4.1 Visão Geral das Coleções em Java

No dia a dia, estamos cercados por conjuntos de elementos: uma sala de aula com alunos, um estacionamento com veículos, uma empresa com funcionários. Em Java, esses conjuntos são representados por **coleções**, estruturas de dados que facilitam o armazenamento e manipulação de grupos de objetos.

4.1.1 Características das Coleções

Cada tipo de coleção possui regras específicas. Por exemplo:

- Em uma **biblioteca**, livros repetidos são permitidos.
- Em uma **sala de aula**, não podem haver alunos repetidos.

Essas diferenças refletem o comportamento das coleções em Java, organizadas a partir da interface `Collection`, que é estendida por três interfaces principais:

- `List` – permite elementos repetidos e acesso por posição (índice).
- `Set` – não permite elementos repetidos.
- `Queue` – usada para estruturas do tipo fila, com ordenação baseada em prioridade ou ordem de chegada.

Além delas, existe a interface `Map`, que não estende `Collection`, mas é igualmente importante, pois trabalha com **pares chave-valor**.

Conceitos importantes

- **Índice**: posição de um elemento em coleções que suportam ordenação (ex: `List`).
- **Chave**: identificador único usado em mapas (`Map`) para acessar valores.
- **Ordenação**: ordem dos elementos conforme foram inseridos ou com base na prioridade.
- **Classificação**: reordenação dos elementos com base em critérios definidos (ex: ordem alfabética).

Principais Implementações

Classe	Interface	Ordenação	Classificação
`HashMap`	Map	Não	Não
`Hashtable`	Map	Não	Não
`TreeMap`	Map	Sim, por ordem natural ou customizada	Sim
`LinkedHashMap`	Map	Sim, por ordem de inserção ou acesso	Não
`HashSet`	Set	Não	Não
`TreeSet`	Set	Sim, por ordem natural ou customizada	Sim
`LinkedHashSet`	Set	Sim, por ordem de inserção	Não
`ArrayList`	List	Sim, por índice	Não
`Vector`	List	Sim, por índice	Não
`LinkedList`	List	Sim, por índice	Não
`PriorityQueue`	Queue	Sim, por prioridade	Sim

Resumo das Interfaces

- **`List`**:
    - Permite duplicatas
    - Acesso por índice
    - Ordem mantida
- **`Set`**:
    - Não permite duplicatas
    - Ordem nem sempre garantida

- **`Map`**:
    - Armazena pares chave-valor
    - Cada chave deve ser única

Aqui está a versão otimizada da sua explicação sobre **características das coleções em Java**, com foco em `List` e nas classes `ArrayList`, `Vector` e `LinkedList`:

4.1.1 Características das Coleções

Interface `Collection` e classe utilitária `Collections`

- A interface `Collection` é a raiz da hierarquia de coleções em Java.
- A classe `Collections` fornece métodos utilitários para manipular coleções, como ordenação e busca.

Interface `List`

A interface `List` representa coleções ordenadas que aceitam elementos duplicados. Suas principais implementações são:

- **`ArrayList`**:
    - Usa array dinâmico internamente.
    - Mais rápida para buscas, mais lenta para inserções/remoções no meio da lista.
    - **Não sincronizada** (não segura para múltiplas threads, mas mais eficiente).
    - Recomendada atualmente em vez de `Vector`.
- **`Vector`**:
    - Semelhante ao `ArrayList`, mas com métodos **sincronizados** (thread-safe).
    - Obsoleta em novos projetos — existia antes da introdução de `ArrayList`.
- **`LinkedList`**:
    - Baseada em lista duplamente ligada.

- Mais rápida para inserções e remoções em qualquer ponto da lista.
- Mais lenta para acesso direto (busca por índice).

Comparativo entre as implementações:

Implementação	Inserção/Remoção	Busca
`LinkedList`	Rápida	Lenta
`ArrayList`/`Vector`	Lenta	Rápida

Uso de `List` na prática

Exemplo sem Generics – Código 18

```
List lista = new ArrayList();
lista.add("João");
lista.add(123);
System.out.println(lista.get(0)); // João
System.out.println(lista.get(1)); // 123
```

- Sem generics, `add()` aceita qualquer tipo de objeto.
- `get()` retorna `Object`, exigindo **typecast** se quisermos um tipo específico.

Exemplo com Typecast – Código 19

```
List lista = new ArrayList();
lista.add("João");
String nome = (String) lista.get(0); // Typecast necessário
```

Se colocarmos um inteiro na lista e tentarmos forçar para `String`, causamos um erro:

Código 20 – Erro `ClassCastException`

```
lista.add(10); // Integer
String nome = (String) lista.get(1); // ERRO: não é String
```

Solução pré-Java 5 – Verificação com `instanceof` – Código 21

```
Object obj = lista.get(1);
if (obj instanceof String) {
```

```
    String nome = (String) obj;
}
```

Solução moderna – Uso de Generics – Código 22

```
List<String> lista = new ArrayList<>();
lista.add("Maria");
// lista.add(123); // ERRO em tempo de compilação
String nome = lista.get(0); // Sem necessidade de cast
```

Com **Generics**, o compilador garante que apenas elementos do tipo indicado sejam adicionados à lista, eliminando erros em tempo de execução.

Conclusão

- Use **ArrayList** para buscas rápidas e inserções esporádicas.
- Use **LinkedList** se precisar inserir/remover elementos com frequência em posições variadas.
- Sempre prefira **Generics** para garantir segurança de tipos em tempo de compilação.

Excelente! Você fez um resumo bem amplo das **coleções em Java**, passando por `List`, `Set` e `Map`, além de tópicos importantes como **remoção, ordenação, generics** e uso da **classe `Collections`**.

✅ Interface Collection vs Classe Collections

- `Collection` é uma **interface raiz** para estruturas como `List`, `Set` e `Queue`.
- `Collections` (com "s") é uma **classe utilitária** com métodos estáticos para operações como `sort()`, `reverse()`, `shuffle()`, etc.

📋 Interface List

Implementações:

- `ArrayList`: mais rápida para **leitura (get)**, **não sincronizada**.
- `Vector`: parecida com `ArrayList`, mas **sincronizada** (mais lenta).
- `LinkedList`: melhor para **inserção/remoção no meio da lista**, mas leitura mais lenta.

```
List<String> lista = new ArrayList<>();
lista.add("Olá");
System.out.println(lista.get(0)); // "Olá"
```

🔄 Remoção de elementos

```
lista.remove(2);       // Remove pelo índice
lista.remove("João");  // Remove pelo objeto
```

🔠 Ordenação com Collections.sort()

Para tipos como `String`, `Integer`, etc., a **ordem natural** é usada.

```
Collections.sort(lista);
```

👤 Classe Pessoa com toString()

```
class Pessoa {
    private int codigo;
    private String nome;
```

```
    @Override
    public String toString() {
        return "[" + codigo + ", " + nome + "]";
    }
}
```

Assim, quando você imprime a lista de `Pessoa`, o `System.out.println()` chama `toString()` automaticamente.

🚫 Interface Set

- Não permite **elementos duplicados**.
- Implementações:
    - `HashSet`: não mantém ordem
    - `TreeSet`: ordena os elementos pela ordem natural ou por `Comparator`
    - `LinkedHashSet`: mantém ordem de inserção

```
Set<String> nomes = new HashSet<>();
nomes.add("Ana");
nomes.add("Ana"); // não será adicionado novamente
```

🧮 Interface Map

- Estrutura de **pares chave-valor**.
- Implementações:
    - `HashMap`: sem ordem
    - `TreeMap`: ordenado pelas chaves
    - `LinkedHashMap`: ordem de inserção
- Chaves **não se repetem**; valores podem.

```
Map<Integer, String> alunos = new HashMap<>();
alunos.put(123, "João");
alunos.put(456, "Maria");

System.out.println(alunos.get(123)); // "João"
```

Iterando com `Map.Entry`:

```
for (Map.Entry<Integer, String> aluno : alunos.entrySet()) {
    System.out.println(aluno.getKey() + ": " + aluno.getValue());
}
```

🔎 Busca no Map

```
if (alunos.containsKey(123)) {
    System.out.println(alunos.get(123));
}
```
