JDK 23 Exemplos Praticos

Neste artigo, vamos explorar mais profundamente as principais novidades do **Java JDK 23**.

Abordaremos cada recurso com exemplos práticos e explicações detalhadas, permitindo que você compreenda como utilizar essas funcionalidades no desenvolvimento do dia a dia.

1. Primitive Types in Patterns (JEP 455 - Preview)

Com o JDK 23, agora é possível usar tipos primitivos em padrões, tanto em instanceof quanto em switch.

Antes dessa melhoria, o Pattern Matching lidava apenas com tipos de referência, exigindo conversões para Integer, Double, etc.

Agora, os tipos primitivos podem ser usados diretamente, tornando o código mais eficiente e fácil de ler.

Exemplo de switch com tipos primitivos:

java

```
public class PrimitivePatternExample {
    public static void main(String[] args) {
        Object value = 42;

        switch (value) {
            case Integer i -> System.out.println("Integer: " + i);
            case Double d -> System.out.println("Double: " + d);
            default -> System.out.println("Unknown type");
        }
    }
}
```

No exemplo acima, estamos usando um switch que trabalha diretamente com o tipo primitivo Integer.

Quando o valor é um int, ele será correspondido no caso Integer.

Isso torna o código mais claro e eficiente, pois elimina a necessidade de realizar conversões manuais ou usar wrappers desnecessários.

Benefícios:

- **Performance:** O uso de tipos primitivos elimina a necessidade de conversões e boxing.
- **Legibilidade:** O código fica mais direto ao lidar com tipos primitivos.

2. ZGC: Generational Mode by Default (JEP 474)

O **ZGC** (Z Garbage Collector) é um coletor de lixo projetado para lidar com grandes quantidades de memória, proporcionando pausas extremamente curtas, mesmo com heaps enormes.

No JDK 23, ele foi aprimorado com o suporte ao modo geracional, que melhora a coleta de objetos de curta duração, otimizando o uso de memória.

Exemplo de uso com ZGC:

Para utilizar o **ZGC** com o novo modo geracional, basta iniciar a JVM com as seguintes opções:

```
java -XX:+UseZGC -XX:+ZGenerationalGC -jar sua-aplicacao.jar
```

Com essa configuração, o ZGC agora gerencia a memória em duas gerações: **nova** e **antiga**.

Objetos que sobrevivem por mais tempo são promovidos para a geração antiga, enquanto objetos temporários são coletados rapidamente na geração nova. Isso aumenta a eficiência em aplicações que lidam com grandes volumes de dados temporários, como sistemas web de alta demanda.

Benefícios:

- **Melhoria de performance:** Coleta de lixo mais eficiente em aplicações com grande volume de objetos temporários.
- **Baixa latência:** Mantém as pausas do garbage collector extremamente curtas, mesmo em ambientes de produção.

3. **Vector API (JEP 469 - Eighth Incubator)**

A **Vector API** oferece uma maneira eficiente de realizar operações matemáticas em vetores, aproveitando o paralelismo e os recursos de processamento SIMD (Single Instruction Multiple Data) das CPUs modernas.

Esse recurso é útil para otimizar cálculos numéricos e manipulação de grandes volumes de dados.

Exemplo de código com a Vector API:

```
import jdk.incubator.vector.*;

public class VectorExample {
    public static void main(String[] args) {
        // Cria um vetor de inteiros com capacidade para 256 bits
        var species = IntVector.SPECIES_256;

        int[] a = {1, 2, 3, 4, 5, 6, 7, 8};
        int[] b = {1, 1, 1, 1, 1, 1, 1, 1};
        int[] result = new int[8];

        // Carrega os arrays em vetores
        var va = IntVector.fromArray(species, a, 0);
        var vb = IntVector.fromArray(species, b, 0);

        // Soma os vetores
        var vc = va.add(vb);

        // Armazena o resultado de volta em um array
        vc.intoArray(result, 0);

        // Exibe o resultado
        for (int r : result) {
            System.out.print(r + " "); // Saída: 2 3 4 5 6 7 8 9
        }
    }
}
```

Neste exemplo, estamos utilizando a **Vector API** para realizar a soma de dois vetores de inteiros.

O método `IntVector.add()` permite que a operação seja realizada de forma otimizada, aproveitando as instruções SIMD do processador.

Isso pode melhorar significativamente o desempenho em operações que envolvem grandes conjuntos de dados.

Benefícios:

- **Otimização de performance:** Operações matemáticas são executadas em paralelo, aproveitando ao máximo a CPU.

- **Escalabilidade:** Ideal para sistemas que processam grandes volumes de dados, como algoritmos científicos e financeiros.

4. **Scoped Values (JEP 481 - Third Preview)**

Os **Scoped Values** permitem compartilhar dados de forma segura entre diferentes threads em um contexto limitado, sem a necessidade de variáveis globais ou compartilhamento manual de estado.

Eles são especialmente úteis em ambientes concorrentes, onde é importante garantir a imutabilidade de dados.

Exemplo de código com Scoped Values:

```
import jdk.incubator.concurrent.ScopedValue;

public class ScopedValueExample {
    // Define um ScopedValue
    public static final ScopedValue<String> SCOPED_VALUE = ScopedValue.newInstance();

    public static void main(String[] args) {
        // Executa uma operação com um valor específico
        ScopedValue.where(SCOPED_VALUE, "Valor no escopo")
            .run(() -> {
                System.out.println("Scoped Value: " + SCOPED_VALUE.get());
            });
    }
}
```

O ScopedValue permite que você associe um valor a uma tarefa e tenha acesso a ele apenas durante a execução dessa tarefa.

Isso simplifica o compartilhamento de informações entre threads de maneira segura e imutável, evitando o uso de variáveis globais ou compartilhamento manual de estado.

Benefícios:

- **Segurança:** Valores só podem ser acessados dentro do escopo definido.
- **Imutabilidade:** Garante que os dados não serão modificados acidentalmente por outras partes do código.

5. **Structured Concurrency (JEP 480 - Third Preview)**

A **Structured Concurrency** simplifica o gerenciamento de várias threads, agrupando tarefas em um escopo que é controlado de forma estruturada.

Isso ajuda a evitar erros comuns no manuseio de threads, como concorrência acidental e vazamento de recursos.

Exemplo de código com Structured Concurrency:

```
import jdk.incubator.concurrent.StructuredTaskScope;
import java.util.concurrent.ExecutionException;

public class StructuredConcurrencyExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            var future1 = scope.fork(() -> {
                // Tarefa 1
                return "Resultado da tarefa 1";
            });

            var future2 = scope.fork(() -> {
                // Tarefa 2
                return "Resultado da tarefa 2";
            });

            scope.join();  // Aguarda todas as tarefas
            scope.throwIfFailed();  // Lança exceção se alguma tarefa falhar

            System.out.println(future1.resultNow());
            System.out.println(future2.resultNow());
        }
    }
}
```

O StructuredTaskScope gerencia automaticamente a execução e o término das threads criadas, garantindo que todas as tarefas dentro do escopo sejam concluídas antes de prosseguir.

Isso evita problemas de vazamento de threads e torna o código mais seguro e fácil de manter.

Benefícios:

- **Gerenciamento automático de threads:** Reduz a complexidade de controle manual de threads.
- **Prevenção de vazamento de recursos:** Threads são sempre corretamente encerradas ao final do escopo.

Esses são apenas alguns dos recursos inovadores que o JDK 23 oferece. Com cada nova versão, o Java continua se expandindo para atender às necessidades modernas de desenvolvimento, oferecendo ferramentas que aumentam a produtividade, melhoram a performance e reforçam a segurança.

EducaCiência FastCode para a comunidade