Manipulação de Datas e Horas em Java: Comparativo entre as Versões 8, 11 e 17

Introdução

Java tem evoluído significativamente desde a introdução da API de Datas e Horas no Java 8, com melhorias contínuas nas versões 11 e 17. Este artigo explora a otimização de código de manipulação de datas e horas em Java, detalhando as melhores práticas e comparando as implementações para Java 8, 11 e 17.

Objetivos

1. **Otimização do código** para operações com datas e horas.
2. **Comparação de recursos** introduzidos nas versões Java 8, 11 e 17.
3. **Adoção de melhores práticas** para maximizar a eficiência e clareza do código.

Código Otimizado: Versão Base (Java 8)

Vamos começar com uma versão performática e moderna para Java 8 usando a nova API java.time.

```
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DatePerformance {

    // Formatos de data e hora
    private static final DateTimeFormatter dateTimeFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    private static final DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy");
    private static final DateTimeFormatter timeFormatter =
DateTimeFormatter.ofPattern("HH:mm:ss");

    public static void main(String[] args) {
        // Hora atual do sistema
        LocalDateTime now = LocalDateTime.now();

        // Formatando as informações de data/hora
        String horaSistemaCompleta = formatDateTime(now);
```

```
        String horaSistemaData = formatDate(now);

        String horaSistemaHora = formatTime(now);

        // Exibindo resultados
        printResults(horaSistemaCompleta, horaSistemaData, horaSistemaHora);

        // Adicionando 30 minutos
        LocalDateTime newTime = now.plusMinutes(30);
        System.out.println("Data/Hora após 30 minutos = " + formatDateTime(newTime));

        // Comparações de datas
        LocalDate hoje = LocalDate.now();
        LocalDate ontem = hoje.minusDays(1);
        LocalDate amanha = hoje.plusDays(1);

        printComparisons(hoje, ontem, amanha);
    }

    // Métodos para formatar as datas
    private static String formatDateTime(LocalDateTime dateTime) {
        return dateTime.format(dateTimeFormatter);
    }

    private static String formatDate(LocalDateTime dateTime) {
        return dateTime.format(dateFormatter);
    }

    private static String formatTime(LocalDateTime dateTime) {
        return dateTime.format(timeFormatter);
    }

    // Exibe resultados formatados
    private static void printResults(String completa, String data, String hora) {
        System.out.println("Data/Hora do Sistema = " + completa);
        System.out.println("Data do Sistema = " + data);
        System.out.println("Hora do Sistema = " + hora);
    }

    // Exibe comparações de datas
    private static void printComparisons(LocalDate hoje, LocalDate ontem, LocalDate amanha) {
        System.out.println("Hoje: " + hoje);
        System.out.println("Ontem: " + ontem);
        System.out.println("Amanhã: " + amanha);

        if (hoje.isAfter(ontem)) {
            System.out.println("Hoje é maior que ontem");
        } else {
            System.out.println("Hoje é menor ou igual a ontem");
        }

        if (hoje.minusDays(1).equals(ontem)) {
            System.out.println("Ontem está correto.");
        } else {
            System.out.println("Ontem está incorreto.");
        }
    }
}
```

Análise da Implementação no Java 8

- **LocalDateTime e DateTimeFormatter**: São usados para manipulação de datas e horas. O LocalDateTime é imutável, eficiente e thread-safe, substituindo o antigo java.util.Date.
- **Formatação de Data**: Os métodos para formatação foram extraídos em funções utilitárias (formatDateTime, formatDate, formatTime) para garantir a modularidade e reuso.
- **Comparações de Data**: A comparação entre datas é feita com métodos claros como isAfter() e equals().

Essa implementação é eficiente, legível e segue as boas práticas da API java.time.

Código para Java 11

O código Java 11 permanece praticamente o mesmo que o da versão 8, com algumas otimizações adicionais em nível de JVM e melhorias na API de strings. O foco está em aproveitar os novos métodos e as melhorias do garbage collection (GC).

```
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DatePerformanceJava11 {

    // Formatos de data e hora
    private static final DateTimeFormatter dateTimeFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    private static final DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy");
    private static final DateTimeFormatter timeFormatter =
DateTimeFormatter.ofPattern("HH:mm:ss");

    public static void main(String[] args) {
        // Hora atual do sistema
        LocalDateTime now = LocalDateTime.now();

        // Formatando as informações de data/hora
        String horaSistemaCompleta = formatDateTime(now);
        String horaSistemaData = formatDate(now);
        String horaSistemaHora = formatTime(now);

        // Exibindo resultados
        printResults(horaSistemaCompleta, horaSistemaData, horaSistemaHora);

        // Adicionando 30 minutos
        LocalDateTime newTime = now.plusMinutes(30);
        System.out.println("Data/Hora após 30 minutos = " + formatDateTime(newTime));
```

```
        // Comparações de datas
        LocalDate hoje = LocalDate.now();

        LocalDate ontem = hoje.minusDays(1);
        LocalDate amanha = hoje.plusDays(1);

        printComparisons(hoje, ontem, amanha);

        // Exemplo de novo método String em Java 11
        String emptyString = "";
        if (emptyString.isBlank()) {
            System.out.println("String vazia ou em branco");
        }
    }

    // Métodos para formatar as datas
    private static String formatDateTime(LocalDateTime dateTime) {
        return dateTime.format(dateTimeFormatter);
    }

    private static String formatDate(LocalDateTime dateTime) {
        return dateTime.format(dateFormatter);
    }

    private static String formatTime(LocalDateTime dateTime) {
        return dateTime.format(timeFormatter);
    }

    // Exibe resultados formatados
    private static void printResults(String completa, String data, String hora) {
        System.out.println("Data/Hora do Sistema = " + completa);
        System.out.println("Data do Sistema = " + data);
        System.out.println("Hora do Sistema = " + hora);
    }

    // Exibe comparações de datas
    private static void printComparisons(LocalDate hoje, LocalDate ontem, LocalDate amanha) {
        System.out.println("Hoje: " + hoje);
        System.out.println("Ontem: " + ontem);
        System.out.println("Amanhã: " + amanha);

        if (hoje.isAfter(ontem)) {
            System.out.println("Hoje é maior que ontem");
        } else {
            System.out.println("Hoje é menor ou igual a ontem");
        }

        if (hoje.minusDays(1).equals(ontem)) {
            System.out.println("Ontem está correto.");
        } else {
            System.out.println("Ontem está incorreto.");
        }
    }
}
```

Novidades do Java 11

- **Métodos de String**: Como isBlank() e repeat(), utilizados para operações com strings. Esses métodos facilitam a verificação de strings vazias ou repetidas.
- **Garbage Collection**: Java 11 traz melhorias no ZGC (Z Garbage Collector), que é uma ferramenta importante para aplicações que exigem baixa latência.

Código para Java 17

O Java 17 traz várias melhorias na linguagem, como pattern matching e classes seladas. No entanto, no contexto de manipulação de datas e horas, o código permanece similar ao Java 8 e 11, com melhorias automáticas de performance em nível de JVM e GC.

```
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class DatePerformanceJava17 {

    // Formatos de data e hora
    private static final DateTimeFormatter dateTimeFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    private static final DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy");
    private static final DateTimeFormatter timeFormatter =
DateTimeFormatter.ofPattern("HH:mm:ss");

    public static void main(String[] args) {
        // Hora atual do sistema
        LocalDateTime now = LocalDateTime.now();

        // Formatando as informações de data/hora
        String horaSistemaCompleta = formatDateTime(now);
        String horaSistemaData = formatDate(now);
        String horaSistemaHora = formatTime(now);

        // Exibindo resultados
        printResults(horaSistemaCompleta, horaSistemaData, horaSistemaHora);

        // Adicionando 30 minutos
        LocalDateTime newTime = now.plusMinutes(30);
        System.out.println("Data/Hora após 30 minutos = " + formatDateTime(newTime));

        // Comparações de datas
        LocalDate hoje = LocalDate.now();
        LocalDate ontem = hoje.minusDays(1);
        LocalDate amanha = hoje.plusDays(1);

        printComparisons(hoje, ontem, amanha);

        // Exemplo de pattern matching em Java 17
        Object obj = "Exemplo de string";
        if (obj instanceof String s) {
```

```
            System.out.println("É uma string: " + s);
        }
    }

    // Métodos para formatar as datas
    private static String formatDateTime(LocalDateTime dateTime) {
        return dateTime.format(dateTimeFormatter);
    }


    private static String formatDate(LocalDateTime dateTime) {
        return dateTime.format(dateFormatter);
    }

    private static String formatTime(LocalDateTime dateTime) {
        return dateTime.format(timeFormatter);
    }

    // Exibe resultados formatados
    private static void printResults(String completa, String data, String hora) {
        System.out.println("Data/Hora do Sistema = " + completa);
        System.out.println("Data do Sistema = " + data);
        System.out.println("Hora do Sistema = " + hora);
    }

    // Exibe comparações de datas
    private static void printComparisons(LocalDate hoje, LocalDate ontem, LocalDate amanha) {
        System.out.println("Hoje: " + hoje);
        System.out.println("Ontem: " + ontem);
        System.out.println("Amanhã: " + amanha);

        if (hoje.isAfter(ontem)) {
            System.out.println("Hoje é maior que ontem");
        } else {
            System.out.println("Hoje é menor ou igual a ontem");
        }

        if (hoje.minusDays(1).equals(ontem)) {
            System.out.println("Ontem está correto.");
        } else {
            System.out.println("Ontem está incorreto.");
        }
    }
}
```

Novidades do Java 17

- **Pattern Matching**: O Java 17 traz o suporte para pattern matching com instanceof, tornando o código mais limpo e legível.
- **Classes Seladas**: Não aplicável diretamente neste exemplo, mas são uma novidade importante na linguagem que permite definir hierarquias de classes mais seguras.

Conclusão

A evolução da API de manipulação de datas e horas em Java reflete um compromisso contínuo da linguagem com a simplificação e otimização de operações complexas. A transição do java.util.Date e java.util.Calendar para a API java.time no Java 8 introduziu uma abordagem imutável, thread-safe e altamente performática, proporcionando aos desenvolvedores uma forma mais intuitiva e robusta de lidar com datas e horas.

O Java 11 não trouxe mudanças substanciais nessa API, mas otimizou a JVM com melhorias como o ZGC, impactando diretamente a performance em sistemas de baixa latência, beneficiando até mesmo as operações de manipulação de datas em ambientes de alta carga. A introdução de novos métodos na API de strings, como isBlank(), também contribuiu para a clareza e redução de código boilerplate, o que se alinha com as boas práticas de código limpo e eficiente.

Com o Java 17, além das melhorias contínuas de performance, a introdução do pattern matching para instanceof e a consolidação do G1 GC como coleta de lixo padrão elevaram a produtividade e a eficiência de código em situações de alta concorrência e uso intensivo de memória. Embora as funcionalidades de manipulação de datas tenham se mantido consistentes desde o Java 8, as melhorias subjacentes no gerenciamento de memória e o refinamento da JVM proporcionam um ganho de performance significativo para operações intensivas.

Em todas as versões, o uso adequado das classes LocalDate, LocalDateTime e DateTimeFormatter, combinado com métodos como isAfter(), plusMinutes() e minusDays(), assegura uma manipulação eficiente e legível. A modularização do código e a separação de responsabilidades garantem sua escalabilidade, facilitando a manutenção em sistemas de larga escala.

Portanto, ao comparar as três versões (8, 11 e 17), vemos que, do ponto de vista de manipulação de datas, a API se manteve consistente, com os ganhos de performance provenientes de melhorias internas na JVM e nas coletas de lixo. Para sistemas críticos, a adoção do Java 17 é fortemente recomendada devido ao balanceamento entre estabilidade, performance e novos recursos que permitem uma codificação mais expressiva e otimizada.

EducaCiência FastCode para a comunidade