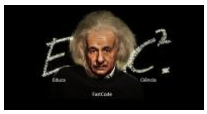


Coleções em Java Focadas no Java 17: Boas Práticas e Desenvolvimento Profissional

As **Collections** em Java são fundamentais para a manipulação eficiente de grupos de objetos. Com o lançamento do **Java 17**, que faz parte da versão de Long Term Support (LTS), diversas melhorias foram introduzidas. É essencial adotar boas práticas ao trabalhar com essas coleções para garantir código limpo, legível e eficiente, características indispensáveis no desenvolvimento moderno, especialmente em engenharia de dados.

Neste artigo, vamos explorar as principais interfaces das **Collections**, abordando implementações recomendadas e trazendo exemplos de boas práticas que podem ser aplicadas no dia a dia do desenvolvedor.

Principais Interfaces de Coleções

A interface **Collection** é a raiz da hierarquia de coleções no Java. Ela define métodos básicos que são herdados e especificados por subinterfaces como **List**, **Set**, **Queue** e **Map**. Cada uma delas possui comportamentos distintos que devem ser usados conforme o caso de uso apropriado.

1. List

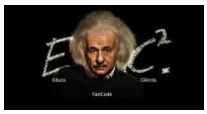
A interface List representa uma coleção ordenada que permite elementos duplicados. A implementação mais comum é o **ArrayList**, amplamente utilizado para listas que necessitam de acesso rápido por índices.

2. Set

A interface Set representa uma coleção que não permite elementos duplicados. Uma das implementações mais populares é o **HashSet**, que oferece operações de inserção e consulta eficientes.

3. Queue

A interface Queue é ideal para coleções que seguem o padrão FIFO (First-In-First-Out). Implementações como **LinkedList** também podem ser usadas para estruturas de filas.



4. Map

Embora não seja uma subinterface de **Collection**, Map é frequentemente utilizada para armazenar pares chave-valor, com destaque para a implementação **HashMap**.

Boas Práticas no Uso de Collections

1. Uso de Coleções Imutáveis

Com o **Java 9**, foram introduzidos métodos de fábrica que permitem a criação de coleções imutáveis de maneira simples e eficiente. No **Java 17**, essas práticas continuam sendo recomendadas para evitar modificações acidentais em coleções que deveriam ser estáticas.

Exemplo:

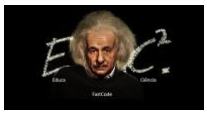
```
import java.util.List;
import java.util.Set;
import java.util.Map;

public class ExemploCollections {

    public static void main(String[] args) {
        List<String> listaImutavel = List.of("A", "B", "C");
        Set<Integer> setImutavel = Set.of(1, 2, 3);
        Map<Integer, String> mapaImutavel = Map.of(1, "Um", 2, "Dois");

        System.out.println(listaImutavel);
        System.out.println(setImutavel);
        System.out.println(mapaImutavel);
    }
}
```

Boas práticas:

- Utilize List.of(), Set.of() e Map.of() para criar coleções imutáveis sempre que possível.
- Coleções imutáveis são preferíveis em cenários onde a integridade dos dados precisa ser garantida, minimizando o risco de alterações indesejadas.

2. Utilização de Genéricos

Genéricos são essenciais para garantir a segurança de tipo em coleções. Isso evita erros comuns como ClassCastException em tempo de execução, proporcionando um código mais robusto.



Exemplo:

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

public class TipagemGenerica {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();
        lista.add("Elemento 1");
        lista.add("Elemento 2");

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

Boas práticas:

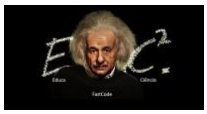
- Sempre declare coleções usando genéricos (List<String>, Set<Integer>) para garantir segurança em tempo de compilação.
- Evite coleções "brutas" (raw types), como List, que não especificam o tipo dos elementos, pois isso pode introduzir erros de tipagem e prejudicar a legibilidade.

3. Escolha Correta de Implementações

Cada tipo de **Collection** possui características únicas em termos de complexidade de tempo para operações como inserção, remoção e acesso. A escolha adequada da implementação pode otimizar significativamente a performance do sistema.

Exemplo:

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

public class PerformanceCollections {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        List<String> linkedList = new LinkedList<>();

        // Uso eficiente de ArrayList para acesso rápido por índice
        arrayList.add("Elemento 1");
        System.out.println(arrayList.get(0)); // O(1) para acesso por índice

        // Uso eficiente de LinkedList para remoções no início da lista
        linkedList.add("Elemento 2");
        linkedList.remove(0); // O(1) para remoção no início
    }
}
```



Boas práticas:

- Use ArrayList para listas onde o acesso por índice é frequente e a inserção/remoção ocorre esporadicamente.
- Utilize LinkedList em cenários onde a inserção e remoção no início/fim da lista são comuns.

4. Uso de Streams para Manipulação de Coleções

As **Streams** introduzidas no Java 8 oferecem uma abordagem funcional e mais expressiva para o processamento de coleções. No **Java 17**, o uso de streams continua sendo uma prática recomendada para manipular coleções de forma concisa e eficiente.

Exemplo:

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

public class StreamExample {
    public static void main(String[] args) {
        List<String> lista = List.of("A", "B", "C", "D");

        List<String> filtrada = lista.stream()
                                .filter(s -> s.startsWith("A"))
                                .collect(Collectors.toList());

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

Boas práticas:

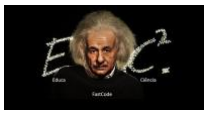
- Prefira **Streams** para manipulação de grandes coleções ou quando há necessidade de realizar várias transformações e filtros.
- Evite usar parallelStream() indiscriminadamente. Certifique-se de que o código seja thread-safe antes de paralelizar operações.

5. Métodos computeIfAbsent e computeIfPresent em Map

A partir do Java 8, os métodos computeIfAbsent e computeIfPresent em mapas se tornaram um padrão eficiente para evitar a necessidade de verificações manuais de existência de chave, promovendo código mais conciso e performático.

Exemplo:

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

public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> mapa = new HashMap<>();
```



```
        // Adiciona se a chave não estiver presente
        mapa.computeIfAbsent("chave1", k -> 42);

        // Atualiza se a chave estiver presente
        mapa.computeIfPresent("chave1", (k, v) -> v + 1);

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

Boas práticas:

- Use computeIfAbsent para evitar verificações manuais de existência de chave e garantir inicializações consistentes.
- Utilize computeIfPresent para modificar valores em mapas de maneira eficiente e segura.

Conclusão

O **Java 17** traz consigo a continuidade de boas práticas que já eram incentivadas em versões anteriores, mas com melhorias significativas em termos de imutabilidade, manipulação eficiente de coleções e segurança de tipo com genéricos.

Ao adotar práticas como o uso de coleções imutáveis, a escolha correta de implementações e o emprego adequado de streams e métodos avançados de Map, desenvolvedores podem garantir que suas aplicações sejam não apenas eficientes, mas também fáceis de manter e seguras.

EducaCiência FastCode para a comunidade.