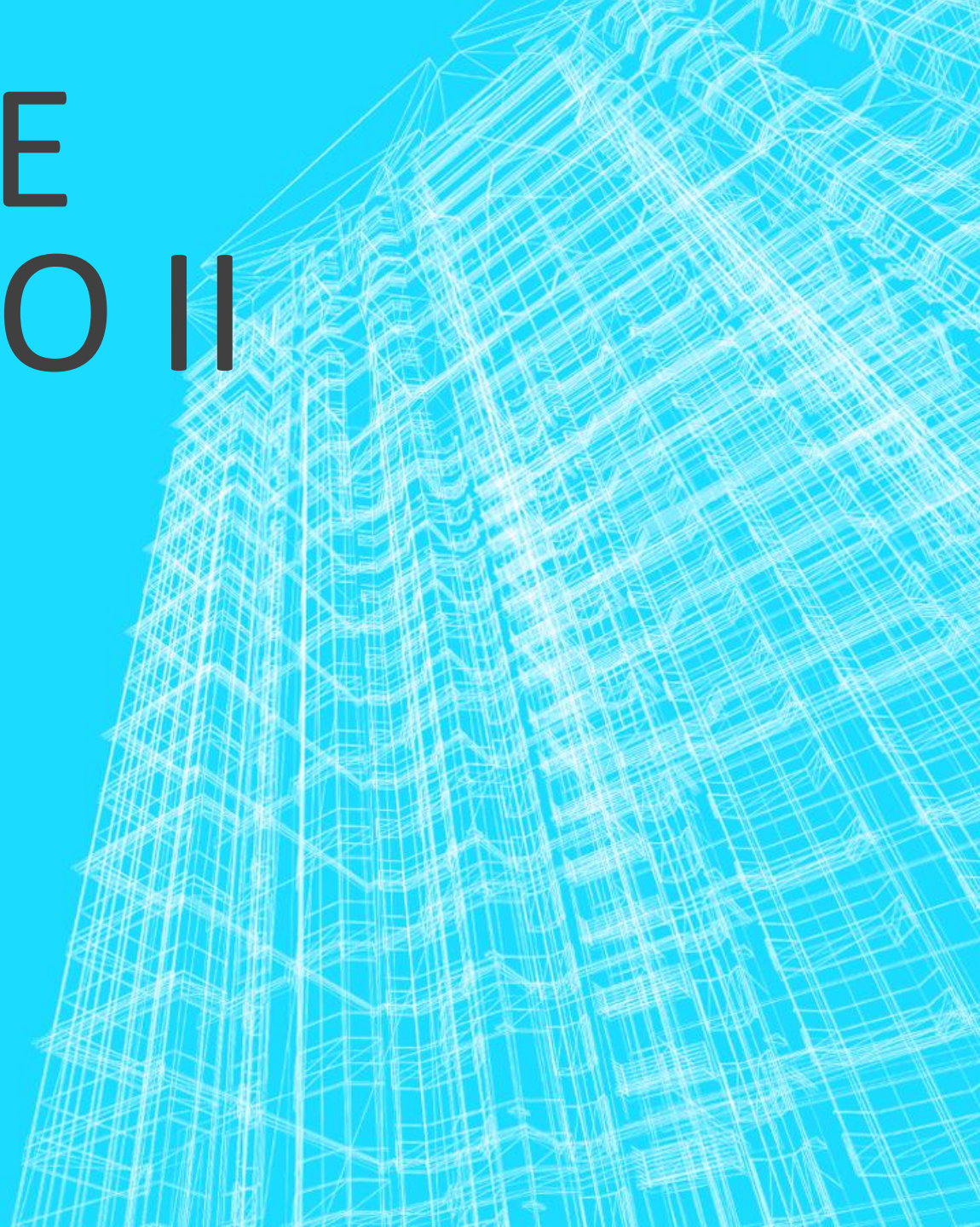
# LINGUAGEM DE PROGRAMAÇÃO II
# AULA 13



Prof. Dr. Alan de Oliveira Santana
alandeoliveirasantana@gmail.com

# OBJETIVOS

- Revisar o conceito de Generics em Java e o uso dos identificadores de tipo (T, E, K, V, etc.).
- Compreender a importância da tipagem genérica para segurança e reuso de código.
- Explorar as interfaces e classes da hierarquia de coleções (List, ArrayList, LinkedList).
- Entender como ArrayList e LinkedList funcionam internamente:
  - alocação, encadeamento, e custo de operações.

# REVISÃO: O QUE SÃO GENERICS

- Antes dos Generics, coleções como ArrayList armazenavam objetos do tipo Object, exigindo casts constantes e abrindo espaço para erros em tempo de execução.

# REVISÃO: O QUE SÃO GENERICS

- Com generics, o tipo é parametrizado:

```
ArrayList<String> nomes = new ArrayList<>();
nomes.add("Alan");
nomes.add("Maria");
// nomes.add(10); // Erro de compilação
```

- Agora o compilador sabe que a lista aceita apenas String.
- Isso traz segurança de tipo (type safety) e reuso.

# PARÂMETROS DE TIPO GENÉRICO

- Os nomes (T, E, K, V, N) são convenções, não obrigatórios.
- O que importa é a posição e o significado:

| Símbolo | Significado comum |
|---|---|
| T | Type (tipo genérico) |
| E | Element (em coleções) |
| K | Key (chave em mapas) |
| V | Value (valor em mapas) |
| N | Number (para classes numéricas) |

# PARÂMETROS DE TIPO GENÉRICO

Exemplo:

```
public class Caixa<T> {
    private T valor;
    public void set(T v) { valor = v; }
    public T get() { return valor; }
}
```

Uso:

```
Caixa<Integer> caixaInt = new Caixa<>();
caixaInt.set(100);
System.out.println(caixaInt.get()); // 100
```

O compilador substitui T por Integer durante a compilação.

# BOUNDED TYPE PARAMETERS

- Às vezes, queremos restringir o tipo genérico:

```
public <T extends Number> double soma(T a, T b) {
   return a.doubleValue() + b.doubleValue();
}
```

- Aqui T precisa herdar de Number, logo soma(5, 2.3) funciona, mas soma("a", "b") não.

# HIERARQUIA DE COLEÇÕES EM JAVA

- A Collections Framework é organizada em interfaces e classes concretas:

```
Collection
├── List
│   ├── ArrayList
│   └── LinkedList
├── Set
│   ├── HashSet
│   └── TreeSet
└── Queue
    ├── PriorityQueue
    └── LinkedList
```

- Para a aula de hoje, focaremos em List, ArrayList e LinkedList.

# INTERFACE LIST

- Uma List é uma coleção ordenada, com acesso por índice e elementos duplicados permitidos.
- Principais métodos:
  - add(E e)
  - get(int index)
  - remove(Object o)
  - size()
  - contains(Object o)

# ARRAYLIST – IMPLEMENTAÇÃO BASEADA EM VETOR

- ArrayList é implementada com um array dinâmico (internamente Object[] elementData).
- Características:
  - Acesso direto por índice (O(1))
  - Inserção/remoção no meio: custo O(n)
  - Cresce automaticamente (usa dobramento da capacidade)

# ARRAYLIST – IMPLEMENTAÇÃO BASEADA EM VETOR

- Exemplo:

```
List<Integer> numeros = new ArrayList<>();
numeros.add(10);
numeros.add(20);
numeros.add(30);
System.out.println(numeros.get(1)); // 20
```

# ARRAYLIST – IMPLEMENTAÇÃO BASEADA EM VETOR

- Estrutura interna simplificada:

```
public class ArrayList<E> {
   private Object[] elementData;
   private int size;

   public void add(E e) {
      ensureCapacity(size + 1);
      elementData[size++] = e;
   }
}
```

- O método ensureCapacity cria um novo array 1,5x maior quando o atual enche.

# LINKEDLIST – IMPLEMENTAÇÃO ENCADEADA

- LinkedList é baseada em nós conectados (Node), cada um com:
  - Dado (E item)
  - Ponteiro para o próximo (next)
  - Ponteiro para o anterior (prev)
- Isso permite inserções e remoções rápidas nas extremidades (O(1)), mas acesso sequencial (O(n)).

# LINKEDLIST – IMPLEMENTAÇÃO ENCADEADA

- Exemplo:

```
List<String> nomes = new LinkedList<>();
nomes.add("Ana");
nomes.add("Beto");
nomes.addFirst("Zara");
System.out.println(nomes); // [Zara, Ana, Beto]
```

# LINKEDLIST – IMPLEMENTAÇÃO ENCADEADA

- Estrutura interna simplificada:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
}
```

# VANTAGENS E DESVANTAGENS

| Estrutura | Vantagens | Desvantagens |
|---|---|---|
| ArrayList | Acesso rápido por índice; ocupa menos memória | Inserção e remoção caras no meio |
| LinkedList | Inserção e remoção rápidas em qualquer ponto | Acesso lento; mais memória (ponteiros) |

# ITERANDO COLEÇÕES GENÉRICAS

- Códigos com e sem iterator

```
List<String> lista = List.of("A", "B", "C");

for (String s : lista)
    System.out.println(s);

lista.forEach(System.out::println);
```

```
Iterator<String> it = lista.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

# O QUE É UM ITERATOR

- Em Java, um Iterator é um objeto que permite percorrer uma coleção de elementos sequencialmente, sem expor a estrutura interna da coleção.

# O QUE É UM ITERATOR

- Ele faz parte do pacote:

```
import java.util.Iterator;
```

- É usado principalmente em classes que implementam a interface Collection, como ArrayList, LinkedList, HashSet, etc.

# INTERFACE ITERATOR

- A interface Iterator<E> possui três métodos principais:

```
public interface Iterator<E> {
    boolean hasNext(); // Há mais elementos?
    E next();          // Retorna o próximo elemento
    void remove();     // Remove o elemento atual (opcional)
}
```

# EXEMPLO BÁSICO

```
import java.util.*;

public class ExemploIterator {
    public static void main(String[] args) {
        List<String> nomes = new ArrayList<>();
        nomes.add("Ana");
        nomes.add("Bruno");
        nomes.add("Carlos");

        Iterator<String> it = nomes.iterator();

        while (it.hasNext()) {
            String nome = it.next();
            System.out.println(nome);
        }
    }
}
```

# COMO FUNCIONA INTERNAMENTE

1. iterator() → cria um cursor inicializado antes do primeiro elemento.
2. hasNext() → verifica se ainda há elementos à frente.
3. next() → move o cursor e retorna o elemento atual.
4. (Opcional) remove() → remove o último elemento retornado por next().

- Durante a iteração, o Iterator mantém um índice interno, controlado pela própria coleção.

# POR QUE USAR ITERATOR?

- Funciona em qualquer tipo de coleção (List, Set, Queue, etc.).
- Evita erros de concorrência como ConcurrentModificationException.
- Permite remover elementos com segurança durante a iteração.

# ERRO COMUM SEM ITERATOR

```
for (String nome : nomes) {
    if (nome.equals("Bruno")) nomes.remove(nome); // ERRO:
ConcurrentModificationException
}
```

- Isso acontece porque o for-each usa um Iterator internamente, mas não permite modificações diretas.

- Forma correta:

```
Iterator<String> it = nomes.iterator();
while (it.hasNext()) {
    if (it.next().equals("Bruno")) {
        it.remove(); // forma segura de remover
    }
}
```

# RELAÇÃO COM O LAÇO “FOR-EACH”

- Todo laço for-each em Java usa um Iterator por baixo dos panos.
- Códigos equivalente:

```
for (String s : lista) {
    System.out.println(s);
}
```

```
Iterator<String> it = lista.iterator();
while (it.hasNext()) {
    String s = it.next();
    System.out.println(s);
}
```

- O for-each é apenas uma abstração sintática que simplifica o uso de Iterator.

# ITERATORS E LINKEDLIST

- No caso de LinkedList, o Iterator é especialmente eficiente, pois ele não precisa acessar índices (o que seria lento).
- Em vez disso, ele segue os ponteiros dos nós encadeados, percorrendo os elementos de forma natural.

```
LinkedList<String> lista = new LinkedList<>();
lista.add("X");
lista.add("Y");
lista.add("Z");

Iterator<String> it = lista.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

# LISTITERATOR — VERSÃO MAIS PODEROSA

- ListIterator é uma subinterface de Iterator, disponível apenas para coleções do tipo List.
- Ela permite navegar em ambas as direções e inserir elementos durante a iteração.

# LISTITERATOR — VERSÃO MAIS PODEROSA

- ListIterator é útil quando precisamos percorrer para frente e para trás, ou inserir durante a iteração sem perder a posição.

```
ListIterator<String> it = lista.listIterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
while (it.hasPrevious()) {
   System.out.println(it.previous());
}
```

# STREAMS EM JAVA

- Um Stream em Java (introduzido no Java 8) é uma abstração de sequência de dados sobre a qual podemos realizar operações de forma declarativa, ou seja, sem precisar escrever loops manuais.
- Ele representa um fluxo de elementos (de uma coleção, array, arquivo, etc.) que passam por uma cadeia de operações como filtros, mapeamentos e reduções.

# EXEMPLO BÁSICO

- Sem stream (modo imperativo):

```
List<String> nomes = List.of("Ana", "Bia", "Carlos",
"Bruno");

for (String nome : nomes) {
    if (nome.startsWith("B")) {
        System.out.println(nome.toUpperCase());
    }
}
```

```
nomes.stream()
    .filter(n -> n.startsWith("B"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

# EXEMPLO BÁSICO

- O for precisa gerenciar iteração e condição.
- O Stream descreve o que fazer, não como fazer.

# PRINCIPAIS CARACTERÍSTICAS

| Característica | Descrição |
|---|---|
| **Não armazena dados** | Apenas processa os dados da fonte |
| **Imutável** | Cada operação cria um novo Stream |
| **Lazy** | Só executa quando há operação terminal (ex: forEach) |
| **Encadeável** | As operações podem ser compostas (pipeline) |
| **Paralelizável** | Pode usar vários núcleos com parallelStream() |

# TIPOS DE OPERAÇÕES

- Intermediárias (criam novo Stream)
  - filter(Predicate)
  - map(Function)
  - sorted()
  - distinct()
  - limit(n)
  - skip(n)
- Terminais (consomem o Stream)
  - forEach()
  - collect()
  - reduce()
  - count()
  - findFirst()
  - anyMatch()

# EXEMPLO PRÁTICO COMPLETO

```
import java.util.*;
import java.util.stream.*;

public class ExemploStream {
    public static void main(String[] args) {
        List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        int somaPares = numeros.stream()
            .filter(n -> n % 2 == 0)
            .mapToInt(Integer::intValue)
            .sum();

        System.out.println("Soma dos pares: " + somaPares);
    }
}
```

# EXEMPLO PRÁTICO COMPLETO

- Explicando o fluxo:
  - stream() cria o fluxo de elementos da lista.
  - filter(n -> n % 2 == 0) mantém apenas números pares.
  - mapToInt(Integer::intValue) converte para IntStream.
  - sum() é a operação terminal que executa o fluxo e devolve o resultado.

# STREAMS PARALELOS

- Se quisermos aproveitar múltiplos núcleos de CPU:

```
int soma = numeros.parallelStream()
            .filter(n -> n % 2 == 0)
            .mapToInt(Integer::intValue)
            .sum();
```

- Isso divide a lista internamente e processa os elementos em paralelo.
- O Java gerencia a sincronização e a fusão dos resultados.

# STREAMS E GENERICS

- Streams são genéricos:

```
Stream<String> s1 = Stream.of("A", "B", "C");
Stream<Integer> s2 = Stream.of(1, 2, 3);
```

- E podem vir de várias fontes:
  - Collection.stream()
  - Arrays.stream()
  - Files.lines(Path)
  - Stream.of()

# OPERAÇÕES DE REDUÇÃO (REDUCE)

- A função reduce() combina todos os elementos do stream em um único resultado.

```
int soma = Stream.of(1, 2, 3, 4)
    .reduce(0, (a, b) -> a + b);
System.out.println(soma); // 10
```

- Ou, de forma mais funcional:

```
Optional<Integer> max = Stream.of(5, 9, 3, 7).reduce(Integer::max);
max.ifPresent(System.out::println); // 9
```

# COLETA DE RESULTADOS

- O método collect() permite converter um Stream em outra estrutura (lista, conjunto, mapa).

```
List<String> maiores = List.of("Ana", "Bruno", "Carlos")
    .stream()
    .filter(n -> n.length() > 3)
    .collect(Collectors.toList());
```

# COMPARAÇÃO: ITERATOR VS STREAM

| Aspecto | **Iterator** | **Stream** |
|---|---|---|
| **Abordagem** | Imperativa (você controla o loop) | Declarativa (você descreve o que fazer) |
| **Introduzido em** | Java 1.2 | Java 8 |
| **Forma de uso** | while (it.hasNext()) { ... } | stream().filter(...).map(...).forEach(...) |
| **Mutável / Imutável** | Permite modificar a coleção (remove()) | Imutável (não altera a coleção original) |
| **Execução** | Passo a passo, síncrona | Lazy e otimizada (pipeline) |
| **Paralelismo** | Manual (precisa usar threads) | Nativo (parallelStream()) |
| **Legibilidade** | Mais verboso | Mais legível e expressivo |
| **Uso típico** | Iteração e remoção de elementos | Processamento, filtragem, agregação |

# ARROW FUNCTIONS (EXPRESSÕES LAMBDA) EM JAVA

- Introduzidas no Java 8, as arrow functions são uma forma mais curta e direta de escrever funções anônimas, ou seja, métodos sem nome que podem ser passados como parâmetros.

# ARROW FUNCTIONS (EXPRESSÕES LAMBDA) EM JAVA

- Introduzidas no Java 8, as arrow functions são uma forma mais curta e direta de escrever funções anônimas, ou seja, métodos sem nome que podem ser passados como parâmetros.

```
(parâmetros) -> expressão
```

```
(parâmetros) -> { bloco de código }
```

# EXEMPLO BÁSICO

- Antes do Java 8

```
List<String> nomes = List.of("Ana", "Bruno", "Carlos");

nomes.forEach(new Consumer<String>() {
    public void accept(String nome) {
        System.out.println(nome);
    }
});
```

- Depois do java 8

```
nomes.forEach(nome -> System.out.println(nome));
```

ou

```
nomes.forEach(System.out::println);
```

# SINTAXE DETALHADA

| Forma | Exemplo | Descrição |
|---|---|---|
| **1 parâmetro, 1 linha** | x -> x * 2 | Não precisa de parênteses nem chaves. |
| **Múltiplos parâmetros** | (a, b) -> a + b | Parênteses obrigatórios. |
| **Várias instruções** | (a, b) -> { int s = a + b; return s; } | Bloco com {} e return. |
| **Sem parâmetros** | () -> System.out.println("Olá!") | Usa parênteses vazios. |

# OBRIGADO!