Aula sobre Collections Java

Pra que serve (objetivo)

- Guardar e manipular **múltiplos elementos** de forma eficiente.
- Ter **operações padronizadas** (adicionar, remover, buscar, iterar, ordenar, filtrar).
- Permitir **trocar implementação** sem reescrever regras (ex.: `List` pode ser `ArrayList` hoje e `LinkedList` amanhã).

Onde aparece no mercado (uso real)

- APIs REST retornando listas de DTOs
- Regras de negócio com filtros, ordenação, agrupamento, deduplicação
- Cache em memória (`Map`)
- Controle de filas (processamento assíncrono, jobs)
- Contagem/estatística (frequência, ranking)

Performance (o que cai MUITO em projeto real)

List

- **ArrayList**
  - `get(i)`: **O(1)**
  - `add(fim)`: **amortizado O(1)**
  - `add(início/meio)` e `remove(i)`: **O(n)** (shift)
  - Boa para: leitura e iteração, append no final
- **LinkedList**
  - `get(i)`: **O(n)** (ruim pra acesso por índice)

- Boa para: inserir/remover nas pontas (mas `ArrayDeque` costuma ser melhor)

Set

- **HashSet**
  - `add/contains/remove`: **O(1)** médio
  - Sem ordem
  - Depende de `equals/hashCode` bem feitos
- **TreeSet**
  - `add/contains/remove`: **O(log n)**
  - Ordenado (natural ou comparator)

Map

- **HashMap**
  - `get/put/containsKey`: **O(1)** médio
  - Sem ordem
- **LinkedHashMap**
  - Mantém ordem de inserção (ou LRU com `accessOrder=true`)
- **TreeMap**
  - Ordenado por chave: **O(log n)**

Dica rápida de escolha (bem prática)

- Precisa **ordem** e **índice**? → `ArrayList`
- Precisa **não repetir**? → `HashSet`
- Precisa **chave → valor**? → `HashMap`
- Precisa **ordenado** automaticamente? → `TreeSet` / `TreeMap`

- Precisa **fila/pilha/deque**? → `ArrayDeque`

Métodos mais usados (no dia a dia)

Em `Collection` / `List` / `Set`

- `add`, `addAll`
- `remove`, `removeAll`, `removeIf`
- `contains`, `containsAll`
- `isEmpty`, `size`, `clear`
- `iterator` / `for-each`
- `toArray`

Em `List`

- `get`, `set`, `indexOf`, `subList`
- `sort` (ou `Collections.sort`)

Em `Map`

- `put`, `get`, `getOrDefault`
- `containsKey`, `remove`
- `keySet`, `values`, `entrySet`
- `putIfAbsent`
- `computeIfAbsent`, `merge`

Utilitários `Collections`

- `Collections.sort`
- `Collections.unmodifiableList` (e variações)
- `Collections.frequency`
- `Collections.max/min`

- `Collections.reverse`, `shuffle`

10 exemplos práticos (cada um com código + Main + explicação)

Exemplo 01 — `List`: filtrar e remover com `removeIf` (limpeza de dados)

Caso real: remover pedidos cancelados / inválidos.

```
import java.util.ArrayList;
import java.util.List;

class Pedido {
    private final String id;
    private final String status; // "OK", "CANCELADO"

    public Pedido(String id, String status) {
        this.id = id;
        this.status = status;
    }

    public String getId() { return id; }
    public String getStatus() { return status; }

    @Override
    public String toString() {
        return "Pedido{id='" + id + "', status='" + status + "'}";
    }
}

public class Main {
    public static void main(String[] args) {
        List<Pedido> pedidos = new ArrayList<>();
        pedidos.add(new Pedido("P1", "OK"));
        pedidos.add(new Pedido("P2", "CANCELADO"));
        pedidos.add(new Pedido("P3", "OK"));
        pedidos.add(new Pedido("P4", "CANCELADO"));

        pedidos.removeIf(p -> "CANCELADO".equals(p.getStatus()));

        System.out.println("Pedidos válidos:");
        pedidos.forEach(System.out::println);
    }
}
```

Explicação detalhada

- `removeIf(Predicate)` percorre a coleção e remove os itens que retornarem `true`.
- É muito usado para “sanear” dados antes de processar.

- Performance: em `ArrayList`, remoções podem custar **O(n)** por causa do shift, mas para limpeza em lote é bem comum e legível.

Exemplo 02 — `Set`: deduplicar e validar repetição (`HashSet`)

Caso real: impedir e-mails duplicados em uma campanha.

```
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> emails = new HashSet<>();

        System.out.println(emails.add("a@teste.com")); // true
(entrou)
        System.out.println(emails.add("b@teste.com")); // true
        System.out.println(emails.add("a@teste.com")); // false
(duplicado)

        System.out.println("Total únicos: " + emails.size());
        System.out.println(emails);
    }
}
```

Explicação detalhada

- `HashSet` mantém **itens únicos**.
- `add()` retorna `false` se o item já existia → ótimo pra validação.
- `contains()` é **O(1)** médio: excelente para checagens frequentes.

Exemplo 03 — `Map`: contagem de frequência com `getOrDefault`

Caso real: contar acessos por endpoint, status HTTP, tags, etc.

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        String[] endpoints = {"/login", "/home", "/login",
"/produtos", "/login", "/home"};

        Map<String, Integer> contagem = new HashMap<>();

        for (String ep : endpoints) {
            int atual = contagem.getOrDefault(ep, 0);
            contagem.put(ep, atual + 1);
        }

        System.out.println("Frequência:");
        contagem.forEach((k, v) -> System.out.println(k + " => " +
v));
    }
}
```

Explicação detalhada

- `getOrDefault(chave, default)` evita `null` e simplifica contagens.
- Padrão clássico para métricas e relatórios.
- `HashMap` dá ótima performance em grandes volumes.

Exemplo 04 — `Map`: agrupar itens com `computeIfAbsent`

Caso real: agrupar produtos por categoria / usuários por perfil.

```
import java.util.*;

class Produto {
    private final String nome;
    private final String categoria;

    public Produto(String nome, String categoria) {
        this.nome = nome;
        this.categoria = categoria;
    }

    public String getNome() { return nome; }
    public String getCategoria() { return categoria; }
}

public class Main {
    public static void main(String[] args) {
        List<Produto> produtos = List.of(
                new Produto("Arroz", "Alimentos"),
                new Produto("Feijão", "Alimentos"),
                new Produto("Detergente", "Limpeza"),
                new Produto("Sabão", "Limpeza")
        );

        Map<String, List<Produto>> porCategoria = new HashMap<>();

        for (Produto p : produtos) {
            porCategoria
                .computeIfAbsent(p.getCategoria(), k -> new
ArrayList<>())
                .add(p);
        }

        porCategoria.forEach((cat, lista) -> {
            System.out.println(cat + ": " +
lista.stream().map(Produto::getNome).toList());
        });
    }
}
```

Explicação detalhada

- `computeIfAbsent` cria a lista automaticamente quando a chave ainda não existe.

- Evita `if (map.get(...) == null)` e deixa o código robusto.
- Muito usado em relatórios, dashboards e regras de agrupamento.

Exemplo 05 — `List`: ordenar com `sort(Comparator)`

Caso real: ordenar usuários por nome, pedidos por data, etc.

```
import java.util.*;

class Usuario {
    private final String nome;
    private final int pontos;

    public Usuario(String nome, int pontos) {
        this.nome = nome;
        this.pontos = pontos;
    }

    public String getNome() { return nome; }
    public int getPontos() { return pontos; }

    @Override
    public String toString() {
        return nome + " (" + pontos + ")";
    }
}

public class Main {
    public static void main(String[] args) {
        List<Usuario> usuarios = new ArrayList<>();
        usuarios.add(new Usuario("Carlos", 120));
        usuarios.add(new Usuario("Ana", 200));
        usuarios.add(new Usuario("Bruno", 150));

        usuarios.sort(Comparator.comparing(Usuario::getNome));
        System.out.println("Ordenado por nome: " + usuarios);


usuarios.sort(Comparator.comparingInt(Usuario::getPontos).reversed()
);
        System.out.println("Ranking por pontos: " + usuarios);
    }
}
```

Explicação detalhada

- `list.sort(comparator)` é o padrão moderno.
- `Comparator.comparing` e `reversed` são muito comuns.
- Ordenação custa **O(n log n)**.

Exemplo 06 — `Map`: `putIfAbsent` (não sobrescrever configuração)

Caso real: configurar defaults sem apagar o que já veio do usuário.

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, String> config = new HashMap<>();
        config.put("timezone", "America/Sao_Paulo");

        config.putIfAbsent("timezone", "UTC"); // não sobrescreve
        config.putIfAbsent("lang", "pt-BR");   // adiciona

        System.out.println(config);
    }
}
```

Explicação detalhada

- `putIfAbsent` é perfeito pra “defaults”.
- Evita sobrescrever configurações já definidas.
- Muito usado em bootstrap de sistemas, leitura de `.properties`, etc.

Exemplo 07 — `Collections.unmodifiableList` (retornar lista imutável)

Caso real: evitar que alguém altere uma lista interna de um serviço.

```
import java.util.*;

class CatalogoService {
    private final List<String> categorias = new ArrayList<>
(List.of("Alimentos", "Limpeza", "Bebidas"));

    public List<String> listarCategorias() {
        return Collections.unmodifiableList(categorias);
    }
}

public class Main {
    public static void main(String[] args) {
        CatalogoService service = new CatalogoService();
        List<String> categorias = service.listarCategorias();

        System.out.println(categorias);

        try {
            categorias.add("Eletrônicos"); // explode
        } catch (UnsupportedOperationException e) {
            System.out.println("Não pode alterar lista retornada: "
+ e);
        }
    }
}
```

Explicação detalhada

- Você protege o estado interno do objeto.
- Evita bugs onde outro dev altera a lista e “quebra” a regra do sistema.
- Alternativa moderna: `List.copyOf(...)` ou `List.of(...)` quando possível.

Exemplo 08 — `Queue/Deque`: fila de processamento com `ArrayDeque`

Caso real: processar tarefas em ordem (jobs).

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<String> fila = new ArrayDeque<>();

        fila.addLast("job-1");
        fila.addLast("job-2");
        fila.addLast("job-3");

        while (!fila.isEmpty()) {
            String job = fila.removeFirst();
            System.out.println("Processando: " + job);
        }
    }
}
```

Explicação detalhada

- `ArrayDeque` é muito eficiente para fila/pilha.
- `addLast/removeFirst` modela FIFO.
- Em projetos: processamento, rate-limit, pipeline simples.

Exemplo 09 — `Map.merge`: somar valores de forma elegante

Caso real: somar totais por cliente, por produto, por dia.

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Double> totalPorCliente = new HashMap<>();

        // cliente, valor
        adicionar(totalPorCliente, "Ana", 50.0);
        adicionar(totalPorCliente, "Ana", 30.0);
        adicionar(totalPorCliente, "Bruno", 20.0);

        System.out.println(totalPorCliente);
    }

    static void adicionar(Map<String, Double> map, String cliente,
double valor) {
        map.merge(cliente, valor, Double::sum);
    }
}
```

Explicação detalhada

- `merge(chave, valorNovo, func)`:
  - se não existe chave → coloca valorNovo
  - se existe → combina com a função (`sum`)
- Muito usado para agregações e relatórios.

Exemplo 10 — `LinkedHashMap` para “Top-N” com ordem e exibição amigável

Caso real: ranking exibido ordenado (depois de ordenar, manter na ordem).

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

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> pontos = new HashMap<>();
        pontos.put("Ana", 200);
        pontos.put("Bruno", 150);
        pontos.put("Carlos", 120);
        pontos.put("Duda", 180);

        // Ordena por valor desc e guarda em LinkedHashMap (mantém
ordem de inserção)
        Map<String, Integer> ranking = pontos.entrySet().stream()
                .sorted(Map.Entry.<String,
Integer>comparingByValue().reversed())
                .limit(3)
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        Map.Entry::getValue,
                        (a, b) -> a,
                        LinkedHashMap::new
                ));

        System.out.println("Top 3:");
        ranking.forEach((nome, pts) -> System.out.println(nome + "
=> " + pts));
    }
}
```

Explicação detalhada

- `HashMap` não garante ordem; `LinkedHashMap` preserva a ordem de inserção.
- Esse padrão é comum pra ranking/top-N em dashboards.
- Usa Stream aqui porque é o jeito mais direto de ordenar `Map` por valor.

Checklist final (pra você fechar a aula com ouro)

- **Sempre programe para a interface**: `List`, `Set`, `Map`.
- **Use `HashSet/HashMap`** para performance de busca.
- **Imutabilidade** quando for retorno de API interna (`unmodifiableList`, `copyOf`).

- **Nunca ignore** `equals/hashCode` quando usar `HashSet/HashMap` com objetos.
- **Evite** `contains()` **em List gigante** (vira O(n) e pode virar gargalo).