Novos Recursos do Java

Nivelamento de Lógica de Programação e Programação Orientada a Objeto

Novos Recursos do Java

O Novo Java

Como citamos na primeira aula, Java é uma linguagem de programação com quase 30 anos de história, criada em 1995.

Desde lá e até hoje, sempre se manteve entre as top 5 linguagens mais populares entre as preferências das pessoas desenvolvedoras de software e das empresas que buscam contratações.

Apesar de certo abalo sofrido entre meados de 2010 a 2015, quando linguagens como **Python** e **Kotlin** surgiram e ganharam popularidade por serem simples e performáticas, essa fase também marca a transição Java da Sun para a Oracle e, logo após em 2017, a Oracle anuncia a mudança na política de *releases* (lançamentos).

As versões 6, 7 e 8 foram muito criticadas pela demora do seu lançamento, embora a última tenha sido muito comemorada por trazer uma cara mais moderna para o Java, com as expressões lambdas e a Stream API.

Até hoje, a versão 8 é a mais popular, de acordo com pesquisa da *JetBrains*.

Ainda assim, essa versão foi de 2014 e já estamos falando de mais de 10 anos passados! É hora de entender os novos recursos até aqui.

Versões LTS

A estratégia atual de lançamentos se baseia em LTS (Long-Term Support) de 24 meses e versões rapid-release a cada 6 meses.

A última *LTS* é a 21 - embora as versões 8, 11 e 17 ainda estejam vigentes pelo suporte estendido - e a próxima é a 25, esperada para setembro/2025.

Novos recursos do Java

De forma geral, Java tem avançado em direção à modernização, como uma linguagem mais simples e rápida, mas sem abrir mão da segurança e confiabilidade que sempre lhe foram características.

Os novos recursos que mais trouxeram impacto recente são a tipagem em tempo de compilação (`var`), a imutabilidade dos `records`, o controle de herança com as classes `seladas`, os blocos de texto e as novas sintaxes de `switch` e "*pattern matching*".

Tipagem em tempo de compilação

Tipagem em tempo de compilação

Algumas linguagens usam tipagem dinâmica, diferente do Java que é fortemente tipada.

A tipagem dinâmica permite que uma variável não tenha um tipo fixo, podendo armazenar valores de diferentes tipos a qualquer momento.

Se por um lado isso entrega liberdade e fluidez, por outro lado, pode levar a erros por descuido.

Tipagem em tempo de compilação

Java define uma estratégia de tipagem em tempo de compilação:

```java
public class PrimeiroPrograma {
  public static void main(String[] args) {
      var texto = "Olá Mundo!";
      var num = 123;
      num = texto;//erro de compilação, num é do tipo int
  }
}
```

Record

Record

Os records são um tipo que o Java introduziu para simplificar o uso de classes de dados ("*data classes*"), também chamados de **POJO's** imutáveis.

É uma forma compacta, mas que não deve ser confundida com uma nova forma de escrever classes nem também pode ser usada em qualquer situação.

```
record Pessoa(String nome, String sobrenome) {}
```

A sintaxe básica de um **record** é apenas isso.

Record

E é equivalente à seguinte classe usando a sintaxe clássica.

```java
public final class Pessoa {
    private final String nome;
    public Pessoa(String nome) {
        this.nome = nome;
    }
    public String nome() {
        return nome;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (obj == null || obj.getClass() != this.getClass()) {
            return false;
        }
        var that = (Pessoa) obj;
        return Objects.equals(this.nome, that.nome);
    }
    @Override
    public int hashCode() {
        return Objects.hash(nome);
    }
    @Override
    public String toString() {
        return "Pessoa[" +
                "nome=" + nome + ']';
    }
}
```

A princípio pode ser tentador e achar que a forma com **records** seria muito mais interessante.

Porém, lendo mais atentamente, a substituição da classe Pessoa por uma **record** causaria a quebra do sistema de escola.

Isso porque, como podemos perceber na forma "expandida", a classe é final o que implica que não pode ser herdada.

Excluindo cenários que usam herança ou que precisam existir os modificadores **setter**, os **records** podem ajudar muitas situações comuns de desenvolvimento que envolvem imutabilidade.

Outra particularidade são os métodos acessadores **get** que não possuem o prefixo **get**.

Veja que o acesso acontece por **nome()**.

Isso também é algo que precisamos ficar atentos, principalmente, se for mudar um código já existente.

Por fim, todo **record** ainda pode definir construtores e métodos.

Record

```
record Turma(String nome, LocalDate dataInicio, LocalDate dataFim, List<Aluno>
alunosList) {
    public Turma {
        if (!dataInicio.isAfter(LocalDate.now()) || !dataInicio.isBefore(dataFim))
{
            throw new IllegalArgumentException("A data início deve ser posterior a
hoje e a data de fim deve ser posterior ao início.");
        }
    }
    public Turma(String nome, LocalDate dataInicio, LocalDate dataFim) {
        this(nome, dataInicio, dataFim, Collections.emptyList());
    }
}
```

Exercícios

Exercício 1: Sistema de Gerenciamento de Biblioteca Parte 3

Evolução do projeto: Uso de `var` para simplificar o código e aplicação de **records** para criar modelos imutáveis como **Autor** e **Categoria**.

Integração dos records com as interfaces definidas anteriormente, mantendo o baixo acoplamento e destacando a imutabilidade.

Desafio:

Reestruturar as classes existentes para utilizar **records** onde for apropriado, mantendo a interface *Emprestavel* e o baixo acoplamento no projeto.

Obrigada