# Convenções e Conceitos

- Nome de classe: sempre maiúsculo

- Nome de métodos: sempre minúsculo

- NÃO é orientado a identação, mas é sempre bom manter a organização

- Aspas duplas é String e simples é char

**Classes**

- As classes são organizadas em pacotes

Ex: A classe Scanner existe dentro do pacote java.util

*Por padrão, a única classe que é automaticamente importada em todo arquivo Java é java.lang(contém classes fundamentais como String, System, Integer, etc)*

**Criar pacotes e classe**

- Criar packeges: clica com o botão direito no arquivo e escolha new java packege

- Criar classe: clica com o botão direito no arquivo e vai em New Java File – Class

- Bloco de comentário: /\* e \*/

- Comentário simples: //

**Importante:**

println 🡪 imprime o valor com \n ou seja pula pra linha de baixo

print 🡪 só imprime o valor

printf 🡪 É com o format

O Scanner em Java é uma classe fundamental usada para **obter entrada de dados de várias fontes**, principalmente do teclado (entrada padrão, System.in) ou de arquivos. Usa delimitador (por padrão é o espaço em branco)

**Explicando a linha principal**

public static void main(String[] *args*)

* O java sempre procura esse método para começar a executar o programa

**static**: Quando um método ou variável é static, ele pertence a classe em si e não a uma instancia (objeto) específica da classe

**Por que main é static?** Porque a JVM precisa chamar main para iniciar seu programa sem ter que criar um objeto da sua classe primeiro.

**void:** é uma palavra-chave que indica que um método não retorna nenhum valor

**Convenção**

private final static String WELCOME\_MESSAGE = "Olá mundo"; *//por ser privado nenhuma outra classe pode acessar, só essa. o final é como uma constante, o valor não pode ser alterado*

**Importante:** Por convenção em Java, constantes (final static) são nomeadas em **letras maiúsculas** com sublinhados para separar palavras (ex: WELCOME\_MESSAGE, MAX\_VALUE

# Keywords

- Inteiro: byte, short, int, long

- Ponto Flutuante: float, double

- Caractere: char

- Booleano: Boolean

# Operadores

**Lógicos e Atribuição**

|| 🡪 Ou

&& 🡪 E

!= 🡪 Diferente de

! 🡪 Not (vai ser ao contrário)

**Sobre operadores Aritméticos:**

🚨No Java se tentar dividir dois números inteiros, ele irá retornar um inteiro, mesmo que o resultado seja decimal. Por isso, é importante receber a entrada como Float

- O java também aceita valor+= ou valor-=

- Para incrementar ++valor **importante:** se declarar o sinal a esquerda valor-- ele só decrementa na próxima linha, se for a direita --valor ele já retorna na mesma linha

Operadores Bitwise (Bit-a-Bit)

* São usados para manipular e testar bits individuais de números inteiros
* Trabalham com a representação binária do número
* Pouco usado
* Mais usado em sistema embarcado

- **xor** 🡪 é representado por ^. Quando os números são iguais ele retorna 0 e se for diferente retorna 1

- **complement** 🡪 é representado por ~. Ele funciona como o NOT, ou seja, inverte todos os valores

- **left shift operator** 🡪 é representado por <<. Desloca o número para esquerda e preenche espaço vazio com 0.

Ex: 9 tem o binário 1001

Se deslocar duas posições então ele completará: 100100

9 << 2

- **rigth shift operator** 🡪 é representado por >>.Funciona da mesma forma que o left, só que desloca para a direita e ao completar o número ele verifica, se for negativo (ex: -8) ele completa com **1**, se for positivo ele completa com **0**

Ex: -8 tem binário 111111111111111111111000

Se deslocar duas posições ele completará 11111111111111111111111110 porque é negativo

-8 >> 2

Agora, 8 tem binário 1000

Se deslocar duas posições ele completará 0010 restando só 10

8 >> 2

**- Unsigned rigth shift operator** 🡪 é representado por >>>. Igual ao rigth, mas ele sempre completa com 0, independente se for negativo ou positivo

# Estrutura Condicional

**Operador Ternário** 🡪 ? :

- Funciona em uma linha fazendo atribuição, para maiores códigos de lógica é melhor usar o bloco if / else

var canDrive = (age >= 18) || (age >= 16 && isEmancipated);

        System.out.println(canDrive ?

                name + ", você pode dirigir" :

                name + ", você não pode dirigir");

**Importante:** Em java não podemos atribuir sinal de igual = entre Strings, deve usar .equals e IgnoreCase se quiser levar em conta maiúsculas e minúsculas

**Estrutura de repetição do e while:**

- **While:** primeiro ele verifica se a condição já foi atendida, depois ele executa;

Então se primeiro eu passar

var nome = “exit”

e depois no while fizer

while (!nome.equalsIgnoreCase(“exit”))

NÃO vai funcionar, pois a condição já foi atendida antes do while executar

*obs: pode funcionar com o while true e depois passar a condição no if*

**- Do:** primeiro ele executa e depois verifica, então nesse exemplo acima ele VAI funcionar

var nome = “exit”;

do{

System.out.println(“informe um nome”)

if (name.equalsIgnoreCase("exit")) break;

}

# Orientação a objetos

- Variáveis estáticas NÃO são acessíveis pelo this (pois pertencem a classe e não ao objeto)

- Se quiser que um valor não seja alterado, deve passa-lo no construtor e atribuir final

- Ao trabalhar com setter usamos this

**Records**

- São classes imutáveis

- só é possível declarar atributos estáticos

- tem métodos (getters automáticos)

- Adicionamos os atributos no construtor

- Todo atributo declarado no record é privado

**Para estruturar a classe:**

Getter:

- public tipo getAlgumaCoisa()

Métodos e setter:

- public void addWater()

- public void setClean(boolean clean)

**Importante:** para conseguir acessar um método sem criar um objeto ele deve ser estático

**Em Java, o uso de** this **é opcional** **quando não há ambiguidade entre variáveis.**

**Não** há ambiguidade:

private int speed;

public void accelerate() {

speed += 1; // aqui "speed" já se refere ao atributo da classe

}

Quando há ambiguidade:

public void setSpeed(int speed) {

this.speed = speed; // agora é necessário usar "this" pra diferenciar

}

**Importante:** O return; (sem valor) é usado em métodos void pra **encerrar a execução do método ali**. Isso evita que o resto do método continue.

* quando o if é só um aviso mas o resto do método **ainda pode continuar**, não precisa usar return.

**Overloading**: São duas versões do mesmo método

# ATENÇÃO!!

**Else if e mais de um if seguido**

**else if:** só é testado se o anterior for falso

**mais de um if:** todos são testados. É ótimo para quando mais de uma sentença é verdadeira

# Herança e Polimorfismo

* UMA CLASSE ABSTRATA NÃO PODE SER INSTANCIADA

- Ela serve como modelo para outras classes.

- É bom para evitar que alguém crie um objeto genérico demais (como new Employee())

- Mas pode ter uma classe do “tipo” dela. Ex: a classe gerente é do tipo colaborador (abstrata)

- Ao definir a classe como final não mais pode estender. É a última da hierarquia

- Quando a classe NÃO é abstrata, ela é concreta

**Classes sealed:**

Definir a classe como sealed, permite acesso/extensão, mas com regras específicas para as subclasses. A classe é “selada”

- As subclasses devem ser colocadas como non-sealed para funcionar

Employee:

public sealed abstract class Employee permits Manager, Salesman

Manager e Salesman:

public non-sealed class Manager extends Employee

Garante mais segurança e proíbe classes que não seguem a lógica herdarem da classe pai. Como proibir a classe Client de acessar Employee.

**Polimorfismo:** Na prática em Java, **polimorfismo é a capacidade de uma variável de referência de um tipo pai (superclass) apontar para objetos de tipos filhos (subclasses)**.

Employee manager = new Manager(); *//classe gerente do tipo colaborador*

- tratar objetos diferentes como se fossem do mesmo tipo base. (tratar manager e salesman como employee)

**Record:** Não pode ter extends e, por isso, não funciona na Herança

🚨Métodos static não podem ser passados por herança e nem são sobrescritos

**Variáveis podem ser públicas, privadas ou protegidas**

**- Não precisa ficar escrevendo os getters/setters (são automáticos) no record**

**Protect** -> permite acesso a subclasses, classes herdeira e classes no mesmo package. Se uma classe NÃO herdeira estiver fora do package e tentar acessar, vai dar erro.

**SOBRECARGA DE MÉTODO**

|  |
| --- |
|  |

|  |
| --- |
| * Criar vários métodos com o mesmo nome, mas com parâmetros diferentes |

* Dentro da **mesma** classe
* Recebe outros argumentos (pode ser de tipos diferentes ex: o primeiro é int e o segundo double)

public double getFullSalary() { //sem argumentos

        return this.salary + (soldAmount \* percentPerSold)/100;

    }

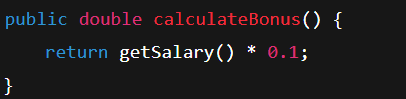
    public double getFullSalary(double *extra*){ *//\* A sobrecarga de metodos também pode receber mais de um arg*

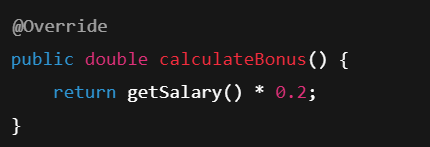
        return getFullSalary() + *extra*;

    }

**SOBRESCRITA DE MÉTODO**

* Redefine um método que existe na superclasse
* **@Override**

Na classe pai Na subclasse (filha)



# Quando usar classe abstrata

**Criar um molde base**

- Ex: criar o método *fazerSom* na superclasse Animal, e depois implementa nas subclasses o som que cada animal faz chamando esse método

public abstract class Animal {

    String nome;

    public void dormir() {

        System.out.println("Dormindo...");

    }

    public abstract void fazerSom(); *// cada animal faz um som diferente*

}

**Garantir que uma função seja implementada nas subclasses**

**- Métodos abstratos** obrigam as classes filhas a **implementarem** esses métodos

- São sobrescritos na superclasse

✅ **Classe abstrata:**

* Não pode ser instanciada.
* Serve de modelo/base para outras classes.
* Geralmente representa conceitos genéricos demais pra existirem sozinhos.

# Interface

- Só aceita métodos abstratos

- TODAS as propriedades que formos declarar automaticamente serão: **públicas, final e static**

- pode implementar mais de uma interface

- NÃO pode ter construtores

- Toda interface é **implicitamente abstrata**, então **você não precisa (nem pode) colocar** abstractna frente dela. Ou seja, a classe que implementar a interface tem que usar todos os seus métodos

**IMPORTANTE:** Desde o Java 8+, **interfaces podem ter métodos com implementação** usando a palavra-chave default. Esses **não precisam ser sobrescritos**

**Quando utilizar?**

* Se quiser reutilizar comportamento: Várias classes diferentes precisam implementar o mesmo conjunto de ações, mas cada uma do seu jeito (Override)
* Ideal pra comportamento comum entre classes não relacionadas

Default 🡪 permite fornecer uma implementação padrão. Sem obrigar as classes a sobrescreverem. A interface pode ter vários métodos default

**Interface Funcional:** é a interface com um único método abstrato

- Opcional usar a anotação @FunctionalInterface.Mas DEVE ter apenas 1 método abstrato

- ex: Function<T,R>, Consumer<T>

**Function<User, String> //recebe User e retorna String**

**Lambda:** **função sem nome** (função anônima) que pode ser passada como **parâmetro (ao invés de criar um método só pra passar um comportamento)**

**Classe Nested: é só uma classe dentro de outra. Pode ser estática ou não-estática.**

**- a classe interna pode acessar membros privados da classe externa**

**- se a classe interna for static, não precisa da instancia da externa. E por não ter a instancia, só acessa os métodos static da externa**

**- classes externas NÃO podem ser static. Porque** você só pode usar static em **contextos onde há algo "pai**

**- Não é tão comum porque ter várias classes internas gera complexidade no código!**

**Ex:**

public class Externa {

    private String msg = "Olá";

    class Interna {

        void mostrar() {

            System.out.println(msg); *// acessa direto*

        }

    }

**Na main:**

Externa = new Externa();

Externa.Interna interna = externa.new Interna();

interna.mostrar();

**Final:** uma classe final não pode ser herdada. E um atributo final não pode ser alterado

Importação Estática

- Usado para importar métodos sem escrever o nome da classe

- deixa mais limpo quando você usa vários métodos ou constantes da mesma classe estática.

# Listas e Array

- Se quiser pode criar sua própria Lista, mas deve implementar os métodos obrigatórios e fazer os imports

- O java tem vários tipos além do Array

- Deve definir o tamanho do array ao criá-lo, ou pode inicializar na criação. E não é possível modificar o tamanho adicionando mais itens

**Int ou Integer?**

🚨 Ao trabalhar com Array de objetos **NÃO** é possível trabalhar com o tipo int. **Deve** usar o objeto Integer

* Se usar **int** como em int[] numeros = new int[5]; estamos trabalhando com tipo primitivo (só permite int). Também é mais leve
* Se usar **Integer** como em Integer[] numeros = new Integer[5]; estamos trabalhando com objetos. É útil quando trabalhamos com coleções genéricas e aceita o null

**Listas/ Arrays Dinâmicos**

List<Integer> lista = new ArrayList<>();

* Cria uma lista dinâmica e manipula elementos com .add(), .remove(), .contains(), .sort()
* os **genéricos** (<>) do Java **só funcionam com objetos. Então sempre usa Integer e NUNCA int ou tipos primirivos**

**LinkedList<>** é mais usado quando há muitas inclusão e remoção de itens do array, traz uma melhor performance

**Vector<>** indicado em cenários multi threads. Usado quando **precisa de sincronização automática**, e tá lidando com **código simples de múltiplas threads. ULTRAPASSADO**

**IMPORTANTE:** Ao utilizar .equals() em objetos, ele verifica se o endereço de memória de um objeto é igual ao endereço de memória do outro

**Cast 🡪** é o ato de converter um tipo de dado em outro.

*Object obj = new User();*

*User u = (User) obj;*

Com **pattern machine** o cast fica: if (obj instanceof User user)

# Trabalhando com Set

- Set NÃO permite duplicidades

- É impresso de forma aleatória

- Não acessa por índice

HashCode(): É um número gerado para representar um **objeto** de forma rápida

- Quando o set usa o Hash ele precisa do equals e do hashcode para funcionar de forma correta

# Trabalhando com Map e Wrappers

**Map**

- é uma coleção de pares chave–valor (*key–value pairs*)

- não usa índices como List, mas sim chaves. (Quando você faz map.get(chave), o Java procura diretamente pela chave)

- Se você tentar inserir uma chave que já existe, o valor antigo é **substituído** pelo novo.

- NÃO tem ordem garantida

O **Map** em Java é uma interface (tipo genérico), e **TreeMap**, **LinkedHashMap**, **HashMap** etc. são **implementações** dessa interface.

**TreeMap<>()**

- **Por padrão**: usa a ordem natural da chave (Ex.: String → ordem alfabética, Integer → ordem crescente mesmo q você passe fora da ordem).

- **Se você passar um Comparator no construtor** ele vai usar essa lógica para ordenar (pode personalizar se é decrescente, por tamanho da string etc)

TreeMap<String, Integer> map2 = new TreeMap<>(Comparator.reverseOrder());

- **NÃO aceita chave** null (porque precisa comparar).

**HashMap():**

- Armazena sem ordem garantida.

- Mais rápido na maioria dos casos.

- Aceita 1 chave null e vários valores null.

**LinkedHashMap():**

- Mantem a ordem de inserção

- funciona igual o hash() mas mantendo a ordem

Métodos Importantes:

**- merge()** serve para **inserir** ou **atualizar** um valor no Map de forma inteligente

Map.merge(chave, valorSeNaoExistir, funcaoSeExistir)

**- size()** no Map retorna **quantos pares chave–valor existem** no mapa

System.out.println(users.size()); // se tiver 4 elementos, imprime 4

**Wrappers**

**Wrappers** são classes que "envolvem" os tipos de dados primitivos. Int 🡪 Integer, char 🡪 Character, double 🡪 Double etc

**Autoboxing:** faz a **conversão automática entre tipos primitivos** (como int, double, boolean) e suas **classes wrapper** (como Integer, Double, Boolean)

**Unboxing:** é o contrário. Converte o objeto de volta para o tipo primitivo

int meuPrimitivo = 10;

Integer meuObjeto = meuPrimitivo; // Autoboxing acontece aqui!

* Passa o 10 de int para Integer
* NUNCA faça unboxing de um objeto null

# String builder e String buller

- Uma String comum é imutável, e ao tentar alterá-la (como na concatenação) o java cria um outro objeto em memória (péssimo em loops)

- String builder/buffer é mutável, e o conteúdo é alterado no mesmo objeto

* Funciona de forma semelhante, com uma diferença:

**String Builder**: uma thread (**não é sincronizado**. Mais rápido, mas inseguro se várias threads alterarem ao mesmo tempo.

**String Buffer:** várias threads (**é sincronizado** - só **uma thread** pode modificar aquele objeto por vez)

# Classe Big Decimal, Enum e Classe Optional

- Em casos que é preciso muita precisão de dados, não é bom trabalhar com o float e o double

**Classe BigDecimal()** 🡪 para trabalhar com precisão

- recomenda trabalhar com string

- soma: add()

- subtração: subtract()

- RoundingMode 🡪 arredonda o número UP para cima, se depois da vírgula for maior que zero (se tiver 1.1 arredonda para 2) DOWN arredonda para baixo. Ver as outras formas de arredondamento na documentação

**Enum** 🡪 tipo especial de classe para trabalhar com conjunto fixo e limitado de valores

Exemplo: operações matemáticas, estados do brasil, dia da semana (é fixo, limitado e o usuário não muda)

**Classe** **Optional** 🡪 é um container para **evitar NullPointerException** e **deixar claro** que um valor pode estar ausente.

- serve para **forçar** você a lidar com a possibilidade de ausência de valor de forma explícita, usando métodos como .isPresent(), .orElse(...), .map(...), .ifPresent(...)

- Alguns frameworks (como Spring) retornam optional

✅ **Bom para**:

* Retorno de métodos que podem não ter valor.
* Evitar null explícito no retorno.
* Forçar o programador a lidar com ausência de valor.

Map 🡪 transforma um valor e retorna um novo Optional com o valor transformado:

optional.map(u -> u.getName());

# API de Streams

**Inspirada em programação funcional**, tornando o código mais legível, expressivo e menos propenso a erros ao manipular listas, conjuntos e outros tipos de coleções. É bom porque evita ficar criando loops com for toda hora

Use Streams quando:

* Você quer escrever código mais **declarativo** (focado no *o que* fazer, não no *como*).
* Vai processar coleções de forma **encadeada** (filtrar, mapear, ordenar, agrupar).

O Stream.generate é um jeito de criar **Streams infinitas ou sob demanda**, geradas a partir de uma **função fornecida** (um Supplier).

🚨**DEVE limitar o tamanho** com limit() para não criar um loop infinito.

**Operações intermediárias** 🡪transformam o Stream sem produzir resultado final. Pode retornar outro Stream

filter → filtra elementos

map → transforma elementos

sorted → ordena

distinct → remove duplicados

peek → inspeciona elementos (útil para debug)

**Operações Terminais** 🡪 Produzem um resultado final ou efeito colateral, executam as operações intermediárias. Depois de uma terminal, o Stream não pode mais ser usado

forEach → itera sobre elementos

collect → transforma Stream em coleção (List, Set, Map)

count → conta elementos

reduce → agrega os elementos em um único valor

**Alguns métodos:**

O compareTo é o método que compara strings alfabeticamente (retorna número negativo se a primeira for "menor", positivo se for "maior" e zero se iguais).

users.sort((u1, u2) -> u1.name().compareTo(u2.name()));

compara o nome dos usuários para colocar em ordem alfabética. É o mesmo que Comparator.comparing(User::name)

**allMatch(), anyMatch, noneMatch** 🡪 testam alguma condição, **retornando true ou false.** Se quiser imprimir na lista deve usar o .filter() e .toList()

boolean temMulher = users.stream().anyMatch(u -> u.sex() == FEMALE);

- Retorna true se tiver mulher

boolean temMulher = users.filter(u 🡪 u.stream()

.anyMatch(u -> u.sex() == FEMALE)).toList();

- Filtra os usuários que são mulheres e coloca na lista

**sort()** 🡪 é um método de List. Modifica a lista original, organizando seus elementos

**sorted()** 🡪 É um método de Stream. NÃO modifica a lista original, ele cria um novo fluxo ordenado, pode continuar processando ou coletar em uma nova lista

List<Integer> numeros = Arrays.asList(5, 3, 1, 4, 2);

*// sort (muda a lista original)*

Collections.sort(numeros);

System.out.println(numeros); *// [1, 2, 3, 4, 5]*

*// sorted (gera um novo fluxo)*

List<Integer> numerosOrdenados = numeros.stream()

        .sorted()

        .toList();

System.out.println(numeros);          *// [1, 2, 3, 4, 5] (não mudou)*

System.out.println(numerosOrdenados); *// [1, 2, 3, 4, 5]*

# Generics

**Generics** são um recurso que permite que você escreva classes, interfaces e métodos que podem **operar em objetos de diferentes tipos**

GenericDAO<T> significa que a classe pode operar em qualquer tipo:

GenericDAO<String>: O DAO (Data Access Object) vai trabalhar com objetos String.

GenericDAO<User>: O DAO vai trabalhar com objetos de uma classe User.

**Pesquisar mais sobre:**

**ORM (Object-Relational Mapping):** Ferramentas como Hibernate ou JPA (Java Persistence API) 🡪 Simplifica a manipulação de dados,mapeando automaticamente objetos Java para tabelas de banco de dados, evitando escrever GenericDAO

# API Date and Time

**-** LocalDate, LocalDateTime, ZonedDateTime — são **classes** dentro dessa API

**O que é offset (ZoneOffset)?** É uma **diferença fixa** entre o horário local e o **UTC**.

- É representado como +HH:mm ou -HH:mm.

Exemplo:

* *Brasília normalmente é -03:00.*
* *Londres no inverno é +00:00 (igual ao UTC).*
* *Japão é +09:00.*

# Classes Thread e Interface Runnable

**Thread 🡪** Linha de execução dentro do programa java. Normalmente, começa com uma Thread principal, mas pode criar várias threads para executarem ao mesmo tempo

- permite fazer tarefa em paralelo

- evita que uma tarefa demorada trave o programa

*Ex: baixa arquivo enquanto o usuário continua usando a interface*

**Runnable** **🡪** É uma interface funcional que define o que a Thread vai executar. Tem apenas um método run()

- Serve para **separar o código que será executado pela thread** da própria thread

Qual forma utilizar para multi-threads?

**Se precisa armazenar uma lista/coleção →** LinkedBlockingQueue **(melhor desempenho e thread-safe pronto).**

**Se precisa contar ou somar valores →** AtomicInteger **(super rápido e seguro).**

**Se precisa proteger código genérico →** synchronized **com ArrayList(mais universal, mas pode ser mais lento)**

Quando você **não escreve nenhum construtor** na classe, o compilador **gera automaticamente um construtor vazio (sem parâmetros)**.