

# Linguagem de Programação Java

## Agenda:

- O que é Java
- Tipos de Linguagens
- Imports
- Primitivo vs Referência
- Anotações e Enums
- Casting vs Promoção
- Estruturas de Condição e Repetição
- Interfaces
- Collections API
- I/O API
- Tratamento de Exceções





# O que é Java

# Java

Java é uma linguagem de programação de **alto nível** conhecida por sua portabilidade, segurança e facilidade de uso. Java é **orientada a objetos**, o que promove **reutilização** de código e modelagem do mundo real. A segurança é uma prioridade, com a JVM isolando o código. A linguagem possui uma vasta biblioteca padrão. Ela é usada em **várias aplicações**, desde desenvolvimento web até aplicativos móveis, com uma comunidade de desenvolvedores ativa.



# Tipos de Linguagens

# Linguagens estruturadas

A **linguagem estruturada**, é uma forma de escrever os códigos **sem encapsular dados**, ou seja, em qualquer parte do código é possível utilizar um dado guardado em uma variável, sem necessidade de planejamento prévio.

Não há **abstrações**, assim sendo, todos os tipos de códigos estão nos mesmos arquivos.

```
# include <stdio.h>

struct Pessoa
{
    char nome[64]; // vetor de 64 chars para o nome
    unsigned short int idade;
    char cpf[13];
};

int main()
{
    // declaração da variável "exemplo"
    struct Pessoa exemplo = {"Fulano", 16, "00.000.000-00"};
    printf("Nome: %s\n", exemplo.nome);
    printf("Idade: %hu\n", exemplo.idade);
    printf("CPF: %s\n", exemplo.cpf);

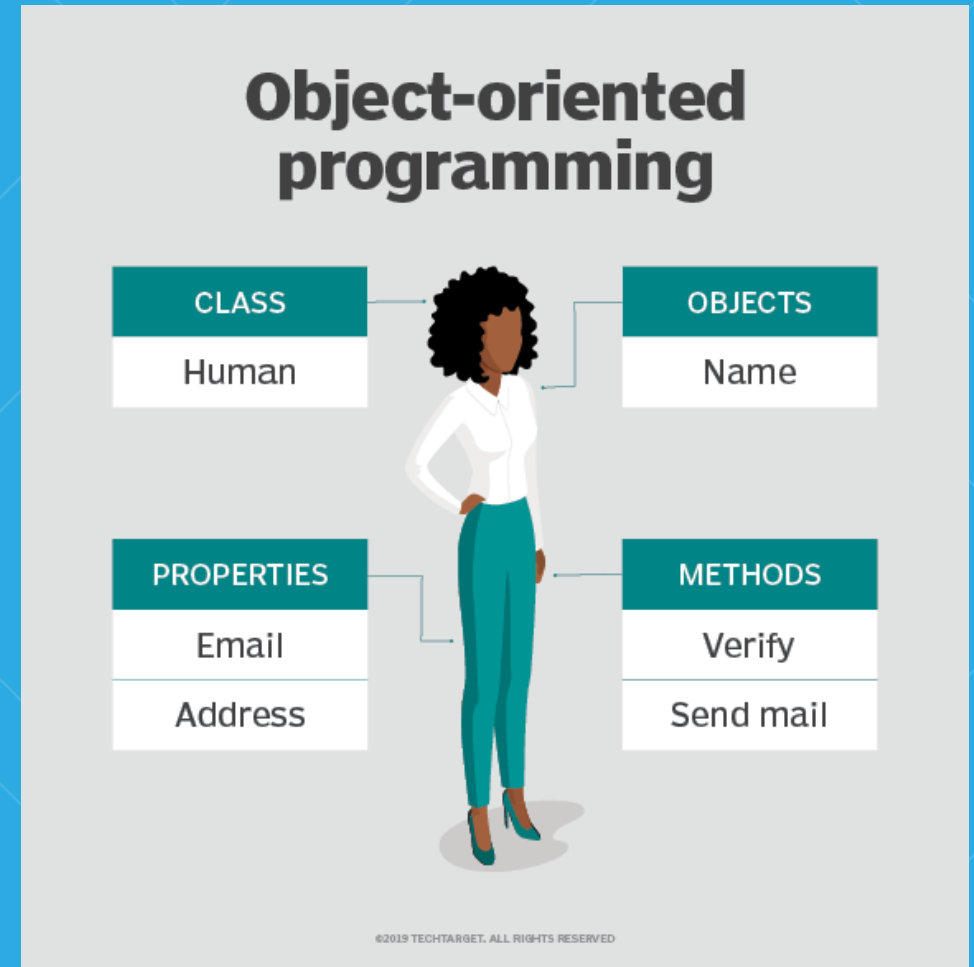
    getchar();
    return 0;
}
```

# Linguagens orientadas a objetos

A **linguagem orientada a objetos** se dá ao fato de a escrita do código se propor a traduzir objetos do mundo real para código. Com o uso de **classes**, podemos criar **objetos** que **encapsulam dados** e **abstraem complexidades** desnecessárias ao mundo exterior a eles.

Dentre as vantagens de uso:

- Reutilização de código
- Melhor legibilidade
- Separação de responsabilidades





# Imports



# Imports

Java é uma linguagem que permite organização modular de classes e interfaces através do conceito de **pacotes**, e os imports servem justamente para utilizar-se recursos (classes e interfaces) que estão localizadas em outros pacotes. A palavra **import** é uma das muitas **palavras reservadas** da linguagem e não deve ser usada para nomear variáveis.

Exemplo.:

```
import java.util.List; // importação explícita
```

```
import java.util.Date; // importação explícita
```

```
import java.util.*; // importação implícita
```

Dica: O wildcard "\*" serve para indicar que se está importando todos os recursos do pacote em questão. Porém, é uma boa prática se fazer importações de forma explícita ao invés do usá-lo.

# Imports

```
// Importando o módulo utilitário do Java para Arrays
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Double> list = new ArrayList<Double>();
        list.add(2.0);
    }
}
```

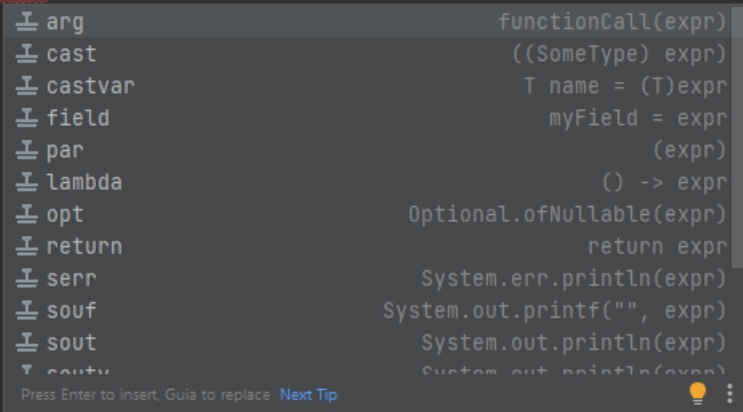


# Primitivo vs Referência

# Primitivo vs Referência

Os tipos de dados primitivos não tem características (atributos) nem comportamentos (métodos), logo, estes encontram-se localizados nos wrappers correspondentes.

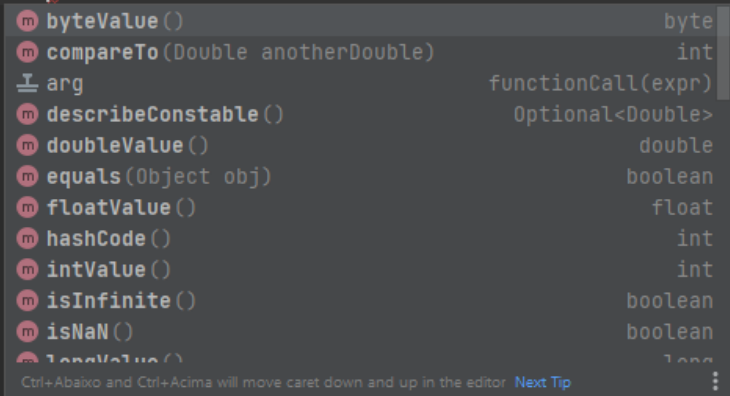
```
public class Tipos {  
    public static void main(String[] args) {  
        double primitivo = 2;  
        Double referencia = 2.0;  
  
        primitivo.  
    }  
}
```



Tipo **primitivo** *double* - data type

Armazenado na stack

```
public class Tipos {  
    public static void main(String[] args) {  
        double primitivo = 2;  
        Double referencia = 2.0;  
  
        referencia.  
    }  
}
```



Tipo **de referência** (wrapper) *Double* - object type

Armazenado em stack + heap

# Primitivo vs Referência

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean



# Anotações e Enums

# Anotações

As anotações provém uma forma de se adicionar **metadados** nas classes, atributos e métodos. Elas não só são informativas como também podem alterar o comportamento dos elementos em si.

Exemplos.:

@Override @Deprecated @SuppressWarnings

```
@Override
public String toString() {
    return "Carro {" +
        "marchas=" + marchas +
        "}";
}
```

```
@Tag("Regressao")
@Test
void junitTesteMetodo() {
    assertEquals(2, 1 + 1);
}
```

# Enums

Os Enumeradores (ou enums) são um **conjunto de dados imutáveis** e pré-definidos, ou seja, um **conjunto de constantes**. As constantes de um enum são, implicitamente, *static final*.

- As instâncias dos tipos enum são criadas e nomeadas como se fosse uma declaração de classe, sendo **fixas e imutáveis**;
- Podem incluir atributos de instância, construtor, métodos de instância e de classe;
- Não é possível **criar-se instâncias** de enums (usando **new**);
- O **construtor** é sempre *private*, embora não precise de um modificador private explícito;
- Como são considerados objetos constantes e imutáveis (*static final*), logo os nomes declarados são **SNAKE\_CASE**;
- Podemos empregar as constantes enum em qualquer local onde usaríamos constantes comuns;





# Casting vs Promoção

# Casting vs Promoção

O **casting** refere-se à **conversão** de uma variável de **um tipo** de dados em **outro tipo** de dados. Existem dois tipos de conversão:

**Implícita (widening)**

```
int in = 5;  
long lon = in;
```

**Explícita (narrowing)**

```
long lon = 10000;  
int in = (long)lon;
```

A **promoção** se dá na conversão **automática** de tipos de dados **menores** em tipos de dados **maiores** (se necessário) durante a **avaliação da expressão ou chamada de método**:

**Avaliação de expressão**

```
int menor = 150;  
double maior = 150.5;  
double res = menor + maior;
```

**Chamada de método**

```
void metodo(double maior){ }  
int menor = 150;  
metodo(menor);
```



# Estruturas de Condição e Repetição

# Estruturas - Condição

<b>Se (If)</b>	Comando lógico: "se essa expressão for verdadeira, faça isso".
<b>Senão (else)</b>	Comando lógico <u>complementar ao If</u> : "caso a expressão acima NÃO for verdadeira, faça isso".
<b>Senão-Se (else If)</b>	Comando lógico <u>complementar ao If</u> : "caso a expressão acima NÃO for verdadeira e essa nova expressão for verdadeira, faça isso".
<b>Escolha-Caso (switch case)</b>	Comando lógico: "dada essa entrada, caso ela seja IGUAL a X, faça isso, caso seja IGUAL a Y, faça aquilo, caso seja IGUAL a Z, faça aquele outro"
<b>Se/Senão ternário</b>	Comando lógico if/else <b>simplificado</b> de uma linha para conveniência.

# Estruturas - Repetição

<b>Enquanto</b> (while)	Esta instrução é usada quando não sabemos quantas vezes um determinado bloco de instruções precisa ser repetido.
<b>Repita-Até</b> (do-while)	O <i>do/while</i> tem quase o mesmo funcionamento que o <i>while</i> , a diferença é que com o uso dele teremos os comandos executados <b>ao menos uma vez</b> .
<b>Para</b> (For)	Estrutura que repete um bloco um determinado número de vezes. Recebe um <i>contador</i> , <i>uma condição de repetição</i> e <i>um incremento/decremento</i> .
<b>Para-Cada</b> (For-each)	For extra com capacidade de iterar sobre arrays e listas de forma <b>simplificada</b> .



# Interfaces

# Interfaces

Uma interface é utilizada para determinar que um **grupo de classes** tenham métodos e/ou propriedades em comum (que todas essas classes obedecem um **contrato**)

Os **métodos** nas classes que implementam a interface continuam **independentes**

Uma classe só tem uma implementação válida de uma interface quando implementa **todos os seus métodos**

Interfaces **não contém implementações** de métodos

Diferente de heranças, uma classe **pode** implementar **mais de uma** interface diretamente (múltipla)

```
public interface IFormaGeometrica {  
    String getNomeFigura();  
    int getArea();  
    int getPerimetro();  
}
```



# Collections API



# Collections API

**Objetos** que permitem armazenar variadas **estruturas de dados** (Listas, Filas, Pilhas, Árvores binárias, entre outras). Podemos entender como **abstrações de arrays** já providas pela própria linguagem, com uma série de recursos adicionais a depender do objeto utilizado.

Vantagens:

- Redução de esforço de programação e consequente aumento de desempenho
- Aumento de qualidade do código e possível aumento de performance

Armazenar

Recuperar

Manipular

Pesquisar



Objetos

# Collections API – Documentação

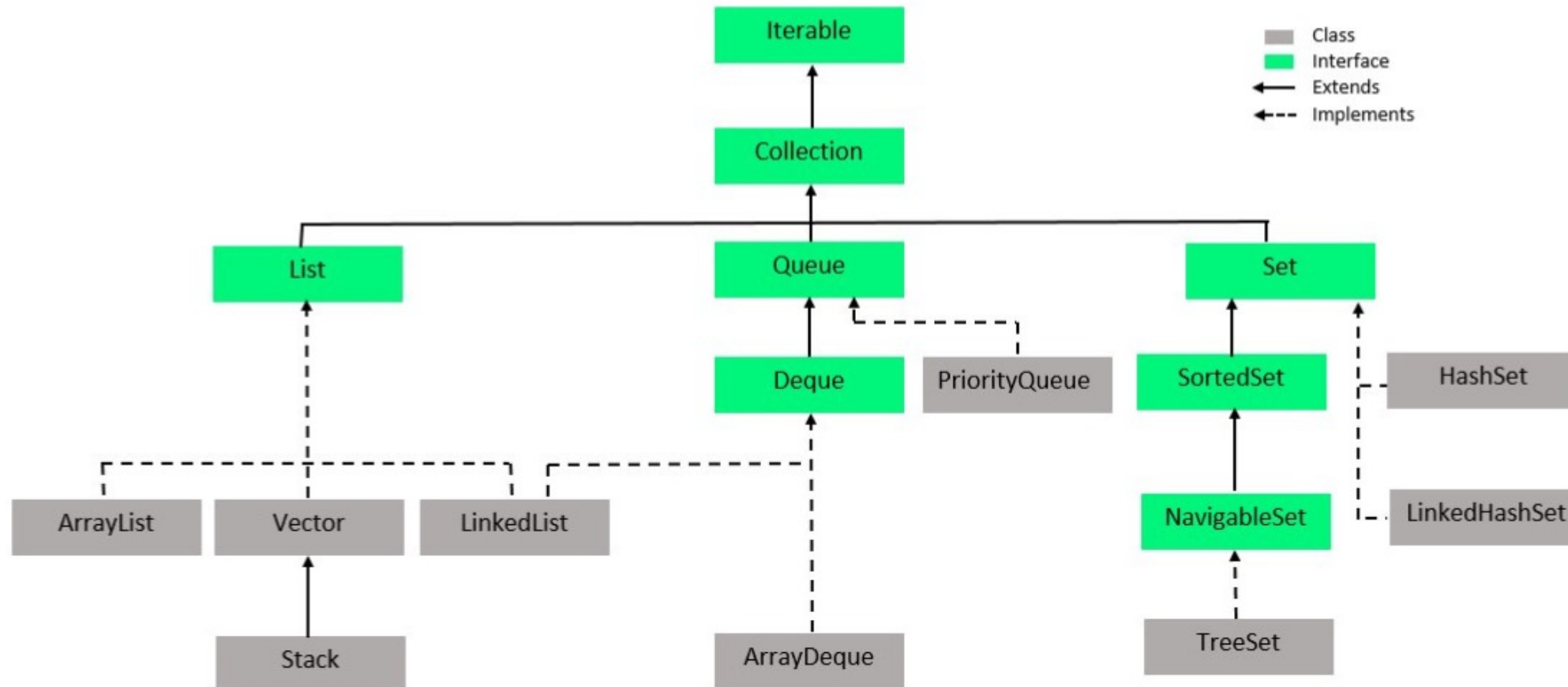
A despeito de qualquer compilação de informações aqui registrada, entenda que a API de Collections **é extensa**, e não há uma fonte de informações melhor do que a **própria documentação da linguagem** para cada uma das hierarquias.

***Collections API - Iterable***

***Collections API - Map***

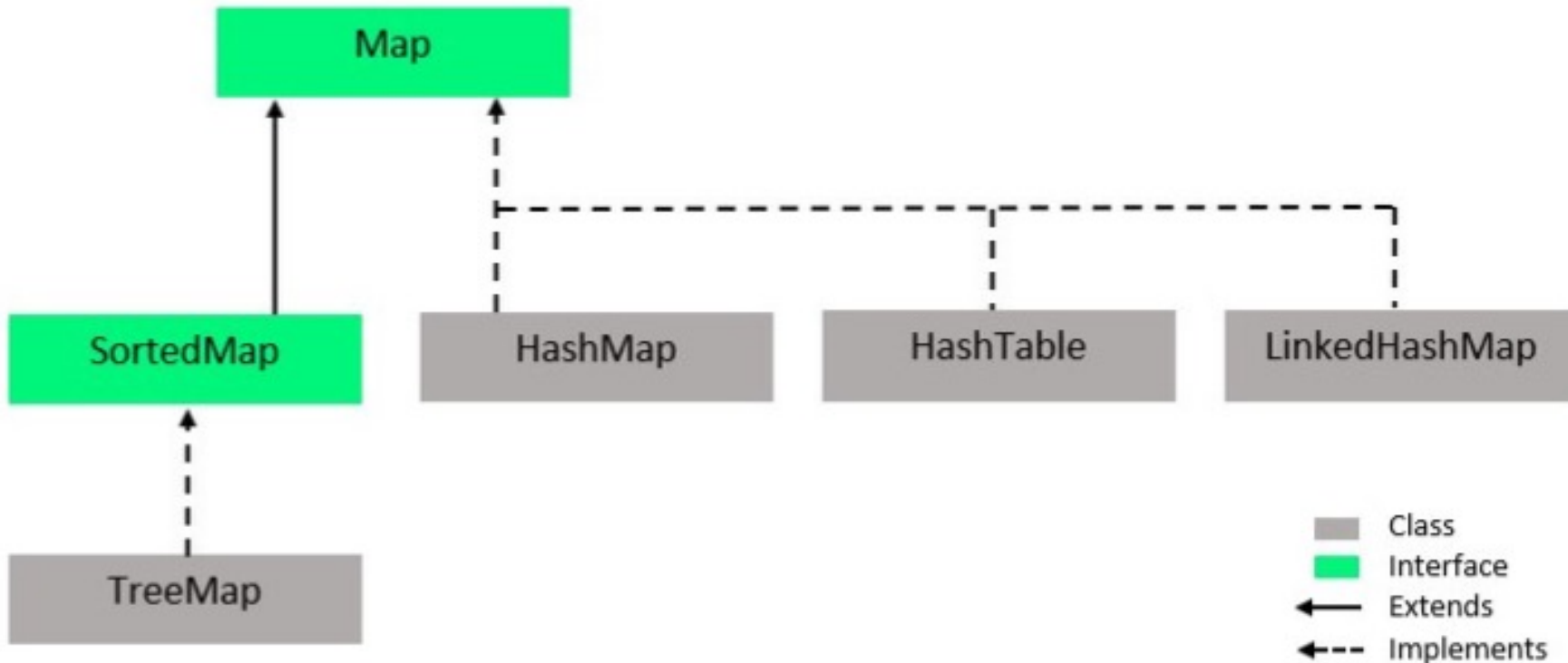
# Collections API – Hierarquia Iterable

Tem como principal objetivo fornecer um meio comum de **iterar** sobre elementos de coleções (listas, sets, pilas, filas, entre outros).



# Collections API – Hierarquia Map

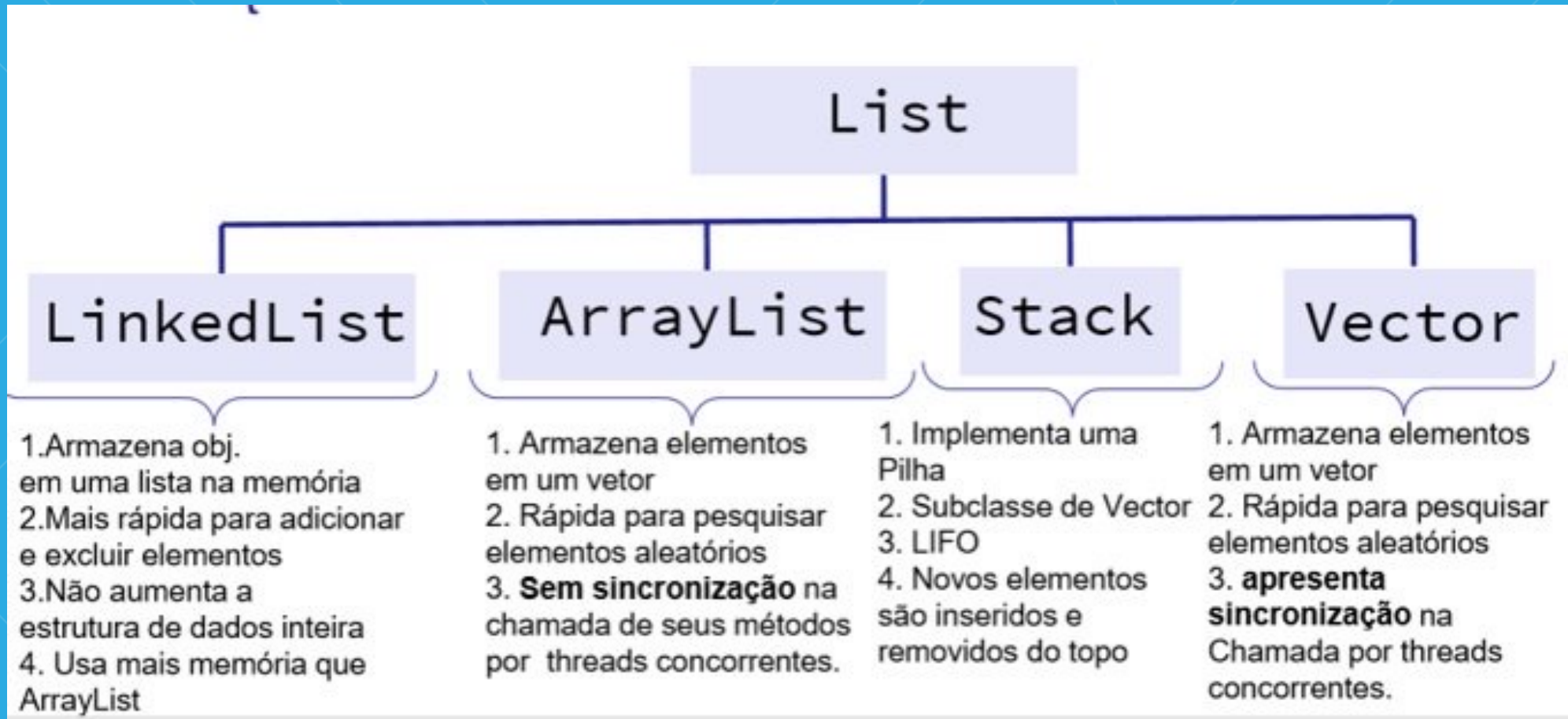
Tem como principal objetivo fornecer um meio comum trabalhar-se com **pares de chave-valor** (key-value pair) únicos, possibilitando também iterações. Também são conhecidos como **dicionários**.



# Interface List

- A API de Collections traz a **interface** `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista.
- Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.
- Define coleções que se organizam como arrays de **tamanho dinâmico**, de forma que cada elemento seja acessível por um índice.
- Novos elementos podem ser criados ou removidos em qualquer posição e pode haver elementos duplicados.
- Nem todas as listas garantem acesso indexado com tempo constante.
- **ArrayList não é uma Array!**

# Hierarquia das classes de lista



# Interface Map (dicionário)

- A interface Map **não** descende de **Collection**, porém faz parte do framework de coleções da linguagem Java.
- Não contém chaves duplicadas e mapeia **chaves K para valores V**.
- Uma chave (única e exclusiva) é mapeada para um valor específico – tanto a chave quanto o objeto são valores.
- A chave é usada para achar um elemento rapidamente. Permite procurar um valor com base na chave, solicitar um conjunto apenas com os valores ou somente com as chaves, caso tiver uma chave repetida é sobrescrito pela última chamada.

```
Map<K, V> mapa = new Type();
```

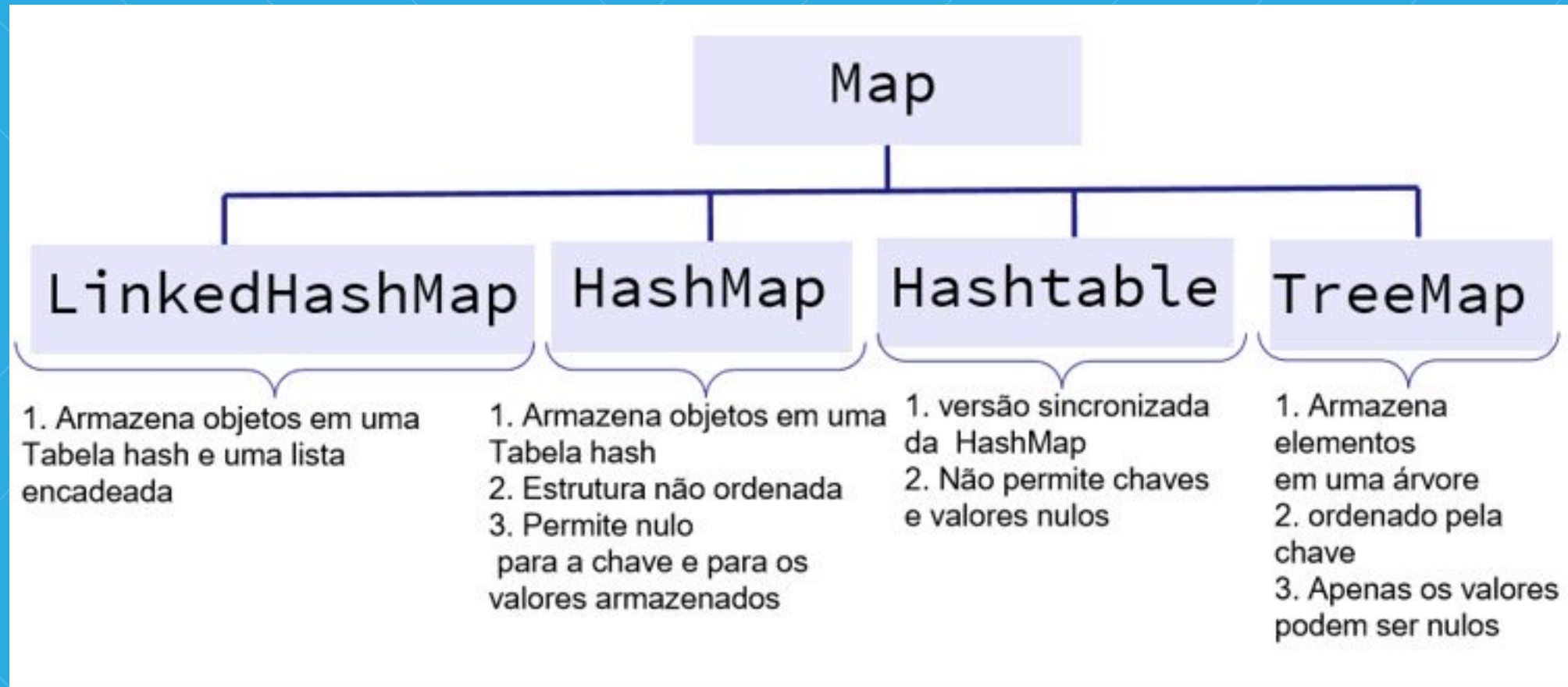
## Sintaxe

- E - é o objeto declarado, podendo ser classes **Wrappers** ou tipo de coleção.
- Type - é o tipo de objeto da coleção a ser usado.

```
Map<E> mapa = new Type();
```



# Hierarquia das classes de Map





# Como iterar qualquer mapa em Java

- Não podemos iterar um mapa diretamente usando iteradores, porque os mapas não são coleções. Além disso, antes de prosseguir, você deve saber um pouco sobre a interface `Map.Entry <K, V>`;
- Iterando sobre **`Map.entrySet ()`** usando o loop For-Each;
- O método `Map.entrySet ()` retorna uma visão de coleção (`Set <Map.Entry <K, V >>`) dos mapeamentos contidos neste mapa. Portanto, podemos iterar sobre o par de valores-chave usando os métodos **`getKey()`** e **`getValue()`** de `Map.Entry <K, V>`. Este método é mais comum e deve ser usado se você precisar mapear chaves e valores no loop;

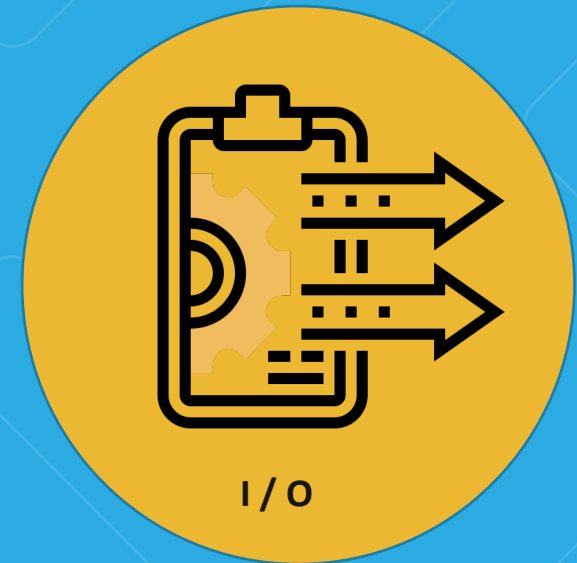


# I/O API

# I/O API

**I/O** (Input/Output) define-se na computação como uma forma de **comunicação** entre uma fonte de processamento de informação (um computador) **originária** e um ente externo **destinatário**, podendo este ser representado por um ser humano ou até mesmo por um outro computador através de uma rede.

O Java disponibiliza esses recursos de comunicação através da **I/O Api**, que contém uma série de **Objetos** para gerenciamento de arquivos e diretórios, além de execuções de leitura e escrita nesses mesmos arquivos, e também em fluxos de rede e memória.



# I/O API – Documentação

A despeito de qualquer compilação de informações aqui registrada, entenda que a API de Inputs e Outputs é **extensa**, e não há uma fonte de informações melhor do que a **própria documentação da linguagem** para cada uma das hierarquias.

***I/O API - File***

***I/O API - InputStream***

***I/O API - OutputStream***

***I/O API - Reader***

***I/O API - Writer***

# I/O API - File

Um arquivo (file) é uma **abstração** utilizada para **uniformizar a interação** entre a fonte de processamento e o ente externo **destinatário**.

A classe **File** provê esses recursos, permitindo trabalhar-se com criação, ajuste e exclusão de diretórios, listagem de conteúdos de diretórios ou até obtenção de uma série de atributos comuns de arquivos e diretórios.



# File separator

A classe `java.io.File` contém quatro variáveis separadoras estáticas:

***separator***: caractere separador de diretórios dependente da plataforma. Retorna `String`.

**Exemplo:** Windows -> `\`; Unix -> `/`.

**`separatorChar`**: O mesmo que `separator`, porém retorna `char`;

**`pathSeparator`**: variável dependente da plataforma para separador de caminho. Retorna `String`.

**Exemplo:** lista de variáveis de ambiente `PATH` ou `CLASSPATH` para separar os caminhos de arquivo.

Windows -> `';'`. Unix -> `':'`.

**`pathSeparatorChar`**: mesmo que `pathSeparator`, mas é `char`.

# System Properties

A classe **System** tem dois métodos usados para ler as propriedades do sistema: **getProperty** e **getProperties**. Ambos recuperam o valor da propriedade nomeada na lista de argumentos. O mais simples dos dois métodos *getProperty* usa um único argumento, uma chave de propriedade. Por exemplo, para obter o valor de `path.separator`, use a seguinte instrução:

```
System.getProperty("path.separator");
```

O método `getProperty` retorna uma string contendo o valor da propriedade. Se a propriedade não existir, esta versão de `getProperty` retornará nulo.

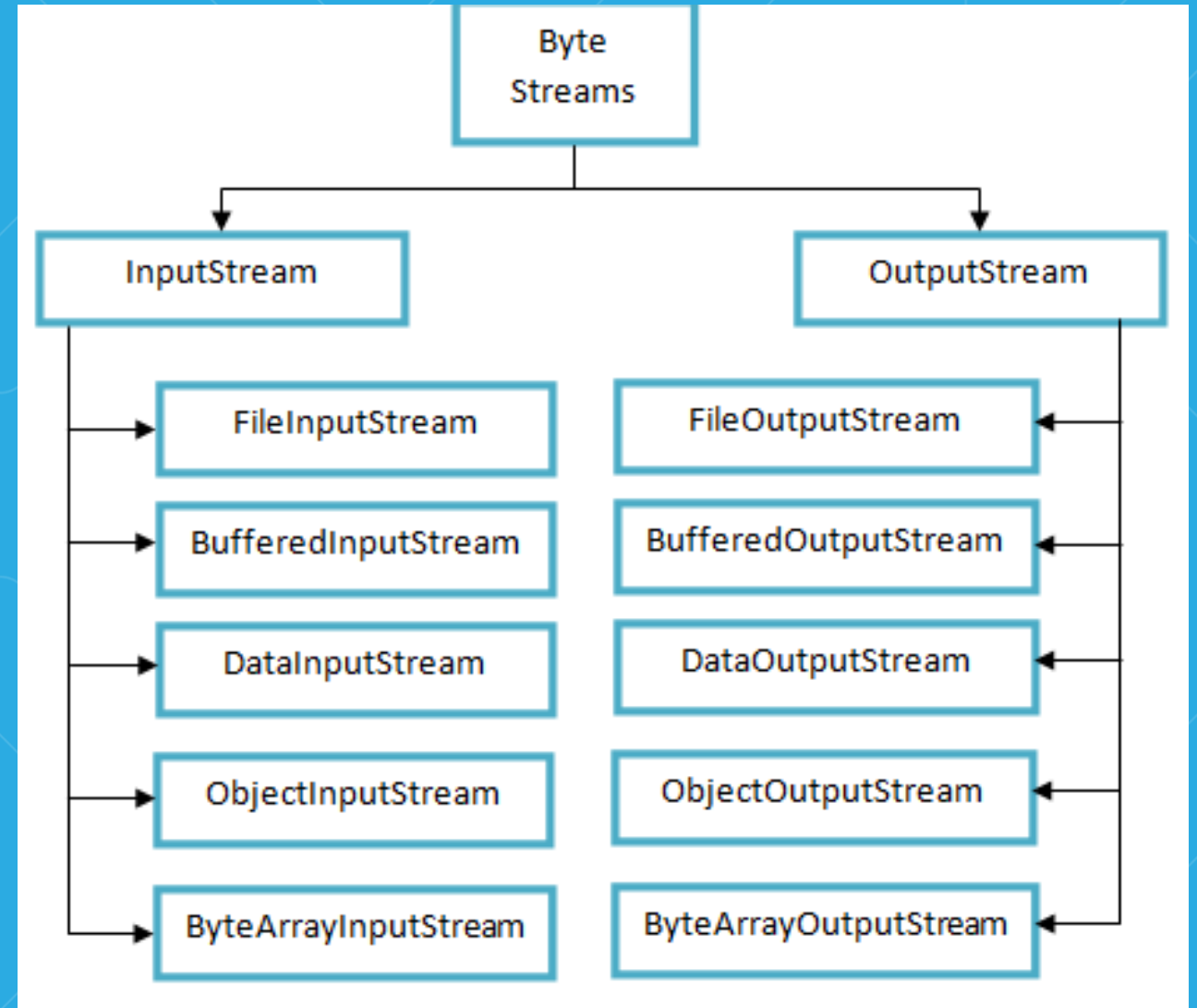
# I/O API – Hierarquia ByteStream

Tem como principal objetivo fornecer um meio de comunicação ágil e simplificado através do fluxo binário (0s e 1s) sem nenhum tipo de codificação ou decodificação. Devido a essa natureza é recomendada para uso com arquivos binários brutos como imagens, áudios, vídeos e executáveis em geral, localmente ou em rede. Exemplos:

- ByteArrayInputStream e ByteArrayOutputStream.
- **FileInputStream** e **FileOutputStream**
- DataInputStream e DataOutputStream

**I/O API - InputStream**

**I/O API - OutputStream**



Não existe um elemento "ByteStreams" em Java. As classes base de fato são **InputStream** e

**OutputStream**



# I/O API – Hierarquia CharacterStream

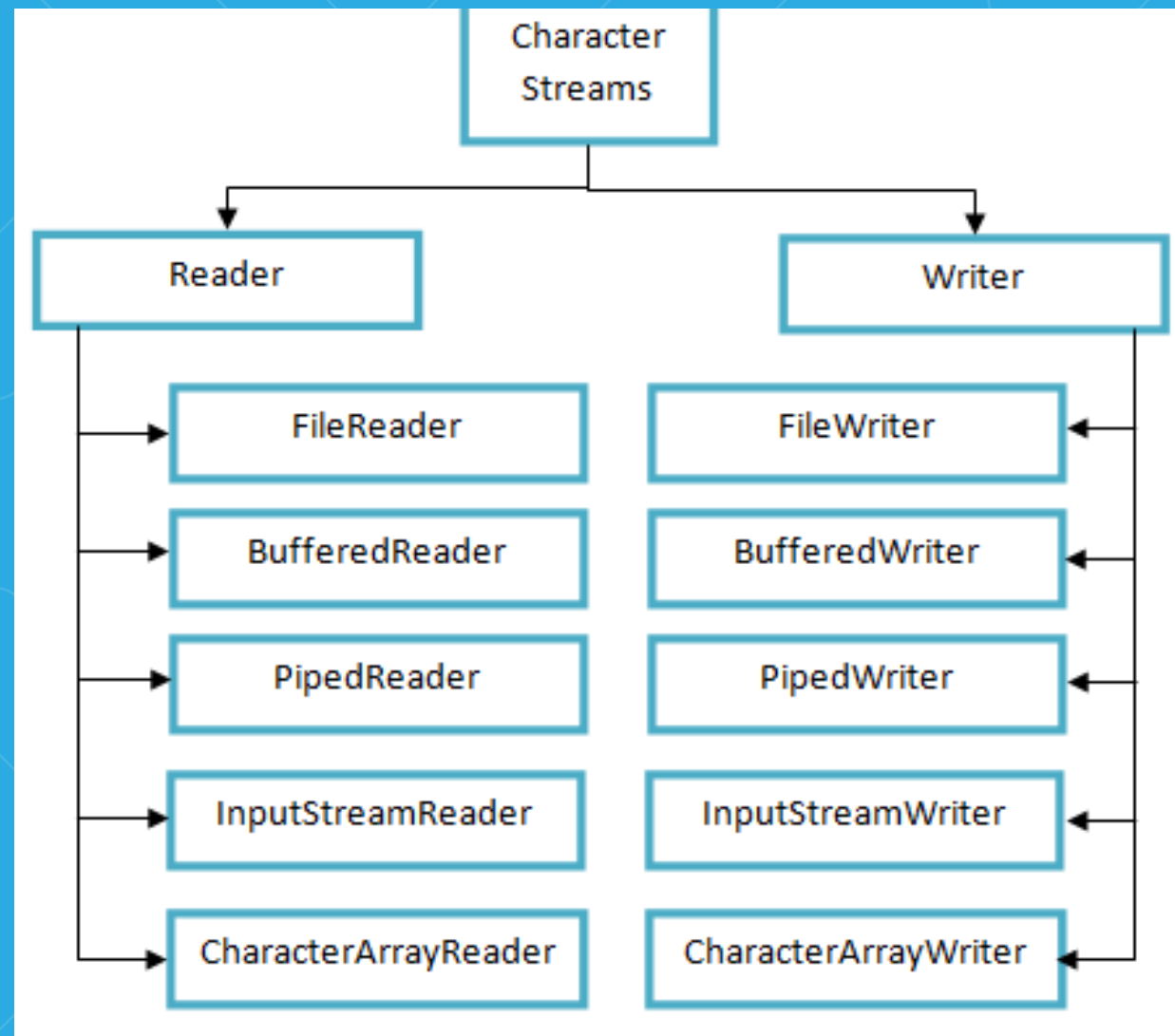
Tem como principal objetivo fornecer um meio de comunicação intuitivo através de fluxos de caracteres com codificações Unicode como UTF-8 ou UTF-16. Devido a essa natureza é recomendada para uso com arquivos texto em geral, localmente ou em rede.

Exemplos:

- `CharacterArrayReader` e `CharacterArrayWriter`
- `BufferedReader` e `BufferedWriter`
- **`FileReader` e `FileWriter`**

**I/O API - Reader**

**I/O API - Writer**



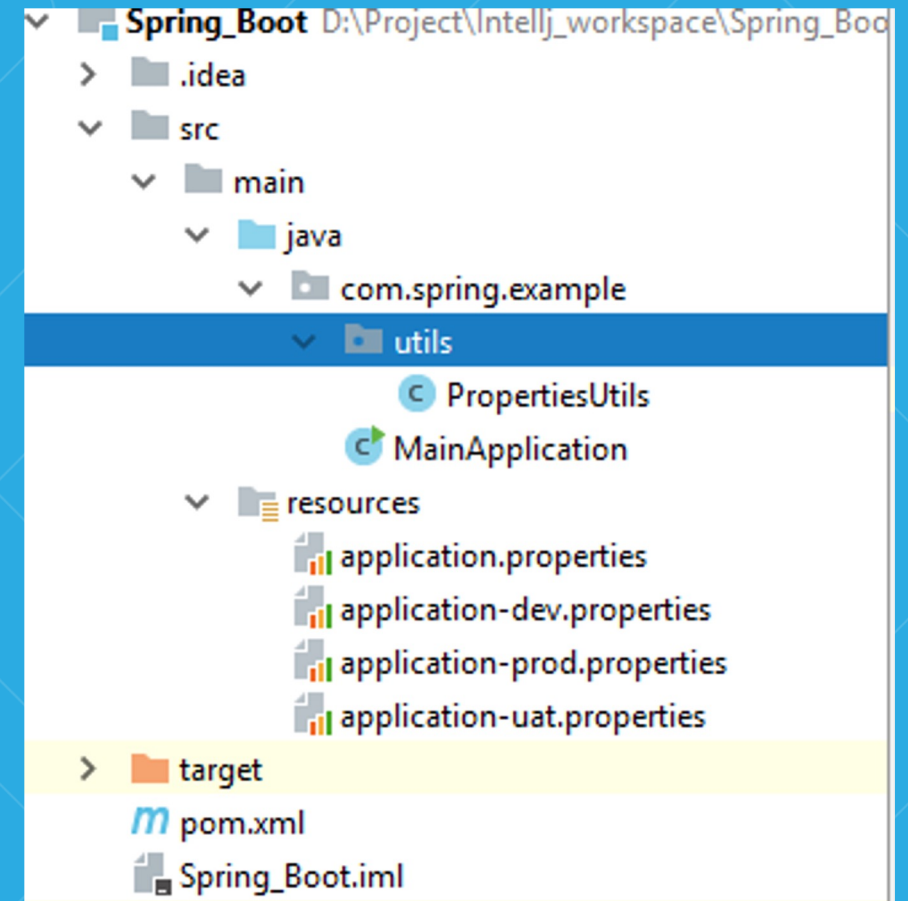
Não existe um elemento "CharacterStreams" em Java. As classes base de fato são **Reader** e

**Writer**

# Properties

O arquivo de propriedade é uma ótima opção para passar configurações para uma determinada aplicação que necessita de configurações externas e a mesma em si não pode ser alterada. As propriedades podem ser salvas em um Stream ou carregadas de um Stream. Um exemplo seria um programa que conecta a um banco de dados e precisa de dados para realizar a conexão, sem que o código fonte do mesmo seja alterado.

Esses arquivos **.properties** geralmente são criados dentro de uma pasta a parte. Muitas vezes dentro da pasta com o nome **resources**.



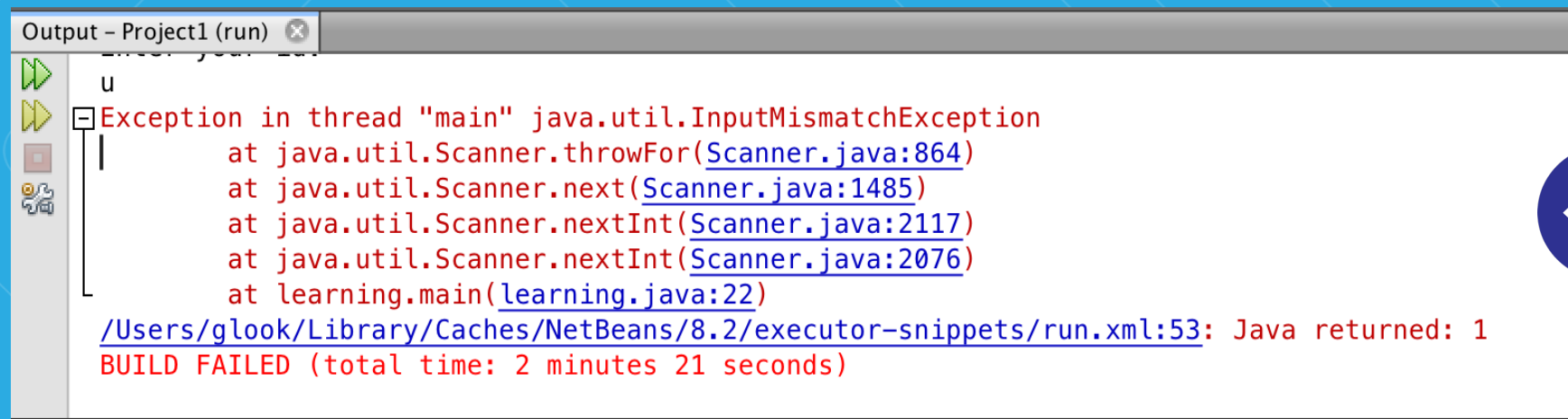


# Tratamento de Exceções

# Tratamento de exceções

Uma exceção é uma “**condição excepcional no fluxo**”, ou seja, é uma **ocorrência que altera o fluxo normal** de execução de um programa. Exceções são extremamente comuns em um sistema computacional e podem ser originadas por uma série de motivos, entre eles: erros de sistema, informações não correspondentes às esperadas em determinado trecho, ou mesmo uma geração proposital de uma exceção para um controle do fluxo. Exceções precisam ser **tratadas** a fim de não causarem comportamentos inesperados e desagradáveis.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at primeiro_programa.ProgramaPrincipal.main(ProgramaPrincipal.java:10)
```



The screenshot shows a window titled "Output - Project1 (run)". It displays a stack trace for an exception. The exception is "Exception in thread 'main' java.util.InputMismatchException". The stack trace lists the following frames from bottom to top: "at learning.main(learning.java:22)", "at java.util.Scanner.nextInt(Scanner.java:2076)", "at java.util.Scanner.nextInt(Scanner.java:2117)", "at java.util.Scanner.next(Scanner.java:1485)", and "at java.util.Scanner.throwFor(Scanner.java:864)". Below the stack trace, it says "/Users/glook/Library/Caches/NetBeans/8.2/executor-snippets/run.xml:53: Java returned: 1" and "BUILD FAILED (total time: 2 minutes 21 seconds)".

```
Output - Project1 (run)
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at learning.main(learning.java:22)
/Users/glook/Library/Caches/NetBeans/8.2/executor-snippets/run.xml:53: Java returned: 1
BUILD FAILED (total time: 2 minutes 21 seconds)
```

O rastro da pilha (stacktrace) é bastante útil para a resolução do problema.

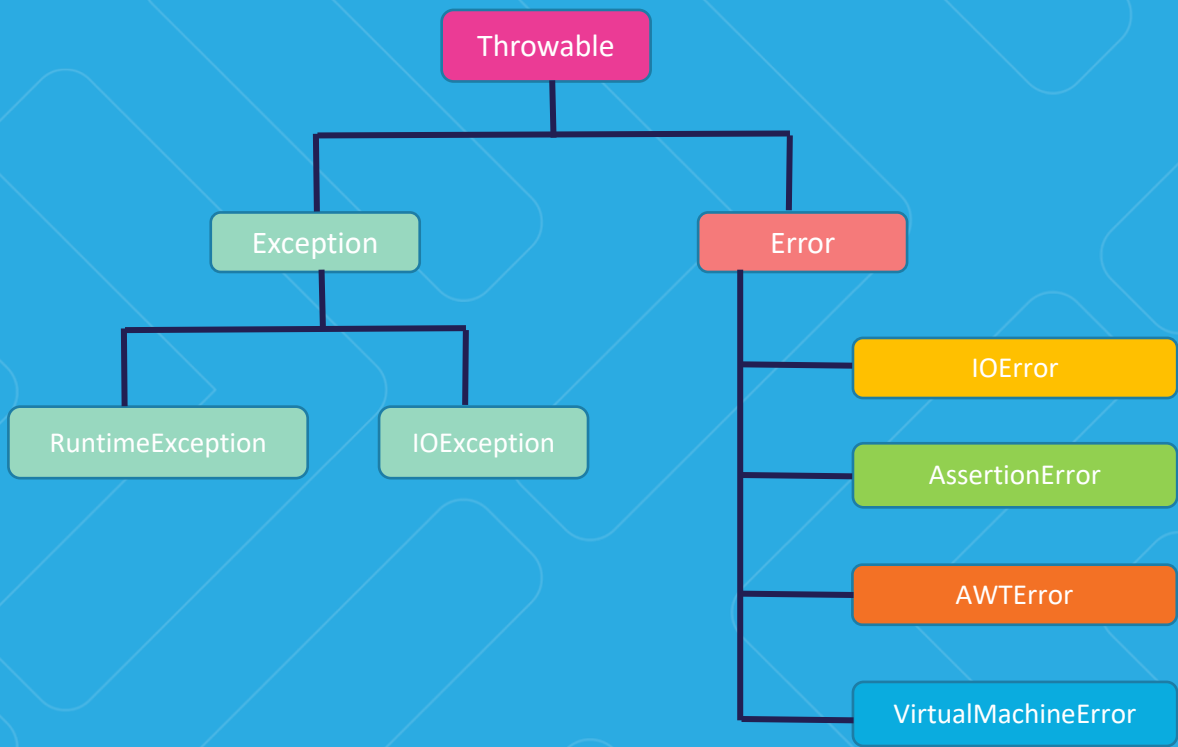
# Tratamento de exceções – Documentação

A despeito de qualquer compilação de informações aqui registrada, entenda que o controle de exceções envolve **uma série de classes** que representam estas, logo, não há uma fonte de informações melhor do que a **própria documentação da linguagem** para cada uma das hierarquias.

***Exceções - Error***

***Exceções - Exception***

# Exceções



## Diferença entre error e exception

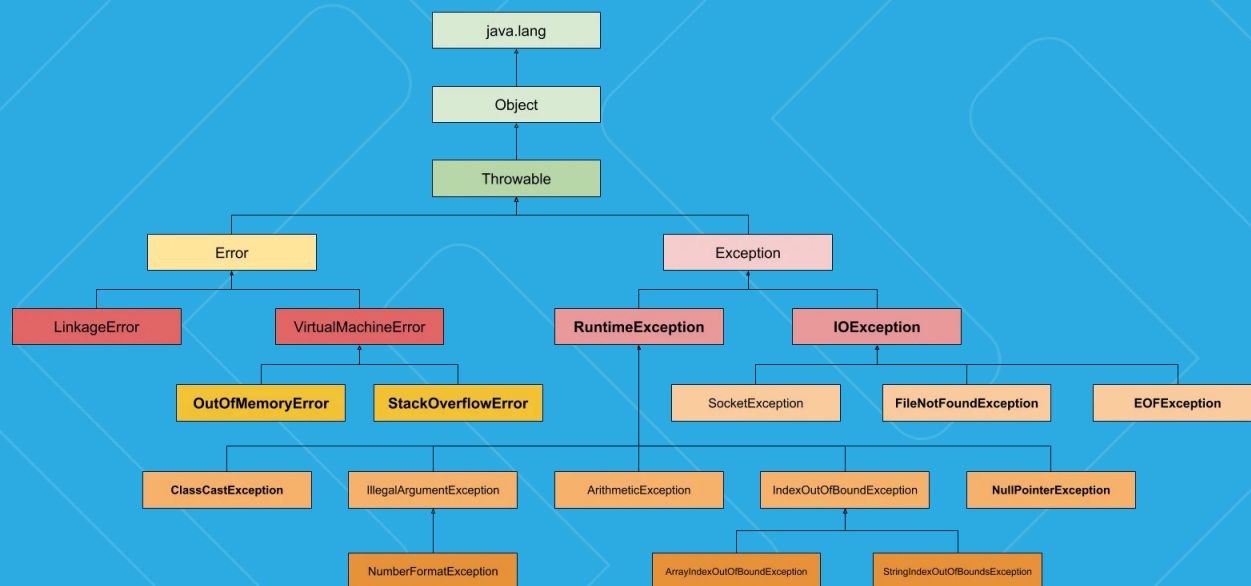
Base para Comparação	Erro	Exceção
Basic	Um erro é causado devido à falta de recursos do sistema.	Uma exceção é causada por causa do código.
Recuperação	Um erro é irrecuperável.	Uma exceção é recuperável.
Palavras-chave	Não há meios de manipular um erro pelo código do programa.	Exceções são tratadas usando três palavras-chave "try", "catch" e "throw".
Consequêncis	Quando o erro é detectado, o programa terminará de forma anormal.	Como uma exceção é detectada, ela é lanç ada e capturada pelas palavras-chave "throw" e "catch" correspondentemente.
Tipos	Erros são classificados como tipo desmarcado.	As exceções são classificadas como tipo marcado ou desmarcado.
Pacote	Em Java, os erros são definidos no pacote "java.lang.Error".	Em Java, as exceções são definidas em "java.lang.Exception".
Exemplo	OutOfMemory, StackOverFlow.	Exceções verificadas: NoSuchMethod, ClassNotFound. Exceções não verificadas: NullPointerException, IndexOutOfBounds.

# Exceções

O Exceção significa “**condição excepcional**” e é uma ocorrência que altera o fluxo normal do programa. Em qualquer linguagem de programação essas exceções existem, então, foram criadas formas de “driblar” essas exceções onde, caso tenha ocorrido uma exceção, apareça uma mensagem amigável para o usuário e não um monte de linhas com informações desnecessárias ao usuário.

Uma exceção representa uma situação que normalmente não ocorre e é algo de estranho ou inesperado no sistema.

## Árvore de exceções Java



# Checked exception

Esses tipos de exceções são aquelas que já são cheçadas pelo próprio compilador do java, não deixando executar o sistema se essa exceção não for tratada ou propagada novamente (throws).

Exemplos: **IOException**, **FileNotFoundException**

```
class Teste {  
    public static void metodo() {  
  
        new java.io.FileInputStream("arquivo.txt");  
  
    }  
}
```

```
Teste.java:3: unreported exception java.io.FileNotFoundException; must be caught  
or declared to be thrown  
    new java.io.FileReader("arquivo.txt");  
    ^  
1 error
```



# Unchecked exception

Esse tipo de exceções são aquelas que acontecem em tempo de execução, ou seja, aquelas que o compilador não consegue prever se pode acontecer.

Exemplo: todas as RuntimeException.

```
public void saca(double valor) {  
    if (this.saldo < valor) {  
        throw new RuntimeException();  
    } else {  
        This.saldo-=valor;  
    }  
}
```

RuntimeException é a exception mãe de todas as exceptions unchecked. A desvantagem aqui é que ela é muito genérica; quem receber esse erro não saberá dizer exatamente qual foi o problema.

# Error Vs Exception

Existem dois tipos de exceções no Java, as Error's e as Exceptions. Error é aquele erro que **não deve ser tratado** ou não é responsabilidade do programador tratar, como por exemplo, estouro de memória. Já as Exceptions são **condições excepcionais** que alteram o fluxo do programa, existindo técnicas de “driblar” essas condições, como por exemplo, divisão por zero.

## Como tratar uma exceção?

- **try** - identifica um bloco de comandos que podem disparar uma exceção;
- **catch** - captura as exceções e implementa seus tratadores;
- **finally** - usado para códigos de liberação de recursos;
- **throw** - usado para causar uma exceção;
- **throws** - usado para propagar uma exceção causada por um método.

# Try / catch

O bloco try tenta processar o código que está dentro, sendo que se ocorrer uma exceção, a execução do código pula para a primeira captura do erro no bloco catch. O uso do try serve para indicar que o código está tentando realizar algo arriscado no sistema.

O bloco catch trata a exceção lançada. Caso a exceção não seja esperada, a execução do código pula para o próximo catch, se existir. Portanto, se nenhum do bloco catch conseguir capturar a exceção, dependendo o tipo que for, é causada a interrupção ao sistema, lançando a exceção do erro. Um exemplo do uso desse bloco é visto em transações de Rollback, onde são utilizados para que a informação não persista no banco se for capturada uma exceção nesse bloco catch.

```
public static void main(String[] args) {  
  
    try{  
        System.out.println(5/0);  
    }catch (ArithmeticException e){  
        System.out.println("Ocorreu uma exceção");  
    }  
  
}
```

# Finally

## Sim, ele sempre será executado!

O bloco finally é utilizado para garantir que um código seja executado após um try, mesmo que uma exceção tenha sido gerada. Mesmo que tenha um return no try ou no catch, o bloco finally é sempre executado.

## Por qual motivo isso acontece?

O bloco finally, geralmente, é utilizado para fechar conexões, arquivos e liberar recursos utilizados dentro do bloco try/catch. Como ele é sempre executado, nós conseguimos garantir que sempre após um recurso ser utilizado dentro do try/catch ele poderá ser fechado/liberado no finally.

```
public static void main(String[] args) {  
  
    try{  
        System.out.println(5/3);  
    }catch (ArithmeticException e){  
        System.out.println("Ocorreu uma exceção");  
    }finally {  
        System.out.println("Sempre cai no finally");  
    }  
}
```

# Throws

A palavra-chave `throw`, que está no imperativo, lança uma `Exception`. Isso é bem diferente de `throws`, que está no presente do indicativo e só avisa da possibilidade daquele método lançá-la, obrigando o outro método que vá utilizar-se daquele a se preocupar com essa exceção em questão.

A cláusula `throws` declara as exceções que podem ser lançadas em determinado método, sendo uma vantagem muitas vezes para outros desenvolvedores que mexem no código, pois serve para deixar de modo explícito o erro que pode acontecer no método, para o caso de não haver tratamento no código de maneira correta.

```
public static void main(String[] args) {  
    try{  
        leInteiro();  
    }catch (Exception e){  
        System.out.println("Ocorreu um erro");  
    }  
}  
  
public static int leInteiro() throws Exception{  
    return new Scanner(System.in).nextInt();  
}
```

# Throw

As cláusulas `throw` e `throws` podem ser entendidas como ações que propagam exceções, ou seja, em alguns momentos existem exceções que não podem ser tratadas no mesmo método que gerou a exceção. Nesses casos, é necessário propagar a exceção para um nível acima na pilha.

A cláusula **`throw`** cria um novo objeto de exceção que é lançada.

```
public static void main(String[] args) {
    int[] vetor1 = {5, 4, 0, 65, 66, 7, 12};
    int[] vetor2 = {4, 0, 6, 8, 2, 4, 2, 4, 10};

    for(int i = 0; i < vetor2.length; i++){
        try{
            if(vetor1[i] % 2 != 0){
                throw new Exception("Divisão não exata");
            }
            System.out.println(vetor1[i] / vetor2[i]);
        }catch (ArithmeticException e){
            System.out.println("Indivisível por Zero");
        }catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Índice Inválido");
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

# Agradecemos Sua Participação!

[educacao@db.tec.br](mailto:educacao@db.tec.br)

[gestao\\_terra@dbserver.com.br](mailto:gestao_terra@dbserver.com.br)

