ocorrência.

## C) “+”

Significa uma ou mais ocorrências.

## D) “[]”

Usado se quisermos customizar, por exemplo, [n,m] que significa números de valor mínimo **n** e máximo **m**.

## E) “()”

Usado para agrupar expressão.

## F) “|” (pipe)

Usado para usar a expressão lógica tipo OU.

## G) “$”

Usado para caso quisermos pular linha.

Então por exemplo, se quiséssemos ter uma expressão regular e temos algo como “o(v|c)o“, estamos pedindo para procurar algo que comece com ‘o’ e tenha ‘v’ OU ‘c’ seguido de ‘o’ novamente. Isso então poderá casar com “ovo” ou “oco”. Outro exemplo seria “maca(rr|c)ão” e isso casaria com “macarrão” ou “macacão”.

Nestes casos precisamos usar o agrupamento, pois o agrupamento vai servir para o nosso exemplo que está em nossa variável regex do nosso último código.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  String regex = "0[xX]([0-9a-fA-F])?";    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

Perceba que no caso acima, tivemos um problema, pois queremos que os valores dentro do parênteses poderiam repetir de 0 a 1 e considerou o “0x” como válido. Na verdade esse valor tem que se repetir de uma ou mais vezes. Para corrigir isso, então trocamos ‘?’ por ‘\*’.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  String regex = "0[xX]([0-9a-fA-F])+";    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

Agora retornou, para a posição 9, o que queríamos, mas o que na verdade foi feito foi uma varredura da variável texto começando do ‘0’ até o último caractere definido no regex e quando não encontrou mais, retorna a procurar do ‘0’ seguido de ‘x’ novamente. Perceba que o nosso primeiro está correto mas o segundo, referente a posição 17, não pois a segunda expressão que encontrou, no texto, tem o ‘G’ e podemos identificar isso como uma expressão inteira, ou seja, um numero hexadecimal não válido. Se repararmos aqui temos um delimitador que é um espaço em branco para cada valor.

Para resolver isso, temos que solicitar que para cada expressão, deva seguir com espaço em branco, ou seja, “\\s”.

|  |
| --- |
| .  .  .  String regex = "0[xX]([0-9a-fA-F])+\\s";  .  .  . |

Acabamos de eliminar o errado mas também deixamos para trás o último que estava correto, o “0x1”, justamente porque este último não está seguido de espaço em branco, então para resolver isso, basta dizermos que cada expressão deverá estar seguido de espaço em branco OU fim de linha.

|  |
| --- |
| .  .  .  String regex = "0[xX]([0-9a-fA-F])+([\\s|$)](about:blank)";  .  .  . |

Desta vez iremos trabalhar um pouco mais de quantificadores e os meta caracteres. Vamos buscar e-mails e data de um determinado texto. Começando com os e-mails:

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  //String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  //String regex = "0[xX]([0-9a-fA-F])+(\\s|$)";    String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail, teste@gmail.br, teste@mail";    /\*o ponto "." é um coringa usado em expressões  \*regulares, portanto devemos sempre usar "\\."  \*no lugar, para dizer que estamos usando um "."  \*texto e não como caractere coringa\*/  String regex = "([a-zA-Z0-9\\.\_-])+@([a-zA-z])+\\.([a-zA-z])+";    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

Mas e se colocássemos [teste@gmail.com.br](mailto:teste@gmail.com.br)?

|  |
| --- |
| String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail, teste@gmail.com.br, teste@mail"; |

Perceba que ele trouxe a mesma coisa, pois em nosso regex colocamos que aceite apenas um “.” após o “@”.

|  |
| --- |
| String regex = "([a-zA-Z0-9\\.\_-])[+@([a-zA-z])+(\\.([a-zA-z])+)](about:blank)+"; |

Mas e se colocássemos o email “[\*@!abrao@mail.com.br](about:blank)”?

|  |
| --- |
| String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail.br, teste@gmail.com.br, teste@mail"; |

Perceba que ele nos trouxe o e-mail, pois quando montamos uma expressão regular estamos querendo encontrar determinado padrão dentro de um texto. Se quisermos validar um texto podemos usar o seguinte da classe *String* ou *Pattern*.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  //String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  //String regex = "0[xX]([0-9a-fA-F])+(\\s|$)";    String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail.br, teste@gmail.com.br, teste@mail";    /\*o ponto "." é um coringa usado em expressões  \*regulares, portanto devemos sempre usar "\\."  \*no lugar, para dizer que estamos usando um "."  \*texto e não como caractere coringa\*/  String regex = "([a-zA-Z0-9\\.\_-])+@([a-zA-z])+(\\.([a-zA-z])+)+";    /\*recomendável usar String\*/  System.***out***.println("E-mail valido? "+"\*@!abrao@mail.br".matches(regex));    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

Perceba que retornou falso pois ele vai validar e não buscar a expressão regular dentro do texto, pois perceba que o e-mail não retornou por completo. Mas podemos melhorar isso e ter o mesmo retorno.

|  |
| --- |
| String regex = "([\\w\\.])[+@([a-zA-z])+(\\.([a-zA-z])+)](about:blank)+"; |

Agora vamos buscar datas. Supondo que queremos sempre que nossa regex obedeça o seguinte formato: dd/mm/aaaa.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  //String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  //String regex = "0[xX]([0-9a-fA-F])+(\\s|$)";    /\*String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail.br, teste@gmail.com.br, teste@mail";\*/    String texto = "05/10/2010 05/05/2015 1/1/01 01/05/95";    /\*o ponto "." é um coringa usado em expressões  \*regulares, portanto devemos sempre usar "\\."  \*no lugar, para dizer que estamos usando um "."  \*texto e não como caractere coringa\*/  //String regex = "([a-zA-Z0-9\\.\_-])+@([a-zA-z])+(\\.([a-zA-z])+)+";  //String regex = "([\\w\\.])+@([a-zA-z])+(\\.([a-zA-z])+)+";    String regex = "\\d\\d/\\d\\d/\\d\\d\\d\\d";    /\*recomendável usar String\*/  System.***out***.println("E-mail valido: "+"\*@!abrao@mail.br".matches(regex));    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  } |

Mas podemos ainda melhorar isso, pois queremos que deve ter exatamente dois dígitos para dia e mês e de 2 a 4 para ano.

|  |
| --- |
| String regex = "\\d{2}/\\d{2}/\\d{2,4}"; |

Perceba que agora ele pegou a última data também.

## H) “^”

Para fechar esse assunto de regex, iremos tratar do caractere de negação em regex, que é “^”, usado para excluir algo que não queremos na nossa regex. Por exemplo, que em nosso texto apareça qualquer coisa que não seja nem ‘a’ nem ‘b’ e nem ‘c’, nossa regex será [^abc]. Esse caractere de negação é muito útil dependendo da lógica que faz.

|  |
| --- |
| **public** **class** RegexTeste  { **public** **static** **void** main(String[] args)  { //int numeroHex = 0xaff;  //String texto = "12 0x 0X 0xFFABC 0x10G 0x1";    //queremos encontrar "ab" dentro da variável texto  //String regex = "0[xX]([0-9a-fA-F])+(\\s|$)";    /\*String texto = "fulano@hotmail.com, 102Abc@gmail.com, "  + "\*@!abrao@mail.br, teste@gmail.com.br, teste@mail";\*/    //String texto = "05/10/2010 05/05/2015 1/1/01 01/05/95";    /\*o ponto "." é um coringa usado em expressões  \*regulares, portanto devemos sempre usar "\\."  \*no lugar, para dizer que estamos usando um "."  \*texto e não como caractere coringa\*/  //String regex = "([a-zA-Z0-9\\.\_-])+@([a-zA-z])+(\\.([a-zA-z])+)+";  //String regex = "([\\w\\.])+@([a-zA-z])+(\\.([a-zA-z])+)+";    String texto = "projeto1.bkp, proj1.java, proj1.class, "  + "proj1final.java, proj2.bkp, proj3.java";    //queremos pegar tudo que comece com proj, mas...  String regex = "proj([^,])\*";    /\*recomendável usar String\*/  System.***out***.println("E-mail valido: "+"\*@!abrao@mail.br".matches(regex));    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())  { //toda vez que encontrar um valor que esta no  //regex, vai escrever  System.***out***.println("posição " + matcher.start() +  ": " + matcher.group());  }    //System.out.println(numeroHex);  }  }  **RESULTADO:**  E-mail valido: false  texto: projeto1.bkp, proj1.java, proj1.class, proj1final.java, proj2.bkp, proj3.java  indice: 0123456789  expressao: proj([^,])\*  posições encontradas  posição 0: projeto1.bkp  posição 14: proj1.java  posição 26: proj1.class  posição 39: proj1final.java  posição 56: proj2.bkp  posição 67: proj3.java |

# Tokens e delimitadores com String e Scanner

Vamos tratar desses dois recursos nesta aula e para isso criemos uma classe com método estático main e declarar um objeto *String* com vários nomes nele.

|  |
| --- |
| String nomes = "Willian, Paulo, Joana, Camila, Anna, John, Matheus"; |

Precisamos agora é separar cada um desses nomes e guarda-los em variáveis para trabalharmos individualmente com cada um deles. Os nomes são os tokens e o delimitador que iremos usar é a vírgula. Temos três formas mas iremos abordar apenas duas de trabalharmos com delimitadores e tokens, que será através de String e Scanner.

## A) Tratamento de token e delimitadores por String

|  |
| --- |
| **public** **class** TokenTest  { **public** **static** **void** main(String[] args)  { String nomes = "Willian, Paulo, Joana, Camila, Anna, John, Matheus";    //pelo método String, passando um regex ou apenas String  String[] tokens = nomes.split(",");    //iteração com array de String  **for**(String nome:tokens) {System.***out***.print(nome);}  }  } |
| **Resultado:**  Willian Paulo Joana Camila Anna John Matheus |

## B) Tratamento de token e delimitadores por Scanner

O construtor do scanner não aceita parâmetros vazios, mas vem sobrecarregado de várias opções, pois pode receber um arquivo, stream e também String. Quando se trabalha com *Scanner*, temos a opção de pegar o token e fazer uma conversão automaticamente. Por padrão essa classe utiliza o espaço. Caso queiramos trocar o delimitador temos que chamar a variável objeto do tipo Scanner e usar o método *userDelimiter()*. Mas o jeito mais simples de se trabalhar é transformando tudo para *String*. Diferentemente da classe String precisamos trabalhar com *while*.

|  |
| --- |
| **public** **class** ScannerTest  { **public** **static** **void** main(String[] args)  { Scanner scan = **new** Scanner("1 true 100 oi");  //verifica dentro desse ojeto scan através  //do delimitador " " se existe uma próxima  //posição. Então primeiro verifica se existe,  //para depois pegarmos. O método next() irá  //devolver o valor e em seguida irá mudar o  //índice automaticamente. E quando chegar no  //final, quando hasNext() irá finalizar o looping  **while**(scan.hasNext()) {System.***out***.println(scan.next());}  }  } |
| 1  true  100  Oi |

Vamos agora criar um novo Scanner e transformar o antigo em tipos primitivos.

|  |
| --- |
| **public** **class** ScannerTest  { **public** **static** **void** main(String[] args)  { Scanner scan = **new** Scanner("1 true 100 oi");  **while**(scan.hasNext()) {System.***out***.print(scan.next()+" ");}  System.***out***.println("\n");  Scanner scan2 = **new** Scanner("1 true 100 oi");  **while**(scan2.hasNext())  { **if**(scan2.hasNextInt())  { **int** i = Integer.*parseInt*(scan2.next());  System.***out***.print(i + " ");  }    **else** **if**(scan2.hasNextBoolean())  { **boolean** bol = Boolean.*parseBoolean*(scan2.next());  System.***out***.print(bol + " ");  }    **else** {System.***out***.print(scan2.next() + " ");}  }  }  } |
| **Resultado**  1 true 100 oi  1 true 100 oi |

# Resources Bundles

Imagine o seguinte problema, que estamos criando um sistema de tudo isso que fizemos até agora e temos a seguinte tarefa: traduzir todas as saídas do sistema para um cliente de outro idioma. Precisamos primeiramente criar e preparar um arquivo do tipo *Resource bundle*. O java irá procurar este arquivo dentro do pacote *class path* dentro do nosso pacote padrão. Podemos cria-lo de duas formas, uma usando a IDE e outra usando um arquivo tipo *file* criado do zero, com algumas regras.

**Figura – Criação de um arquivo de internacionalização. NomeArquivo\_idioma\_PAISORIGEM.properties**

|  |
| --- |
| #CHAVE = VALOR  hello = hello  good.morning = Good morning |
| **public** **class** ResourceBundleTest  { **public** **static** **void** main(String[] args)  { //instanciando um ResourceBundle, com apenas o nome do arquivo  //e um objeto do tipo Locale, para pegar a localização do  //idioma que queremos traduzir  //System.out.println(Locale.getDefault());  ResourceBundle rb = ResourceBundle.*getBundle*(  "messages", **new** Locale("en", "US"));  System.***out***.println(rb.getString("hello"));  System.***out***.println(rb.getString("good.morning"));  }  } |
| **Resultado:**  hello  Good morning |

Se tentarmos passar um valor que não existe no arquivo, teremos um erro em tempo de execução. Para termos para outros idiomas basta criarmos o mesmo arquivo properties com outro idioma e país, seguindo aquela norma usada para definição do *Locale*. Supondo que queremos usar agora um properties que não seja para uma língua específica e que palavras são muito próximas para qualquer outra língua com mesma escrita, podemos usar o recurso de herança.

**Figura – Criação de arquivo properties genérica.**

|  |
| --- |
| #arquivo genérico  show = show |
| **public** **class** ResourceBundleTest  { **public** **static** **void** main(String[] args)  { //instanciando um ResourceBundle, com apenas o nome do arquivo  //e um objeto do tipo Locale, para pegar a localização do  //idioma que queremos traduzir  //System.out.println(Locale.getDefault());  ResourceBundle rb = ResourceBundle.*getBundle*(  "messages", **new** Locale("en", "US"));  System.***out***.println(rb.getString("hello"));  System.***out***.println(rb.getString("good.morning"));  System.***out***.println(rb.getString("show"));  }  } |
| **Resultado:**  hello  Good morning  show |

Mas se replicarmos essa mesma palavra no arquivo genérico mas com outro significado no arquivo específico?

|  |
| --- |
| #CHAVE = VALOR  hello = hello  good.morning = Good morning  show = concert |
| **Resultado:**  hello  Good morning  Concert |

# File Classes

A partir de agora iremos tratar de um assunto I/O, ou seja, entradas e saídas e as classes que mais iremos utilizar são:

## A) File

É a classe mais básica para criar arquivos.

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **if**(file.createNewFile())  {System.***out***.println("Arquivo criado com sucesso");}  }  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  Arquivo criado com sucesso |

Se tentarmos rodar novamente, ocorrerá uma exceção pois o arquivo já existe.

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **boolean** criado = file.createNewFile();  **if**(criado) {System.***out***.println("Arquivo criado com sucesso: " + criado);}    **else** {System.***out***.println("Arquivo já existe " + criado);}  }  **catch** (IOException e)  {System.***out***.println("Arquivo já existe - " + e.getMessage());}  }  } |
| **Resultado:**  Arquivo já existe false. |

Os métodos interessantes a saber são:

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **boolean** criado = file.createNewFile();  **if**(criado)  {System.***out***.println("Arquivo criado com sucesso: " + criado);}    **else** {System.***out***.println("Arquivo já existe " + criado);}    //se o arquivo ja existe  **if**(file.exists())  { **if**(file.delete())  {System.***out***.println("Arquivo já existe e será deletado");}  }  }  **catch** (IOException e)  { System.***out***.println("Arquivo já existe - " +  e.getMessage());  }  }  } |
| **Resultado:**  Arquivo já existe false  Arquivo já existe e será deletado |

Agora faremos um processo que acontece na vida real.

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("files\\arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **if**(!file.exists())  { **if**(file.createNewFile())  { System.***out***.println("Arquivo criado com "  + "sucesso em " + file.getPath());  }  }    **else**  { **if**(file.delete())  { System.***out***.println("Arquivo ja "  + "existe e foi deletado");  }  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:** |

Se quisermos saber qual o path absoluto do arquivo.

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("files\\arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **if**(!file.exists())  { **if**(file.createNewFile())  { System.***out***.println("Arquivo criado com "  + "sucesso em " + file.getPath());  }  }    **else**  { System.***out***.println(file.getAbsolutePath());  **if**(file.delete())  { System.***out***.println("Arquivo ja "  + "existe e foi deletado");  }  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  F:\Users\eduardowmu\Desktop\meusdoc\estudo\FATEC\7 - OUTROS SEMESTRES\YOUTUBE\Maratona java\Repositorio\JavaKnowledges\MaratonaJava\files\arquivo.txt  Arquivo ja existe e foi deletado |

Para finalizar, em termos de log ou registros de modificação de arquivo.

|  |
| --- |
| **public** **class** FileTest  { **public** **static** **void** main(String[] args)  { //Criando um novo arquivo  //se for criado com apenas o nome, o  //mesmo será criado dentro do projeto.  File file = **new** File("files\\arquivo.txt");  //método que retorna booleano indicando  //se o arquivo foi gerado.  **try**  { **if**(!file.exists())  { **if**(file.createNewFile())  { System.***out***.println("Arquivo criado com "  + "sucesso em " + file.getPath() +  " na data: " + **new** Date(file.lastModified()));  }  }    **else**  { System.***out***.println(file.getAbsolutePath());  **if**(file.delete())  {System.***out***.println("Arquivo ja " + "existe e foi deletado");}  }  }  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  Arquivo criado com sucesso em files\arquivo.txt na data: Sat Jan 16 00:23:48 GMT-03:00 2021 |

## B) FileReader

É uma classe em que podemos criar arquivos e ler o conteúdo do arquivo que desejar. Serve para ler respectivamente caracteres em um arquivo. Pegaremos o exemplo aplicado no item D, para ler o arquivo criado.

|  |
| --- |
| **public** **class** FileWritterReaderTeste  { **public** **static** **void** main(String[] args)  { //criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if**(file.exists())  { **try**  { FileWriter fw = **new** FileWriter(file, **true**);  fw.write("Escrevendo uma mensagem no arquivo"  + "\nEscrevendo mais mensagem.");  fw.flush();  //fecha o arquivo.  fw.close();    //leitura do arquivo  FileReader fr = **new** FileReader(file);  **char**[] text = **new** **char**[500];  **int** size = fr.read(text);  **for**(**char** c:text) {System.***out***.print(c);}  fr.close();  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  }  } |
| **Resultado:**  Escrevendo uma mensagem no arquivoEscrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.\_ |

## C) BufferedReader

É uma classe em que temos uma otimização em cima da *FileReader*, pois esta não é tão otimizada para abertura de grandes arquivos. Já o *BufferedReader* já trabalha criando um buffer de memória e que agiliza bastante o processo de leitura, pois ao invés de fazer a leitura, caractere por caractere, o faz linha por linha.

Otimizando o mesmo código feito em *BufferedWriter*:

|  |
| --- |
| **public** **class** BufferedWRTeste  { **public** **static** **void** main(String[] args)  { //criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if**(file.exists())  { **try**  { FileWriter fw = **new** FileWriter(file, **true**);  BufferedWriter bw = **new** BufferedWriter(fw);    bw.write("Escrevendo uma mensagem no arquivo"  + "\nEscrevendo mais mensagem.");  /\*este método irá se encarregar de encontrar o  \*caractere especial que vai pular linha baseado  \*no sistema operacional.\*/  bw.newLine();  bw.write("Pulando uma linha");  bw.flush();  //fecha o arquivo.  bw.close();    //leitura do arquivo  FileReader fr = **new** FileReader(file);    //Buffered Reader muda um pouco as coisas  BufferedReader br = **new** BufferedReader(fr);  //este método, ao invés de ler caractere por  //caractere, este faz leitura linha a linha  br.readLine();    String str = **null**;  //enquanto não houver linha nula...  **while**((str = br.readLine()) != **null**)  {System.***out***.println(str);}  /\*Não há necessidade de fechar os files mais  \*internos, quando temos os mais externos fechando  \*, pois estes já se encarregam de fechar os mais  \*internos ao mesmo tempo, se houver.\*/  br.close();  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  }  } |
| **Resultado:**  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.  Pulando uma linhaEscrevendo uma mensagem no arquivo  Escrevendo mais mensagem.  Pulando uma linha |

## D) FileWritter

É o mesmo que *FileReader* só que para escrita e não para leitura. Classe que serve para escrever caracteres em um arquivo.

|  |
| --- |
| **public** **class** FileWritterReaderTeste  { **public** **static** **void** main(String[] args)  { //criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if**(file.exists())  { **try**  { FileWriter fw = **new** FileWriter(file);  fw.write("Escrevendo uma mensagem no arquivo");  fw.flush();  //fecha o arquivo.  fw.close();  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  }  } |
|  |

Existe também um construtor uma opção booleana de fazer um append toda vez que quisermos inserir algo a mais do que já existe, ao invés de sobreescrever.

|  |
| --- |
| **public** **class** FileWritterReaderTeste  { **public** **static** **void** main(String[] args)  { //criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if**(file.exists())  { **try**  { FileWriter fw = **new** FileWriter(file, **true**);  fw.write("Escrevendo uma mensagem no arquivo"  + "\nEscrevendo mais mensagem.");  fw.flush();  //fecha o arquivo.  fw.close();  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  }  } |
|  |

## E) BufferedWritter

É o análogo de *BufferedReader* só que para escrita. Pegando o mesmo código usado em FileWriter para escrever, e o mesmo arquivo criado.

|  |
| --- |
| **public** **class** BufferedWRTeste  { **public** **static** **void** main(String[] args)  { //criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if**(file.exists())  { **try**  { FileWriter fw = **new** FileWriter(file, **true**);  BufferedWriter br = **new** BufferedWriter(fw);    br.write("Escrevendo uma mensagem no arquivo"  + "\nEscrevendo mais mensagem.");  /\*este método irá se encarregar de encontrar o  \*caractere especial que vai pular linha baseado  \*no sistema operacional.\*/  br.newLine();  br.write("Pulando uma linha");  br.flush();  //fecha o arquivo.  br.close();    //leitura do arquivo  FileReader fr = **new** FileReader(file);  **char**[] text = **new** **char**[500];  **int** size = fr.read(text);  **for**(**char** c:text) {System.***out***.print(c);}  fr.close();  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  }  } |
| **Resultado:**  Escrevendo uma mensagem no arquivoEscrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.  Pulando uma linha\_ |

Podemos ainda fazer melhor, usando o *try with resources*, escrevendo menos e aumentando muito a performance.

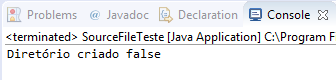
|  |
| --- |
| **public** **class** BufferedWRTeste  { **public** **static** **void** main(String[] args)  { // criando um arquivo  File file = **new** File("files\\arquivo.txt");  **if** (file.exists())  { **try** (BufferedWriter bw = **new** BufferedWriter(**new** FileWriter(file));  BufferedReader br = **new** BufferedReader(**new** FileReader(file));)  { bw.write("Escrevendo uma mensagem no arquivo" +  "\nEscrevendo mais mensagem.");  /\*  \* este método irá se encarregar de encontrar o caractere especial que vai pular  \* linha baseado no sistema operacional.  \*/  bw.newLine();  bw.write("Pulando uma linha");  bw.flush();  String str;  // enquanto não houver linha nula...  **while** ((str = br.readLine()) != **null**) {  System.***out***.println(str);  }  }  **catch** (IOException e) {System.***out***.println(e.getMessage());}  /\*  \* try { FileWriter fw = new FileWriter(file, true); BufferedWriter bw = new  \* BufferedWriter(fw);  \*  \* bw.write("Escrevendo uma mensagem no arquivo" +  \* "\nEscrevendo mais mensagem."); este método irá se encarregar de encontrar o  \* caractere especial que vai pular linha baseado no sistema operacional.  \* bw.newLine(); bw.write("Pulando uma linha"); bw.flush(); //fecha o arquivo.  \* bw.close();  \*  \* //leitura do arquivo FileReader fr = new FileReader(file);  \*  \* //Buffered Reader muda um pouco as coisas BufferedReader br = new  \* BufferedReader(fr); //este método, ao invés de ler caractere por //caractere,  \* este faz leitura linha a linha br.readLine();  \*  \* String str = null; //enquanto não houver linha nula... while((str =  \* br.readLine()) != null) {System.out.println(str);} Não há necessidade de  \* fechar os files mais internos, quando temos os mais externos fechando , pois  \* estes já se encarregam de fechar os mais internos ao mesmo tempo, se houver.  \* br.close(); } catch (IOException e) {System.out.println(e.getMessage());}  \*/  }  **else**  { **try** {file.createNewFile();}  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  }  } |
| **Resultado:**  Escrevendo uma mensagem no arquivo  Escrevendo mais mensagem.  Pulando uma linha |

# Classe File para diretórios

Até agora só trabalhamos com arquivos, mas também podemos trabalhar com diretórios na classe File.

|  |
| --- |
| **public** **class** SourceFileTeste  { **public** **static** **void** main(String[] args)  { File file = **new** File("folder");  //para quem estiver acostumado com linux  **boolean** mkdir = file.mkdir();  System.***out***.println("Diretório criado " + mkdir);  }  } |
|  |

Caso um diretório já existir, o mesmo não será criado. Então se executarmos novamente essa tarefa, veremos que nossa booleana retornará *false*.



Para criarmos um arquivo dentro deste novo diretório:

|  |
| --- |
| **public** **class** SourceFileTeste  { **public** **static** **void** main(String[] args)  { File file = **new** File("folder");  //para quem estiver acostumado com linux  **boolean** mkdir = file.mkdir();  System.***out***.println("Diretório criado " + mkdir);  File file2 = **new** File("folder\\arquivo.txt");  **try**  { file2.createNewFile();  System.***out***.println("Arquivo criado " +  file2.getName());  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **public** **class** SourceFileTeste  { **public** **static** **void** main(String[] args)  { File file = **new** File("folder");  //para quem estiver acostumado com linux  **boolean** mkdir = file.mkdir();  System.***out***.println("Diretório criado " + mkdir);  File file2 = **new** File(file, "arquivo.txt");  **try**  { file2.createNewFile();  System.***out***.println("Arquivo criado " +  file2.getName());  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
|  |

É possível também renomarmos nossos arquivos.

|  |
| --- |
| **public** **class** SourceFileTeste  { **public** **static** **void** main(String[] args)  { File file = **new** File("folder");  //para quem estiver acostumado com linux  **boolean** mkdir = file.mkdir();  System.***out***.println("Diretório criado " + mkdir);  File file2 = **new** File("folder\\arquivo.txt");  **try**  { file2.createNewFile();  System.***out***.println("Arquivo criado " +  file2.getName());  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}    //renomeando arquivo existente, sempre devemos também incluir  //a sua localização.  File renamedFile = **new** File("folder\\novoArquivo.txt");  System.***out***.println("Arquivo renomeado: " +  file2.renameTo(renamedFile));  }  } |
|  |

Para finalizar, podemos também buscar o nome do diretório do arquivo.

|  |
| --- |
| **public** **class** SourceFileTeste  { **public** **static** **void** main(String[] args)  {*findSourceFile*();}  **public** **static** **void** findSourceFile()  { File file = **new** File("folder");  //método que retorna os arquivos dentro  //do dietório  **for**(String fileName:file.list())  {System.***out***.println(fileName);}  }  } |
|  |

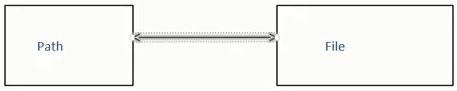
# Classe Console

Essa classe nunca será usado para aplicações de projeto, mas ela foi inserida no java 6 e a única utilidade é para conseguir maior chance de passar na prova de certificação. Um console é nada mais nada a menos que um teclado com o qual podemos entrar com dados diretamente em programa.

# NIO

Tudo o que vimos agora sobre o pacote IO, é considerado como apenas o básico. Esse novo recurso de entrada e saída foi introduzido no java 7, NIO, nada mais conhecido como New IO. Existem 3 classes que devemos reparar muito e que podem acabar confundindo.

Deve se lembrar da classe *File*, que faz parte do pacote IO, para trabalhar tanto com diretório como com arquivos. Porém, iremos chamar aqui a classe *File* e a nova Interface, que faz parte do NIO, que veio para substituir esta classe, que é a *Path*. Isso nada mais é que uma conversão.



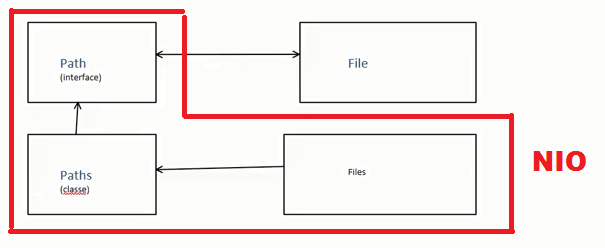
E para trabalhar com a classe *Path*, foram criadas mais duas classes:

## A) Paths:

Esta sim é uma classe concreta. Para criarmos um *Path* teremos que utilizar essa classe.

## B) Files

É uma classe que utiliza os recursos de Paths e faz, por exemplo, copiar ou deletar arquivos.



|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");  }  } |

Veremos agora que podemos usar um objeto da classe *Path* e transformá-lo em um tipo *File* e vice e versa.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();  }  } |

Com a classe *File*, aprendemos como criar um diretório, usando o mkdir. Com a *Path*, temos o seguinte.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();    Path path4 = Paths.*get*("diretorio");    **try** {Files.*createDirectory*(path4);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
|  |

Se caso esse diretório já existisse, daria um erro em tempo de execução. Portanto, para otimizar nosso código, podemos fazer a seguinte melhoria para evitar esse tipo de erro.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();    Path path4 = Paths.*get*("diretorio");    **try**  { **if**(Files.*notExists*(path4))  {Files.*createDirectory*(path4);}    **else** {System.***out***.println("Diretório já existente");}  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |

Agora, a partir de *Path*, criar um arquivo.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();    Path path4 = Paths.*get*("diretorio");    Path path5 = Paths.*get*("diretorio\\arquivo.txt");    **try**  { /\*if(Files.notExists(path4))  {Files.createDirectory(path4);}\*/    **if**(Files.*notExists*(path5))  {Files.*createFile*(path5);}    **else** {System.***out***.println("Arquivo já existente");}  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
|  |

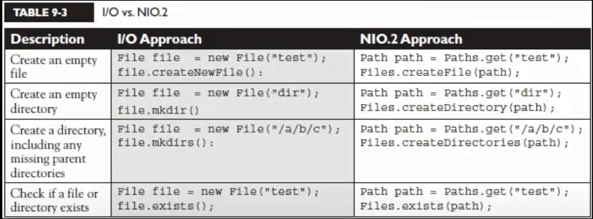
Se tentássemos chamar o método *createDirectory()* para criar o “arquivo.txt”, isso iria criar um diretório. Para isso, podemos usar o método *getParent()*, e assim será criado apenas o diretório.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();    Path path4 = Paths.*get*("diretorio");    Path path5 = Paths.*get*("diretorio\\arquivo.txt");    **try**  { /\*if(Files.notExists(path4))  {Files.createDirectory(path4);}\*/    /\*if(Files.notExists(path5))  {Files.createFile(path5);}\*/    **if**(Files.*notExists*(path5.getParent()))  {Files.*createDirectory*(path5.getParent());}    **else** {System.***out***.println("Diretório já existente");}  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |

Outras manipulações interessantes que podemos fazer com essas classes.

|  |
| --- |
| **public** **class** CriandoPathTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\arquivo.txt");  //outra forma de separar cada local e recuperar  //os arquivos  Path path2 = Paths.*get*("folder", "novoArquivo.txt");    //Transformando um tipo Path em um tipo File  File file = path2.toFile();    //Processo inverso  Path path3 = file.toPath();    Path path4 = Paths.*get*("diretorio");    Path path5 = Paths.*get*("diretorio\\arquivo.txt");    **try**  { /\*if(Files.notExists(path4))  {Files.createDirectory(path4);}\*/    /\*if(Files.notExists(path5))  {Files.createFile(path5);}\*/    **if**(Files.*notExists*(path5.getParent()))  {Files.*createDirectory*(path5.getParent());}    **else** {System.***out***.println("Diretório já existente");}  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}    **try** //copia um arquivo existente de um diretório para outro  { Files.*copy*(path, path4,  StandardCopyOption.***REPLACE\_EXISTING***);  //deleta arquivo caso exista.  Files.*deleteIfExists*(path2);  }  **catch**(IOException e2)  {System.***out***.println(e2.getMessage());}  }  } |
|  |

Um resumo dessa aula é:



# Normalizando Paths

Quando temos vários diretórios e subdiretórios, ao invés de localizar um diretório ou arquivo de acordo com o *path* absoluto, podemos simplesmente usar “..\”.

|  |
| --- |
| **public** **class** NormalizacaoTeste  { **public** **static** **void** main(String[] args)  { Path p1 = Paths.*get*("files\\..\\teste.txt");  **if**(Files.*notExists*(p1.getParent()))  { **try** {Files.*createDirectories*(p1.getParent());}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  **if**(Files.*notExists*(p1))  { **try** {Files.*createFile*(p1);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  System.***out***.println(p1);  System.***out***.println(p1.normalize());  }  } |

# Resolvendo paths

Pode ser que em determinado momento tenhamos dois paths, e desejamos agora juntá-los, isso significa resolver path. Imagine a seguinte situação:

|  |
| --- |
| **public** **class** ResolvendoPaths  { **public** **static** **void** main(String[] args)  { Path dir = Paths.*get*("home\\willian");  Path file = Paths.*get*("dev\\arquito.txt");  Path result = dir.resolve(file);  }  } |
| **Resultado:**  home\willian\dev\arquito.txt |

Problema disso é quando temos caminhos absolutos e relativos.

|  |
| --- |
| **public** **class** ResolvendoPaths  { **public** **static** **void** main(String[] args)  { Path dir = Paths.*get*("home\\willian");  Path file = Paths.*get*("dev\\arquito.txt");  Path result = dir.resolve(file);  System.***out***.println(result);    Path absoluto = Paths.*get*("/home/willian");  Path relativo = Paths.*get*("dev");  Path file2 = Paths.*get*("file.txr");    System.***out***.println("1: " + absoluto.resolve(relativo));  System.***out***.println("2: " + absoluto.resolve(file));  System.***out***.println("3: " + relativo.resolve(file));  System.***out***.println("4: " + relativo.resolve(absoluto));  System.***out***.println("5: " + file.resolve(absoluto));  System.***out***.println("6: " + file.resolve(relativo));  }  } |
| **Resultado**  home\willian\dev\arquito.txt  1: \home\willian\dev  2: \home\willian\dev\arquito.txt  3: dev\dev\arquito.txt  4: \home\willian  5: \home\willian  6: dev\arquito.txt\dev |

Perceba que nas situações 4 e 5 temos dois caminhos absolutos, sendo que estamos pegando um caminho relativo e tentando resolver com caminho absoluto. Estamos tentando conectar esse caminho relativo com o caminho absoluto, mas o caminho absoluto, como o nome já diz, é absoluto e não precisa de nada antes dele e ele mesmo se resolve por isso que em ambas as situações foi repetido.

Toda vez que formos tentar utilizar um caminho relativo ou um *file*, com o absoluto teremos de volta aquele caminho absoluto, ao contrário, funcionará normalmente. No último índice, temos um *file* que tenta resolver com um caminho relativo, perceba que foi feito uma bagunça, pois existe um diretório dentro de um arquivo, o que não é valido em nenhuma hipótese.

# Relativizando Paths

Relativizar paths significa, quando temos dois paths e queremos eliminar o que existe em comum entre aqueles paths.

|  |
| --- |
| **public** **class** RelativizarTeste  { **public** **static** **void** main(String[] args)  { Path dir = Paths.*get*("/home/willian");  Path classe = Paths.*get*("/home/willian/java/Pessoa.java");    //partindo do objeto "dir", como chegar em "classe".  Path dirToClasse = dir.relativize(classe);    System.***out***.println(dirToClasse);  }  } |
| **Resultado:**  java\Pessoa.java |

Veja que o método *relativize()* realiza o processo contrário do *resolve()* da aula anterior, pois este irá eliminar os subdiretórios em comuns, enquanto que o anterior junta. Porém, mostraremos que nem tudo são flores.

|  |
| --- |
| **public** **class** RelativizarTeste  { **public** **static** **void** main(String[] args)  { Path dir = Paths.*get*("/home/willian");  Path classe = Paths.*get*("/home/willian/java/Pessoa.java");    //partindo do objeto "dir", como chegar em "classe".  Path dirToClasse = dir.relativize(classe);    System.***out***.println(dirToClasse);    //paths absolutos  Path abs1 = Paths.*get*("/home/willian");  Path abs2 = Paths.*get*("/usr/local");  Path abs3 = Paths.*get*("/home/willian/java/Pessoa.java");    //paths relativos  Path rel1 = Paths.*get*("temp");  Path rel2 = Paths.*get*("temp/Funcionario.java");    System.***out***.println("1: " + abs1.relativize(abs3));  System.***out***.println("2: " + abs3.relativize(abs1));  System.***out***.println("3: " + abs1.relativize(abs2));  System.***out***.println("4: " + rel1.relativize(rel2));  }  } |
| **Resultado**  java\Pessoa.java  1: java\Pessoa.java  2: ..\..  3: ..\..\usr\local  4: Funcionario.java |

**OBS: sempre teremos erro em tempo de execução quando tentarmos fazer de um absoluto para um relativo, pois o java simplesmente não consegue encontrar onde tal diretório relativo está localizado, pois um relativo sempre pode estar dentro de qualquer outra pasta.**

# BasicFileAttributes e BasicFileAttributeView

São atributos básicos dentro do pacote NIO. Até agora estávamos apenas lendo, por exemplo, datas e agora iremos modificar a data de criação em que o arquivo foi modificado pela última vez.

|  |
| --- |
| **public** **class** BasicAttributesTest  { **public** **static** **void** main(String[] args)  { Date primeiroDezembro = **new** GregorianCalendar(2015,  Calendar.***DECEMBER***, 1).getTime();    File file = **new** File("files\\arquivo2.txt");  **try**  { **if**(!file.exists() && file.createNewFile())  { file.setLastModified(primeiroDezembro.getTime());  System.***out***.println(file.lastModified());  file.delete();    //usando Path  Path path = Paths.*get*("folder\\file.txt");    Files.*createFile*(path);  FileTime fileTime = FileTime.*fromMillis*(  primeiroDezembro.getTime());    Files.*setLastModifiedTime*(path, fileTime);  System.***out***.println(Files.*getLastModifiedTime*(path));  Files.*deleteIfExists*(path);  }  }  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  1448938800000  2015-12-01T03:00:00Z |

Agora iremos usar os atributos básicos.

|  |
| --- |
| **public** **class** BasicAttributesTest  { **public** **static** **void** main(String[] args)  { Date primeiroDezembro = **new** GregorianCalendar(2015,  Calendar.***DECEMBER***, 1).getTime();    File file = **new** File("files\\arquivo2.txt");  **try**  { **if**(!file.exists() && file.createNewFile())  { file.setLastModified(primeiroDezembro.getTime());  System.***out***.println(file.lastModified());  file.delete();    //usando Path  Path path = Paths.*get*("folder\\file.txt");    Files.*createFile*(path);  FileTime fileTime = FileTime.*fromMillis*(  primeiroDezembro.getTime());    Files.*setLastModifiedTime*(path, fileTime);  System.***out***.println(Files.*getLastModifiedTime*(path));  Files.*deleteIfExists*(path);    path = Paths.*get*("F:\\Users\\eduardowmu\\Desktop\\"  + "meusdoc\\estudo\\FATEC\\7 - OUTROS SEMESTRES\\"  + "YOUTUBE\\Maratona java\\Repositorio\\JavaKnowledges\\"  + "MaratonaJava\\src\\br\\edu\\maratona\\CarroTeste.java");    System.***out***.println(Files.*isReadable*(path));  System.***out***.println(Files.*isWritable*(path));  System.***out***.println(Files.*isExecutable*(path));  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  1448938800000  2015-12-01T03:00:00Z  true  true  true |

Os atributos basicos de arquivos são compostos por 3 interfaces, que servem para conseguir ler os atributos dos arquivos ou dos diretórios:

## A) BasicFileAttributes

Utilizado para qualquer sistema operacional.

## B) PosixFileAttributes

Para sistemas UNIX.

## C) DosFileAttributes

Sistemas voltados para Windows.

|  |
| --- |
| **public** **class** BasicAttributesTest  { **public** **static** **void** main(String[] args)  { Date primeiroDezembro = **new** GregorianCalendar(2015,  Calendar.***DECEMBER***, 1).getTime();    File file = **new** File("files\\arquivo2.txt");  **try**  { **if**(!file.exists() && file.createNewFile())  { file.setLastModified(primeiroDezembro.getTime());  System.***out***.println(file.lastModified());  file.delete();    //usando Path  Path path = Paths.*get*("folder\\file.txt");    Files.*createFile*(path);  FileTime fileTime = FileTime.*fromMillis*(  primeiroDezembro.getTime());    Files.*setLastModifiedTime*(path, fileTime);  System.***out***.println(Files.*getLastModifiedTime*(path));  Files.*deleteIfExists*(path);    path = Paths.*get*("F:\\Users\\eduardowmu\\Desktop\\"  + "meusdoc\\estudo\\FATEC\\7 - OUTROS SEMESTRES\\"  + "YOUTUBE\\Maratona java\\Repositorio\\JavaKnowledges\\"  + "MaratonaJava\\src\\br\\edu\\maratona\\CarroTeste.java");    System.***out***.println(Files.*isReadable*(path));  System.***out***.println(Files.*isWritable*(path));  System.***out***.println(Files.*isExecutable*(path));    //BasicFilesAttributes, PosixFileAttributes, DosFileAttributes  BasicFileAttributes basicFiles = Files.*readAttributes*(path,  BasicFileAttributes.**class**);    System.***out***.println(basicFiles.creationTime());  System.***out***.println(basicFiles.lastAccessTime());  System.***out***.println(basicFiles.lastModifiedTime());  System.***out***.println(basicFiles.isDirectory());  System.***out***.println(basicFiles.isSymbolicLink());  System.***out***.println(basicFiles.isRegularFile());  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  1448938800000  2015-12-01T03:00:00Z  true  true  true  2020-12-20T17:55:31.859496Z  2020-12-20T17:55:31.859496Z  2020-12-20T19:25:23.659889Z  false  false  true |

Perceba que a data de criação e a ultima modificação foram as mesmas. Se caso quiséssemos alterar uma coisa desses atributos. Aí que entra outras interfaces.

## D) BasicFileAttributeView, PosyxFileAttributeView e DosFileAttributeView

Todos são para modificações, análogos aos respectivos itens A, B e C.

## E) FileOwnerAttributeView

Usado para ajustar o principal arquivo ou diretório.

## F) AclFileAttributeView

Possui permissões mais avançadas para trabalhar com diretórios ou arquivos.

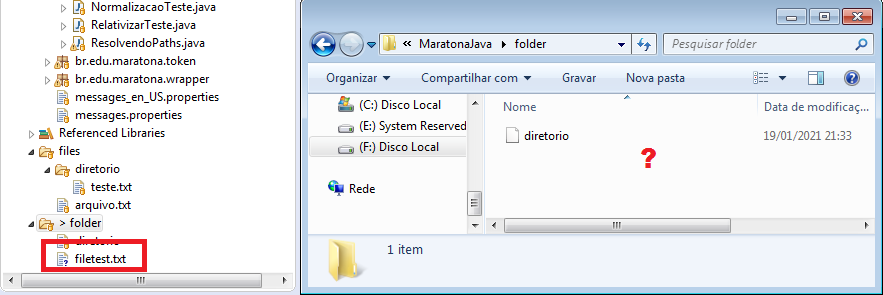
|  |
| --- |
| **public** **class** BasicAttributesTest  { **public** **static** **void** main(String[] args)  { Date primeiroDezembro = **new** GregorianCalendar(2015,  Calendar.***DECEMBER***, 1).getTime();    File file = **new** File("files\\arquivo2.txt");  **try**  { **if**(!file.exists() && file.createNewFile())  { file.setLastModified(primeiroDezembro.getTime());  System.***out***.println(file.lastModified());  file.delete();    //usando Path  Path path = Paths.*get*("folder\\file.txt");    Files.*createFile*(path);  FileTime fileTime = FileTime.*fromMillis*(  primeiroDezembro.getTime());    Files.*setLastModifiedTime*(path, fileTime);  System.***out***.println(Files.*getLastModifiedTime*(path));  Files.*deleteIfExists*(path);    path = Paths.*get*("F:\\Users\\eduardowmu\\Desktop\\"  + "meusdoc\\estudo\\FATEC\\7 - OUTROS SEMESTRES\\"  + "YOUTUBE\\Maratona java\\Repositorio\\JavaKnowledges\\"  + "MaratonaJava\\src\\br\\edu\\maratona\\CarroTeste.java");    System.***out***.println(Files.*isReadable*(path));  System.***out***.println(Files.*isWritable*(path));  System.***out***.println(Files.*isExecutable*(path));    //BasicFilesAttributes, PosixFileAttributes, DosFileAttributes  BasicFileAttributes basicFiles = Files.*readAttributes*(path,  BasicFileAttributes.**class**);    System.***out***.println(basicFiles.creationTime());  System.***out***.println(basicFiles.lastAccessTime());  System.***out***.println(basicFiles.lastModifiedTime());  System.***out***.println("------------------------------------");  /\*  System.out.println(basicFiles.isDirectory());  System.out.println(basicFiles.isSymbolicLink());  System.out.println(basicFiles.isRegularFile());  \*/    FileTime lastModified = basicFiles.lastModifiedTime();  FileTime created = basicFiles.creationTime();  FileTime lastAccess = FileTime.*fromMillis*(System.*currentTimeMillis*());    //Para modificação:  //BasicAttributeView, PosyxFileAttribureView, DosFileAttributesView  //FileOwnerAttributeView, AclFileAttribureView  BasicFileAttributeView basicFileView = Files.*getFileAttributeView*(  path, BasicFileAttributeView.**class**);    //alterando time do path  basicFileView.setTimes(lastModified, lastAccess, created);  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
|  |

# DosFileAttributes e DosFileAttributeView

A interface *DosFileAttributes* é bem simples pois só adiciona 4 novos atributos com relação aos básicos. Se vermos, temos a interface *BasicFileAttributes* e as interfaces *DosFileAttributes* e *PosixFileAttributes* são interfaces que herdam da primeira, que esta nada mais é que os atributos básicos de qualquer sistema operacional, relacionados a arquivos e diretórios.

|  |
| --- |
| **public** **class** DosFileAttributeTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("folder\\filetest.txt");  **try**  { **if**(!Files.*exists*(path))  { Files.*createFile*(path);  //setando atributos  Files.*setAttribute*(path, "dos:hidden", **true**);  //lembrando que em muitas partes o java é  //case sensitive, ou seja, tem diferenças com  //letras maiusculas e minusculas  Files.*setAttribute*(path, "dos:readonly", **true**);  DosFileAttributes dosFileAtt = Files.*readAttributes*(  //embora seja uma interface, devemos encará-la  //como uma classe  path, DosFileAttributes.**class**);  System.***out***.println(dosFileAtt.isHidden());  System.***out***.println(dosFileAtt.isReadOnly());  }  }  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  true  true |

Perceba que acabamos criando um arquivo oculto.



# PosixFileAttributes e PosixFileAttributesView

Ultimas interfaces dos atributos *File* e para trabalhar com essas interfaces teríamos que trabalhar com sistemas UNIX/LINUX e suas distribuições. Veja que no código abaixo tivemos uma exceção devido ao sistema operacional onde foi executado ser Windows.

|  |
| --- |
| **public** **class** PosixFileAttTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files\\diretorio\\teste.txt");  **try**  { PosixFileAttributes pfa = Files.*readAttributes*(path,  PosixFileAttributes.**class**);  System.***out***.println(pfa.permissions());  }  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |
| Exception in thread "main" java.lang.UnsupportedOperationException  at sun.nio.fs.WindowsFileSystemProvider.readAttributes(Unknown Source)  at java.nio.file.Files.readAttributes(Unknown Source)  at br.edu.maratona.nio.PosixFileAttTeste.main(PosixFileAttTeste.java:13) |

# DirectoryStream e SimpleFileVisitor

Nesta aula iremos falar sobre como buscar arquivos em diretórios.

## A) DirectoryStream

Uma das classes mais simples para se buscar arquivos e determinado pacote em determinado diretório. Mas esta funciona um pouco limitado, pois funciona como se fosse um “ls” que damos no Windows ou um “dir”, em que irão buscar apenas os arquivos de determinada pasta.

|  |
| --- |
| **public** **class** DirectoryStreamTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files");    **try**  { DirectoryStream<Path> stream =  Files.*newDirectoryStream*(path);    **for**(Path p:stream)  {System.***out***.println(p.getFileName());}  }  **catch**(IOException ioe){}  }  } |
| **Resultado:**  arquivo.txt  diretorio |

## B) FileVisitor

Aqui as coisas complicam um pouco pois teremos que usar REGEX. Através disso, iremos percorrer diretórios, mas para isso iremos criar uma outra classe e chamaremos de FindAllTxt.

|  |
| --- |
| **public** **class** FileVisitorTeste  { **public** **static** **void** main(String[] args) **throws** IOException  {Files.*walkFileTree*(Paths.*get*("files"), **new** FindAllTxt());}  }  **class** FindAllTxt **extends** SimpleFileVisitor<Path>  { @Override **public** FileVisitResult visitFile(Path file,  BasicFileAttributes attrs)  { **if**(file.getFileName().toString().endsWith(".txt"))  {System.***out***.println(file.getFileName());}    **return** FileVisitResult.***CONTINUE***;  }  } |
| **Resultado:**  arquivo.txt  teste.txt |

Vejamos cada um dos outros métodos da classe *SimpleFileVisitor*.

|  |
| --- |
| **public** **class** FileVisitorTeste  { **public** **static** **void** main(String[] args) **throws** IOException  {Files.*walkFileTree*(Paths.*get*("files"), **new** PrintPaths());}  }  **class** FindAllTxt **extends** SimpleFileVisitor<Path>  { @Override **public** FileVisitResult visitFile(Path file,  BasicFileAttributes attrs)  { **if**(file.getFileName().toString().endsWith(".txt"))  {System.***out***.println(file.getFileName());}    **return** FileVisitResult.***CONTINUE***;  }  }  **class** PrintPaths **extends** SimpleFileVisitor<Path>  { **public** FileVisitResult preVisitDirectory(Path path,  BasicFileAttributes attrs)  { System.***out***.println("pre " + path);  **return** FileVisitResult.***CONTINUE***;  }    @Override **public** FileVisitResult visitFile(Path path,  BasicFileAttributes attrs)  { System.***out***.println(path.getFileName());    **return** FileVisitResult.***CONTINUE***;  }    @Override **public** FileVisitResult visitFileFailed(Path path,  IOException exc)  {**return** FileVisitResult.***CONTINUE***;}    **public** FileVisitResult postVisitFileDirectory(Path path,  IOException ioe)  { System.***out***.println(path);    **return** FileVisitResult.***CONTINUE***;  }  } |
| **Resultado**  pre files  arquivo.txt  pre files\diretorio  teste.txt |

Perceba que o percurso foi do ultimo arquivo para o primeiro diretório.

# PathMatcher

Esta classe foi criada justamente para termos mais opções e poder de fogo para na hora de buscar por arquivos. Então será aliada ao SingleFileVisitor.

|  |
| --- |
| **public** **class** PathMactherTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files/diretorio");  **try** {Files.*walkFileTree*(path, **new** FindAllTxt());}  **catch** (IOException e) {System.***out***.println(e.getMessage());}  }  } |
| **Resultado:**  teste.txt |

O código acima foi só um código demonstrativo como com classes básicas como Path e Files podemos encontrar os arquivos ou diretórios. Mas com a classe *PathMatcher*, podemos usar REGEX e o padrão *glob*.

|  |
| --- |
| **public** **class** PathMactherTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files/diretorio/teste.txt");  Path path2 = Paths.*get*("arquivo.txt");    PathMatcher matcher = FileSystems.*getDefault*()  //aqui começamos com REGEX e sempre iniciamos  //com um padrão glob  .getPathMatcher("glob:\*.txt");  System.***out***.println(matcher.matches(path));  System.***out***.println(matcher.matches(path2));  }  } |
| **Resultado:**  false  true |

O “\*” serve para encontrar qualquer coisa, menos as “/”, e por isso precisamos usar REGEX nesses casos. Para encontrar diretórios, podemos usar \*\* para encontrar qualquer coisa seguido da extensão que queremos, ou seja, no caso, o txt.

|  |
| --- |
| **public** **class** PathMactherTeste  { **public** **static** **void** main(String[] args)  { Path path = Paths.*get*("files/diretorio/teste.txt");  Path path2 = Paths.*get*("arquivo.txt");    PathMatcher matcher = FileSystems.*getDefault*()  //aqui começamos com REGEX e sempre iniciamos  //com um padrão glob  .getPathMatcher("glob:\*.txt");  System.***out***.println(matcher.matches(path));  System.***out***.println(matcher.matches(path2));  *matches*(path, "glob:\*.txt");  //para incluir os diretórios são \*\*  *matches*(path, "glob:\*\*/\*.txt");  }  **private** **static** **void** matches(Path path, String glob)  { PathMatcher matcher = FileSystems.*getDefault*().getPathMatcher(glob);  System.***out***.println(glob + ": " + matcher.matches(path));  }  } |
| **Resultado:**  false  true  glob:\*.txt: false  glob:\*\*/\*.txt: true |

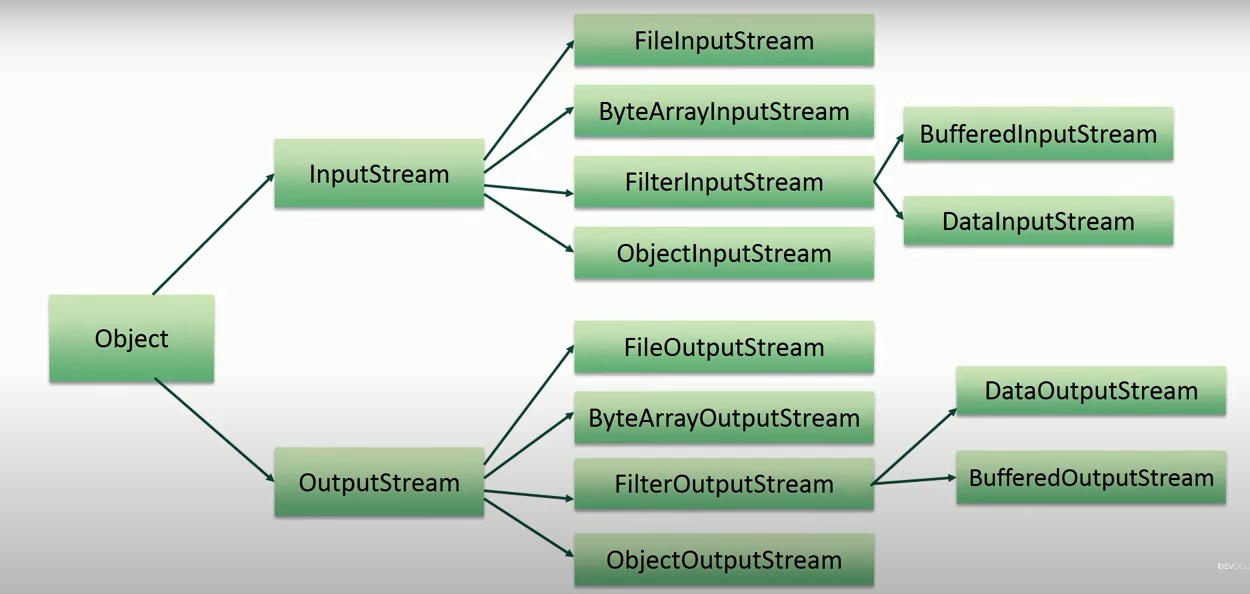
Veja que o objeto *path* teve seu arquivo txt encontrado usando \*\*/\*. Vamos supor agora que queiramos encontrar todos os arquivos “.java”?

|  |
| --- |
| **class** FindAllTest **extends** SimpleFileVisitor<Path>  { **private** PathMatcher matcher = FileSystems.*getDefault*()  .getPathMatcher("glob:\*\*/\*{Teste\*}.{java,class}");  @Override **public** FileVisitResult visitFile(Path file,  BasicFileAttributes attrs) **throws** IOException  { **if**(matcher.matches(file))  {System.***out***.println(file.getFileName());}    **return** FileVisitResult.***CONTINUE***;  }  }  **public** **class** PathMactherTeste  { **public** **static** **void** main(String[] args) **throws** IOException  { Path path = Paths.*get*("files/diretorio/teste.txt");  Path path2 = Paths.*get*("arquivo.txt");    PathMatcher matcher = FileSystems.*getDefault*()  //aqui começamos com REGEX e sempre iniciamos  //com um padrão glob  .getPathMatcher("glob:\*.txt");  System.***out***.println(matcher.matches(path));  System.***out***.println(matcher.matches(path2));  *matches*(path, "glob:\*.txt");  //para incluir os diretórios são \*\*  *matches*(path, "glob:\*\*/\*.txt");  Files.*walkFileTree*(Paths.*get*("./"), **new** FindAllTest());  }  **private** **static** **void** matches(Path path, String glob)  { PathMatcher matcher = FileSystems.*getDefault*().getPathMatcher(glob);  System.***out***.println(glob + ": " + matcher.matches(path));  }  } |
| **Resultado:**  false  true  glob:\*.txt: false  glob:\*\*/\*.txt: true  CarroTeste.class  ClienteTeste.class  ConsoleTeste.class  DateFormatTeste.class  ErrorTeste.class  BufferedWRTeste.class  FileWritterReaderTeste.class  SourceFileTeste.class  LocaleTeste.class  ManipulacaoDatasTeste.class  BasicAttributesTest.class  CriandoPathTeste.class  DirectoryStreamTeste.class  DosFileAttributeTeste.class  FileVisitorTeste.class  NormalizacaoTeste.class  PathMactherTeste.class  PosixFileAttTeste.class  RelativizarTeste.class  RegexTeste.class  SimpleDateFormatTeste.class  CarroTeste.java  ClienteTeste.java  ConsoleTeste.java  DateFormatTeste.java  ErrorTeste.java  BufferedWRTeste.java  FileWritterReaderTeste.java  SourceFileTeste.java  LocaleTeste.java  ManipulacaoDatasTeste.java  BasicAttributesTest.java  CriandoPathTeste.java  DirectoryStreamTeste.java  DosFileAttributeTeste.java  FileVisitorTeste.java  NormalizacaoTeste.java  PathMactherTeste.java  PosixFileAttTeste.java  RelativizarTeste.java  RegexTeste.java  SimpleDateFormatTeste.java |

# InputStream e OutPutStream

São classes de streaming de bytes muito úteis para trabalhar em baixo nível com os arquivos. Nada mais é que um fluxo de dados. Quando damos por exemplo um play no vídeo do youtube e damos um play em uma música no spotfy, um fluxo constante de dados de bytes está vindo do seu computador ou do seu celular e o reprodutor ele vai executando esse fluxo, então isso é o que chamamos de streaming. Por isso que quando a conexão está lenta sentimos aquelas travadas e o fluxo de dados está sendo interrompido.

É a mesma coisa quando fazemos um download, a diferença é que enquanto não finalizarmos completamente o fluxo de dados, não conseguimos executar o arquivo. Já para player de vídeos e música conseguimos executar conforme ele vai recebendo.



## A) InputStream:

Quando estamos fazendo a leitura de bytes.

|  |
| --- |
| **public** **class** InOutPutStreamTeste  { **public** **static** **void** main(String[] args)  { *save*();  *read*();  }  **private** **static** **void** save()  { **byte**[] dados = {65, 66, 67, 68, 69};  **try**(FileOutputStream gravador =  **new** FileOutputStream("files/arquivo.txt"))  { **for**(Byte b:dados) {gravador.write(b);}  System.***out***.println("Dados salvo com sucesso");  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }    /\*Para leitura de dados, o código é um pouco  \*maior, pois para fazer leitura fazemos a leitura  \*byte a byte e quando chegar no final do arquivo  \*vai nos retornar -1\*/  **private** **static** **void** read()  { **try**(FileInputStream leitor =  **new** FileInputStream("files/arquivo.txt"))  { /\*variavel para saber quando não há mais  byte para leitura\*/  **int** leitura;  **while**((leitura = leitor.read()) != -1)  { **byte** b = (**byte**)leitura;  System.***out***.print(" "+b);  }  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }  } |
| Dados salvo com sucesso  65 66 67 68 69 |

## B) OutPutStream

Quando estamos fazendo escrita de bytes.

|  |
| --- |
| **public** **class** InOutPutStreamTeste  { **public** **static** **void** main(String[] args)  {*save*();}  **private** **static** **void** save()  { **byte**[] dados = {65, 66, 67, 68, 69};  **try**(FileOutputStream gravador =  **new** FileOutputStream("files/arquivo.txt"))  { **for**(Byte b:dados) {gravador.write(b);}  System.***out***.println("Dados salvo com sucesso");  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }  } |
| **Resultado**  Dados salvo com sucesso |
| **arquivo.txt**  ABCDE |

## C) Buffer

Para trabalhar de uma forma mais otimizada.

|  |
| --- |
| **public** **class** InOutPutStreamTeste  { **public** **static** **void** main(String[] args)  { //save();  *bufferedSave*();  //read();  readBuffered();  }  **private** **static** **void** save()  { **byte**[] dados = {65, 66, 67, 68, 69};  **try**(FileOutputStream gravador =  **new** FileOutputStream("files/arquivo.txt"))  { **for**(Byte b:dados) {gravador.write(b);}  System.***out***.println("Dados salvo com sucesso");  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }    **private** **static** **void** bufferedSave()  { **byte**[] dados = {65, 66, 67, 68, 69};  /\*O buffer tem por padrão 8142 bytes\*/  **try**(BufferedOutputStream gravadorBuffer =  **new** BufferedOutputStream(//limitação do numero de byte  **new** FileOutputStream("files/arquivo.txt"), 4098))  { **for**(Byte b:dados) {gravadorBuffer.write(b);}  /\*como estamos trabalhando com streaming de bytes,  \*existe a possibilidade de fecharmos aquele streaming  \*antes de pegarmos todos os dados que estavam passando  \*pelo tunel (buffer)\*/  gravadorBuffer.flush();  System.***out***.println("Dados salvo com sucesso");  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }    /\*Para leitura de dados, o código é um pouco  \*maior, pois para fazer leitura fazemos a leitura  \*byte a byte e quando chegar no final do arquivo  \*vai nos retornar -1\*/  **private** **static** **void** read()  { **try**(FileInputStream leitor =  **new** FileInputStream("files/arquivo.txt"))  { /\*variavel para saber quando não há mais  byte para leitura\*/  **int** leitura;  **while**((leitura = leitor.read()) != -1)  { **byte** b = (**byte**)leitura;  System.***out***.print(" "+b);  }  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }  **private** **static** **void** readBuffered()  { **try**(BufferedInputStream leitorBuffer = **new**  BufferedInputStream(**new**  FileInputStream("files/arquivo.txt"), 4098))  { /\*variavel para saber quando não há mais  byte para leitura\*/  **int** leitura;  **while**((leitura = leitorBuffer.read()) != -1)  { **byte** b = (**byte**)leitura;  System.***out***.print(" "+b);  }  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }  } |
| **Resultado**  Dados salvo com sucesso  65 66 67 68 69 |

# Zipando arquivos com java

Nesta aula iremos aprender como zipar todos os arquivos que estiverem nos diretórios que desejarmos. Para isso iremos misturar tanto usar classes do tipo IO como NIO. Existe um meio de fazermos de criar um arquivo zip e fazer compressão de arquivos. Utilizando o novo pacote NIO, mas sinceramente é mais complicado do que o IO, um dos motivos é por que o NIO utiliza mais memória.

Temos assim 3 incónitas: a primeira é onde iremos salvar nosso zip, onde está nosso arquivo e nosso zip em si.

|  |
| --- |
| **public** **class** ZipingTeste  { **public** **static** **void** main(String[] args)  { //definindo diretório do ZIP  Path zipPath = Paths.*get*("F:\\Users"  + "\\eduardowmu\\Desktop\\meusdoc"  + "\\estudo\\FATEC\\7 - OUTROS SEMESTRES"  + "\\YOUTUBE\\Maratona java\\Repositorio"  + "\\JavaKnowledges");    //criando diretório  Path filePath = Paths.*get*("F:\\Users\\eduardowmu"  + "\\Desktop\\meusdoc\\estudo\\FATEC"  + "\\7 - OUTROS SEMESTRES\\YOUTUBE\\Maratona java"  + "\\Repositorio");    //definindo nome do arquivo zip  Path zipFile = zipPath.resolve("file.zip");    /\*usando a classe ZipOutPutStream para  \*compimir arquivos\*/  **try**(ZipOutputStream zip = **new** ZipOutputStream(  **new** FileOutputStream(zipFile.toFile()));  DirectoryStream<Path> stream =  Files.*newDirectoryStream*(filePath))  { **for**(Path path:stream)  {System.***out***.println(path);}  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }  } |
| **Resultado**  F:\Users\eduardowmu\Desktop\meusdoc\estudo\FATEC\7 - OUTROS SEMESTRES\YOUTUBE\Maratona java\Repositorio\JavaKnowledges  F:\Users\eduardowmu\Desktop\meusdoc\estudo\FATEC\7 - OUTROS SEMESTRES\YOUTUBE\Maratona java\Repositorio\teste.txt |

Agora iremos inserir arquivos dentro do nosso zip.

|  |
| --- |
| **public** **class** ZipingTeste  { **public** **static** **void** main(String[] args)  { //definindo diretório do ZIP  Path zipPath = Paths.*get*("F:\\Users"  + "\\eduardowmu\\Desktop\\meusdoc"  + "\\estudo\\FATEC\\7 - OUTROS SEMESTRES"  + "\\YOUTUBE\\Maratona java\\Repositorio"  + "\\JavaKnowledges");    //criando diretório  Path filePath = Paths.*get*("F:\\Users\\eduardowmu"  + "\\Desktop\\meusdoc\\estudo\\FATEC"  + "\\7 - OUTROS SEMESTRES\\YOUTUBE\\Maratona java"  + "\\Repositorio");    //definindo nome do arquivo zip  Path zipFile = zipPath.resolve("file.zip");    /\*usando a classe ZipOutPutStream para  \*compimir arquivos\*/  **try**(ZipOutputStream zip = **new** ZipOutputStream(  **new** FileOutputStream(zipFile.toFile()));  DirectoryStream<Path> stream = Files.*newDirectoryStream*(filePath))  { **for**(Path path:stream)  { /\*Inserindo entradas dentro desse zip\*/  ZipEntry zipEntry = **new** ZipEntry(path.getFileName().toString());  /\*inserindo arquivos ou pastas dentro  \*zip\*/  zip.putNextEntry(zipEntry);    /\*Leitura de arquivos\*/  FileInputStream fis = **new** FileInputStream(path.toFile());    **byte**[] b = **new** **byte**[2048];  **int** bytesRead;  **while**((bytesRead = fis.read(b)) > 0) {zip.write(b, 0, bytesRead);}  zip.closeEntry();  fis.close();  }  }  **catch**(IOException ioe) {System.***out***.println(ioe.getMessage());}  }  } |
| **Resultado**  F:\Users\eduardowmu\Desktop\meusdoc\estudo\FATEC\7 - OUTROS SEMESTRES\YOUTUBE\Maratona java\Repositorio\JavaKnowledges (Acesso negado) |

Utilizando buffer agora para tornar mais otimizado e reduzido o processo e código.

|  |
| --- |
| **public** **class** ZipingTeste  { **public** **static** **void** main(String[] args)  { //definindo diretório do ZIP  Path zipPath = Paths.*get*("F:\\Users"  + "\\eduardowmu\\Desktop\\meusdoc"  + "\\estudo\\FATEC\\7 - OUTROS SEMESTRES"  + "\\YOUTUBE\\Maratona java\\Repositorio"  + "\\JavaKnowledges");    //criando diretório  Path filePath = Paths.*get*("F:\\Users\\eduardowmu"  + "\\Desktop\\meusdoc\\estudo\\FATEC"  + "\\7 - OUTROS SEMESTRES\\YOUTUBE\\Maratona java"  + "\\Repositorio");    //definindo nome do arquivo zip  Path zipFile = zipPath.resolve("file.zip");    /\*usando a classe ZipOutPutStream para  \*compimir arquivos\*/  **try**(ZipOutputStream zip = **new** ZipOutputStream(  **new** FileOutputStream(zipFile.toFile()));  DirectoryStream<Path> stream =  Files.*newDirectoryStream*(filePath))  { **for**(Path path:stream)  { /\*Inserindo entradas dentro desse zip\*/  ZipEntry zipEntry = **new** ZipEntry(path.getFileName().toString());  /\*inserindo arquivos ou pastas dentro  \*zip\*/  zip.putNextEntry(zipEntry);    /\*Leitura de arquivos\*/  FileInputStream fis = **new** FileInputStream(path.toFile());    /\*Trbalhando com Buffer\*/  BufferedInputStream bf = **new** BufferedInputStream(fis);    //byte[] b = new byte[2048];  //int bytesRead;    zip.write(bf.read());    /\*  while((bytesRead = fis.read(b)) > 0)  {zip.write(b, 0, bytesRead);}  \*/    zip.flush();  zip.closeEntry();  fis.close();  }  }  **catch**(IOException ioe) {System.***out***.println(ioe.getMessage());}  }  } |
| **Resultado**  O mesmo do anterior |

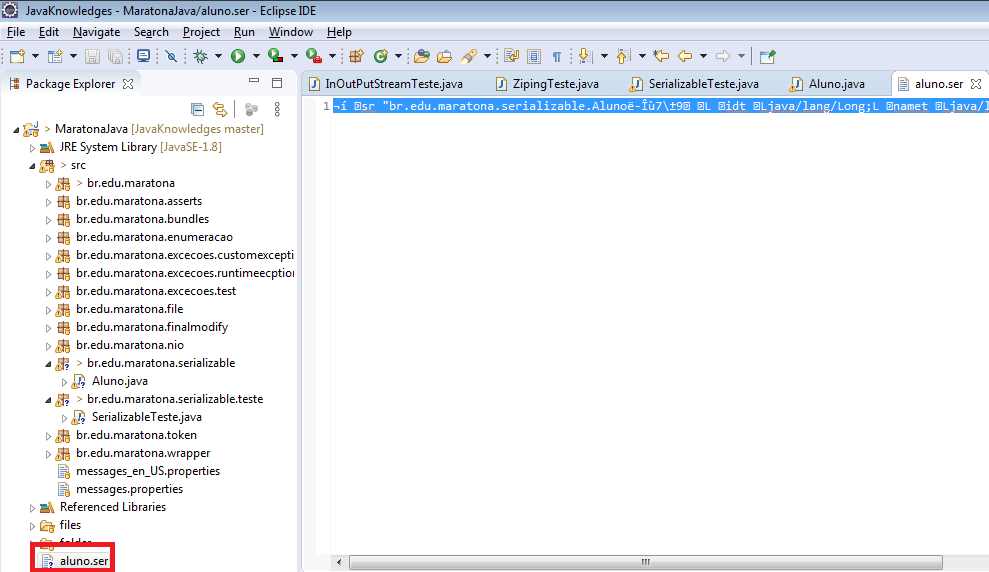
# Serialização

Não é algo que se use direto em projetos mas é utilizado em frameworks e APIs por todo o pacote do java. Serialização é nada mais e nada menos que pegar um objeto e fazer um fluxo, ou seja, passarmos o objeto por um canal de dados, que tem muito a ver com Streaming que estávamos fazendo duas aulas passadas. Então iremos fazer um Stream de objeto para um arquivo.

Iremos trabalhar então com duas classes: uma será o *OutputStream*, pois iremos fazer diretamente com arquivo, então iremos usar o *FileOutputStream*, e como iremos trabalhar com objetos, iremos usar a *ObjectOutpoutStream*.

|  |
| --- |
| **public** **class** SerializableTeste  { **public** **static** **void** main(String[] args)  {*gravadorObjeto*();}  **private** **static** **void** gravadorObjeto()  { Aluno aluno = **new** Aluno(1L, "Eduardo", "123");  **try**(ObjectOutputStream out = **new** ObjectOutputStream(  **new** FileOutputStream("aluno.ser")))  {out.writeObject(aluno);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  } |
| **public** **class** Aluno **implements** Serializable  { **private** Long id;  **private** String name;  **private** String password;  **private** **static** String *nomeEscola* = "DevDojo";    **public** Aluno(Long id, String name, String password)  { **this**.id = id;  **this**.name = name;  **this**.password = password;  }    **public** Long getId() {**return** id;}  **public** **void** setId(Long id) {**this**.id = id;}    **public** String getName() {**return** name;}  **public** **void** setName(String name) {**this**.name = name;}    **public** String getPassword() {**return** password;}  **public** **void** setPassword(String password)  {**this**.password = password;}    **public** **static** String getNomeEscola() {**return** *nomeEscola*;}  **public** **static** **void** setNomeEscola(String nomeEscola)  {Aluno.*nomeEscola* = nomeEscola;}    @Override  **public** String toString()  { **return** "Aluno ["  + "id=" + id  + ", name=" + name  + ", password=" + password  + ", nomeEscola="+ *nomeEscola*  +"]";  }  } |

O java não sabe se uma classe é serializado ou não. Para dizermos se uma classe pode ser serializada ou não, a única coisa que temos que fazer é usar a clausura *Serializable*. Esta é uma interface que chamamos de *flag*, pois ela serve apenas para dizer que a classe que a implementa e seus objetos serão serializáveis.



|  |
| --- |
| **public** **class** SerializableTeste  { **public** **static** **void** main(String[] args)  { //gravadorObjeto();  *leitorObjeto*();  }  **private** **static** **void** gravadorObjeto()  { Aluno aluno = **new** Aluno(1L, "Eduardo", "123");  **try**(ObjectOutputStream out = **new** ObjectOutputStream(  **new** FileOutputStream("aluno.ser")))  {out.writeObject(aluno);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }    **private** **static** **void** leitorObjeto()  { **try**(ObjectInputStream input = **new** ObjectInputStream(  **new** FileInputStream("aluno.ser")))  { Aluno aluno = (Aluno)input.readObject();  System.***out***.println(aluno);  }  **catch** (IOException | ClassNotFoundException e)  {System.***out***.println(e.getMessage());}  }  } |
| Aluno [id=1, name=Eduardo, password=123, nomeEscola=DevDojo] |

Quando usamos a serialização, o construtor não é chamado, e isso afeta diretamente a ordem que temos na construção dos objetos. Lembre-se, se tivéssemos aqui herança, o super não seria chamado. Perceba que estamos serializando um password, mas como podemos dizer para o Java ignorar um atributo, devemos falar que esse atributo é *transient*, ou seja, não se serializa esse atributo.

|  |
| --- |
| **public** **class** Aluno **implements** Serializable  { **private** Long id;  **private** String name;  **private** **transient** String password;  **private** **static** String *nomeEscola* = "DevDojo";    **public** Aluno(Long id, String name, String password)  { **this**.id = id;  **this**.name = name;  **this**.password = password;  }    **public** Long getId() {**return** id;}  **public** **void** setId(Long id) {**this**.id = id;}    **public** String getName() {**return** name;}  **public** **void** setName(String name) {**this**.name = name;}    **public** String getPassword() {**return** password;}  **public** **void** setPassword(String password)  {**this**.password = password;}    **public** **static** String getNomeEscola() {**return** *nomeEscola*;}  **public** **static** **void** setNomeEscola(String nomeEscola)  {Aluno.*nomeEscola* = nomeEscola;}    @Override **public** String toString()  { **return** "Aluno ["  + "id=" + id  + ", name=" + name  + ", password=" + password  + ", nomeEscola="+ *nomeEscola*  +"]";  }  } |

Com isso, o atributo considerado *transient* será ignorado.

|  |
| --- |
| **public** **class** SerializableTeste  { **public** **static** **void** main(String[] args)  { *gravadorObjeto*();  *leitorObjeto*();  }  **private** **static** **void** gravadorObjeto()  { Aluno aluno = **new** Aluno(1L, "Eduardo", "123");  **try**(ObjectOutputStream out = **new** ObjectOutputStream(  **new** FileOutputStream("aluno.ser")))  {out.writeObject(aluno);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }    **private** **static** **void** leitorObjeto()  { **try**(ObjectInputStream input = **new** ObjectInputStream(  **new** FileInputStream("aluno.ser")))  { Aluno aluno = (Aluno)input.readObject();  System.***out***.println(aluno);  }  **catch** (IOException | ClassNotFoundException e)  {System.***out***.println(e.getMessage());}  }  } |
| Aluno [id=1, name=Eduardo, password=null, nomeEscola=DevDojo] |

O transiente serve justamente para ignorar o atributo. Mas e quando temos serialização e atributos de outras classes?

|  |
| --- |
| **public** **class** Turma  { **private** String nome;  **public** Turma(String nome)  {**this**.nome = nome;}  **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  @Override **public** String toString()  {**return** "Turma [nome=" + nome + "]";}  } |
| **public** **class** Aluno **implements** Serializable  { **private** Long id;  **private** String name;  **private** **transient** String password;  **private** **static** String *nomeEscola* = "DevDojo";  **private** Turma turma;    **public** Aluno(Long id, String name, String password)  { **this**.id = id;  **this**.name = name;  **this**.password = password;  }    **public** Long getId() {**return** id;}  **public** **void** setId(Long id) {**this**.id = id;}    **public** String getName() {**return** name;}  **public** **void** setName(String name) {**this**.name = name;}    **public** String getPassword() {**return** password;}  **public** **void** setPassword(String password)  {**this**.password = password;}    **public** **static** String getNomeEscola() {**return** *nomeEscola*;}  **public** **static** **void** setNomeEscola(String nomeEscola)  {Aluno.*nomeEscola* = nomeEscola;}    **public** Turma getTurma() {**return** turma;}  **public** **void** setTurma(Turma turma) {**this**.turma = turma;}  @Override **public** String toString()  { **return** "Aluno ["  + "id=" + id  + ", name=" + name  + ", password=" + password  + ", turma="+ turma  +"]";  }  } |
| Aluno [id=1, name=Eduardo, password=null, turma=null] |

Isso aconteceu porque não atribuímos a nenhuma turma, mas e se fizermos?

|  |
| --- |
| .  .  .  **private** **static** **void** gravadorObjeto()  { Turma turma = **new** Turma("Maratona Java");  Aluno aluno = **new** Aluno(1L, "Eduardo", "123");  aluno.setTurma(turma);  **try**(ObjectOutputStream out = **new** ObjectOutputStream(  **new** FileOutputStream("aluno.ser")))  {out.writeObject(aluno);}  **catch** (IOException e)  {System.***out***.println(e.getMessage());}  }  .  .  . |
| br.edu.maratona.serializable.Turma  writing aborted; java.io.NotSerializableException: br.edu.maratona.serializable.Turma |

Isso ocorre pois a classe Turma não é serializável e as vezes não teremos acesso às classes como temos aqui. Mas se fizermos com que essa classe seja serializável:

|  |
| --- |
| **public** **class** Turma **implements** Serializable  { **private** String nome;  **public** Turma(String nome)  {**this**.nome = nome;}  **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  @Override **public** String toString()  {**return** "Turma [nome=" + nome + "]";}  } |
| Aluno [id=1, name=Eduardo, password=null, turma=Turma [nome=Maratona Java]] |

Ou então podemos declarar que o objeto atributo da classe que a tem seja transiente, porém sabemos que o retorno será nulo para seu valor. Portanto, podemos usar dois métodos.

|  |
| --- |
| **public** **class** Aluno **implements** Serializable  { **private** Long id;  **private** String name;  **private** **transient** String password;  **private** **static** String *nomeEscola* = "DevDojo";  **private** **transient** Turma turma;    **public** Aluno(Long id, String name, String password)  { **this**.id = id;  **this**.name = name;  **this**.password = password;  }    **public** Long getId() {**return** id;}  **public** **void** setId(Long id) {**this**.id = id;}    **public** String getName() {**return** name;}  **public** **void** setName(String name) {**this**.name = name;}    **public** String getPassword() {**return** password;}  **public** **void** setPassword(String password)  {**this**.password = password;}    **public** **static** String getNomeEscola() {**return** *nomeEscola*;}  **public** **static** **void** setNomeEscola(String nomeEscola)  {Aluno.*nomeEscola* = nomeEscola;}    **public** Turma getTurma() {**return** turma;}  **public** **void** setTurma(Turma turma) {**this**.turma = turma;}    /\*na hr da serialização, ele vai verificar existe o método  \*Se existir, será chamado este writeObject()\*/  **private** **void** writeObject(ObjectOutputStream oos)  { //a órdem não importa, contanto que na hora da leitura  //siga a mesma órdem que foi colocado  **try**  { oos.defaultWriteObject();  oos.writeUTF(turma.getNome());  }  **catch**(IOException ioe)  {System.***out***.println(ioe.getMessage());}  }    **private** **void** readObject(ObjectInputStream ois)  { **try**  { ois.defaultReadObject();  turma = **new** Turma(ois.readUTF());  }  **catch**(IOException | ClassNotFoundException e)  {System.***out***.println(e.getMessage());}  }  @Override **public** String toString()  { **return** "Aluno ["  + "id=" + id  + ", name=" + name  + ", password=" + password  + ", turma="+ turma  +"]";  }  } |
| Aluno [id=1, name=Eduardo, password=null, turma=Turma [nome=Maratona Java]] |

**OBS: atributos estáticos não são ligados a objetos e sim às classes, então estes atributos não são serializáveis.**

**Mesmo que tenhamos uma classe não serializável mas que herda de uma classe serializável, irá funcionar perfeitamente, pois o java na hora da deserialização não chama o construtor. Então por mais que tenhamos o *super*, este não será executado.**

# Genérics

Até a versão 1.1 do java, tínhamos que criar nossas listas através da classe *List* sem o <T>, ou seja:

*List lista = new ArrayList();*

E assim podíamos adicionar qualquer dentro dessa lista, sem se importar com o tipo. Mas e se tivéssemos que apresentar esses valores dentro de um looping? Teríamos que fazer todas as condições de verificação de cada tipo.

Então a partir do java 5, foram implementados as classes *Generics*, que são exatamente as classes que inserimos dentro do <CLASS>. A partir do java 7, não seria necessário repetir a classe na hora de instanciar um *List* com *Generics*.

*List<CLASS> lista = new ArrayList<>();*

Porém, no java, todas as vezes que lançam uma nova versão, é pensando que a versão será compatível. Então *Genérics* é algo que existe somente em tempo de compilação. Isso é uma informação muito importante, pois imagine o seguinte, que tenhamos o seguinte código.

|  |
| --- |
| **public** **class** GenericsTeste  { **public** **static** **void** main(String[] args)  { List<String> lista = **new** ArrayList<>();    lista.add("Ola");  lista.add("Edu");    **for**(String word:lista)  {System.***out***.print(word + " ");}    *add*(lista, 1L);    **for**(String word:lista)  {System.***out***.print(word + " ");}  }  **public** **static** **void** add(List lista, Long l)  {lista.add(l);}  } |
| Ola Edu Ola Edu Exception in thread "main" java.lang.ClassCastException: java.lang.Long cannot be cast to java.lang.String  at br.edu.maratona.generics.GenericsTeste.main(GenericsTeste.java:18) |

Se retirarmos tudo o que está em amarelo, veremos que irá funcionar, pois em tempo de compilação o java irá fazer o checklist e ficar o que está tentando colocar é uma *String*, pois define como *String*. O problema é que isso é permitido, ou seja, usamos uma lista genérica e passamos para esta que essa nossa lista que definimos como String, pois o java em tempo de execução não sabe que essa *ArrayList* é de String. Foi dessa forma que conseguiram manter essa retro compatibilidade com todas as versões anteriores do java, do 1 ao 4.

# Wild Card

Considerando caractere coringa de Generics. Vamos primeiramente criar uma classe abstrata chamada Animal com um um único método.

|  |
| --- |
| **public** **abstract** **class** Animal  {**public** **abstract** **void** consulta();}  **class** Cachorro **extends** Animal  { @Override **public** **void** consulta()  {System.***out***.println("Consultando Cachorro");}  }  **class** Gato **extends** Animal  { @Override **public** **void** consulta()  {System.***out***.println("Consultando Gato");}  }  **class** WildCardTest  { **public** **static** **void** main(String[] args)  { Cachorro[] cachorros = {**new** Cachorro(), **new** Cachorro()};  Gato[] gatos = {**new** Gato(), **new** Gato()};  *consultarAnimais*(cachorros);  *consultarAnimais*(gatos);    List<Cachorro> cachorroList = **new** ArrayList<>();  cachorroList.add(**new** Cachorro());  List<Gato> gatoList = **new** ArrayList<>();  gatoList.add(**new** Gato());    *consultaAnimaisLista*(cachorroList);  *consultaAnimaisLista*(gatoList);  }    **public** **static** **void** consultarAnimais(Animal[] animais)  { **for**(Animal animal:animais)  {animal.consulta();}  }    /\*O java em tempo de compilação, por si só não  \*não consegue identificar o tipo de objeto da classe  \*que pertence, então pra isso foi inventado o wildcard '?'  \*Isso diz para o java que estamos inserindo qualquer lista  \*que seja da subclasse de Animal, mas garante que não  \*iremos inserir nada\*/  **public** **static** **void** consultaAnimaisLista(  List<? **extends** Animal> animais)  { **for**(Animal animal:animais)  {animal.consulta();}  }  } |
| Consultando Cachorro  Consultando Cachorro  Consultando Gato  Consultando Gato  Consultando Cachorro  Consultando Gato |

Perceba então que esse “?” garante que recebamos, no caso, qualquer lista que tenha uma subclasse de Animal. Além dessa forma, podemos também criar listas das seguintes formas.

# Classes Genéricas

|  |
| --- |
| **public** **class** Carro  { **private** String nome;  **public** Carro(String nome) {**this**.nome = nome;}  **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  @Override **public** String toString()  {**return** "Carro [nome=" + nome + "]";}  } |
| **public** **class** GenericClassTeste  { **public** **static** **void** main(String[] args)  { CarroAlugavel ca = **new** CarroAlugavel();  Carro carroAlugado = ca.getCarroDipsonivel();  System.***out***.println("Usando o carro por um mês");  ca.devolverCarro(carroAlugado);  }  }  /\*Criação de objetos que possam ser alugados, sabemos  \*que só podemos alugar se estiver disponível.\*/  **class** CarroAlugavel  { **private** List<Carro> carrosDisponíveis = **new** ArrayList<>();    /\*Bloco de inicialização\*/  { **this**.carrosDisponíveis.add(**new** Carro("Gol"));  **this**.carrosDisponíveis.add(**new** Carro("BMW"));  }    **public** Carro getCarroDipsonivel()  { Carro carro = **this**.carrosDisponíveis.remove(0);  System.***out***.println("Alugando carro: " + carro);  System.***out***.println("Carros disponíveis " + **this**.carrosDisponíveis);  **return** carro;  }    **public** **void** devolverCarro(Carro carro)  { System.***out***.println("Devolvendo carro " + carro);  **this**.carrosDisponíveis.add(carro);  System.***out***.println("Carros disponíveis " + **this**.carrosDisponíveis);  }  } |

Se quiséssemos fazer a mesma coisa para uma outra classe, teríamos que criar uma nova classes, com os mesmos métodos etc. Mas ao invés disso, podemos criar uma classe genérica, a fim de diminuir em muito essa quantidade de código.

|  |
| --- |
| **public** **class** ClasseGenericaTeste  { **public** **static** **void** main(String[] args)  { List<Carro> carrosDisponiveis = **new** ArrayList<>();    carrosDisponiveis.add(**new** Carro("GOL"));  carrosDisponiveis.add(**new** Carro("BMW"));  ObjetosAlugaveis<Carro> carroAlugavel = **new**  ObjetosAlugaveis<>(carrosDisponiveis);    Carro carro = carroAlugavel.getObjetoDipsonivel();  System.***out***.println("Usando o carro por um mês");  carroAlugavel.devolverObjeto(carro);  }  }  /\*Essa notação "T" significa que iremos passar um tipo  \*para essa classe. Essa letra significa TYPE = tipo.  \*Quando estamos trabalhando com coleções, como uma lista  \*utilizamos a letra "E" de elemento. Fora das coleções iremos  \*usar um "T". Onde tiver essa letra, no caso abaixo iremos trocar  \*por um outro nome. Onde tiver Carro, iremos substituir por T, ou seja  \*o tipo que será passado.\*/  **class** ObjetosAlugaveis<T>  { **private** List<T> objDisponiveis = **new** ArrayList<>();    **public** ObjetosAlugaveis(List<T> objDisponiveis)  {**this**.objDisponiveis = objDisponiveis;}    **public** T getObjetoDipsonivel()  { T t = **this**.objDisponiveis.remove(0);  System.***out***.println("Alugando objeto: " + t);  System.***out***.println("Carros disponíveis " + **this**.objDisponiveis);  **return** t;  }    **public** **void** devolverObjeto(T t)  { System.***out***.println("Devolvendo objeto " + t);  **this**.objDisponiveis.add(t);  System.***out***.println("Objetos disponíveis " + **this**.objDisponiveis);  }  } |
| **Resultado**  Alugando objeto: Carro [nome=GOL]  Carros disponíveis [Carro [nome=BMW]]  Usando o carro por um mês  Devolvendo objeto Carro [nome=GOL]  Objetos disponíveis [Carro [nome=BMW], Carro [nome=GOL]] |

Se quiséssemos que fosse feito para uma outra classe, simplesmente substituiríamos, por exemplo, Carro por Computador. Isso ajuda a reduzir e muito as linhas de códigos de seus projetos.

# Métodos Genéricos

|  |
| --- |
| **public** **class** MetodoGenericoTeste  { **public** **static** **void** main(String[] args)  {*criarArray*(**new** Carro("Dudu"));}  /\*Queremos criar um array do tipo T, adicioná-lo  \*nesse array e imprimir\*/  **public** **static** <T> **void** criarArray(T t)  { List<T> lista = **new** ArrayList<>();  lista.add(t);  System.***out***.println(lista);  }  } |
| **Resultado:**  [Carro [nome=Dudu]] |

Repare que para o método, foi necessário inserir o tipo, mesmo sendo um método do tipo void. Se não colocarmos esse tipo <T>, o código não irá compilar.

# Inner class

Essa classe foi criada na verdade com intuito de criação de interfaces, para manipularmos eventos. Pois imagina que temos uma tela e nessa tela temos métodos normais que vão trabalhar pegando texto, convertendo-o ou fazendo outras operações. Temos operações que devem manipular eventos, como por exemplo no clique de um botão. Se pararmos para pensar do princípio da orientação a objeto, deveriam ser classes diferentes, uma para manipular o chat e outra para manipular os eventos de chat. Por isso no java foram criadas as classes internas.

O problema é que não sabemos como trabalhar com interfaces gráficas e não é o objetivo do curso. Será então apenas explicado como usa-las, como as utilizamos, criamos, chamamos, etc. Quando começarmos com a parte de *Design Patterns*, existe um projeto chamado *builder*, onde fazemos o uso de dessas classes internas.

|  |
| --- |
| **public** **class** InnerClassTeste  { **public** **static** **void** main(String[] args)  {  }  }  **class** Externa  { **private** String nome = "Eduardo W. Murakoshi";    /\*Classe interna é como se fosse um método,  \*pois tem acesso a todos os atributos da  \*classe externa, inclusive os privados.\*/  **class** Interna  { **public** **void** verClasseExterna()  {System.***out***.println(nome);}  }  } |

Para chamar uma classe Interna, no nosso caso dentro da *main*, precisamos primeiro necessariamente de um objeto da classe Externa para chamarmos assim o objeto da classe interna. Tecnicamente falando:

|  |
| --- |
| **public** **class** InnerClassTeste  { **public** **static** **void** main(String[] args)  { Externa e = **new** Externa();  Externa.Interna i = e.**new** Interna();  i.verClasseExterna();  }  }  **class** Externa  { **private** String nome = "Eduardo W. Murakoshi";    /\*Classe interna é como se fosse um método,  \*pois tem acesso a todos os atributos da  \*classe externa, inclusive os privados.\*/  **class** Interna  { **public** **void** verClasseExterna()  {System.***out***.println(nome);}  }  } |
| **Resultado**  Eduardo W. Murakoshi |

Podemos também instanciar uma classe interna de outra maneira, considerando que não temos o objeto “e”.

|  |
| --- |
| **public** **class** InnerClassTeste  { **public** **static** **void** main(String[] args)  { Externa e = **new** Externa();  Externa.Interna i = e.**new** Interna();  i.verClasseExterna();  Externa.Interna i2 = **new** Externa().**new** Interna();  i2.verClasseExterna();  }  }  **class** Externa  { **private** String nome = "Eduardo W. Murakoshi";    /\*Classe interna é como se fosse um método,  \*pois tem acesso a todos os atributos da  \*classe externa, inclusive os privados.\*/  **class** Interna  { **public** **void** verClasseExterna()  {System.***out***.println(nome);}  }  } |
| **Resultado**  Eduardo W. Murakoshi  Eduardo W. Murakoshi |

A palavra chave *this* pode ser usado em ambas as classes, ou seja, tanto na interna como na externa, só vai depender onde esta sendo chamado. Se tentarmos fazer como no código abaixo, teremos um erro de compilação, pois estamos colocando um *this* que esta referente a classe Interna com um atributo da classe externa.

|  |
| --- |
| **class** Externa  { **private** String nome = "Eduardo W. Murakoshi";    /\*Classe interna é como se fosse um método,  \*pois tem acesso a todos os atributos da  \*classe externa, inclusive os privados.\*/  **class** Interna  { **public** **void** verClasseExterna()  {System.***out***.println(**this**.nome);}  }  } |

Se quisermos usar a palavra *this* no mesmo local, podemos fazer da seguinte maneira.

|  |
| --- |
| **public** **class** InnerClassTeste  { **public** **static** **void** main(String[] args)  { Externa e = **new** Externa();  Externa.Interna i = e.**new** Interna();  i.verClasseExterna();  Externa.Interna i2 = **new** Externa().**new** Interna();  i2.verClasseExterna();  i2.verParametroInterno();  }  }  **class** Externa  { **private** String nome = "Eduardo W. Murakoshi";    /\*Classe interna é como se fosse um método,  \*pois tem acesso a todos os atributos da  \*classe externa, inclusive os privados.\*/  **class** Interna  { **private** **int** id = 10;  **public** **void** verClasseExterna()  {System.***out***.println(Externa.**this**.nome);}    **public** **void** verParametroInterno()  {System.***out***.println(**this**.id);}  }  } |
| **Resultado**  Eduardo W. Murakoshi  Eduardo W. Murakoshi  10 |

# Classes Anônimas

Imagine que temos uma classe chamada Animal, que terá um método:

|  |
| --- |
| **class** Animal  { **public** **void** caminhar()  {System.***out***.println("Caminhando");}  }  **public** **class** ClassesAnonimasTeste  { **public** **static** **void** main(String[] args)  { Animal animal = **new** Animal();  animal.caminhar();  }  } |
| **Resultado**  Caminhando |

Agora imagine que precisamos mudar o comportamento, ou seja, alterar o método, por exemplo, caminhando rápido, mas não queremos criar uma subclasse de Animal só para alterar o método andar. A forma correta de fazer isso é sobre escrevendo o método em outra classe. Porém criar uma outra classe para resolver um problema pontual não é uma boa regra ou uma boa opção. Pra isso, podemos utilizar as classes Anônimas.

Uma classe anônima é uma classe que é definida em determinado ponto de código, onde não terá nome e simplesmente vai deixar de existir depois que utilizarmos. Por exemplo, ao invés de criar um objeto de animal, como no código acima, terminando com “;”, simplesmente iremos abrir e fechar {CHAVES}, como se fosse um método. No final, precisamos de um ponto e vírgula.

|  |
| --- |
| **class** Animal  { **public** **void** caminhar()  {System.***out***.println("Caminhando");}  }  **public** **class** ClassesAnonimasTeste  { **public** **static** **void** main(String[] args)  { /\*Temos uma variável de referência do tipo  animal, que está fazendo referência pra um  objeto animal, que é uma subclasse de animal  e que não possui nenhum nome.\*/  Animal animal = **new** Animal()  { **public** **void** caminar()  {System.***out***.println("Andando mais rapidamente");}  };  /\*Estamos aqui fazendo uma referência a subclasse  \*sem nome, que é a classe anônima que foi criada.\*/  animal.caminhar();  }  } |
| **Resultado.**  Andando mais rapidamente |

O bom de classes anônimas é que não precisamos criar somente classes, podemos também usá-los em interfaces.

|  |
| --- |
| **class** Animal  { **public** **void** caminhar()  {System.***out***.println("Caminhando");}  }  **class** CarroComparator **implements** Comparator<Carro>  { @Override **public** **int** compare(Carro arg0, Carro arg1)  {**return** arg0.getNome().compareTo(arg1.getNome());}  }  **public** **class** ClassesAnonimasTeste  { **public** **static** **void** main(String[] args)  { /\*Temos uma variável de referência do tipo  animal, que está fazendo referência pra um  objeto animal, que é uma subclasse de animal  e que não possui nenhum nome.\*/  Animal animal = **new** Animal()  { **public** **void** caminhar()  {System.***out***.println("Andando mais rapidamente");}  };  /\*Estamos aqui fazendo uma referência a subclasse  \*sem nome, que é a classe anônima que foi criada.\*/  animal.caminhar();    /\*Digamos que queiramos ordenar esses carros.\*/  List<Carro> carros = **new** ArrayList<>();  carros.add(**new** Carro("BMW"));  carros.add(**new** Carro("Audi"));  /\*Sabemos que não podemos criar um objeto direto de uma  \*interface, mas aqui estamos criando um objeto anonimo  \*que implementa uma interface\*/  Collections.*sort*(carros, **new** Comparator<Carro>()  { @Override **public** **int** compare(Carro o1, Carro o2)  {**return** o1.getNome().compareTo(o2.getNome());}  });  System.***out***.println(carros);  }  } |
| **Resultado**  Andando mais rapidamente  [Carro [nome=Audi], Carro [nome=BMW]] |

Perceba que neste caso, conseguimos fazer com que os carros ficassem em ordem alfabética.

# Classes Internas Estáticas

As classes estáticas são como se fossem membros estáticos, não são na verdade bem classes, apesar de utilizarmos o termo classe. A regra de atributos estáticos e não estáticos é aplicada igualmente para classes estáticas. Então se por exemplo tivermos um atributo chamado “nome”, não podemos acessar este atributo a partir da classe estática, a não ser que o atributo também seja estático.

|  |
| --- |
| **class** Externa  { **private** String *nome =* “ola”;  **static** **class** Interna  { **public** **void** imprimir()  {System.***out***.println(*nome*);}//erro  }  } |
| **class** Externa  { **private** **static** String *nome =* “ola”;  **static** **class** Interna  { **public** **void** imprimir()  {System.***out***.println(*nome*);}//correto  }  } |

Para chamarmos o método e a forma de instanciarmos um objeto de uma classe interna estática é muito simples.

|  |
| --- |
| **class** Externa  { **private** **static** String *nome* = "Ola";  **static** **class** Interna  { **public** **void** imprimir()  {System.***out***.println(*nome*);}  }  }  **public** **class** StaticInnerClassTest  { **public** **static** **void** main(String[] args)  { Externa.Interna interna = **new** Externa.Interna();  interna.imprimir();  }  } |
| **Resultado:**  Ola |

Vamos agora, dentro da classe que contém o método *main*, criar uma nova classe interna. Para este caso, para chamarmos o método desta classe interna, devemos somente instanciar a própria classe interna e chamar seu método.

|  |
| --- |
| **class** Externa  { **private** **static** String *nome* = "Ola";  **static** **class** Interna  { **public** **void** imprimir()  {System.***out***.println(*nome*);}  }  }  **public** **class** StaticInnerClassTest  { **public** **static** **void** main(String[] args)  { Externa.Interna interna = **new** Externa.Interna();  interna.imprimir();  Interna2 interna2 = **new** Interna2();  interna2.imprimir();  }  **static** **class** Interna2  { **public** **void** imprimir()  {System.***out***.println("Nova classe interna");}  }  } |
| **Resultado:**  Ola  Nova classe interna |

# Threads

Iremos agora tratar de um dos assuntos mais difíceis do java, que são as threads, pois a única certeza que temos é que não podemos ter certeza de quase nada. Uma *thread* no java possui dois significados diferentes: O primeiro é que é uma instância, um objeto comum do java, pois temos a sua classe Thread e temos também uma thread ou linha de execução. A primeira thread fala de uma instância, um objeto comum que vive no *hippie* e pode ser removido no garbage collector e uma linha de execução é como se fosse um processo paralelo.

Thread então é um trabalhador que irá trabalhar em nossos processos. A JVM, é quem vai trabalhar fazendo o schedulle, decidindo qual *thread* começa e qual termina. É como se fosse um mini sistema operacional que cuidará das próprias linhas de excução. Existem threads nativas mas não faz parte da certificação e também não precisamos saber. Mas as JVMs trabalham juntamente com as threads iniciadas pelo sistema operacional.

Temos duas formas de criarmos threads e dois tipos de threads. A diferença que existem entre elas e que precisamos saber é que uma JVM só finaliza uma execução quando todas as threads users forem finalizadas.

## A) Threads daimon

Se tiver uma thread desse tipo rodando, a JVM não se importa e parar a própria execução por causa desta. Exemplo: *Garbage collector*.

## B) Usuário

Enquanto que se tiver uma thread dessa executando, a JVM não parará de ser executado.

Existem então duas opções para criarmos uma thread, ou nós extendemos da classe java.lang.thread ou implementamos uma interface, chamada *runnable*. Vamos começar da maneira mais simples que é extendendo da classe *Thread*, embora não seja, do ponto de vista da POO, pois só podemos extender uma classe quando formos especializá-la.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  ThreadExemplo thread = **new** ThreadExemplo('A');  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  } |
| **Resultado**  Main |

Veremos agora uma série de threads sendo executados.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  ThreadExemplo t1 = **new** ThreadExemplo('A');  ThreadExemplo t2 = **new** ThreadExemplo('B');  ThreadExemplo t3 = **new** ThreadExemplo('C');  ThreadExemplo t4 = **new** ThreadExemplo('D');    t1.run();  t2.run();  t3.run();  t4.run();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  } |
| **Resultado:**  main  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD |

Veja que embora tenhamos colocado em sequência, uma thread só começou depois que a anterior terminou. Para iniciarmos uma thread, devemos usar o método *start()*. Aí sim teremos um resultado diferente da anterior.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  ThreadExemplo t1 = **new** ThreadExemplo('A');  ThreadExemplo t2 = **new** ThreadExemplo('B');  ThreadExemplo t3 = **new** ThreadExemplo('C');  ThreadExemplo t4 = **new** ThreadExemplo('D');    t1.start();  t2.start();  t3.start();  t4.start();  }  } |
| **Resultado:**  main  A  AAAAAAAAAAAAAAAAAAAAB  ABABBABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAA  AAAAAAAAAAAAD  DDAC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADDDDDDDDDDDDDAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD |

Isso nos mostra como a JVM separa as nossas threads. Podemos ver que neste caso não temos nenhuma ordem sendo executadas. Uma vez que demos o *start()*, será responsabilidade da JVM qual dessas threads será executada e por quanto tempo elas ficarão sendo executadas.

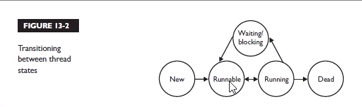
A outra forma que temos para executar uma thread, é implementando a interface *runnable*, que é por padrão uma boa prática de orientação a objetos.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    t1. start ();  t2. start ();  t3. start ();  t4. start ();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  }  **class** ThreadExemploRunnable **implements** Runnable  { **private** **char** c;  **public** ThreadExemploRunnable(**char** c)  {**this**.c = c;}    @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  } |
| Thread-0  A  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAThread-2  C  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCThread-1  B  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBThread-3  D  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD |

# Estados de threads

Na aula anterior, vimos que cada thread será executado e irá parar, mas não sabemos como e quando. Isso acontece pois temos um *scheduller*, que controla quando uma *thread* começa e quando termina. Quando uma *thread* finaliza quando o método *run()* termina a sua execução, significa que essa thread está morta. Uma coisa muito importante é, uma vez que iniciamos uma *thread*,“start()”, nunca mais poderemos inicia-la novamente, pois caso tentarmos iniciar, teremos uma exceção.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*Método da interface runnable\*/  t1.start();  t2.start();  t3.start();  t4.start();  //tentando iniciar t1 novamente  t1.start();  }  } |
| Exception in thread "main" java.lang.IllegalThreadStateException  at java.lang.Thread.start(Unknown Source)  at br.edu.maratona.threads.ThreadClassTeste.main(ThreadClassTeste.java:21) |



Quando executamos o método *start()*, significa que a thread está pronta para ser executada. A decisão de inicio e fim de uma thread é da scheduller. Existe momento em que a thread está em estado de bloqueio ou espera, depois do estado de execução, e em seguida entra em estado de pronto para ser executado e assim por diante, fica nesse ciclo até que seja finalizado.

Uma thread nunca poderá dizer para outra thread para ser bloqueada, coisa que era possível antes mas os métodos foram depreciados, pois é muito complicado pegar uma thread e essa dizer para outra esperar, quer dizer que a thread está tomando a decisão.

# Parando Execução com Sleep, Yeld e Join

## Sleep

Nessa aula iremos ver como prevenir uma execução de uma thread ou como fazê-la sair do estado de *run*. Sabemos que quando uma thread está no estado *running* e for para o estado *dead* significa que não podemos mais ativá-la novamente e sim criar uma nova. Mas sabemos que do estado de *running* a thread pode partir para outro estado, e existem algumas formas de forçarmos isso mas nem todas são garantidas. A primeira delas é utilizando método *sleep*, e quando a usamos, que é uma método estático da classe Thread, pois nunca podemos mandar uma outra thread dormir, estamos fazendo a referida thread pausar por um determinado tempo.

Imagina que uma thread está rodando em um looping e nesse looping estamos fazendo uma atualização nos preços de estoque, se fazemos uma thread rodando diretamente no looping, estamos consumindo banda e processamento. Uma forma de executar essa thread a cada 5 minutos por exemplo é mandar a thread dormir por 5 minutos e quando acordar novamente continue o download e atualiza o estoque.

Este método sleep é um método que pode lançar uma exceção *CheckedInterrupeterException* pois precisa dormir por aquele tempo determinado.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*Método da interface runnable\*/  t1.start();  t2.start();  t3.start();  t4.start();  //tentando iniciar t1 novamente  //t1.start();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  }  **class** ThreadExemploRunnable **implements** Runnable  { **private** **char** c;  **public** ThreadExemploRunnable(**char** c)  {**this**.c = c;}    @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  //exemplo de uso do sleep  **try** {Thread.*sleep*(1000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}  }  }  } |

A cada um segundo, está saindo um bloco de 4 (ABCD ou em qualquer outra ordem). Problema é que não podemos ter certeza por quanto tempo aquela thread será executado, antes de ser colocada para dormir. Uma das formas que temos de forçar as threads a dar uma chance para outras threads serem executadas, é usando o sleep. Este método é estático, que faz uma thread dormir pelo tempo determinado e a thread podemos colocar em qualquer lugar. Pois todo programa está sendo executado por algum tipo de thread. Nesse caso iremos fazer a thread main dormir por 1000 ms antes de finalizar a execução.

Um assunto que precisamos falar também é sobre prioridades de threads. A JVM tem uma especificação, pois temos várias empresas que podem criar JVMs e na especificação ela não diz exatamente como deve ser tratada a execução das threads e geralmente as threads elas são divididas por tempo. Então cada schedule define um tempo para aquela thread, por exemplo, uma thread que será executado por tantos segundos e outras por tantos segundos e além disso eles tem algo que é a prioridade. A prioridade das threads geralmente vai de 1 a 10, sendo a primeira a de menor prioridade e 10 a de maior, mas não é sempre então depende muito da especificação da JVM e não deveríamos programar nossos softwares multi thread baseado nisso. Por exemplo, se precisarmos definir uma thread com maior prioridade, por exemplo, queremos que t4 tenha maior prioridade, basta invocarmos o método *setPriority(VALOR DE 1 A 10)*. A thread de maior prioridade, será a thread que voltará a ser executada, mas isso não é certeza, apenas o que diz a maioria da especificação.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*como boas prática, sempre usar o atributo estático\*/  t4.setPriority(Thread.***MAX\_PRIORITY***);  /\*Método da interface runnable\*/  t1.start();  t2.start();  t3.start();  t4.start();  //tentando iniciar t1 novamente  //t1.start();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  }  **class** ThreadExemploRunnable **implements** Runnable  { **private** **char** c;  **public** ThreadExemploRunnable(**char** c)  {**this**.c = c;}    @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  //exemplo de uso do sleep  /\* try {Thread.sleep(1000);} catch (InterruptedException e)  \* {System.out.println(e.getMessage());}  \*/  }  }  } |
| D  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAThread-1  B  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC |

Mas e se tivermos mais de uma thread com as mesmas prioridades, o scheduler provavelmente vai decidir entre 1 ou 2 das 10 mas isso também não é certeza. O que temos mais certeza é que a maioria dos casos a thread que tiver a maior prioridade ou igual da thread que estiver executando, se sair irá voltar a ser executada no lugar da running anterior.

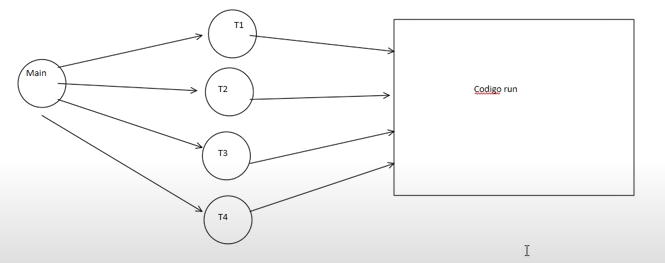
## B) Yield

Este método é também um método estático como o sleep e não tem a mesma garantia de que o sleep oferece. Este irá fazer com que a thread que estiver em execução no momento voltar para o estado *runnable* e dar uma chance para threads com as mesmas prioridades ser executado. Porém não é garantido que essa thread será executada seguindo esse pensamento.

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*como boas prática, sempre usar o atributo  \*estático\*/  t4.setPriority(Thread.***MAX\_PRIORITY***);  /\*Método da interface runnable\*/  t1.start();  t2.start();  t3.start();  t4.start();  //tentando iniciar t1 novamente  //t1.start();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  }  **class** ThreadExemploRunnable **implements** Runnable  { **private** **char** c;  **public** ThreadExemploRunnable(**char** c)  {**this**.c = c;}    @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  **if**(!Thread.*currentThread*().getName().equalsIgnoreCase("t4"))  {Thread.*yield*();}  //exemplo de uso do sleep  /\*  \* try {Thread.sleep(1000);} catch (InterruptedException e)  \* {System.out.println(e.getMessage());}  \*/  }  }  } |
| DA  DBC  BDADBCBDADBCBDADDDDBCBDADBCBDADBCBDADBB  CBDADDBCBDADBCBDADBCBDADBCBDADBCBDADBCB  DADBCCCCCCCCBDADBCCCCCCCCCCCBDAAAAAAAAA  AAAAADBBCBDADBCBDADBCBDADBCBDADBCBDADBC  BDADBCBDADBCBDADBCBDADBCBDADBCBDADBCBDA  DBCBDADBCBDADBCBDADBCBDADBCBDADBCBDADBC  BDADBCBDADBCBDADDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDBCBDBCBDBCBDBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBCBDBCBDBCDCDCDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  CCCBADABCBADABCBADABCBADABBBCBAD  ABCBADABCBADAABCBADABCBADABCBADA  BCBADABCBADABCBADABCBADABCBADABC  BADABCBADABCBADABCBADABCBADABCBADABCBADABCCCCCCCCCCCCCCCC  BADABCBADABCBADA  BCBADABCBADAAAAAAAAAAAABCCCCCCCCCCCCCCC  CCCCBADABCBADABCBADABCBADABCBABABABABAB  ABABABABABABABABABABABABABABAB  ABABABABABABAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAADAAAAAAAAAAAAAAAAAAAAAAABCBADA  AAAAAAAAAAAAABCBCADACBCADACBCADACBCADACBCADACBCADACBCADA  CBCADACBCADACBCACDACBCADACBCADACBCADACBCADACBCADACBCADAC  BCADACBCADACBCADACBCADACBCADACBCADACBCADACBCADACBCADACBC  ADACBCADACBCADACBCADACBCADACBCADACBCADACBCADACBCADACBCAD  ACBCADACBCADACBCCCADACBCAADACBCCADACBCADACBCADACBCADACBC  ADACBCADACBCADACBCADACBCADACBCADACCCCCCCCCCCBCADACBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBCBBBBBBBBBBADABCBADABCBADABCBADABCBADABCBADABCA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAADACBBCADACBCADACCCCCCCCCCCCCBCADACBCCADACCCBCAD  ACBCADACBCAADACBCADACBCADACBCADACBCADDDDACBACDCCCCCCABB  BBBBBBACDCCCCCCCCCABBBBBBBBACDCCCCCCCCCABBBBBACDCCCCCCC  CCCABACDCCCCABBBBBBBBACDCCCCABBBBBBBBBBBBBBA  CDCCCCCABBBBBBBACDCCCCCCCCCCCCCCABBBBABCCDCBB  BBBBBBBBBBBBBBBABCCCCCCCDCBBBBBBBABCCCCCCCCCC  CCDCBBBBBABCCCCCCCCDCBBBBBABCCCCDCBBBBBABCDCBBBBBABCCC  CCCCCCDCBBBBBABCCCCCCCCDCBABCDCBABCDCBABCDCBABCDCBABCD  CBABCDCBABCDCBABCDCBABCDCBABCDCBABCDCBABCDCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBABCDCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCBAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABC  DCBABCDCBABCDCBABCDCBABCDCBABBCDCBABCDCBABCDCB  ABCDCBABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCDCCCCCCBAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDCCCCCCCCCCCCCCCCCCCCCC  CCCCCCCCCCCCCCCCCCCCBAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDC  CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  CCCCCBAAAAAAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDCCBAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCCCCCCCCCCCCCCCC  CCCCCCCCCCDCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBABDBABDBAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABDDDDDDDDD  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  DDDDDDDDDDDDDDDDDDBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBA  BDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBBAB  DBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABBABDBAB  DBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDBABDB  ABDBABDBABBBBBBBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDB  DBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDB  DBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBDBBBBDBDBDBDBDDDBDBDBDBDBDBDDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDB  DBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDB  DBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDB  DBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBDBD  BDBBBDBDBDBDBDBDBDBDBDBDBDDDDDDDDDDDDDDDDDDDDBDBDBDBDBDBDBDDDD |

## C) Join

Outro método mas que é um pouco mais complicado de entender a primeira vista. Colocando nossa aplicação como exemplo, temos nela 4 threads e mais uma que é a *main*. A thread *main* é a aquela que inicia todas as outras threads. Todas elas estão executando o código *run()*.



A thread main irá executar as threads mas não sabemos se irá executar todas de uma vez ou se vai parar a execução por um momento uma delas, apenas sabemos que todas as threads estão sob o comando do schedullar. O que acontece é que este irá executar todas as threads. O método join vai fazer com que a thread que estiver em execução no momento só poderá ser executado novamente depois que a outra thread terminar sua execução. Exemplificando por código, usando apenas as threads 1 e 2:

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*como boas prática, sempre usar o atributo  \*estático\*/  t4.setPriority(Thread.***MAX\_PRIORITY***);  /\*Método da interface runnable\*/  t1.start();  t2.start();  /\*  \* t3.start(); t4.start();  \*/  //tentando iniciar t1 novamente  //t1.start();  }  } |
| **ABABABABABABABABABABABABABABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBABABABABABABABABABABABABABABABABABABABABABABA**  **BABABABABABABABABABABABABABABABABABABABABABABABABABABABABA**  **BAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**  **AAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**  **BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB** |

Veja que sem o método join, a thread *main* está executando ambas as threads 1 e 2 de forma alternada. O join simplesmente irá dizer para a *main* mandar executar a thread 2 somente após thread 1 ter finalizado. Então:

|  |
| --- |
| **public** **class** ThreadClassTeste  { **public** **static** **void** main(String[] args)  { /\*ThreadExemplo t1 = new ThreadExemplo('A');  ThreadExemplo t2 = new ThreadExemplo('B');  ThreadExemplo t3 = new ThreadExemplo('C');  ThreadExemplo t4 = new ThreadExemplo('D');  \*/  Thread t1 = **new** Thread(**new** ThreadExemploRunnable('A'));  Thread t2 = **new** Thread(**new** ThreadExemploRunnable('B'));  Thread t3 = **new** Thread(**new** ThreadExemploRunnable('C'));  Thread t4 = **new** Thread(**new** ThreadExemploRunnable('D'));    /\*como boas prática, sempre usar o atributo  \*estático\*/  t4.setPriority(Thread.***MAX\_PRIORITY***);  /\*Método da interface runnable\*/  t1.start();  **try** {t1.join();}  **catch** (InterruptedException e)  {JOptionPane.*showMessageDialog*(**null**, e);}  t2.start();  /\*  \* t3.start(); t4.start();  \*/  //tentando iniciar t1 novamente  //t1.start();  }  }  **class** ThreadExemplo **extends** Thread  { **private** **char** c;  **public** ThreadExemplo(**char** c)  {**this**.c = c;}    /\*Tudo funciona através desse  \*método. Que quisermos que  \*mais de uma coisa aconteça ao  \*mesmo tempo, devemos inserir  \*nesse método.\*/  @Override **public** **void** run()  { /\*para sabermos o nome da thread que estamos  executando no momento é chamando da seguinte  maneira logo abaixo\*/  System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  }  }  }  **class** ThreadExemploRunnable **implements** Runnable  { **private** **char** c;  **public** ThreadExemploRunnable(**char** c)  {**this**.c = c;}    @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName());  **for**(**int** i = 0; i < 1000; i++)  { System.***out***.print(c);  **if**(i%100 == 0) {System.***out***.println();}  **if**(!Thread.*currentThread*().getName().equalsIgnoreCase("t4"))  {Thread.*yield*();}  //exemplo de uso do sleep  /\*  \* try {Thread.sleep(1000);} catch (InterruptedException e)  \* {System.out.println(e.getMessage());}  \*/  }  }  } |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  AAAAAAAAAAAAAAAAAAAAAAAAAAAAThread-1BBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB |

# Sincronismo de código

O que é um sincronismo de código? Imaginamos um exemplo geral como contas bancárias. Imagine que tem uma conta bancária em conjunta com outra pessoa e com saldo de apenas R$10,00. Imagine agora que criamos um código onde todas as vezes que alguém vai tentar fazer um saque e criamos uma thread, mas nesse exato momento duas pessoas vão tentar efetuar o saque exatamente no mesmo minuto e segundo. Como temos apenas R$10,00, elas querem efeturar o saque de exatamente neste valor. O que acontece se essas duas threads tentarem acessar ao mesmo tempo esse saldo que temos? Vamos criar um novo pacote e mostrar de maneira prática.

|  |
| --- |
| **public** **class** Conta  { **private** **int** saldo = 50;  **public** **int** getSaldo() {**return** saldo;}    **public** **void** saque(**int** valor) {saldo -= valor;}  } |
| **public** **class** ContaTeste **implements** Runnable  { **private** Conta conta = **new** Conta();    **public** **static** **void** main(String[] args)  { ContaTeste contaTeste = **new** ContaTeste();  /\*Criando threads para cada usuário que tentarão  \*realizar saques\*/  Thread willian = **new** Thread(contaTeste, "Willian");  Thread ana = **new** Thread(contaTeste, "Ana");  /\*Iniciando ambas as threads\*/  willian.start();  ana.start();  }    /\*Teremos nesse método a sequinte regra:  \*1 - Iremos verificar se o saldo é maior que a quantidade  \*de que queremos efetuar;  \*  \*2 - Se for maior, iremos efetuar o saque;  \*  \*3 - Caso contrário, isso não será possível\*/  **private** **void** saque(**int** valor)  { **if**(conta.getSaldo() > valor)  { System.***out***.println(  Thread.*currentThread*().getName() +  " esta indo sacar");    conta.saque(valor);  System.***out***.println(  Thread.*currentThread*().getName() +  " saque realizado com sucesso");  }    **else**  { System.***out***.println(  Thread.*currentThread*().getName() +  " saldo indisponível para o valor de saque");  }  }    /\*Supondo que as notas disponíveis são apenas no valor de  \*R$10,00. Como o saldo é de R$50,00\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 5; i++)  { saque(10);  **if**(conta.getSaldo() < 0)  {System.***out***.println("Algo de errado não está certo");}  }  }  } |
| Willian esta indo sacar  Willian saque realizado com sucesso  Willian esta indo sacar  Willian saque realizado com sucesso  Willian esta indo sacar  Willian saque realizado com sucesso  Willian esta indo sacar  Willian saque realizado com sucesso  Willian saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque |

Esse gerenciamento é extremamente complicado, pois na verdade esse processo de executar o saque e depois disponibilizar o saldo deve ser feito de forma atômica, ou seja, indivisível, uma única thread tem que acessar o método *run()* por vez. Uma thread só pode ser executada se a anterior já estiver finalizada. Isso é o que chamamos de sincronismo de código.

Para que seja possível isso ser feito, os atributos de classes devem ser provados, em seguida temos que sincronizar o pedaço de código que queremos tornar indivisível o pedaço de código que seja acessado apenas por uma thread. Assim eliminamos a concorrência.

|  |
| --- |
| **public** **class** ContaTeste **implements** Runnable  { **private** Conta conta = **new** Conta();    **public** **static** **void** main(String[] args)  { ContaTeste contaTeste = **new** ContaTeste();  /\*Criando threads para cada usuário que tentarão  \*realizar saques\*/  Thread willian = **new** Thread(contaTeste, "Willian");  Thread ana = **new** Thread(contaTeste, "Ana");  /\*Iniciando ambas as threads\*/  willian.start();  ana.start();  }    /\*Teremos nesse método a sequinte regra:  \*1 - Iremos verificar se o saldo é maior que a quantidade  \*de que queremos efetuar;  \*  \*2 - Se for maior, iremos efetuar o saque;  \*  \*3 - Caso contrário, isso não será possível\*/  **private** **synchronized** **void** saque(**int** valor)  { **if**(conta.getSaldo() > valor)  { System.***out***.println(  Thread.*currentThread*().getName() +  " esta indo sacar");    conta.saque(valor);  System.***out***.println(  Thread.*currentThread*().getName() +  " saque realizado com sucesso");  }    **else**  { System.***out***.println(  Thread.*currentThread*().getName() +  " saldo indisponível para o valor de saque");  }  }    /\*Supondo que as notas disponíveis são apenas no valor de  \*R$10,00. Como o saldo é de R$50,00\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 5; i++)  { saque(10);  **if**(conta.getSaldo() < 0)  {System.***out***.println("Algo de errado não está certo");}  }  }  } |
| Willian esta indo sacar  Willian saque realizado com sucesso  Ana esta indo sacar  Ana saque realizado com sucesso  Ana esta indo sacar  Ana saque realizado com sucesso  Ana esta indo sacar  Ana saque realizado com sucesso  Ana saldo indisponível para o valor de saque  Ana saldo indisponível para o valor de saque  Willian saldo indisponível para o valor de saque  Willian saldo indisponível para o valor de saque  Willian saldo indisponível para o valor de saque  Willian saldo indisponível para o valor de saque |

Perceba que desta forma não podemos garantir quem irá começar e quem terminará com o saque. Todas as vezes que executarmos esse código, sairá um cenário diferente de sequência.

Isso acontece por que todo objeto java possui um *lock*, que é como se fosse uma chave e quando trabalhamos com métodos sincronizados, esse *lock* é utilizado. Neste caso, temos um lock para o objeto tipo Conta. Quando a Thread willian começa a executar esse bloco sincronizado, é como se pegasse uma chave e essa diz que nenhuma outra thread pode ser executado enquanto não tiver a chave. Por isso quando willian vai fazer o saque, o mesmo pega a chave e se a ana tentar, por mais que coloquemos um *sleep()*, a ana está preparada para executar mas quando tentar, irá olhar o *lock* da conta e ver que está indisponível. Se o método *sleep()* estiver travado por um tempo muito grande, essa thread ficará travado por esse tempo todo.

O sincronismo podemos utilizar em métodos ou blocos, nunca em classes ou variáveis. Em um método sincronizado, normalmente, a primeira que for executado será a primeira, mas não é 100% garantido. O sincronismo afeta a performance por conta disso. Se definirmos que, no caso, o objeto conta for um tipo *final*, aí garantimos o que foi afirmado.

O que será que acontece ser tivermos um método sincronizado e uma classe que também possui um método sincronizado? Uma thread pode adquirir mais de um *lock*. Lembrando que se tivermos apenas um *lock* por objeto, mais uma thread pode pegar mais de um *lock*. Então se uma thread entrar em um método sincronizado e adquirir um lock para essa conta, e essa conta tiver um outro método de uma outra classe e sincronizado, este pode adquirir o lock também. E essa thread pode chamar outros métodos sincronizados, pois o lock é por objeto e não por método.

Não somos obrigados a utilizar diretamente no método, podemos sincronizar o bloco de código.

|  |
| --- |
| **public** **class** ContaTeste **implements** Runnable  { **private** Conta conta = **new** Conta();    **public** **static** **void** main(String[] args)  { ContaTeste contaTeste = **new** ContaTeste();  /\*Criando threads para cada usuário que tentarão  \*realizar saques\*/  Thread willian = **new** Thread(contaTeste, "Willian");  Thread ana = **new** Thread(contaTeste, "Ana");  /\*Iniciando ambas as threads\*/  willian.start();  ana.start();  }  **private** **void** saque(**int** valor)  { **synchronized**(conta)  { **if**(conta.getSaldo() >= valor)  { System.***out***.println(  Thread.*currentThread*().getName() +  " esta indo sacar");    conta.saque(valor);  System.***out***.println(  Thread.*currentThread*().getName() +  " saque realizado com sucesso");  }    **else**  { System.***out***.println(  Thread.*currentThread*().getName() +  " saldo indisponível para o valor de saque");  }  }  }    /\*Supondo que as notas disponíveis são apenas no valor de  \*R$10,00. Como o saldo é de R$50,00\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 5; i++)  { saque(10);  **if**(conta.getSaldo() < 0)  {System.***out***.println("Algo de errado não está certo");}  }  }  } |

Podemos conseguir sincronismo em métodos estáticos, pois existe apenas uma cópia de métodos estáticos que estamos tentando proteger, pois pertencem à classe e não a objetos. Imaginemos então que tenhamos um método novo estático e manda-lo sincronizar, lembre se que isso seria como se estivesse adquirindo um lock em uma classe inteira, isso é a mesma coisa que fazer isso:

|  |
| --- |
| **public** **class** ContaTeste **implements** Runnable  { **private** **final** Conta conta = **new** Conta();    **public** **static** **void** main(String[] args)  { ContaTeste contaTeste = **new** ContaTeste();  /\*Criando threads para cada usuário que tentarão  \*realizar saques\*/  Thread willian = **new** Thread(contaTeste, "Willian");  Thread ana = **new** Thread(contaTeste, "Ana");  /\*Iniciando ambas as threads\*/  willian.start();  ana.start();  }    **private** **static** **void** imprime()  { /\*veremos mais para frente uma classe  que chamamos de literal, que é uma forma  que temos do compilador dizer para a JVM  olhar para cada instância dessa classe\*/  **synchronized**(ContaTeste.**class**)  {System.***out***.println("blablabla");}  }    /\*Teremos nesse método a sequinte regra:  \*1 - Iremos verificar se o saldo é maior que a quantidade  \*de que queremos efetuar;  \*  \*2 - Se for maior, iremos efetuar o saque;  \*  \*3 - Caso contrário, isso não será possível\*/  **private** **void** saque(**int** valor)  { **synchronized**(conta)  { **if**(conta.getSaldo() >= valor)  { System.***out***.println(Thread.*currentThread*().getName() +  " esta indo sacar");    conta.saque(valor);  System.***out***.println(Thread.*currentThread*().getName() +  " saque realizado com sucesso");  }    **else**  { System.***out***.println(Thread.*currentThread*().getName() +  " saldo indisponível para o valor de saque");  }  }  /\*  \* if(conta.getSaldo() >= valor) { System.out.println(  \* Thread.currentThread().getName() + " esta indo sacar");  \*  \* conta.saque(valor); System.out.println( Thread.currentThread().getName() +  \* " saque realizado com sucesso"); }  \*  \* else { System.out.println( Thread.currentThread().getName() +  \* " saldo indisponível para o valor de saque"); }  \*/  }    /\*Supondo que as notas disponíveis são apenas no valor de  \*R$10,00. Como o saldo é de R$50,00\*/  @Override **public** **void** run()  { **for**(**int** i = 0; i < 5; i++)  { saque(10);  **if**(conta.getSaldo() < 0)  {System.***out***.println("Algo de errado não está certo");}  }  }  } |

O sincronismo se não for bem feito, pode causar vários problemas e o pior deles é o *deadlock*, que é a situação de duas threads estiverem esperando pelo mesmo lock de objeto e elas ficam impossibilitadas de continuar a execução travando completamente o seu sistema.

# Interação entre as threads (wait, notify e notifyAll)

Existem basicamente 3 métodos para fazermos interação de threads e todos eles foram definidos na classe *Object* e todos eles foram criados para trabalharmos com comunicação entre threads. Imagine que temos um sistema, onde recebe e-mails. Se não pudéssemos comunicação entre threads, teríamos duas threads: uma para receber os e-mails e outra para envia-los. O problema é que a thread que recebe não sabe quando recebeu. Portanto era necessário ficar checando os e-mails a cada 2 ou 10 segundos.

Esses três métodos devem estar dentro de um contexto sincronizado, pois uma thread não pode invocar 2 métodos, a não ser que tenha o locker. Vamos exemplificar com um sisteminha de convite de membros.

|  |
| --- |
| **public** **class** ListaMembros  { **private** **final** Queue<String> emails = **new** LinkedList<>();  **private** **boolean** aberta = **true**;    **public** **int** getEmailsPendentes()  { **synchronized**(emails)  {**return** emails.size();}  }    **public** **boolean** isAberta() {**return** **true**;}    **public** String obterEmailMembro()  { String email = **null**;  **try**  { **synchronized**(**this**.emails)  { /\*Se os e-mails estivarem vazios, sginifica  que vai ter que esperar\*/  **while**(emails.size() == 0)  { **if**(!**this**.aberta) {**return** **null**;}  System.***out***.println("Lista vazia. Colocando a "  + "thread: " + Thread.*currentThread*().getName()  + " em modo wait");  **this**.emails.wait();  }  /\*Este método nos retorna e remove o primeiro valor  \*desta queue de e-mails\*/  email = **this**.emails.poll();  }  }  **catch**(InterruptedException ie) {System.***out***.println(ie.getMessage());}  **return** email;  }    //método para add e-mail  **public** **void** addEmailMembro(String email)  { **synchronized**(**this**.emails)  { **this**.emails.add(email);  System.***out***.println("Email adicionado na lista");  System.***out***.println("Notificando as threads que "  + "estão em espera");  /\*avisa todas as threads e fazem nas sairem do estado  \*blocked para runnable\*/  **this**.emails.notifyAll();  }  }  /\*Iremos notificar todas as listas que não fará mais  \*nenhum serviço\*/  **public** **void** fecharLista()  { System.***out***.println("Notificando todas as threads e fechando a lista");  **this**.aberta = **false**;  **synchronized**(**this**.emails) {**this**.emails.notifyAll();}  }  } |
| **public** **class** Entregador **implements** Runnable  { **private** ListaMembros listaMembros;    **public** Entregador(ListaMembros listaMembros)  {**this**.listaMembros = listaMembros;}  @Override **public** **void** run()  { String nomeThread = Thread.*currentThread*().getName();  System.***out***.println("Começando o trabalho de entrega");  **int** qtdePendente = listaMembros.getEmailsPendentes();  **boolean** aberta = **this**.listaMembros.isAberta();  **while**(aberta || qtdePendente > 0)  { **try**  { String email = **this**.listaMembros.obterEmailMembro();  **if**(email != **null**)  { System.***out***.println(nomeThread  + " Enviando email para " + email);  Thread.*sleep*(2000);  System.***out***.println("Envio para " + email  + " concluído com sucesso");  }    }  **catch**(InterruptedException ie)  {System.***out***.println(ie.getMessage());}  aberta = **this**.listaMembros.isAberta();  }  System.***out***.println("Atividade finalizada");  }  } |
| **public** **class** EntregadorTeste  { **public** **static** **void** main(String[] args)  { ListaMembros listaMembros = **new** ListaMembros();  Thread t1 = **new** Thread(**new** Entregador(listaMembros), "Entregador 1");  Thread t2 = **new** Thread(**new** Entregador(listaMembros), "Entregador 2");    t1.start();  t2.start();    **while**(**true**)  { String email = JOptionPane.*showInputDialog*("Digite o email do membro");  **if**(email == **null** || email.isEmpty())  { listaMembros.fecharLista();  **break**;  }    listaMembros.addEmailMembro(email);  }  }  } |
| Começando o trabalho de entrega  Lista vazia. Colocando a thread: Entregador 1 em modo wait  Começando o trabalho de entrega  Lista vazia. Colocando a thread: Entregador 2 em modo wait  Email adicionado na lista  Notificando as threads que estão em espera  Entregador 2 Enviando email para eduardo@gmal.com  Lista vazia. Colocando a thread: Entregador 1 em modo wait  Envio para eduardo@gmal.com concluído com sucesso  Lista vazia. Colocando a thread: Entregador 2 em modo wait  Email adicionado na lista  Notificando as threads que estão em espera  Entregador 2 Enviando email para rodrigo@gmail.com  Lista vazia. Colocando a thread: Entregador 1 em modo wait  Envio para rodrigo@gmail.com concluído com sucesso  Lista vazia. Colocando a thread: Entregador 2 em modo wait  Notificando todas as threads e fechando a lista |

Lembre-se que sempre dentro de um contexto sincronizado esses métodos podem ser utilizado, pois se tentar chamar o *notify* ou wait sem ter o *lock* daquele objeto teremos uma exceção em tempo de execução.

# Concorrência

## Variáveis atômicas

|  |
| --- |
| **class** Contador  { **private** **int** count;    **public** **void** increment() {count++;}    **public** **int** getCount() {**return** count;}  }  **class** IncrementadorThread **extends** Thread  { **private** Contador contador;  **public** IncrementadorThread(Contador contador)  {**this**.contador = contador;}  **public** **void** run()  { **for**(**int** i = 0; i < 10000; i++)  {contador.increment();}  }  }  **public** **class** Concorrencia  { **public** **static** **void** main(String[] args)  { Contador c = **new** Contador();  IncrementadorThread ic1 = **new** IncrementadorThread(c);  IncrementadorThread ic2 = **new** IncrementadorThread(c);    ic1.start();  ic2.start();    **try**  { ic1.join();  ic2.join();  System.***out***.println(c.getCount());  }  **catch**(InterruptedException ie)  {System.***out***.println(ie.getMessage());}  }  } |
| ***Resultado: dificilmente chegaremos no valor de 20000***  19834 |

Só garantiríamos obtermos sempre o valor de 20000 se o método de incremento for sincronizado, ou seja.

|  |
| --- |
| **class** Contador  { **private** **int** count;    **public** **synchronized** **void** increment() {count++;}    **public** **int** getCount() {**return** count;}  }  **class** IncrementadorThread **extends** Thread  { **private** Contador contador;  **public** IncrementadorThread(Contador contador)  {**this**.contador = contador;}  **public** **void** run()  { **for**(**int** i = 0; i < 10000; i++)  {contador.increment();}  }  }  **public** **class** Concorrencia  { **public** **static** **void** main(String[] args)  { Contador c = **new** Contador();  IncrementadorThread ic1 = **new** IncrementadorThread(c);  IncrementadorThread ic2 = **new** IncrementadorThread(c);    ic1.start();  ic2.start();    **try**  { ic1.join();  ic2.join();  System.***out***.println(c.getCount());  }  **catch**(InterruptedException ie)  {System.***out***.println(ie.getMessage());}  }  } |
| ***Resultado:***  20000 |

Porém, lembrando que o *synchronized* sempre será algo custoso para o processador. Pois precisamos prender e obter o lock daquele objeto. O segredo todo é justamente no momento em que é realizado o incremento, pois podemos pensar que isso é simplesmente uma linha de instrução, mas na verdade não é somente isso. Na verdade nem temos como saber se é realmente uma linha, mas devemos presumir que nunca é apenas uma linha. O que acontece é que o valor do *count* é copiado por uma variável temporária e essa é incrementada e o valor dessa variável temporária é copiada de novo para esse campo *count* e isso é feito pelo processador, ou seja, não temos controle sobre isso.

Para resolver isso iremos utilizar o pacote java.util.concurrentatommic e substituir o uso do *synchronized*.

|  |
| --- |
| **import** java.util.concurrent.atomic.AtomicInteger;  **class** Contador  { **private** **int** count;  **private** AtomicInteger atomicInteger = **new** AtomicInteger();    **public** **void** increment()  { **this**.count++;  **this**.atomicInteger.getAndIncrement();  }    **public** **int** getCount() {**return** **this**.count;}  **public** **int** getAtomicInteger(){**return** atomicInteger.intValue();}  }  **class** IncrementadorThread **extends** Thread  { **private** Contador contador;  **public** IncrementadorThread(Contador contador)  {**this**.contador = contador;}  **public** **void** run()  { **for**(**int** i = 0; i < 10000; i++)  {contador.increment();}  }  }  **public** **class** Concorrencia  { **public** **static** **void** main(String[] args)  { Contador c = **new** Contador();  IncrementadorThread ic1 = **new** IncrementadorThread(c);  IncrementadorThread ic2 = **new** IncrementadorThread(c);    ic1.start();  ic2.start();    **try**  { ic1.join();  ic2.join();  System.***out***.println(c.getCount());  System.***out***.println(c.getAtomicInteger());  }  **catch**(InterruptedException ie)  {System.***out***.println(ie.getMessage());}  }  } |
| ***Resultado:***  19997  20000 |

Numa variável atômica é garantido pois, no caso, continuou seguindo os mesmos passos para executar e o método *getAndIncrement()* foi criada de uma forma *thread safe* chamada CAS, que significa *Compare and Suap*, ou seja este compara e depois troca o valor. Este segue 3 passos, o primeiro o valor temporário é copiado numa variável temporária, em seguida este valor é incrementado e depois, antes de você retornar o valor para *AtomicInteger*, a CPU irá comparar o valor do *count* com o valor da variável temporária. Se for diferente então faz o SUAP, ou seja, troca o valor antigo pelo novo valor.

## Lock e ReentrantLock

Desta vez iremos conhecer a classe Lock, que também está no pacote *Concurrent*. O Lock é uma outra forma que temos para obter o lock de um objeto sem utilizar a palavra reservada *synchronized*. Podemos por enquanto utilizar o exemplo da aula anterior de Contador. O problema com o synchronized é que quando temos, por exemplo, 5 threads tentando acessar o método *incremente()*, a partir do momento que a primeira thread acessar, esta irá obter o lock do objeto, neste caso o *this* e este objeto não poderá ser utilizado por outras threads. Essas threads terão que esperar enquanto a thread que está executando terminar seu processo. Assim que finalizado, o agendador irá escolher as threads que estão em hold para iniciar.

Uma vez que a thread entra, esta obtém o lock e as outras threads ficam em *hold*. Aí que entra a diferença do *synchronized* do *lock*, pois esta possui uma habilidade de definir um timeout, por exemplo, tentar conseguir ter o lock desse objeto por 5 segundos, mas se não conseguir, continua a execução. A segunda diferença é que existe um conceito de fernes, ou seja, equalidade ou seja, quando todas as threads teriam supostamente a mesma condição de executar aquele código.

Imagine que temos 5 threads executando várias vezes um mesmo pedaço de código, pode ser que uma dessas threads elas nunca cheguem a executar um código e através desse Lock, do pacote *Concurrent*, temos a possibilidade de dizer “tente passar esse lock para a thread que está esperando mais tempo sem executar. Não é garantido mas existe essa possibilidade através do construtor.

|  |
| --- |
| **class** Contador  { **private** **int** count;  **private** AtomicInteger atomicInteger = **new** AtomicInteger();  //classe Lock, cuja interface que esta classe implementa é  //ReentrantLock. O parâmetro "true" é para indicar o fernes.  //Se não tiver esse parametro booleano, por default é false.  **private** Lock lock = **new** ReentrantLock(**true**);    **public** **void** increment()  { **this**.count++;  **this**.atomicInteger.getAndIncrement();  }    **public** **int** getCount() {**return** **this**.count;}  **public** **int** getAtomicInteger(){**return** atomicInteger.intValue();}  }  **class** IncrementadorThread **extends** Thread  { **private** Contador contador;  **public** IncrementadorThread(Contador contador)  {**this**.contador = contador;}  **public** **void** run()  { **for**(**int** i = 0; i < 10000; i++)  {contador.increment();}  }  }  **public** **class** Concorrencia  { **public** **static** **void** main(String[] args)  { Contador c = **new** Contador();  IncrementadorThread ic1 = **new** IncrementadorThread(c);  IncrementadorThread ic2 = **new** IncrementadorThread(c);    ic1.start();  ic2.start();    **try**  { ic1.join();  ic2.join();  System.***out***.println(c.getCount());  System.***out***.println(c.getAtomicInteger());  }  **catch**(InterruptedException ie) {System.***out***.println(ie.getMessage());}  }  } |

Quando estamos utilizando um synchronized, uma thread é bloqueada quando estiver esperando por um lock por um tempo indefinido de período. Enquanto naquele período, aquele método sincronizado não for finalizado, aquela thread ficará esperando eternamente. Mas temos um método dentro da classe ReentrantLock() que podemos dizer que aquela thread pode ser interrompida, ou seja, sair do seu modo de espera.

|  |
| --- |
| **class** Contador  { **private** **int** count;  **private** AtomicInteger atomicInteger = **new** AtomicInteger();  //classe Lock, cuja interface que esta classe implementa é  //ReentrantLock. O parâmetro "true" é para indicar o fernes.  //Se não tiver esse parametro booleano, por default é false.  **private** Lock lock = **new** ReentrantLock(**true**);    **public** **void** increment()  { //obtendo o lock.  **this**.lock.lock();  //toda vez que obtemos o lock, também  //precisamos liberá-lo. Pelas boas práticas  //devemos fazer isso com try-finally  **try**  { **this**.count++;  **this**.atomicInteger.getAndIncrement();  }  **finally** {**this**.lock.unlock();}    //this.count++;  //this.atomicInteger.getAndIncrement();  }    **public** **int** getCount() {**return** **this**.count;}  **public** **int** getAtomicInteger()  {**return** atomicInteger.intValue();}  }  **class** IncrementadorThread **extends** Thread  { **private** Contador contador;  **public** IncrementadorThread(Contador contador)  {**this**.contador = contador;}  **public** **void** run()  { **for**(**int** i = 0; i < 10000; i++)  {contador.increment();}  }  }  **public** **class** Concorrencia  { **public** **static** **void** main(String[] args)  { Contador c = **new** Contador();  IncrementadorThread ic1 = **new** IncrementadorThread(c);  IncrementadorThread ic2 = **new** IncrementadorThread(c);    ic1.start();  ic2.start();    **try**  { ic1.join();  ic2.join();  System.***out***.println(c.getCount());  System.***out***.println(c.getAtomicInteger());  }  **catch**(InterruptedException ie)  {System.***out***.println(ie.getMessage());}  }  } |
| **Resultado:**  20000  20000 |

Uma das desvantagens que temos entre o *synchronized* e o *lock* é que em questão de performance são praticamente a mesma coisa, mas o código fica bem mais feio, principalmente pelo fato de usar o try-finally. Não existe um *catch* pois não é uma obrigatoriedade, porém o unlock precisa sempre ser executado.

Se o que precisarmos fazer consegue ser atingido utilizando sinchronized, devemos ir pelo *sinchronized*. O uso da classe *ReentrantLock* é um pouco avançado, quando por exemplo, temos um bloco que precisa ser interrompida por um *wait* ou obter o lock em objeto e liberar em outro, que através do *ReentrantLock*, conseguimos fazer isso e com o *synchronized* não, ou por exemplo queremos escalar, pois temos muita contenção entre as threads, devemos usar o *ReentrantLock*.

Vamos agora criar uma nova classe chamada LockTest.

|  |
| --- |
| **public** **class** LockTest  { **public** **static** **void** main(String[] args) **throws** InterruptedException  { **final** ReentrantLock lock = **new** ReentrantLock();  **new** Thread(**new** Worker("A", lock)).start();  **new** Thread(**new** Worker("B", lock)).start();  **new** Thread(**new** Worker("C", lock)).start();  **new** Thread(**new** Worker("D", lock)).start();  **new** Thread(**new** Worker("E", lock)).start();  **new** Thread(**new** Worker("F", lock)).start();  **new** Thread(**new** Worker("G", lock)).start();  }  **static** **class** Worker **implements** Runnable  { **private** String nome;  **private** ReentrantLock rlock;    **public** Worker(String nome, ReentrantLock rlock)  { **this**.nome = nome;  **this**.rlock = rlock;  }    @Override **public** **void** run()  { **this**.rlock.lock();  **try**  { **if**(**this**.rlock.isHeldByCurrentThread())  { System.***out***.printf("\n Thread %s entrou "  + "em uma sessão crítica ", **this**.nome);    System.***out***.printf("\n %d Threads entrando na fila ",  **this**.rlock.getQueueLength());    System.***out***.printf("\n Thread %s vai trabalhar por ?"  + " segundos ", **this**.nome);    System.***out***.printf("\n Thread %s finalizou o trabalho"  + " ", **this**.nome);    System.***out***.println("\n----------------------------------------");  }  }  **finally** {**this**.rlock.unlock();}  }  }  } |

## Conditions

|  |
| --- |
| **public** **class** ListaMembros  { **private** **final** Queue<String> emails = **new** LinkedList<>();    //quando criamos uma variavel final inicializadas, todas as threads  //vem com aquele objeto imediatamente  **private** **final** ReentrantLock reentrantLock = **new** ReentrantLock();    //vamos agora criar a condição, que vem através do lock  **private** **final** Condition condition = **this**.reentrantLock.newCondition();    **private** **boolean** aberta = **true**;    **public** **int** getEmailsPendentes()  { **this**.reentrantLock.lock();  **try** {**return** emails.size();}  **finally** {**this**.reentrantLock.unlock();}  /\*  \* synchronized(emails) {return emails.size();}  \*/  }    **public** **boolean** isAberta() {**return** **this**.aberta;}    **public** String obterEmailMembro()  { String email = **null**;  **try**  { **this**.reentrantLock.lock();  //synchronized(this.emails)  //{ /\*Se os e-mails estivarem vazios, sginifica  //que vai ter que esperar\*/  **while**(emails.size() == 0)  { **if**(!**this**.aberta) {**return** **null**;}  System.***out***.println("Lista vazia. Colocando a "  + "thread: " + Thread.*currentThread*().getName()  + " em modo wait");  //this.emails.wait();    //precisamos liberar o lock após o try, e não o lock do  //emails.  **this**.condition.await();  }    /\*Este método nos retorna e remove o primeiro valor  \*desta queue de e-mails\*/  email = **this**.emails.poll();  //}  }  **catch**(InterruptedException ie) {System.***out***.println(ie.getMessage());}  **finally** {**this**.reentrantLock.unlock();}  **return** email;  }    //método para add e-mail  **public** **void** addEmailMembro(String email)  { **this**.reentrantLock.lock();  **try**  { **this**.emails.add(email);  System.***out***.println("Email adicionado na lista");  System.***out***.println("Notificando as threads que "  + "estão em espera");  **this**.condition.signalAll();  }  **finally** {**this**.reentrantLock.unlock();}  //synchronized(this.emails)  //{    /\*avisa todas as threads e fazem nas sairem do estado  \*blocked para runnable\*/  //this.emails.notifyAll();  //}  }  /\*Iremos notificar todas as listas que não fará mais  \*nenhum serviço\*/  **public** **void** fecharLista()  { System.***out***.println("Notificando todas as threads e fechando a lista");  **this**.aberta = **false**;  **this**.reentrantLock.lock();  **try** {**this**.emails.notifyAll();}  **finally** {**this**.reentrantLock.unlock();}  //synchronized(this.emails) {this.emails.notifyAll();}  }  } |
| **public** **class** EntregadorTeste  { **public** **static** **void** main(String[] args)  { ListaMembros listaMembros = **new** ListaMembros();  Thread t1 = **new** Thread(**new** Entregador(listaMembros), "Entregador 1");  Thread t2 = **new** Thread(**new** Entregador(listaMembros), "Entregador 2");    t1.start();  t2.start();    **while**(**true**)  { String email = JOptionPane.*showInputDialog*("Digite o email do membro");  **if**(email == **null** || email.isEmpty())  { listaMembros.fecharLista();  **break**;  }    listaMembros.addEmailMembro(email);  }  }  } |
| **Resultado:**  Começando o trabalho de entrega  Lista vazia. Colocando a thread: Entregador 1 em modo wait  Começando o trabalho de entrega  Lista vazia. Colocando a thread: Entregador 2 em modo wait  Email adicionado na lista  Notificando as threads que estão em espera  Email adicionado na lista  Notificando as threads que estão em espera  Email adicionado na lista  Notificando as threads que estão em espera  Email adicionado na lista  Notificando as threads que estão em espera  Notificando todas as threads e fechando a lista |

O verbo await() funciona como um substituto para os métodos wait e notify que temos no object. Só precisamos lembrar que precisamos ter o lock do objeto.

# **Concorrência: CopyOnWriteArrayList**

É uma *ArrayList* que está no pacote java.util.concurrent e que é imutável, ou seja, pelo menos no mundo java, é apenas para leitura. Uma vez que cria um objeto de uma classe *final* sem modificador de acesso, este será imutável para sempre. Se quisermos alterá-lo teremos que criar um objeto completamente novo. Este tipo é muito útil em sistemas multithread, pois um objeto imutável é por natureza thread save, sem até mesmo precisar utilizar a palavra synchronized, pois não tem como mudar e então não teremos problemas de concorrência.

|  |
| --- |
| //para tornar essa classe imutável, a palavra chave  //final é a que torna imutável se não houver SETTER.  **final** **class** Pessoa  { **private** String nome;    **public** Pessoa(String nome) {**this**.nome = nome;}  //não é deixado nenhum método SETTER, impedindo  //de torná-lo mutável  **public** String getNome() {**return** nome;}  } |

Um problema de concorrência, quando altera estado de objeto, então o conceito de imutabilidade é muito interessante.

# Coleções Concorrentes

Os iterators usados na aula anterior, o problema de usá-los é que são *weakly consystent*, ou seja, fracamente consistente.

|  |
| --- |
| **public** **class** ColecoesConcorrentesTeste  { **public** **static** **void** main(String[] args)  { //ConcurrentHashMap  //ConcurrentLinkedDeque  //ConcurrentLinkedQueue  //ConcurrentSkipListMap  //ConcurrentSkipListSet  /\*Essas classes de lists permitem multiplas threads a  \*lerem e escreverem concorrentemente, sem ter que criar  \*multiplas cópias internas numa coleção. No java 8, os  \*métodos de ConcurrentHashMap por exemplo foram adicionados  \*na interface Map\*/  Map<String, String> map = **new** HashMap<>();  ConcurrentHashMap<String, String> cmap = **new** ConcurrentHashMap<>();    //valores de map  map.put("1", "1");  map.put("1", "2");    //valores de cmap  cmap.put("1", "1");  //irá permitir inserir um valor se já não existir.  cmap.putIfAbsent("1", "2");    System.***out***.println(map.get("1"));    //lembrando que esse map é atômico, ou seja  //não sofre alterações  System.***out***.println(cmap.get("1"));  }  } |
| 2  1 |

A diferença entre ConcurrentHashMap e ConcurrentSkipListMap e ConcurrentSkipListSet é que os dois últimos são sortidos e que se quisermos criar um *sort()*, devemos primeiro usar o *Comparator* e o *ConcurrentLinkedDeque* e *ConcurrentLinkedQueue* possuem as mesmas características mas que são sincronizados, ou seja, são *thread safe*. Essas são as coleções de concorrência que temos.

# Executors

Vimos em aulas passadas que *Thread* é uma boa ferramenta mas precisamos tomar cuidado pois sua criação causa alocação de memória, pois cada uma dela ocupa espaço na memória, e tudo isso consome tempo de execução do processador. Se criarmos muitas threads, faremos com que o processador perda mais tempo ordenando essas threads, mandando entrar e sair de um estado de block, ou executando ou trocando entre outras threads do que executando tarefas entre si.

Para evitar esse problema, podemos usar o *Executors*, que é uma classe que nos permite trabalhar no mais alto nível. Então a primeira coisa que iremos fazer é desacoplar a criação da Thread e a submissão das tarefas. Com isso, o principal conceito é que, a submissão das tarefas é desacoplado da execução das tarefas. Em outras palavras, essa classe executa o método *run()* de uma forma muito parecido de uma thread, porém faz de uma forma menos tradicional, por exemplo, esse pode startar nenhuma thread no momento em que pedimos para executar, este pode executar por exemplo uma tarefa, ou pode colocar em uma fila as tarefas e processar ao longo das threads, que mantém a CPU sempre utilizada, ou pode selecionar apenas as threads que estão em execução no momento, mantendo sempre a CPU em uso. É muito interessante pois existe a possibilidade onde podemos reutilizar e limitar a criação de threads, que são as thread pools.

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);    System.***out***.println(Runtime.*getRuntime*().availableProcessors());  }  } |
| 4 |

Perceba que a maquina local possui apenas 4 processadores que podem trabalhar ao mesmo tempo. Portanto devemos sempre nos atentar com relação ao número de threads para serem executados.

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  ExecutorService eService = Executors.*newFixedThreadPool*(4);  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));    System.***out***.println("Finalizado");  }  } |
| pool-1-thread-3 iniciou 3  pool-1-thread-2 iniciou 2  Finalizado  pool-1-thread-4 iniciou 4  pool-1-thread-1 iniciou 1  pool-1-thread-4 finalizou.  pool-1-thread-1 finalizou.  pool-1-thread-3 finalizou.  pool-1-thread-2 finalizou. |

Mas o que será que acontece se inserirmos mais que podemos?

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  ExecutorService eService = Executors.*newFixedThreadPool*(4);  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(5)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(6)));    System.***out***.println("Finalizado");  }  } |
| Finalizado  pool-1-thread-2 iniciou 2  pool-1-thread-4 iniciou 4  pool-1-thread-3 iniciou 3  pool-1-thread-1 iniciou 1  pool-1-thread-2 finalizou.  pool-1-thread-2 iniciou 5  pool-1-thread-4 finalizou.  pool-1-thread-4 iniciou 6  pool-1-thread-3 finalizou.  pool-1-thread-1 finalizou.  pool-1-thread-2 finalizou.  pool-1-thread-4 finalizou. |

Perceba que quem iniciou e finalizou as threads 5 e 6 não foram os respectivos números. Então não importa se sua máquina possui um número muito limitado de processadores, sempre permanecerá o mesmo mas este *Executors* faz uso da reutilização de threads. Para finalizarmos, basta fazermos um shutdown.

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  ExecutorService eService = Executors.*newFixedThreadPool*(4);  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(5)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(6)));  /\*O problema é que, mesmo que executemos o shutdown, não quer dizer  \*que irá simplesmente parar, pois isso só irá finalizar mesmo depois que  \*todas as threads pararem.\*/  eService.shutdown();  //para resolver isso  **while**(!eService.isTerminated()) {}  //a prova disso esta nessa verificação.  System.***out***.println(eService.isTerminated());  System.***out***.println("Finalizado");  }  } |
| pool-1-thread-3 iniciou 3  pool-1-thread-1 iniciou 1  pool-1-thread-4 iniciou 4  pool-1-thread-1 finalizou.  pool-1-thread-3 finalizou.  pool-1-thread-2 finalizou.  pool-1-thread-4 finalizou.  pool-1-thread-3 iniciou 5  pool-1-thread-1 iniciou 6  pool-1-thread-3 finalizou.  pool-1-thread-1 finalizou.  true  Finalizado |

Mas se quisermos que o numero de processadores seja realmente o numero de threads a serem excutados?

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  ExecutorService eService = Executors.*newFixedThreadPool*(4);    /\*Além do ExecutorService, podemos usar isso\*/  ThreadPoolExecutor tpe = (ThreadPoolExecutor) eService;  tpe.setCorePoolSize(6);    eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(5)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(6)));  /\*O problema é que, mesmo que executemos o shutdown, não quer dizer  \*que irá simplesmente parar, pois isso só irá finalizar mesmo depois que  \*todas as threads pararem.\*/  eService.shutdown();  //para resolver isso  **while**(!eService.isTerminated()) {}  //a prova disso esta nessa verificação.  System.***out***.println(eService.isTerminated());  System.***out***.println("Finalizado");  }  } |
| pool-1-thread-1 iniciou 1  pool-1-thread-4 iniciou 4  pool-1-thread-3 iniciou 3  pool-1-thread-2 iniciou 2  pool-1-thread-6 iniciou 6  pool-1-thread-5 iniciou 5  pool-1-thread-3 finalizou.  pool-1-thread-2 finalizou.  pool-1-thread-6 finalizou.  pool-1-thread-4 finalizou.  pool-1-thread-1 finalizou.  pool-1-thread-5 finalizou.  true  Finalizado |

Outro tipo que podemos trabalhar também é a *CachedThreadPool()*, que não recebe nada como parâmetro, mas este também precisamos tomar muito cuidado ao usá-lo, pois este cria novas threads conforma for necessário, ou seja, não tem limite, cria threads conforme a necessidade. Se as threads estiverem ociosas por 60s elas são removidas no pool de threads.

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  //ExecutorService eService = Executors.newFixedThreadPool(4);  ExecutorService eService = Executors.*newCachedThreadPool*();  /\*Além do ExecutorService, podemos usar isso\*/  ThreadPoolExecutor tpe = (ThreadPoolExecutor) eService;  tpe.setCorePoolSize(Runtime.*getRuntime*().availableProcessors());    eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(5)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(6)));  /\*O problema é que, mesmo que executemos o shutdown, não quer dizer  \*que irá simplesmente parar, pois isso só irá finalizar mesmo depois que  \*todas as threads pararem.\*/  eService.shutdown();  //para resolver isso  **while**(!eService.isTerminated()) {}  //a prova disso esta nessa verificação.  System.***out***.println(eService.isTerminated());  System.***out***.println("Finalizado");  }  } |
| pool-1-thread-1 iniciou 1  pool-1-thread-4 iniciou 4  pool-1-thread-3 iniciou 3  pool-1-thread-6 iniciou 6  pool-1-thread-2 iniciou 2  pool-1-thread-5 iniciou 5  pool-1-thread-1 finalizou.  pool-1-thread-3 finalizou.  pool-1-thread-6 finalizou.  pool-1-thread-5 finalizou.  pool-1-thread-4 finalizou.  pool-1-thread-2 finalizou.  true  Finalizado |

Para finalizar, temos também o *singleThreadExecutor()*, que pelo próprio nome diz, retorna apenas uma thread para realizar todas as tarefas e as tarefas que não podem ser executadas imediatamente são colocadas em uma *Queue*, o qual usa o FIFO (*First in is First out*). A diferença desta com a *newFixedThreadPool()* é que com este não podemos fazer alteração do número de processadores, ou seja.

|  |
| --- |
| **class** ThreadTrabalhadoraExecutor **implements** Runnable  { **private** String num;    **public** ThreadTrabalhadoraExecutor(String num)  {**this**.num = num;}  @Override **public** **void** run()  { System.***out***.println(Thread.*currentThread*().getName()  + " iniciou " + num);  //parada de 3s  **try** {Thread.*sleep*(3000);}  **catch** (InterruptedException e)  {System.***out***.println(e.getMessage());}    System.***out***.println(Thread.*currentThread*().getName()  + " finalizou.");  }  }  **public** **class** ExecutorTeste  { **public** **static** **void** main(String[] args)  { /\*Método mais comum utilizado pelo framework Executors  e é construído utilizando número de threads que desejamos  executar para a tarefa que passaremos aqui no Runnable\*/  //Executors.newFixedThreadPool(4);  //System.out.println(Runtime.getRuntime().availableProcessors());    /\*Criaremos serviços de execução, ExecutorService que extende  \*de Executor\*/  //ExecutorService eService = Executors.newFixedThreadPool(4);  //ExecutorService eService = Executors.newCachedThreadPool();  ExecutorService eService = Executors.*newSingleThreadExecutor*();  /\*Além do ExecutorService, podemos usar isso\*/  /\*  \* ThreadPoolExecutor tpe = (ThreadPoolExecutor) eService;  \* tpe.setCorePoolSize(Runtime.getRuntime().availableProcessors());  \*/    eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(1)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(2)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(3)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(4)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(5)));  eService.execute(**new** ThreadTrabalhadoraExecutor(String.*valueOf*(6)));  /\*O problema é que, mesmo que executemos o shutdown, não quer dizer  \*que irá simplesmente parar, pois isso só irá finalizar mesmo depois que  \*todas as threads pararem.\*/  eService.shutdown();  //para resolver isso  **while**(!eService.isTerminated()) {}  //a prova disso esta nessa verificação.  System.***out***.println(eService.isTerminated());  System.***out***.println("Finalizado");  }  } |
| pool-1-thread-1 iniciou 1  pool-1-thread-1 finalizou.  pool-1-thread-1 iniciou 2  pool-1-thread-1 finalizou.  pool-1-thread-1 iniciou 3  pool-1-thread-1 finalizou.  pool-1-thread-1 iniciou 4  pool-1-thread-1 finalizou.  pool-1-thread-1 iniciou 5  pool-1-thread-1 finalizou.  pool-1-thread-1 iniciou 6  pool-1-thread-1 finalizou.  true  Finalizado |

Repare que foi sincronizado, ou seja, finaliza um para começar outro. Portanto, se definíssemos a quantidade de processadores, teríamos um *Exception*.

# Interface Callable

É uma interface muito similar ao *Runnable*, com a diferença que o Callable retorna um valor, enquanto que o *Runnable* é um void.

|  |
| --- |
| /\*Não é recomendável usar os métodos wait(), notify() e  \*notifyAll() quando dentro de metodo run() ou call()  \*quando estivermos utilizando algum tipo de executor,  \*pois não sabemos como é o comportamento da thread de  \*Executor. Portanto é boa pratica evitar usar esses metodos  \*que possam interferir com a execução das threads com Executors.\*/  **public** **class** CallableTeste **implements** Callable<String>  { @Override **public** String call() **throws** Exception  { /\*a partir do java 7, a diferença deste para o  Math.random é que estamos usando método sincronizado  ou seja, se estivermos em um ambiente concorrente  teremos o problema de adquirir o lock do objeto.  Já com o ThreadLocalRandom não teremos esse problema,  cada thread terá seu número aleatório sendo gerado  exclusivamente para ela\*/  **int** count = ThreadLocalRandom.*current*().nextInt(1, 11);  **for**(**int** i = 0; i < count; i++)  { System.***out***.println(Thread.*currentThread*().getName()  + " executando...");  }  **return** "Trabalho finalizado o numero aleatorio é " + count;  }  **public** **static** **void** main(String[] args)  { CallableTeste ct = **new** CallableTeste();  ExecutorService eService = Executors.*newCachedThreadPool*();  Future<String> resultFuture = eService.submit(ct);  System.***out***.println(resultFuture);  eService.shutdown();  }  } |
| java.util.concurrent.FutureTask@5c647e05  pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando... |

Perceba que o objeto resultFuture foi impresso antes de todas as threads.

|  |
| --- |
| **public** **class** CallableTeste **implements** Callable<String>  { @Override **public** String call() **throws** Exception  { /\*a partir do java 7, a diferença deste para o  Math.random é que estamos usando método sincronizado  ou seja, se estivermos em um ambiente concorrente  teremos o problema de adquirir o lock do objeto.  Já com o ThreadLocalRandom não teremos esse problema,  cada thread terá seu número aleatório sendo gerado  exclusivamente para ela\*/  **int** count = ThreadLocalRandom.*current*().nextInt(1, 11);  **for**(**int** i = 0; i < count; i++)  { System.***out***.println(Thread.*currentThread*().getName()  + " executando...");  }  **return** "Trabalho finalizado o numero aleatorio é " + count;  }  **public** **static** **void** main(String[] args)  { CallableTeste ct = **new** CallableTeste();  ExecutorService eService = Executors.*newCachedThreadPool*();  Future<String> resultFuture = eService.submit(ct);  //isso bloqueia a impressão antes das threads serem executados  **try**  { String result = resultFuture.get();  System.***out***.println(result);  }  **catch** (InterruptedException | ExecutionException e)  {System.***out***.println(e.getMessage());}  eService.shutdown();  }  } |
| pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando...  pool-1-thread-1 executando...  Trabalho finalizado o numero aleatorio é 5 |

# **Padrões de Projetos (*Design Patterns*)**

Padrão de projeto é uma maneira de implementar que vai resolver um problema específico.

## Padrão Builder

Uma classe, vamos supor Pessoa, e uma classe que fará a instanciação dessa classe e sua definição. Instanciar um objeto com parâmetros através de um construtor com todos os parâmetros, muitas vezes é complicado. Para resolver esse problema, podemos utilizar o padrão builder, que começa na própria classe, no nosso caso Pessoa, criando uma classe estática dentro dela e que vai utilizar os nomes dos atributos para inicializar.

|  |
| --- |
| **public** **class** Pessoa  { **private** String nome;  **private** String ultimoNome;  **private** String meioNome;  **private** String apelido;  **private** String nomePai;    /\* Será criano uma classe estática no lugar.  \* public Pessoa(String nome, String ultimoNome, String meioNome, String  \* apelido, String nomePai) { this.nome = nome; this.ultimoNome = ultimoNome;  \* this.meioNome = meioNome; this.apelido = apelido; this.nomePai = nomePai; }  \*/    **public** Pessoa() {}    **public** **static** **class** PessoaBuilder  { **private** String nome;  **private** String ultimoNome;  **private** String meioNome;  **private** String apelido;  **private** String nomePai;    **public** PessoaBuilder() {}    **public** PessoaBuilder nome(String nome)  { **this**.nome = nome;  **return** **this**;  }    **public** PessoaBuilder ultimoNome(String ultimoNome)  { **this**.ultimoNome = ultimoNome;  **return** **this**;  }    **public** PessoaBuilder meioNome(String meioNome)  { **this**.nome = meioNome;  **return** **this**;  }    **public** PessoaBuilder apelido(String apelido)  { **this**.apelido = apelido;  **return** **this**;  }    **public** PessoaBuilder nomePai(String nomePai)  { **this**.nomePai = nomePai;  **return** **this**;  }  }  **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  **public** String getUltimoNome() {**return** ultimoNome;}  **public** **void** setUltimoNome(String ultimoNome) {**this**.ultimoNome = ultimoNome;}  **public** String getMeioNome() {**return** meioNome;}  **public** **void** setMeioNome(String meioNome) {**this**.meioNome = meioNome;}  **public** String getApelido() {**return** apelido;}  **public** **void** setApelido(String apelido) {**this**.apelido = apelido;}  **public** String getNomePai() {**return** nomePai;}  **public** **void** setNomePai(String nomePai) {**this**.nomePai = nomePai;}  } |

A idéia é fazer uma chamada concatenada de métodos, em forma de código temos:

|  |
| --- |
| **package** br.edu.maratona.pattherns;  **public** **class** Pessoa  { **private** String nome;  **private** String ultimoNome;  **private** String meioNome;  **private** String apelido;  **private** String nomePai;    **private** Pessoa(String nome, String ultimoNome,  String meioNome, String apelido, String nomePai)  { **this**.nome = nome;  **this**.ultimoNome = ultimoNome;  **this**.meioNome = meioNome;  **this**.apelido = apelido;  **this**.nomePai = nomePai;  }      **public** **static** **class** PessoaBuilder  { **private** String nome;  **private** String ultimoNome;  **private** String meioNome;  **private** String apelido;  **private** String nomePai;    **public** PessoaBuilder() {}    **public** PessoaBuilder nome(String nome)  { **this**.nome = nome;  **return** **this**;  }    **public** PessoaBuilder ultimoNome(String ultimoNome)  { **this**.ultimoNome = ultimoNome;  **return** **this**;  }    **public** PessoaBuilder meioNome(String meioNome)  { **this**.nome = meioNome;  **return** **this**;  }    **public** PessoaBuilder apelido(String apelido)  { **this**.apelido = apelido;  **return** **this**;  }    **public** PessoaBuilder nomePai(String nomePai)  { **this**.nomePai = nomePai;  **return** **this**;  }    **public** Pessoa criatePessoa()  { **return** **new** Pessoa(nome, ultimoNome, meioNome,  apelido, nomePai);  }  }  **public** String getNome() {**return** nome;}  **public** **void** setNome(String nome) {**this**.nome = nome;}  **public** String getUltimoNome() {**return** ultimoNome;}  **public** **void** setUltimoNome(String ultimoNome) {**this**.ultimoNome = ultimoNome;}  **public** String getMeioNome() {**return** meioNome;}  **public** **void** setMeioNome(String meioNome) {**this**.meioNome = meioNome;}  **public** String getApelido() {**return** apelido;}  **public** **void** setApelido(String apelido) {**this**.apelido = apelido;}  **public** String getNomePai() {**return** nomePai;}  **public** **void** setNomePai(String nomePai) {**this**.nomePai = nomePai;}  } |
| **package** br.edu.maratona.pattherns;  **public** **class** PessoaTeste  { **public** **static** **void** main(String[] args)  { Pessoa pessoa = **new** Pessoa.PessoaBuilder()  .nome("Eduardo").apelido("Dudu")  .ultimoNome("Murakoshi").meioNome("Watanabe")  .nomePai("Osamu").criatePessoa();  }  } |

Assim podemos ver que fica muito mais claro de entender como o objeto está sendo criado, utilizando os parâmetros. Este padrão é útil quando temos vários atributos que precisam ser inicializados no construtor.

## Padrão Factory

Este padrão é algo que geralmente já existe e que simplesmente usamos em nossos projetos. Por exemplo, quando temos um método que retorna um objeto concreto, como o de conexão com banco de dados, usando o *DriverManager.getConnection()*, não importando para qual banco de dados. Isso já está implementado nas JDBCs.

Resumidamente, o padrão Factory nos ajuda a desacoplar a criação do objeto com a utilização das nossas classes. Vamos criar uma interface de Moeda e duas classes de Exemplos de moedas mundiais, como Real e Dolar e construir uma Factory que irá criar objetos do tipo Real ou Dolar.

|  |
| --- |
| public interface Moeda {String getSimbolo();} |
| public enum Pais {BRASIL, EUA} |
| public class Real implements Moeda  { @Override public String getSimbolo()  {return "R$";}  } |
| public class Dolar implements Moeda  { @Override public String getSimbolo()  {return "US$";}  } |
| public class MoedaFactory  { public static Moeda createCoin(Pais country)  { if(country.equals(Pais.BRASIL)) {return new Real();}    else if(country.equals(Pais.EUA))  {return new Dolar();}    throw new IllegalArgumentException("Não existe este país");  }  } |
| public class FactoryTeste  { public static void main(String[] args)  { Moeda moeda = MoedaFactory.createCoin(Pais.BRASIL);  Moeda moeda2 = MoedaFactory.createCoin(Pais.EUA);  System.out.println(moeda.getSimbolo());  System.out.println(moeda2.getSimbolo());  }  } |
| **Resultado:**  R$  US$ |

# **Parametrizando comportamentos**

Iremos ver algumas coisas de java 8, que [e passar comportamentos por meio de parâmetros. Sabemos que no mundo do desenvolvimento de software, os requisitos de software estarão em constantes mudanças. Portanto a parametrização dos comportamentos através da passagem de parâmetros venha a calhar em se tratando de java.

|  |
| --- |
| public class Carro  { private String nome = "Gol";  private String cor;  private int ano;    public Carro(String cor, int ano)  { this.cor = cor;  this.ano = ano;  }  public String getNome() {return nome;}  public void setNome(String nome) {this.nome = nome;}  public String getCor() {return cor;}  public void setCor(String cor) {this.cor = cor;}  public int getAno() {return ano;}  public void setAno(int ano) {this.ano = ano;}  @Override public String toString()  { return "Carro [nome=" + nome + ", cor=" + cor  + ", ano=" + ano + "]";  }  } |
| public class CarroTeste  { private static List<Carro> filtroCarrosVerdes(List<Carro> carros)  { List<Carro> result = new ArrayList<>();  for(Carro carro:carros)  { if(carro.getCor().equalsIgnoreCase("verde"))  {result.add(carro);}  }  return result;  }  public static void main(String[] args)  { List<Carro> carros = asList(new Carro("verde", 2011),  new Carro("vermelho", 1995),  new Carro("preto", 2017));  System.out.println(filtroCarrosVerdes(carros));  }  } |
| **Resultado:**  [Carro [nome=Gol, cor=verde, ano=2011]] |

No exemplo acima, vimos que um pedido de um cliente é que filtre todos os carros da lista que seja verde. Mas vamos supor que o requisito mudou e agora queremos que filtre por carros de cor vermelho. Nunca faríamos um código, semelhante a um que ja esteja pronto, e simplesmente copiar, mudando apenas alguns nomes, basta passarmos parâmetros no método.

|  |
| --- |
| public class CarroTeste  { private static List<Carro> filtroCarrosVerdes(List<Carro> carros, String cor)  { List<Carro> result = new ArrayList<>();  for(Carro carro:carros)  { if(carro.getCor().equalsIgnoreCase(cor))  {result.add(carro);}  }  return result;  }  public static void main(String[] args)  { List<Carro> carros = asList(new Carro("verde", 2011),  new Carro("vermelho", 1995),  new Carro("preto", 2017));  System.out.println(filtroCarrosVerdes(carros, "vermelho"));  }  } |
| [Carro [nome=Gol, cor=vermelho, ano=1995]] |

E conforme vai mudando de requisitos, tudo que estamos fazendo é apenas mudando nome de método e as condições. Se por um acaso agora queremos apenas fazer filtros por ano ou nome? O padrão de projeto que usamos para fazer chamadas de métodos com apenas um único nome é o *Strategy*. Uma interface que ajuda a criarmos métodos customizados com esse padrão é o Predicate. Várias funções do java usa essa interface, e temos a habilidade de pegar vários métodos de vários comportamentos e passá-los como parâmetros, ou seja, assim reduzimos o nosso código e aumentamos a sua qualidade, fazendo com que nossos métodos sejam adaptáveis às mudanças de requisitos.

# **Lambda**

Este foi criado no java 8 com o objetivo de eliminar a verbosidade, ou seja, reduzir o máximo possível de linhas de código em java. A definição mais simples de compreender é, uma representação concisa de uma função anônima que pode ser passada por parâmetro ou podemos associá-la a uma variável de referência. Esta é anônima, uma função (ou seja não é um método pois não está associado a nenhuma classe), pode ser passada por argumento para um método ou ser guardada em uma variável e é concisa pois podemos resolver problemas em poucas linhas, o que melhora muito a qualidade do seu código.

O mais importante em absorver sobre este assunto é: Quando podemos usá-la? Quando temos uma interface com apenas um método abstrato, essa interface é chamada de *FunctionalInterface*. Pegando a aula anterior, temos a Interface CarroPredicate por exemplo.

|  |
| --- |
| @FunctionalInterface  public interface CarroPredicate {boolean teste(Carro carro);} |

Existem interfaces, consideradas Funcionais, com mais de um método mas que não são obrigatórios sua sobre escrita, através do uso da palavra reservada *default*. As lambdas só podem ser usadas quando formos usar interfaces funcionais.

|  |
| --- |
| **//sem uso de lambda**  public class LambdaTeste  { public static void main(String[] args)  { CarroPredicate carroPredicate = new CarroPredicate()  { @Override public boolean teste(Carro carro)  {return carro.getCor().equals("verde");}  };  System.out.println(carroPredicate.teste(new Carro("verde", 2011)));;  }  } |
| **//com uso de lambda**  public class LambdaTeste  { public static void main(String[] args)  { CarroPredicate carroPredicate = (Carro carro) -> carro.getCor().equals("verde");  //ou CarroPredicate carroPredicate = (Carro carro) -> {return carro.getCor().equals("verde");}  System.out.println(carroPredicate.teste(new Carro("verde", 2010)));  }  } |

Também podemos fazer isso com a interface *Runnable*

|  |
| --- |
| public class LambdaTeste  { public static void main(String[] args)  { CarroPredicate carroPredicate = (Carro carro) -> carro.getCor().equals("verde");  System.out.println(carroPredicate.teste(new Carro("verde", 2011)));  Runnable run = () -> System.out.println("Dentro da run");  new Thread(run).start();  }  } |

Uma forma simples de obter obter resultado de uma lista:

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  import java.util.function.Consumer;  import java.util.function.Function;  import br.edu.maratona.comportamento.Carro;  import static java.util.Arrays.asList;  public class LambdaTeste2  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));    List<Integer> list = map(asList("Willian", "DevDojo", "facebook.com/devedojo"), (String s) -> s.length());    List<String> listCores = map(asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009)), (Carro carro) -> carro.getCor());    System.out.println(list);  System.out.println(listCores);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| DevDojo  facebook.com/devedojo  [7, 7, 21]  [preto, verde, vermelho] |

Vamos demonstrar agora uma utilidade da Interface Predicate.

|  |
| --- |
| public class LambdaTeste2  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));    List<Integer> list = map(asList("Willian", "DevDojo",  "facebook.com/devedojo"), (String s) -> s.length());    List<String> listCores = map(asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009)),  (Carro carro) -> carro.getCor());    System.out.println(list);  System.out.println(listCores);    Predicate<Integer> pares = (Integer i) -> i % 2 == 0;  System.out.println(pares.test(1000));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| DevDojo  facebook.com/devedojo  [7, 7, 21]  [preto, verde, vermelho]  true |

Para cada classe *wrapper* (Integer, Boolean, Double, etc) existe uma interface *Predicate* equivalente, ou seja, respectivamente, *IntPredicate*, *BooleanPredicate, etc.*

|  |
| --- |
| public class LambdaTeste2  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));    List<Integer> list = map(asList("Willian", "DevDojo",  "facebook.com/devedojo"), (String s) -> s.length());    List<String> listCores = map(asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009)),  (Carro carro) -> carro.getCor());    System.out.println(list);  System.out.println(listCores);    Predicate<Integer> pares = (Integer i) -> i % 2 == 0;  System.out.println(pares.test(1000));  **//ou podemos escrever dessa forma**  IntPredicate impar = (int i) -> i % 2 > 0;  System.out.println(impar.test(1001));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [7, 7, 21]  [preto, verde, vermelho]  true  true |

Existe a interface consumer, ao invés de Predicate, que possui um método do tipo void.

|  |
| --- |
| public class LambdaTeste2  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));    List<Integer> list = map(asList("Willian", "DevDojo",  "facebook.com/devedojo"), (String s) -> s.length());    List<String> listCores = map(asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009)),  (Carro carro) -> carro.getCor());    System.out.println(list);  System.out.println(listCores);    Predicate<Integer> pares = (Integer i) -> i % 2 == 0;  System.out.println(pares.test(1000));  //ou podemos escrever dessa forma  IntPredicate impar = (int i) -> i % 2 > 0;  System.out.println(impar.test(1001));    **/\*Usos de Predicate e Consumer\*/**  Predicate<String> p = (String s) -> listCores.add(s);  **//p.and("Verde");**  System.out.println(listCores);  Consumer<String> c = (String s) -> listCores.add(s);  c.accept("Amarelo");  System.out.println(listCores);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [7, 7, 21]  [preto, verde, vermelho]  true  true  [preto, verde, vermelho]  [preto, verde, vermelho, Amarelo] |

Para fechar iremos falar de uma outra interface funcional chamado Supplier, que retorna um tipo.

|  |
| --- |
| public class LambdaTeste2  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));    List<Integer> list = map(asList("Willian", "DevDojo",  "facebook.com/devedojo"), (String s) -> s.length());    List<String> listCores = map(asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009)),  (Carro carro) -> carro.getCor());    System.out.println(list);  System.out.println(listCores);    Predicate<Integer> pares = (Integer i) -> i % 2 == 0;  System.out.println(pares.test(1000));  //ou podemos escrever dessa forma  IntPredicate impar = (int i) -> i % 2 > 0;  System.out.println(impar.test(1001));    /\*Usos de Predicate e Consumer\*/  Predicate<String> p = (String s) -> listCores.add(s);  //p.and("Verde");  System.out.println(listCores);  Consumer<String> c = (String s) -> listCores.add(s);  c.accept("Amarelo");  System.out.println(listCores);  //Interface Supplier  Supplier<String> supp = () -> "Eduardo";  System.out.println(supp.get());  Supplier<Carro> supp1 = () -> new Carro("Verde", 2011);  System.out.println(supp1.get().getAno());  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [7, 7, 21]  [preto, verde, vermelho]  true  true  [preto, verde, vermelho]  [preto, verde, vermelho, Amarelo]  Eduardo  2011 |

## Lambdas com Metodos References

References methods foram criados para facilitar ainda mais a leitura das nossas lambdas. Esses métodos é bem simples de se utilizar mas é difícil de entender, pois existem 4 tipos. Vamos pegar exemplos da aula anterior.

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  (String s) -> System.out.println(s));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |

Podemos muito bem substituir a expressão (String s) -> System.out.println(s)); por:

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  **//(String s) -> System.out.println(s));**  System.out::println);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo |



Vamos por exemplo fazer um sort() na nossa classe Carro das últimas aulas e pela sua cor. Aliás criemos uma classe que possui apenas uma método estatico de comparação de dois carros pela cor.

|  |
| --- |
| public class ComparadorCarro  { public static int compareByColor(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  } |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("preto", 2011),  new Carro("verde", 2010), new Carro("vermelho", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/  Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |

Sendo assim, agora que temos nossa classe criada ComparadorCarro, vamos usá-lo substituindo nossa expressão lambda criada anteriormente.

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  **//(String s) -> System.out.println(s));**  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    **/\*Deixando nossa lista de carros em órdem por cor\*/**  **/\*Collections.sort(carros, new Comparator<Carro>()**  **{ @Override public int compare(Carro c1, Carro c2)**  **{return c1.getCor().compareToIgnoreCase(c2.getCor());}**  **}); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/**    **//Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));**    **//de forma mais simplificada**  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]] |

Iremos agora para um método com referência a um objeto específico.

|  |
| --- |
| public class ComparadorCarro  { public static int compareByColor(Carro c1, Carro c2) {return c1.getCor().compareToIgnoreCase(c2.getCor());}  public int compareByYear(Carro c1, Carro c2) {return Integer.compare(c1.getAno(), c2.getAno());}  } |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  //Collections.sort(carros, ComparadorCarro::compareByColor);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]] |

Vamos agora para uma referência ainda mais complicada, que é a referência a um método de instância de um objeto arbitrário de um tipo particular.

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);    **//referência a um método de instância de um objeto arbitrário de um tipo particular**  List<String> nomes = asList("Eduardo", "Amanda", "Willian", "Fabio");  **//nomes.sort((n1, n2) -> n1.compareToIgnoreCase(n2)); OU ENTÃO...**  nomes.sort((String::compareToIgnoreCase));  System.out.println(nomes);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Amanda, Eduardo, Fabio, Willian] |

Neste caso ainda podemos fazer com que os nomes apareçam na ordem contrária:

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);    //referência a um método de instância de um objeto arbitrário de um tipo particular  List<String> nomes = asList("Eduardo", "Amanda", "Willian", "Fabio");  //nomes.sort((n1, n2) -> n1.compareToIgnoreCase(n2)); OU ENTÃO...  /\*  nomes.sort((String::compareToIgnoreCase));  System.out.println(nomes);  \*/  nomes.sort((n1, n2) -> n2.compareToIgnoreCase(n1));  System.out.println(nomes);  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Willian, Fabio, Eduardo, Amanda] |

Continuando com essa referência, vamos supor que queremos entregar um valor String e que este método nos retorne um valor Integer.

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);    //referência a um método de instância de um objeto arbitrário de um tipo particular  List<String> nomes = asList("Eduardo", "Amanda", "Willian", "Fabio");  //nomes.sort((n1, n2) -> n1.compareToIgnoreCase(n2)); OU ENTÃO...  /\*  nomes.sort((String::compareToIgnoreCase));  System.out.println(nomes);  \*/  nomes.sort((n1, n2) -> n2.compareToIgnoreCase(n1));  System.out.println(nomes);    //Function<String, Integer> stringToInteger = (String s) -> Integer.parseInt(s); OU...  Function<String, Integer> stringToInteger = Integer::parseInt;    /\*BiPredicate<List<String>, String> contains = (lista, elemento) -> lista.contains(elemento);  OU...\*/  BiPredicate<List<String>, String> contains = List::contains;    System.out.println(stringToInteger.apply("100"));  System.out.println(contains.test(nomes, "Eduardo"));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Willian, Fabio, Eduardo, Amanda]  100  true |

Para finalizarmos, iremos para a última referência que é dos construtores. Este é bem interessante pois nos permite criar um objeto usando um método reference.

|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);    //referência a um método de instância de um objeto arbitrário de um tipo particular  List<String> nomes = asList("Eduardo", "Amanda", "Willian", "Fabio");  //nomes.sort((n1, n2) -> n1.compareToIgnoreCase(n2)); OU ENTÃO...  /\*  nomes.sort((String::compareToIgnoreCase));  System.out.println(nomes);  \*/  nomes.sort((n1, n2) -> n2.compareToIgnoreCase(n1));  System.out.println(nomes);    //Function<String, Integer> stringToInteger = (String s) -> Integer.parseInt(s); OU...  Function<String, Integer> stringToInteger = Integer::parseInt;    /\*BiPredicate<List<String>, String> contains = (lista, elemento) -> lista.contains(elemento);  OU...\*/  BiPredicate<List<String>, String> contains = List::contains;    System.out.println(stringToInteger.apply("100"));  System.out.println(contains.test(nomes, "Eduardo"));    //Referencia dos construtores.  Supplier<ComparadorCarro> comparadorCarroSupplier = ComparadorCarro::new;  System.out.println(comparadorCarroSupplier.get());  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Willian, Fabio, Eduardo, Amanda]  100  true  br.edu.maratona.lambda.ComparadorCarro@16b98e56 |

Uma excelente interface é o BiFunction para trabalharmos com esta referência.

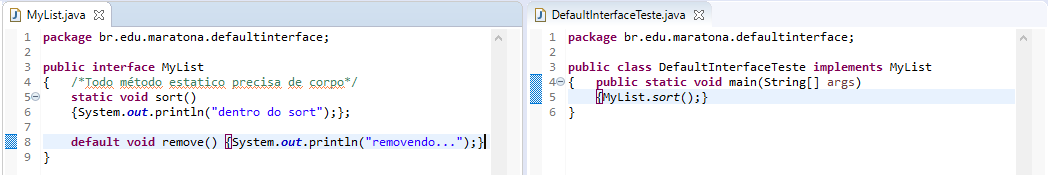
|  |
| --- |
| public class LambdaTeste3  { public static void main(String[] args)  { forEach(asList("Willian", "DevDojo", "facebook.com/devedojo"),  //(String s) -> System.out.println(s));  System.out::println);    List<Carro> carros = asList(new Carro("vermelho", 2011),  new Carro("preto", 2010), new Carro("verde", 2009));    /\*Deixando nossa lista de carros em órdem por cor\*/  /\*Collections.sort(carros, new Comparator<Carro>()  { @Override public int compare(Carro c1, Carro c2)  {return c1.getCor().compareToIgnoreCase(c2.getCor());}  }); OU PODEMOS FAZER DE UMA FORMA MAIS SIMPLIFICADA, COMO ABAIXO\*/    //Collections.sort(carros, (c1, c2) -> c1.getCor().compareToIgnoreCase(c2.getCor()));    //de forma mais simplificada: referencia para um método estatico  Collections.sort(carros, ComparadorCarro::compareByColor);  System.out.println(carros);    //Referencia a uma intancia de metodo de um objeto particular  ComparadorCarro cc = new ComparadorCarro();  Collections.sort(carros, cc::compareByYear);  System.out.println(carros);    //referência a um método de instância de um objeto arbitrário de um tipo particular  List<String> nomes = asList("Eduardo", "Amanda", "Willian", "Fabio");  //nomes.sort((n1, n2) -> n1.compareToIgnoreCase(n2)); OU ENTÃO...  /\*  nomes.sort((String::compareToIgnoreCase));  System.out.println(nomes);  \*/  nomes.sort((n1, n2) -> n2.compareToIgnoreCase(n1));  System.out.println(nomes);    //Function<String, Integer> stringToInteger = (String s) -> Integer.parseInt(s); OU...  Function<String, Integer> stringToInteger = Integer::parseInt;    /\*BiPredicate<List<String>, String> contains = (lista, elemento) -> lista.contains(elemento);  OU...\*/  BiPredicate<List<String>, String> contains = List::contains;    System.out.println(stringToInteger.apply("100"));  System.out.println(contains.test(nomes, "Eduardo"));    //Referencia dos construtores.  Supplier<ComparadorCarro> comparadorCarroSupplier = ComparadorCarro::new;  System.out.println(comparadorCarroSupplier.get());    BiFunction<String, Integer, Carro> carroBiFunction = (s, i) -> new Carro(s, i);  System.out.println(carroBiFunction.apply("Marrom", 2020));  }  public static <T> void forEach(List<T> list, Consumer<T> c)  {for(T t:list) {c.accept(t);}}    public static <T, R> List<R> map(List<T> list, Function<T, R> f)  { List<R> result = new ArrayList<>();  for(T t:list) {result.add(f.apply(t));}  return result;  }  } |
| Willian  DevDojo  facebook.com/devedojo  [Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Carro [nome=Gol, cor=verde, ano=2009], Carro [nome=Gol, cor=preto, ano=2010], Carro [nome=Gol, cor=vermelho, ano=2011]]  [Willian, Fabio, Eduardo, Amanda]  100  true  br.edu.maratona.lambda.ComparadorCarro@16b98e56  Carro [nome=Gol, cor=Marrom, ano=2020] |

# **Novas regras para interfaces: default**

A partir do java 8 podemos criar métodos com corpo de código, principalmente se for método *static*, que torna obrigatório ter um corpo de código.

|  |
| --- |
| public interface MyList  { /\*Todo método estatico precisa de corpo\*/  static void sort()  {System.out.println("dentro do sort");};  } |
| public class DefaultInterfaceTeste implements MyList  { public static void main(String[] args)  {MyList.sort();}  } |
| **Resultado:**  dentro do sort |

Nas interfaces, a partir do java 8, podemos usar métodos concretos, desde que tenham a palavra *default* acompanhada. A vantagem disso é que podemos deixar as classes que implementam tal interface não necessariamente ter todos os métodos. Perceba que na interface foi adicionado um método *default* mas este não está obrigando a classe que a implementa ter o método.



E ainda assim podemos invocar este método da seguinte maneira:

|  |
| --- |
| public class DefaultInterfaceTeste implements MyList  { public static void main(String[] args)  { MyList.sort();  new DefaultInterfaceTeste().remove();  }  } |
| **Resultado:**  dentro do sort  removendo... |

Outras regras foram adicionadas com relação a interfaces no java 8. Vejamos a seguinte situação, temos duas interfaces A e B, sendo que a B herda de A e uma classe C que implementa ambos.

|  |
| --- |
| public interface A  { default void oi()  {System.out.println("Oi A");}  } |
| public interface B extends A  { default void oi()  {System.out.println("Oi B");}  } |
| public class C implements A, B  { public static void main(String[] args)  {new C().oi();}  } |
| **Resultado:**  Oi B |

Perceba que pelo resultado, a classe C ao invocar o método oi(), a saída foi da interface B, e isso aconteceu porque no java 8, foi criada a regra de que sempre será executada o método da interface mais específica. Porém se essa classe C passasse a herdar de uma classe D que extende de A? A regra passa a ser que, a classe passará a ter prioridades com relação a interfaces.

|  |
| --- |
| public class D implements A  { @Override public void oi()  {System.out.println("Oi de A em D");}  } |
| public class C extends D implements A, B  { public static void main(String[] args)  {new C().oi();}  } |
| **Resultado:**  Oi de A em D |

# **Optional**

O *NullPointerException* é um dos erros mais comuns de acontecer no JAVA, e esta classe nova no java 8 *Optional* foi criada justamente, como uma de suas várias utilidades, para evitar esse tipo de ocorrência. Seguimos o seguinte exemplo.

|  |
| --- |
| public class OptionalTeste1  { private String nome;  public static void main(String[] args)  { OptionalTeste1 op1 = new OptionalTeste1();  System.out.println(op1.nome.toUpperCase());  }  } |
| Exception in thread "main" java.lang.NullPointerException  at br.edu.maratona.optional.OptionalTeste1.main(OptionalTeste1.java:7) |

Mesmo sem a classe *Optional* poderíamos resolver isso com apenas uma simples condição.

|  |
| --- |
| public class OptionalTeste1  { private String nome;  public static void main(String[] args)  { OptionalTeste1 op1 = new OptionalTeste1();  if(op1.nome != null) {System.out.println(op1.nome.toUpperCase());}  }  } |

Mas pense que se para cada atributo de uma classe tivéssemos que fazer essa condição, teríamos um código enorme só com essas condições.

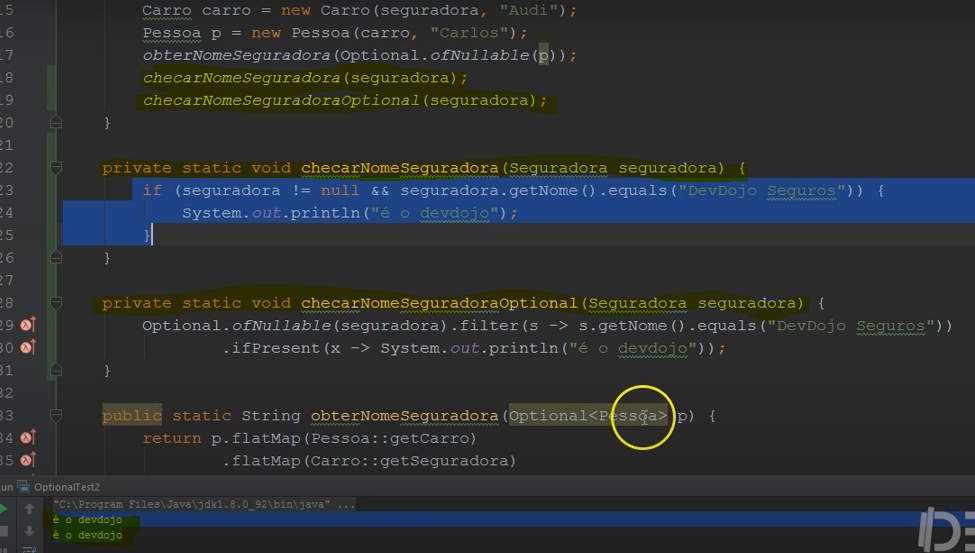
|  |
| --- |
| public class OptionalTeste1  { private String nome;  public static void main(String[] args)  { OptionalTeste1 op1 = new OptionalTeste1();  /\*A string ela pode tanto existir como não e como podemos  \*encapsular um atributo dentro de um optional? Temos 3 opções  \*1 - Quando temos certeza absoluta de que a variável que queremos  \*encapsular não será nula.\*/  Optional<String> optional = Optional.of("DevDojo");  System.out.println(optional);    /\*2 - Se quisermos inicializar que um optional seja vazio...\*/  Optional<String> optional2 = Optional.empty();  System.out.println(optional2);    /\*3 - Quando temos uma variável e não temos certeza se é nula ou não\*/  Optional<String> optional3 = Optional.ofNullable(op1.nome);  System.out.println(optional3);  }  } |
| Optional[DevDojo]  Optional.empty  Optional.empty |

Por que esse *Optional* foi criado? O java é uma linguagem que tenta sempre deixar as coisas bem esclarecida com um olhar. Numa aplicação, quando olhamos para algo, cuja assinatura possui esse Optional, significa então que aquela variável ou objeto não necessariamente precisa existir.

|  |
| --- |
| public class OptionalTeste1  { private String nome;  public static void main(String[] args)  { OptionalTeste1 op1 = new OptionalTeste1();  /\*A string ela pode tanto existir como não e como podemos  \*encapsular um atributo dentro de um optional? Temos 3 opções  \*1 - Quando temos certeza absoluta de que a variável que queremos  \*encapsular não será nula.\*/  Optional<String> optional = Optional.empty();  //System.out.println(optional);    /\*2 - Se quisermos inicializar que um optional seja vazio...\*/  Optional<String> optional2 = Optional.empty();  System.out.println(optional2);    /\*3 - Quando temos uma variável e não temos certeza se é nula ou não\*/  Optional<String> optional3 = Optional.ofNullable(op1.nome);  System.out.println(optional3);    /\*Podemos fazer o seguinte, quando quisermos desprezar valores  \*nulos ou vazios\*/  System.out.println(optional.orElse("Hello World"));  System.out.println(optional2.orElse("Eduardo"));  System.out.println(optional3.orElse("Murakoshi"));  }  } |
| Optional.empty  Optional.empty  Hello World  Eduardo  Murakoshi |

Supondo agora a seguintes situação:

|  |
| --- |
| public class Seguradora  { private String nome;  public Seguradora() {}  public Seguradora(String nome) {this.nome = nome;}    public Optional<String> getNome() {return Optional.ofNullable(nome);}  public void setNome(String nome) {this.nome = nome;}  } |
| public class SeguradoraOptionalTeste  { public static void main(String[] args)  { Seguradora seg = new Seguradora("Hello");  System.out.println(obterNomeSeguradoraOptional(seg));  }  private static String obterSeguradora(Seguradora seg)  { String nome = null;  if(seg.getNome() != null) {  //nome = seg.getNome();  }  return nome;  }    public static String obterNomeSeguradoraOptional(Seguradora seg)  { /\*Esse flatMap(), diferente do map(), extrai diretamente a String  do getNome() e podemos usar o orElse()\*/  System.out.println(Optional.ofNullable(seg).flatMap(  Seguradora::getNome).orElse("Vazio"));  return "";  }  } |
| **Resultado:**  Hello |



# **Stream**

Esse recurso deu ao java uma capacidade que não existia nas versões anteriores, obtermos de forma mais simples e por isso era uma linguagem muito criticada, para trabalharmos diretamente com as coleções quando estivéssemos, por exemplo, trabalhando com banco de dados diretamente.

Imagine que temos uma lista de pessoas, onde temos nome, idade, genero e imagine que queremos fazer o seguinte, que essa coleção de pessoas, tenhamos todas as pessoas menores de 25 anos queiramos ordená-las por nome e em seguinda queiramos extrair os 3 primeiros nomes.

O stream foi criado, primeiro a API *Collection* é a mais utilizado da java. Através dela podemos manipular os dados que estão dentro dessas coleções de uma forma mais declarativa, evitando de ficarmos criando um monte de IF-ELSEs e loopings FOR e ainda com melhor performance.

Uma definição resumida de stream é uma sequência de elementos de uma origem que suporta operação de processamento de dados. Uma sequência de elementos vai transformar os streans que pode vir de uma coleção ou de um *array* ou IO e vai providenciar uma interface para se trabalhar com essa sequência de elementos. A diferença entre stream coleções é que com estes estamos falando de dados, já stream estamos falando sobre processamento, ou seja, computação pois iremos computar esses dados.

|  |
| --- |
| import java.util.List;  import static java.util.Arrays.asList;  public class Pessoa  { private String nome;  private int idade;  private double salario;    public Pessoa(String nome, int idade, double salario)  { this.nome = nome;  this.idade = idade;  this.salario = salario;  }  public String getNome() {return nome;}  public void setNome(String nome) {this.nome = nome;}  public int getIdade() {return idade;}  public void setIdade(int idade) {this.idade = idade;}  public double getSalario() {return salario;}  public void setSalario(double salario) {this.salario = salario;}    public static List<Pessoa> bancoDePessoas()  { return asList(new Pessoa("Eduardo Murakoshi", 37, 4000.0),  new Pessoa("Rodrigo Murakoshi", 41, 20000.0),  new Pessoa("Osamu Murakoshi", 73, 10000.0),  new Pessoa("Marie Murakoshi", 67, 10000.0),  new Pessoa("Camila da Costa", 31, 4000.0));  }  } |
| import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  import java.util.List;  import java.util.stream.Collectors;  import br.edu.maratona.stream.classes.Pessoa;  **/\*Aqui queremos pegar as 3 primeiras pessoas, ordenadas, com menos de 50 anos\*/**  public class StreamTeste  { public static void main(String[] args)  { **//com java 7**  List<Pessoa> pessoas = Pessoa.bancoDePessoas();  **//Organizando em ordem**  Collections.sort(pessoas, new Comparator<Pessoa>()  { @Override public int compare(Pessoa p1, Pessoa p2)  {return p1.getNome().compareToIgnoreCase(p2.getNome());}  });  **//limitando a 3 com menos de 50 anos**  List<String> nomes = new ArrayList<>();  for(Pessoa pessoa:pessoas)  { if(pessoa.getIdade() < 50)  { nomes.add(pessoa.getNome());  if(nomes.size() >= 3) break;  }  }  System.out.println(nomes);    **//com java 8, usando stream**  List<String> nomes2 = pessoas.stream()  **//filtrando pela idade < 50**  .filter(p->p.getIdade() < 50)  **//ordenando por nome**  .sorted(Comparator.comparing(Pessoa::getNome))  **//limitado a 3 pessoas**  .limit(3)  **//coletando todos pelo nome em uma lista**  .map(Pessoa::getNome).collect(Collectors.toList());    System.out.println(nomes2);  }  } |
| **Resultado:**  [Camila da Costa, Eduardo Murakoshi, Rodrigo Murakoshi]  [Camila da Costa, Eduardo Murakoshi, Rodrigo Murakoshi] |

Resumindo, a diferença de se trabalhar com *Collections* e Streams é que com o primeiro é necessário fazer filtros a partir de Iterações, enquanto que Streams não precisamos nos preocupar com iteração, pois a Stream mesmo internamente já faz o tratamento.

Precisamos agora saber que existem dois tipos de Streams: Intermediate e Terminal. A diferença entre os dois é que o primeiro retorna outro Stream, o que nos permite encadear vários métodos, como os ultimos visto anteriormente para filtros. O tipo Terminal retorna um valor que não é um Stream, podendo retornar uma lissta, void, um valor numérico, String, etc. No caso do exemplo anterior, o método *collect()* é o único exemplo de Terminal.

É muito importante saber que os métodos intermediários não vão fazer nenhum tipo de processamento até que uma operação terminal seja invocada naquele Stream, pois este encadeamento é do tipo laze. Isso acontece porque as operações Intermediate podem ser mescladas e processadas em um único passo peça operação terminal. Iremos demonstrar mais dois métodos do Stream que podem ser muito úteis, mas antes vamos adicionar os métodos equals() e hashcode(), **filtrando apenas pelo nome**, na classe Pessoa e inserir mais um objeto desse tipo com mesmo nome porém com valores de idade e salario diferentes.

|  |
| --- |
| public class Pessoa  { private String nome;  private int idade;  private double salario;    public Pessoa(String nome, int idade, double salario)  { this.nome = nome;  this.idade = idade;  this.salario = salario;  }  public String getNome() {return nome;}  public void setNome(String nome) {this.nome = nome;}  public int getIdade() {return idade;}  public void setIdade(int idade) {this.idade = idade;}  public double getSalario() {return salario;}  public void setSalario(double salario) {this.salario = salario;}    public static List<Pessoa> bancoDePessoas()  { return asList(new Pessoa("Eduardo Murakoshi", 37, 4000.0),    new Pessoa("Rodrigo Murakoshi", 41, 20000.0),    new Pessoa("Osamu Murakoshi", 73, 10000.0),    new Pessoa("Marie Murakoshi", 67, 10000.0),    new Pessoa("Camila da Costa", 31, 4000.0),    new Pessoa("Camila da Costa", 32, 5000.0));  }  @Override public int hashCode()  { final int prime = 31;  int result = 1;  result = prime \* result + ((nome == null) ? 0 : nome.hashCode());  return result;  }  @Override public boolean equals(Object obj)  { if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  Pessoa other = (Pessoa) obj;  if (nome == null) {  if (other.nome != null)  return false;  } else if (!nome.equals(other.nome))  return false;  return true;  }  } |
| public class StreamTeste  { public static void main(String[] args)  { //com java 7  List<Pessoa> pessoas = Pessoa.bancoDePessoas();  //Organizando em ordem  Collections.sort(pessoas, new Comparator<Pessoa>()  { @Override public int compare(Pessoa p1, Pessoa p2)  {return p1.getNome().compareToIgnoreCase(p2.getNome());}  });  //limitando a 3 com menos de 50 anos  List<String> nomes = new ArrayList<>();  for(Pessoa pessoa:pessoas)  { if(pessoa.getIdade() < 50)  { nomes.add(pessoa.getNome());  if(nomes.size() >= 4) break;  }  }  System.out.println(nomes);    //com java 8, usando stream  List<String> nomes2 = pessoas.stream()  //filtrando pela idade < 50  .filter(p->p.getIdade() < 50)  //ordenando por nome  .sorted(Comparator.comparing(Pessoa::getNome))  //limitado a 3 pessoas  .limit(4)  //coletando todos pelo nome em uma lista  .map(Pessoa::getNome).collect(Collectors.toList());    System.out.println(nomes2);    //com java 8, usando stream  Long qtd = pessoas.stream()  **//irá fazer uma distinção de acordo com o filtro.**  .distinct()  .filter(p->p.getIdade() < 50)  .sorted(Comparator.comparing(Pessoa::getNome))  .map(Pessoa::getNome)  **/\*outro método, que equivale ao size() do collections, com a diferença**  **\*este retorna um Long e não in inteiro\*/**  .count();    System.out.println(qtd);    **//Para fechar, um método do tipo Terminal**  pessoas.stream().forEach(System.out::println);  }  } |
| **Resultado:**  [Camila da Costa, Camila da Costa, Eduardo Murakoshi]  [Camila da Costa, Camila da Costa, Eduardo Murakoshi]  3  br.edu.maratona.stream.classes.Pessoa@91ab5c41  br.edu.maratona.stream.classes.Pessoa@91ab5c41  br.edu.maratona.stream.classes.Pessoa@1dccfbca  br.edu.maratona.stream.classes.Pessoa@8e4a1f92  br.edu.maratona.stream.classes.Pessoa@b9f7743d  br.edu.maratona.stream.classes.Pessoa@a453d53e |

Como vimos que o método *limit(NUMERO\_INT)* faz com que imponha um limite de valores de um Stream. Mas se quisermos que a Stream despreze um valor?

|  |
| --- |
| **public** **class** StreamTeste  { **public** **static** **void** main(String[] args)  { //com java 7  List<Pessoa> pessoas = Pessoa.*bancoDePessoas*();  //Organizando em ordem  Collections.*sort*(pessoas, **new** Comparator<Pessoa>()  { @Override **public** **int** compare(Pessoa p1, Pessoa p2)  {**return** p1.getNome().compareToIgnoreCase(p2.getNome());}  });  //limitando a 3 com menos de 50 anos  List<String> nomes = **new** ArrayList<>();  **for**(Pessoa pessoa:pessoas)  { **if**(pessoa.getIdade() < 50)  { nomes.add(pessoa.getNome());  **if**(nomes.size() >= 4) **break**;  }  }  System.***out***.println(nomes);    //com java 8, usando stream  List<String> nomes2 = pessoas.stream()  //filtrando pela idade < 50  .filter(p->p.getIdade() < 50)  //ordenando por nome  .sorted(Comparator.*comparing*(Pessoa::getNome))  //limitado a 3 pessoas  .limit(4)  //pulando um valor pelo index  .skip(1)  //coletando todos pelo nome em uma lista  .map(Pessoa::getNome).collect(Collectors.*toList*());    System.***out***.println(nomes2);    //com java 8, usando stream  Long qtd = pessoas.stream()  //irá fazer uma distinção de acordo com o filtro.  .distinct()  //filtrando pela idade < 50  .filter(p->p.getIdade() < 50)  //ordenando por nome  .sorted(Comparator.*comparing*(Pessoa::getNome))  //coletando todos pelo nome em uma lista  .map(Pessoa::getNome)  /\*outro método, que equivale ao size() do collections, com a diferença  \*este retorna um Long e não in inteiro\*/  .count();    System.***out***.println(qtd);    //Para fechar, um método do tipo Terminal  pessoas.stream().forEach(p->System.***out***.println(p.getNome()));  }  } |
| [Camila da Costa, Camila da Costa, Eduardo Murakoshi, Rodrigo Murakoshi]  [Camila da Costa, Eduardo Murakoshi, Rodrigo Murakoshi]  3  Camila da Costa  Camila da Costa  Eduardo Murakoshi  Marie Murakoshi  Osamu Murakoshi  Rodrigo Murakoshi |

Veja que pelo resultado, o mesmo menosprezou a primeira Camila da Stream anterior. Assim como o *Optional*, os Streams também tem os métodos *map()* e *flatMap()*. No *Optional*, assim quando você não quer um determinado resultado usamos o *flatMap()* e quando quisermos um outro, usamos o *map()*. Já nos Streams, usamos o *map()*, este aplica a função para cada elemento e retornamos valores coletados em um novo *Stream*. Já o *flatMap()* é usado quando temos vários valores e queremos concatenar e literalmente achatar esses valores em um valor só.

|  |
| --- |
| **public** **class** StreamTeste2  { **public** **static** **void** main(String[] args)  { List<List<String>> nomes = **new** ArrayList<>();  nomes.add(*asList*("Eduardo", "Osamu", "Rodrigo",  "Camila", "Marie"));  System.***out***.println(nomes);  //nomes.stream().flatMap(Collection::stream); OU  //nomes.stream().flatMap(s->s.stream());  Stream<String> stringStream = nomes.stream().flatMap(Collection::stream);  stringStream.forEach(System.***out***::println);  }  } |
| **Resultado:**  [[Eduardo, Osamu, Rodrigo, Camila, Marie]]  Eduardo  Osamu  Rodrigo  Camila  Marie |

Mas se quisermos que esses nomes sejam representados como uma lista?

|  |
| --- |
| **public** **class** StreamTeste2  { **public** **static** **void** main(String[] args)  { List<List<String>> nomes = **new** ArrayList<>();  nomes.add(*asList*("Eduardo", "Osamu", "Rodrigo",  "Camila", "Marie"));  System.***out***.println(nomes);  //nomes.stream().flatMap(Collection::stream); OU  //nomes.stream().flatMap(s->s.stream());  List<String> stringStream = nomes.stream()  .flatMap(Collection::stream).collect(Collectors.*toList*());  System.***out***.println(stringStream);  }  } |
| Resultado:  [[Eduardo, Osamu, Rodrigo, Camila, Marie]]  [Eduardo, Osamu, Rodrigo, Camila, Marie] |

Vamos brincar mais um pouco para sentir a diferença dos métodos *map()* e *flatMap()*.

|  |
| --- |
| **public** **class** StreamTeste2  { **public** **static** **void** main(String[] args)  { List<List<String>> nomes = **new** ArrayList<>();  nomes.add(*asList*("Eduardo", "Osamu", "Rodrigo",  "Camila", "Marie"));  System.***out***.println(nomes);  //nomes.stream().flatMap(Collection::stream); OU  //nomes.stream().flatMap(s->s.stream());  List<String> stringStream = nomes.stream()  .flatMap(Collection::stream).collect(Collectors.*toList*());  System.***out***.println(stringStream);    /\*Exemplificando outros métodos\*/  List<String> palavras = *asList*("Ola", "Eduardo");  //separação de cada caractere da String com ','  String[] split = palavras.get(0).split("");  System.***out***.println(Arrays.*toString*(split));  //usando Stream  List<String[]> collect1 = palavras.stream().map(p->p.split(""))  .collect(Collectors.*toList*());  Stream<String> stream = Arrays.*stream*((String[])palavras.toArray());  //até aqui temos um Stream<Sting[]>  List<String> collect2 = palavras.stream().map(p->p.split(""))  //aqui teremos um Stream<String>  .flatMap(Arrays::*stream*).collect(Collectors.*toList*());  System.***out***.println(collect2);  }  } |
| **Resultado:**  [[Eduardo, Osamu, Rodrigo, Camila, Marie]]  [Eduardo, Osamu, Rodrigo, Camila, Marie]  [O, l, a]  [O, l, a, E, d, u, a, r, d, o] |

## Metodos *finding()* e *matching()*

Os Streams ainda possuem métodos que facilitam nossas vidas. Na verdade esses nomes no títulos são conjuntos de métodos com nome de *find* e *match*. Vejamos alguns na prática:

|  |
| --- |
| **public** **class** StreamTeste3  { **public** **static** **void** main(String[] args)  { List<Pessoa> pessoas = Pessoa.*bancoDePessoas*();  System.***out***.println(pessoas.stream().anyMatch(p->p.getSalario() > 4000));  }  } |
| **Resultado:**  true |

Traduzindo resumidamente, este método *anyMatch()* encontra dentro da stream se existe algum elemento com um determinado atributo e valor. Agora vejamos outro método interessante que é o *allMatch()*:

|  |
| --- |
| **public** **class** StreamTeste3  { **public** **static** **void** main(String[] args)  { List<Pessoa> pessoas = Pessoa.*bancoDePessoas*();  //anyMatch()  System.***out***.println(pessoas.stream().anyMatch(p->p.getSalario() > 4000));    //allMatch()  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20));  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20 &&  p.getIdade() < 40));  }  } |
| **Resultado:**  true  true  false |

Este segundo método da família *matching* ele verifica se todos os elementos da streamo obedece a condição definida, que no caso, o primeiro verificou se todos possui uma idade acima de 20 anos e no segundo se todos possuem a idade no intervalo de ]20, 40[.

Da família *finding*, existe o *findAny()* e *findFirst()*, que fazem praticamente a mesma coisa, com a diferença que o primeiro é relativamente mais rápido, dependendo do tamanho da sua stream, pois vai encontrar um elemento de forma aleatória e não ordenado, e o segundo irá buscar o primeiro elemento que obedece a mesma condição.

|  |
| --- |
| **public** **class** StreamTeste3  { **public** **static** **void** main(String[] args)  { List<Pessoa> pessoas = Pessoa.*bancoDePessoas*();  //anyMatch()  System.***out***.println(pessoas.stream().anyMatch(p->p.getSalario() > 4000));    //allMatch()  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20));  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20 &&  p.getIdade() < 40));    //noneMatch(). Este é equivalente ao allMatch() porém negado  System.***out***.println(pessoas.stream().noneMatch(p->p.getIdade() > 20));  System.***out***.println(pessoas.stream().noneMatch(p->p.getIdade() < 20));    //findAny()  System.***out***.println(pessoas.stream().filter(p->p.getIdade() < 40)  //encontra algum registro, o primeiro que encontrar  .findAny()  //retorna em forma de objeto  .get()  //identifica o objeto.  .toString());  /\*este mesmo processo poderia ser feito usando ifPresent() ao  \*invés de get\*/  pessoas.stream().filter(p->p.getIdade() < 40).findAny()  .ifPresent(p->System.***out***.println(p.toString()));  //findFirst()  pessoas.stream().filter(p->p.getIdade() < 40).findFirst()  .ifPresent(p->System.***out***.println(p.toString()));  System.***out***.println("----------------------------------------------------------");  //se quisermos encontrar todos  pessoas.stream().filter(p->p.getIdade() < 40).forEach(  p->System.***out***.println(p.toString()));  }  } |
| **Resultado:**  true  true  false  false  true  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0]  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0] //– **findAny()**  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0] //– **findFirst()**  ----------------------------------------------------------  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0]  Pessoa [nome=Camila da Costa, idade=31, salario=4000.0]  Pessoa [nome=Camila da Costa, idade=32, salario=5000.0] |

Veremos agora alguns métodos extras, que na verdade não fazem parte de nenhuma dessas famílias. A idéia é tentar entender o que foi feito e o que esses métodos das Streams fazem.

|  |
| --- |
| **public** **class** StreamTeste3  { **public** **static** **void** main(String[] args)  { List<Pessoa> pessoas = Pessoa.*bancoDePessoas*();  //anyMatch()  System.***out***.println(pessoas.stream().anyMatch(p->p.getSalario() > 4000));    //allMatch()  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20));  System.***out***.println(pessoas.stream().allMatch(p->p.getIdade() > 20 &&  p.getIdade() < 40));    //noneMatch(). Este é equivalente ao allMatch() porém negado  System.***out***.println(pessoas.stream().noneMatch(p->p.getIdade() > 20));  System.***out***.println(pessoas.stream().noneMatch(p->p.getIdade() < 20));    //findAny()  System.***out***.println(pessoas.stream().filter(p->p.getIdade() < 40)  //encontra algum registro, o primeiro que encontrar  .findAny()  //retorna em forma de objeto  .get()  //identifica o objeto.  .toString());  /\*este mesmo processo poderia ser feito usando ifPresent() ao  \*invés de get\*/  pessoas.stream().filter(p->p.getIdade() < 40).findAny()  .ifPresent(p->System.***out***.println(p.toString()));  //findFirst()  pessoas.stream().filter(p->p.getIdade() < 40).findFirst()  .ifPresent(p->System.***out***.println(p.toString()));  System.***out***.println("-------------------------EXTRAS-----------------------------");  //se quisermos encontrar todos  pessoas.stream().filter(p->p.getIdade() < 40).forEach(  p->System.***out***.println(p.toString()));  System.***out***.println("------------------------------------------------------------");  //usando métodos do tipo sort  pessoas.stream().filter(p->p.getIdade() < 40)  .sorted(Comparator.*comparing*(Pessoa::getNome))  .findFirst().ifPresent(p->System.***out***.println(p.toString()));  System.***out***.println("------------------------------------------------------------");  pessoas.stream().filter(p->p.getIdade() < 40)  .sorted(Comparator.*comparing*(Pessoa::getNome).reversed())  .findFirst().ifPresent(p->System.***out***.println(p.toString()));  }  } |
| **Resultado:**  -------------------------EXTRAS-----------------------------  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0]  Pessoa [nome=Camila da Costa, idade=31, salario=4000.0]  Pessoa [nome=Camila da Costa, idade=32, salario=5000.0]  ------------------------------------------------------------  Pessoa [nome=Camila da Costa, idade=31, salario=4000.0]  ------------------------reversed()------------------------------------  Pessoa [nome=Eduardo Murakoshi, idade=37, salario=4000.0] |

## Stream especializados

### Reduce

Isso é basicamente pegarmos um Streaming e transformá-lo em um resultado final e único. Supondo que temos um stream de números e queremos pegar a soma de todos esses números.

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { Stream<Integer> numeros = *asList*(1,2,3,4,5,6,7,8,9).stream();  Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  numeros.reduce((x, y) -> x + y);  System.***out***.println(reduce.get());  }  } |
| **Resultado:**  45 |

Podemos fazer de uma outra forma, menosprezando o retorno do tipo *Optional*, porém quando temos uma Stream, só podemos executá-la apenas uma vez, caso contrário teremos uma Exceção. Caso queira comprovar isso, realize um teste junto com a Stream já criada.

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { Stream<Integer> numeros = *asList*(1,2,3,4,5,6,7,8,9).stream();  //Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  // numeros.reduce((x, y) -> x + y);  //System.out.println(reduce.get());    /\*Podemos também fazer da seguinte maneira, porém temos que comentar a stream  \*de cima, pois quando temos uma stream, só podemos operá-lo apenas uma vez\*/  Integer soma = numeros.reduce(0, (x, y) -> x + y);  System.***out***.println(soma);  }  } |
| **Resultado:**  45 |

Podemos ainda fazer de uma forma mais reduzida e ainda intuitiva e abstrata, porém com retorno de um Optional novamente.

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { Stream<Integer> numeros = *asList*(1,2,3,4,5,6,7,8,9).stream();  Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  numeros.reduce((x, y) -> x + y);  System.***out***.println(reduce.get());  /\*Podemos também fazer da seguinte maneira, porém temos que comentar a stream  \*de cima, pois quando temos uma stream, só podemos operá-lo apenas uma vez, ou  \*podemos também simplesmente repetir a mesma stream após seu uso\*/  numeros = *asList*(1,2,3,4,5,6,7,8,9).stream();  Integer soma = numeros.reduce(0, (x, y) -> x + y);  System.***out***.println(soma);    numeros = *asList*(1,2,3,4,5,6,7,8,9).stream();  Optional<Integer> soma2 = numeros.reduce(Integer::*sum*);  System.***out***.println(soma2.get());  }  } |
| **Resultado:**  45  45  45 |

Vamos testar a multiplicação e obter o produto.

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { //Stream<Integer> numeros = getStream();  Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  *getStream*().reduce((x, y) -> x + y);  System.***out***.println(reduce.get());  /\*Podemos também fazer da seguinte maneira, porém temos que comentar a stream  \*de cima, pois quando temos uma stream, só podemos operá-lo apenas uma vez, ou  \*podemos também simplesmente repetir a mesma stream após seu uso\*/  //numeros = getStream();  Integer soma = *getStream*().reduce(0, (x, y) -> x + y);  System.***out***.println(soma);    //numeros = getStream();  Optional<Integer> soma2 = *getStream*().reduce(Integer::*sum*);  System.***out***.println(soma2.get());    Optional<Integer> produto = *getStream*().reduce((x,y)->x\*y);  System.***out***.println(produto.get());  }  **private** **static** Stream<Integer> getStream() {**return** *asList*(1,2,3,4,5,6,7,8,9).stream();}  } |

Supondo agora que queremos obter o valor máximo de uma Stream, existem duas maneiras:

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { //Stream<Integer> numeros = getStream();  Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  *getStream*().reduce((x, y) -> x + y);  System.***out***.println(reduce.get());  /\*Podemos também fazer da seguinte maneira, porém temos que comentar a stream  \*de cima, pois quando temos uma stream, só podemos operá-lo apenas uma vez, ou  \*podemos também simplesmente repetir a mesma stream após seu uso\*/  //numeros = getStream();  Integer soma = *getStream*().reduce(0, (x, y) -> x + y);  System.***out***.println(soma);    //numeros = getStream();  Optional<Integer> soma2 = *getStream*().reduce(Integer::*sum*);  System.***out***.println(soma2.get());    Optional<Integer> produto = *getStream*().reduce((x,y)->x\*y);  System.***out***.println(produto.get());    /\*Duas maneiras de obter o valor máximo:  \*1 - usando operador ternário\*/  Optional<Integer> max = *getStream*().reduce((x,y)-> x > y ? x:y);  System.***out***.println(max.get());    //2 - Usando métodos da classe Integer mesmo  Optional<Integer> max2 = *getStream*().reduce(Integer::*max*);  System.***out***.println(max.get());  }  **private** **static** Stream<Integer> getStream() {**return** *asList*(1,2,3,4,5,6,7,8,9).stream();}  } |
| **Resultado:**  45  45  45  362880  9  9 |

Como um extra, iremos fazer um outro exemplo:

|  |
| --- |
| **import** **static** java.util.Arrays.*asList*;  **import** java.util.Optional;  **import** java.util.stream.Stream;  **import** br.edu.maratona.stream.classes.Pessoa;  **public** **class** StreamTeste4Reduce  { **public** **static** **void** main(String[] args)  { //Stream<Integer> numeros = getStream();  Optional<Integer> reduce =  /\*isso retorna um Optional<Integer>, pois não tem como ter certeza  \*absoluta de que vai ter algum valor para fazermos o reduce da stream\*/  *getStream*().reduce((x, y) -> x + y);  System.***out***.println(reduce.get());  /\*Podemos também fazer da seguinte maneira, porém temos que comentar a stream  \*de cima, pois quando temos uma stream, só podemos operá-lo apenas uma vez, ou  \*podemos também simplesmente repetir a mesma stream após seu uso\*/  //numeros = getStream();  Integer soma = *getStream*().reduce(0, (x, y) -> x + y);  System.***out***.println(soma);    //numeros = getStream();  Optional<Integer> soma2 = *getStream*().reduce(Integer::*sum*);  System.***out***.println(soma2.get());    Optional<Integer> produto = *getStream*().reduce((x,y)->x\*y);  System.***out***.println(produto.get());    /\*Duas maneiras de obter o valor máximo:  \*1 - usando operador ternário\*/  Optional<Integer> max = *getStream*().reduce((x,y)-> x > y ? x:y);  System.***out***.println(max.get());    //2 - Usando métodos da classe Integer mesmo  Optional<Integer> max2 = *getStream*().reduce(Integer::*max*);  System.***out***.println(max2.get());    /\*vamos através da classe de pessoas, achar todas as pessoas com salário  acima de 4000.\*/  Stream<Pessoa> streamPessoas = Pessoa.*bancoDePessoas*().stream();  Optional<Double> salarioSoma = streamPessoas.filter(p -> p.getSalario() > 4000)  .map(Pessoa::getSalario)  .reduce(Double::*sum*);    System.***out***.println(salarioSoma.get());  }  **private** **static** Stream<Integer> getStream() {**return** *asList*(1,2,3,4,5,6,7,8,9).stream();}  } |
| **Resultado:**  45  45  45  362880  9  9  45000.0 |