**DIO – Collections e Streams com Java**

**Java.util.List**

* Garante a ordem de inserção;
* Mesma funcionalidade que os arrays possuem nativamente, mas de forma **dinâmica**
* Permite adição, atualização, leitura e remoção sem regras adicionais
* Permite mudar a **ordenação através de Comparators**

**Java.util.Queue**

* Garante a ordem de inserção;
* Permite adição, leitura e remoção considerando a regra básica de uma **fila**:   
  Primeiro que entra, primeiro que sai  
  **FIFO** - First in, first out
* Não permite mudança de ordenação
* **Ex.:** java.util.LinkedList

**Java.util.Set**

* Por padrão, **não garante ordenação**;

Foi criada para ser **performática**, para trabalhar com grandes conjuntos de dados, por isso não possui ordenação (manter a ordem é um processo muito custoso para a memória);

Existem implementações alternativas de java.util.Set que permitem trabalhar com **ordenação**, mas acabam **perdendo** a característica de performance (**LinkedHashSet**);

* Não permite **itens repetidos**;

**Impacta na performance da adição**, pois antes de adicionar, há uma **verificação** se o elemento a ser adicionado já existe no Set;

A falta de itens repetidos também **garante a performance da busca**, pois a busca é sempre **indexada**, já que não há valores repetidos;

* A **leitura é performática**, mas a **adição nem tanto**;
* Permite adição e remoção normalmente. Não possui busca por item específico (pois não é indexada) e atualização. Para leitura, apenas navegação;
* Não permite mudança de ordenação;
* **Ex.:** java.util.HashSet, java.util.TreeSet, java.util.LinkedHashSet

Table

Description automatically generated

**TreeSet**

* Toda vez em que é modificada a sua estrutura, o algoritmo de ordenação de árvore é aplicado novamente  
   Causa um problema de performance a mais.
* A **leitura** do TreeSet é bastante performática, mas a **modificação** dispara uma nova ordenação, que pode ser um grande problema em termos de performance, dependendo do tamanho do TreeSet

**Comparação dos tipos de Set**

* **HashSet** -> performático, mas não garante ordem e não permite ordenação
* **LinkedHashSet** -> Um pouco menos performático, pois garante a ordem de inserção dos elementos.  
   Não deve ser usado onde performance é extremamente importante.
* **TreeSet** -> Muito performático para leitura, mas muito demorado para se aplicar ordenação.  
   Permite alteração da ordem através de Comparators

**Java.util.Map**

* Entrada de **chave e valor**;
* Permite **valores repetidos**, mas não permite **repetição de chave**;
* Permite adição, busca por chave ou valor, atualização, remoção e navegação;
* Pode ser **ordenado**;
* Não extende de **java.util.Collections**, mas tem alguns métodos semelhantes;  
   java.util.Map não extende nenhuma outra interface;
* **Ex.:** java.util.HashMap, java.util.TreeMap, java.util.HashTable

**HashMap**

* Mais performática, bastante consistente;
* Mais usado em produção;

**TreeMap**

* Implementações de árvore binária;
* Usado em casos onde se preza pela ordenação, mas não há tanta preocupação com performance;

**HashTable**

* Versão mais antiga do HashMap;
* Mais usada em cenários com sincronização de threads;
* Pouco usada em produção;

**Comparators**

* Algoritmos de ordenação;
* Utilizado primariamente em java.util.List;
* Permite a ordenação de objetos complexos;
* **Exemplos:**   
   java.util.Comparator – interface para definir classe com regra de ordenação;  
   java.util.Comparable – interface para definir regra de ordenação em uma classe de domínio;
* Interface **Comparable** exige a implementação do método ***“compareTo”***
* Interface **Comparator** exige a implementação do método ***“compare”***

**Retornos – métodos compareTo e compare**

* **positivo**: valor atual é maior do que o valor do objeto recebido como parâmetro
* **negativo**: valor atual é menor do que o valor do objeto recebido como parâmetro
* **zero**: os valores são iguais
* **Dica**  
  - Subtrair do valor atual o valor recebido como parâmetro – **ordem natural**- Subtrair do valor recebido o valor atual – **ordem reversa**

**Optional**

* Tratamento para valores que podem ser nulos;
* Possui 2 estados:  
   - Presente;  
   - Vazio;
* Permite que você execute operações em valores que podem ser nulos, sem se preocupar com as famosas **NullPointerExceptions**;

**Funções Comuns de Optional**

* **“of”**: Usado quando se tem certeza de que o valor está presente;
* **“ofNullable”**: Usado quando o valor pode ser nulo;
* **“empty”**: Usado quando se quer construir um Optional “zerado” (sem valor);
* **“get”**: Usado para retornar o valor do Optional;
* **“map”**: Usado para transformar valores;
* **“ifPresent”**: Usado para executar uma *function* caso o valor esteja presente;
* **“ifPresentOrElse”:** Usado para executar uma ou outra *function* caso o valor esteja presente ou não;
* **“orElseThrow”**: Usado para retornar um erro caso o valor não esteja presente;
* **“isPresent”**: Usado para verificar se valor está presente (boolean);

**Extensões para Tipos Primitivos**

* **OptionalInt** – método “of” e “ofNullable” só aceita **int**;
* **OptionalDouble** – método “of” e “ofNullable” só aceita **double**;
* **OptionalLong** – método “of” e “ofNullable” só aceita **long**;

**Stream API**

* Manipulação de coleções com o **paradigma funcional** de forma paralela;
  + - Paralelismo garante melhor **performance** na entrega da manipulação;
* **Imutável** – Não altera a coleção origem, sempre cria uma nova coleção;
* Principais funcionalidades:
  + - - **Mapping** – Retorna uma coleção com mesmo tamanho da origem com os elementos alterados;
    - **Filtering** – Retorna uma coleção igual ou menor que a coleção origem, com os elementos intactos;
    - **ForEach** – Executa uma determinada lógica para cada elemento, retornando nada;
    - **Peek** – Executa uma determinada lógica para cada elemento, retornando a própria coleção;
    - **Counting** – Retorna um inteiro que representa a contagem de elementos;
    - **Grouping** – Retorna uma coleção agrupada de acordo com a regra definida;