

# Paradigmas de Programação

Como se classificam?

# Paradigmas de programação

```
graph TD; A[Paradigmas de programação] --> B[imperativo]; A --> C[declarativo]; B --> D[programação orientada a objetos]; B --> E[programação procedural]; B --> F[...]; C --> G[programação funcional]; C --> H[programação lógica]; C --> I[...];
```

imperativo

→ programação  
orientada a objetos

→ programação  
procedural


→ ...

declarativo

→ programação  
funcional

→ programação  
lógica

→ ...

Aspecto	Paradigma Imperativo	Paradigma Declarativo
Definição	Foca em <b><u>como fazer algo</u></b> , detalhando o passo a passo.	Foca em <b><u>o que deve ser feito</u></b> , descrevendo o resultado desejado.
Abordagem	Define a sequência de comandos e instruções que o computador deve executar.	Define o resultado desejado sem especificar o fluxo de controle.
Controle de Fluxo	Controle explícito do fluxo de execução (e.g., loops, condicionais).	Controle de fluxo é geralmente implícito, não explicitamente controlado.
Ex. Linguagens 	C, C++, C#, Java, Python, Fortran, Assembly.	SQL, HTML, CSS, Prolog, Haskell.
Estado Mutável	Geralmente usa variáveis e permite modificações no estado.	Evita ou minimiza a modificação de estado; usa variáveis imutáveis.
Popularidade	Mais conhecido	Menos conhecido
Legibilidade	Pode ser menos legível se o código não for bem estruturado; mais explícito.	Frequentemente mais legível e conciso; menos explícito.

# Multiparadigmas



- As linguagens de programação frequentemente têm um paradigma "principal" ou predominante, que orienta seu design e uso.
- Esse paradigma é geralmente o mais enfatizado e utilizado na maioria das situações para as quais a linguagem foi projetada.
- No entanto, muitas linguagens modernas são **multi-paradigma**, o que significa que suportam e incentivam o uso de múltiplos estilos de programação, permitindo uma maior flexibilidade e adaptabilidade na solução de problemas.



# Vamos pensar um pouco

- Tenho uma lista de números inteiros. Quero imprimir a lista com somente os números pares. Como faço ???

```
public class ValidaParComFor {  
    public static void main(String[] args) {  
        List<Integer> lista = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 10);  
    }  
}
```

# Vamos pensar um pouco

- Uma das maneiras mais fáceis seria com
  - O laço FOR (for-i ou for-each)



```
public class ValidaParComFor {  
    public static void main(String[] args) {  
        List<Integer> lista = Arrays.asList(0, 1, 2, 3,4,5, 6, 7, 8, 10);  
        //Crio uma nova lista que irá receber somente os valores PAR  
        // da lista original  
        List<Integer> listaPar = new ArrayList<>();  
        //Iterajo com a lista original através do For-each  
        // adicionando na nova lista somente os valores Pares  
        for (Integer numero : lista) {  
            if (numero % 2 == 0) listaPar.add(numero);  
        }  
        //Imprimo a nova lista  
        System.out.println(listaPar);  
    }  
}
```



# Vamos pensar um pouco

- Mas se você conhece um pouco do **Paradigma Funcional**, saberá que pode melhorar esse código com funções de Ordem Superior e expressões Lambdas.

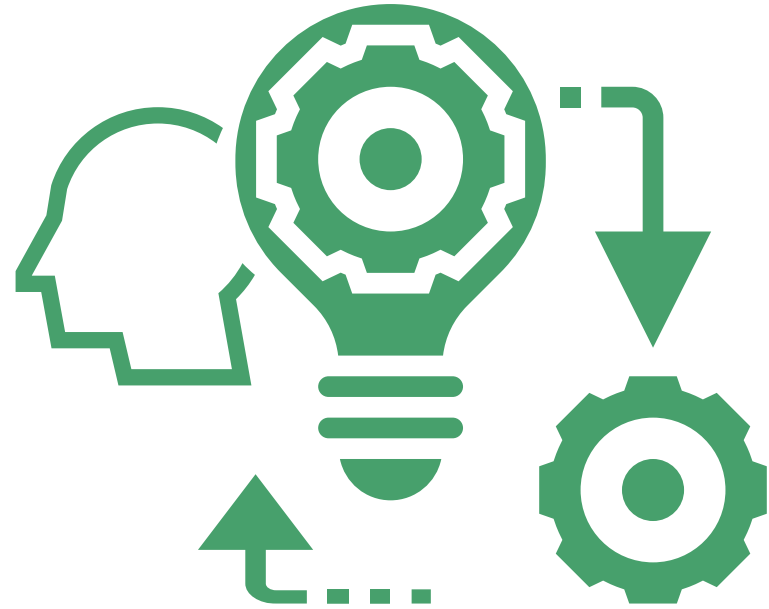
```
public class ValidaParComFilter {  
    public static void main(String[] args) {  
        List<Integer> lista = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 10);  
        //Aqui estou usando vários conceitos de programação Funcional  
        // Funções Lambda: numero -> numero % 2 == 0  
        // Funções de Ordem Superior: filter()  
        // Imutabilidade: A lista original não é alterada  
        // Programação Declarativa:  
        //     Descrevo O QUE desejo fazer com os dados (filtrar pares)  
        //     sem me preocupar com o COMO os dados são iterados ou armazenados.  
        List<Integer> listaPar = lista.stream().filter(numero -> numero % 2 == 0).toList();  
        System.out.println(listaPar);  
    }  
}
```

# Imperativo ou Declarativo ???

SQL

```
SELECT id, nome, cpf  
FROM usuarios  
WHERE nome = 'Zezinho';
```





# Paradigmas Imperativos

Quais os principais?

Quais as linguagens predominantemente imperativas?

# Paradigmas Imperativos

- Apesar das Linguagens de Programação de Alto Nível surgirem na década de 50, o termo “Paradigma de Programação” não era usado.

## Uso de goto e Spaghetti Code

- Nos primeiros dias da programação, o *goto* era amplamente utilizado para controlar o fluxo de execução. Sem regras rígidas para controle de fluxo, isso frequentemente resultava em código desordenado e difícil de manter.

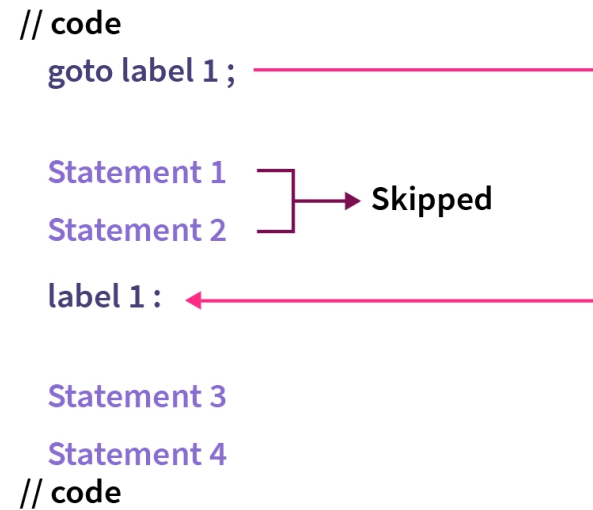
# goto - Exemplo

```
// code
goto label 1;

Statement 1
Statement 2 ] → Skipped

label 1:

Statement 3
Statement 4
// code
```



Jumped to where label 1 has been declared

# Spaghetti Code – Uso demasiado do *goto*

## Edgar Dijkstra: Go To Statement Considered Harmful

### Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like: while *B* repeat *A*

$un \rightarrow en$ ), the fact remains that the progress of the process remains characterized by a single textual index.

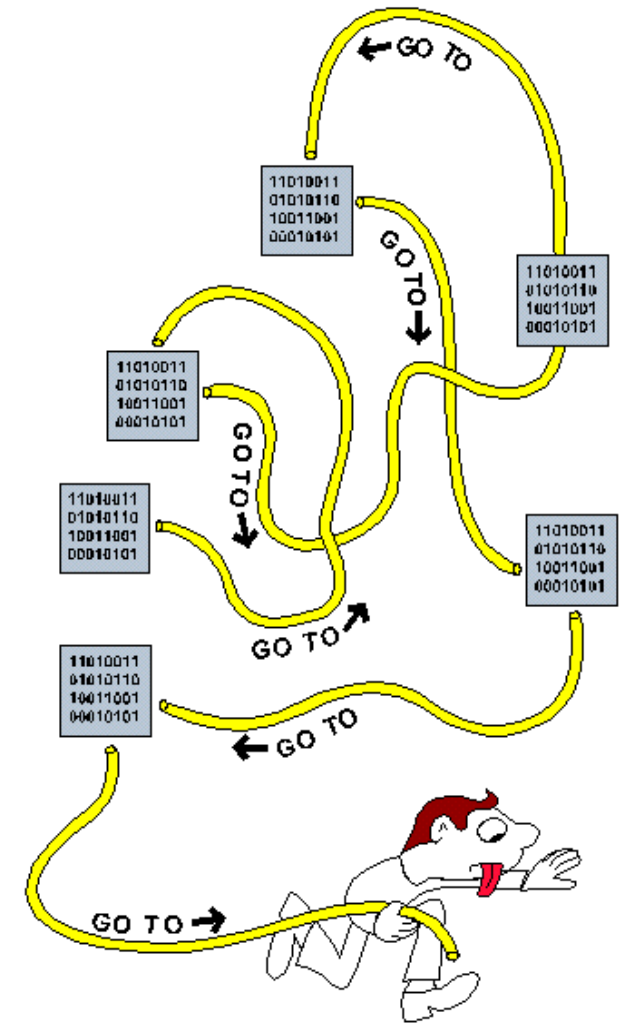
As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the

that they will satisfy all needs, but (e.g. abortion clauses) they should programmer independent coordina describe the process in a helpful a

It is hard to end this with a

Communic

Volume 11 / Number 3 / March, 1968



# Java Language Keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords cannot be used as identifiers in your programs.

abstract

assert<sup>\*\*\*</sup>

boolean

break

byte

case

catch

char

class

const<sup>\*</sup>

\* not used

\*\* added in 1.2

\*\*\* added in 1.4

\*\*\*\* added in 5.0

continue

default

do

double

else

enum<sup>\*\*\*\*</sup>

extends

final

finally

float

for

goto<sup>\*</sup>

if

implements

import

instanceof

int

interface

long

native

# Paradigma Procedural e Estruturado

## Década 1970

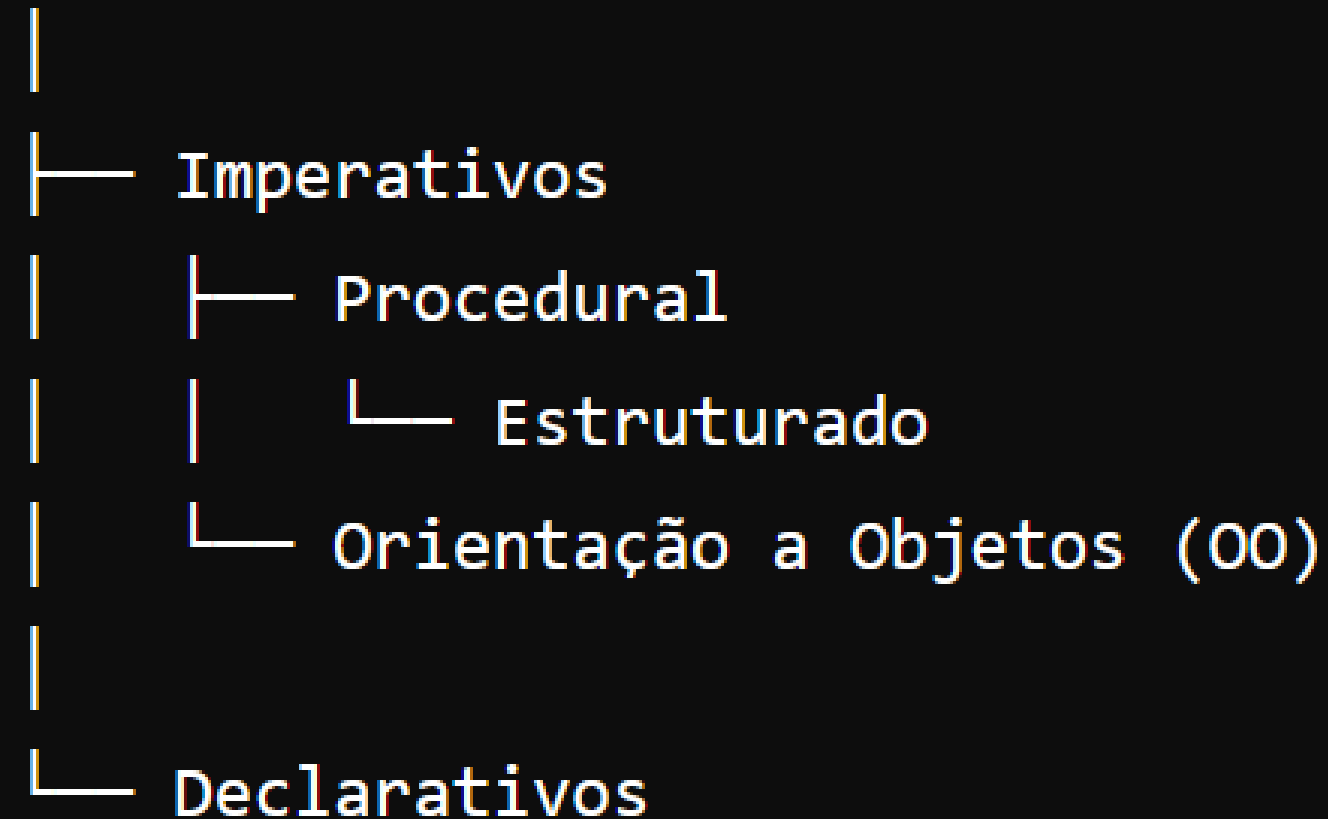
- **Programação Estruturada:** Estabeleceu práticas para evitar o uso excessivo de *goto*, usando estruturas de controle como **loops** (for, while) e **condicionais** (if, else) para criar um fluxo de controle mais claro e organizado.
- **Programação Procedural:** A programação procedural organiza o código em **procedimentos** ou **funções**, encapsulando a lógica em blocos modulares. Isso ajuda a criar um código mais legível e estruturado, minimizando a necessidade de saltos desordenados e promovendo uma abordagem modular e reutilizável.

# Paradigma Procedural, Estruturado e Imperativo

## Década 1970

- Alguns autores consideram o Paradigma Estruturado (Estrutural) como um subconjunto do Paradigma Procedural.
- Outros, já consideram que Paradigma Imperativo refere-se ao uso de estruturas de controle como loops e condicionais e a organização do código em procedimentos ou funções.
- Para Nossas aulas iremos utilizar o conceito de que o Paradigma Estruturado faz parte do Paradigma Procedural, e este por sua vez, faz parte do grupo de Paradigmas Imperativos.

# Paradigmas de Programação

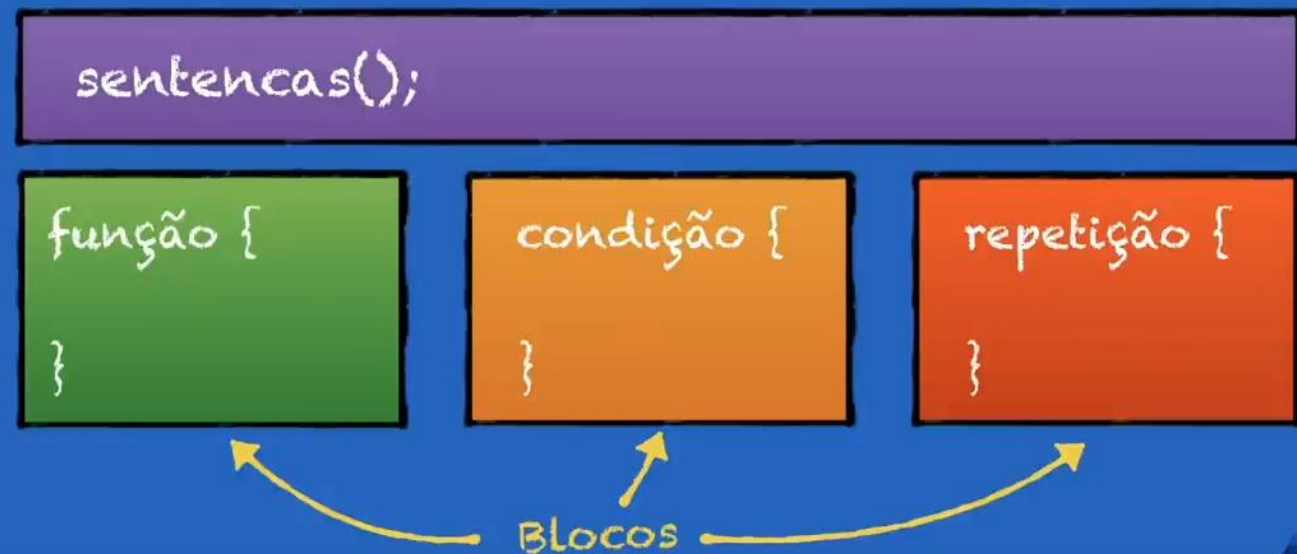




# Paradigma Procedural: Escopos



# Paradigma Procedural



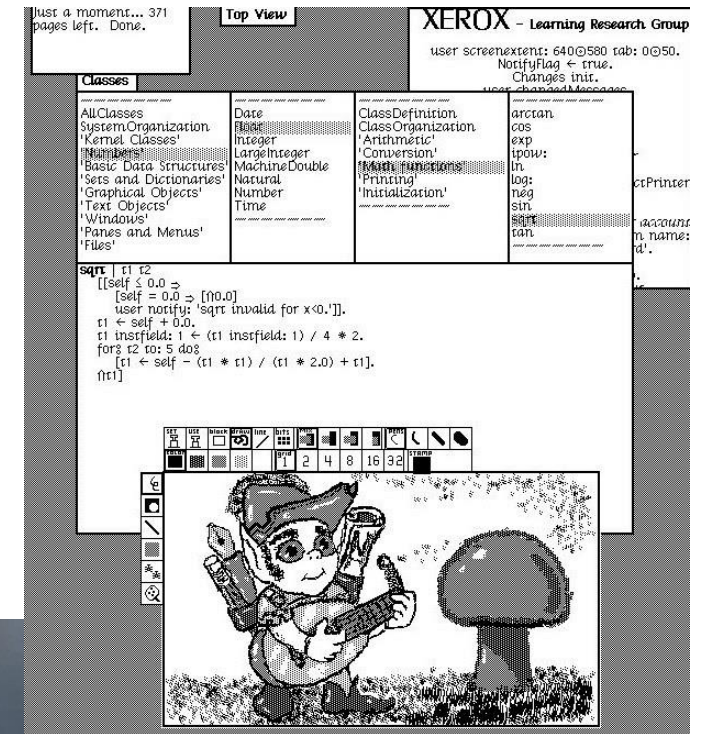
# Paradigma Procedural

## Linguagens de Programação

- Fortran – 1957
- COBOL – 1959
- ALGOL – 1960
- CPL (Combined Programming Language) – 1960-1963
- BCPL (Basic Combined Programming Language) – 1966
- B – 1969
- C – 1972
- Pascal – 1970

# Paradigma Orientado a Objetos

- Linguagem: Smalltalk-80
  - Introduziu o conceito de IDE
- Criador: Alan Kay
- Linguagem Simula implementou os primeiros conceitos de Classes



1950 – 1960 Era do Caos	1970 – 1980 Era da Estruturação	1990 até agora Era dos Objetos
Saltos, gotos, variáveis não estruturadas, variáveis espalhadas ao longo do programa	If-then-else Blocos Registros Laços-While	Objetos Mensagens Métodos Herança



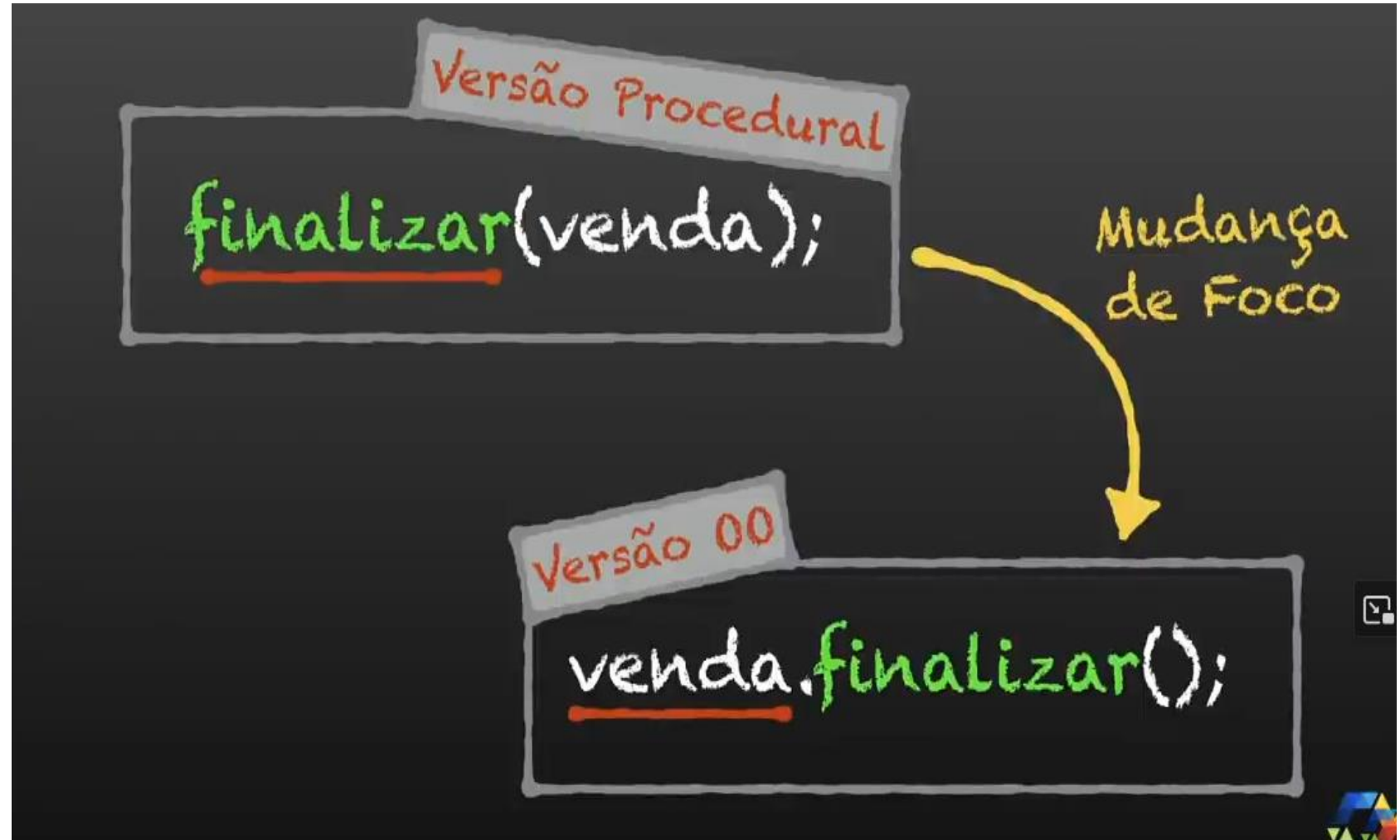
Ciência da  
**Computação**

Procedural

→ Foco na Ação

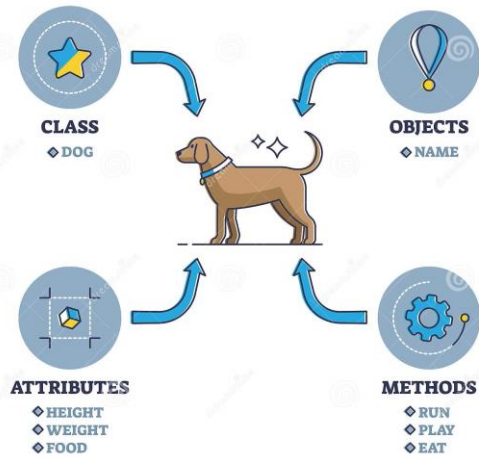
Orientado a Objetos

→ Foco nos Dados (objetos)





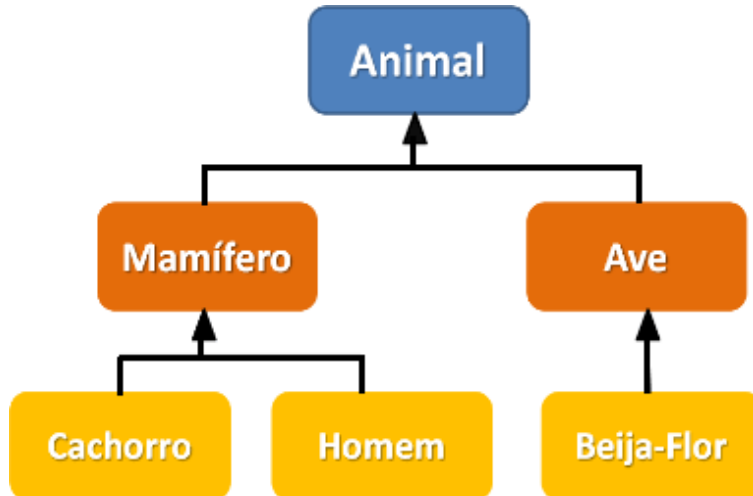
## OBJECT ORIENTED PROGRAMMING



dreamstime.com

ID 239724045 © VectorMine

## The Four Pillars



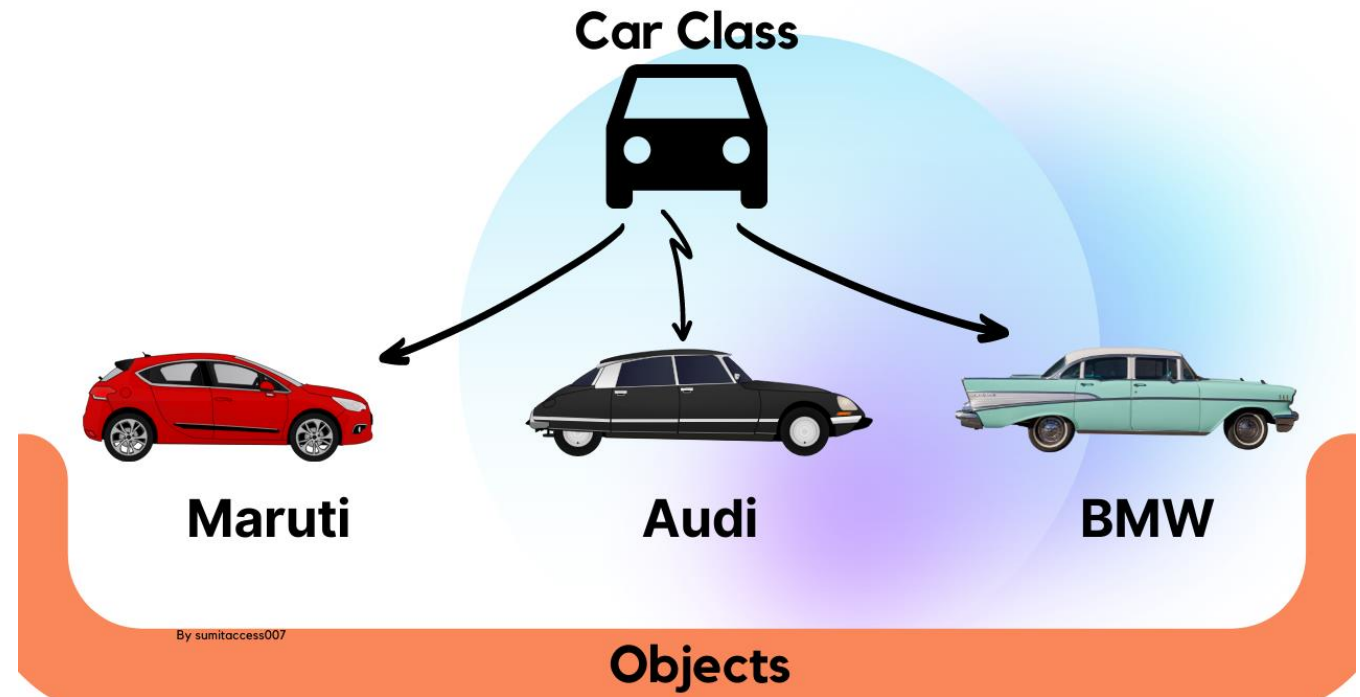
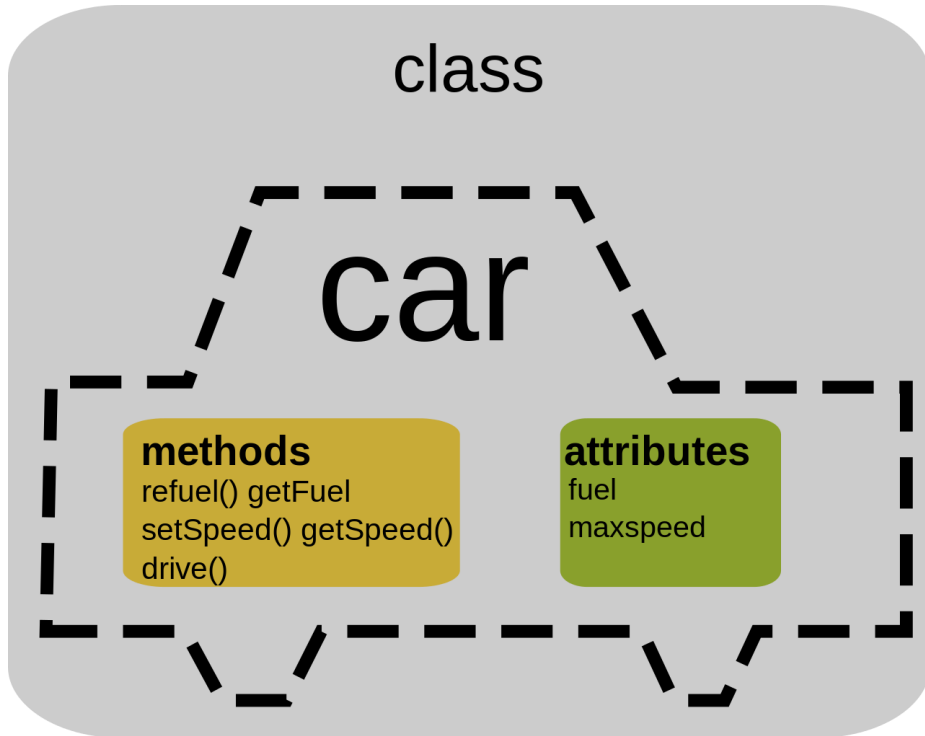
arbabwaseer@gmail.com

# Abstração

- **Abstração** é o processo de identificar e definir características essenciais de objetos do mundo real, representando essas características em forma de **classes** e **objetos**.
- Uma **classe** é uma estrutura que define atributos e métodos representando um tipo de objeto, fornecendo um modelo para criar instâncias desse tipo.
- Um **objeto** é uma instância de uma classe, caracterizado por seus atributos e comportamentos definidos na classe.
  - Ele pode interagir com outros objetos por meio de métodos e troca de dados.

Objeto é uma **instância** concreta de uma classe na POO.

# Abstração



# Encapsulamento

- **Encapsulamento** é a prática de ocultar os detalhes internos de um objeto, expondo apenas as operações relevantes para manipular esses detalhes. Isso promove a modularidade, a segurança e a manutenibilidade do código.
- Na maioria das linguagens de programação orientada a objetos, o encapsulamento é alcançado através da definição de **modificadores de acesso**.
  - **public, private, protected e default** (sem modificador)



# Herança

- **Herança** é o princípio que permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse), facilitando a reutilização de código, a organização hierárquica e a promoção de relações entre objetos.
- Quando há duas ou mais classes com atributos e métodos em comum, a herança facilita a criação de uma estrutura hierárquica. Nessa estrutura, uma classe "pai" define os elementos comuns que serão herdados pelas classes "filhas".

# Polimorfismo

- **Polimorfismo** é o princípio que permite que objetos instanciados a partir de sub-classes de uma mesma classe base possam invocar métodos que, apesar de terem a mesma assinatura, comportam-se de maneira diferente dependendo do tipo específicos do objeto.
- Com o Polimorfismo, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes.

# Paradigma Orientado a Objetos

## Linguagens de Programação

- Smalltalk – 1972
- Objective-C – 1984
- C++ - 1985
- Python\* - 1991
- Java – 1995
- Ruby – 1995
- C# - 2000
- Kotlin – 2011
- Swift - 2014

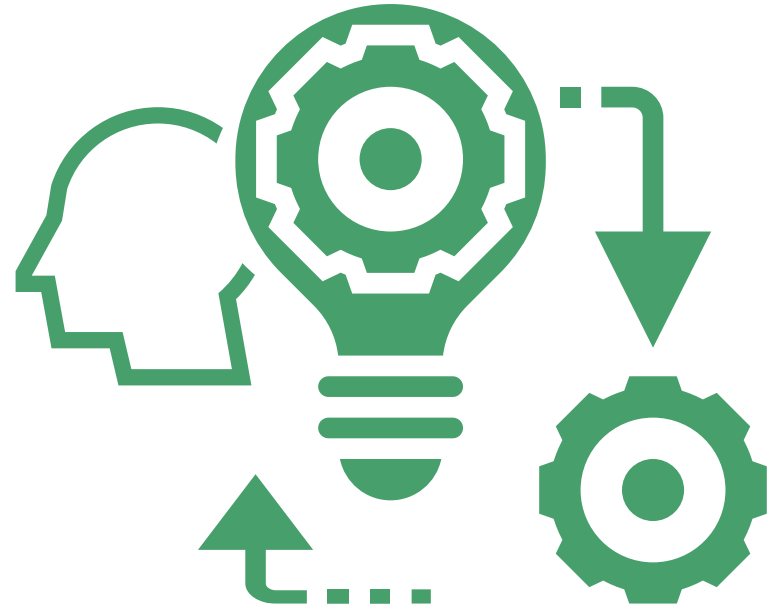
\* Embora seja uma linguagem multi-paradigma, Python oferece robusto suporte para programação orientada a objetos e é amplamente utilizada para esse fim.

# Pesquisa!!! –Em grupo



"A sabedoria da vida ensina que, o acúmulo de conteúdo das aulas, tende a permanecer acumulado, ao menos que uma força chamada prova mude seu estado de repouso." *by José Giombelli (estagiário IFSUL-PF) kkkk*





# Paradigmas Declarativos

Quais os principais?

Quais as linguagens predominantemente declarativas?

# Paradigmas de Programação

|

|— Imperativos

| |— Procedural

| | |— Estruturado

| |— Orientação a Objetos (OO)

|

|— Declarativos

|— Funcional

|— Lógico

# Paradigmas Declarativos

- **Características Principais**

## **1. Descrição do Que Fazer:**

Em vez de especificar um conjunto de passos para resolver um problema, você declara o que precisa ser feito. Por exemplo, em uma consulta de banco de dados, você descreve quais dados deseja recuperar sem definir explicitamente como o banco deve realizar essa recuperação.

## **2. Abstração de Implementação:**

O paradigma declarativo abstrai detalhes de implementação. O foco está na lógica de negócios e nos resultados desejados, enquanto o sistema lida com a execução e a otimização das operações necessárias.

## **3. Imutabilidade:**

Muitas linguagens declarativas enfatizam a imutabilidade, onde os dados não são alterados após a sua criação. Em vez disso, novas versões dos dados são criadas.

## **4. Menos Controle Sobre o Fluxo de Execução:**

Ao contrário dos paradigmas imperativos, onde você controla explicitamente o fluxo de execução do programa, o paradigma declarativo é mais sobre descrever o que deve ser feito e deixar o sistema lidar com o controle do fluxo.

# Paradigmas Declarativos

- Os principais Paradigmas Declarativos apontados pela maioria dos autores são:
- **Paradigma Funcional**
- **Paradigma Lógico**
- Alguns autores também citam o **Paradigma de Consulta** para linguagens como o SQL (Structured Query Language)



# Paradigma Funcional

- O **paradigma funcional** compreende um estilo de programação com alto **nível de abstração**, com soluções “elegantes”, concisas e poderosas.
- O foco principal está em como transformar e combinar funções para resolver problemas.

# Paradigma Funcional

## Características Principais

- **Expressões Lambda:**
- Funções anônimas (ou expressões lambda) são frequentemente usadas para operações de curto prazo e para passar funções como argumentos.
- Elas são particularmente úteis quando você deseja passar comportamentos como argumentos para métodos ou quando está trabalhando com coleções e quer realizar operações funcionais.

# Expressões Lambda em JavaScript

- Em JavaScript, funções lambda são conhecidas como **funções de seta** (**arrow functions**). A sintaxe é bastante simples e concisa.

```
const functionName = (parameters) => expression;
```

# Expressões Lambda em JavaScript

```
JS LambdaFunctions.js > ...
1 //Sem Parâmetros
2 const greet = () => console.log("Hello, World!");
3 greet(); // Saída: Hello, World!
4
5 //Com um parâmetro
6 const square = x => x * x;
7 console.log(square(5)); // Saída: 25
8
9 //Com múltiplos parâmetros
10 const add = (x, y) => x + y;
11 console.log(add(3, 4)); // Saída: 7
12
13 //Com bloco de código
14 const multiplyAndPrint = (x, y) => {
15     const result = x * y;
16     console.log(result);
17 };
18 multiplyAndPrint(3, 4); // Saída: 12
19
20 //Passando como parâmetro para uma Função de Ordem Superior
21 const numbers = [1, 2, 3, 4];
22 const doubled = numbers.map(num => num * 2);
23 console.log(doubled); // Saída: [2, 4, 6, 8]
```

# Expressões Lambda em Python

- Em Python, funções lambda são conhecidas como **expressões lambda**. A sintaxe também é simples.

```
lambda parameters: expression
```

# Expressões Lambda em Python

```
LambdaFunctions.py > ...  
1   #Sem parâmetros  
2   greet = lambda: "Hello, World!"  
3   print(greet()) # Saída: Hello, World!  
4  
5   #Com um parâmetro  
6   square = lambda x: x * x  
7   print(square(5)) # Saída: 25  
8  
9   #Com múltiplos parâmetros  
10  add = lambda x, y: x + y  
11  print(add(3, 4)) # Saída: 7  
12  
13  #Passando como parâmetro para uma Função de Ordem Superior  
14  numbers = [1, 2, 3, 4]  
15  doubled = list(map(lambda x: x * 2, numbers))  
16  print(doubled) # Saída: [2, 4, 6, 8]
```

# Expressões Lambda em Java

- Expressões lambda são uma característica poderosa em Java que permite criar funções anônimas de maneira concisa. Elas são particularmente úteis quando você deseja passar comportamentos como argumentos para métodos ou quando está trabalhando com coleções e quer realizar operações funcionais.

```
(parameters) -> expression
```

```
(parameters) -> { statements; }
```

# Expressões Lambda em Java

```
//Sem parâmetros
Runnable greet = () -> System.out.println("Olá, Mundo!");
greet.run(); // Saída: Olá, Mundo!

//Com um parâmetro
Function<Integer, Integer> square = x -> x * x;
System.out.println("O quadrado de 5 é: " + square.apply(5));

//Com múltiplos parâmetros
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println("A soma de 10 e 20 é: " + add.apply(10,10)); // Saída: A soma de 10 e 20 é: 30

//Com Bloco de código
BiFunction<Integer, Integer, String> divisibilityCheck = (a, b) -> {
    if (b == 0) {
        return "Divisão por zero não é permitida!";
    }
    return a % b == 0 ? a + " é divisível por " + b : a + " não é divisível por " + b;
};
System.out.println(divisibilityCheck.apply(10, 2)); // Saída: 10 é divisível por 2
System.out.println(divisibilityCheck.apply(10, 3)); // Saída: 10 não é divisível por 3
```



# Interfaces Funcionais em Java

- As interfaces funcionais em Java são interfaces que possuem exatamente um método abstrato, tornando-as ideais para uso com expressões lambda e referências de método.
- A introdução dessas interfaces foi uma das grandes mudanças no Java 8, como parte da introdução de recursos funcionais na linguagem.

# Principais Interfaces Funcionais em Java

- `Function<T, R>`
  - Descrição: Recebe um argumento de tipo T e retorna um resultado de tipo R.
  - Método Abstrato: `R apply(T t)`
- `BiFunction<T, U, R>`
  - Descrição: Recebe dois argumentos de tipos T e U e retorna um resultado de tipo R.
  - Método Abstrato: `R apply(T t, U u)`

# Principais Interfaces Funcionais em Java

- `Consumer<T>`
  - Descrição: Recebe um argumento de tipo T e não retorna nenhum resultado. É frequentemente usado para realizar operações em um objeto.
  - Método Abstrato: `void accept(T t)`
- `BiConsumer<T, U>`
  - Descrição: Recebe dois argumentos de tipos T e U e não retorna nenhum resultado.
  - Método Abstrato: `void accept(T t, U u)`

# Principais Interfaces Funcionais em Java

- `Supplier<T>`
  - Descrição: Não recebe nenhum argumento, mas retorna um resultado de tipo T. É frequentemente usado para gerar ou fornecer valores.
  - Método Abstrato: `T get()`

# Principais Interfaces Funcionais em Java

- Predicate<T>
  - Descrição: Recebe um argumento de tipo T e retorna um valor booleano. É frequentemente usado para testar se um objeto satisfaz uma condição.
  - Método Abstrato: `boolean test(T t)`
- BiPredicate<T, U>
  - Descrição: Recebe dois argumentos de tipos T e U e retorna um valor booleano.
  - Método Abstrato: `boolean test(T t, U u)`

# Principais Interfaces Funcionais em Java

- `UnaryOperator<T>`
  - Descrição: É uma especialização de `Function` que recebe e retorna o mesmo tipo `T`.
  - Método Abstrato: `T apply(T t)`
- `BinaryOperator<T>`
  - Descrição: É uma especialização de `BiFunction` que recebe dois argumentos do mesmo tipo `T` e retorna um resultado do mesmo tipo `T`.
  - Método Abstrato: `T apply(T t1, T t2)`

# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe:**
- Funções podem ser passadas como argumentos para outras funções, retornadas como valores de outras funções e atribuídas a variáveis. Isso permite uma programação altamente modular e reutilizável.

# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior:**
- Funções que aceitam outras funções como parâmetros ou retornam funções como resultados. Isso permite a criação de abstrações poderosas e reutilizáveis.



# Funções de Ordem Superior – Lambdas Functions

@Configuration

```
public class ConfigSecurity {
```

@Bean

```
SecurityFilterChain getFilterChain(HttpSecurity http, AuthTokenFilter authTokenFilter) throws Exception {  
    http  
        .csrf(csrf -> csrf.disable())  
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))  
        .authorizeHttpRequests(auth -> auth  
            .requestMatchers("/ws**", "/ws/**").authenticated()  
            .anyRequest().permitAll())  
        .addFilterBefore(authTokenFilter, UsernamePasswordAuthenticationFilter.class);  
    return http.build();  
}
```

# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior;**
- **Imutabilidade:**
- Dados são imutáveis, o que significa que uma vez criados, não podem ser alterados.
- Em vez de modificar um dado existente, você cria uma nova versão do dado.
- Essa imutabilidade facilita a programação funcional, promovendo a ausência de efeitos colaterais e permitindo que funções sejam puras.
- Linguagens como Haskell e Elm, que são puramente funcionais, aplicam rigorosamente esse conceito.

# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior;**
- **Imutabilidade;**
- **Funções Puras:**
  - Funções são consideradas puras se para uma determinada entrada, sempre retornam a mesma saída e não têm efeitos colaterais (não alteram estados externos).

# Função Pura x Impura

```
public static int add(int a, int b) {  
    return a + b;  
}  
  
public static void main(String[] args) {  
    int result1 = add(2, 3); // Sempre retorna 5  
    int result2 = add(2, 3); // Sempre retorna 5  
  
    System.out.println(result1); // 5  
    System.out.println(result2); // 5  
}
```

```
private static int counter = 0;  
  
public static int addAndIncrementCounter(int a, int b) {  
    counter++; // Modifica o estado externo (variável global)  
    return a + b;  
}  
  
public static void main(String[] args) {  
    int result1 = addAndIncrementCounter(2, 3); // Resultado 5, counter = 1  
    int result2 = addAndIncrementCounter(2, 3); // Resultado 5, counter = 2  
  
    System.out.println("Result 1: " + result1); // 5  
    System.out.println("Result 2: " + result2); // 5  
    System.out.println("Counter: " + counter); // 2  
}
```

# Dados Globais Mutáveis e Funções Impuras

- **Dados globais mutáveis e funções impuras** são fatores que podem introduzir efeitos colaterais e, em particular, criar problemas significativos em ambientes de programação multithread, onde várias threads executam em paralelo.



# Um breve parênteses sobre Multithreads

- A programação multithreaded envolve a criação e o gerenciamento de múltiplas threads dentro de um único processo para realizar tarefas simultâneas.
- Cada thread pode ser vista como uma "fila" de instruções ou tarefas que precisa ser processada.
- Em um ambiente multithreaded, múltiplas filas (threads) podem ser processadas em paralelo, especialmente em sistemas com múltiplos núcleos de CPU.

# Um breve parênteses sobre Multithreads

- **Benefícios da Programação Multithreaded:**

- 1.Melhor Utilização de Recursos:** Threads podem executar em paralelo em múltiplos núcleos de processador, aumentando a eficiência e a utilização do hardware disponível.
- 2.Melhoria de Desempenho:** Em tarefas que são independentes ou que podem ser paralelizadas, o uso de múltiplas threads pode reduzir o tempo total de execução.
- 3.Responsividade:** Em aplicações com interface gráfica (GUI), o uso de threads pode manter a interface responsiva ao realizar operações em background (como leitura de arquivos ou operações de rede).

# Dados Globais Mutáveis e Funções Impuras

- **Efeitos Colaterais:** Em um ambiente multithread, se uma função impura modifica o estado global ou depende dele, diferentes threads podem observar estados diferentes ou interferir nas alterações uma da outra.
- **Problemas Específicos:**
  - **Condições de Corrida (Race Conditions):** Quando múltiplas threads acessam e modificam recursos compartilhados simultaneamente sem a devida sincronização, resultados inesperados podem ocorrer.
  - **Deadlocks:** Ocorrem quando duas ou mais threads ficam bloqueadas esperando por recursos que nunca serão liberados, levando a um impasse no sistema.
  - **Dificuldade de Depuração:** Bugs em programas multithreaded podem ser difíceis de reproduzir e depurar, pois o comportamento do programa pode variar entre execuções.



# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior;**
- **Imutabilidade;**
- **Funções Puras;**
- **Composição de Funções:**
  - Funções podem ser combinadas para formar funções mais complexas. A composição permite criar funções novas a partir de funções existentes.

# Paradigma Funcional

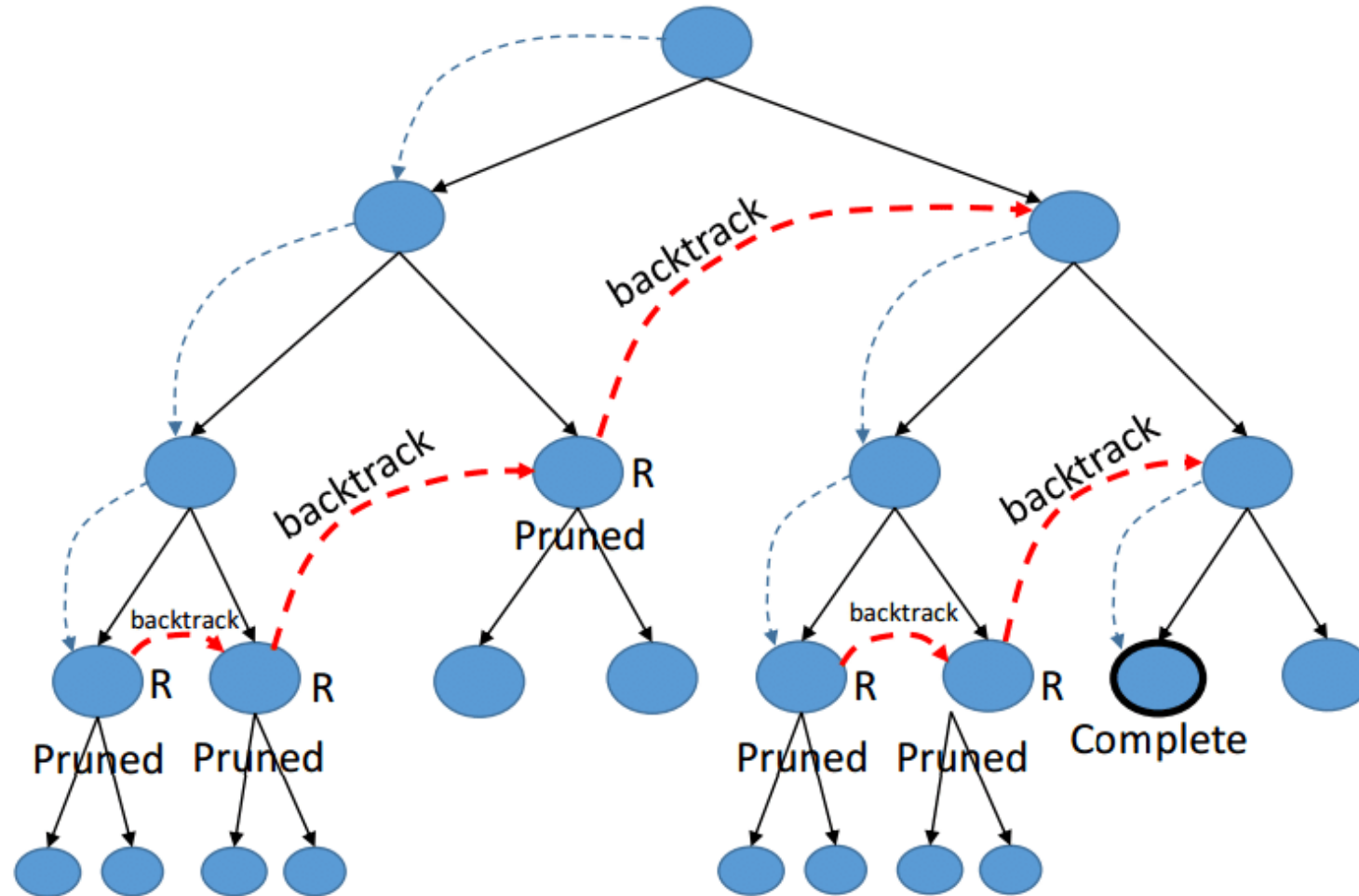
## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior;**
- **Imutabilidade;**
- **Funções Puras;**
- **Composição de Funções;**
- **Recursão:**
- Como as estruturas de controle de loop (como for e while) são menos comuns em programação funcional, a recursão é frequentemente usada para iterar sobre dados. ➔ Funções: *map, filter, reduce*.

# Recursão -Utilização

- **Problemas que podem ser decompostos em subproblemas menores**
  - **Divisão e conquista:** Problemas como a ordenação por merge sort ou quicksort, onde o problema é dividido em subproblemas menores de forma recursiva, são naturalmente adequados para recursão.
  - **Pesquisa binária:** Em árvores ou listas ordenadas, a pesquisa binária é implementada de forma recursiva, dividindo a lista ao meio a cada passo.
- **Trabalhando com estruturas de dados recursivas**
  - **Árvores e Grafos:** Muitas operações em árvores, como percorrer (in-order, pre-order, post-order) ou buscar em uma árvore binária, são implementadas naturalmente de forma recursiva.
  - **Problemas de backtracking:** Problemas como a solução de labirintos, sudoku, ou o problema das N rainhas são comumente resolvidos usando recursão para explorar diferentes possibilidades.

# Recursão –Utilização (Backtracking)



# Recursão -Utilização

- **Quando a solução recursiva é mais simples e intuitiva**
  - **Simplicidade do código:** Em alguns casos, a solução recursiva é mais fácil de escrever e entender do que a iterativa, mesmo que a recursão possa ser menos eficiente em termos de memória. Um exemplo é o cálculo de potências em certos algoritmos matemáticos.
- **Problemas que têm uma definição recursiva natural**
  - **Sequência de Fibonacci:** Como mencionado anteriormente, Fibonacci é definido recursivamente e pode ser implementado da mesma forma, embora com otimizações como memoization ou abordagem iterativa.
  - **Fatorial:** O cálculo do fatorial de um número é recursivo por natureza:

$$n! = n \times (n-1)!$$

# Recursão –Utilização (Fatorial)

```
public static long fatorialIterativo(int n) {  
    long resultado = n;  
    for (long i = n - 1; i > 1; i--) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

# Recursão –Utilização (Fatorial)

```
public static long fatorialIterativo(int n) {  
    long resultado = n;  
    for (long i = n - 1; i > 1; i--) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

```
public static long fatorialRecursivo(int n) {  
    if (n == 0 || n == 1) return 1;  
    return n * fatorialRecursivo(n - 1);  
}
```

# Paradigma Funcional

## Características Principais

- **Expressões Lambda;**
- **Funções como Cidadãos de Primeira Classe;**
- **Funções de Ordem Superior;**
- **Imutabilidade;**
- **Funções Puras;**
- **Composição de Funções;**
- **Recursão:**



# Paradigma Funcional

## Linguagens de Programação

- Lisp – 1958
- ML (Meta Language) – 1973
- Erlang – 1986
- Haskell – 1990
- Scala – 2003
- F# - 2005

# Entregáveis

Vamos lá. Está na hora!!!



